PPMMLG : 다차원 색인구조를 위한 다중 레벨 그리드 방식의 유령현상 방지 기법

(PPMMLG : A Phantom Protection Method based on Multi-Level Grid Technique for Multi-dimensional Index Structures)

이 석 재 † 송 석 일 †† 유 재 수 †††
(Seok Jae Lee) (Seok Il Song) (Jae Soo Yoo)

요 약 이 논문에서는 다중 레벨 그리드 기법을 기반으로 하는 새로운 다차원 색인구조의 유령 방지 기법을 제안한다. 제안하는 기법은 트리 기반 및 비 트리 기반의 색인구조 모두에 적용이 가능하다. 또한, 제안하는 방법은 데이타베이스 관리시스템에 통합이 용이하며 적은 잠금 부담에 높은 동시성을 제공한다. 기존의 방법과 비교를 하기 위해 실제로 구현을 하였으며 다양한 환경에서의 실험을 통해 우수성을 입증하였다.

키워드 : 동시성 제어, 유령 방지, 다차원 색인구조

Abstract In this paper, we propose a new phantom protection method for multi-dimensional index structures that uses multi-level grid technique. The proposed mechanism is independent of the types of multi-dimensional index structures, i.e., it can be applied to all types of index structures such as tree-based, file-based and hash-based index structures. Also, it achieves low development cost and high concurrency with low lock overhead. It is shown through various experiments that the proposed method outperforms existing phantom protection methods for multi-dimensional index structures.

Key words : Concurrency control, Phantom protection, Multi-dimensional index structure

1. 서 론

지난 20여 년 동안 GIS(Geographic Information System), MLS(Mobile Location Service), CAD(Computer-Aided Design) 등과 같은 새로운 데이타베이스 응용들이 각광을 받아 왔다. 이들 응용들은 공통적으로 다차원 데이타를 처리한다. 예를 들어 GIS는 건물, 강, 도시와 같은 2차원의 지형 데이타를 저장하고 검색한다. MLS 시스템들은 휴대폰과 같이 움직이는 객체의 현재 위치를 사용자에게 제공하기 위해서 움직이는 객체의 위치를 2차원의 점 데이타로 표현한다. 이 새로 출현한 데이타베이스 응용들이 다차원 데이타를 효과적으로 저장 검색할 수 있도록 다양한 다차원 색인구조들이 제안되었다. 이 색인구조들은 크게 Grid-파일[1], KDB-트리[2], Quad-트리[3]와 같이 영역을 중심으로 분할을 수행하는 부류와 데이타 분포에 따라 분할을 수행하는 색인구조들인 R-트리[4], R+-tree[5], R*-tree [6], X-tree[7], SR-tree[8], M-tree[9], TV-tree[10], CIR-tree[11]들로 나누어 볼 수 있다. 이 외에도 두 방법의 장점을 혼합한 Hybrid-트리[12], 파일 구조를 이용하는 VA-파일[13], 해싱 기법을 이용하는 색인구조[14]가 존재한다.

이런 다양한 형태의 다차원 색인구조가 위에서 언급한 새로운 데이타베이스 응용을 효과적으로 지원하기 위해서는 기존의 상업용 DBMS(Database Management System)의 한 접근방법(Access Method)으로 통합되어야 한다. 색인구조를 DBMS의 접근방법으로 통합하는 것이 중요한 문제이지만 실제로 이에 대한 연구는 그리 활발히 진행되지 않았다. 이 통합을 위해서는 일반적으로 동시성 제어와 회복이라는 두 가지 문제를 같이 고려해야 한다. 이중 이 논문에서 관심을 갖는 동시성 제어 기법은 다시 다음과 같은 두 가지 문제를 포함한다.

· 본 연구는 한국과학재단 목적기초연구(특정기초연구 과제번호 R01-2003-000-10627-0) 지원 및 산업자원부의 지역혁신 인력양성사업의 연구결과로 수행되었음.

† 비 회 원 : 충북대학교 정보통신공학과
sjlee@netdb.cbnu.ac.kr
†† 비 회 원 : 충주대학교 전기전자정보공학부 교수
sisong@chungju.ac.kr
††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@cbucc.chungbuk.ac.kr

논문접수 : 2004년 8월 26일
심사완료 : 2005년 1월 20일

첫 번째는 동시에 수행되는 삽입, 삭제 연산으로부터 데이타 구조의 물리적 일관성을 유지하는 방법이다. 두 번째는 탐색 영역이 동시에 수행되는 삽입이나 삭제의 철회로 인해 유령 레코드가 발생하지 않도록 하는 유령 방지 기법이다[15,16].

위에서 이야기한 유령 방지 기법은 DBMS의 고립 수준과 관계가 있다. 색인구조가 DBMS의 한 접근 방법이 되기 위해서는 통합되는 DBMS에서 제공하는 모든 트랜잭션 고립수준(Isolation)을 지원해야 한다. ISO와 ANSI SQL 표준안은 최고수준의 트랜잭션 고립성(no phantom read)을 기본으로 할 것을 권고하고 있다. 그러나 성능상의 이유 때문에 대부분의 DBMS는 dirty read, committed read, repeatable read, no phantom read의 4가지 고립수준을 지원하며 SQL 사용자 및 응용 프로그래머로 하여금 응용 영역에 따라서 적절히 고립수준을 결정하도록 하고 있다. 첫 번째 문제(색인구조의 물리적 일관성을 유지하는 문제)에 대해서는 잠금 결합과 링크 기법을 사용하는 여러 방법들이 제안되었다[17-21]. 반면에 두 번째 문제(유령 현상을 방지하는 문제)에 대한 연구는 [15,16,19]로 상대적으로 적은 편이다.

B+-트리를 위한 유령 방지 기법들[22,23]은 이미 20여 년 전에 제안 되어서 현재는 여러 상업용 DBMS에서 안정적으로 쓰이고 있다. 이들 방법들은 공통적으로 데이타들이 키 값에 따라 정렬된다는 점을 이용하고 있다. 하지만, 다차원 색인구조에서는 다차원 특징을 갖는 데이타들간에 순서가 존재하지 않기 때문에 B+-트리에서 사용되고 있는 유령 방지 기법을 가져다 적용할 수 없다. 그래서, 다차원 색인구조를 위해 처음으로 개발된 유령 방지 기법은[22,23]와 같이 키 값들 간에 순서를 이용하는 방법이 아니고 수정된 프레디킷 잠금 기법[19]이다. 이 프레디킷 잠금 기법은 이론적으로 높은 동시성을 제공하지만 잠금에 대한 부담이 그래뉼러 잠금 기법보다 높아서 실제로는 그래뉼러 잠금 기법이 선호되고 있다. 이에 [11,12]에서는 다차원 색인구조에서 사용할 수 있는 그래뉼러 잠금 기법을 제안하였다. 이 그래뉼러 잠금 기법은 [19]의 프레디킷 잠금 기법에 비해 효과적이다.

그러나, 이들도 몇 가지 문제점을 가지고 있다. 첫 번째는 잠금 대상이 되는 그래뉼이 색인 트리의 노드이다. 이로 인해 색인 트리의 노드에 잠금을 획득하는 기존의 물리적 일관성 유지 동시성 제어기법들과의 통합이 쉽지 않다. 두 번째는 영역 분할부류의 그래뉼간 겹침이 없는 색인구조에서는 효과적이지만 데이타 분할 기법을 사용하는 색인구조에서는 그래뉼간 겹침 문제와 삽입으로 인한 잦은 트리구조 변경으로 인해 효율성이 많이 떨어진다. 마지막으로, 이 기법은 트리 기반의 색인구조에만 적용이 가능하다는 것이다.

기존의 다차원 색인구조들이 대부분 트리 기반이긴 하지만 VA-파일[13]나 해쉬 기법을 쓰는 색인구조[14]와 같이 비트리 기반의 색인구조들도 다수 존재한다. 이들 색인구조들은 트리 기반의 색인구조와는 달리 물리적 일관성을 유지하기 위한 복잡한 동시성 제어기법이 필요하지 않다. 하지만 적절한 유령 방지 기법은 필요할 것이다.

기존의 유령 현상 방지 기법들은 프레디킷 잠금 기반(탐색 영역에 잠금 획득)의 방법과 그래뉼러 잠금 기반(노드에 잠금 획득)의 방법으로 나누어 볼 수 있다. 이 논문에서 제안하는 기법은 기존의 방법들과는 다른 접근 방법을 취한다. 제안하는 방법은 다중 레벨 그리드 기법을 기반으로 하여 탐색 영역을 그리드 상에 사상하여 잠금을 획득하는 새로운 유령 방지기법을 제안한다. 제안하는 방법의 기본 아이디어는 다차원 데이타 공간을 고정된 숫자의 셀로 나누고 각 셀에 유일한 번호를 할당한 후 이들을 잠금 객체로 사용하는 것이다. 또한 효과를 높이기 위해 셀들을 여러 레벨에 걸쳐 계층적으로 구성하여 잠금의 개수를 줄인다. 탐색자들의 프레디킷은 이와 겹치는 셀들의 번호로 매핑되며 탐색자들이 이들 셀 번호에 잠금을 획득하여 유령 현상을 방지한다. 삽입자(삭제자)들은 삽입(삭제)하려는 객체를 셀들의 집합으로 매핑하고 이 셀들에 잠금을 획득한다.

제안하는 유령 방지 기법은 색인구조의 알고리즘에 어떠한 수정도 필요로 하지 않는다. 또한, 색인구조의 노드에 전혀 잠금을 획득하지 않는다. 이로 인해서, 기존에 제시된 물리적 일관성 유지를 위한 동시성 제어 기법과의 통합이 매우 용이하다. 또한, 색인구조가 트리 기반인지 비 트리 기반인지에 상관없이 적용될 수 있다. 제안하는 기법의 우수성을 입증하기 위해서 다양한 실험을 수행한다. 이 논문은 다음과 같이 구성되어 있다. 2장에서는 기존의 유령 방지 기법에 대한 자세한 설명을 기술한다. 3장에서는 제안하는 유령 방지기법을 설명한다. 4 장에서는 성능평가 환경 및 결과를 보여주고 5장에서는 결론을 맺는다.

2. 관련 연구

이미 언급한 대로 다차원 색인구조에서는 객체들이 정렬되지 않기 때문에 프레디킷 잠금을 이용하는 방법이 최초로 제안되었다. 탐색자들은 탐색 프레디킷을 프레디킷 테이블에 기록한 후 탐색을 수행하고 삽입자(삭제자)는 자신이 삽입(삭제)하려는 객체가 프레디킷 테이블에 기록된 프레디킷과 겹치는지 확인한 후 연산을 수행한다.

프레디킷 잠금 기법을 이용하면 탐색자, 삽입자, 삭제자 모두 하나의 잠금만을 획득하면 되므로 데이타 레코드에 잠금을 별도로 획득할 필요가 없어 잠금에 대한 부담이 적다. 하지만, 프레디킷 테이블 전체를 조회한 후 해당 프레디킷들에 잠금을 획득하는 과정이 매우 소모적이다. 그리고, 프레디킷 테이블 자체를 유지하는데 드는 비용이 매우 높다. 이런 부담을 경감하기 위해서 [6]에서는 프레디킷 테이블을 전역적으로 관리하지 않고 색인구조의 각 노드에 분산하는 방법을 제안했다. 하지만 이 방법 역시 색인구조의 각 노드에 별도의 공간을 마련해서 프레디킷을 저장해야 하며 프레디킷의 개수가 지속적으로 변하기 때문에 테이블 유지에 어려움이 있다. 또한, 노드가 분할되거나 노드의 MBR이 변경되면 프레디킷 테이블의 내용이 같이 변경되어야 한다.

[15,16]은 그래뉼러 잠금을 이용하는 유령 방지 기법을 제안했다. 이 방법에서는 최하위 레벨의 MBR을 잠금 대상이 되는 그래뉼로 정의하고 있다. 이 최하위 레벨 MBR은 R-트리에서의 단말 노드가 된다. R-트리의 그래뉼은 엔트리가 삽입/삭제 되면서 객체의 분포에 맞게 동적으로 팽창 및 축소한다. 또한, R-트리의 최하위 레벨의 MBR 의 합집합이 전체 데이타 공간이 되지 않을 수도 있다. 이것은 모든 그래뉼에 잠금을 획득한다 해도 탐색 프레디킷에 유령 레코드가 발생하는 것을 완전히 막을 수 없다는 뜻이다. 이 때문에 트리의 비 단말 노드에 해당하는 그래뉼을 추가적으로 정의하고 있다. 최하위 레벨의 그래뉼과 추가적으로 정의한 외부 그래뉼을 합하면 전체 데이타 공간과 같아지게 된다.

삽입자(삭제자)는 삽입하려는 객체에 x(exclusive)-잠금을 그리고 이 객체의 MBR과 겹치는 그래뉼의 최소 집합에 ix(intentional exclusive)-잠금을 획득한다. 탐색자들은 탐색 프레디킷과 겹치는 모든 그래뉼에 s(shared)-잠금을 획득한다. 이렇게 하면 탐색 프레디킷과 삽입하려는 객체의 MBR이 겹칠경우 객체의 삽입이 허용되지 않는다. 하지만 안타깝게도 이것으로 완전하게 유령 현상을 방지할 수는 없다. R-트리의 경우 그래뉼이 삽입이나 삭제 때문에 동적으로 변경되는 상황이 발생하기 때문이다. 두개의 겹침이 없는 그래뉼이 있고 하나의 그래뉼에는 s-잠금이 설정 되었다. 만일 다른 그래뉼이 삽입으로 인해 확장되어 s-잠금이 설정된 그래뉼과 겹치게 된다면 겹침이 발생하는 부분에서는s-잠금이 더 이상 무의미 해진다. 이를 해결하기 위해서 삽입으로 인한 MBR 확장, 분할로 새로 생성되는 노드들을 고려하여 추가적인 잠금 전략을 사용한다. 탐색자 역시 탐색 프레디킷과 겹치는 모든 그래뉼에 s-잠금을 획득한다. 그리고, 삭제자들은 삭제할 객체를 포함하는 모든 그래뉼에 잠금을 획득해야 한다. 그림 1에서 이 내용을 간략히 설명하고 있다.

그림에서 탐색자가 탐색영역을 검색하고 있다. 이때 잠금은 노드 M과 노드 M의 하위 노드인 J에 획득한다. 이때 새로운 객체가 삽입이 되고 이 삽입객체가 노드 N의 하위 노드인 R에 삽입된다. 그 결과 노드 R과 노드 N의 MBR은 확장되어 노드 M과 J와 겹치게 된다. 탐색자는 노드 R과 N에는 잠금을 획득하지 않았으므로 이어지는 삽입자는 노드 R, N에 객체를 삽입할 수 있게 되고 이 객체들은 탐색자에게는 유령 객체가 된다. 이를 방지하기 위해서는 삽입자는 MBR 확장으로 겹치는 다른 노드들에 탐색이 수행되는지를 잠금을 이용해 확인한 후 삽입을 해야 할 것이다. 이 확인을 위해서는 삽입자가 삽입을 위한 트리 순회 외에 추가적인 트리 순회를 해야 하며 이로 인해 전체적인 성능의 저하를 가져올 수 있을 것이다.

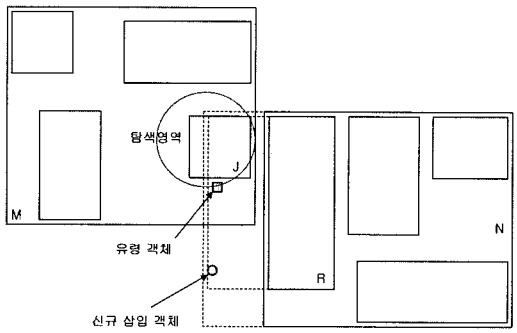

그림 1 그래뉼러 방법의 삽입과 탐색

기존 연구를 참조할 때 그래뉼러 잠금을 사용하는 방법이 프레디킷 잠금을 사용하는 방법에 비해서 훨씬 더 효과적이다. 그 이유는 잠금 대상이 되는 그래뉼이 색인 트리의 노드이므로 DBMS에서 지원하는 잠금 기법을 그대로 사용할 수 있다. 또한, 프레디킷 잠금처럼 잠금을 위해서 각 노드에 추가적인 정보를 유지할 필요가 없다. 하지만, 그래뉼러 잠금을 이용하는 방법의 경우 삽입 이나 삭제로 인해 그래뉼이 변경되거나 분할이 발생되면 객체와 겹치는 모든 노드에 잠금을 획득해야 한다. 이를 위해서는 삽입자가 겹침이 발생하는 노드를 찾기 위해 노드를 추가적으로 순회 해야 한다. 그리고, 잠금이 색인 노드에 설정되기 때문에 기존에 색인 노드에 잠금을 사용하는 물리적 일관성 유지를 위한 동시성 제어기법과 통합하는 것이 어려워진다.

3. 제안하는 유령 현상 방지 기법

3.1 기본 알고리즘

제안하는 유령현상 방지 기법은 사용자가 지정한 b

값에 따라 다차원 데이타 공간을 균등한 크기의 2^b 개 셀들로 분할한다. 각각의 셀은 길이가 b인 고유한 비트열로 표현이 가능하다. 각 셀은 같은 크기를 갖고, 전체 셀의 합집합은 전체 데이타 공간과 같다. 이 각각의 셀들이 잠금 대상이 된다. 각 셀에 할당되는 길이가 b인 고유한 비트열은 정수로 변환이 가능하며 이 정수 값은 통합되는 DBMS의 잠금 식별자로 사용된다. 대부분의 DBMS가 정수값을 잠금 식별자로 사용하므로 이들의 잠금 기법을 그대로 사용할 수 있다. 탐색자들은 탐색 프레디킷과 겹치는 셀들에 s-잠금을 획득하게 된다. 삽입자(삭제자)들은 삽입(삭제) 하려는 객체의 MBR과 겹치는 셀들에 x-잠금을 획득한다.

b_i 만큼의 비트를 각 차원 i에 할당하면 차원 i에는 길이가 같은 2^{b_i} 개의 조각이 생긴다. b는 b_i의 총 합이 된다. 즉, $b=\sum_{i=1}^{d} b_i$, i는 차원 수이다. 전체 데이타 공간은 2^b개의 셀로 분할 되며 각각의 셀은 길이 b의 비트열로 표현 가능하다. 각 셀은 같은 크기의 영역을 포함 하며 셀들의 합은 전체 데이타 공간과 같다.

그림 2는 제안하는 알고리즘의 간단한 예를 보여주고 있다. 2차원의 데이타 공간은 (0, 0)에서 (15, 15)에 해당하는 영역을 차지하며 b는 8이라고 가정하자. 데이타 공간은 2^8 개의 셀로 분할되게 된다. 그림에서 음영이 있는 사각형 영역은 탐색 프레디킷이며 그 범위는 (0, 0) 에서 (2, 2) 이다. 탐색 영역과 겹치는 셀들은 모두 9개이다. (0, 0)에서 (1, 1)에 해당하는 셀에 길이 b의 비트 열을 할당하면 00000000 이 되며 이것을 정수로 변환하면 0이 된다. 이런 식으로 나머지 8개의 셀을 정수로 변환하면 1, 2, 16, 17, 18, 32, 33, 34 가 된다. 탐색자는 탐색을 수행하기 전에 먼저 이 정수값에 해당하는 셀에 s-잠금을 획득한다.

이어서, 한 삽입자가 (0, 0)의 값을 갖는 객체를 삽입하기 위해 탐색자와 마찬가지로 객체와 겹치는 셀을 골라내고 이 셀들에 x-잠금을 설정하게 되는데 이 예에서 객체와 겹치는 셀은 (0, 0), (1, 1) 이며 이를 정수로 변환하면 0 이다. 하지만 이미 탐색자가 이 셀에 s-잠금을 획득하고 있으므로 삽입자는 이 탐색자가 완료할 때까지 대기한 후 잠금을 획득하게 될 것이다.

제안하는 기법에서 탐색자의 잠금 개수는 b와 질의 크기에 따라서 결정된다. 탐색자의 잠금 개수는 수식 $sn=\lceil 2^b \cdot querysize \rceil$에 의해서 대략적으로 계산될 수 있다. 여기서 sn은 탐색 프레디킷과 겹치는 셀의 개수 이며 $querysize$는 탐색자의 질의 크기이다. 질의 크기는 데이타 공간의 전체 크기에 대한 탐색 프레디킷의 크기 비율이다.

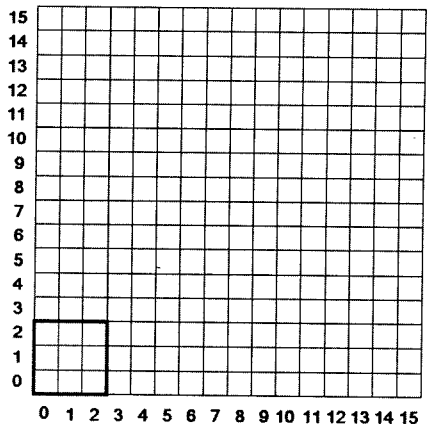

그림 2 탐색 프레디킷의 매핑

b가 10이고 $querysize$가 0.05일 때 sn은 51이다. 같은 $querysize$ 크기에 b가 16이면 sn은 3237이다. 4장의 성능평가에서도 보여지겠지만, b의 값으로는 6~10이면 충분하며 적당한 값으로 고정이 가능하다. 그러나 $querysize$는 질의자에 따라서 가변적이다. 그러므로 b가 10으로 고정되더라도 $querysize$가 증가하면 sn은 선형적으로 증가한다.

제안하는 유령 방지 기법의 단점은 탐색자의 프레디킷을 보호하기 위한 잠금의 개수가 질의 크기에 따라서 많아지고 이에 따른 잠금 부담은 전체적인 성능을 감소시킨다는 것이다. 이 문제를 해결하기 위해서 Multi-Level Grid-file[24]과 같이 셀들을 계층적으로 구성한다. 각 레벨에서 셀들을 $2 \cdot l \cdot d$ 개의 클러스터로 묶는다. 여기서 l은 레벨을 뜻한다. 예를 들어서 그림 3을 보자. 레벨 1에서는 4개의 클러스터가 존재하고 각 클러스터는 64 개의 셀들을 포함한다. 레벨 2에서는 16개의 클러스터가 존재하고 각 클러스터는 16개의 셀들을 포함한다. 최상위 레벨인 레벨 0에서는 하나의 클러스터만 존재하고 이 클러스터는 모든 셀들을 포함한다. 레벨의 개수는 $\left\lceil \frac{b}{d} \right\rceil + 1$ 식에 의해서 결정된다.

셀들을 각 레벨별로 클러스터링 한 후에 각 클러스터에 길이가 $(lb+b)$인 비트 열을 할당한다. lb는 각 레벨을 대표하는 비트의 개수이다. 클러스터를 위한 비트 열은 길이가 lb인 레벨을 위한 비트 열과 클러스터에 포함된 셀들 중 가장 작은 값을 갖는 셀의 비트 열을 합친 것이다. 이렇게 계층적으로 셀들을 구성할 때는 잠금 식별자가 다음과 같이 결정된다. 탐색영역의 사상은 탐색영역과 겹치는 셀들을 골라낸 다음, 각 레벨별로 선택된 셀들로 구성 가능한 그룹을 찾아낸다. 이 작업은 가장 낮은 레벨에서 높을 레벨로 진행된다.

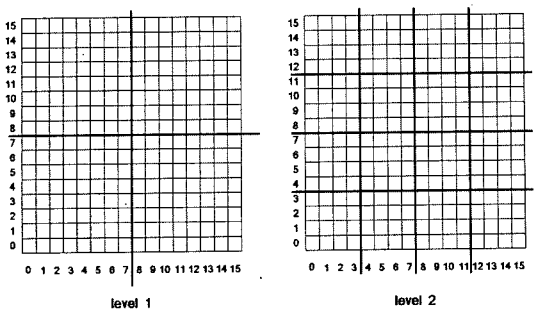

그림 3 셀들의 계층 구조

그림 4는, 탐색자의 탐색 영역 (0, 0), (2, 2) 를 잠금 식별자로 매핑하는 예를 보여주고 있다. 탐색영역과 겹치는 셀들을 그림 2에서와 같이 0, 1, 2, 16, 17, 18, 32, 33, 34로 매핑하였다. 다음, 선택된 셀들로부터 그룹을 선택해 내는 것이다. 그림에서 판별하기로는 레벨 3에 해당하는 하나의 그룹이 있고 레벨 4에 해당하는 5개의 그룹이 있다. 레벨 3의 그룹은 011(레벨)+00000000(가장 낮은값의 셀 비트열) 의 비트열을 갖게 되며 이것은 768이라는 정수값으로 변환이 된다. 나머지 그룹들도 같은 방법으로 정수값으로 매핑된다. 이렇게 하면 잠금의 개수가 그림 2에 비해서 3개가 줄어든 6개이다.

삽입자(삭제자)의 경우 삽입(삭제)할 객체의 MBR과 겹치는 셀들에 x-잠금을 획득하는 것만으로는 부족하다. 예를 들어, 그림 4에서 탐색자가 레벨 3에 하나의 클러스터와 레벨 4에 5개의 클러스터에 s-잠금을 획득한다. 그리고, 삽입자가 객체 (0, 0)을 삽입하려 한다. 삽입자는 객체의 MBR을 매핑한 결과 셀 0를 획득하고 이 셀에 x-잠금을 요구한다. 이때 탐색자는 셀 0에 잠금을 획득한 것이 아니고 클러스터 768에 잠금을 획득한 것이므로 삽입자는 잠금을 획득할 수 있다. 이 문제를 해결하기 위해서 제안하는 방법에서는 삽입하려는 객체의 MBR과 겹치는 최하위 레벨의 클러스터를 제외한 나머지 레벨의 클러스터에 ix-잠금을 획득한다. 물론 최하위 레벨의 클러스터에는 x-잠금을 획득한다. 즉, (0,0)를 삽입하려면 최하위 레벨의 클러스터인 100+00000000에는 x-잠금을 획득하고 000+00000000, 001+00000000, 010+00000000, 011+00000000, 100+00000000 에는 모두 ix-잠금을 획득한다.

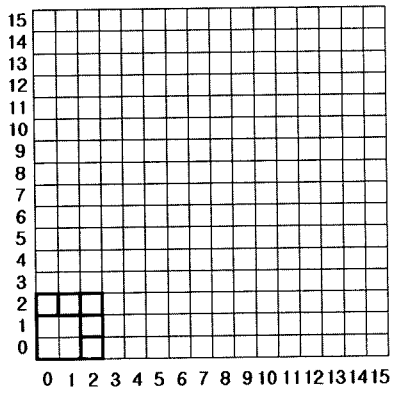

그림 4 계층 방법의 사상

3.2 동적 유령 방지 기법

이제까지 설명한 유령현상 방지 기법은 다차원 데이타 공간이 정적이라는 것을 가정했다. 데이타 공간이 정적이고 데이타 공간을 미리 알 수 있다면 제안한 알고리즘이 제대로 동작할 것이다. 그러나, 전체 데이타 공간이 동적으로 변경된다면, 최대 데이타 공간을 추정해서 제안하는 방법을 적용해야 할 것이다. 그러나, 이것은 사장공간을 증가시켜 전체적인 성능을 감소시킬 것이다.

이런 이유로 동적 유령 방지 기법을 제안한다. 동적으로 변하는 데이타 공간에서 유령현상을 효과적으로 방지하려면, 2^b 개의 셀들이 데이타 공간의 확장 및 수축에 따라서 적절하게 대응해야 한다. 그림 5(a)는 변경되기 전 데이타 공간이다. 그림 5(b)는 객체 삽입으로 데이타 공간이 확장되고 이에 따라 셀들이 변경된 데이타 공간에 맞게 재조정된 것을 보여주고 있다. 데이타 공간이 변경되고 셀들의 크기가 재조정되면 삽입자, 삭제자, 탐색자는 반드시 재조정된 셀들에 잠금을 획득해야 한다.

예를 들어보자. 그림 5에서 프레디킷이 (6, 6)에서 (8, 8)인 탐색자가 데이타 공간이 변경 되기전에 시작되어 탐색 프레디킷과 겹치는 셀 15에 s-잠금을 획득했다. 다른 삽입에 의해서 데이타 공간이 확장되고 셀들은 그림 6에서 처럼 변경된 데이타 공간에 맞도록 크기가 재조정된다. 삽입자는 새로운 객체 (7, 7)을 삽입하기 위해 셀 10′ 에 x-잠금을 요청하고 셀 10′ 에 설정된 잠금이 없기 때문에 바로 x-잠금을 획득한다. 이어서, 탐색자는 (6, 6)에서 (8, 8)에 해당하는 영역에s-잠금을 잃어 버린

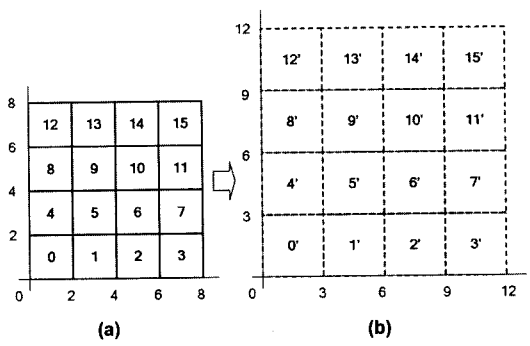

그림 5 데이타 공간의 확장-1

다. 그리고, 새로운 객체 (7, 7)은 탐색자가 같은 영역을 반복해서 스캔하게 되면 유령 객체가 된다.

이 문제를 피하기 위해서는 삽입자가 변경되기 전 데이타 공간의 셀 10과 겹치는 셀 15′과 10′에도 x-잠금을 획득해야 한다. 그러나, 데이타 공간이 변경 되기 전에 시작된 트랜잭션들이 종료 했다면 삽입자는 셀 15′에 x-잠금을 획득할 필요가 없다. 데이타 공간이 변경되기 전에 시작된 트랜잭션을 OT (old transaction) 라 하고 변경된 후 시작된 트랜잭션을 NT(new transaction)라 하자. 그림 6을 보자. 음영이 있는 셀 (6, 6)에서 (9, 9)는 10′이다. 셀 10′은 변경전 데이타 공간에서 (6, 6)에서 (8, 8)에 해당하는 셀 15와 겹친다. 따라서, 변경전 데이타 공간의 셀 15에 잠금이 유지되어야 한다는 것을 알 수 있다. 전체 잠금이 획득될 셀은 변경된 데이타공간의 10′과 15′이다. 또 다른 예로, 탐색자가 그림 6의 셀 5′에 s-잠금을 획득하려면 변경된 데이타 공간의 셀 5′와 겹치는 변경전 데이타 공간의 셀 5, 6, 9, 10에 잠금을 획득해야 한다.

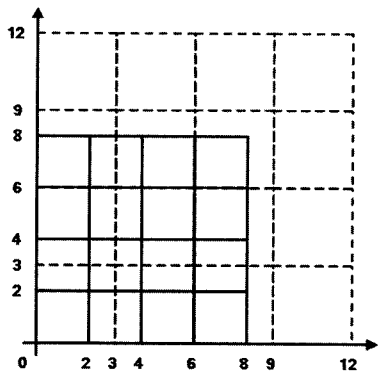

그림 6 데이타 공간의 확장 -2

규칙 1. 데이타 공간이 변경되고, OT 가 여전이 수행되고 있다면 NT 는 반드시 변경된 데이타공간의 셀과 변경전 데이타 공간의 셀들중 변경된 데이타공간의 잠금을 획득할 셀과 겹치는 모든 셀에 잠금을 획득해야 한다.

규칙 1로 데이타 공간이 동적으로 변경되는 상황에서도 유령 레코드들을 차단 할 수 있다. 그러나, 이 규칙을 준수하기 위해서는 다음과 같은 두 가지 문제를 고려해야 한다. 첫 번째는 누가 데이타 공간의 영역을 변경할 시점을 결정하는가 와 어떻게 다른 트랜잭션에 영역이 변경된 사실을 알려줄 것인가 하는 것이다. 두 번째는 어떻게 NT가 OT가 있는지 알아내는가 하는 문제이다.

트리기반의 색인구조에서는 루트 노드의 MBR이 전체 데이타 공간의 MBR 이다. 색인구조가 비트리 기반일 때라도 데이타 공간의 MBR을 유지하는 것은 그기 어려운 문제가 아니다. 데이타 공간의 MBR 변경 시점은 루트노드의 MBR을 변경한 삽입자가 결정할 수 있다. 만일 데이타 공간의 MBR이 루트노드의 MBR 이 변경될 때 마다 변경된다면 영역 변경을 처리하기 위한 부담 때문에 동시성이 오히려 감소될 것이다. 그래서, 이 논문에서는 데이타 공간의 MBR을 변경하는 시점을 데이타 공간의 MBR 중 어느 한 차원이 $\left\lceil \frac{sidelength_i}{2^{b_i}} \right\rceil$ 만큼 변경될 때로 고정한다. $sidelength_i$는 데이타 공간의 i 번째 차원의 길이를 말하며 b_i는 i번째 차원에 할당된 비트의 개수이다. 즉, 데이타 공간의 MBR을 변경하고 다른 트랜잭션들에 이 사실을 알리는 시점이 i차원의 $sidelength_i$가 하나의 셀의 크기 만큼 변경될 때가 되는 것이다.

변경을 다른 트랜잭션에 알리기 위해서는 그림 7에 나오는 데이타 구조를 사용한다. 그림 6에서 *dataspace_mbr[]*는 변경되기 전 데이타공간의 MBR과 변경된 새로운 MBR을 저장한다. *current_mbr*은 0나 1을 값으로 갖는 플래그 이며 *dataspace_mbr[]*의 인덱스로 사용된다. *current_mbr*은 *dataspace_mbr[current_mbr]*와 *dataspace_mbr[1-current_mbr]* 중 어떤 것이 새로운 MBR인지를 나타낸다. 마지막으로 *cnt[]*는 현재 수행되고 있는 OT와 NT의 개수를 나타낸다. *cnt[current_mbr]*는 새로운 MBR인 *dataspace_mbr[current_mbr]*을 참조하는 트랜잭션의 개수를 뜻한다.

```
struct Header
{
  current_mbr; // 0 or 1
  cnt[2];
  dataspace_mbr[2];
}
```

그림 7 데이타 공간의 MBR 변경을 알리기 위한 데이타 구조

삽입자가 루트노드의 MBR을 변경할 때마다 루트노드의 MBR과 *dataspace_mbr[current_mbr]*을 비교한다. 만일 어떤 차원의 $sidelength_i$가 $\left\lceil \frac{sidelength_i}{2^{b_i}} \right\rceil$ 보다 더 커지거나 작아졌다면 *current_mbr*을 *1-current_mbr*로 변경하고 *current_mbr*가 이전 MBR인 *dataspace_mbr[]*을 나타내도록 한다. 그리고 나서 *dataspace_mbr[current_mbr]*을 변경한다.

삽입자, 삭제자, 탐색자는 연산을 시작하기 전에

*cnt[current_mbr]*를 1 증가 시키고 트랜잭션이 완료되거나 철회될 때는 1 감소 시킨다. 이때 *cnt[current_mbr]*와 *cnt[1-current_mbr]*를 읽어서 OT 가 존재하는지를 검사한다. 삽입자, 삭제자, 탐색자는 s-래치를 Header 데이타 구조에 획득하고 읽기를 수행한다. 그리고, *cnt[]*의 값을 증/감 할 때는 x-래치를 획득하지 않는다. 이 값이 워드 크기이기 때문에 원자적으로 읽기 변경이 가능하기 때문이다. x-래치를 획득하는 경우는 *dataspace_mbr[]*을 변경할 때이다. X-래치를 획득하는 경우는 전체 데이타 영역의 MBR을 변경할 때인데, 이 경우가 빈번하게 나타나는 것이 아니다. 따라서, 대부분의 경우는 s-래치를 획득하기 때문에 Header 데이타 구조로 인한 병목현상은 일어나지 않는다.

마지막으로 그림 8의 상황을 고려해 보자. 삽입자는 데이타 영역의 외부에 그림 8에서 점으로 표현되는 객체를 삽입할 수 있다. 또한, 그림 8(a)에서 원으로 표시되는 탐색자의 프레디킷 역시 데이타 영역의 외부에 올 수도 있다. 지금까지 설명한 방법에서는 삽입자나 탐색자는 삽입되는 객체의 MBR 또는 탐색 프레디킷과 겹치는 셀들에 잠금을 획득하게 되어 있어 위와 같은 경우에 탐색 프레디킷은 유령현상으로부터 완벽히 보호될 수 없다. 이 예에서 삽입자는 잠금을 획득하지 않고 탐색자는 셀 14 와 15에만 획득한다. 이로 인해, 삽입자는 객체를 탐색 프레디킷 부분에 삽입할 수 있고 유령 현상이 나타날 수 있다.

이 문제를 해결하기 위해서 추가 잠금 대상을 정의한다. 그림 8(b)에서 보는 것처럼, 음영이 있는 셀들이 추가로 정의한 잠금 대상이다. 실제 데이타 공간은 셀 0~15로 잠금이 가능하고 이 외의 영역에 잠금을 획득하기 위해 0~15의 셀 주변에 추가 잠금 대상을 정의했다. 그림 7(b)에서 탐색자의 프레디킷과 겹치는 0~15까지의 셀에 잠금을 획득하고, 추가적으로 정의된 잠금 대상에 잠금을 획득한다.

제안하는 방법의 전체 알고리즘은 그림 9과 10에 나타나 있다. 탐색자들은 s-래치를 Header 에 획득한 후

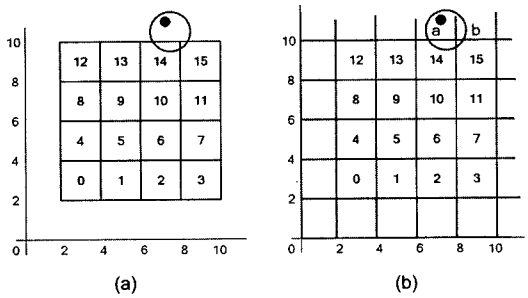

그림 8 추가 잠금 객체

```
Search(query_range, etc)
{
  acquire s-latch on Header;
  cnt[current_mbr]++;
  read dataspace_mbr[current_mbr];
  cell_ids[] = get overlapped cells with query_range;
  if ( cnt[1-current_mbr] != NULL )
  {
    read dataspace_mbr[1-current_mbr];
    cell_ids[] = get cells to be locked;
  }
  release s-latch on Header;
  acquire commit duration s-locks on cell_ids[];
  perform operations;
  acquire s-latch on Header;
  cnt[current_mbr]--;
  release s-latch on Header;
}
```

그림 9 탐색 알고리즘

```
Insert/Delete(entry, etc)
{
  acquire s-latch on Header;
  cnt[current_mbr]++;
  read dataspace_mbr[current_mbr];
  cell_ids[] = get overlapped cells with entry.mbr;
  if ( cnt[1-current_mbr] != NULL )
  {
    read dataspace_mbr[1-current_mbr];
    cell_ids[] = get cells to be locked;
  }
  release s-latch on Header;
  acquire locks on cell_ids[];
  perform insert/delete operation;
  acquire s-latch on Header;
  if (root_mbr is changed more than threshold value)
  {
    if(cnt[1-current_mbr] == 0)
    {
      upgrade s-latch on Header to x-latch;
      current_mbr = 1-current_mbr;
      update dataspace_mbr[current_mbr];
      cnt[1-current_mbr]--;
      release x-latch;
    }
    release s-latch;
  }
  else
  {
    cnt[current_mbr]--;
    release s-latch;
  }
}
```

그림 10 삽입/삭제 알고리즘

*cnt[]*를 읽고 탐색 영역과 겹치는 셀들을 획득한 후 해제한다. 획득한 잠금은 트랜잭션이 완료된 후 해제 된다. *cnt[]*를 읽었을 때 *cnt[1-current_mbr]*가 0이 아니면 *dataspace_mbr[1-current_mbr]*을 참조하는 OT들이 존재한다는 뜻이다. 이 경우에는 이전 데이타 공간과 새로운 데이타 공간 모두에서 잠금 대상을 찾아야 한다.

삽입자와 삭제자들은 탐색자보다 복잡한 행동을 취한다. 먼저, 연산을 끝낼 때 마다 *dataspace_mbr[current_mbr]*가 변경되어야 하는지를 검사한다. 즉, MBR의 변경이나 분할을 전파할 때 *root_mbr*이 특정 경계값 보다 많이 변경되었는지를 검사한다. 만일 변경된 양이 정해놓은 경계값보다 많이 변경되었으면 *dataspace_mbr[1-current_mbr]*를 참조하는OT가 존재하는지를 검사한다. OT가 존재하면 OT가 *dataspace_mbr[1-current_mbr]*을 참조한다는 뜻이므로 *dataspace_mbr[1-current_mbr]*을 변경하지 않는다. 그렇지 않으면, s-래치를 x-래치로 변환하고 *current_mbr*을 *1-current_mbr*로 변경하고 *dataspace_mbr[current_mbr]*을 *root_mbr*로 변경한다. x-래치는 다른 연산들을 지연시킬 수 있는데 *dataspace_mbr[]*의 변경이 그리 빈번하게 나타나는 경우가 아니라서 큰 문제는 없다.

4. 성능 평가

4.1 실험 환경

제안하는 유령 방지 기법의 성능을 평가하기 위해서 [15]에서 제안한 그래뉼러 잠금 기반의 방법(GL)과 비교하였다. GL과 제안하는 방법을 [21]에서 제안한 동시성 제어 기법에 통합하였다. [21]에서 제안하는 동시성 제어 기법은 물리적 일관성을 유지하기 위한 방법이다. 두 유령 방지 기법을 BADA DBMS[25]의 스토리지 시스템인MIDAS에서 구현하였다. 두 방법 모두 MIDAS에서 제공하는 잠금, 래치, 로깅 API를 이용하여 구현되었다. GL을 구현하기 위해서 [21]에서 제안하는 동시성 제어 기법의 삽입 알고리즘을 일부 수정하였다. 수정된 삽입 알고리즘은 삽입 이나 삭제로 인해 MBR 이 변경될 때 마다 추가적인 트리 순회를 수행한다.

제안하는 유령 방지 기법 역시 [21]에서 제안하는 방법에 통합되어 구현하였다. 상대적으로 구현은 간단하고 쉬웠다. [21]의 삽입, 삭제, 탐색 알고리즘을 전혀 수정하지 않았다. 기존 탐색 과 삽입 알고리즘에 탐색 프레디킷과 삽입(삭제)될 객체를 위한 잠금을 생성하는 함수를 추가하였다. Header 데이타 구조는 MIDAS의 색인 테이블에 추가하였다. MIDAS는 색인을 관리하기 위해서 생성된 색인별로 별도의 테이블을 유지한다. 그리고, 셀들을 위한 잠금 식별자를 위해 루트 노드를 이용하였다. 즉, 루트 노드에서 나타날 수 있는 레코드 식별자를 셀들의 잠금 식별자를 위해 사용하였다. [21]의 알고리즘은 색인을 폐기할 때 까지 루트노드가 변경되지 않는다. 루트노드의 첫 번째 두 번째 레코드는 [21]의 동시성제어기법에서 트리 잠금과 노드 잠금으로 예약 되어 있다. 따라서 유령 방지를 위한 셀의 잠금 식별자로 사용할 수 있는 범위는 $2 \sim 2^b+2$까지의 레코드 식별자이다.

실험에 사용된 데이타는 2 ~ 3 차원의 가상 데이타 200,000 개이다. 색인구조의 중요한 파라미터 중 하나가 노드 크기 이다. 노드 크기에 따라서 색인구조의 성능이 다양하게 나타난다. 실험에서는 노드 크기를 4Kbytes에서 16Kbytes까지 변경시켜 보았다. 모든 실험에서 제안하는 방법의 성능이 GL 보다 우수하였다. 전체적인 성능은 노드 크기가 커질수록 증가하였다. 논문에는 노드 크기가 16Kbytes이고 차원이 2일 때의 실험 결과만을 제시하고 있는데 나머지 실험 결과 역시 이와 비슷한 양상을 보여서 생략했다. 또한, 제안하는 방법에서 삭제는 삽입연산과 크게 다르지 않다. 그래서 이 논문에서는 삽입 과 탐색 연산만 평가 하였다.

기본 색인구조는 MIDAS에 구현된 CIR-트리를 이용하였다. 최초에는 CIR-트리[11]를 벌크로딩 기법을 이용해서 구축하고 특정 작업부하에 따라서 객체들을 동시에 다중 프로세스를 이용해 삽입한다. 표 1에서 작업부하 파라미터를 보여주고 있다. 작업부하 생성기는 입력되는 작업부하 파라미터에 따라서 탐색과 삽입 프로세스의 숫자, 동시 수행되는 프로세스의 개수, 색인구조를 구축할 때 사용하는 객체 개수, 범위 질의의 선택도를 결정한다.

표 1 파라미터 및 값

Parameters	Values
Number of feature vectors	200000
Insert probability	0% ~ 100 %
Range of queries	0.2 ~ 0.8 %
Number of concurrent processes(MPL)	10 ~ 50
Number of bits	6, 8, 10 (64, 256, 1024 cells)

작업부하 생성기는 결정한 값들을 C 언어와 MIDAS API를 이용해서 작성한 드라이버 프로그램에 전달한다. 이 드라이버는 탐색과 삽입 프로세스를 수행시키는데 이 각각의 프로세스는 약속한 개수의 트랜잭션을 수행한다. 실험할 때는 MIDAS의 버퍼 개수를 100으로 고정하였다. 실험에 사용된 플랫폼은 듀얼 Ultra Sparc. Processor에 128 Mbytes의 메인 메모리의 H/W에 솔라이스 2.7을 탑재하였다.

4.2 실험 결과

그림 11과 12는 삽입과 탐색의 프로세스를 변경시켜 가면서 제안하는 방법과 GL의 성능을 응답시간 관점에서 비교하였다. 제안하는 방법의 경우 비트의 수를 6~10으로 변경시켜 가면서 실험을 수행하였다. 그림 11과 12의 64, 256, 1024는 비트의 수가 각각 6, 8, 10일 때 제안 하는 알고리즘의 응답시간을 표시한 것이다.

그림 11은 삽입 프로세스의 비율이 0 ~ 100%로 변할 때 탐색 연산의 응답시간을 보여준다. 비트의 개수에 관계 없이 제안하는 방법이 GL보다 높은 성능을 나타낸다. 그러나, 삽입 프로세스의 비율이 0일 때 즉, 탐색 연산만 존재할 때는 거의 같은 결과를 보인다. 기본적인 탐색 알고리즘은 약간의 잠금 전략을 제외하고는 같기 때문에 이 결과가 의미 하는 것은 GL의 잠금 부담이 제안하는 방법과 유사함을 의미한다.

삽입 프로세스의 비율이 증가할수록 GL과 제안하는 방법의 성능 차이가 점점 더 커진다. 특히, 비트의 개수가 10일 때 성능 차이가 최대가 된다. 이것은, 셀의 개수가 증가할수록 제안하는 알고리즘의 성능이 더 높아진다. 그러나, 사실상 64, 256, 1024 그래프 사이에는 별 차이가 없다. 그 이유는 셀의 개수가 커질수록 보다 많은 잠금을 획득해야 하기 때문에 잠금 부담이 증가하기 때문이다. 유사한 양상이 그림 12에서 보여진다. 그러나, GL과 제안하는 방법 사이의 성능 차이는 그림 11에서 보다 크다. GL의 삽입 알고리즘은 제안하는 방법보다 복잡하다. 삽입자는 반드시 새로운 객체의 삽입에 의해 변경된 MBR과 겹치는 노드들을 찾기 위해서 순회해야 한다. 이러한 점이 성능 차이로 나타나는 것으로 추정된다.

그림 13와 14는 선택도를 변경하면서 두 방법의 응답시간을 측정한 결과를 보여준다. 탐색도가 증가할수록, 두 방법의 전체적인 응답시간이 증가하고 제안하는 방법이 GL의 성능을 모든 경우에 능가한다. 그 이유는 탐색자의 프레디킷의 크기가 증가하고 이로 인해 잠금 충돌이 증가하기 때문이다. 제안하는 방법의 단점은 질의의 크기가 증가할수록 잠금의 개수가 증가하는 것이다. 이로 인해서 잠금 부담이 제안하는 방법의 동시성을 감소시킨다. 그러나, 표 2에서 보는 것처럼 탐색자당 평균 잠금의 개수는 그리 크지 않다. 이 잠금 부담은 높은 동시성에 비하면 충분히 무시할 수 있다.

그림 15와 16은 동시 수행 프로세스를 변화 시키면서

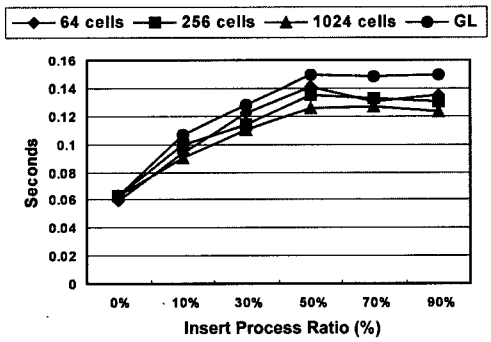

그림 11 삽입연산의 응답시간(selectivity=0.02, database size = 200K, MPL = 50)

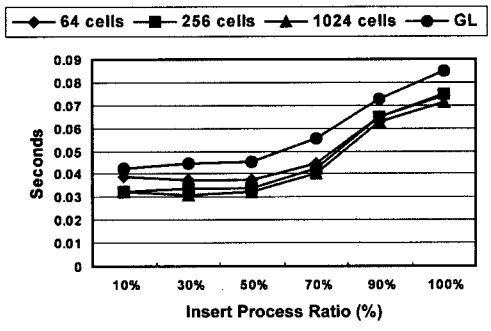

그림 12 탐색연산의 응답시간(selectivity=0.2%, database size = 200K, MPL = 40)

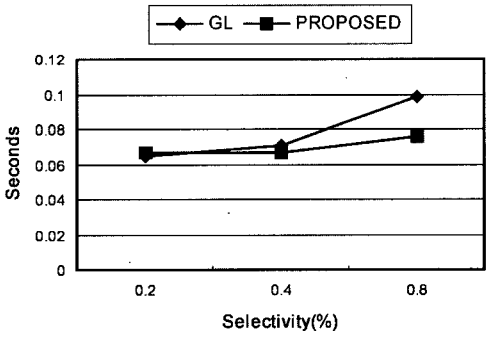

그림 13 탐색연산의 응답시간(MPL=40, insert probability=20%, database size=200K)

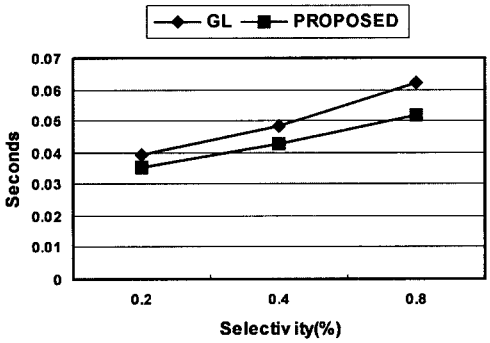

그림 14 삽입 연산의 응답시간(MPL=40, insert probability=20%, database size=200K)

표 2 잠금 개수

Selectivity(%)	0.2	0.4	0.8
Number of locks	11.63	13.893	17.445

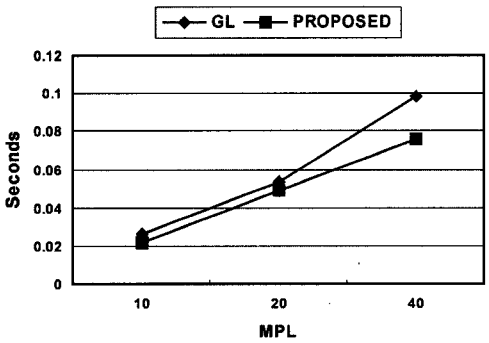

그림 15 탐색연산의 응답시간(selectivity=0.8%, insert probability=20%, database size=200K)

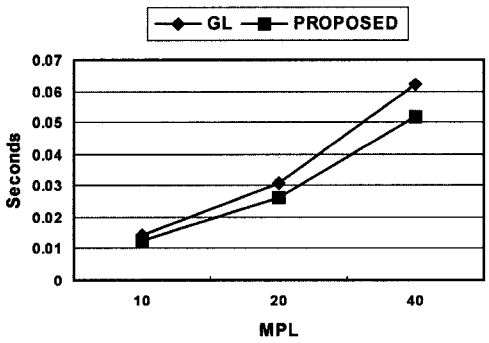

그림 16 삽입 연산의 응답시간(selectivity=0.8%, insert probability=20%, database size=200K)

성능을 측정한 결과이다. 역시 제안하는 방법의 성능이 GL에 비해서 높다. 동시에 수행되는 프로세스의 숫자가 증가 할수록 두 방법의 성능 차이는 점점 커진다. 동시 수행 프로세스가 증가하면 그만큼 잠금 충돌도 많아지기 때문에 제안하는 방법이 잠금 충돌이 많을 때에도 효과적임을 알 수 있다.

5. 결 론

이 논문에서 다차원 색인구조를 위한 유령 방지 기법을 제안하였다. 제안하는 유령 방지 기법은 프레디킷 잠금과 그래뉼러 잠금의 특징을 적절히 혼합하였다. 제안하는 방법은 색인구조의 노드에 잠금을 획득하지도 않고 색인구조의 알고리즘을 변경하지 않아도 된다. 그러므로, 기존에 제안된 물리적 일관성 유지 동시성 제어기법과 쉽게 통합이 가능하다. 또한, 트리 기반 및 비 트리 기반의 색인구조 모두에 적용이 가능하다. 제안하는 방법을 멀티미디어 DBMS BADA의 스토리지 시스템인 MIDAS가 제공하는 잠금, 로깅, 래치 API를 이용해서 구현하였다. 그리고, 다양한 조건하에서 성능평가를 실시 하였다. 실험 결과 제안하는 방법이 기존의 그래뉼러 잠금 기법을 이용한 유령 방지 기법에 비해 우수한 성능을 보임을 알 수 있었다. 제안하는 방법은 질의 크기 및 동시 수행 프로세스의 숫자가 증가할수록 기존 방법에 비해 높은 성능을 보였다.

향후 연구에서는 보다 확장된 실험을 수행한다. 이 논문에서는 가상 데이타를 이용해 실험을 수행 하였다. 실험 결과가 제안하는 방법의 우수성을 보여주기는 했지만 실제 데이타를 이용한 실험을 통해서 보다 확실하게 제안하는 방법의 우수성을 보일 것이다. 또한, 제안하는 방법이 보다 고차원의 데이타 공간에서도 효과적으로 동작함을 보인다.

참 고 문 헌

[1] J. Nievergelt, H. Hinterberger, and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey Structure," ACM Transactions on Database Systems, Vol. 9, No. 1, 1984, pp. 38-71.

[2] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," Proceedings of ACM SIGMOD, 1981, pp. 10-18.

[3] R. A. Finkel and J. L. Bentley, "Quad Trees: A Data Structure for Retrieval on Composite Keys," Acta Informatica, Vol. 4, 1974, pp. 1-9.

[4] A. Guttman. "R-Trees: A Dynamic Index Structure for Spatial Searching," Proceedings of ACM SIGMOD, 1984, pp. 47-57.

[5] T. Sellis, N. Roussopoulos and C. Faloutsos, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," Proceedings of VLDB, 1987, pp. 507-518.

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

[7] S. Berchtold, D. A. Keim, and H. P. Kriegel, "The X-Tree: An Index Structure for High-dimensional Data," Proceedings of VLDB, 1996, pp. 28-39.

[8] N. Katayama and S. Satoh, "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," Proceedings of ACM SIGMOD, 1997, pp. 369-380.

[9] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An Efficient Access Method for Similarity Search in Metric Spaces," Proceedings of VLDB, 1997, pp. 426-435.

[10] K. I. Lin, H. Jagadish and C. Faloutsos, "The TV-tree: An Index Structure for High Dimensional Data," Journal of VLDB, Vol. 3, No. 4,

1994, pp. 517-542.
[11] J. S. Yoo, M. G. Shin, S. H. Lee, K. S. Choi, K. H. Cho and D. Y. Hur, "An Efficient Index Structure for High Dimensional Image Data," Proceedings of AMCP, 1998, pp. 134-147.
[12] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree : An Index Structure for High-dimensional Feature Spaces," Proceedings of ICDE, 1999, pp. 440-447.
[13] R. Weber, H. Schek and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," Proceedings of VLDB, 1998, pp 194-205.
[14] P. Indyk and R. Motwani "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," Proceedings of STOC, 1998, pp. 604-613.
[15] K. Chakrabarti and S. Mehrotra, "Dynamic Granular Locking Approach to Phantom Protection in R-Trees," Proceedings of ICDE, 1998, pp. 446-454.
[16] K. Chakrabarti and S. Mehrotra, "Efficient Concurrency Control in Multidimensional Access Methods," Proceedings of ACM SIGMOD, 1999, pp. 25-36.
[17] J. K. Chen and Y. F. Huang, "A Study of Concurrent Operations on R-Trees," Journal of Information Sciences, Vol. 98, No. 1-4, 1997, pp 263-300.
[18] K. V. Ravi, Kanth, D. Serena and A. K. Singh, "Improved Concurrency Control Techniques for Multi-Dimensional Index Structures," Proceedings of Symposium on Parallel and Distributed Processing, 1998, pp. 580-586.
[19] M. Kornacker, C. Mohan and J. M. Hellerstein, "Concurrency and Recovery in Generalized Search Trees," Proceedings of ACM SIGMOD, 1997, pp. 62-72.
[20] M. Kornacker and D. Banks, "High-Concurrency Locking in R-Trees," Proceedings of VLDB, 1995, pp. 134-145.
[21] S. I. Song, Y. H. Kim and J. S. Yoo, "An Enhanced Concurrency Control Algorithm for Multi-dimensional Index Structures," IEEE Transactions on Knowledge and Data Engineering, Vol. 16, No. 1, 2004, pp. 97-111.
[22] C. Mohan, "ARIES/KVL: A Key Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes," Proceedings of VLDB, 1990, PP. 392-405.
[23] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," Proceedings of ACM SIGMOD, 1992, pp. 371-380.
[24] K. Y. Whant, S. W. Kim, G. Wiederhold, "Dynamic Maintenance of Data Distribution for Selectivity Estimation," Journal of VLDB, Vol. 3, 1994, pp. 29-51.
[25] 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," Proceedings of Workshop on Network and System Management, 1995.

이 석 재

1993년 3월~2000년 2월 충북대학교 정보통신공학과 정보통신공학 전공(학사) 2000년 3월~2002년 2월 충북대학교 정보통신공학과 정보통신공학전공(석사). 2002년 3월~현재 충북대학교 정보통신공학과 정보통신공학전공 박사과정 재학 중 관심분야는 데이타베이스 시스템, 주기억 장치 데이타베이스 시스템, SAN, 분산 실시간 시스템 등

송 석 일

1998년 충북대학교 정보통신공학과(공학사). 2000년 충북대학교 정보통신공학과(공학석사). 2003년 충북대학교 정보통신공학과(공학박사). 2003년 3월~2003년 8월 KAIST 전자전산학과 박사후연구원 2003년 8월~현재 충주대학교 컴퓨터공학과. 관심분야는 데이타베이스 시스템, 트랜잭션, 저장 시스템, 멀티미디어 정보검색, XML, 정보검색 프로토콜 등

유 재 수

1989년 전북대학교 공과대학 컴퓨터공학과(학사). 1991년 한국과학기술원 전산학과(공학석사). 1995년 한국과학기술원 전산학과(공학박사). 1995년~1996년 목포대학교 전산통계학과 전임강사. 1996년~현재 충북대학교 전기전자컴퓨터공학부 부교수 및 컴퓨터·정보통신연구소 소장. 관심분야는 데이타베이스 시스템, 정보검색, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅