다차원 색인구조를 위한 동시성제어 기법 및 회복기법

송 석 일[†]·유 재 수[††]

요 약

이 논문에서는 다차원 색인구조의 동시성을 최대화하는 동시성제어 알고리즘과 이를 위한 회복기법을 제안한다. 다차원 색인구조에서 동시성을 저하는 가장 큰 요인은 MBR 변경연산과 분할 연산이다. 제안하는 알고리즘은 먼저 MBR 변경연산으로 인한 동시성 저하를 최소화하기 위해서 PLC(Partial Lock Coupling) 기법을 제안한다. 이 기법은 기존 방법에 비해 잠금결합을 사용하는 회수를 크게 줄여 동시성을 높인다. 또한, MBR 변경의 수행 중에도 탐색자들이 해당 노드를 접근할 수 있도록 하는 MBR 변경 방법을 제안한다. 분할로 인한 동시성 저하를 해결하기 위해서 노드 분할로 인한 탐색자의 지연 시간을 최소화 할 수 있는 새로운 분할방법을 제안한다. 제안하는 알고리즘을 BADA-4 DBMS의 저장시스템인 MiDAS-3에서 구현하여 성능평가를 수행한다. 다양한 실험을 통해 제안하는 방법이 기존 방법보다 우수함을 보인다. 마지막으로, 이 논문에서는 제안하는 동시성제어 방법에 적절한 회복기법을 제안한다. 회복기법은 동시성을 최대한 보장할 수 있도록 설계되었으며 빠른 회복시간을 보장한다.

Concurrency Control and Recovery Methods for Multi-Dimensional Index Structures

Seok Il Song[†]·Jae Soo Yoo[††]

ABSTRACT

In this paper, we propose an enhanced concurrency control algorithm that maximizes the concurrency of multi-dimensional index structures. The factors that deteriorate the concurrency of index structures are node splits and minimum bounding region (MBR) updates in multi-dimensional index structures. The proposed concurrency control algorithm introduces PLC (Partial Lock Coupling) technique to avoid lock coupling during MBR updates. Also, a new MBR update method that allows searchers to access nodes where MBR updates are being performed is proposed. To reduce the performance degradation by node splits the proposed algorithm holds exclusive latches not during whole split time but only during physical node split time that occupies the small part of a whole split process. For performance evaluation, we implement the proposed concurrency control algorithm and one of the existing link technique-based algorithms on MIDAS-3 that is a storage system of a BADA-4 DBMS. We show through various experiments that our proposed algorithm outperforms the existing algorithm in terms of throughput and response time. Also, we propose a recovery protocol for our proposed concurrency control algorithm. The recovery protocol is designed to assure high concurrency and fast recovery.

키워드 : 다차원 색인구조(Multi-Dimensional Index Structure), 동시성 제어(Concurrency Control), 회복기법(Recovery Method)

1. 서 론

다차원 색인구조는 지리정보시스템, 내용기반 이미지 검색 시스템, 멀티미디어 데이터베이스 시스템, 이동객체 관리 시스템과 같이 다차원의 특징 벡터에 대한 유사성 검색 시스템에서 중요한 위치를 차지한다. 지난 20여년 동안 R-트리[1], R*-트리[12], TV-트리[8], X-트리[16], SS-트리[4], SR-트리[13], CIR-트리[6], Hybrid-트리[7]와 같은 다양한 트리구조의 다차원 색인구조들이 제안되었다. 이 색인구조들을 다중 사용자 환경의 실세계의 응용에서 사용하기 위해서는 동시성제어 기법과 적절한 회복기법이 필수적이다. 이에 여러 동시성제어 기법 및 회복기법들이 제시되었다[5, 10, 11, 15, 17, 19, 20].

기존의 동시성제어 알고리즘들은 두 가지 문제를 해결하고자 하였다. 첫 번째는 삽입, 삭제, 변경, 검색 연산들이 동시에 발생할 때 동시 연산처리율을 최대로 하면서 색인구조의 일관성을 유지하고자 하는 것이다[5, 10, 11, 15, 17]. 두 번째 문제는 탐색영역을 유령(Phantom) 삽입이나 삭제로부터 보호하는 것이다[10, 19, 20]. DBMS는 반드시 유령현상 방지 기법을 제공해야 한다. 그러나, DBMS 구현자들은 4단계의 고립 수준(Isolation Level)을 두어서 정확성과 성능 사이에 타협책을 만들어 놓고 있다. 응용 프로그래머들이나

※ 이 연구는 2001년도 학술진흥재단(KRF-2001-041-E00233)의 지원으로 수행되었음.
† 준 회 원 : 충북대학교 대학원 전기전자컴퓨터공학부 정보통신공학과
†† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
논문접수 : 2002년 7월 4일, 심사완료 : 2002년 12월 9일

SQL 사용자들은 해당 응용영역에 적합한 고립수준을 선택할 수 있게 된다[10, 19, 20].

위에 언급한 다차원 색인구조를 사용하는 응용분야들에서는 유령현상 보다는 성능이 보다 중요한 관심사가 된다. 예를 들면, 이동객체 데이터베이스 시스템에서는 대량의 변경이나 탐색 연산들을 실시간으로 처리할 필요가 있다. 유령현상 방지를 위해서 고성능을 포기한다면 원래의 목적을 달성 할 수 없게 된다. 또한 내용기반 검색 시스템이나 전자도서관 시스템들에서는 탐색이 주로 발생하며 변경은 상대적으로 덜 빈번하게 나타난다. 이런 응용들에서 유령현상이 재산이나 인명을 해치는 손실을 끼치거나 큰 문제를 일으키지 않는다.

이 논문에서는 위에서 언급한 두 문제 중에서 전자에 해당하는 성능을 최대화하기 위한 향상된 동시성제어 기법을 제안한다. 다차원 색인구조의 동시성 성능의 향상을 위해서는 몇 가지 문제들을 해결해야 한다. B-트리나 B+-트리와 같은 단일 차원의 색인구조와는 다르게 노드 분할이 매우 긴 시간을 필요로 한다. 일반적으로 다차원 색인구조의 노드에 저장된 엔트리들은 정렬이 되어 있지 않기 때문에 분할 차원과 분할 위치를 계산하는데 매우 긴 시간을 필요로 한다. 다차원 색인구조를 위한 대부분의 기존 동시성제어 알고리즘은 분할이 수행되는 노드에 배타잠금(Exclusive Lock)이나 배타래치(Exclusive Latch)를 획득한다. 이 배타잠금 및 래치들은 탐색자들을 지연시킨다. 분할연산은 색인트리를 거슬러 올라가면서 조상노드에 분할을 반영하고 다시 조상노드는 분할 될 수 있다. 분할 연산은 다차원 색인구조의 동시성을 떨어뜨리는 주요원인이 된다.

MBR 변경 연산 역시 탐색자들을 지연시킨다. MBR 변경은 분할연산에 비해서 덜 복잡하고 짧은 시간을 필요로 하지만 훨씬 더 빈번히 발생하여 색인구조의 동시성을 크게 저하시키는 원인이 된다. 기존에 다차원 색인구조를 위한 다수의 동시성제어 알고리즘들이 제안되었지만 이상에서 열거한 탐색지연 요인을 완전히 없애지 못했다. 사실, 이런 탐색지연을 완전히 제거하는 것은 매우 어려운 일이지만 최소화할 수는 있다.

이 논문에서는 PLC(Partial Lock Coupling) 기법을 제안해서 MBR 변경에 의한 탐색지연을 줄인다. 그리고, 분할연산에 의한 지연시간을 줄이기 위해서 노드 분할중에 배타래치 또는 잠금 시간을 최소화하기 위한 분할기법을 제안한다. 또한, 동시성제어 알고리즘에 대한 회복기법에 대해서 언급한다. 회복기법은 시스템이나 트랜잭션 실패가 발생했을 때 색인구조의 일관성을 유지하는데 필수적이다.

이 논문의 구성은 다음과 같다. 2장에서는 다차원 색인구조를 위한 기존의 동시성제어 알고리즘들을 설명하고 분석한다. 제안하는 동시성제어 알고리즘을 3장에서 자세하게 설명한다. 4장에서는 제안한 알고리즘의 정확성을 증명하고, 6장에서는 제안한 동시성제어 알고리즘에 적합한 회복기법에 대해서 설명한다. 제안한 동시성제어 알고리즘의 우수성을 보이기 위해서 다양한 실험을 하고 그 결과를 7장에서 기술한다. 마지막으로, 8장에서 결론을 맺는다.

2. 관련 연구 및 동기

다수의 다차원 색인구조를 위한 동시성제어 알고리즘이 제안되었다. 이들은 잠금 결합기반과 링크기법 기반으로 분류할 수 있다. 잠금 결합 기반의 알고리즘들은 탐색을 수행할 때 현재 노드에 대한 잠금을 다음 방문할 노드에 대한 잠금을 획득한 다음에 해제한다. 노드 분할이나 MBR 변경중에는 여러 노드에 잠금을 동시에 획득하며 이 점은 동시성을 매우 심각하게 저하시킨다. 링크기반의 알고리즘들은 언급한 잠금결합 기반의 알고리즘들의 문제점을 해결하기 위해서 제안되었다. 이 기법들은 탐색 연산중에도 잠금 결합을 수행할 필요가 없다. 즉, 최대 한 노드에만 잠금을 획득한다. 그러나, 노드분할이나 MBR 변경을 위해서는 잠금결합을 수행해야 한다.

링크 기법은 Leman과 Yao에[14] 의해서 B-트리를 위한 동시성제어 기법으로 제안되었다. 원래의 트리구조를 단말노드뿐 아니라 같은 높이의 모든 노드들을 서로 오른쪽 링크를 통해 연결하도록 수정하였다. 노드가 둘로 분할될 때 적절한 오른쪽 링크를 노드에 할당한다. 같은 높이의 모든 노드들은 자연스럽게 노드내의 최대키를 기준으로 정렬이 된다. 탐색자가 분할된 그러나 이를 아직 부모노드에 반영하지 못한 노드에 접근했을 때 그 노드의 최대키가 찾고자 하는 키보다 작은지 비교하고 작을 경우 분할이 되었음을 알게 되어 오른쪽 링크를 따라서 이동하게 된다. 이 방법은 특정 시점에 최대 하나의 잠금만 설정하면 된다. 따라서, 삽입자들은 탐색자들을 I/O 시간 동안 지연시키지 않게되어 동시성이 향상된다.

다차원 색인구조에서는 B-트리에서와 같이 노드들간에 순서화가 이루어지지 않아서 기존의 링크기법을 그대로 따를 수 없다. [11]에서는 노드들이 순서화 되어있지 않은 점을 보완하기 위해서 각 노드에 오른쪽 링크 외에 LSN(Logical Sequence Number)를 추가로 할당하였다. LSN은 B-트리의 최대키와 같은 역할을 하게 되며 트리 순회자들이 상위노드에서 감지하지 못한 분할을 찾아낸다. 그러나, 노드분할이나 MBR 변경을 처리하기 위해서 트리를 거슬러 올라갈 때는 잠금 결합을 수행해야 한다. 이것은 트리의 동시성을 떨어뜨린다. 또한, 비 단말노드의 각 엔트리들은 연계된 하위노드에 대한 LSN을 추가로 가지고 있어서 노드의 펜-아웃을 감소시키는 문제도 가지고 있다.

다차원 색인구조를 위한 또 다른 링크기법 기반의 동시성제어 알고리즘으로 [10]이 있다. 이 알고리즘은 [11]의 각

엔트리가 부가정보를 갖는 문제를 해결하고 있다. CGIST는 전역 순차번호(Global Sequence Number)를 이용해서 부가정보를 엔트리에 포함시켜야 하는 필요성을 제거했다. 이 연구에서, 전역 계수기는 부가적인 정보를 엔트리에 추가시켜야 하는 필요성을 제거했지만 몇 가지 부작용이 따른다. [11]에서의 LSN과 같은 역할을 하는 NSN(Node Sequence Number)는 트리 전역적인 순차번호이다. 노드 분할 중에 이 전역 계수기는 증가하고 새로운 값이 원래 노드에 할당되고 새롭게 생성된 이웃노드에는 원래 노드의 이전 NSN을 할당한다. 이 알고리즘이 정확하게 동작하기 위해서는 노드를 분할할 때 그 부모노드에 먼저 잠금을 획득한 후에 노드를 분할하고 NSN을 할당하고 전역계수기를 증가시킨다. 이런 이유 때문에 노드 분할을 수행하는 중에는 둘 또는 그 이상의 트리 수준(Level)에 여러 잠금을 획득해야 한다. 이것은 탐색자들의 지연시간을 매우 길게 한다.

이상 설명한 동시성제어 알고리즘들은 노드 분할이나 MBR 변경에 참여하는 여러 수준의 노드에 배타모드의 잠금이나 래치를 획득한다. 배타모드의 잠금이나 래치는 동시에 수행되는 탐색자들을 지연시킨다. 그 결과로 전체적인 탐색자의 성능이 크게 저하된다. [15]는 이런 문제를 해결하기 위한 가장 최근에 제안된 링크 기반의 동시성제어 알고리즘이다. 여기에서는 TDIM(Top Down Index Modification) 기법을 소개하고 있다. 즉, 삽입자가 새로운 엔트리를 삽입할 적당한 노드를 찾기 위해서 색인 트리를 순회하는 도중에 MBR 변경을 수행한다. 추가적으로, MBR 변경을 수행하는 중에 노드에 획득한 잠금은 탐색자가 획득하는 잠금과 호환이 가능하다. 즉, 변경중인 MBR의 내용을 탐색자들이 읽을 수 있다. 하지만 MBR 변경을 한차원씩 수행하기 때문에 탐색자들은 정상적으로 탐색을 수행할 수 있다. [15]에서는 TDIM외에도 CCU(Copy Based Concurrent Update)나 CCUNQ(Concurrent Update with Nonblocking Queries)와 같은 최적화된 분할 알고리즘을 제안하고 있다.

[15]에서 제안하는 알고리즘에는 몇 가지 문제가 있다. TDIM은 삽입자의 잠금 결합 필요성을 제거했다. 그러나, 이 알고리즘에서는 삭제를 고려하지 않고 있다. 삭제자들은 삭제할 엔트리를 찾기 위해서 정확 탐색(Exact Match)과 같이 트리 순회를 해야한다. 다차원 색인구조는 루트노드에서 목적노드까지 다중 경로가 존재하기 때문에 삭제자들은 현재 방문하는 노드가 목적노드의 조상이라고 확신할 수 없다. 그 결과 하향으로 MBR을 변경하기 위해서는 먼저 삭제할 엔트리를 찾아야 한다. TDIM을 사용하는 삭제자나 삽입자가 동시에 수행되고 있을 때 색인 트리가 비 일관된 상태에 빠지는 현상이 발생할 수 있다. 예를 들어보자. 삽입자가 새로운 엔트리 NE를 색인 트리에 삽입하기 위해서 트리순회를 시작한다. 순회도중 비단말 노드 N을 방문하고 노드의 여러 엔트리들중 NE를 삽입할 부트리를 가리키는 엔트리 E를 선택한다. 엔트리 E는 자식노드에 대한 포인터 CN과 CN에 대한 MBR의 쌍으로 구성된다. 이어서 삭제자가 N을 방문하고 E의 CN에 대한 MBR을 변경한다. 이 MBR 축소는 NE를 배제하게 되어 색인구조는 비 일관된 상태로 가게된다. 이상 설명한 것처럼 TDIM은 삭제에 대한 특별한 대처없이는 삭제를 필요로 하는 실세계의 응용에서 사용되기 어렵다.

또한, CCU와 CCUNQ는 분할로 인한 질의의 지연시간을 효과적으로 줄이고 있지만 분할을 수행하기 위해서 부가 공간을 필요로 하며 특히, CCUNQ는 쓰레기 수거 작업을 주기적으로 수행해 주어야 한다. 이런 특징들은 알고리즘의 구현을 어렵게 하며 개발비용을 높이는 요인이 된다. 이 논문에서는 PLC라는 새로운 MBR 변경방법을 제안한다. 이 방법은 대부분의 경우에 잠금 결합을 필요로 하지 않으며 삭제연산도 같이 고려하고 있다. 또한, 부가적인 공간이나 추가적인 작업을 필요로 하지 않으면서 질의의 지연을 최소화하는 간단한 분할 알고리즘을 제안한다.

3. 제안하는 동시성제어 알고리즘

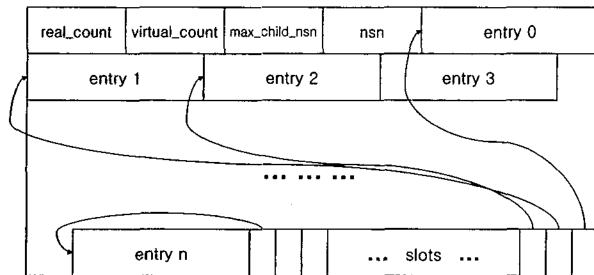
이 장에서는 제안하는 알고리즘을 상세하게 설명한다. 이전 장에서 언급한대로 제안하는 알고리즘의 가장 중요한 부분은 MBR 변경과 노드분할이다. 이 두 방법에 대해서 먼저 자세히 설명하고 다음에 전체적인 알고리즘을 설명한다.

3.1 노드 구조

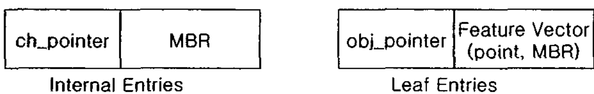
(그림 1)은 제안하는 알고리즘의 엔트리와 노드 구조를 보여준다. 노드는 헤더, 데이터, 슬랏 영역으로 나뉘어진다. 헤더 영역에는 nsn, max_child_nsn, real_count, virtual_count가 저장된다. nsn(node sequence number)은 노드순차번호를 뜻하며 노드가 분할될때 할당되는 값이다. max_child_nsn은 현재 노드의 자식노드들 중에서 가장 최근에 분할된 노드의 nsn을 의미한다. 이 값은 자식노드가 분할될 때마다 증가한다. 그리고, 새로 생성된 자식노드에 대한 엔트리를 현재 노드에 반영할 때 변경된다. max_child_nsn과 nsn은 2장에서처럼 미처 감지하지 못한 분할을 보완하는데 사용한다. 색인 트리를 순회할 때 (그림 2)에서 보는 것 처럼 감지하지 못한 분할에 대한 보상을 수행한다.

real_count는 노드에 저장된 실제 엔트리들의 개수이다. 이것은 제안하는 MBR 변경 기법에서 이용된다. MBR 변경 기법은 후에 자세히 설명할 것이다. 엔트리들은 데이터 영역에 저장되고 슬랏들은 실제 엔트리들을 가리키는 변위를 가지고 있다. 노드의 엔트리들을 읽기 위해서는 먼저 슬랏을 읽고 슬랏에 저장된 변위를 이용해서 엔트리들을 접근한다. 이런 엔트리 접근 형태는 MBR을 일관되게 변경하기 위해서 사용된다. 삽입자나 삭제자들이 슬랏과 virtual_count

를 어떻게 이용해서 MBR을 변경하는지 3.3절에서 설명한다.



(a) Node structure



(b) Entry structure

(그림 1) 노드와 엔트리 구조

```
if (parent nodes max_child_nsn < current nodess nsn )
    follow the right link of the current node
else
    normally traverse the next level
```

(그림 2) 감지하지 못한 분할에 대한 보완

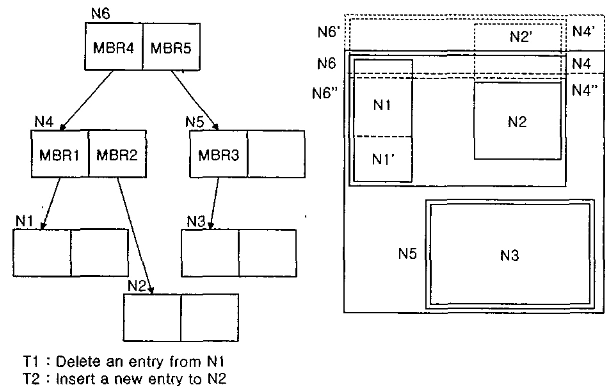
3.2 PLC(Partial Lock Coupling) : MBR 변경 방법

[10, 11]과 같은 기존의 동시성제어 기법들은 래치(잠금) 결합을 이용해서 삽입자와 삭제자들이 트리를 위로 올라가면서 변경을 수행하는 순서를 유지하여 색인 트리의 일관성을 유지한다. TDIM이 이러한 문제를 해결하기 위한 방법으로 제안되었지만 2장에서 설명한데로 완전한 알고리즘이라고 할 수 없다. 이 논문에서 제안하는 알고리즘은 MBR을 상향식으로 변경한다. 그러나, 기존의 방법과는 다르게 잠금 결합을 부분적으로 이용하는 PLC 기법을 제안한다.

(그림 3)은 잠금 결합을 사용하지 않고 변경연산을 상향으로 수행했을 때 색인트리의 일관성이 흐트러지는 예를 보여준다. 트랜잭션 T1은 노드 N1에서 엔트리 하나를 삭제한다. 그리고, 트랜잭션 T2는 새로운 엔트리하나를 N2에 삽입한다. 이어서, T1과 T2는 트리를 거슬러 올라가면서 N1과 N2의 MBR 변경내용을 조상노드들에 반영하게 된다. 각 트랜잭션의 동작을 시간의 흐름에 따라서 나열한다. t_i(i : 0~n, $t_{i-1} < t_i$)는 트랜잭션이 특정 행위를 취하는 시간을 말한다.

- **t_0** : T1은 N1에서 엔트리를 삭제하고 이로 인해 MBR이 N1′으로 변경된다. T1은 N1에 대한 잠금을 해제하고 배타 잠금을 N4에 요청한다.
- **t_1** : T2는 N2에 새로운 엔트리를 삽입하고 이로 인해 MBR이 N2′으로 확장되었다. T2는 N2에 대한 배타잠금을 해제하고 N4에 배타잠금을 요청한다.
- **t_3** : T1은 N4에 배타잠금을 획득하고 N1을 위한 MBR1을 변경한다. N1의 변경으로 N4의 MBR은 N4″으로 축소된다. 다시 T1은 N4에 대한 배타잠금을 해제하고 배타잠금을 N6에 요청한다.
- **t_4** : T2는 N4에 배타잠금을 획득하고 N2를 위한 MBR2를 변경하고 이것은 N4 전체의 MBR을 N4′으로 확장시키게 된다.
- **t_5** : T2가 T1을 앞지른다. 즉, T2가 N6에 대한 배타잠금을 먼저 획득하고 N4에 대한 MBR4를 N4′으로 변경한후 완료(commit)한다.
- **t_6** : T1은 N6에 배타잠금을 획득하고 N4에 대한 MBR4를 N4″으로 변경한다.
- **t_6 이후** : 다른 탐색자들은 T2에 의해서 새롭게 삽입된 엔트리를 찾을수 없게 된다.

이상의 시나리오에 따르면 단말노드의 MBR 축소를 조상노드에 반영하기 위해 트리를 거슬러 올라가는 삭제자들을 역시 트리를 거슬러 올라가면서 같은 조상노드들에 MBR을 변경하는 삭제자나 삽입자들이 앞지르게 되면 문제가 발생한다. 그러나, 반대의 경우 즉, 삽입자를 다른 삭제자나 삽입자가 앞지르는 경우에는 기존의 MBR 변경방법을 조금만 변경하면 트리의 일관성을 유지할 수 있다.



(그림 3) 트리의 일관성이 깨지는 예

이와 같은 사실로부터 트리를 거슬러 올라가면서 연산을 수행하는 삭제자나 삽입자들이 다른 삭제자를 앞지르는 것을 방지한다면 트리의 일관성이 유지된다는 결론을 얻을 수 있다. 이 논문에서 제안하는 PLC 기법은 이와 같은 사실을 이용한다. PLC는 노드 분할이나 삭제와 같이 MBR 축소를 수행해야 하는 연산들의 경우에만 잠금 결합을 수행하고 그 외의 연산들은 잠금 결합을 수행하지 않는다. 일반적으로 다차원 색인구조가 이용되는 분야에서는 MBR 확장의 경우가 MBR 축소의 경우보다 많으며 그렇지 않은 경우라도 평

균적으로 반정도의 성능개선을 얻을 수 있다.

그러나, PLC 기법을 정확하게 사용하기 위해서는 반드시 색인 트리의 일관성이 깨지는 것을 막아야 한다. 제안하는 MBR 변경 알고리즘은 부분적인 잠금 결합을 이용하므로 적절한 보상장치가 없이는 트리가 비일관된 상태가 될 수가 있다. 새로운 엔트리가 단말노드에 삽입될 때 그 노드의 MBR은 새로운 엔트리를 포함하기 위해서 확장된다. 그리고, 변경된 MBR을 조상노드들에 반영한다. 새로이 삽입된 엔트리가 다른 트랜잭션에 의해 삭제되거나 엔트리를 삽입한 트랜잭션이 철회(Rollback)되기 전까지는 확장된 MBR은 축소되지 않는다. 그러나 삽입자나 삭제자들은 연산을 수행하기 전에 완료기간의 배타잠금을 데이터 파일의 객체에 획득하기 때문에 삽입된 엔트리들은 트랜잭션이 완료될 때까지 삭제되지 않는다.

새로운 엔트리가 단말노드에 추가되고 단말노드의 MBR이 변경될 때 삽입자는 MBR의 변경을 조상노드들에 반영한다. 삽입자가 변경된 MBR을 반영하는 동안 단말노드에 추가된 새로운 엔트리는 다른 트랜잭션에 의해서 삭제될 수 없다. 그러므로, 삽입자는 단지 단말노드의 조상노드의 MBR이 새로 삽입한 엔트리를 포함하는지만 검사하면 된다. 만일 부모노드의 단말노드를 표현하는 MBR이 새로운 엔트리를 포함한다면 삽입자는 MBR을 더 이상 변경할 필요가 없다. 그렇지 않다면 삽입자는 새로운 엔트리를 포함하도록 MBR을 변경한다. 그리고 변경내용을 같은 방법으로 조상들에 반영한다.

삭제자들은 잠금 결합을 이용한다. 삭제자들과 삽입자가 동시에 수행되는 경우에는 MBR이 새로운 엔트리를 포함하는지 검사하는 것만으로는 부족하다. 엔트리의 삭제로 인해 축소된 MBR은 엔트리를 삭제한 트랜잭션이 완료되기 전에 다른 삽입연산에 의해서 확장될 수 있다. 그러므로 삭제자들은 잠금 결합을 이용해서 삭제자들의 순서를 유지시켜야 한다. (그림 3)의 시나리오로 돌아가서 N6에 배타잠금이 획득될 때까지 T1이 N4에 배타잠금을 유지하므로 t5에서 T2는 T1을 앞지를 수 없다. 그러므로, 결과 MBR 4는 새로이 삽입된 엔트리를 포함하도록 N4'이 될 것이다. 그리고, 다른 탐색자들은 그 엔트리를 찾을 수 있게 된다.

보조정리 1 : MBR 축소나 MBR 확장 행위는 항상 색인 트리의 일관성을 보존한다.

증명 : 동시에 수행되는 MBR 변경의 결과가 직렬로 수행된 MBR 변경의 결과와 같다는 것을 보여주어야 한다. EA를 MBR 확장 행위라 하고 SA를 MBR 축소 행위라 하자. $N_0, N_1, N_2, \cdots, N_i$를 높이가 i인 색인트리의 루트노드 N_i에서 단말노드 N_0까지의 경로상에 있는 노드들이라 한다. SA들은 잠금 결합을 사용하면서 수행이 되므로 SA들은 직렬로 수행이 된다. 그러나, EA들은 잠금 결합을 수행하지 않으므로 다음의 시나리오들이 가능하다.

시나리오 1 : t_0 : $EA(N_k)$; t_1 : $SA(N_k)$; t_3 : $SA(N_{k+1})$; t_4 : $EA(N_{k+1})$ $(k = i = 0,\ t_0 < t_1 < t_3 < t_4)$

시나리오 2 : t_0 : $EA1(N_k)$; t_1 : $EA2(N_k)$; t_3 : $EA2(N_{k+1})$; t_4 : $EA1(N_{k+1})$ $(k = i = 0,\ t_0 < t_1 < t_3 < t_4)$

시나리오 1에서 SA는 EA를 t_3에서 앞지른다. 그러나, EA는 항상 MBR이 새롭게 삽입된 엔트리를 포함하는지를 검사하고 포함하지 않을 경우 MBR을 확장한다. 그러므로, t_4에서 축소된 MBR은 새롭게 삽입된 엔트리를 포함하기 위해서 확장되며 트리의 일관성은 보존된다. 시나리오 2에서 EA1은 EA2를 t_3에서 앞지르게 된다. 이미 언급한대로 t_4에서 EA는 MBR을 검사하고 필요하면 MBR을 새롭게 삽입된 엔트리를 포함하도록 확장한다. 그러므로 역시 트리의 일관성은 보존된다. 이상 예시한 것처럼 두 가지의 경우가 가능하며 이에 대해서 항상 트리의 일관성이 보존되므로 제안하는 알고리즘의 MBR 변경 행위는 항상 색인 트리의 일관성을 보존한다.

삽입자들은 MBR 변경중에 공유래치를 획득한다. 따라서 탐색자들은 변경중인 MBR을 읽을 수 있다. 이때 탐색자들은 잘못된 MBR을 읽고 탐색에 실패할 수 있다. [15]에서는 이런 문제를 해결하기 위한 방법으로 MBR을 구성하는 각 차원의 값들을 하나씩 변경한다. 제안하는 알고리즘에서도 이 방법을 사용할 수 있지만 이 방법의 경우 MBR을 구성하는 각 차원의 값의 데이터 형태가 반드시 워드 크기가 되어야 한다. 그렇지 않으면 탐색자들은 완전히 변경되지 않은 값을 읽게 될 수 있다.

이 논문에서는 MBR을 표현하는 데이터 형에 제한을 주지 않는 새로운 MBR 변경 방법을 제시한다. 변경자는 먼저 변경할 MBR을 노드의 여유공간에 복사하고 변경을 수행한다. 이 시점까지는 변경되고 있는 MBR을 가리키는 슬랏의 변위는 이전 MBR을 가리키므로 해당 노드들 접근하는 탐색자들은 이전 버전의 MBR을 읽는다. 이어서 변경자는 virtual_count를 하나 증가시키고 슬랏의 변위를 복사되어 변경된 MBR을 가리키도록 변경한다. 이 시점부터 탐색자들은 변경된 MBR을 읽는다. 만일 노드에 여유공간이 없다면 즉, virtual_count가 꽉찬 상태이면 노드의 공유래치를 해제하고 배타래치를 획득하여 다른 연산이 노드에 남아있지 않은 것을 확인한 후 원하는 MBR을 그 자리에서 변경한다. 그리고, virtual_count를 real_count와 같게 한다.

보조정리 2 : 탐색자들은 항상 완전히 변경된 MBR만을 접근한다.

증명 : 탐색자들은 엔트리들을 읽기 위해서 먼저 해당 슬랏의 변위를 읽는다. 노드내의 변위를 표현하기 위해서는

2~4바이트의 정수이면 충분하다. 대부분의 현대 컴퓨터에서 2~4바이트의 값은 원자적으로 변경이 가능하다. 또한, MBR 변경자는 여유공간에서 먼저 변경을 하고 변위를 나중에 변경한다. 그러므로 탐색자들은 항상 완전한 MBR을 읽게 된다.

(그림 4)와 (그림 5)에서 MBR 변경에 대한 예를 보여주고 있다. (그림 4)에서 삽입자는

① 노드를 분할한다.

② 엔트리 2의 MBR을 변경하기 위해서 자유공간 5를 할당하고 엔트리 2를 5로 복사한후, 5의 엔트리의 MBR을 변경한다. virtual_count를 증가하고 슬랏 S2에 대한 변위를 5로 수정한다.

③ 엔트리 5의 MBR을 같은 방법으로 변경한다.

④ 두 개의 새로운 엔트리를 삽입했고 real_count를 증가했다.

⑤ 노드가 꽉차게 되어 엔트리 0의 MBR을 변경할 수 없다.

(그림 5)에서 삽입자는

① 다른 탐색자들이 노드를 읽고 있지 않다는 것을 보장하기 위해서 노드에 배타래치를 획득한다. virtual_count를 감소하고 엔트리 0의 MBR을 변경하거나 virtual_count를 감소하고 노드를 재구성한 후 엔트리 0의 MBR을 변경한다.

(그림 5)에서처럼, 배타래치를 획득했을 때 노드를 재구성하거나 단지 virtual_count를 감소만 시킬수 있다. 재구성은 자유 슬랏을 쉽게 찾게 해주지만, 탐색자들의 지연시간을 길게 한다.

(1) real_count : 5 virtual_count : 5
(2) real_count : 5 virtual_count : 6
(3) real_count : 5 virtual_count : 8
(4) real_count : 7 virtual_count : 9

(그림 4) MBR 변경 방법(1/2)

real_count : 7 virtual_count : 9
Before update S0

real_count : 7 virtual_count : 7
After update S0 (only decreased Vcount)

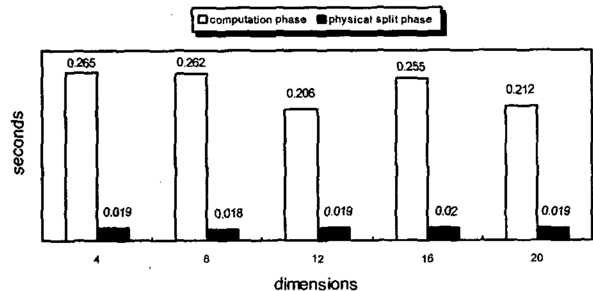
real_count : 7 virtual_count : 9
After update S0 (reorganized page)

(그림 5) MBR 변경 방법(2/2)

3.3 노드 분할 방법

일반적으로 R-트리 계열의 다차원 색인구조에서 노드의 엔트리들이 순서화 되어있지 않기 때문에 노드 분할은 긴 시간을 요구한다. 대부분의 기존 다차원 색인구조를 위한 동시성제어 기법들은 배타잠금/래치를 분할이 수행되고 있는 노드에 획득해서 탐색연산을 지연시킨다. 분할 연산은 색인 트리를 거슬러 올라가면서 분할된 내용을 조상노드들에 반영하며 이 과정에서 다시 다른 분할연산을 발생시킬 수도 있다.

일반적으로 노드 분할은 다음과 같이 두 단계로 수행된다. 첫 번째 단계에서는 분할 차원과 분할 위치를 계산한다. 이 단계는 색인구조에 따라서 다르긴 하지만 일반적으로 R-트리 계열의 색인구조에서는 매우 긴 계산시간을 필요로 한다. 두 번째 단계는 첫 번째 단계에서 계산한 분할 차원과 분할위치에 따라서 넘침이 발생한 노드를 물리적으로 두 개의 노드로 분할한다. 이미 언급한대로 첫 번째 단계가 분할에서 대부분의 시간을 차지한다. (그림 6)은 분할 연산에 걸리는 시간을 첫 번째와 두 번째 단계로 나누어 보여준다. 그림에서 보는 것처럼 첫 번째 단계가 전체 분할시간의 약 93%를 차지하고 있다. 실험에 사용된 색인구조는 R-트리계열의 색인구조중 하나인 CIR-트리이다.



(그림 6) 계산시간과 물리적 분할시간의 비교

3.4 삽입 연산

전체적인 삽입 절차는 두 단계에 걸쳐서 수행된다. 첫 번째 단계에서, 먼저 엔트리를 삽입할 단말노드를 찾기 위해서 트리를 순회한다. 이 단계에서 루트부터 해당 단말노드까지의 경로에 존재하는 노드들을 경로 스택에 저장한다. 두 번째 단계에서는 새 엔트리를 단말노드에 삽입한다. 이때 단말노드의 MBR이 새로운 엔트리의 삽입으로 변경이 되면, 변경된 MBR을 조상노드에 반영한다. 새 엔트리를 단말노드에 삽입할 수 없다면 분할 연산이 수행된다. 분할 연산은 조상노드들 중 새 엔트리를 위한 공간이 있을때까지 반복된다.

분할 연산들은 제안하는 동시성제어 알고리즘을 위한 회복기법을 위해서 트리잠금에 의해서 직렬로 수행된다. 분할을 수행하기 전에 먼저 트리잠금을 획득한다. 이어서 부모노드에 분할로 생성된 노드에 대한 엔트리를 삽입하고, 원래 노드를 위한 MBR을 변경한다. 만일 부모노드에서 다시 넘침이 발생한다면 단말노드처럼 부모노드를 분할한다. 위의 과정은 새로운 엔트리를 위한 여유공간이 있는 노드를 만날때 까지 반복하거나 루트노드를 분할할때 까지 계속 수행된다. (그림 7)~(그림 10)은 제안하는 동시성제어 알고리즘의 삽입 연산에 대한 의사코드를 보여준다.

```
InsertEntry (Entry leafentry, Node rootnode)
{
  leafnode := FindNode (leafentry, root, path) ;
  If (overflow is occurred in leafnode due to leafentr)
  {
    Obtain tree lock ;
    Split (leafentry, leafnode, path) ;
    Release tree lock ;
    Release all node locks ;
    Function End ;
  }
  Add leafentry to leafnode ;
  If (the MBR of leafnode is changed)
  {
    Release S-latch on leafnode. ;
    FixMBR (leafnode, path) ;
    Release all node locks ;
    Function End
  }
  Release S-latch on leafnode ;
  Release all node locks ;
}
```

(그림 7) 삽입 알고리즘 - Insert

● 삽입 알고리즘-Insert

맨 처음 Insert는 엔트리(leafentry)를 삽입할 단말노드(leafnode)를 찾기 위해서 FindNode를 호출한다. FindNode는 leafnode와 경로스택(path)를 반환한다. Insert는 leafnode에 leafentry를 위한 공간이 있는지 검사한다. leafnode에 공간이 있으면, leafentry를 leafnode에 공유래치를 획득하고 삽입한다. 공유래치가 leafnode에 설정이 되어있으면 탐색자들은 leafentry를 삽입하는 도중에 leafnode를 접근해서 삽입되고 있는 불완전한 leafentry를 읽을 수 있다. 이런 경우를 피하기 위해서 제안하는 알고리즘은 leafentry를 다음과 같이 노드에 삽입한다. 먼저, leafentry를 위해서 자유공간을 할당하고 슬랏을 할당한다. 이후에 leafentry를 삽입하고 leafentry를 위해 할당한 공간에 대한 변위를 슬랏에 적어놓는다. 이후에 헤더에 있는 real_count를 증가한다.

보조정리 3 : 탐색자들은 항상 일관된 노드를 읽는다.

증명 : 탐색자들은 헤더의 real_count를 읽어서 노드에 몇 개의 엔트리가 있는지 알아낸다. real_count는 워드 크기의 정수이므로 이 값의 변경은 원자적이다. 그러므로, 탐색자들은 항상 완전한 real_count를 읽는다. 또한, 삽입자들은 새로운 엔트리를 완전히 삽입한후에 real_count를 증가시킨다. 결론적으로, 탐색자들은 항상 일관된 노드들을 읽게 된다.

leafentry를 추가한 후에 Insert는 leafnode의 MBR이 확장되었는지 검사한다. 만일 MBR이 변경되었으면 FixMBR을 호출하여 변경된 내용을 조상노드들에 반영한다. 이 경우는 MBR 확장의 경우이므로 PLC에 의해서 FixMBR 호출전에 leafnode에 대한 배타잠금과 공유래치를 해제한다.

그렇지 않으면 즉, leafnode에 충분한 공간이 없으면, Insert는 SplitNode를 호출하고 넘침을 처리한다. SplitNode를 호출하기 전에 leafnode에 대한 공유래치는 해제되지만 배타잠금은 해제되지 않는다. 제안하는 알고리즘은 분할된 leafnode에 대한 배타잠금을 삽입 연산이 종료된 후 해제한다. 이 이유는 6장에서 자세하게 설명된다.

● 삽입 알고리즘-FindNode

(그림 8)의 FindNode는 엔트리를 삽입할 단말노드를 찾기 위해서 색인 트리를 따라 내려온다. FindNode는 비 단말노드에 공유래치만을 획득하고 단말노드에서는 공유래치와 배타잠금을 같이 획득한다. 단말노드의 배타잠금은 색인트리의 일관성유지를 위해서 다른 삽입자나 삭제자가 단말노드를 접근하고 변경하는 것을 막기 위함이다. FindNode는 마지막으로 leafnode와 path를 Insert에 반환한다.

```
FindNode (Entry leafentry, Node node, PathStack path)
{
  Obtain S-latch on node ;
  currentlevel = node.level ;
  Push [node, node.nsn] into path ;
  For ( ; ; )
  {
    childentry [Node node, MBR MBR] := Select the child
    entry
      from node ;
    Subtract 1 from currentlevel ;
    Release S-latch on node ;
    node := childentry.node ;
    S-latch on node ;
    If (currentlevel == leaf level)
      Obtain X-lock on node ;
```

```
        If (global_nsn > node.nsn)
           Select the most appropriate node from its sibling
           nodes ;
        If (currentlevel == leaflevel)
           Exist Loop
        Push [node, node.nsn] into path ;
    }
}
```

(그림 8) 삽입 알고리즘 - FindNode

● 삽입 알고리즘-SplitNode

(그림 9)의 SplitNode는 단말노드에 넘침이 발생했을 때 처음 호출된다. SplitNode는 먼저 단말노드의 공유래치를 해제하고 배타래치를 획득한다. 배타래치를 획득하면 바로 leafnode를 분할한다. 분할로 생성된 새로운 노드 newnode에 대해서 배타잠금을 획득한다. 분할된 내용을 색인 트리를 거슬러 올라가면서 반영할 때 leafnode와 newnode의 배타잠금은 SplitNode가 끝날 때까지 유지한다. leafnode의 분할을 끝내면 다시 부모노드(parentnode)에 배타잠금과 공유래치를 획득한 후에 newnode에 대한 엔트리(internalentry)를 삽입한다. 또한, parentnode에서 node에 대한 엔트리의 MBR을 변경한다. 이때 parentnode에 internalentry를 삽입할 수 있는 여유공간이 없으면 이상의 분할과정을 반복한다. 분할은 조상노드들중 새로운 엔트리를 삽입할 공간이 있는 노드를 만나거나 그렇지 않으면 루트를 분할하고 끝나게 된다. 비 단말노드를 분할한 후에는 그 노드의 부모노드에 배타잠금을 획득하고 분할한 노드와 분할로 생성된 노드에 대한 배타잠금을 해제한다.

SplitNode는 분할차원과 위치를 계산하는 동안에는 공유래치와 배타잠금을 획득한다. 이 단계가 끝나면 공유래치를 해제하고 배타래치를 획득한 후 노드를 물리적으로 분할한다. 또한, 이미 간단히 설명한 것처럼 SplitNode는 트리잠금을 획득한 후 분할을 수행한다. 이 때문에 SplitNode는 직렬로 수행된다.

```
FixMBR (Node node, PathStack path)
{
   parentnode := POP (path)   //decide the parent of node ;
   If (MBR Shrinking)         //delete operations
   {
      Obtain X-lock and S-latch on parentnode ;
      Release X-lock on node ;
   }
   else
   {
      Obtain X-lock and S-latch on parentnode ;
   }
   Modify the MBR for node in parent parentnode ;
   Release S-latch on parentnode ;
   If (the MBR of parentnode is changed)
   {
      If (MBR Expansion)
         Release X-lock on parentnode ;
      FixMBR (parentnode, path) ;
   }
}
```

(그림 9) 삽입 알고리즘 - FixMBR

● 삽입 알고리즘-FixMBR

(그림 10)의 FixMBR은 단말노드에 새로운 엔트리가 삽입되거나 삭제되어 변경된 MBR을 전파하기 위해서 호출된다. MBR 축소의 경우 잠금 결합을 수행한다. 앞서 설명했던 내용들은 의사코드에서는 표현하지 않았다.

```
SplitNode (Entry leafentry, Node node, PathStack path)
{
   Compute split dimension and split position of node ;
   newnode := allocate a new node ;
   Obtain X-lock and X-latch on newnode ;
   entries := with split dimension and split position,
      select entries from node to be moved to newnode ;
   Copy entries to newnode ;

   If (node is not latched in exclusive mode)
      Release S-latch of node and obtain X-latch on node ;

   Delete entries from node and reorganize node ;
   Copy sibling pointer of node to newnode
      and set sibling pointer of node to newnode ;
   Copy node.nsn of node to newnode ;
   Increase global_nsn and install its value as the node.nsn ;
   Create an internal entry internalentry [newnode, MBR] ;
   Release X-latch of node and newnode ;
   parentnode := POP (path) ;
   Obtain S-latch and X-lock on parentnode, and modify the
      MBR for node in parentnode ;

   If (node is not leaf node)
      Release X-lock of node and newnode ;

   If (overflow occurred in parentnode due to internalentry)
   {
      Release S-latch and obtain X-latch on parentnode ;
      SplitNode (internalentry, parentnode, path) ;
   }
   Add internalentry to parentnode. ;
   Release X-latch on parentnode ;
   If (the MBR of parentnode is changed)
   {
      Release X-latch on parentnode ;
      FixMBR (parentnode, path) ;
   }
}
```

(그림 10) 삽입 알고리즘-SplitNode

3.5 탐색 연산

기본적인 탐색 알고리즘은 [10, 11]과 같다. 다른 점이 있다면 제안하는 방법은 노드에 공유래치만을 획득하고 래치 결합을 수행하지 않는다는 것이다. 탐색 연산은 노드분할시와 MBR 변경시 노드에 여유공간이 없는 경우에만 지연된다. 실험 결과로 보여주겠지만 이런 특징은 탐색 연산의 지연시간을 매우 크게 줄인다.

3.6 삭제 연산

제안하는 동시성제어 알고리즘의 삭제 연산은 삽입 연산과 유사하게 두 단계로 수행된다. 첫 번째 단계는 삭제할 엔트리를 포함하고 있는 단말노드를 찾는 것이다. 단말노드를 찾기 위해서는 탐색연산의 exact 매치와 같은 방법으로 트리를 순회하게 된다. 단말노드를 찾고 엔트리를 삭제한 후에 두 번째 단계가 수행된다. 삭제로 인해서 노드의 MBR이 변경되면 FixMBR을 호출한다. FixMBR을 호출할 때 삭제자는 잠금 결합을 수행하기 위해서 노드의 배타잠금을 해제하지 않는다. 또한, 회복 목적을 위해서 엔트리 삭제를 수행할 때 공유 트리잠금을 획득한다. 이에 대한 자세한 이유는 6장에서 설명한다.

엔트리들은 물리적으로 삭제되고 삭제로 MBR이 축소되면 조상노드에 이를 물리적으로 반영한다. 만일 노드가 비게 되면 저장공간 활용도를 높이기 위해서 바로 해당 노드를 삭제한다. 그러나, 노드 삭제는 트리 순회자들이 이미 삭제되어 없는 노드에 대한 포인터를 따라가게 되는 문제를 발생시킨다. 이 문제를 해결하기 위해서 노드 포인터와 노드 구조를 변경한다. 노드의 헤더영역에 reuse_counter를 추가한다. 그리고 노드포인터에 reuse_counter를 추가한다. 즉, 노드 포인터는 원래의 노드 포인터와 reuse_counter를 갖게 된다. 노드의 헤더영역에 저장한 reuse_counter는 노드가 최초에 생성되는 시점에서 0로 초기화되며 노드가 할당되어 사용될 때마다 1씩 증가한다. 순회자들은 현재 노드의 reuse_counter가 해당 노드에 대한 노드 포인터의 reuse_counter와 같은지 확인한다. 만일 틀리면 순회자들은 가장 낮은 정확한 조상노드에서 순회를 다시 시작한다.

4. 정확성 증명

[14]에 의하면 색인구조를 위한 동시성제어 알고리즘의 정확성을 증명하기 위해서는 다음의 두 가지 명제를 증명해야 한다. 첫째 교착상태가 발생하지 않는다. 두 번째 색인구조의 모든 변경연산은 색인구조의 일관성을 보존하며 동시에 수행되는 다른 프로세스들은 항상 일관된 트리를 접근한다. 다음부터 이 두 가지에 대한 증명을 한다.

4.1 교착 상태

먼저 제안하는 알고리즘이 교착상태에 빠지지 않는다는 것을 증명한다. 제안하는 알고리즘에서 교착상태는 래치간, 잠금간, 또는 래치와 잠금간에 발생할 수 있다. 그러므로 이 세 가지 경우에 대해서 모두 교착상태가 발생하지 않음을 보여야 한다. 일반적으로 교착상태는 상호배제, 잠금의 유지 및 대기, 비선점, 환형 대기의 네 가지 조건이 동시에 만족될 때 발생한다. 상호배제는 최소한 하나의 노드는 배타잠금이 설정이 되어있는 것을 말한다. 유지 및 대기는 어떤 노드에 대해서 잠금을 유지하는 트랜잭션이 있고 서로 호환이 안되는 모드로 이 노드에 잠금을 요청하는 트랜잭션이 있는 것을 말한다. 비 선점이란 노드가 다른 트랜잭션에 의해서 잠금이 설정된 상태에서 다른 트랜잭션은 이 잠금이 해제 될 때까지 노드를 접근할 수 없음을 말한다. 마지막으로 환형대기란 트랜잭션 집합 $\{T_0, T_1, \cdots, T_n\}$ 존재할 때, T_0는 T_1에 의해 잠금이 설정된 노드에 잠금을 요청하고, 다시 T_1은 T_2에 의해서 잠금이 설정된 노드에 잠금을 요청하고, $\cdots$, T_{n-1}은 T_n에 의해서 잠금이 설정된 노드에 잠금을 요청하고, T_n은 T_0에 의해서 잠금이 설정된 노드에 대해서 잠금을 요청하는 상태를 말한다.

보조정리 4 : 래치간에는 어떠한 교착상태도 발생하지 않는다.

증명 : 제안하는 알고리즘에서 삽입자, 탐색자, 삭제자 모두 한번에 하나의 래치만을 획득한다. 이것은 래치간에는 유지 및 대기의 경우가 발생하지 않는 다는 것을 의미한다. 따라서 래치간에는 절대로 교착상태가 발생하지 않는다.

보조정리 5 : 잠금간에는 어떠한 교착상태도 발생하지 않는다.

증명 : 제안하는 알고리즘에서 하나의 트랜잭션은 분할시와 MBR 축소시 최대 두 개의 노드에 동시에 잠금을 유지한다. 그러나 이 경우에 잠금을 획득하는 방향이 항상 아래에서 위쪽 그리고 왼쪽에서 오른쪽을 향한다. 위에서 아래로 순회하면서는 반드시 동시에 획득되는 잠금은 하나이다. 따라서 제안하는 알고리즘에서는 환형대기 상태가 발생하지 않는다. 따라서 잠금간에는 절대로 교착상태가 발생하지 않는다.

보조정리 6 : 잠금과 래치간에는 교착상태가 발생하지 않는다.

증명 : 제안하는 알고리즘에서는 트랜잭션이 특정 노드에 대해서 잠금과 래치를 동시에 획득할때는 항상 잠금을 먼저 획득한 후에 래치를 획득한다. 또한, 래치는 위에서 아래, 아래에서 위로 획득이 가능하지만 잠금은 항상 아래에서 위의 방향으로 획득한다. 따라서 잠금과 래치간에는 환형대기 상태가 발생할 수 없다. 이로부터 잠금과 래치간에는 절대로 교착상태가 발생할 수 없음을 알수 있다.

정리 1 : 제안하는 동시성제어 알고리즘에서는 교착상태가 발생하지 않는다.

증명 : 보조정리 4, 5, 6으로부터 제안하는 알고리즘은 교착상태가 발생하지 않음을 알 수 있다.

4.2 트리구조의 일관성

삽입자가 색인구조를 변경하는 동안에도 다른 연산들이 정확하게 수행될 수 있다는 것을 보여준다. 이를 보여주기

위해서 먼저 제안하는 알고리즘의 모든 연산들을 고려해 본다. 제안하는 알고리즘에서 발생할 수 있는 연산의 종류는 순회(Traverse), 탐색(Search), 변경(Update), 분할(sPlit) 그리고 추가(Add)이다. 순회란 삽입자가 엔트리를 삽입할 적당한 노드를 찾거나 삭제자가 삭제할 엔트리를 탐색하는 과정이다. 사실 순회는 탐색과 같은 특징을 가지므로 순회와 탐색을 순회 하나로 고려하고 설명한다.

정리 2 : 제안하는 알고리즘의 모든 연산들은 동시에 수행되는 다른 트랜잭션이 정확하게 동작하는 것을 보장한다.

증명 : 정의를 증명하기 위해서는 위에서 제시한 네 가지 연산들의 상호작용이 정확히 수행됨을 보여야 한다. 위의 네가지 연산이 가능한 상호작용에는 T-T, T-U, T-P, T-A, U-U, U-P, U-A, P-P, P-A 그리고 A-A의 총 10가지가 있다. 지금부터 이 10가지 상호작용이 정확히 수행됨을 보인다.

4.2.1 상호작용 : T-T

순회는 색인트리의 아무것도 변경하지 않으므로 이 경우는 무시할 수 있다.

4.2.2 상호작용 : T-U

이 상호작용은 항상 비 단말 노드에서만 발생한다. U 가 t_0에 노드 n에서 수행되고 있다고 가정하자. 그리고, t'에 T가 노드 n을 읽는다고 생각해 보자. 이때 다음과 같은 세가지 경우가 발생한다.

① Case 1 : $t_0 > t'$

U는 n의 한 엔트리의 MBR을 변경한다. T는 다음 방문할 노드를 선택하기 위해서 n을 읽는다. 이때 보조정리 1에 의해서 U는 트리 구조의 일관성을 보존하므로 t'에 트리는 정확하다. 그러므로 T는 정확한 노드 n을 읽는다.

② Case 2 : $t_0 < t'$

T는 n을 읽고 다음 방문할 노드를 결정한다. 다음에, U는 n의 MBR을 변경한다. T는 일관된 노드 n을 읽고 U는 n의 MBR을 정상적으로 변경한다.

③ Case 3 : $t_0 == t'$

T는 n에 공유래치를 획득한후 접근하고 U는 배타잠금과 공유잠금을 획득하고 접근하므로 T는 U에의해서 변경되고 있는 MBR을 읽을 수 있다. 그러나, 보조정리 2에 의해서 T는 항상 완전히 변경되거나 변경되기 전의 MBR을 읽는다. 그러므로, Case 3는 Case 1이나 Case 2중 하나가 된다.

4.2.3 상호작용 : T-P

P가 t'에 노드 n에서 수행되고 있고 t_0에 T가 노드 n을 읽고 있다고 가정해보자.

① Case 1 : $t_0 < t'$

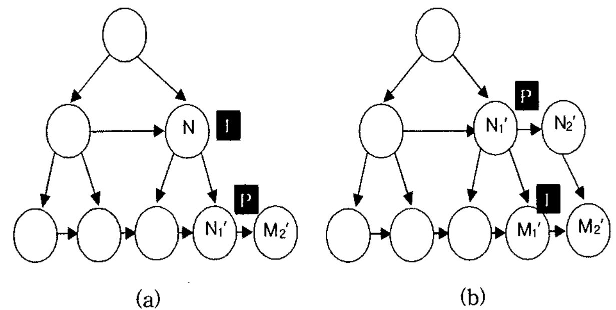
n이 단말노드이면 T는 노드 n을 정확하게 읽고 P는 n을 n1'과 n2'으로 분할한다. 이 경우 P는 T에 전혀 영향을 미치지 않는다. 하지만, (그림 11)에서 보는 것처럼 만일 n이 비단말 노드이면 n의 자식노드 m이 m1'과 m2'로 분할되어서 새로운 MBR과 m에 대한 포인터 쌍이 엔트리로 n에 삽입되는 과정에서 n에 넘침이 발생한 것이다. T는 노드 n에 m에 대한 엔트리가 추가되기 전에 n을 읽게된다. (그림 11)(b)에서 처럼 T는 계속해서 트리를 순회하게 되고 m에 도착했을 때 m1'을 읽게 되고 이미 설명한데로 오른쪽 링크를 따라서 m2'으로 이동해서 무사히 순회를 계속하게 된다.

② Case 2 : $t_0 > t'$

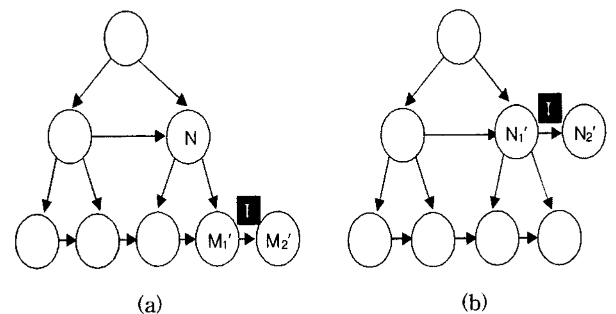
(그림 12)에서 처럼 P는 노드 n을 n_1'과 n_2'으로 분할한다. 후에 T가 노드 n을 방문한다 하지만 n은 이미 n_1' 과 n_2'으로 분할된 후이기 때문에 T는 n_1'을 읽게 된다. 다시 트리 순회를 정확하게 하기 위해서는 T는 n의 모든 엔트리들을 보아야 한다. (그림 12)(b)에서 처럼 T는 오른쪽 링크를 따라서 n_2'에 도착하게 되고 이전 n에 있던 모든 엔트리와 새로 삽입된 엔트리들을 읽고 무사히 순회를 계속할 수 있다.

③ Case 3 : $t_0 == t'$

실제로 P가 물리적으로 노드 n을 분할하는 시점에서 T는 n을 접근할 수 없다. 따라서 이 경우는 Case 1이거나 Case 2와 같다.



(그림 11) 분할과 순회의 상호작용(Case 1)



(그림 12) 분할과 순회의 상호작용(Case 2)

4.2.4 상호작용 : T-A

A가 t_0에 노드 n에서 수행되고 있고 t'에 T가 노드 n을 읽는다고 가정하자.

① Case 1 : $t_0 < t'$

A가 새로운 엔트리를 노드 n에 추가하는 것은 트리의 일관성을 보존한다. 그러므로 T는 새롭게 삽입된 엔트리를 포함하는 정확한 노드 n을 읽게 되며 순회를 정상적으로 계속할 수 있다.

② Case 2 : $t_0 > t'$

n이 단말노드이면 T와 A 사이에는 상호작용이 없다. 그러나 n이 비단말 노드이면 A는 새로운엔트리를 n에 삽입하기 전에 n의 자식노드인 m을 m_1'과 m_2'으로 분할했고 분할로 생성된 노드에 대한 엔트리를 n에 삽입하려하는 것이다. T는 m_2'에 대한 엔트리가 n에 삽입되기 전에 n을 읽으므로 T가 m을 방문했을때는 m_1'을 방문하게 되며 T는 오른쪽 링크를 따라서 m_2'에 도착하게 되고 정상적으로 다음 순회를 계속 할 수 있게 된다.

③ Case 2 : $t_0 == t'$

보조정리 3에 의해서 T는 정확한 노드 n을 읽게 된다.

4.2.5 상호작용 : U-U

보조정리 1에 의해서 이 상호작용은 정확하게 수행됨을 알 수 있다.

4.2.6 상호작용 : U-P

U는 t0에 노드 n에서 수행하고 있고 t'에 P가 노드 n에서 수행되고 있다고 가정하자. 두 연산은 모두 트리 구조를 거슬러 올라가는 연산들이다.

① Case 1 : $t_0 > t'$

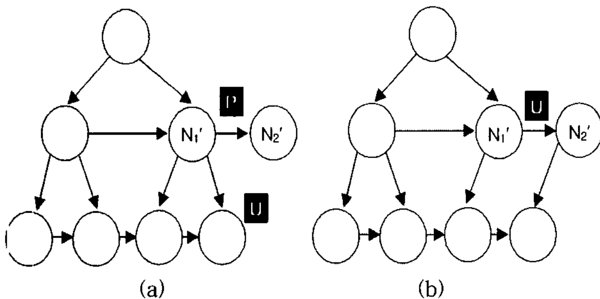
U는 트리의 일관성을 보존하므로 t'에서 트리는 일관된다. 그리고, P는 정상적으로 노드 n을 분할한다.

② Case 2 : $t_0 < t'$

(그림 13)에서 처럼 노드 n이 n1'과 n2'으로 분할되고 U가 도착하는 노드는 n1'이 된다. 그러나 (그림 13)(b)에서 처럼 U는 오른쪽 링크를 따라서 n1'과 n2'을 모두 읽고 정상적인 U를 수행한다.

③ Case 3 : $t_0 == t'$

P와 U는 모두 배타잠금을 노드에 획득한후 연산을 수행하기 때문에 두연산이 동시에 같은 노드에서 작업을 할 수 없다. 항상 Case 1과 Case 2 뿐이다.



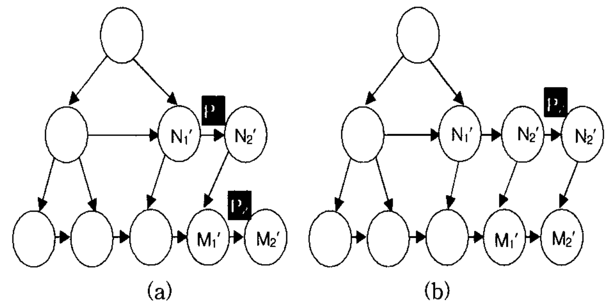
(그림 13) 분할과 MBR 변경의 상호작용

4.2.7 상호작용 : U-A

두 연산모두 트리를 거슬러 올라가면서 작업을 수행한다. 또한, U와 A는 항상 트리구조의 일관성을 보존하므로 U와 A는 서로 영향을 미치지 않는다.

4.2.8 상호작용 : P-P

(그림 14)에서 보는 것처럼 t0에 P가 n을 n_1'과 n_2'으로 분할하고 또 다른 P는 경로스택의 노드 n을 보고 t_1에 노드 n1'에 도착하는 경우만 존재한다. P가 t'에서 노드 n을 접근하려 하지만 n_1'에 도착하게 되는데 이것은 (그림 14)(b)에서 처럼 n_1'의 오른쪽 링크를 따라가면서 이전 n에 있던 모든 엔트리를 볼수 있게 된다. 따라서 정상적인 분할을 수행할 수 있게 된다.



(그림 14) 분할과 분할의 상호작용

4.2.9 상호작용 : P-A

이 경우는 P-U 상호작용과 동일한 경우이다.

4.2.10 상호작용 : A-A

이 경우는 U-U 상호작용과 동일한 경우이다.

이상의 10개의 상호작용이 일관되게 수행된다. 그러므로 제안하는 알고리즘의 연산들의 동시 수행은 색인 트리의 일관성을 보존한다.

5. 회 복

이 논문에서 제안하는 회복기법은 [10]과 [11]에서 처럼 WAL (Write Ahead Logging)을 기반으로 페이지 지향 재수행(Redo)와 논리적 복구(Undo)를 수행하는 환경을 기반으로 한다. 또한, 높은 동시성을 얻기 위해서 변경 연산을 내용변경연산과 구조변경 연산으로 나누었다. 단말 노드에서의 엔트리 추가/삭제는 내용 변경연산이고 노드 분할, MBR 변경, 노드 삭제는 구조 변경 연산이다.

단일 엔트리의 삭제나 삽입을 통해 단말노드가 내용이 변경될 때는 해당 연산을 한 트랜잭션의 일부로 WAL 규약에 맞추어서 로깅을 수행한다. 이것은 트랜잭션이 취소되면 복구를 수행할 수 있고 트랜잭션이 완료되었을 때는 재수행을 수행할 수 있게 한다. 반면에 트리구조 변경연산은 별개의 회복가능한 단위인 NTA(Nested Top Action)로 처리한

다. 예를 들어서 트랜잭션이 꽉찬 노드에 엔트리하나를 삽입하는 과정에서 발생하는 노드의 분할은 NTA이다. NTA는 이를 포함하는 트랜잭션과는 별도로 로깅되고 회복된다. 또한, NTA는 해당 연산이 종료함에 따라서 바로 CLR(Compensation Log Record)[2, 3]를 이용해서 완료된다. 그리고 이를 포함하는 트랜잭션이 실패하더라도 철회되지 않는다.

구조변경 연산을 다음과 같이 정의한다.

① 노드 분할 : 트리를 거슬러 올라가면서 반복적으로 수행이 되며 조상노드들의 분할과 새로운 노드에 대한 엔트리를 부모노드에 삽입하는 것까지 포함한다.
② MBR 변경 : 단일 조상노드에 대한 MBR 변경을 말한다.
③ 노드 삭제 : 노드의 삭제와 이에 대한 부모노드의 엔트리 삭제를 포함한다.

모든 구조변경 연산들은 페이지 지향형태로 복구한다. 반면에 단말노드에서의 엔트리 삽입/삭제는 논리적으로 복구한다. 즉, 삭제의 경우 트리를 루트부터 순회하여 대상 엔트리를 찾고 이에 대한 복구를 수행한다. 삽입의 경우 삽입한 엔트리를 가지고 있는 노드를 접근하고 필요에 따라서는 오른쪽 링크를 따라가면서 해당 엔트리를 찾아서 복구를 수행한다. 오른쪽 링크를 따라가야 하는 이유는 엔트리의 삽입을 수행한 후 단말노드에 대한 잠금을 해제하고 트랜잭션이 완료되기 전에 트리구조가 변경될 수도 있기 때문이다. 즉, 엔트리를 삽입한 노드가 분할이 되어서 엔트리의 일부가 오른쪽 노드로 이동할 수 있다. 이를 찾기 위해서는 오른쪽 링크를 따라가야 한다.

시스템 고장 이후 복구 회복을 수행할 때 이전에 획득했던 모든 노드 래치가 해제된다. 그리고, 아직 끝나지 않은 구조변경 연산이 남아 있을 수도 있다. 그러므로 이런 상황에서는 단말엔트리 삽입이나 삭제에 대한 논리적 복구는 절대로 또 다른 구조 변경 연산을 일으켜서는 안되고 트리를 루트부터 순회해서 논리적 복구를 수행해서는 안된다.

위와 같은 요구조건을 만족하기 위해서 제안하는 회복 프로토콜은 다음과 같은 작업을 수행한다. 단말 엔트리 삽입의 복구는 항상 물리적으로 수행한다. 노드가 비게 되더라도 노드 삭제를 수행하지 않는다. 제안하는 방법에서는 트리를 루트부터 순회하지 않고 복구를 위해 삭제할 엔트리를 찾을 수 있다. 왜냐하면 엔트리들은 분할에 의해서 항상 오른쪽 노드로만 이동되며 오른쪽 링크를 계속 따라가면 원하는 엔트리를 찾을 수 있기 때문이다. 삽입한 엔트리에 대한 복구를 수행할 때 삭제한 후에 조상노드들에 변경된 MBR을 반영하지 않아도 탐색이 항상 정확하게 이루어질 수 있다.

하지만 엔트리 삭제에 대한 복구는 반드시 루트부터 트리를 순회해서 루트부터 단말노드까지의 경로를 획득해야 한다. 왜냐하면 삭제에 대한 복구를 수행해서 엔트리를 삽입하면 반드시 조상노드들에 MBR을 적절히 확장해줘야 한다. 만일 확장하지 않으면 탐색이 정확하게 이루어질 수 없다. 그러므로 삭제자들은 반드시 삭제를 수행하기 전에 공유모드의 트리잠금을 획득한다. 노드분할은 분할을 수행하기 전에 배타모드의 트리잠금을 획득하므로 엔트리 삭제에 대한 로그는 항상 노드분할에 대한 로그가 완전히 쓰여진 후 기록된다. 이것은 엔트리 삭제의 복구는 항상 색인 트리가 일관된 상태에서 수행됨을 보장한다.

마지막으로, 노드분할을 수행할 때 분할할 단말노드와 새롭게 생성된 노드에 대한 배타잠금을 분할이 모두 종료될 때까지 설정한다. 이것은 다른 삽입자들이 엔트리들을 분할에 참여하는 노드에 추가할 수 없게 한다. 분할이 종료전에 다른 엔트리가 삽입되면 분할을 수행하는 트랜잭션이 철회할 때 삽입된 엔트리들을 잃게 된다. 또한, 여러 개의 완료되지 못한 트리구조 변경을 트리에 허용하지 않기 위해서 노드 분할을 배타 모드의 트리잠금을 이용해서 직렬화 한다. 이 트리잠금은 삭제자의 회복을 일관되게하는 역할도 수행한다.

6. 성능 평가

제안하는 알고리즘(RPLC)의 성능을 평가하기 위해서 GiST를 위한 동시성제어 알고리즘(CGiST)[10]과 비교를 수행한다. 사실상 [19]와 [20]이 가장 최근의 CGiST를 위해 제안된 동시성제어 알고리즘이다. 하지만 이들은 유령현상을 방지하는데 초점이 맞추어져 있다. 그러나, 이미 언급한 것처럼 제안하는 알고리즘의 주목적은 반복 읽기(Repeatable Read) 고립 수준에서 동시성제어 알고리즘의 성능을 향상시키고 적절한 회복기법을 제안하고자 하는 것이다. 이런 이유로 동시성제어 기법의 두 가지 측면을 모두 고려하고 있는 CGiST를 선택했다.

6.1 실험 환경

6.1.1 구 현

이 논문에서는 두 가지 알고리즘을 BADA DBMS[9]의 저장 시스템인 MiDAS의 CIR-트리 관리자의 동시성제어 기법으로 구현하였다. 이미 언급한 것처럼 제안하는 알고리즘은 유령현상 방지보다는 성능을 향상시키는데 초점이 맞추어져 있다. 그러므로 실험의 공정성을 위해서 CGiST의 유령현상 방지를 위한 부분은 생략하고 구현하였다. CGiST는 술부 잠금과 데이터 레코드에 대한 2단계 잠금을 적절히 조합한 혼합 기법을 제시하고 있다. 술부를 전역적으로 관리하지 않고 색인구조의 각 노드에서 관리한다. 탐색자들은 탐색 술부와 겹치는 노드에 술부를 추가하고 술부 잠금을 설정한다. 각 노드에 추가된 술부는 노드분할이나 MBR 변경시에 같이 변경이 되어야 한다. 삽입자들은 노드에 추가된 술부에 의해서 지연된다. 이 논문에서는 위와 같은 유령현

상 방지에 해당하는 부분을 모두 제거하고 구현하였다. 또한, 노드 삭제에 의한 잘못된 포인터를 따라가는 것을 회피하기 위한 신호 잠금기법 역시 CGiST 구현에서 제외했다. 두 동시성제어 알고리즘은 MiDAS에서 제공하는 잠금, 래치, 로깅 API를 이용해서 구현되었다.

6.1.2 데이터 집합

두 개의 데이터 집합이 실험에 사용되었다. 하나는 10차원의 가상 데이터 집합이고 다른 하나는 9차원의 실 데이터 집합이다. 가상 데이터 집합은 50000개의 특징벡터를 가지고 있고 균등분포의 특성을 갖는다. 실 데이터 집합은 뉴스 비디오의 대표 프레임으로부터 컬러 히스토그램을 이용해 추출하였으며 총 200000개의 데이터를 가지고 실험을 수행했다.

색인구조의 가장 중요한 파라미터중 하나는 노드의 크기이다. 노드의 크기에 따라서 색인구조의 성능이 변하게 된다. 이 논문에서는 4Kbytes와 16Kbytes로 변화시키면서 실험을 수행하였다. 두 실험에서 RPLC가 모두 CGiST보다 성능이 좋게 나왔다. 전체적인 성능은 노드 사이즈가 16Kbytes일 때 보다 더 낳았다. 또한, 두 알고리즘의 성능 차이도 노드크기가 16Kbytes일 때 10% 정도 더 증가하였다. 성능차이가 더 나게된 이유는 충돌이 증가하였기 때문이다.

6.1.3 작업부하와 파라미터

최초에 CIR-트리는 벌크로딩 기법에 의해서 구성된다. 이어서 다중 프로세스가 특징 벡터들을 특정 작업부하 아래에서 하나씩 삽입한다. <표 1>은 작업부하 파라미터들을 보여준다. 입력 파라미터에 따라서, 작업부하 생성기는 탐색과 삽입 프로세스의 개수, 동시수행 프로세스의 개수, 색인트리를 구축할 초기 특징벡터의 개수 그리고 K-NN 질의의 K의 수를 결정한다. 다시 작업부하 생성기는 결정된 값들을 C 언어와 MiDAS API로 쓰여진 드라이버 프로그램에 결정된 값을 넘긴다. 드라이버는 탐색과 삽입 프로세스를 동시에 수행시킨다. 질의에 사용될 특징벡터는 이미 색인구조에 삽입된 것들 중에서 무작위로 선택하고 삽입할 특징벡터는 데이터집합에서 선택한다. 각각의 프로세스는 여러 개의 트랜잭션을 수행시킨다.

MiDAS를 초기화할 때 버퍼풀의 버퍼개수를 100으로 하였다. 실험에서 사용된 플랫폼은 Ultra Sparc 프로세서에 128 MBytes 주기억장치를 장착하고 있으며 운영체제로 Solaris 2.5를 사용한다.

〈표 1〉 작업부하 파라미터와 값

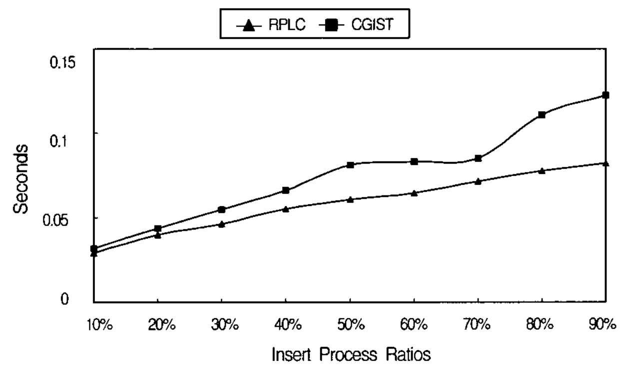
파라미터	값
특징벡터의 개수	50,000~200,000
삽입자의 비율	0%~100%
K-NN 질의에서 K	1, 5, 10
동시수행되는 프로세스개수(MPL)	50, 100

6.2 실험 결과

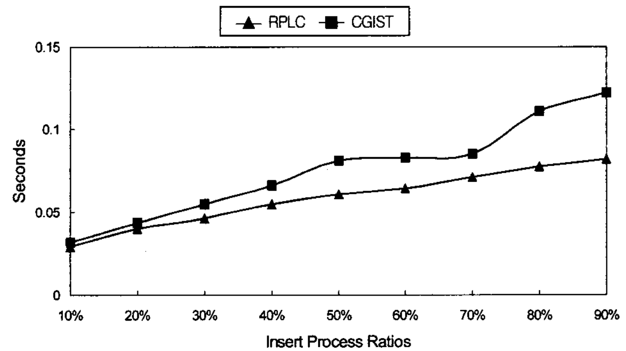
먼저 삽입 연산의 비율을 10~90%로 변화시키면서 응답시간과 처리율을 측정한 결과를 보여준다. 실험은 균등분포의 가상데이터와 실데이터 집합에 대해서 모두 수행하였다. 그 다음에 MPL(Multi Programming Level)과 데이터베이스의 크기를 변화시키면서 측정한 결과를 보인다.

6.2.1 균등 분포를 갖는 가상 데이터

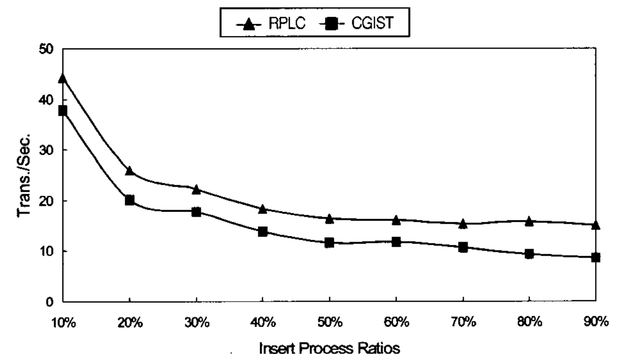
(그림 15)~(그림 18)는 CGiST와 RPLC의 성능을 삽입과 탐색 프로세스의 비율을 변화시키면서 측정한 응답시간과



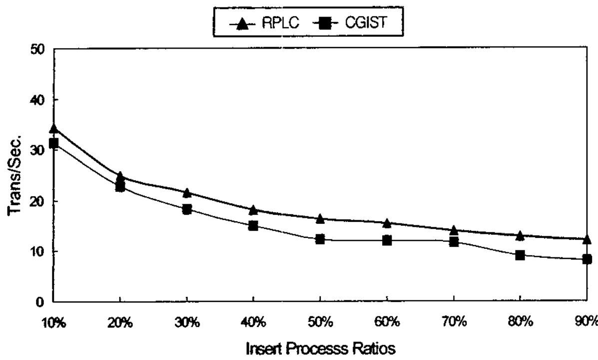
(그림 15) 탐색자의 응답시간(K = 5, database size = 100K, MPL = 50)



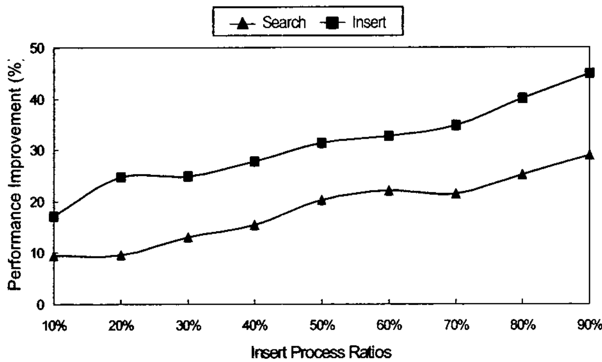
(그림 16) 삽입자의 응답시간(K = 5, database size = 100K, MPL = 50)



(그림 17) 탐색자의 처리율(K = 5, database size = 100K, MPL = 50)



(그림 18) 삽입자의 처리율(K = 5, database size = 100K, MPL = 50)



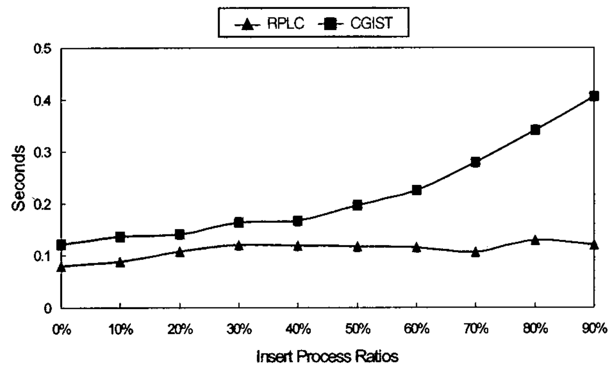
(그림 19) 성능 증가율(K = 5, database size = 100K, MPL = 50)

처리율을 보여주고 있다. (그림 11)은 삽입 프로세스의 비율이 10~90%으로 변할 때 탐색연산의 응답시간을 보여준다. RPLC의 응답시간은 삽입 프로세스의 비율이 증가해도 CGiST의 응답시간 보다 천천히 증가한다. (그림 16)는 같은 조건에서 삽입 연산의 응답시간을 보여준다. (그림 15)에서의 탐색 프로세스의 응답시간처럼 RPLC가 CGiST의 응답시간보다 훨씬 높게 나타난다. (그림 17)과 (그림 18)는 탐색 프로세스와 삽입 프로세스의 처리율을 보여주고 있다. RPLC에서 탐색 프로세스의 처리율이 삽입 프로세스의 비율의 증가에 따라서 서서히 감소하는 반면 CGiST의 처리율은 급격하게 감소한다.

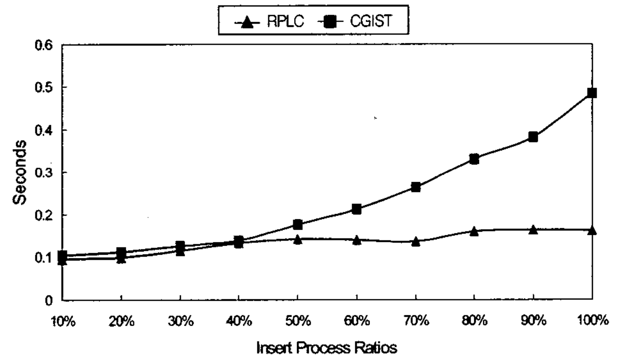
(그림 19)가 보여주는 것처럼 균등분포 데이터 집합에 대해서 RPLC는 약 25%의 성능향상을 보이고 있다. RPLC가 CGiST보다 성능이 높아지는 이유는 명확하다. 제안하는 알고리즘에서 탐색자는 거의 물리적인 노드분할 중에만 지연된다. 심지어 삽입자가 노드의 엔트리가 삽입될 때도 탐색자들은 지연되지 않는다. 또한, 삽입자들은 대부분의 경우에 잠금 결합을 수행하지 않는다. 반면에 CGiST에서는 탐색자들은 MBR 변경중, 노드 분할, 엔트리 삽입시에 지연된다. 또한, MBR 변경과 노드 분할은 트리의 여러 높이의 여러 노드에 배타 잠금을 획득한다. 이런 점은 탐색과 삽입 연산의 성능을 크게 저하시키는 요인이 된다. RPLC의 경우에도 분할 연산중에 래치의 해제 및 획득이 매우 빈번히 발생하게 되는데 이점은 전체적인 성능을 떨어뜨리는 요인이 될 수 있다.

6.2.2 실 데이터

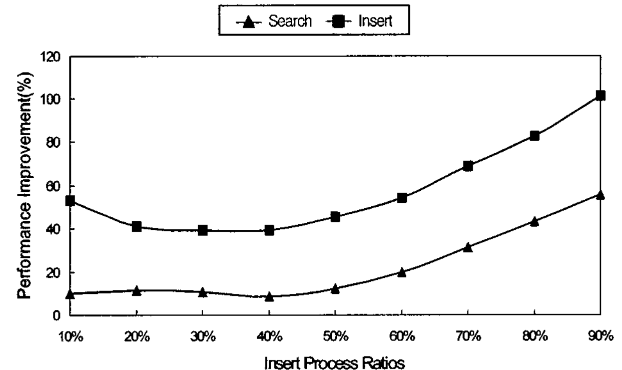
(그림 20)~(그림 23)는 실 데이터 집합에 대한 성능평가 결과를 보여준다. 그림에서 볼 수 있는 것처럼 CGiST와 RPLC 사이의 성능차이가 균등분포의 가상데이터 보다 훨씬 큼을 알 수 있다. (그림 24)는 RPLC가 실 데이터 집합에 대해서 약 85% 정도의 성능향상이 있음을 보여주고 있다. RPLC



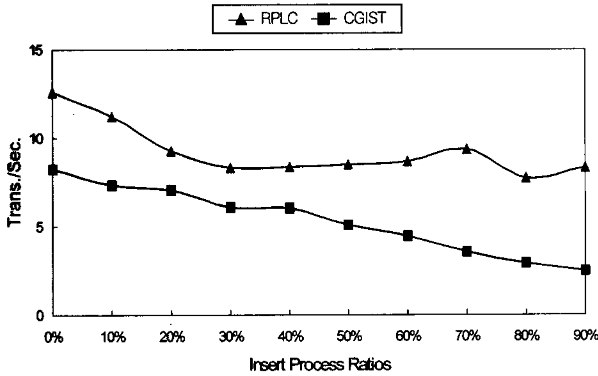
(그림 20) 탐색자의 응답시간(K = 5, database size = 100K, MPL = 50)



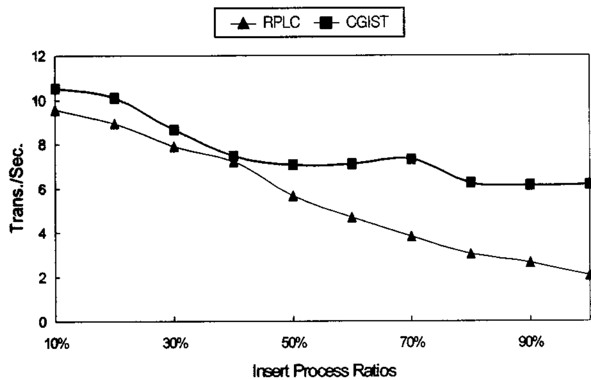
(그림 21) 삽입자의 응답시간(K = 5, database size = 100K, MPL = 50)



(그림 22) 탐색자의 처리율(K = 5, database size = 100K, MPL = 50)



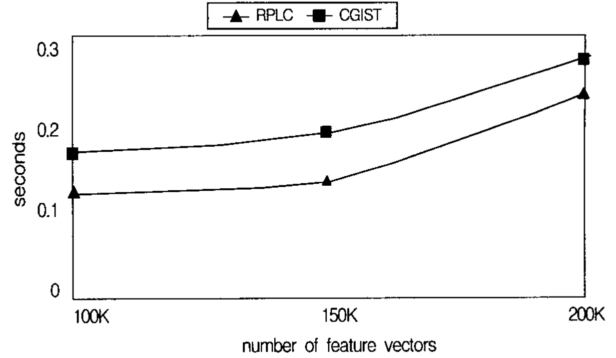
(그림 23) 삽입자의 처리율(K = 5, database size = 100K, MPL = 50)



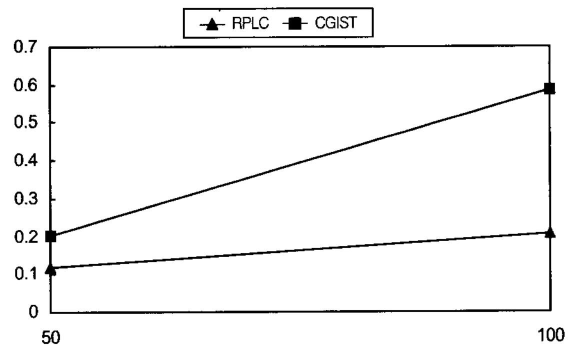
(그림 24) 성능 향상율(K = 5, database size = 100K, MPL = 50)

가 CGiST보다 성능이 왜 좋은지에 대해서는 이미 6.2.1에서 설명한 바 있다. 하지만 여기에서는 실 데이터 집합에 대해서 성능평가를 실시한 결과 균등분포에 대한 성능평가 결과보다 훨씬 더 그 성능차이가 큰 것에 대해서 관심을 가질 필요가 있다. 실험에 사용한 실 데이터 집합은 비디오 데이터의 대표 프레임으로부터 특징을 추출한 것으로써 비디오의 특성상 매우 불균등한 분포를 갖는 데이터이다. 이것은 실 데이터 집합에서 잠금 및 래치 충돌이 더 증가한다는 것을 의미한다. 종합해 보면, 제안하는 방법은 충돌이 많을수록 CGiST 보다 우수한 성능을 낸다는 것을 의미한다.

(그림 25)와 (그림 26)은 두 알고리즘이 데이터베이스의 크기와 MPL의 변화에 대해서 어떤 성능을 보이는가를 보여주고 있다. 데이터베이스의 크기가 커지는 것은 두 알고리즘의 성능차이에는 큰 영향을 미치지 않는다. 그러나, 전체적인 성능은 데이터베이스의 크기가 증가함에 따라서 감소한다. 이 이유는 노드의 개수와 트리의 높이가 증가하면서 전체적인 색인구조의 성능이 저하되기 때문이다. 특히 데이터베이스의 크기가 200K 일때 성능이 저하가 큰 것을 볼 수 있는데 이것은 150k~200k 사이에 트리의 높이가 증가한 것으로 볼 수 있다.



(그림 25) 특징벡터 수의 변화에 따른 탐색자의 응답시간(insert ratios = 40%, K = 5)



(그림 26) MPL의 변화에 따른 탐색자의 응답시간(insert ratios = 40%, K = 5)

7. 결 론

이 논문에서는 다차원 색인구조를 위한 동시성제어 알고리즘과 회복기법을 제안하였다. 제안하는 알고리즘의 특징을 살펴보면 다음과 같다. 첫 번째, 새로운 MBR 변경 기법을 제안하였다. 이 기법은 MBR을 표현하는 데이터형에 제한을 주지 않으며 탐색자들이 MBR이 변경되는 노드에 대해서도 정확하게 탐색을 수행할 수 있도록 한다. 두 번째 PLC 기법을 제안하였다. PLC는 삽입자들이 잠금 결합을 이용하지 않고도 MBR 변경을 조상노드에 전파할 수 있도록 한다. 이것은 색인구조의 전체적인 성능을 증가시킨다. 세 번째, 노드에 엔트리를 추가하는 방법을 새롭게 제안하였다. 이 방법을 이용하면 탐색자들은 엔트리가 추가되고 있는 노드에 대해서도 탐색이 가능하다. 네 번째, 최적화된 노드 분할 방법을 제안하였다. 탐색자들은 노드 분할 중 극히 적은 시간을 차지하는 물리적 분할 동안에만 지연된다. 마지막으로 제안하는 동시성제어 알고리즘을 위한 회복 방법을 제안하였다.

이상 언급한 알고리즘의 특징은 다차원 색인구조의 동시성을 매우 크게 향상시켰다. CGiST와의 성능평가 결과 실 데이터를 가지고 실험했을 때 응답시간 관점에서는 약 3배 처리율관점에서는 약 2배의 평균적인 성능향상을 보였다. 그리고, 데이터베이스의 크기가 증가하거나 MPL이 증가하

는 상황에서도 CGiST에 비해서 매우 우수한 성능을 보여 주었다. 향후연구에서는 제안하는 동시성제어 기법이 유령 현상을 방지하는 영역에서도 잘 동작할 수 있도록 유령현상 방지 기법을 제안한다. 기존에 이미 유령현상을 방지하는 기법이 제안되었지만 K-NN에 대한 고려가 없다. 향후 연구에서는 보다 효율적이며 K-NN 질의도 고려하는 유령현상 방지 기법을 제안한다.

참 고 문 헌

[1] A. Guttman. "R-Trees : a dynamic index structure for spatial searching," In Proc. ACM SIGMOD, pp.47-57, 1984.

[2] 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," In Journal of ACM TODS, 17(1), pp.94-162, March, 1992.

[3] C. Mohan and F. Levine, "ARIES/IM : An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," In Proc.of ACM SIGMOD, pp.371-380, June, 1992.

[4] D. A. White and R. Jain, "Similarity Indexing with the SS-tree," In Proc. of ICDE, pp.516-523, 1996.

[5] J. K. Chen and Y. F. Huang, "A Study of Concurrent Operations on R-Trees," In Journal of Information Sciences 98, pp.263-300, 1997.

[6] Jae Soo Yoo, Myung Geun Shin, Seok Hee Lee, Kil Seong Choi, Ki Hyung Cho and Dae Young Hur, "An Efficient Index Structure for High Dimensional Image Data," In Proc. of AMCP, pp.134-147, 1998.

[7] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree : an index structure for high-dimensional feature spaces," In Proc. of ICDE., pp.440-447, 1999.

[8] K. I. Lin, H. Jagadish and C. Faloutsos, "The TV-tree : An Index Structure for High Dimensional Data," In Journal of VLDB, Vol.3, pp.517-542, 1994.

[9] M. Chae, K. Hong, M. Lee, J. Kim, O. Joe, S. Jeon and Y. Kim, "Design of the Object Kernel of BADA-Ⅲ : An Object-Oriented Database Management System for Multimedia Data Service," The 1995 Workshop on Network and System Management, 1995.

[10] M. Kornacker, C. Mohan and J. M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," Proceeding of International Conference ACM SIGMOD, pp. 62-72, May, 1997.

[11] M. Kornacker and D. Banks, "High-Concurrency Locking in R-Trees," Proceeding of International Conference VLDB, pp.134-145, September, 1995.

[12] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-Tree : An Efficient and Robust Access Method for Points and Rectangles," Proceeding of International Conference ACM SIGMOD, pp.322-331, 1990.

[13] N. Katayama and S. Satoh, "The SR-tree : An Index Structure for High-Dimensional Nearest Neighbor Queries," Proceeding of International Conference ACM SIGMOD, May, 1997.

[14] P. L. Lehmann and S. B. Yao, "Efficient Locking for Concurrent Operations on B-Trees," ACM TODS, 6(4), pp. 650-670, December, 1981.

[15] K. V. Ravi. Kanth, D. Serena and A. K. Singh, "Improved concurrency control techniques for multi-dimensional index structures," Proceedings of the First Merged International and Symposium on Parallel and Distributed Processing (IPPS/SPDP), pp.580-586, 1998.

[16] S. Berchtold, D. A. Keim and H. P. Kriegel, "The X-Tree : an index structure for high-dimensional data," In Proc. of VLDB Conf., pp.28-39, 1996.

[17] V. Ng and T. Kamada, "Concurrent Accesses to R-Trees," Proceeding of Symposium on Large Spatial Databases, pp. 142-161, 1993.

[18] A. Silberschatz and P. B. Galvin, "Operating System Concepts," Addison-Wesley, 1995.

[19] K. Chakrabarti and S. Mehrotra, "Dynamic Granular Locking Approach to Phantom Protection in R-Trees," In Proc. of ICDE Conf., pp.446-454, 1998.

[20] K. Chakrabarti and S. Mehrotra, "Efficient Concurrency Control in Multidimensional Access Methods," In Proc. of SIGMOD Conf., pp.25-36, 1999.

송 석 일

e-mail : prince@pretty.chungbuk.ac.kr

1998년 충북대학교 정보통신공학과 (공학사)

2000년 충북대학교 정보통신공학과 (공학석사)

2000~현재 충북대학교 전기전자컴퓨터 공학부 정보통신공학과 박사과정

관심분야 : 데이터베이스 시스템, 트랜잭션, 저장 시스템, 멀티미디어 정보검색, XML, SAN 등

유 재 수

e-mail : yjs@cbucc.chungbuk.ac.kr

1989년 전북대학교 컴퓨터공학과(학사)

1991년 한국과학기술원 전산학과(공학석사)

1995년 한국과학기술원 전산학과(공학박사)

1995년~1996년 목포대학교 전산통계학과 전임강사

1996년~현재 충북대학교 전기전자컴퓨터공학부 부교수

관심분야 : 데이터베이스 시스템, 멀티미디어 데이터베이스, 실시간 데이터베이스, SAN 등