고차원 색인 구조를 위한 동시성 제어 기법의 설계 및 구현

(Design and Implementation of a Concurrency Control Algorithm for High-Dimensional Index Structures)

송 석 일 † 박 춘 서 † 이 석 희 † 유 재 수 ††
(Seok Il Song) (Choon Seo Park) (Seok Hee Lee) (Jae Soo Yoo)

요 약 이 논문에서는 고차원 색인 구조를 위한 동시성 제어 기법을 설계하고 이를 구현한다. 일반적으로 고차원 색인구조에서는 삽입보다 탐색연산이 빈번하고 탐색연산의 수행은 질의의 특성상 매우 많은 노드를 접근한다. 제안하는 동시성 제어 알고리즘에서는 이런 특성을 고려하여 탐색 연산의 지연이 최소가 되도록 한다. 또한 인덱스의 성능향상을 위해 재삽입 연산을 이용하는 고차원색인 구조를 고려하여 재삽입 연산 수행중에도 정확한 탐색을 보장할 수 있는 방법을 지원한다. 제안하는 동시성 제어 알고리즘을 CIR-Tree에 적용하여 실제 상용 DBMS의 하부 저장 시스템인 MiDAS-Ⅲ에서 구현한다. 실험을 통하여 제안된 동시성 제어기법이 기존 동시성 제어 기법보다 성능이 우수함을 보인다.

Abstract In this paper, we design and implement a concurrency control algorithm based on Link Technique for high-dimensional index structures. In the high dimensional index structures search operations are generally more frequent than insert or delete operations and search operations need to access much more nodes than other index structures such as B-Tree due to the properties of queries. In the proposed algorithm, we focus on minimizing the delay of search operations at any cases. It also supports the concurrency control on reinsert operations for the high dimensional index structures employing reinsert operations to improve their performance. We apply the algorithm to the CIR-Tree and implement it on MiDAS-Ⅲ that is the storage system of a multimedia DBMS, called BADA-Ⅲ. It is shown through experiments that our proposed method outperforms the existing one.

1. 서 론

지난 수년 동안 다차원의 특징벡터를 기반으로 하는 유사성 검색이 데이타 베이스 분야에서 매우 중요한 연구 과제로 부각되어 왔다. 이의 응용분야는 지리정보 시스템부터 의료 데이타 저장 시스템, 멀티미디어 데이타 베이스 시스템에 이르기까지 매우 폭 넓게 분포되어 있다. 이런 유사성 검색의 가장 핵심적인 문제 중 하나가 바로 수많은 다차원의 특징 벡터들 중에서 어떻게 질의와 유사한 것들을 효율적으로 찾아내는가 하는 것이다. 이 문제를 해결하기 위해 다차원 색인구조에 대한 연구가 매우 활발히 진행되어 왔고 그 결과 Grid File[11], R-Tree[12], R*-Tree[13], TV-Tree[14], X-Tree[15], SS-Tree[16], SR-Tree[17], CIR-Tree[2], Hybrid-Tree[18]등의 수많은 색인 구조들이 제안되었다.

기존의 다차원 색인 구조들이 실제 응용에 사용되기 위해서는 여러 사용자가 삽입 및 탐색을 동시에 수행할 수 있는 다중 사용자 환경이 고려되어야 한다. 즉, 색인 구조에 대한 적절한 동시성 제어 알고리즘과 회복 방법이 있어야 한다는 것이다. 기존에 제안된 다차원 색인 구조에 대한 동시성 제어 알고리즘으로는 참고문헌 [1], [5], [6], [9] 정도이다. 하지만 이들 기존의 방법들은 다음에 제시하는 유사성 검색 시스템의 일반

† 비 회 원 : 충북대학교 정보통신공학과
prince@pretty.chungbuk.ac.kr
seoklee@pretty.chungbuk.ac.kr
parkcs@pretty.chungbuk.ac.kr
†† 종신회원 : 충북대학교 정보통신공학과 교수
yjs@cbucc.chungbuk.ac.kr
논문접수 : 1999년 11월 17일
심사완료 : 2000년 9월 27일

적인 특징에 비추어 볼 때 다차원 색인구조를 위한 적절한 동시성 제어 알고리즘이라고 볼 수 없다. 유사성 검색 시스템은 보통 다음과 같은 특징을 갖는다.

가. 일반적으로 다차원 색인 구조는 다른 색인 구조(B-트리 계열)들에 비해 탐색 연산 수행 시 노드를 접근하는 회수가 훨씬 많다. 유사성 검색에서는 범위 질의나 K-최근접 질의와 같은 독특한 형태의 질의가 존재하며 다차원 색인 구조의 특성상 하나의 비단말 노드의 엔트리가 표현하는 벡터 공간과 다른 엔트리가 표현하는 벡터공간이 겹칠 수 있다.

나. 유사성 검색 시스템에서는 데이타 베이스가 일단 구축이 되고 나면 삽입이나 삭제 연산 보다 탐색 연산의 빈도가 훨씬 많다. 색인 구조에서 변경연산은 존재하지 않는다. 변경이 발생하면 삭제와 삽입으로 대체될 것이다. 하지만 유사성 검색에서 한번 구축된 데이타(예, 이미지 데이타)는 그 특성상 좀처럼 변경되지 않는다. 또한 신규 삽입도 예약 시스템이나 은행업무 시스템에 비하면 훨씬 적다.

다. 보통 고차원을 대상으로 하는 다차원 색인 구조(R*-Tree, SS/SR-Tree, TV-Tree, CIR-Tree)에서는 재삽입 연산을 통해 색인 구조를 보다 효율적으로 구성하여 탐색 성능을 향상시키고 있다. 재삽입 연산이란 특정 노드에서 넘침(overflow)이 발생했을 때 이를 바로 분할하지 않고 그 노드의 중심점에서 멀리 떨어져 있는 엔트리들을 색인구조에서 일시적으로 삭제하고 다시 삽입하는 연산을 말한다.

이상의 특징들이 있지만 기존의 연구에서 제시하고 있는 알고리즘들에서는 이들이 잘 고려되어 있지 않다. 특히 재삽입 연산에 관해서는 참고문헌 [10]에서 여러 실험을 통해 재삽입 연산에 의한 색인의 성능향상을 입증하였음에도 불구하고 이에 대한 동시성측면의 지원을 고려하고 있는 알고리즘은 없다. 이에 이 논문에서는 탐색 연산에 중점을 두고 재삽입 연산에 대한 동시성을 고려한 동시성 제어 알고리즘을 설계한다. 그리고 이를 순수 국내기술로 개발된 멀티미디어 DBMS인 BADA의 하부저장구조 MiDAS에 CIR-Tree를 바탕으로 구현한다.

이 논문의 구성은 다음과 같다. 먼저 2장에서 기존 연구에 대한 분석과 그 문제점을 지적하고 3장에서 제안하는 동시성 제어 알고리즘에 대해 자세히 기술한다. 4장에서는 이 논문에서 제안하는 알고리즘을 구현하고 이에 대한 성능 평가를 실시 하고, 마지막으로 5장에서 결론을 맺는다.

2. 관련연구

이 장에서는 먼저 기존의 다차원 색인 구조에 대한 동시성 제어 기법들을 분석하여 그들의 문제점을 지적하고 이 논문에서 설계하는 동시성 제어 알고리즘에서 고려해야 할 사항들에 대해 기술한다.

기존의 다차원 색인구조에 대한 동시성 제어알고리즘들은 크게 잠금 결합기법(Lock-Coupling Technique)을 기반으로 하는 방법([9])과 링크기법(Link-Technique)을 기반으로 하는 방법([1], [5], [6])으로 분류할 수 있다. 잠금 결합 기반의 동시성 제어 알고리즘은 트리를 순회할 때 현재 노드의 잠금을 해제하기 전에 다음에 방문할 노드에 잠금을 획득한다. 그리고, 노드 분할을 수행하거나 MBR(Minimum Bounding Region)변경을 상위 노드에 반영할 때는 변경에 참여하는 모든 노드들에 잠금을 획득해야 한다. 이 방법은 설명한 대로 탐색이나 삽입 연산 시에 하나의 트랜잭션이 동시에 여러 노드에 잠금을 유지해야 하므로 동시성 성능의 저하를 초래하게 된다.

이에 대한 문제점을 해결하기 위해 링크기법을 기반으로 하는 동시성 제어 알고리즘이 제안되었다. 링크 기법기반의 동시성 제어 알고리즘은 잠금 결합을 수행할 필요가 없다. 즉, 트리를 순회하는 동안에 최대 하나의 노드에만 잠금을 획득하면 된다. 하지만 여전히 MBR 변경이나 분할연산을 수행할 때에는 동시에 여러 노드에 잠금을 획득해야 한다. 제안하는 알고리즘은 기본적으로 링크 기법에 바탕을 두고 있기 때문에 다음에서 이 기법에 대해 보다 자세히 설명한다.

링크 기법을 기반으로 하는 다차원 색인 구조에 대한 동시성 제어 알고리즘의 대표적인 것으로 참고문헌 [5]과 [6]를 들 수 있다. [5]에서 제안한 RLink-Tree의 동시성 제어 알고리즘은 BLink-Tree[8]에서 제안된 링크 기법을 R-Tree에 맞게 수정한 것이다. 링크 기법의 기본 개념은 각 레벨의 모든 노드들을 오른쪽 링크를 통해 연결하여 탐색 연산 시에 잠금 결합[5, 6]을 수행하지 않아도 상위 노드에 반영되지 않은 하위 노드의 분할을 감지하여 이를 보상할 수 있도록 하는 것이다. RLink-Tree에서는 R-Tree와 B-Tree의 구조적인 차이로 인해 BLink-Tree 의 링크 기법을 수정 및 보완하였다.[5] 이 과정에서 NSN(Node Sequence Number)을 도입하여 이를 각 노드에 할당하게 되었고 엔트리의 구조도 <MBR, Child_Page>에서 <MBR, Child_Page, NSN>로 수정되었다. NSN 이라는 것은 노드가 새로

생성될 때 증가하는 값으로 각 노드는 유일한 NSN을 부여받는다. 노드에 대한 NSN의 할당방법은 다음의 예에서 자세히 설명한다.

그림 1 에서 RLink-Tree에서 분할과 탐색이 동시에 수행되는 경우에 대해서 설명하고 있다. 노드가 분할되면 새로운 노드에 기존의 노드가 가지고 있던 NSN을 할당하고 기존의 노드에는 새로운 NSN을 할당한다. 이때 새로 생성되는 노드는 항상 기존노드의 오른쪽에 위치하게 된다. 따라서 탐색 연산은 트리를 순회할 때 대응하는 자식노드의 분할 여부를 이 NSN 값을 가지고 판단 할 수 있다. 즉, 부모노드의 대응하는 엔트리 <MBR, Child_Page, NSN>에 기록된 NSN을 기억하고 있는 탐색 프로세스는 그 자식노드(Child_Page)의 실제 NSN 값이 기억하고 있는 값보다 크면, 오른쪽으로 이동한다. 이동하면서 기억하고 있는 NSN을 가진 노드들을 만나면 옆으로의 이동을 멈추고 다시 하위 노드로 탐색을 진행하게 된다.

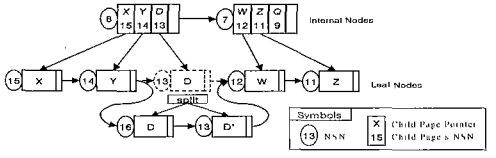


그림 1 R^{Link}-Tree에서의 분할과 탐색

이 방법은 몇 가지 단점이 있다. 하나는 어떤 트랜잭션이 노드 분할과 MBR변경 등과 같은 트리구조를 변경하는 연산을 수행할 때는 동시에 둘 이상 레벨의 노드들에 잠금을 유지해야 하며 이로 인해 탐색연산이 다음 노드로 탐색을 진행할 때 I/O 시간동안 지연 될 수 있다는 것이다. 다른 하나는 엔트리를 표현하는 자료 구조가 <MBR, Child_Page, NSN>로 바뀌게 되면서 부가 적인 정보 NSN으로 인해 저장공간 이용률이 저하되어 비단말 노드의 팬 아웃(Fan Out)이 작아 진다는 것이다.

[6]에서 제안하고 있는 동시성 제어 알고리즘에서는 [5]의 문제점으로 지적된 엔트리의 자료구조에 NSN이 추가되어 저장공간을 낭비하는 문제를 해결하고 있다. [6]에서도 역시 RLink-Tree에서와 같이 각각의 노드에 NSN을 할당하고 오른쪽 링크를 두어 분할을 검사하고 어디까지 오른쪽 링크를 따라 순회 할 것인지를 판별한다. 하지만 RLink-Tree와는 다르게 비단말 노드의 엔트리에 하위 노드의 NSN을 기록해두지 않고도 하위 노드의 분할을 판별할 수 있는 방법을 제안하고있다. 이것은 전역계수기를 이용해 NSN을 할당함으로써 가능해진다.

노드가 분할될 때 새로 생성된 노드에는 원래 노드의 NSN 값이 주어지고 원래 노드에는 새로운 NSN 값이 할당된다. 이때 트리 순회 연산은 노드를 방문할 때 전역 계수기의 값을 기억하고 다시 하위 노드를 방문 할 때 기억했던 전역 계수기 값과 현재 노드의 NSN을 비교하여 현재 노드의 NSN이 더 크면 상위 노드에 반영되지 않은 분할로 판단하고 오른쪽 링크를 따라서 이동한다. 이동하다가 기억했던 전역계수기 값보다 작은 NSN을 갖는 노드를 만나면 이웃 노드로의 이동을 멈추고 다시 하위노드로의 순회를 계속하게 된다. 이에 대한 것을 그림 2 에서 예를 들어 설명한다.

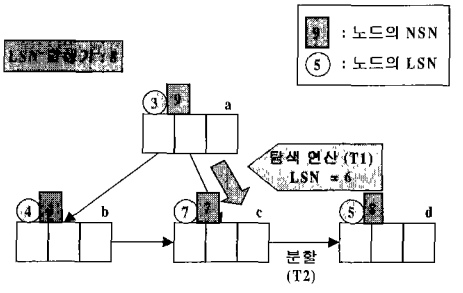


그림 2 전역계수기를 이용한 링크 기법

그림 2는 전역 계수기로써 LSN을 사용한 예이다. T1은 탐색연산을 수행하는 트랜잭션이다. 이 트랜잭션은 노드 a에서 다음 탐색을 진행할 하위 노드(c)를 결정하고 그때의 노드 a의 LSN을 기억하고 잠금을 해제한다. 바로 이어서 노드 c에 잠금을 요청한다. T2는 노드 c에 잠금을 획득하고 삽입을 수행중 넘침이 발생하여 이를 분할하기 위해 노드 a에 래치를 잠금을 T1이 잠금을 해제할 때 이를 획득한다. 이후에 분할을 수행하여 노드 d를 생성하고 노드 d의 NSN을 노드 c의 NSN으로 할당하고 노드 c의 NSN은 노드 c의 LSN으로 할당한다. 분할이 끝나면 노드 c, 노드 d 그리고 노드 a에 획득했던 잠금들을 해제한다. 동시에 T1은 노드 c에 대한 잠금을 획득하고 그 노드의 NSN과 기억했던 노드 a의 LSN값을 비교하여 분할 여부를 판단한다. 이때 기억했던 LSN이 노드의 NSN보다 작으므로 분할되었음을 감지하고 오른쪽 링크를 따라가면서 노드의 NSN이 기억하고 있는 LSN 보다 작을 때 까지 탐색을 수행하게 된다.

이 방법 역시 문제가 있다. 전역계수기를 사용하면

노드 분할을 할 때 현재 노드를 분할하기 전에 먼저 상위 노드에 잠금을 획득하고 분할을 진행해야 한다. 이외에도 회복 목적을 위해 분할이 끝날 때까지 분할에 참여하는 모든 노드들에 잠금을 유지한다. 이로 인해 [5]에서 보다 더 오랜 시간 동안 탐색연산이 지연될 수 있다.

이상 언급한 동시성 제어 알고리즘의 문제점은 공통적으로 삽입 연산 수행 시 트리 구조 변경연산을 수행할 때는 동시에 트리상의 여러 레벨에 있는 노드에 잠금을 유지하고 있어야 한다는 것이다. 이 잠금은 탐색연산을 지연시키게 되어 동시성 성능을 떨어뜨린다. 이외에도 재삽입 연산에 대한 동시성 제어방법에 대해서는 전혀 고려하고 있지 않고 있기 때문에 이들 알고리즘은 재삽입 연산을 삽입연산의 일부로 사용하는 색인 구조에서는 사용할 수 없다. 이런 점을 착안하여 이 논문에서는 다음과 같은 동시성 제어 알고리즘의 설계 목표를 세울 수 있었다. 먼저 탐색연산의 지연시간을 최소화 할 수 있는 동시성 제어 알고리즘을 설계한다. 두 번째로 재삽입 연산에 대한 효율적인 동시성제어 방법을 제공하여 재삽입 연산 중에도 탐색 연산이 제대로 동작할 수 있도록 한다.

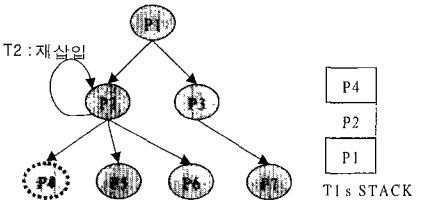
3. 제안하는 동시성 제어 알고리즘

이 장에서는 이 논문에서 제안하는 동시성 제어 방법에 대해 자세히 기술한다. 먼저 제안하고자 하는 동시성 제어 알고리즘의 전체적인 특징에 대해서 설명하고 제안하는 알고리즘을 삽입, 삭제, 탐색 연산으로 나누어 설명한다.

3.1 제안하는 알고리즘의 특징

이 논문에서 제안하는 동시성 제어 알고리즘의 특징은 다음과 같이 요약할 수 있다. 첫 번째로, 제안하는 방법은 링크 기법을 기반으로 한다. 2장에서 설명한대로 링크 기법은 BLink-Tree에서 처음 제안된 것으로써 각 레벨의 모든 노드들을 오른쪽 링크를 통해 연결하여 탐색 연산시에 잠금 결합을 수행하지 않아도 상위 노드에 반영되지 않은 하위 노드의 분할을 감지하여 이를 보상할 수 있도록 하는 기법이다. 본 연구에서는 [6]에서 제시하고 있는 링크 기법을 사용한다. 즉, 전역 계수기를 사용하여 NSN을 할당하는 방법을 취해서 비단말 노드의 엔트리에 NSN이 포함되어 저장공간의 효율을 떨어뜨리는 문제를 방지한다. 이때 전역계수기는 LSN(Log Sequence Number)을 이용한다. LSN역시 NSN처럼 단조증가하며 절대 감소하지 않는 특성을 갖는다.[2]

두 번째 특징은 래치와 잠금을 혼용한다는 것이다. 래치와 잠금 모두 색인 구조의 노드에 사용된다. 색인 구조의 노드에 대한 래치는 여러 트랜잭션들이 색인 구조의 한 노드를 접근할 때 이들간의 동기화를 이루어 주며 노드의 물리적 일관성을 보장하기 위한 것이다. 잠금은 동시에 수행되는 재삽입 연산으로 인한 다른 삽입 및 삭제 연산의 경로 유실의 문제(Path-Loss Problem)를 해결하기 위한 것이다. 그림 3 에서 이 경로 유실의 문제가 어떤 경우에 나타나는가를 보여주고 있다.



T1 : P4에 새로운 엔트리 삽입
P4에 대한 엔트리를 수정하기 위해 P2에 접근
T2 . 재삽입 수행시 P4에 대한 엔트리를 삭제
T1 : P2에서 P4에 대한 엔트리를 찾을 수 없음

그림 3 경로 유실의 문제

재삽입 연산이 수행되는 노드를 루트노드로 하는 부트리 내에서 수행중인 삽입이나 삭제연산이 트리를 거슬러 올라가면서 삽입이나 삭제로 인한 MBR정보를 변경하려 할 때 경로를 유실할 수 있다. 색인 노드에 대한 잠금은 이와 같은 문제를 해결한다. 삽입과 삭제를 위해 하향으로 트리를 순회 할 때는 거치는 모든 노드에 공유 잠금을 획득하고, 이를 트랜잭션이 끝나는 시점에서 해제한다. 이후 트리를 거슬러 올라가면서 작업을 수행하는 과정에서 재삽입이 발생하여 이를 수행하기 위해서는 그 노드에 먼저 배타 잠금을 획득해야 한다. 따라서 트랜잭션은 그 노드에 재삽입을 수행하기 위해 그 노드의 부 트리 내에서 수행되는 모든 변경연산이 완료된 것을 확인할 수 있다. 이때 만일 해당 노드에 배타 모드의 잠금을 획득하지 못했을 경우에는 재삽입을 수행하지 않고 분할을 수행한다. 색인 노드에 대한 잠금은 탐색연산에 전혀 영향을 미치지 않는다. 탐색연산은 자신이 접근하려는 노드에 공유모드의 래치만 획득하므로 색인 노드에 대한 삽입 또는 삭제연산의 잠금으로 인한 지연은 발생하지 않는다.

색인 노드 중 루트 노드에 행하는 잠금은 특별한 의미를 갖는다. 이때의 잠금은 트리 잠금을 구현하기 위해서 사용된다. 제안하는 동시성 제어 기법은 재삽입이나 분할연산 같은 넘침 처리연산과 MBR 변경연산을

직렬로 수행한다. 이때 이들을 직렬로 수행시키는 방법으로 트리 잠금을 사용한다. 즉, 한 트잭션이 삽입 연산 수행 도중 노드에 넘침을 발생시키고 이를 처리하기 위해서 먼저 쓰기 모드의 트리 잠금을 획득해야 한다. 이 트랜잭션이 수행되는 동안 다른 트랜잭션은 넘침이 발생하거나 MBR이 변경되어도 수행하지 않고 기다리게 된다. 기타 다른 삽입 연산은 트리 잠금을 획득하지 않고도 수행이 가능하며 탐색연산 역시 수행이 가능하다.

세 번째 특징은 재삽입 노드를 사용하여 재삽입 연산에 대한 동시성을 지원하고 있다는 것이다. 재삽입을 위해 트리에서 삭제한 엔트리들을 일정 영역에 저장하고 이를 탐색연산들이 참조 할 수 있도록 한다. 제안하는 방법에서는 삭제한 엔트리들을 재삽입 노드라고 하는 인덱스를 구성하는 노드외의 별도의 노드에 보관하고 탐색 연산들이 이를 참조할 수 있도록 하고 있다. 이 재삽입 노드는 색인 구조에 하나씩 할당된다. 이유는 동시에 발생할 수 있는 넘침 처리 연산이나 MBR 변경 연산을 하나로 제한하고 있기 때문이다. 다시 말하면, 동시에 발생할 수 있는 재삽입 연산의 수가 하나라는 것을 의미한다.

마지막으로 제안하는 알고리즘에서는 삽입연산을 수행하는 트랜잭션은 최대 한 레벨의 노드에만 래치를 유지하면 되도록 보장한다. [6]에서와 같이 전역계수기를 사용하여 NSN을 노드에 부여하지만 트리잠금을 통해 분할과 MBR변경연산의 수행을 동시에 하나로 제한하므로 분할을 위해 부모노드에 래치를 미리 획득할 필요가 없다. 이것은 탐색연산이 지연시간이 길어지는 것을 막는다.

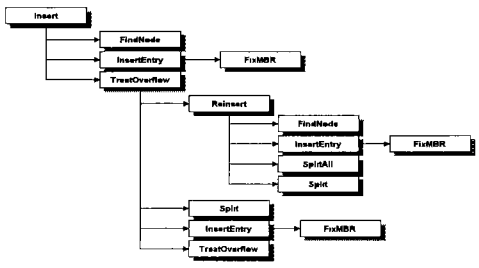


그림 4 삽입 알고리즘의 부 모듈들

3.2 삽입 연산

이 논문에서 제안하는 동시성제어 알고리즘의 삽입연산을 구성하는 주요 부 모듈은 그림4 에서 보는 바와 같이 Insert, FindNode, TreatOverflow, ReInsert, Split, SplitAll, FixMBR 등이다. FindNode는 삽입할 엔트리가 위치할 가장 적절한 노드를 찾아 주는 기능을 하고, TreatOverflow는 어떤 노드에 넘침이 발생할 때 분할을 수행할 것인지 재삽입을 수행할 것인지를 판단해서 이를 수행하게 하는 기능을 한다. ReInsert는 TreatOverflow에서 발생한 넘침을 재삽입으로 처리하고자 할 때 호출하는 모듈이다. 이 모듈에서는 넘침이 발생한 노드에서 재삽입 엔트리들을 선택하여 삭제하고

```
엔트리를 삽입할 단말노드를 찾는다 ( FindNode ) ;
if ( 단말노드에 넘침 발생 )
  트리 잠금 획득 ;
  오버 플로우 처리 수행 ( TreatOverflow ) ;
  goto end ;
else
  엔트리를  단말노드에 삽입 ;
  if ( 단말노드의 MBR이 변경 )
     단말노드의 배타래치 해제 ;
     트리 잠금 획득 ;
     MBR 변경 수행 ( FixMBR ) ;
     goto end ;
  end if
end if
end :
획득한 모든 잠금 해제;
```

그림 5 삽입 알고리즘의 의사 코드 (Insert)

```
curnode = rootnode;
curnode에 공유 래치 획득;
while ( infinite loop )
  for( curnode의 각 엔트리에 대해서 )
     childnode = 삽입하는 엔트리를 위해 가장
적절한 자식                          노드 ;
  end for
  curnode의 공유래치 해제
  if ( childnode의 level이 targetlevel )
     childnode에 공유래치 획득, 공유잠금 획득;
  else
     childnode에 배타래치 획득, 배타잠금 획득;
  end if
  if ( childnode의 nsn 이 curnode의 lsn 보다 크면)
        childnode  = 오른쪽 링크를 따라 가면서
                  적절한 노드 선택;
     childnode에 배타/공유래치, 배타/공유잠금 획득;
  end if
  if ( childnode의 level 이 target level )
     return childnode, pathstack;
  end if
  childnode를 pathstack에 Push;
  curnode = childnode;
end while
```

그림 6 삽입 알고리즘의 의사 코드 (FindNode)

이 재삽입 엔트리들을 재삽입 노드에 저장한 후 각각의 엔트리들을 색인에 삽입한다. 제안하는 알고리즘에서는 재삽입 도중 발생하는 모든 넘침은 분할로 처리하는데 이 기능을 하는 것이 SplitAll이다.

전체적인 삽입연산은 새로운 엔트리를 삽입할 단말 노드를 찾는 것으로부터 시작된다. 이 단말 노드를 찾는 모듈이 그림 6의 FindNode이다. FindNode를 수행하게 되면 엔트리를 삽입할 단말 노드와 루트부터 단말 노드를 찾아간 경로를 저장하는 경로 스택(Path Stack)을 반환한다. 이 경로 스택에 들어 있는 루트 노드를 제외한 모든 노드에는 어떤 형태의 잠금이 걸리게 된다. 단말 노드에는 쓰기 모드의 잠금이 걸리게 되며 비단말 노드에는 공유모드의 잠금이 걸린다. 이 외에도 단말 노드에는 배타래치가 걸리게 된다. 일단 FindNode를 통해 새 엔트리가 삽입될 단말 노드를 찾으면 그 단말 노드가 새 엔트리를 수용할 수 있는지 없는 지를 판단한다.

```
if (curnode == 루트노드 )
  루트 분할 수행;
  return;
end if
if ( curnode 에 조건부 배타잠금 요청, 획득 )
  재삽입 수행( ReInsert );
  if ( 재삽입 결과 curnode가 분할되었으면 )
     curnode의 배타래치 해제 ;
     비단말 노드인 경우 배타잠금 해제;
     if ( parentnode에 분할 반영시 넘침이 발생 )
        curnode = parentnode;
        넘침 처리 수행 (TreatOverflow);
     end if
  else
     return;
  end if
else
  분할 수행;
  if ( parentnode에 분할 반영 시 넘침이 발생 )
     넘침 처리 수행 ( TreatOverflow );
  else
     트리잠금 해제;
  end if
end if
return;
```

그림 7 삽입 알고리즘의 의사 코드(TreatOverflow)

새로운 엔트리를 수용할 공간이 없으면 넘침이 발생하게 되고 그림 7의 TreatOverflow를 호출하여 넘침 처리를 시작하게 된다. TreatOverflow에서는 앞서 설명한 바와 같이 어떤 노드에 넘침이 발생했을 때 이를 처리해 주는 기능을 한다. 여기에서는 가장 먼저 현재 넘침이 발생한 노드에서 재삽입을 수행할 것인지 아니면 분할을 수행할 것인지를 결정한다. 이것은 현재 노드에 조건부 배타 잠금을 시도하여 결정한다. 배타 잠금을 획득하면 재삽입을 수행하고 획득하지 못하면 분할을 수행한다. 단말 노드의 경우 모든 넘침은 재삽입으로 이어진다. FindNode 수행 시 단말 노드에는 배타 잠금을 획득하기 때문이다. 비단말 노드에서는 해당 비단말 노드를 루트 노드로 하는 부 트리 내에서 다른 트랜잭션이 삽입이나 삭제를 수행하는 경우에는 재삽입을 수행하지 않고 분할을 수행한다. 재삽입을 수행할 경우에는 그림 9의 ReInsert를 호출하고 분할을 할 때는 그림 8의 Split을 호출한다. Split에서는 새로운 노드를 할당하고 배타래치를 획득한다. 현재 노드가 단말 노드이면 배타잠금도 획득한다. 새로운 노드에는 현재 노드의 NSN을 NSN으로 할당하고 현재 노드에는 LSN을 이용하여 NSN을 할당한다.

ReInsert에서는 먼저 색인 구조에 하나씩 할당되어 있는 재삽입 노드에 조건부 배타 잠금을 요청한다. 배타 잠금을 획득하면 재삽입을 수행하고 획득하지 못하면 탐색 연산과의 교착상태에 빠지는 것을 방지하기 위해 현재 노드의 배타래치를 해제하고 무조건부 배타 잠금을 재삽입 노드에 요청하여 획득한 다음 재삽입 노드에 배타래치를 획득하고 재삽입을 진행한다. 재삽입 엔트리를 선택하여 이를 삭제하고 삭제한 엔트리들을 재삽입 노드에 기록한다. 이때 재삽입 노드의 첫 번째 레코드에 현재 재삽입이 발생하고 있는 레벨과 노드의 식별자 그리고 재삽입 엔트리들의 MBR을 계산하여 기록한다. 그리고 현재 노드에는 삭제된 재삽입 엔트리들의 숫자를 기록한다. 이것이 끝나게 되면 재삽입 엔트리들을 하나씩 색인 트리에 삽입한다. 재삽입 도중 발생하는 모든 넘침은 SplitAll을 통해 분할로 처리한다. 재삽입을 수행해도 현재 노드에 다시 넘침이 발생하는 경우에는 현재 노드를 분할(Split)하고 ReInsert를 종료하고 그렇지 않으면 그냥 종료한다. 그림 SplitAll에서는 한 노드에서 분할을 수행하고 이를 상위 노드에 반영할 때 다시 넘침이 발생하면 이를 분할로 처리한다.

다시 TreatOverflow에서는 ReInsert가 어떻게 종료하였는가에 따라 다음 수행할 작업을 결정한다. 만일 현재 노드에 넘침이 발생하지 않았으면 트리 잠금을 해제하고 삽입 연산을 종료한다. 그렇지 않고 현재 노드에 넘침이 발생하여 분할을 수행하였으면 경로 스택에서 부모 노드를 읽어오고 다시 그 부모 노드에 분할

```
newnode 할당;
newnode 에 배타래치 획득;
if ( newnode == 단말노드 )
    newnode 에 배타잠금 획득 ;
end if
curnode에서 newnode로 옮길 엔트리들 선택;
curnode에서 선택한 엔트리들 삭제하고 newnode
            에 삽입;
newnode의 nsn = curnode의 nsn;
curnode의 nsn = curnode의 lsn;
curnode의 nextnode = curnode;
curnode와 newnode의 배타래치 해제;
```

그림 8 삽입 알고리즘의 의사 코드(Split)

```
재삽입 노드에 배타잠금 획득, 배타래치 획득, 배타잠금 해제;
재삽입 엔트리를 curnode에서 삭제하고 재삽입 노드에 삽입;
curnode의 첫번째 레코드에 엔트리 개수 기록;
재삽입 노드와 curnode의 배타래치 해제,
curnode의 변경된 MBR을 조상 노드들에 반영 (FixMBR);

for (모든 재삽입 엔트리)
        삽입할 엔트리를 위한 적절한 노드를 찾는다. (FindNode)
        if (찾은 노드에서 오버플로우 발생)
                if (오버플로우가 발생한 노드가 curnode)
                        curnode를 분할 ;
                        return;
                end if
                분할수행 (SplitAll)
        else
                curnode에 삽입한다.
                if (curnode의 MBR이 변경 )
                조상 노드들에 MBR 반영 (FixMBR)
                end if
        end if
        curnode의 첫번째 레코드 값을 1 감소
end for
```

그림 9 삽입 알고리즘의 의사 코드 (ReInsert)

된 엔트리를 삽입할 여유공간이 있는지를 검사한다. 여유공간이 있으면 엔트리를 삽입하고 조상 노드에 그에 대한 MBR 변경을 수행하고 트리 잠금을 해제하고 삽입연산을 종료한다. 여유공간이 없으면 다시 재삽입을 할 것인지 분할을 할 것인지를 결정하기 위해 부모 노드에 조건부 배타 잠금을 요청하게 된다.

이때 경로 스택에 있는 부모 노드에 현재 노드에 대한 엔트리가 없으면 이것은 부모 노드가 분할되었다는 것을 의미하며 조건부 배타 잠금을 현재 노드에 대한 엔트리가 있는 부모 노드에 시도하는 것이 아니라 경로 스택에 있는 부모노드에 시도하게 된다. 이렇게 하는 이유는 현재 노드에 대한 엔트리가 있는 부모노드에는 경로 스택에 있는 부모 노드에 대한 잠금 정보가 없기 때문에 분할된 부모 노드의 부 트리 내에 존재하는 삽입이나 삭제연산을 인식하지 못하는 경우가 있기 때문이다. 그림 10 은 비단말 노드에서 넘침 발생시 재삽입을 할 것인지 분할을 할 것인지를 결정하기 위해 경로 스택에 있는 부모 노드에 조건부 배타 잠금을 요청하는 과정을 설명하고 있다.

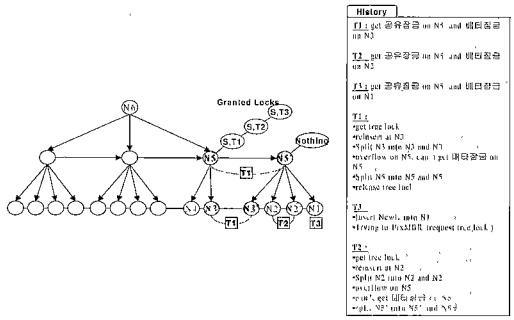


그림 10 경로 스택에 있는 노드에 조건부 배타잠금 시도

여기서 배타 잠금을 획득하게 되면 다시 재삽입을 수행하고 그렇지 않으면 분할을 수행한다. 이런 과정을 반복적으로 수행하여 삽입연산을 마치게 된다. 다시 처음으로 돌아가서 단말 노드에 새로운 엔트리를 삽입할 여유공간이 있는 경우에는 새로운 엔트리를 삽입하고 현재 단말 노드의 MBR이 변경되었는지를 검사한다. MBR이 변경되었으면 이를 조상 노드들에 반영하고 그렇지 않으면 종료한다. 조상 노드들에 변경된 MBR을 반영하는 것은 FixMBR을 호출하여 수행한다. 이 FixMBR 역시 트리 잠금을 획득한 후에 수행되어야 한다. FixMBR의 수행이 끝나면 트리 잠금을 해제한다.

그림 10에 대해서 보다 자세히 설명한다. T1, T2, T3는 모두 엔트리를 삽입하는 트랜잭션들이다. 먼저 T1이 엔트리를 삽입하기 위해 N5를 거쳐 N3에 도달한다. 이때 N5에는 공유 잠금을 그리고 N3에는 배타 잠금을 획득한다. T2와 T3 역시 뒤이어 엔트리를 삽입하기 위해 N5를 거쳐 각각 N2와 N1에 도달한다. 이때 N5에는 T1과 마찬가지로 T2와 T3가 연속으로 공유 잠금을 획득한다. 이후에 T1이 N3에 엔트리를 삽입하는 과정에서 넘침이 발생하여 트리 잠금을 획득한 후 재삽입을 수행하고 이어 N3를 분할한다. 다시 이를 상위 노드 N5에 반영하는 과정에서 넘침이 발생하고 재삽입을 수행하기 위해 N5에 조건부 배타 잠금을 요청한다. 하지만 T2와 T3가 이미 공유 잠금을 획득하고 있기 때문에 배타 잠금을 얻지 못하고 분할을 수행한다. 분할

을 수행하고 T1은 트리 잠금을 해제하고 종료하게 된다.

이후에 T3가 N1에 엔트리를 삽입하고 MBR이 변경되어 이를 상위에 반영하기 위해 트리 잠금을 요청한다. 이 보다 조금 앞서 T2는 N2에 엔트리를 삽입하는 과정에서 넘침이 발생하여 트리 잠금을 획득하고 재삽입을 수행한다. 재삽입을 수행하고 다시 N2를 분할한다. 이를 N5'에 반영하는 과정에서 N5'에 다시 넘침이 발생하여 조건부 배타 잠금을 N5에 요청한다. 하지만 T3가 여전히 공유잠금을 N5에 유지하고 있기 때문에 재삽입을 수행하지 못하고 분할을 수행한다. 만일 이 상황에서 T3가 조건부 배타 잠금을 N5'에 요청한다면 T3는 이를 획득할 것이고 N5'에서 재삽입을 수행할 것이다. 하지만 이것은 N2가 변경된 MBR을 상위에 반영하는 과정에 경로를 유실할 수가 있다.

3.3 탐색 연산

이 논문에서 제안하는 알고리즘에서는 탐색연산 수행 중에는 항상 한 레벨에 있는 노드에만 공유 래치를 유지하고, 링크 기법을 사용하여 현재 방문하는 노드의 하위 노드의 분할을 감지한다. 범위 질의나 K-최근접 질의 모두 탐색을 시작하기 전에 재삽입 노드에 공유 잠금을 획득한다. 이것은 탐색 도중에 재삽입 노드에 변경이 생기는 것을 막기 위해서 이다. 탐색 도중 재삽입 노드에 변경이 생기게 되면 재삽입 엔트리들을 탐색 대상에서 제외하는 경우가 생길 수 있기 때문이다. 범위 질의에 대한 의사코드를 그림 11 에서 보여 준다.

3.4 삭제 연산

제안하는 동시성 제어 알고리즘에서는 노드 삭제를 지원하지 않는다. 삭제 연산의 수행은 가장 먼저 삭제할 엔트리가 어느 단말 노드에 위치하는지 파악한다. 삭제할 엔트리가 존재하는 노드가 파악되면 그 노드에서 엔트리를 삭제한다. 그리고 그 노드에 남아 있는 엔트리가 전체의 20%이상이면 엔트리를 삭제하고 조상 노드에 엔트리 삭제로 인해 변경된 MBR을 반영한다.

그렇지 않고 20% 이하이면 삭제만 하고 변경된 MBR을 상위에 반영하지 않는다. 노드의 엔트리가 모두 삭제되어 노드가 비게 되는 경우에는 상위 노드에서 현재 노드에 대한 엔트리의 MBR을 음수로 바꾼다. 탐색연산의 경우에는 이 음수 MBR을 갖는 엔트리는 탐색 대상에서 제외한다. 비어있는 노드를 탐색할 수 있기 때문이다. 삽입 연산의 경우에는 FindNode에서 음수 MBR의 경우에는 양수로 바꾸어 새로운 엔트리가 삽입될 가능성이 있는 노드로 고려한다. 이렇게 하여 비어 있는 노드가 다시 사용될 수 있는 가능성을 주게 된다. 삭제연산에 대한 의사 코드를 그림 12에서 보여준다.

```
curlevel = 루트 노드의 레벨;
루트노드를 queue 에 push;
parentnsn = 루트노드의 nsn;
재삽입 노드에 공유잠금을 요청하여 획득;
while ( queue != empty )
    curnode = 큐에서 노드를 pop;
    curnode 에 공유래치 획득;
    level = curnode의 레벨;
    if ( level < curlevel )
        curlevel = level;
    end if
    for ( parentnsn < curnode의 lsn )
      if (curnode != 리프노드 )
          if (curlevel == 재삽입이 발생하는 레벨 )
              재삽입 노드에 공유래치 획득;
              질의를 만족하는 재삽입 노드의 자식노드들을
queue에 push;
              공유래치 해제;
          endif
          질의를 만족하는 자식 노드들을 queue 에 push;
      else
          if ( level == 재삽입이 발생하는 레벨 )
              재삽입 노드에 공유래치 획득;
              질의를 만족하는 재삽입 노드의 객체들을
resultset에 push;
              공유래치 해제;
          end if
          질의를 만족하는 리프 노드의 객체들을 resultset에
push;
      end if
      curnode의 공유래치 해제;
      curnode = curnode의 nextnode;
      curnode에 공유래치 획득;
    end for
    curnode에 공유래치 획득;
end while
재삽입 노드의 잠금을 해제;
```

그림 11 범위 질의에 대한 의사코드(Range Search)

```
삭제할 엔트리를 포함하는 단말노드를 찾는다;
엔트리 삭제;
if ( 단말노드의 엔트리들의 크기가 전체 노드크기의 20%
이상 )
  트리 잠금 획득;
  단말노드의 배타래치 해제;
  변경된 MBR을 상위에 반영 (FixMBR);
  트리 잠금 해제;
else if ( 단말 노드가 empty )
  트리 잠금 획득;
  단말 노드의 상위 노드 엔트리의 MBR을 음수로 변환 ;
  트리 잠금 획득;
end if
```

그림 12 삭제 알고리즘의 의사코드(Delete)

3.5 알고리즘의 정확성 증명

제안하는 동시성 제어 알고리즘에서는 래치와 잠금을 혼용한다. 동시에 여러 트랜잭션이 수행되면서 래치와 잠금을 획득하고 해제할 때 래치-래치, 잠금-잠금 또는 래치-잠금에 의한 교착상태가 발생할 수 있다. 이 절

에서는 제안하는 동시성 제어 알고리즘은 교착상태가 발생하지 않음을 보인한다.

설명을 위해서 제안하는 알고리즘에서 발생 가능한 연산들을 다음과 같은 기호를 통해서 표현한다.

OPS : 탐색 연산

OPF : 새로운 엔트리 삽입할 단말 노드를 찾는 연산

OPT : 새로운 엔트리를 삽입한 단말 노드에 넘침이 발생했을 때 이를 처리하는 연산

OPX : 새로운 엔트리를 삽입한 단말 노드의 MBR이 변경되어 이를 상위 노드로 전파하는 연산

이 연산들은 동시에 수행될 수 있는 것들도 있고 절대로 동시에 수행될 수 없는 연산들도 있다. (표 1)에 동시에 수행 가능한 연산들과 그렇지 않은 연산들을 표시한다. ○는 서로 동시에 수행 가능함을 나타내고 ×는 그렇지 않음을 나타낸다.

표 1 동시에 수행 가능한 연산들

	OP_S	OP_F	OP_T	OP_X
OP_S	○	○	○	○
OP_F	○	○	○	○
OP_T	○	○	×	×
OP_X	○	○	×	×

이 표로부터 알 수 있듯이 동시에 수행할 수 있는 연산들은 OPS - OPS, OPS - OPF, OPS - OPT, OPS - OPX, OPF - OPF, OPF - OPT 그리고 OPF - OPX 이다. 이들이 교착상태에 빠지지 않음을 보이면 제안하는 알고리즘은 교착상태를 발생하지 않음을 증명할 수 있다. 이들이 교착상태에 빠지지 않음을 보이기 전에 먼저 각각의 연산이 어떻게 잠금과 래치를 수행하는지 설명한다.

OPS는 OPS, OPT, OPX, 그리고 OPF와 동시에 수행이 가능하다. OPS는 재삽입 노드에 대한 무조건부 공유 잠금 그리고 방문하는 각 노드에 대해서 공유 래치를 획득한다. 노드에 대한 래치는 방문하기 전에 획득하고 다음 방문할 노드에 래치를 요청하기 전에 해제한다. 재삽입 노드에 대한 공유 잠금은 탐색을 시작하기 전에 무조건 부로 요청하여 획득한 후 탐색을 수행한다.

OPF는 OPS, OPT, OPX 그리고 OPF와 동시에 수행이 가능하다. OPF는 비단말 노드에는 공유래치, 단말노드에는 배타래치를 획득한다. 그리고, 루트노드를 제외한 비단말 노드에는 무조건부 공유 잠금을 획득하고 비단말 노드에는 무조건부 배타 잠금을 획득한다. 래치와 잠금의 획득순서는 방문할 노드에 먼저 래치를 획득하면 잠금을 요청하여 획득한다. 획득한 래치는 다음 방문할 노드에 대한 래치를 요청하기 전에 해제한다. 하지만 잠금은 해제하지 않는다.

OPT는 OPS와 OPF와 동시에 수행이 가능하다. OPT는 단말노드의 넘침이 발생하는 순간 트리에 조건부 배타 잠금을 요청한다. 만일 트리 잠금을 획득하지 못하면 현재 노드의 배타래치를 해제하고 트리에 무조건부 배타 잠금을 요청한다. 이를 획득하면 다시 현재 노드에 배타래치를 획득하고 OPT를 계속 수행한다. 또한 OPT는 넘침이 발생한 노드에 조건부 배타 잠금을 요청한다. 이것이 획득되면 재삽입을 수행하고 그렇지 않으면 분할을 수행한다. 재삽입을 수행할 때는 먼저 재삽입 노드에 조건부 배타 잠금을 요청하고 획득하지 못하면 현재노드의 배타래치를 해제한 후 다시 무조건부 배타 잠금을 요청한다. 배타 잠금을 획득하면 현재 노드에 배타래치를 획득하고 재삽입 노드에 배타래치를 획득한 후 재삽입 노드의 배타 잠금을 해제한다. 재삽입이 수행되면 현재노드의 배타래치와 배타 잠금을 해제하고 필요에 따라서 상위노드에 배타래치와 배타 잠금을 획득한 후 다음 작업을 수행한다. OPT는 시작전에 트리 잠금을 획득한다. 따라서 다른 OPT나 OPX와는 동시에 수행될 수 없다. 그리고 OPT가 동시에 유지하는 래치의 수는 하나이다.

OPX는 OPS, OPF와 동시에 수행이 가능하다. OPX는 단말노드에 새로운 엔트리를 삽입한 후 노드의 MBR이 변경되면 현재노드의 배타래치를 해제하고 트리에 무조건부 배타 잠금을 요청하여 획득한다. 트리 잠금을 획득한 후에 상위노드에 배타 래치를 획득하고 변경된 MBR을 반영한다. 상위노드에 획득한 배타 래치는 방문할 상위노드의 상위노드에 래치를 요청하기 전에 해제한다. OPX역시 동시에 유지하는 최대 래치의 수는 하나이다.

다음에서 동시에 수행될 수 있는 연산의 쌍 각각에 대해서 교착상태가 발생하지 않음을 보인다.

1. OPS - OPS

OPS는 공유잠금과 공유래치만을 획득하기 때문에 서로 잠금이나 래치가 충돌하지 않는다. 따라서 이 두연산으로 인한 교착상태는 발생하지 않는다.

2. OPS - OPT

OPS 와 OPT는 서로 교착상태에 빠지지 않는다. 이들의 잠금과 래치를 획득하는 순서를 보면 알 수 잇듯

이 이 두 연산으로 인해서 교착상태가 발생할 수 있는 경우는 다음과 같다. OPS가 재삽입 노드(RN)에 공유 잠금을 획득한 상태에서 재삽입이 발생하는 노드(CN)에 공유래치를 요청하고 동시에 OPT가 CN에 배타래치를 획득한 상태에서 RN에 배타 잠금을 요청하면 교착상태에 빠질수 있다. 하지만 제안하는 알고리즘에서는 이를 방지하기 위해서 OPT가 RN에 배타 잠금을 요청할 때 조건부로 요청하게 한다. 조건부 배타 잠금을 요청해서 실패하면 OPS가 RN에 공유잠금을 유지한다는 뜻이므로 CN에 배타 래치를 해제하고 RN에 무조건부 배타 잠금을 요청하여 획득한다. 따라서, OPS 와 OPT 사이에는 교착상태가 발생하지 않는다.

3. OPS-OPF

OPS 와 OPF 모두 동시에 유지하는 최대 래치의 수가하나이다. 또한 OPS는 인덱스 노드에는 잠금을 획득하지 않으므로 OPF가 획득한 잠금과 충돌이 생기지 않는다.

4. OPS-OPX

OPS 와 OPF 모두 동시에 유지하는 최대 래치의 수가하나이다. 또한 OPS는 인덱스 노드에는 잠금을 획득하지 않으므로 OPF가 획득한 잠금과 충돌이 생기지 않는다.

5. OPF-OPF

OPF는 단말노드를 제외한 노드에는 모두 공유 래치와 공유 잠금을 획득한다. 또한 단말 노드에 배타 래치와 배타 잠금을 획득하는 순서가 항상 래치-잠금 순이므로 OPF 와 OPF는 서로 교착상태에 빠지지 않는다.

6. OPF-OPT

OPF는 하향 순회하면서 래치-잠금의 순서로 노드에 래치와 잠금을 획득한다. 그리고 OPT는 상향 순회하면서 래치-잠금의 순서로 노드에 래치와 잠금을 획득한다. 래치의 경우에 두 연산 모두 다음 방문할 노드에 래치를 요구하기 전에 현재 노드의 래치를 해제하므로 래치로 인한 교착상태는 발생하지 않는다. 이 두 연산간에 교착상태가 발생할 수 있는 경우는 OPT가 단말노드(LN)에 배타 잠금을 획득한 상태에서 상위노드(PN)에 배타 잠금을 요청하고 있고, OPF가 PN에 공유 잠금을 획득한 상태에서 LN에 배타 잠금을 요청하는 상황이다. 제안하는 알고리즘에서는 OPT가 상위노드에 잠금을 요청할 때는 조건부로 요청하도록 해서 실패하면 더 이상 잠금을 요청하지 않고 다음 작업을 수행한다. 따라서 잠금으로 인한 교착상태는 발생하지 않는다.

7. OPF-OPX

OPX는 상향 순회하면서 래치를 요청하여 획득한 후 MBR 변경을 수행한다. OPX역시 다음 방문할 노드에 래치를 획득하기 전에 현재 노드의 래치를 해제한다. 따라서 OPF와 OPX는 서로 교착상태에 빠지지 않는다.

이상의 경우에서 교착상태에 빠지지 않으므로 제안하는 알고리즘은 교착상태가 발생하지 않는다.

4. 실 험

본 장에서는 이 논문에서 제안한 동시성 제어 알고리즘을 CIR-Tree에 적용시켜 이를 MiDAS-Ⅲ에서 구현하고 이의 성능을 실험을 통해 평가한다.

4.1 구현 환경

앞서 설명한 대로 이 논문에서는 제안하는 알고리즘을 CIR-Tree에 적용시켰다. 그리고 이를 실제 DBMS의 하부 저장 시스템인 MiDAS-Ⅲ에서 구현하고 실험하였다. 실험에 사용된 컴퓨터는 Sun UltrSparc-Ⅱ 기종에 2개의 CPU, 그리고 512 MBytes의 주기억 공간을 가지고 있다. 또한, 제안하는 알고리즘의 비교대상으로 [6]에서 제안하는 알고리즘(CGiST) 역시 MiDAS-Ⅲ상에서 구현하여 실험하였다. 실험의 공정성을 기하기 위해서 제안하는 알고리즘의 비교대상인 CGiST역시 CIR-Tree에 적용해서 MiDAS-Ⅲ위에서 구현하였다.

페이지(노드)의 크기는 4K로 하였으며 실험에 사용된 데이타는 10 차원의 가상데이타이며 정규분포를 갖는다. 또한 MiDAS-Ⅲ의 버퍼 개수는 100개로 고정시켰다. 실험 방법은 먼저 MiDAS-Ⅲ의 버퍼영역을 초기화시키고 20,000개의 데이타로 트리를 구축한다. 다음에 탐색연산을 50회, 삽입연산을 500회씩 수행하는 프로세스를 동시에 수행시킨다. 실험은 제안하는 알고리즘이 다른 방법보다 삽입연산의 수에 관계없이 탐색연산을 수행하는 시간이 일정함을 보이기 위해서 다음과 같은 실험을 수행했다. 50회의 탐색연산을 수행하는 프로세스를 4개로 고정하고 500회의 탐색연산을 수행하는 삽입 프로세스의 수를 2 ~ 20으로 변화시켜가면서 총 수행 시간을 측정하고, 이를 이용해 평균 수행시간, 처리율 및 트랜잭션하나의 처리시간을 계산하였다.

4.2 성능 평가

그림 13과 그림 14는 탐색프로세스의 수를 4개로 고정시키고 삽입프로세스의 수를 2 ~ 20으로 변화시키면서 동시에 수행했을 때 각 탐색 및 삽입 프로세스의 평균 수행시간을 나타낸 것이다. 즉, 하나의 탐색프로세스는 50회의 탐색트랜잭션이 수행되므로 50회의 트랜잭션을 수행하는데 드는 평균 시간이 되고 삽입의 경우에는 500회의 삽입 트랜잭션을 수행하는데 드는 평균 시간이 된다. 그림 13에서는 최초에는 제안하는 방법의

삽입 수행시간이 CGiST보다 조금더 느리다가 삽입프로세스의 수가 증가하면서 점점 그 차이가 미약해져 가는 것을 보여준다. 하지만 그림 14에서 볼 수 있듯이 CGiST는 삽입프로세스가 늘어나면 탐색프로세스의 평균 수행시간도 계속해서 같이 늘어나는 반면 제안하는 알고리즘은 탐색 수행시간이 늘어나다가 삽입프로세스가 8, 10, 12을 넘어서면서 거의 증가하지 않고 일정한 값을 유지함을 볼 수 있다.

그림 15부터 그림 18은 각각 그림 13과 그림 14에서 보여준 탐색프로세스와 삽입프로세스의 평균수행시간으로부터 삽입 및 탐색 트랜잭션의 처리율과 응답시간을 계산하여 표시한 것이다. 그림 15과 그림 16은 각각 삽입트랜잭션과 탐색트랜잭션에 대한 처리율을 나타낸다. 삽입트랜잭션에 대한 처리율을 보면 삽입프로세스(트랜잭션)의 수가 2~8 일때는 CGiST가 약간 더 많은 삽입트랜잭션들을 처리하지만 10, 12, 14을 넘어서면서부터는 거의 차이가 없어짐을 볼 수 있다. 탐색의 경우에는 삽입프로세스(트랜잭션)의 수가 2, 4 개일때는 차이가 별로 없다가 6, 8을 넘어서면서 점차 그 차이가 벌어지고 어느 순간에는 처리율이 거의 감소되지 않음을 볼 수 있다.

그림 17와 그림 18은 각각 삽입트랜잭션와 탐색트랜잭션의 응답시간을 나타낸다. 그림 15에는 삽입트랜잭션의 응답시간을 나타내고 있는데 제안하는 알고리즘의 응답시간이 CGiST보다 매우 근소한 차이로 느림을 알 수 있다. 하지만 삽입프로세스의 수가 증가할수록 그 차이가 없어짐을 볼 수 있다. 그림 18은 탐색트랜잭션의 응답시간을 나타내고 있다. CGiST는 삽입프로세스의 수가 증가함에 따라서 응답시간이 계속해서 느려짐을 볼 수 있다. 하지만 제안하는 알고리즘의 탐색트랜잭션의 응답시간은 최초에 CGiST와 별 차이가 없다가 삽입프로세스(트랜잭션)의 수가 14이상이 되면서 부터는 응답시간이 거의 증가하지 않고 일정해짐을 볼 수 있다.

이런 실험 결과가 나오는 원인은 다음과 같이 해석해 볼 수 있다. CGiST는 분할을 수행하기 위해서 분할에 참여하는 모든 노드에 래치를 획득하고 분할수행이 종료된 후에 이를 해제한다. 즉, 만일 트리의 레벨이 3이라면 하나의 트랜잭션이 동시에 최대 6개의 노드에 래치를 유지한다는 것이다. 또한 MBR변경을 상위에 반영하기 위해서 항상 상위 노드에 래치를 획득한 후 현재 노드의 래치를 해제하는 방법을 사용하고 있다. 이것 역시 하나의 트랜잭션이 동시에 2개의 노드에 래치를 획득해야 한다는 것이다.

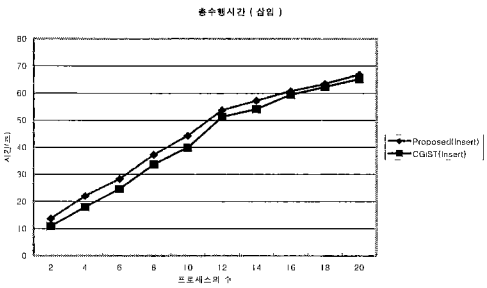


그림 13 삽입 프로세스(500회 삽입 수행)의 총 수행시간

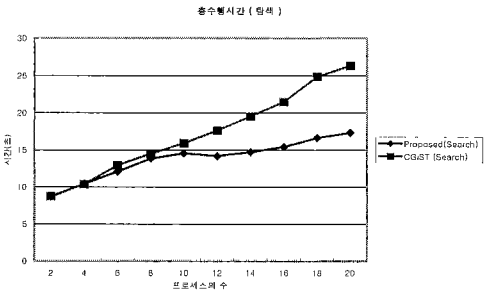


그림 14 탐색 프로세스(50회 탐색 수행)의 총 수행시간

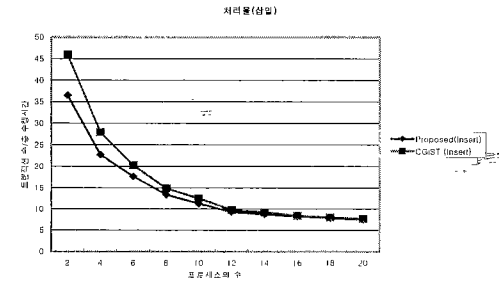


그림 15 처리율 (트랜잭션수/초 : 삽입)

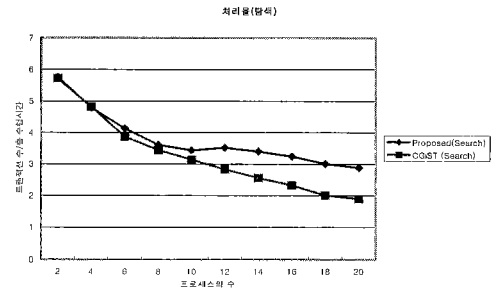


그림 16 처리율 (트랜잭션수/초 : 탐색)

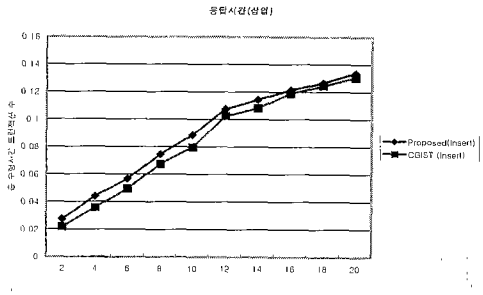


그림 17 응답시간 (초/트랜잭션 : 삽입)

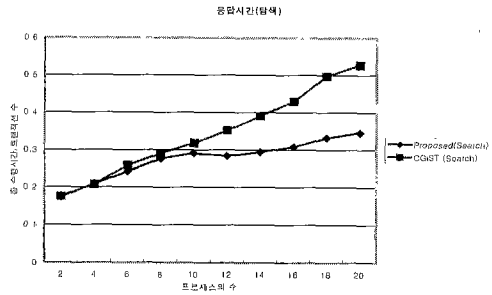


그림 18 응답시간 (초/트랜잭션 : 탐색)

이에 반해 제안하는 알고리즘에서는 분할시에는 보통 2개의 노드에만 배타래치를 유지하면 된다. 루트 분할시에는 3개의 노드에 배타래치를 유지한다. MBR 변경시에도 동시에 최대 1개의 노드에만 래치를 유지하면 된다. 따라서 탐색연산이 배타 래치에 의해 지연되는 시간이 줄게 된다.

삽입 프로세스가 2, 4, 6 일때는 제안하는 알고리즘과 CGiST가 탐색을 수행하는 시간에 있어서 별 차이를 보이지 않고 있다. 이것은 또한 다음과 같이 해석 할 수 있다. 실험에서의 각각의 삽입 프로세스는 500회 삽입을 하고, 탐색 프로세스는 50회 탐색을 한다. 이 경우 삽입 프로세스의 수가 기준 이하이면 삽입을 수행할 때 빈번한 분할 및 MBR 변경이 발생하지 않을 수 있다. 게다가 탐색 프로세스의 수행시간이 짧다 보니 탐색 프로세스가 충분히 분할 및 MBR변경을 경험하지 못한다. 이런 이유로 6개이하의 삽입 프로세스에서는 탐색의 수행시간이 비슷하다.

삽입트랜잭션의 처리율과 응답시간측면에서는 제안하는 알고리즘이 CGiST에 비해 약간 나쁘거나 거의 비슷하다. 원인은 제안하는 알고리즘에서는 동시에 수행될 수 있는 분할이나 MBR변경연산을 하나로 제한했지만 CGiST는 동시에 여러 트랜잭션이 분할과 MBR 변경을 수행할 수 있기 때문이다. 하지만 분할과 MBR을 동시에 수행하게되면 그만큼 배타 래치가 걸리게 되는 노드들이 많아지게 되고 다른 삽입연산이 FindNode를 수행할 수 있는 기회가 줄어들게 되어 결과적으로는 별차이를 보이지 않게된다.

이상을 종합해 볼 때 분할이나 MBR 변경과 같은 연산이 빈번할수록 제안하는 알고리즘의 탐색 성능이 우수함을 알 수 있다. 실험은 10차원의 특징을 갖는 데이타를 사용하였는데 고차원일수록 하나의 특징벡터의 크기가 커지므로 차원의 수가 많은 고차원 데이타일수록 삽입시에 발생하는 분할의 횟수는 더 빈번하게 된다. 따라서, 고차원 일 수록 제안하는 알고리즘은 타 방법보다 더욱 우수한 탐색성능을 보장한다.

5. 결 론

이 논문에서는 고차원 색인 구조의 특징에 맞는 동시성 제어 알고리즘을 제안하고 이를 실제 DBMS의 하부 저장시스템에서 구현하였다. 고차원 색인 구조에서는 일반적으로 탐색연산을 수행할 때 다른 색인 구조에 비해 훨씬 많은 노드를 접근하며 탐색연산의 빈도가 삽입에 비해 많다. 또한 색인 구조의 성능을 향상시키기 위해 재삽입 연산을 사용하는 예가 많다. 이 논문에서는 실험으로 타 방법과의 실제적인 비교를 통해 제안하는 알고리즘이 이러한 특징을 만족시키고 있다는 것을 입증하였다. 제안하는 알고리즘은 DBMS의 하부 구조에 구현되어 있지만 아직 회복을 고려하고 있지 않다. 향후 연구에서는 제안하는 동시성 제어 알고리즘에 맞는 회복기법을 설계하고 또한 이를 DBMS의 하부 저장시스템에 구현한다.

참 고 문 헌

[1] 최길성, 이석희, 송석일, 유재수, 조기형, 고차원 색인 구조를 위한 효율적인 동시성제어 알고리즘, KISS 98 가을 학술발표(I), pages 54-56, October 1998.

[2] 이석희, 송석일, 유재수, 이미지 검색을 위한 고차원 색인구조, KISS 데이타베이스 연구회 논문지, 제14권 제 4호, pages 53-68, December 1998.

[3] 이석희, 유재수, 조기형, 허대영, "CIR-Tree : 효율적인 고차원 색인기법", 한국정보과학회 논문지(B), 한국정보과학회, 제26권 제6호, pages 724~734, Jun 1999.

[4] M. Kornacker, C. Mohan and J. M. Hellerstein, Concurrency and Recovery in Generalized Search Trees, In Proc. ACM SIGMOD Conf., pages 62-72, May 1997.

[5] M. Kornacker and D. Banks, "High-Concurrency

Locking in R-Trees," In Proc. 21st Int'l Conference on VLDB, pages 134-145, September 1995.
[6] C. Mohan, D. Harderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write Ahead Logging," ACM TODS, 17(1), pages 94-162, March 1992.
[7] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," In Proc. ACM SIGMOD Conf., pages 371-380, June 1992.
[8] P.L. Lehmann and S.B. Yao, "Efficient Locking for Concurrent Operations on B-Trees," ACM TODS, 6(4), pages 650-670, December 1981.
[9] V. Ng and T. Kamada, "Concurrent Accesses to R-Trees," In Proc. of Symposium on Large Spatial Databases, pages 142-161, 1993.
[10] Shuanhu Wang, Joseph M. Hellerstein and Ilya Lipkind. "Near-Neighbor Query Performance in Search Trees." UC Berkeley Tech Report CSD-98-1012, eptember 1998.
[11] J. Nievergelt, H. Hinterberger, and K. Sevcik. "The grid file: An adaptable, symmetric multikey file structure." ACM Transactions on Database Systems(TODS), 1984.
[12] A. Guttman. "R-Trees: A dynamic index structure for spatial searching." In Proc. ACM SIGMOD Conf., pages. 47-57, 1984.
[13] N.Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. "The R*-Tree: an efficient and robust access method for points and rectangles." ACM SIGMOD, pages 322-331, May, 1990.
[14] K. Lin, H. V. Jagadish, and C. Faloutsos. "The TV-Tree an index stucture for high dimensional data." In VLDB Journal, 1994.
[15] S. Berchtold, D. A. Keim, and H. P. Kriegel. "The X-Tree: An index structure for high-dimensional data." Proc. of VLDB, 1996.
[16] D. White and R. Jain. "Similarity indexing with the SS-Tree." Proc. of ICDE, 1995.
[17] N. Katayama and S. Satoh. "The SR-Tree: An index structure for high dimensional nearest neighbor queries." Proc. of SIGMOD, 1997.
[18] K. Chakrabarti and S. Mehrotra. "The Hibrid Tree :An Index Structure for High-Dimensional Feature Spaces." Proc. of ICDE, 1999

송 석 일
1988년 충북대학교 공과대학 정보통신공학과(공학사). 1999년 충북대학교 정보통신공학과(석사과정). 관심분야는 고차원 데이타 색인, 정보검색프로토콜(Z39.50), SGML, OODBMS, 저장 시스템, 회복기법, 동시성제어

박 춘 서
1999년 충북대학교 정보통신공학과(공학사) 1999년 ~ 현재 충북대학교 정보통신공학과 석사 과정. 관심분야는 동시성제어, 저장 시스템, 멀티미디어 정보검색, 병렬처리 등.

이 석 희
정보과학회논문지 : 데이타베이스 제 27 권 제 3 호 참조

유 재 수
정보과학회논문지 : 데이타베이스 제 27 권 제 1 호 참조