논 블록킹 검색연산을 위한 R-tree 기반의 동시성 제어 기법

김 명 근[†] · 배 해 영[††]

요 약

본 논문에서는 검색 위주의 공간 데이터베이스 시스템을 위한 R-tree 기반의 동시성 제어 기법을 제안한다. 기존의 제안된 기법들은 검색연산이 갱신연산과 동시에 수행되는 것을 막기 위해 노드에 공유 락이나 래치를 획득함으로써 갱신연산으로 인한 검색연산의 블록킹을 막을 수 없다는 문제를 가지고 있으며, 또한 R-tree같은 다차원 색인의 경우 갱신연산의 락 획득은 여러 노드에 걸쳐 일어날 수 있으며, 노드 분할과 같은 경우 오랜 시간동안 락을 획득하고 있을 수도 있기 때문에 검색연산은 장시간 블록킹이 되어야 하는 문제를 갖는다. 따라서 본 논문에서는 이러한 문제를 해결하기 위하여 노드의 엔트리들을 링크드 리스트로 연결하는 방법을 사용하여 노드에 엔트리를 삽입하고 있는 중에도 락이나 래치를 획득하지 않고 검색연산을 할 수 있는 링크드 리스트 기반의 동시성 제어 기법과, 노드 분할이 진행 중에 있는 노드에도 검색연산이 락이나 래치를 획득하지 않고 노드를 탐색할 수 있는 버전 기반의 동시성 제어 기법을 제안한다.

A Concurrency Control Method for Non-blocking Search Operation based on R-tree

Myung-Keun Kim[†] · Hae-Young Bae[††]

ABSTRACT

In this paper, we propose a concurrency control algorithm based on R-tree for spatial database management system. The previous proposed algorithms can't prevent problem that search operation is to be blocking during update operations. In case of multidimensional indexes like R-tree, locking of update operations may be locked to several nodes, and splitting of nodes have to lock a splitting node for a long time. Therefore search operations have to waiting a long time until update operations unlock. In this paper we propose new algorithms for lock-free search operation. First, we develop a new technique using a linked-list technique on the node. The linked-list enable lock-free search when search operations search a node. Next, we propose a new technique using a version technique. The version technique enable lock-free search on the node that update operations is to be splitting.

키워드 : 공간 데이터베이스 관리 시스템(Spatial Database Management System), 다차원 색인구조(Multi-Dimensional Index Structure), 동시성 제어 기법(Concurrency Control)

1. 서 론

최근 공간 데이터베이스 시스템을 이용한 다양한 종류의 지리 정보 서비스들이 인터넷을 통하여 제공되고 있다. 이러한 서비스들의 특징은 빠른 응답시간을 요구하며, 질의의 대부분이 공간 객체 검색을 위한 연산이다. 따라서 이러한 빠른 응답시간을 요구하는 검색 위주의 공간 데이터베이스 시스템을 위한 효과적인 공간 색인은 필수적이라고 할 수 있다.

지난 수년 동안 공간 객체 검색을 효과적으로 할 수 있는 많은 색인 기법들이 제안되어져 왔다[2, 3, 5, 11-13, 15-18]. 이들 대부분은 R-Tree의 기본 구조를 갖으며[2, 5, 18], 어떻게 검색의 효율을 높일 수 있느냐에 초점을 맞추어 R-Tree를 변형한 동시성 제어 기법들을 제안하였다. 그 결과 CGiST[8], R^{link}-Tree[7], [15]와 같은 기법들이 제안되었다. 근본적으로 기존의 제안되었던 다차원 색인의 동시성 제어기법들의 검색연산은 반드시 검색하고자 하는 노드에 공유 락을 획득함으로써 같은 노드에 갱신연산이 동시에 연산하는 것을 방지한다. 그러나 빠른 응답시간을 요구하는 공간검색질의가 락을 획득하기 위해 기다리는 것 자체도 공간검색질의의 응답을 지연시키는 문제를 갖는다. 뿐만 아니라 R-Tree기반의 색인구조는 비공간 색인과는 달리 검색연산의 블록킹(blocking) 상태가 빈번히 발생할 수 있는 구조이다. 그 이유는 세 가지로 요약될 수 있다. 첫 번째로, R-Tree계열의 색인 기법은 단말노드(leaf node)에 키가 삽

※ 본 연구는 정보통신부의 대학 S/W 연구센터 지원사업의 연구 결과임.
† 준 회 원 : 인하대학교 대학원 전자계산공학과
†† 종신회원 : 인하대학교 전자계산공학과 교수
논문접수 : 2004년 3월 12일, 심사완료 : 2004년 5월 12일

입되거나 삭제될 때마다 단말노드로의 패스(path)에 있는 비단말노드(internal node)들의 영역이 변경되어야 하는 특징을 갖기 때문에 여러 노드에 락을 획득해야만 한다. 최악의 경우에는 갱신연산이 루트 노드(root node)까지 락을 획득함으로써 전체 색인에 락을 거는 효과가 발생할 수도 있다. 두 번째로는 단말노드에 키를 삽입할 공간이 없어서 노드가 분할되어야 하는 경우에 B-Tree와 같은 비공간 색인[1]과는 달리 분할하는데 많은 시간이 걸림으로써 장시간 락을 획득하고 있어야 하는 특징을 갖는다[19]. 세 번째는 삭제연산 시 삭제될 엔트리(entry)를 찾기 위해 여러 노드를 탐색해야하기 때문에 많은 노드에 락을 획득해야 하는 특징을 갖는다. 이러한 세 가지 특징의 공통점은 갱신연산이 노드에 락을 장시간 유지함으로써 검색연산이 지연되는 문제를 갖고 있다는 것이다.

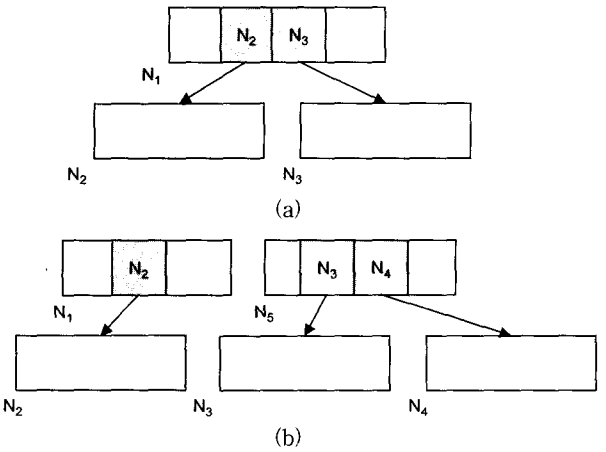
따라서 본 논문에서는 검색연산의 블록킹 상태를 근본적으로 막는 동시성 제어 기법들을 제안한다. 비공간 색인의 경우는 검색연산이 락을 획득하지 않고 색인을 탐색하는 방법에 대한 연구가 진행되었으며, T-tree[9]를 기반으로 한 동시성 제어 기법[14]가 대표적이다. 본 논문에서는 논 블록킹 검색연산을 위한 두 가지 동시성 제어 기법을 제안한다. ① 첫 번째는 검색연산이 노드를 검색할 때 다른 연산으로 인하여 블록킹 되는 상태를 방지하기 위한 방법으로써 검색연산이 노드를 읽고 있는 동안에도 같은 노드에 갱신연산을 할 수 있는 링크드 리스트 기반의 동시성 제어 기법(**CCLL** : a Concurrency Control method based on Linked List)을 제안한다. 노드 내의 엔트리들은 링크드 리스트로 연결되어 있으며, 링크드 리스트에 새로운 엔트리를 삽입하거나 기존 엔트리를 삭제하는 연산은 단위 연산(atomic action)을 보장할 수 있기 때문에 검색연산이 락을 획득함이 없이 노드를 검색할 수 있도록 한다[14]. ② 두 번째는 검색연산이 락이나 래치없이 색인을 탐색할 수 있는 버전 기반의 동시성 제어 기법(**CCV** : a Concurrency Control method based on Version)을 제안한다. CCLL이 하나의 노드를 위한 동시성 제어 기법이라면, CCV는 색인 전체에 대한 동시성 제어 기법이다. CCV는 분할할 노드를 위하여 새로운 버전을 만들고, 만든 버전을 이용하여 분할을 수행함으로써 검색연산이 락이나 래치없이 색인 탐색을 가능하게 한다. 그러나 CCV는 구버전과 같은 쓰레기 노드에 대한 부가적인 처리비용(garbage collection)이 드는 단점이 있지만, 검색연산이 블록킹이 없이 진행할 수 있기 때문에 충돌이 잦은 환경에서 검색연산의 성능이 2배에서 4배 향상되었음을 성능평가 결과에서 보여주고 있다.

논문의 나머지 부분은 다음과 같다. 2장에서는 관련연구에 대해서 기술하고, 3장에서는 본 논문에서 제안하고자 하는 색인 기법들에 대해서 기술한다. 4장에서는 트랜잭션의 일관성에 관하여 기술하고, 5장에서는 알고리즘을 분석한다. 6장에서는 기존 색인 기법과의 성능 비교 평가를 실시하고, 마지막으로 7장에서 결론을 맺는다.

2. 관련 연구

기존의 다차원 색인의 동시성 제어 기법들은 검색연산이 최소한의 락이나 래치를 획득하기 위한 연구를 진행하였으며, 일차원 색인의 동시성 제어 기법 중에는 검색연산이 락이나 래치를 획득하지 않고 색인을 탐색하는 T-tree기반의 동시성 제어 기법 제안되었다.

이 장에서는 다차원 색인을 탐색할 때 발생할 수 있는 문제를 정의함으로써 기존의 다차원 색인의 동시성 제어 기법들은 어떠한 방법을 가지고 이러한 문제를 해결하였는지 설명한다. (그림 1)은 색인을 탐색할 때 발생할 수 있는 문제점을 설명한다.



(그림 1) 탐색 경로의 변경문제

(그림 1)(a)에서 T_1은 N_3의 락을 획득하기 위해 기다리고 있는 삽입연산이고, T_2는 N_3에 삽입을 진행 중인 삽입연산이며, N_3과 N_1은 빈 엔트리가 없는 노드라고 가정하자. T_2가 N_3에 삽입하기 위해 N_3을 분할하고, 새로운 노드 N_4를 만든 다음 N_1에 N_4를 위한 엔트리를 삽입하기 위해 또다시 N_1을 분할하여 N_5를 만들고, (그림 1)(b)의 상태를 만들었을 때, T_1은 N_3의 락을 획득하게 되고, N_3에 엔트리를 삽입하고 나서 N_3의 변경된 MBR을 N_1에 반영하기 위해 N_1을 탐색하게 된다. 그러나 T_2로 인하여 N_1에 있던 N_3의 엔트리는 N_5로 이동하였기 때문에 해당 엔트리를 찾을 수 없는 탐색 경로의 변경문제가 생긴다.

탐색 경로의 변경문제를 해결하기 위한 기존의 동시성 제어 기법들을 두 가지 형태로 나누어 볼 수 있다. ① 첫 번째는 자신이 탐색한 경로상의 변경을 허용하지 않는 방법이고, ② 두 번째는 탐색한 경로상의 변경을 허용하지만 변경을 검출하고 나서 이를 바로 잡는 방법이다. ①의 대표적인 기법은 락 결합(Lock coupling)방식으로써 현재 노드의 락을 획득하고 나서 자식노드로 내려가지 전에 현재 노드의 락을 해제하지 않고, 자식노드를 탐색하는 방법이다. 이러한 방식은 삽입연산 시 단말 노드의 변경된 MBR을 부모노드에 반영해야 하는 R-tree 계열의 동시성 제어 기법으로

는 적합하지 않은 방식이며, 갱신연산으로 인하여 락이 장시간 유지됨으로써 검색연산의 효율을 떨어뜨리는 문제를 갖는다. ②는 자식노드가 변경되었음을 판단할 수 있는 방법과 잘못된 경로를 바로 잡을 수 있는 방법이 필요하며, 이러한 대표적인 기법은 R^{link}-Tree이다. R^{link}-Tree는 자식노드가 변경되었음을 판단하기 위하여 **LSN**(**L**ogical **S**equence **N**umber)를 이용하며, 노드 간 링크를 유지함으로써 잘못된 경로의 문제를 해결한다. R^{link}-Tree는 B^{link}-Tree [10]를 다차원 색인에 적용한 색인 기법이며, R-tree는 엔트리들 간 순서가 없다는 특징을 가지고 있기 때문에 [10]의 최대키와 같은 역할을 LSN이 담당한다. 부모노드의 엔트리에 각각 자식노드의 LSN을 유지하고 있으며, 부모노드에서 읽어온 엔트리 LSN이 자식노드에 저장되어 있는 노드 LSN과 틀릴 경우, 부모노드에서 읽어온 LSN을 만날 때까지 같은 레벨(level)의 링크를 따라감으로써 탐색 경로의 변경 문제를 해결한다. 이러한 방식은 노드에 자식노드들의 LSN을 저장함으로써 노드의 팬 아웃(fan out)을 감소시키는 문제를 가지고 있으며, 단말노드의 MBR이 변경되었을 경우 부모노드에 반영하기 위하여 락 결합 방식을 이용하여 반영하여야 하기 때문에 검색연산의 효율을 떨어뜨리는 문제를 갖는다.

기존의 언급한 다차원 동시성 제어 기법들은 검색연산이 락이나 래치를 적어도 한번 이상은 노드에 획득하고 색인을 탐색해야 하며, 이로 인하여 노드의 탐색시간이 길어지는 문제를 갖는다. 과거 일차원 색인을 위한 동시성 제어 기법 중에서 검색연산이 락이나 래치를 획득하지 않고 색인을 탐색할 수 있는 기법이 제안되었었다. 그 대표적인 기법이 [14]이다. 갱신연산이 노드에 엔트리를 삽입하고자 할 경우 노드의 엔트리들을 새로운 노드에 복사한 다음, 복사된 노드에 엔트리를 삽입하는 방법을 사용하여 검색연산이 락이나 래치를 획득해야 하는 문제점을 해결하였다. 그러나 이와 같은 기법은 하나의 삽입연산으로 인하여 쓰레기 노드가 하나씩 생김으로써, 많은 쓰레기 노드를 삭제하기 위한 부가적인 비용이 드는 단점을 가지고 있다.

본 논문에서는 [14]에서처럼 버전 기반의 동시성 제어 기법을 사용함으로써 검색연산이 색인을 탐색할 때 락이나 래치를 획득하지 않는 기법을 제안한다. 그러나 본 기법은 [14]와는 달리 삽입연산이 노드에 엔트리를 삽입할 때마다 새로운 버전을 생성하지 않고, 분할 시에만 새로운 버전을 생성하고 생성된 버전을 이용하여 분할을 수행하여 과도한 쓰레기 노드로 인해 생기는 문제점을 해결하고, 각 노드의 엔트리를 링크드 리스트로 연결하여 노드 탐색 시 락이나 래치를 걸지 않는 구조를 제안한다.

3. 제안하는 동시성 제어 기법

이 장에서는 본 논문에서 제안하는 R-tree 기반의 동시성 제어 기법에 대하여 자세히 기술한다. 먼저 검색연산이 노드를 탐색하고 있는 동안에도 갱신연산이 가능한 링크드 리스트 기반의 동시성 제어기법(CCLL)에 대해서 설명한다. 다음으로 버전 기반의 동시성 제어 알고리즘(CCV)과 본 기법에서 발생하는 부가적인 쓰레기 노드의 삭제에 대해서 기술한다.

3.1 링크드 리스트 기반의 동시성 제어 기법(CCLL)

본 절에서는 링크드 리스트 기반의 동시성 제어 기법에 대해서 설명한다. 먼저 제안하고자 하는 기법의 이해를 돕기 위해 검색연산이 락을 획득하지 않고 노드를 탐색할 때 발생할 수 있는 문제에 대해서 서술하고, 다음으로 링크드 리스트 기반의 동시성 제어 기법에 대해서 자세히 설명한다.

3.1.1 문제 정의

문제를 간단히 하기 위해 각 노드에 유지해야하는 부가적인 정보를 최소한으로 할 수 있는 노드 구조를 고려한다. 각 단말노드의 엔트리는 [OID, MBR]로 구성되어 있다. OID는 객체를 나타내는 유일한 식별자이고, MBR(**M**inimum **B**ounding **R**ectangle)은 객체 전체를 감싸고 있는 최소 경계 사각형이다. MBR은 4개의 배정밀도 실수로 좌측상단의 포인트 좌표와 우측하단의 포인트 좌표로 구성된다. 노드의 구조는 비공간 색인 기법인 B-Tree와는 달리 R-Tree는 각 엔트리들 간의 순서가 없기 때문에 중간에 엔트리를 삭제하여도 그 뒤에 있는 엔트리들이 앞으로 이동하지 않아도 되며, 다시 삭제된 엔트리 위치에 삽입이 되었다 하여도 문제가 되지 않기 때문에 엔트리 삽입 시 노드 앞에서부터 순차적으로 빈 엔트리에 삽입되고, 삭제 시 단지 삭제되었음을 표시하는 구조라고 가정하자. 또한 서로 다른 갱신연산들은 동시에 같은 노드에 연산을 수행할 수 없고, 단지 검색연산만이 같은 노드에 다른 갱신연산과 병행하여 수행할 수 있다.

문제는 검색연산이 진행 중인 엔트리에 삭제연산이 발생하고, 곧바로 삭제된 엔트리에 삽입이 이루어지는 엔트리 재사용의 문제이다. 삭제연산은 단지 삭제 표시만 하고, 연산을 종료하기 때문에 검색연산에는 영향을 미치지 않는다. 그러나 삭제된 엔트리에 다시 재사용한다면 검색연산은 잘못된 OID나 MBR과 교차연산을 수행할 수 있게 된다. (그림 2)은 그러한 과정을 설명한 예이다.

T_1(select)	T_2(delete)	T_3(insert)

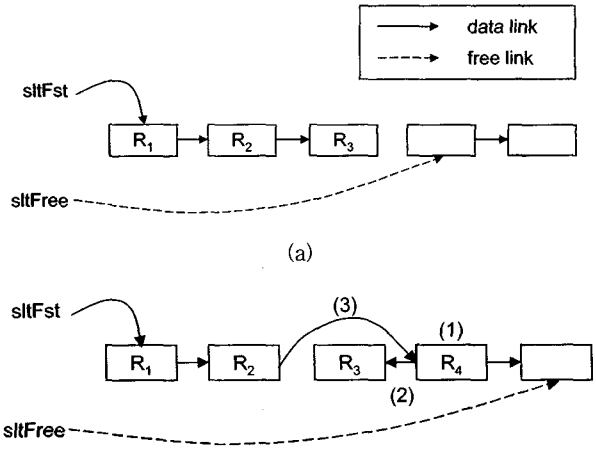
for (all entries E)
if (E.use is true)
⋮
if (E.MBR intersects MBR)
⋮
return E.OID ;
⋮
	E.use = false ;
	unlock(node) ;
	⋮
		⋮
		lock(node)
		if (E.use is false)
		copy (E.MBR, MBR) ;
		copy (E.OID, OID) ;
		E.use = true ;
		end if

(그림 2) 검색, 삭제, 삽입연산의 동시 수행

T_1(검색연산)은 노드내의 모든 엔트리에 질의 영역과 교차되는 엔트리들을 구해낸다. 만약 엔트리(E)가 사용 중이라면 E의 MBR과 질의 영역과 교차 연산을 수행하게 되는데 이때 T_2(삭제연산)로부터 E가 삭제되고, 바로 T_3(삽입연산)이 수행되어 방금 삭제된 E에 객체의 MBR과 OID를 복사하게 된다. 그러나 T_1에 의해서 사용되고 있는 MBR과 OID는 다른 객체의 내용으로 변경되어 잘못된 MBR로 교차연산을 하거나 잘못된 OID를 결과로써 넘겨줄 수가 있다.

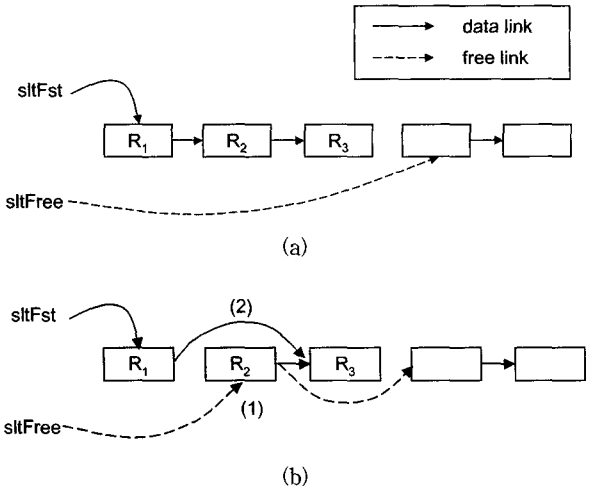
3.1.2 CCLL의 엔트리 삽입/삭제

노드 구조는 엔트리들 간 링크드 리스트로 연결된 구조이다. 예를 들어 링크드 리스트 구조에서의 삽입연산은 삽입되는 엔트리 E_{ins}가 E_1와 E_2 사이에 삽입된다고 할 때 E_1의 링크가 E_{ins}와 연결이 되어야만 검색연산은 E_{ins}를 읽을 수 있다. E_1의 링크와 E_{ins}를 연결하는 연산은 하나의 단위 연산으로 가능하기 때문에 삽입연산이 단위 연산이라고 할 수 있다. 왜냐하면 링크의 크기는 워드 얼라인(word-align)에 맞춘 크기를 가지고 있고, 링크를 연결하는 연산은 하나의 명령어(instruction)로 가능하기 때문에 링크를 연결하는 연산 중에 인터럽트(interrupt)가 발생할 수가 없다. 따라서 링크가 연결되면 검색연산은 삽입된 엔트리를 볼 수 있지만 삽입이 진행 중인 엔트리는 검색연산이 탐색할 수는 없다. (그림 3)은 엔트리를 삽입하는 과정을 설명하고 (그림 4)는 삭제하는 과정을 설명한다.



(그림 3) 엔트리 삽입 과정

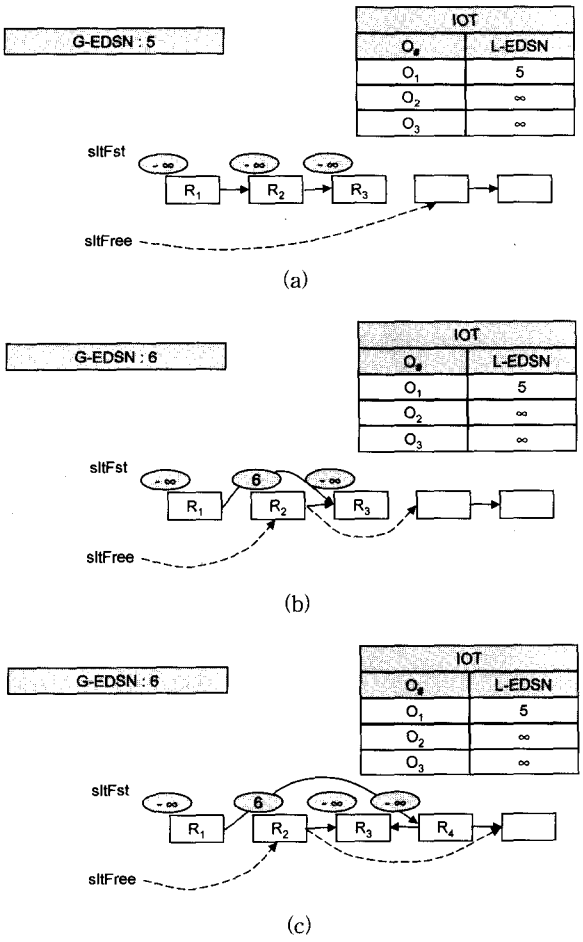
sltFst는 노드의 첫 번째 슬롯(slot)을 말하고, sltFree는 노드의 첫 번째 빈 슬롯을 가리킨다. (그림 3)(a)상태에서 R_2와 R_3사이에 삽입연산을 한다면, (1) 빈 슬롯을 찾은 다음, OID와 MBR을 복사한다. (2) R_3와 R_4의 데이터 링크(data link)를 연결하고, (3) 마지막으로 R_2를 R_4와 연결한다. (3)의 과정이 (2)보다 먼저 일어난다면 검색연산은 R_4를 읽을 수 있고, R_3로의 링크가 아직 연결되어 있지 않기 때문에 R_3를 읽을 수 없는 문제가 생긴다. 따라서 (2)와 같은 과정은 (3)전에 수행되어야 한다. (그림 4)(a)상태에서 R_2를 삭제한다고 하면, (1) 삭제할 슬롯을 찾고, (2) R_1과 R_3을 연결한다. 만약 R_2와 R_3의 데이터 링크를 해제한다면, 삭제하기 전 이미 R_2를 읽고 있는 검색연산은 R_3로 이동할 수 없는 문제가 생기기 때문에 R_2와 R_3의 데이터 링크를 해제해서는 안 된다.



(그림 4) 엔트리 삭제 과정

다시 3.1절의 엔트리 재사용 문제에 대하여 본 논문에서는 어떻게 해결하는지 설명한다. 엔트리 재사용의 근본적인 문제는 삭제된 엔트리에 대해서 검색연산이 읽고 있을지 모르기 때문에 삭제된 엔트리를 재사용할 수 없다는 것이다. 따라서 본 논문에서는 삭제된 엔트리에 검색연산이 읽고 있지 않다는 것을 보장하기 위한 방법을 고려한다. 본 논문에서는 엔트리삭제계수(**EDSN** : Entry Delete Sequence Number)를 노드의 각 엔트리에 유지하고, 색인에 수행되고 있는 연산들을 관리하는 색인연산테이블(**IOT** : Index Operation Table)을 이용함으로써 이러한 문제를 해결한다. EDSN은 순차적으로 증가하는 값이며, 전역엔트리삭제계수(**G-EDSN** : Global **EDSN**)와 지역엔트리삭제계수(**L-EDSN** : Local **EDSN**)가 있다. **G-EDSN**은 색인마다 하나씩 유지하며, 삭제연산을 하기 위해 링크를 해제하고 난후 **G-EDSN**을 증가하고, 증가된 **G-EDSN**을 엔트리에 저장하기 위해 사용한다. **L-EDSN**은 연산을 수행하기 전에 가장 최근의 **G-EDSN**을 기록해 놓은 값이다. **L-EDSN**은 **IOT**에 유지되며, 각 연산들마다 하나씩 유지된다. **IOT**에서 관리되는 연산들의 생명주기(lifetime)는 하나의 노드를 검색하고, 종료되는 시점까지이며, 다음 노드를 탐색하기 위해서는 새로운 **G-EDSN**을 **IOT**의 **L-EDSN**으로 복사하여야 한다. 이는 연산들의 생명주기를 아주 짧게 만듦으로써 엔트리 재사용을 빠르게 하기 위함이다. 검색연산을 제외한 갱신연산들의 **L-EDSN**은 ∞이다. **L-EDSN**은 삽입연산이 수행될 때 삽입되어질 엔트리의 **EDSN**을 확인하고 **EDSN**보다 작은 **L-EDSN**을 가진 연산이 **IOT**에 존재한다면, 그 엔트리는

검색연산에 의해서 검색되고 있는 엔트리일지도 모르기 때문에 그 엔트리에 삽입하지 않기 위해 사용된다. 각 엔트리의 **EDSN** 초기 값은 -∞이다. (그림 5)는 **EDSN**을 이용한 삭제 후 삽입과정에 대한 설명이다.

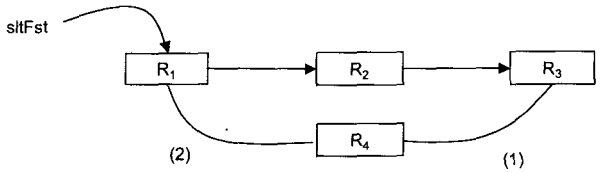


(그림 5) 삭제 후 삽입 과정

(그림 5)(a)에서 초기 **G-EDSN**은 5이고, 3개의 연산이 동시에 실행되고 있다. O_1은 L-EDSN 값이 5이고, 검색연산을 한다. O_2는 삭제연산을 하고, O_3는 삽입연산을 하며, O_1이 R_2에 대하여 키를 비교하는 시점이라고 가정하자. 이 때 O_2가 R_2를 삭제한다고 하면 먼저 R_1에서 R_2로의 링크를 해제하고 난후 R_2의 **EDSN**은 **G-EDSN**을 증가시킨 값인 6으로 저장한다. (그림 5)(b)의 그림이 R_2가 삭제되고, **EDSN**이 저장된 모습을 보여준다. 그 다음으로 (그림 5)(b)의 상태에서 삽입연산이 수행되면 sltFree는 R_2를 가리키기 때문에 R_2에 삽입하려고 할 것이다. 그러나 **IOT**의 L-**EDSN**들 중에 최소 L-**EDSN**인 5보다 R_2의 **EDSN** 6이 크기 때문에 R_2에 삽입을 해서는 안 된다. **EDSN**이 작다는 것은 연산들 중 누군가가 삭제연산으로 링크를 해제하기 전에 그 엔트리를 검색하고 있었을 수도 있다는 가능성을 가지고 있기 때문에 그 엔트리에 삽입을 해서는 안된다. 따라서 (그림 5)(c)와 같은 그림으로 각 엔트리는 연결된다. O_1이 연산을 종료한 이후에 삽입연산을 수행한다면 R_2에 삽입되어질 수 있다.

3.1.3 CCLL의 엔트리 변경

노드의 엔트리에 대한 변경연산은 엔트리의 MBR을 변경하거나, 엔트리의 노드 포인터를 변경하는 연산이다. 검색연산이 락이나 래치없이 노드를 탐색하면서 변경연산이 엔트리의 내용을 변경하는 것은 단위 연산을 보장할 수 없기 때문에 본 논문에서는 변경연산을 엔트리의 삽입연산과 삭제연산을 수행하는 방법을 사용한다. (그림 6)은 그러한 과정을 설명한다.



(그림 6) 엔트리 변경 과정

삽입해야할 엔트리가 R_4이고, 삭제해야하는 엔트리가 R_2라고 한다면 R_4와 R_3의 링크를 먼저 연결하고 다음으로 R_1과 R_4를 연결함으로써 검색연산이 어떠한 문제없이 검색할 수 있도록 보장한다.

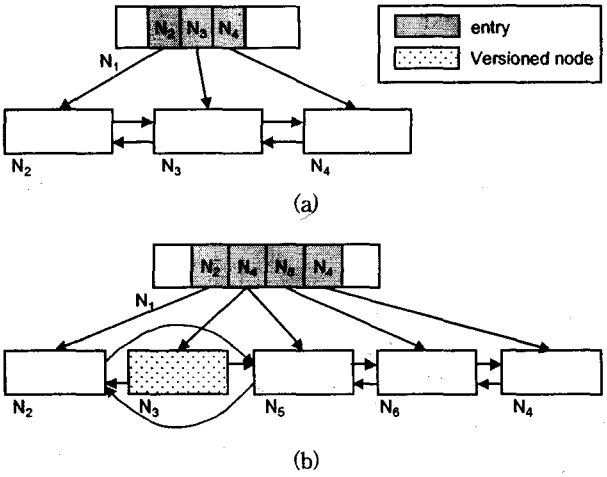
3.2 버전 기반의 동시성 제어 기법(CCV)

본 절에서는 버전 기반의 동시성 제어 알고리즘에 대해서 기술한다. 전체적인 노드 구조에 대해서 설명하고, 마지막으로 버전 기반의 검색, 삽입 및 분할, 삭제 알고리즘에 대해서 기술한다.

3.2.1 색인 구조

본 논문에서 제안하는 색인 구조는 두 가지 목적을 가지고 설계한다. ① 첫 번째는 검색연산이 락이나 래치없이 색인을 탐색할 수 있는 구조와, ② 두 번째는 갱신연산간 동시성을 높이기 위한 색인 구조를 목적으로 한다. ①은 노드 탐색 시 엔트리를 링크드 리스트로 연결하여 검색연산의 블록킹 상태를 제거하고, 분할될 노드의 삭제연산과 새로운 노드의 삽입연산을 하는 기존 기법들의 분할 방법을 새로운 노드 두 개를 만들어 각 노드에 삽입하는 분할 방식으로 대체하여 검색연산의 블록킹 상태를 제거한다. ②는 갱신연산 시 탐색 경로의 변경문제가 발생하는 것을 막기 위하여 R^{link}-Tree처럼 링크를 이용하여 갱신연산간 동시성을 높인다. 그러나 R^{link}-Tree와는 달리 노드 LSN을 사용하지 않고, 노드 간 링크만을 이용한다. LSN을 유지하지 않고도 탐색이 가능한 이유를 LSN이 색인에 이용되는 3가지 형태로 나누어 생각해 볼 수 있다. ① 삽입연산 시 삽입할 위치를 찾고자 할 때와 ② 삽입하고 난후 부모노드의 MBR변경 및 ③ 분할시 부모노드에 분할한 노드를 위한 엔트리를 삽입할 위치를 찾을 때 사용한다. ①의 경우에서는 자신이 노

드를 탐색하고 난후 탐색한 노드의 자식노드가 분할한 경우에 정확한 위치를 찾기 위해서 LSN이 이용된다. 이와 같은 경우, 본 기법에서는 노드가 분할되어 있다고 한다면 스택을 하나 올라가서 다시 재시도를 한다. R^{link}-Tree처럼 링크를 따라가서 삽입해야할 위치를 찾는 것은 노드간 MBR 계산으로 인하여 부하를 가중시키는 결과를 초래하며, 만약 노드가 계속적으로 분할되었다고 할 경우, 삽입할 위치를 찾는 것은 여러 개의 노드를 탐색해야하는 문제를 갖는다. 따라서 이러한 경우에는 부모노드를 다시 재시도하는 방법을 사용한다. ②와 ③ 같은 경우는 자신이 어떤 엔트리를 찾아야 할지를 아는 경우이기 때문에 LSN을 이용하지 않고도, 노드 간 링크만 연결되어 있다면 MBR을 변경해야할 엔트리 또는 분할시 삽입해야할 엔트리를 찾을 수 있다. 따라서 본 논문에서는 LSN을 이용하지 않고, 노드 간 링크만 연결하여 갱신연산의 동시성을 높인다. (그림 7)은 삽입연산이 노드를 분할 한 후의 색인 구조에 대해서 설명하고 있다.



(그림 7) 노드 분할 후의 색인 구조

(그림 7)(a)에서 노드 N_3이 분할을 한다고 가정하자. 새로운 노드 N_5, N_6를 생성하고, N_2, N_5, N_6, N_4의 링크를 연결한다. 만약 N_1을 거쳐 N_3에서 분할이 종료할 때까지 기다리던 연산이 분할 종료 후 N_3에 락을 획득하였을 경우, N_3이 버전되어 있음을 확인한 후에는 다시 스택의 하나 위 N_1으로 올라가서 다시 재 시도를 한다. R^{link}-Tree와는 달리 추가적으로 노드에 왼쪽 링크를 달고 있어야 한다. 예를 들어 (그림 7)(b)와 같은 경우 N_3의 분할로 인하여 N_2의 오른쪽 링크를 N_5로 변경시켜야 하기 때문에 N_3는 왼쪽 노드 N_2를 알고 있어야 한다.

3.2.2 CCV의 검색 알고리즘

검색연산은 질의 범위에 속하는 단말노드들의 모든 엔트리를 찾는 연산이다. 본 논문에서 제안하는 검색연산은 락이나 래치를 걸지 않고 노드를 탐색한다. 이러한 장점을 가능하게 하는 것은 노드의 엔트리들이 링크드 리스트로 연결되었고, 또한 삽입연산의 분할 시에 기존 노드는 그대로 두고, 새로운 두 개의 노드를 만들어 분할하기 때문이다. (그림 8)은 검색과정을 설명한 그림이다.

```
 1 findfirst(STACK Stack, RECT Rect)
 2   push(Stack, Root)
 3   returnfindEntry(Stack, Rect)
 4 end
 5
 6 findnext(STACK Stack, RECT Rect)
 7   returnfindEntry(Stack, Rect)
 8 end
 9
10 findEntry(STACK Stack, RECT Rect)
11   while not empty(Stack)
12     Pointer = pop(Stack)
13     if (Pointer is pointer to object)
14       return Pointer
15     else
16       for all entries Entry of Pointer
17         if(Entry is intersecting Rect)
18           push(Stack, NODE(Entry))
19         end
20       end
21     end
22   end
23 end
```

(그림 8) 검색 알고리즘

findfirst는 질의 범위에 속하는 첫 번째 엔트리를 찾는 함수이다. 루트노드(root node)를 스택에 푸쉬(push)하고, findEntry함수를 호출하는 과정으로 첫 번째 엔트리를 찾아낸다. findEntry는 스택이 비어 있을 때까지 반복하면서 노드를 순회한다. 예를 들어 색인에 루트노드가 하나 밖에 없고, findfirst함수에서 findEntry가 호출되었다고 가정할 때 12라인의 Pointer에는 루트노드의 포인터(pointer)가 복사되고, Pointer는 객체를 가리키는 포인터가 아니기 때문에 16라인을 수행하게 된다. Rect와 교차되는 모든 엔트리들은 스택에 푸쉬되고, 12라인이 다시 수행되었을 때 Pointer는 객체를 가리키는 포인터이기 때문에 해당 Pointer를 리턴(return)하게 된다. 그 다음 findNext를 호출하였다면, 스택에 남아있는 모든 Pointer들을 팝(pop)하면서 하나씩 리턴하게 된다.

3.2.3 CCV의 삽입 알고리즘

삽입연산은 크게 두 부분으로 나누어 볼 수 있다. 첫 번째로는 객체를 삽입할 단말노드를 찾는 연산이고, 두 번째는 찾은 단말노드에 객체를 삽입하는 연산이다. (그림 9)는 삽입에 대한 알고리즘이다.

삽입연산(insert)은 5라인의 findLeaf함수를 호출하면서 시작된다. findLeaf는 비단말노드의 MBR을 최소로 늘릴 수 있는 엔트리를 찾으면서 단말노드 방향으로 색인을 탐색하며, 단말노드를 만나면 함수를 종료한다. findLeaf가 단말노드를 찾기 위해 탐색하면서 탐색된 비단말노드와 단말노드를 스택에 쌓는 과정을 수행하는데, 이는 삽입연산 시 부모

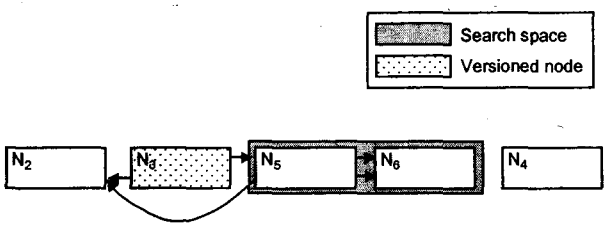
```
1 insert(STACK Stack, RECT Rect)
2   Retry = true
3   while(Retry is true)
4     Retry = false
5     findLeaf(Stack, Root, Rect)
6     Leaf = pop(Stack)
7     if(Leaf is not full)
8       insert Rect to Leaf
9       if(MBR(Leaf) changed)
10        updateParent(Stack, Leaf, MBR(Leaf))
11      end
12    else
13      Retry = splitLeafNode(Stack, Leaf, Rect)
14      if(Retry is not true)
15        set version flag to Leaf
16        add Leaf to garbage collector
17      end
18    end
19    unlock(Leaf)
20  end
21 end
22
23 findLeaf(STACK Stack, NODE Node, RECT Rect)
24   lock(Node)
25   push(Stack, Node)
26   while(Node is not leaf node)
27     while(Node versioned)
28       unlock(Node)
29       Node = pop(Stack)
30       if(Node == null)
31         Node = Root;
32       end
33       lock(Node)
34     end
35      Entry = entry on Node leading to minimal MBR for
Rect
36     push(Stack, NODE(Entry))
37     unlock(Parent)
38     Node = NODE(Entry)
39     lock(Node)
40   end
41 end
42
43 updateParent(STACK Stack, NODE Node, RECT Rect)
44   if(Stack is not empty)
45     Parent = pop(Stack)
46     Parent = findLatestNode(Parent, pointer to Node)
47     updateEntry1(Parent, Entry, Rect)
48     if(MBR(Parent) changed)
49       updateParent(Stack, Parent MBR(Parent))
50     end
51     unlock(Parent)
52   end
53 end
54
55 NODE findLatestNode(NODE Node, POINTER Pointer)
56   lock(Node)
57   while(Node versioned)
58     unlock(Node)
59     Node = right link of Node
60     lock(Node)
61   while(Node is not versioned)
62     Entry = find entry for Pointer
63             in Node
64     if(Entry is not null)
66     end
67     unlock(Node)
68     Node = right link of Node
69     lock(Node)
70   end
71 end
72 end
```

(그림 9) 삽입 알고리즘(insert와 findLeaf, updateParent, findLatestNode)

노드의 MBR을 확장하거나 노드의 분할이 일어나는 경우에 사용되어진다. 27라인에서처럼 만약 노드가 버전되어 있다면 노드의 락을 획득하기 위하여 기다리는 중에 다른 연산에 의해서 노드가 분할되어진 경우이다. 이러한 경우는 스택을 이용하여 부모노드를 다시 탐색하게 된다. 만약 찾은 단말노드에 삽입할 공간이 있다면 해당 노드에 MBR을 삽입하고, 삽입되진 후 단말노드에 MBR이 변경된다면 부모노드의 엔트리 MBR을 updateParent함수를 이용하여 변경시킨다. 빈 엔트리가 없어서 삽입을 하지 못하는 경우에는 노드를 분할하는 과정을 거친다. 노드의 분할이 실패한다면 삽입연산은 다시 처음부터 재 시작한다. 본 논문에서는 빈 엔트리가 없는 상태, 즉 노드가 꽉 차있는 상태를 노드에 들어갈 수 있는 최대 엔트리 개수에서 하나 적은 개수가 들어있는 상태로 정의한다. 이는 제안하는 기법이 updateParent와 같이 MBR을 변경하는 연산에 대하여 삽입연산과 삭제연산으로 MBR 변경연산을 대체하기 때문이다. updateParent는 스택을 이용하여 부모노드 방향으로 이동하면서 자식노드의 노드 MBR이 변경되었을 경우 부모노드의 엔트리 MBR을 변경시키는 작업을 수행한다. 만약 스택에 저장되어 있는 부모노드들이 버전되었다면 findLatestNode함수를 호출하여 버전되어 있는 노드를 위한 최신 버전을 찾는다. 최신 버전을 찾은 다음 updateEntry1을 이용하여 노드의 엔트리 MBR을 변경한다. updateEntry1은 3.1.3절에서 언급한 CCLL의 변경방법을 이용하여 수행한다. 만약 노드의 엔트리 MBR의 변경으로 인하여 노드 MBR이 변경된다면 다시 재귀적으로 부모노드의 updateParent를 호출한다. updateParent는 3.1.1절에서 언급한 엔트리 재사용 문제로 인하여 노드에 빈 엔트리가 있지만 삽입을 못하는 경우가 발생할 수 있다. 이러한 경우는 잠시 다른 연산으로 제어를 넘겨주고 나서 다시 엔트리를 재사용할 수 있는지 재 시도를 한다. 이는 검색연산이 엔트리를 검색하고 있을지 모르기 때문에 검색연산이 노드의 탐색을 마칠 때까지 기다린다. 검색연산은 락이나 래치를 걸지 않는 연산이며, 또한 IOT에 기록되는 검색연산의 L-EDSN은 아주 짧은 생명주기를 갖기 때문에 updateParent는 장시간 기다리지 않고 엔트리를 재사용할 수가 있다. findLatestNode함수는 Pointer의 엔트

리를 포함하는 버전이 안된 최신 노드를 찾는 과정을 수행한다. (그림 10)은 그러한 과정을 설명한다.

(그림 10)에서 N_3이 버전되어 있다고 할 경우 찾아야할 검색 공간(search space)은 N_5와 N_6이기 때문에 N_3의 오른쪽 링크를 따라 N_5와 N_6로 이동하면서 찾고자 하는 엔트리를 갖는 최신 버전을 찾는다.



(그림 10) findLatestNode의 수행 과정

노드의 분할은 splitLeafNode와 splitInternalNode함수를 이용하여 수행된다. 단말노드가 분할되어야 한다면 splitLeafNode를 호출하고, 단말노드의 분할로 인하여 발생하는 비단말노드의 분할을 위해서 splitInternalNode을 호출한다. 기본적으로 분할 함수들은 분할할 노드를 그대로 두고 새로운 노드 두개를 생성하여 분할되기 전의 노드의 상태를 변경하지 않고 유지하는 버전기반의 분할 기법을 사용한다. splitLeafNode와 splitInternalNode의 차이는 분할할 노드의 종류가 단말노드와 비단말노드에 따라 달라진다. 분할할 노드가 단말노드라면 분할 후 적당한 노드에 Rect를 삽입하면 되지만 비단말 노드의 경우는 분할할 노드에 새로 생성한 두 개의 자식노드를 위한 엔트리 두 개를 삽입하여야 한다. 따라서 기존에 분할할 노드를 가리키고 있던 엔트리를 삭제함과 동시에 두개의 새로운 엔트리를 삽입하여야 하기 때문에 본 기법에서의 분할 방법은 분할할 노드의 종류에 따라 다른 함수를 호출한다.

```
1 boolean splitLeafNode(STACK Stack, NODE Node, RECT Rect)
2   Left = the left sibling node of Node
3   Right = the right sibling node of Node
4   unlock(Node)
5   lock(Left)
6   if(Left was vesioned)
7     unlock(Left)
8     lock(Node)
9     return true
10  end
11  lock(Node)
12  if(Node was vesioned)
13    unlock(Left)
14    return true
15  end
16  lock(Right)
17  if(Right was vesioned)
18    unlock(Right)
19    unlock(Left)
20    return true
21  end
22  newLeft = create new node
23  newRight = create new node
24  copy left half of Node to newLeft
25  copy right half of Node to newRight
26  linking newLeft and newRight
27  if((MBR(newLeft+Rect)-MBR(newLeft)) >
      (MBR(newRight+Rect)-MBR(newRight)))
28    insert Rect to newLeft
29  else
30    insert Rect to newRight
31  end
32  if(Stack is empty)
33    newRoot = create new root node
34    insert entry with MBR(newLeft) and newLeft to newRoot
35    insert entry with MBR(newRight)
        and newRight to newRoot
36    set version flag to Root
37    add Root to garbage collector
38    Root = newRoot
39  else
40    Parent = findLatestNode(pop(Stack), Node)
41    if(Parent is full)
42      Retry = splitInternalNode(Stack,
        Parent, Node, newLeft, newRight!)
43      if(Retry is not true)
44        set version flag to Parent
45        add Parent to garbage collector
46      else
47        delete newLeft
48        delete newRight
49        return true
50      end
51    else
52      updateEntry2(Parent, Node, newLeft, newRight)
53      if(MBR(Parent) changed)
54        updateParent(Stack, Parent MBR(Parent))
55      end
56    end
57    Left.leftLink = newLeft
58    Right.rightLink = newRight
59    Node.rightLink = newLeft
60    unlock(Parent)
61    unlock(Right)
62    unlock(Left)
63  end
64  return false
65 end
```

(그림 11) 삽입 알고리즘(splitLeafNode)

(그림 11)에서 splitLeafNode함수는 Node를 분할하고, Rect를 삽입하는 함수이다. splitLeafNode는 Node의 왼쪽 노드와 오른쪽 노드에 대해서 각각 락을 획득한 후 시작한다. 노드의 락을 획득하는 순서는 항상 왼쪽에서 오른쪽 방향으로 획득한다. 만약 그렇지 않는다면 락을 획득하는 동안 다른 연산과의 교착상태(deadlock)가 발생될 수 있다. 따라서 라인 4에서처럼 Node의 락을 풀고, 왼쪽에서부터 오른쪽 방향으로 락을 획득한다. 락을 잡은 후에 해당 노드

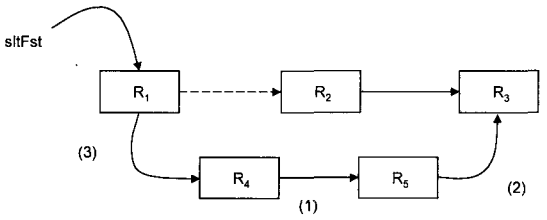
가 버전되어 있는지 확인해야하며, 버전이 되어 있다면 삽입과정을 다시 시작한다. 이는 Node의 락을 풀고 Node와 연결된 왼쪽노드를 찾은 다음 다시 Node의 락을 잡았을 경우, Node가 버전이 되어 있다면 다른 삽입연산에 의해서 Node가 분할된 경우이다. 따라서 해당 Node를 분할하여서는 안되는 경우이기 때문에 다시 삽입연산을 수행한다. 새로운 노드 두 개를 생성하고 서로의 링크를 연결하고, 두 개의 노드 중 가장 적합한 노드에 Rect를 삽입한다. 이제 삽입연산에서 남은 것은 부모노드에 새로 생긴 노드의 포인터와 MBR을 삽입하는 것이다. 만약 부모노드에 빈 엔트리가 없을 경우 부모노드를 splitInternalNode를 이용하여 분할하고, 그렇지 않은 경우는 부모 노드에 엔트리를 삽입하고, 만약 부모노드의 MBR이 변경된다면 updateParent를 호출함으로써 부모노드를 가리키는 상위 노드의 엔트리 MBR을 변경시켜 주어야한다. 32라인에서 스택이 비어 있다는 것은 Node가 루트노드임을 나타내기 때문에 새로운 루트노드를 위한 노드를 생성한 다음, 루트노드 포인터를 새로운 루트노드로 바꾸어주고 기존 루트노드는 버전되었다고 표시한다. 스택이 비어있지 않다면 부모노드에 엔트리를 삽입해야하는 경우이므로 스택에서 노드를 하나 팝하고, 팝한 노드의 최신 노드를 Parent에 저장한다. 찾은 노드에 빈 엔트리가 있으면 삽입하고, 없다면 해당 노드를 분할한다. 빈 엔트리가 있는 경우 updateEntry2함수를 호출한다. 마지막으로 57~59라인처럼 분할되기 전의 노드가 가리키는 왼쪽(Left)과 오른쪽(Right) 노드를 연결하고, 분할된 노드(Node)의 오른쪽 링크(Node.right)를 새로 생성한 왼쪽 노드(newLeft)에 연결함으로써 findLatestNode시에 노드의 최신 버전을 찾을 수 있도록 한다. (그림 12)에서의 splitInternalNode는 splitLeafNode와 유사하며, 노드에 엔트리를 삽입하는 부분만 splitLeafNode와 다르다. splitLeafNode는 삽입할 적당한 노드를 찾아서 엔트리를 삽입하지만 splitInternalNode는 새로 생성된 노드들 중에 분할된 노드가 들어있는 노드를 찾고, 찾은 노드에 두 개의 자식노드를 삽입하는 연산을 한다.

```
1     boolean splitInternalNode(STACK Stack, NODE Node,
                  NODE childOld, NODE childRight, NODE childLeft)
2~26            . . . (equal to 2~26 lines of splitLeafNode)
27    if(found entry for childOld in newLeft)
28      updateEntry2(newLeft, childOld, childLeft, childRight)
29    else
30      updateEntry2(newRight, childOld, childLeft, childRight)
31    end
32~64           . . . (equal to 32~64 lines of splitLeafNode)
65    end
```

(그림 12) 삽입 알고리즘(splitInternalNode)

기존의 분할기법에서는 새로운 하나의 엔트리만을 삽입하는 과정으로 updateEntry2를 완료할 수 있다. 그러나 본 논문의 분할기법에서는 새로운 노드가 두 개 만들어지기 때문에 분할하기전의 노드를 가리키고 있던 엔트리를 새로운 노드를 가리키는 엔트리로 변경함과 동시에 새로운 엔트리를 삽입하는 연산이 단위 연산으로 수행되어야 한다. 본 논문에서는 이러한 연산을 두개의 새로운 엔트리의 삽입과 기존의 엔트리의 삭제연산으로 처리한다. (그림 13)은 그러한 과정을 설명한다.

삽입해야할 엔트리가 R_4와 R_5이고, 삭제해야하는 엔트리가 R_2라고 한다면 R_4와 R_5, R_3의 링크를 먼저 연결하고 다음으로 R_1과 R_4를 연결함으로써 검색연산이 어떠한 문제없이 검색할 수 있도록 보장한다.



(그림 13) updateEntry2의 엔트리 삽입

3.2.4 CCV의 삭제 알고리즘

삭제연산은 두 개의 스택을 이용한다. sStack은 삭제할 엔트리를 찾기 위한 스택이고, dStack은 노드에 엔트리를 삭제하고 난 후 노드의 MBR이 변경된 경우에 부모노드에 반영시켜주기 위해서 사용된다. 삭제연산은 삭제할 엔트리가 있는 단말노드를 찾는 다음 해당 엔트리를 삭제한다. 엔트리를 삭제한 노드의 MBR이 축소된다면 부모노드의 엔트리 MBR을 변경하는 연산을 한다. 만약 노드의 엔트리가 모두 삭제된 경우라면 노드를 삭제하고, 삭제된 노드를 가리키는 부모노드의 엔트리를 삭제한다. 그러나 노드의 삭제는 검색연산이 삭제할 노드를 검색하고 있을지도 모르기 때문에 곧바로 노드를 삭제해서는 안 된다. 삭제연산으로 인해 생기는 노드의 삭제 및 노드의 분할로 인해 생기는 노드의 삭제에 대해서는 3.3절에서 자세히 논하기로 한다. (그림 14)는 삭제 알고리즘에 대해서 설명한다.

삭제연산은 삭제할 객체(Object)와 객체의 MBR(Rect)을 이용하여 삭제할 엔트리를 찾는다. 2~25라인까지는 삭제할 엔트리가 있는 단말노드를 찾는 연산이다. 삭제연산에서의 단말노드를 찾기 위해 스택을 사용하는 방법은 검색연산에서 검색하고자 하는 엔트리들을 찾기 위해 스택을 사용하는 방법과 삽입연산에서의 삽입하고자 하는 단말노드를 찾기 위해 스택을 사용하는 방법을 통합한 방법과 유사하다. 이는 R-tree구조 상 노드들의 MBR이 겹쳐 있는 경우가 발생할 수 있기 때문에 여러 단말 노드를 탐색해야만 삭제한 엔트리를 찾을 수가 있다. 따라서 본 논문에서는 두 개의 스택을 이용하여 삭제할 엔트리를 찾는다. sStack를 팝할 때마다 dStack에 푸쉬하는 방법을 사용한다. 그러나 dStack은 단말노드까지의 경로를 저장하기 위한 스택이기 때문에 같은 레벨의 노드가 dStack내에 여러 개 존재해서는 안 된다. 따라서 dStack에 푸쉬할 때는 dStack에 들어있는 노드들의

레벨을 검사해야만 한다. 노드의 레벨은 색인의 깊이가 증가할 때 마다 하나씩 증가한다. 즉 루트가 가장 작은 값을 갖고, 단말노드들이 가장 큰 값을 갖는다. 만약 dStack 안에 sStack에서 팝한 노드의 레벨보다 높은 레벨이 있다면 낮은 레벨이 나올 때까지 dStack을 팝하여 이러한 문제를 해결한다. 11라인에서 sStack을 팝한 노드가 단말노드라면, dStack에서 노드를 팝하여 단말노드로의 경로에서 해당 노드를 제거하고, 삭제할 엔트리가 해당 노드에 있는 지 확인한다. 만약 삭제할 엔트리를 찾지 못한 경우에는 sStack을 이용하여 다음 노드를 탐색하게 된다. 결국 삭제할 단말노드를 찾게 된다면 dStack에는 단말노드를 제외한 단말노드까지의 경로가 남아있게 된다.

26라인에서 엔트리를 삭제하고 나서 노드에 엔트리가 하나도 없는 경우는 노드를 삭제하여야하며, 또한 부모노드의 삭제할 노드를 가리키는 엔트리를 삭제하여야한다. 본 논문에서는 28라인에서처럼 deleteParent함수를 이용하여 부모노드의 엔트리를 삭제한다. 엔트리가 하나라도 남아있는 경우는 부모노드의 MBR만을 변경시켜준다. deleteParent에서 삭제할 엔트리를 가진 부모노드를 찾는 과정은 updateParent에서 변경할 엔트리를 갖는 부모노드를 찾는 방법과 동일하다. 엔트리를 찾게 되면 엔트리를 삭제하고, 41라인에서처럼 엔트리를 삭제하고, 동일한 방법으로 부모노드에 적용하게 된다.

```
1 delete(STACK sStack, STACK dStack,
        OBJECT Object, RECT Rect)
2   push(sStack, Root)
3   while(sStack is not empty)
4     Node = pop(sStack)
5     if(dStack is not empty)
6       while(LEVEL(Node) <= LEVEL(top(dStack)))
7         pop(dStack)
8       end
9     end
10    push(dStack, Node)
11    if (Node is leaf node)
12      lock(Node)
13      pop(dStack)
14      Entry = find entry for Object in Node
15      if(Entry is not null)
16        break
17      end
18      unlock(Node)
19      Node = pop(sStack)
20    else
21      for all entries Entry of Node intersecting Rect
22      push(sStack, NODE(Entry))
23    end
24  end
25 Node = findLatestNode(Node, pointer to Object)
26   delete Entry in Node
27   if(Node is empty)
28     deleteParent(dStack, NODE(Entry))
29     add Node to garbage collector
30   else
31     if(MBR(Node) was changed)
32       updateParent(dStack, Node, MBR(Node))
33     end
34   end
35 end
36
37 deleteParent(STACK Stack, NODE Node)
38   if(Stack is not empty)
39     Parent = pop(Stack)
40     Parent = findLatestNode(Parent, pointer to Node)
41     delete Entry in Parent
42     if(Parent is empty)
43       deleteParent(Stack, NODE(Entry))
44       add Parent to garbage collector
45     else
46       if(MBR(Parent) was changed)
47         updateParent(Stack, Parent, MBR(Parent))
48       end
49     end
50     unlock(Parent)
51   end
52 end
```

(그림 14) 삭제 알고리즘

3.3 버전 삭제

본 절에서 논의할 내용은 분할시 생기는 구버전의 노드와 삭제연산으로 인해 생기는 노드들을 언제, 어떻게 삭제해야 하느냐에 대해서 논의한다. 노드의 삭제는 분할이나, 삭제연산이 종료한 후 바로 노드를 삭제할 수 없다. 왜냐하면 분할되어진 구버전의 노드나 삭제연산으로 인해 생기는 노드는 분할 또는 삭제연산 당시 다른 검색연산에 의해서 검색되고 있었을 수도 있기 때문에 노드의 삭제는 검색연산이 해당 노드를 사용 안한다고 확신할 때만 삭제를 할 수 있다.

본 논문에서는 이러한 문제를 해결하기 위해 노드삭제계수(**NDSN** : **N**ode **D**elete **S**equence **N**umber)를 이용한다. NDSN은 전역노드삭제계수(**G-NDSN** : **G**lobal **NDSN**)와 지역노드삭제계수(**L-NDSN** : **L**ocal **NDSN**)로 나뉜다. G-NDSN은 색인마다 하나씩 유지되며, 삭제될 노드가 추가될 때 마다 하나씩 증가한다. L-NDSN은 커서를 열고 있는 트랜잭션마다 하나씩 유지되며, 커서를 열 때 G-NDSN을 복사해놓은 값이다.

노드 삭제는 쓰레기 수집기(**GC** : **G**arbage **C**ollector)에 의해서 처리된다. GC는 쓰레기 수집을 위하여 우선순위 큐(priority queue)를 이용하며, GC에 삭제될 노드가 들어오면 G-NDSN값을 하나 증가 시키고, 증가 시킨 G-NDSN과 함께 우선순위 큐에 삽입된다. GC는 주기적으로 활성화되고, 활성화 될 때 마다 현재 커서를 열고 진행하고 있는 트랜잭션들의 최소 NDSN값을 가지고, 큐에 들어 있는 노드들에 대해서 삭제하는 연산을 수행하게 된다. 만약 큐에 삽입된 노드가 갖는 NDSN과 현재 GC가 가지고 있는 최소 NDSN과 비교하여 최소 NDSN이 크다면 삭제된 노드가 GC에 삽입했을 당시 진행 중이던 검색연산들이 종료했다

는 것을 의미하기 때문에 노드는 삭제되어질 수 있다. 그러나 최소 NDSN이 작다면 그 당시 수행하던 검색연산이 아직도 활성화되고 있다는 것을 의미하기 때문에 해당 노드를 삭제하여서는 안 된다. 본 논문에서 제안하는 버전 삭제 방법은 3.1.2절에서 제안한 EDSN을 사용하는 방법과 유사하다. 하지만 노드의 삭제는 커서단위로 이루어지는 반면, 엔트리의 삭제는 커서내의 작은 연산을 한 단위로 처리되기 때문에 버전 삭제보다 빠른 시간 내에 엔트리의 재사용이 이루어지는 특징을 갖는다.

4. 트랜잭션의 일관성

트랜잭션의 독립성 레벨(Isolation level)중 재판독(**RR** : **R**epeatable **R**ead), 유령판독(**PR** : **P**hantom **R**ead)을 방지하기 위해서는 [4]와 같은 기법이 적용되어질 수 있다. [4]는 질의(query)의 서술자(predicate)에 대해서 락을 걸고 트랜잭션이 종료하기 전까지 락을 풀지 않음으로써 RR이나 PR을 방지하는 기법이다. 그러나 제안된 기법에 [4]와 같은 기법을 적용한다면 검색연산이 RR이나 PR을 방지하기 위하여 서술자에 락을 걸어야 하기 때문에 검색연산은 갱신연산으로 인하여 블록킹될 수가 있다. 따라서 제안된 기법의 검색연산의 락이나 래치를 피하기 위해서 Dali[6]과 같은 멀티버전(multi-version) 시스템 위에 제안된 색인기법을 적용한다.

Dali는 하나의 레코드(record)에 대해서 다수의 버전을 만들고 관리하며, 트랜잭션들은 자신의 타임스템프(timestamp)를 이용하여 다수의 버전 중 읽을 수 있는 안정된 버전을 읽어감으로써 검색연산의 효율을 높이기 위한 시스템 구조를 갖는다.

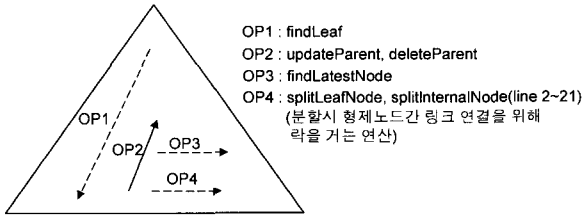
본 기법을 멀티버전 시스템에서 이용하는 방법은 색인으로부터 후보객체(candidate object)를 얻어오고, 얻어진 후보객체의 타임스탬프와 트랜잭션의 타임스탬프와 비교하여 후보객체를 자신이 읽을 수 있는 객체인지를 판단한다. 만약 읽을 수 있는 객체라면 정제연산(refinement)연산을 수행하여 MBR이 아닌 실 객체(real object)와 연산을 하게 된다. 그렇지 않다면 다시 색인으로부터 후보객체를 얻어오고 위의 과정을 반복한다. 이러한 방식은 검색연산이 색인뿐만 아니라 객체에 대해서도 전혀 락이나 래치를 걸지 않고 연산을 수행하기 때문에 검색연산의 성능을 향상 시킬 수가 있다.

5. 알고리즘 증명

본 장에서는 제안하는 기법에서 교착상태(deadlock)가 발생하지 않음을 보이고, 각 연산이 올바르게 수행됨을 증명한다.

5.1 교착상태

교착상태는 락을 거는 연산들이 노드 간 서로 교차 수행됨으로써 사이클(cycle)을 형성하기 때문에 발생한다. 본 절에서는 연산 간 사이클이 발생하지 않음을 보인다. 본 기법에서 사용되어지고 있는 연산들 중 여러 노드에 걸쳐 영향을 미치는 연산들과 이러한 연산들의 방향은 (그림 15)와 같다.



(그림 15) 여러 노드에 영향을 미치는 연산들의 연산방향

(그림 15)의 점선으로 표시되어진 화살표는 다음 노드의 락을 잡기위해 현재 노드의 락을 푼다는 의미이다. 즉 시간 T에서 락을 잡고 있는 노드의 개수는 오직하나이다. 실선으로 표시된 화살표는 현재 락을 잡고 있는 노드를 풀지 않고, 다음 노드의 락을 획득함으로써 시간 T에서 서로 다른 레벨의 노드들에 락을 잡고 있을 수 있다. OP1과 OP2는 서로 교차되어 수행되지만 교착상태가 발생하지 않으며, 오직 실선과 같은 연산들이 교차수행 되어야만 교착상태가 발생할 수 있다. 본 논문에서는 이러한 실선형태의 연산들은 단말에서 루트방향 (OP2)으로 또는 왼쪽에서 오른쪽 방향(OP4)으로만 락을 잡기 때문에 연산들 간 사이클이 형성되지 않는다. 따라서 본 논문에서는 연산들 간 교착상태가 발생될 수 없다.

5.2 알고리즘의 정확성

본 절에서는 색인의 구조를 변경하는 연산(**SMO** : **S**tructure **M**odification **O**peration)으로 인하여 색인의 정확성(correctness)이 위배되지 않음을 증명하고, SMO와 검색연산이 동시에 수행되었을 경우 검색연산이 정확히 색인을 탐색할 수 있다는 것을 증명한다. 삭제연산은 검색연산과 삽입연산의 조합으로 생각할 수 있기 때문에 본 절에서는 언급하지 않는다.

[증명 1] ***SMO****로 인하여 색인의 정확성(correctness)이 위배되지 않는다.*

[증명 1]을 위하여 시간 T에서 삽입연산 I가 노드 A를 분할된다고 가정하고 새로운 노드 B와 C를 생성한다고 가정하자. [증명 1]은 아래의 두 가지 경우를 만족한다면 [증명 1]을 만족할 수 있다.

① 다른 삽입연산이 SMO가 진행중인 노드 A에 연산을 수행하지 못한다.
② 다른 삽입연산이 새로운 노드 B와 C에 연산을 수행하지 못한다.

①의 경우는 시간 T에서 A는 락이 걸려있기 때문에 A에 삽입하려는 연산은 I가 분할이 완료되어 A의 락을 푸는 시점에서만 가능하다. 따라서 분할이 진행 중인 노드 A에 어떠한 다른 삽입연산도 진행할 수 없다.

②의 경우는 B와 C에 대한 포인터가 부모노드에 반영되어 있지 않기 때문에 B와 C에 다른 삽입연산이 수행될 수 없음을 보장할 수 있다.

따라서 본 색인기법은 색인의 정확성을 위배하지 않음을 증명한다.

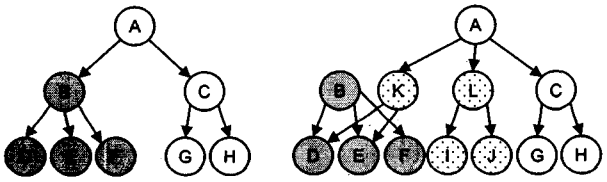
[증명 2] *SMO와 검색연산이 동시에 수행되었을 경우 검색연산이 정확히 색인을 탐색할 수 있다.*

[증명 2]는 아래의 두 가지 경우를 만족한다면 [증명 2]를 만족할 수 있다.

① SMO가 진행 중인 노드를 탐색할 경우에 정확히 색인을 탐색할 수 있다.
② 검색연산이 검색하기 위하여 지나온 경로상의 노드(나중에 탐색하기 위하여 스택에 푸쉬되어 있는 노드)에 SMO를 하더라도 검색연산은 색인을 정확히 탐색할 수 있다.

①의 경우는 삽입연산이 노드 A를 분할하는 중에 검색연산 S가 검색한다고 가정하자. 본 논문의 분할은 노드 A에 어떠한 변경도 하지 않기 때문에 S가 A를 아무런 문제없이 탐색할 수 있다는 것을 보장한다.

②의 경우는 검색연산 S가 검색하기 위하여 노드 A를 시간 T_1 시점에 스택에 쌓아두고, 삽입연산 I가 시간 T_2 시점에서 노드 A를 분할한다고 가정하자 ($T_1 < T_2$). 이러한 경우 또한 I가 노드 A를 분할하기 위하여 새로운 노드들을 생성하기 때문에 A는 그대로 유지된다고 보장할 수 있다. 따라서 T_2 시점에서 A의 서브트리(subtree)는 T_1 시점의 서브트리와 같음을 보장할 수 있다. (그림 16)은 ② 경우에 대한 설명이다.



(a) 분할 전 (b) 분할 후

(그림 16) 색인의 분할 전과 분할 후

검색연산이 나중에 노드 B를 검색하기 위하여 스택에 푸쉬하고 나서 노드 H를 탐색하고 있다고 할 때, 삽입연산이 노드 F의 분할로 인하여 B를 분할한다고 가정하자. 그림에서 보는 바와 같이 분할 후 B의 서브트리는 분할 전의 형태와 같은 형태를 유지한다.

따라서 ①과 ②를 만족하기 때문에 검색연산이 정확히 색인을 탐색할 수 있다는 것을 증명한다.

6. 성능 평가

본 장에서는 R^{link}-Tree와의 성능 비교를 통하여 본 기법의 우수성을 입증한다. 실험환경을 설명하고, 검색과 삽입연산을 하는 프로세스(process)를 증가 시키면서 검색연산의 응답시간과 응답시간들의 편차를 측정한다.

6.1 실험 환경

실험에서 사용될 시스템 사양은 700MHz의 Xeon CPU 2개, 주기억장치 1G, 운영체제로는 windows NT server를 사용한다.

노드의 사이즈는 적은 데이터 집합을 이용하여 색인의 깊이(depth)를 증가시키기 위하여 1K로 하며, 각 노드는 메인 메모리에 상주해 있다고 가정한다. 한 노드에 관리할 수 있는 엔트리의 개수는 R^{link}-Tree가 23개이고, 제안하는 기법이 19개이다.

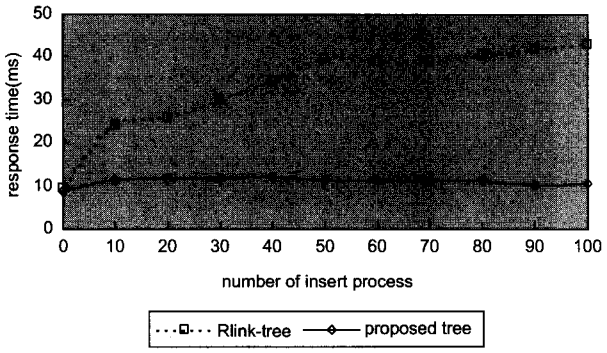
실험 평가에서 GC는 1초마다 활성화되어 쓰레기 노드를 삭제한다. R^{link}-Tree와 제안된 기법의 회복에 대한 부분은 고려하지 않으며, 또한 삭제 알고리즘에 대해서도 고려하지 않는다. 실험에 사용할 기법들의 노드 분할시 사용하는 알고리즘은 사분할 알고리즘(quadratic split algorithm)을 사용한다.

실험하기 전 초기 데이터로는 20000×20000의 정해진 영역 내에서 균등 분포를 갖는 10×10 크기의 10000개 엔트리를 색인에 삽입하고 나서 실험을 시작한다. 각 삽입 프로세스는 전체 영역에서 정 중앙의 2000×2000내에서 무작위(random)로 추출된 10×10의 크기를 갖는 MBR을 색인에 삽입한다. 또한 각 검색 프로세스들은 정 중앙 2000× 2000의 질의 범위를 검색한다. 첫 번째 실험에서는 삽입 프로세스의 수를 1부터 100까지 10개씩 증가시키면서 검색연산을 20회 반복하여 R^{link}-Tree와 제안된 기법과의 검색연산의 평균 응답시간과 응답시간들의 편차를 측정한다. 다음 실험에서는 삽입 프로세스가 1개인 경우(**LC** : **L**ow-**C**ontention)와 50개인 경우(**HC** : **H**igh-**C**ontention)로 나누어 검색 프로세스의 수를 1부터 100까지 10개씩 증가 시키면서 검색연산의 처리속도를 측정한다.

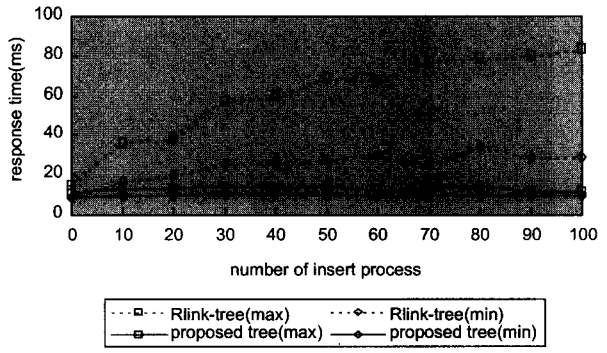
6.2 실험 결과

(그림 15)는 삽입 프로세스 수에 따른 검색연산의 평균 응답시간과 응답시간의 편차를 평가한 결과이다.

(그림 17)(a)는 삽입 프로세스 수에 따른 검색연산의 평균 응답시간을 측정한 결과로 제안된 기법이 R^{link}-Tree보다 2배에서 4배 빠른 성능을 보이고 있다. 삽입연산이 없는 상황에서의 검색연산은 R^{link}-Tree의 노드가 제안된 기법보다 많은 수의 엔트리를 갖기 때문에 제안된 기법보다 조금 빠른 성능을 나타내었다. 반면 삽입 프로세스 수가 많아짐

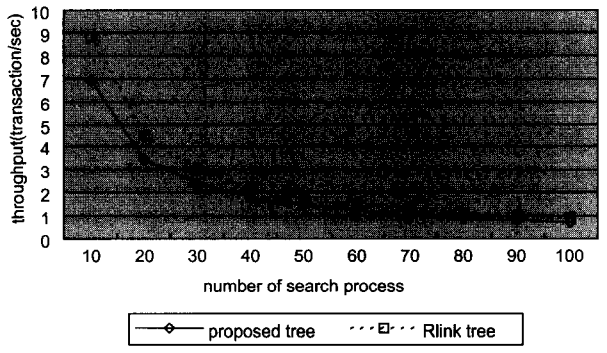


(a) average of response time

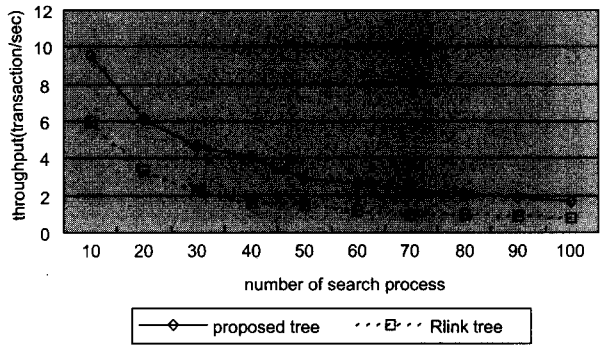


(b) deviation of response time

(그림 17) 삽입 프로세스 수에 따른 검색연산의 평균 응답시간과 응답시간의 편차



(a) low-contention



(b) high-contention

(그림 18) 검색 프로세스 수에 따른 검색연산의 평균 처리속도

에 따라 R^{link}-Tree보다 제안된 기법이 아주 빠른 응답시간을 보장하였다. 이는 R^{link}-Tree의 검색연산이 삽입연산으로 인하여 블록킹 상태가 빈번히 발생되었기 때문이고, 제안된 기법은 검색연산이 락이나 래치를 사용하지 않기 때문에 삽입연산으로 인한 검색연산의 블록킹 상태가 발생하지 않았기 때문이다. (그림 17)(b)는 삽입 프로세스 수에 따른 검색연산의 응답시간 편차이다. 응답시간이 가장 작게 나온 경우(min)과 가장 크게 나온 경우(max)를 가지고 응답시간들의 편차를 나타내었다. R^{link}-Tree는 검색연산이 발생할 때마다 매회 응답시간이 다르게 측정되는 것과 달리 제안된 기법은 매회 비슷한 응답시간이 측정되었다.

(그림 18)은 LC와 HC 상황에서의 검색연산의 평균 처리 속도를 측정하였다.

(그림 18)(a)는 R^{link}-Tree가 제안된 기법에 비해 초당 처리할 수 있는 트랜잭션 양이 약 27% 많음을 볼 수 있다. 이는 R^{link}-Tree 노드의 팬 아웃이 제안된 기법의 팬 아웃보다 크기 때문에 노드를 탐색하는 수가 제안된 기법이 더 많아 졌기 때문이다. 하지만 (그림 18)(b)와 같은 충돌이 잦은 상황에서는 노드의 팬 아웃보다는 충돌 회수가 트랜잭션의 성능에 많은 영향을 미치기 때문에 제안된 기법이 R^{link}-Tree보다 약 100% 많은 처리량을 나타내었다.

6. 결 론

본 논문에서는 다차원 색인이 갱신연산보다는 검색연산의 빈도가 높다는 특성을 고려하여, 검색연산이 어떠한 락이나 래치를 걸지 않고 색인을 탐색할 수 있는 동시성 제어 기법을 제안하였다. 제안된 기법은 검색연산이 노드를 탐색하는 중에도 갱신연산이 일어날 수 있도록 노드내의 엔트리들을 링크드 리스트로 연결하였으며, 노드 분할을 수행할 경우 버전기반의 분할 기법을 사용함으로써 검색연산이 분할되는 노드에 락이나 래치를 획득하지 않고 노드를 탐색할 수 있는 방법을 제안하였다. 본 논문에서는 제안하는 기법의 우수성을 성능 평가를 통하여 입증하였으며, 기존의 기법에 비해서 검색연산의 성능이 2배에서 4배 빠른 성능을 보여주고 있다. 향후 연구로는 제안하는 동시성 제어 기법을 위한 회복 기법에 대해서 연구를 진행할 것이다.

참 고 문 헌

[1] R. Bayer and E. McCreight, Organization and Maintenance of Large Ordered Indexes, Acta Informatica, Vol.1, No.3, pp.173-189, 1972.

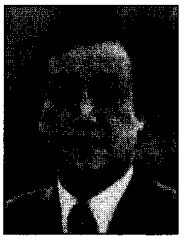
[2] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, The R*tree : An Efficient and robust access method for points and rectangles, Proc. of the ACM SIGMOD Intl. Conf. on Management of Data, pp.322-331, 1990.

[3] S. Berchtold, D. A. Keim and H. P. Kereigel, The X-tree

: An index structure for high dimensional data, Proceedings of the Int. Conf. on Very Large Data Bases, 1996.
[4] K. Eswaren, J. Gray, R. Lorie and I. Traiger, On the Notions of Consistency and Predicate Locks in a Database System, Comm. ACM, Vol.19, No.11, pp.624-633, November, 1976.
[5] A. Guttman, R-trees : A dynamic index structure for spatial searching, Proc. ACM SIGMOD Int. Conf. on Management of Data, pp.47-57, 1984.
[6] H. V. Jagadish, Dan Lieuwen, Rajeev Rastogi, Avi Silberschatz and S. Sudarshan, Dali : A high performance main-memory storage manager, In Proc. of the Int. Conf. on Very Large DataBases, 1994.
[7] M. Kornacker and D. Banks, High-Concurrency Locking in R-Trees, Proceedings of the Int. Conf. on Very Large Data Bases, pp.134-145, September, 1995.
[8] M. Kornacker, C. Mohan, and J. Hellerstein, Concurrency control and recovery in GiST, Proc. ACM SIGMOD Int. Conf. on Management of Data, 1997.
[9] T. J. Lehman and M. J. Carey, A study of index structures for main memory database management systems, In Proc. of the Conf. on Very Large Data Bases, pp.294-303, August, 1986.
[10] P. Lehman and S. Yao, Efficient Locking for Concurrent Operations on B-Trees, ACM TODS, Vol.6, No.4, December, 1981.
[11] H. V. Lin, K. Jagadish and C. Faloutsos, The TV-tree : an index structure for high dimensional data, In VLDB Journal, 1994.
[12] D. Lomet and B. Salzberg, The hB-Tree : A Multiattribute Indexing Method with Good Guaranteed Performance, ACM TODS, Vol.15, No.4, pp.625-685, December, 1990.
[13] J. Nievergelt, H. Hinterberger and K. C. Sevcik, The Grid File : An Adaptable, Symmetric Multikey File Structure, ACM TODS, Vol.9, No.1, March, 1984.
[14] R. Rastogi, S. Seshadri, P. Bohannon, D. Leinbaugh, A. Silberschatz and S. Sudarshan, Logical and Physical Versioning in Main Memory Databases, In Proc of the Int. Conf. on Very Large Data Bases, August, 1997.
[15] K. V. Ravi Kanth, Divyakant Agrawal and Ambuj K Singh, Improved concurrency control techniques for multi-dimensional index structures, Technical Report, Univ. of California at santa Barbara, 1998.
[16] J. T. Robinson, The K-D-B-Tree : A Search Structure for Large Multidimensional Dynamic Indexes, Proc. ACM SIGMOD Conf., pp.10-18, 1981.
[17] D. White and R. Jain, Similarity indexing with the SS-tree, Proc. Int. Conf. on Data Engineering, pp.516-523, 1996.
[18] T. Sellis, N. Roussopoulos, C. Faloutsos, The R+-Tree : A Dynamic Index for Multi-Dimensional Objects, In VLDB Journal, 1987.
[19] V. Srinivasan and M. Carey. Performance of b-tree concurrency control algorithms, Proc. ACM SIGMOD Int. Conf. on Management of Data, pp.416-425, April, 1995.

김 명 근

e-mail : kimmkeun@dblab.inha.ac.kr
1999년 인하대학교 전자계산공학과(공학사)
1999년~현재 인하대학교 대학원 전자계산공학과 박사과정
관심분야 : 공간 데이터베이스 관리 시스템, 공간 데이터베이스 클러스터



배 해 영

e-mail : hybae@inha.ac.kr
1974년 인하대학교 응용물리학과(공학사)
1978년 연세대학교 대학원 전자계산학과 (공학석사)
1989년 숭실대학교 대학원 전자계산학과 (공학박사)
1985년 Univ. of Houston 객원 교수
1992년~1994년 인하대학교 전자계산소 소장
1982년~현재 인하대학교 전자계산공학과 교수
1999년~현재 지능형 GIS 연구센터 소장
2000년~현재 중국 중경우전대학교 대학원 명예 교수
2004년~현재 인하대학교 정보통신대학원 원장
관심분야 : 분산 데이터베이스, 공간 데이터베이스, 지리정보 시스템, 멀티미디어 데이터베이스 등