R-트리에서 빈번한 변경 질의 처리를 위한 효율적인 기법

(An Efficient Technique for Processing Frequent Updates in the R-tree)

권 동 섭 † 이 상 준 †† 이 석 호 †††
(Dongseop Kwon) (Sangjun Lee) (Sukho Lee)

요 약 정보 통신 기술의 발달은 데이타베이스 분야에도 새로운 응용들을 만들고 있다. 예를 들어, 수많은 객체들의 위치를 추적하는 이동 객체 데이타베이스나 각종 센서들로부터 들어오는 데이타 스트림을 처리하는 스트림 데이타베이스에서 다루는 데이타는 일반적으로 매우 빠르고 끊임없이 변경된다. 하지만, 전통적인 데이타베이스에서는 데이타를 사용자의 명시적인 변경이 있기 전까지는 변하지 않는 상대적으로 정적인 것으로 간주하고 있기 때문에, 전통적인 데이타베이스 시스템은 이러한 끊임없고 동적인 데이타의 변화를 효율적으로 처리하는데 문제를 지닌다. 특히 다차원 데이타 처리를 위한 대표적 인덱스 구조인 R-트리의 경우, 데이타의 삽입이나 삭제가 연속적인 노드의 분할이나 합병을 유발하고 있으므로 이러한 문제는 더 심각해진다. 본 논문에서는 이러한 빈번한 변경 효율적으로 처리하기 위하여 새로운 R-트리 갱신 기법인 리프 갱신 기법을 제안한다. 리프 갱신 기법에서는 새로운 데이타가 이전에 속해있던 리프 노드의 MBR 내에 있으면 전체 트리를 변경하지 않고 해당 리프 노드만을 변경시킨다. 이러한 리프 갱신 처리와 리프 노드를 직접 접근하게 해주는 리프 접근 해시 테이블을 이용하여 리프 갱신 기법은 데이타의 변경 연산 비용을 크게 줄인다. 제안기법은 기존 R-트리의 알고리즘과 구조를 그대로 이용하고, R-트리의 정확성을 보장하므로 다양한 R-트리 변종들에도 적용 가능하고 R-트리를 이용하는 다양한 응용 환경에 이용이 가능하다. 본 논문에서는 제안 기법이 기존 기법에 대하여 가지는 갱신 연산의 비용 이득을 수학적으로 분석하였고, 실험을 통하여 제안 기법의 우수성을 확인하였다.

키워드 : R-트리, 인덱스 구조, 질의 처리, 이동 객체, 리프 갱신 기법

Abstract Advances in information and communication technologies have been creating new classes of applications in the area of databases. For example, in moving object databases, which track positions of a lot of objects, or stream databases, which process data streams from a lot of sensors, data processed in such database systems are usually changed very rapidly and continuously. However, traditional database systems have a problem in processing these rapidly and continuously changing data because they suppose that a data item stored in the database remains constant until it is explicitly modified. The problem becomes more serious in the R-tree, which is a typical index structure for multidimensional data, because modifying data in the R-tree can generate cascading node splits or merges. To process frequent updates more efficiently, we propose a novel update technique for the R-tree, which we call the leaf-update technique. If a new value of a data item lies within the leaf MBR that the data item belongs, the leaf-update technique changes the leaf node only, not whole of the tree. Using this leaf-update manner and the leaf-access hash table for direct access to leaf nodes, the proposed technique can reduce update cost greatly. In addition, the leaf-update technique can be adopted in diverse variants of the R-tree and various applications that use the R-tree since it is based on the R-tree and it guarantees the correctness of the R-tree. In this paper, we prove the effectiveness

· 본 연구는 2003년도 두뇌한국21사업과, 정보통신부의 대학 IT연구센터 (ITRC) 지원을 받아 수행되었습니다.

† 학생회원 : 서울대학교 전기·컴퓨터공학부
subby@dbmain.snu.ac.kr
†† 정 회 원 : 자동제어특화연구센터 연구원
freude@db.snu.ac.kr
††† 종신회원 : 서울대학교 전기·컴퓨터공학부 교수
shlee@cse.snu.ac.kr

논문접수 : 2003년 10월 30일
심사완료 : 2004년 3월 4일

of the leaf-update techniques theoretically and present experimental results that show that our technique outperforms traditional one.

Key words : R-tree, Index Structure, Query processing, Moving objects, Leaf-update Technique

1. 서 론

정보 통신 기술의 급속한 발전은 데이타베이스 분야에도 새로운 응용들을 만들어내고 있다. 예를 들어, GPS(Global Positioning System)와 같은 위치 추적 장비의 발달은 일상생활에서도 쉽게 위치 추적 기능을 이용한 다양한 응용 환경을 이용할 수 있도록 해주고 있다. 이러한 위치 추적 기술은 현재는 단순한 차량 네비게이션 시스템이나 휴대폰 서비스 등에 이용되고 있지만, 앞으로는 전자 상거래나 물류 전송 시장 등에서 없어서는 안 될 중요한 요소가 될 것이다. 이러한 응용 환경을 위해서는 대량의 위치 데이타를 효율적으로 저장 관리할 수 있는 방법이 필요하다. 이러한 요구 조건을 위하여 시공간 데이타베이스(Spatial-Temporal Database) [1]나 이동 객체 데이타베이스(Moving Object Database)[8]에 관한 다양한 연구가 이루어지고 있다. 또한, 웹 애플리케이션, 모니터링 시스템, 센서 네트워크 등의 새로운 응용이 대두되면서 가변적이고 대량으로 생성되는 데이타 스트림(Data Stream)을 처리하기 위한 스트림 데이타베이스[3]에 대한 연구가 새로운 분야로서 활발히 진행되고 있다.

이러한 새로운 응용 분야에서 데이타베이스는 매우 가변적이고, 대량인 데이타를 효율적으로 처리할 수 있어야 한다. 예를 들어 이동 객체의 위치나 매우 많은 수의 센서에서 측정되는 값을 실시간으로 모니터링 하는 응용의 경우 데이타베이스가 처리하여야 하는 데이타의 크기는 매우 클 뿐 아니라 쉬지 않고 변하게 된다. 하지만, 일반적으로 데이타베이스 시스템에서 데이타 변경 연산은 검색 연산에 비해 매우 비용이 큰 연산이다. 전통적인 데이타베이스의 인덱스 구조들은 일반적으로 데이타의 삽입이나 삭제는 고려하고 있지만, 데이타의 변경에 대한 고려는 부족하다. 따라서 전통적인 데이타베이스의 인덱스 구조에서는 이동 객체의 위치가 변하거나 특정 센서로부터 측정된 값이 변할 경우 인덱스에 대하여 삭제, 삽입 연산을 계속적으로 수행하여야 하는 문제점이 있다. 그리고 이러한 삽입이나 삭제 연산은 일반적으로 색인 검색에 비하여 매우 복잡하고 시간이 많이 걸리는 작업이므로 시스템의 성능에 큰 영향을 미친다.

공간 데이타나 다차원 데이타를 처리하기 위해 가장 널리 이용되고 있는 인덱스 구조 중 하나인 R-트리[10] 역시 이러한 문제점을 지닌다. 인덱스의 값을 변경하기 위해서는 단지 데이타만 변경해주는 것이 아니라 데이타 변경에 의하여 인덱스 노드들을 분할, 합병해 주어야 하는 트리 구조의 구조적인 변경이 필요하다. 이미지 데이타베이스나 GIS(Geographic Information System)와 같은 기존의 R-트리 응용에서 데이타는 한번 입력받으면 크게 변하지 않는 것으로 가정된다. 이 경우 데이타의 변경은 일반적인 일이 아니므로 전체 시스템의 성능에 큰 영향을 주지 않는다. 따라서 R-트리에서 데이타의 변경은 삭제 연산과 삽입 연산을 이용하여 처리하게 된다. 하지만, 기존의 R-트리에서는 데이타가 변경되면 우선 기존의 데이타가 삭제되어야 하고, 이러한 데이타의 삭제는 노드의 병합을 발생시킬 수 있고, 이러한 노드의 병합은 트리의 루트까지 연속적으로 일어날 수 있어서 최악의 경우 트리의 높이가 줄어드는 것과 같이 전체 트리의 구조에 영향을 주게 된다. 뿐만 아니라, 새로운 데이타가 삽입될 경우 역시 경우에 따라 트리의 노드가 분할되어야 하고, 이러한 트리 노드의 분할 역시 연속적으로 발생하여 트리의 높이가 증가하게 될 수 있다. 기존의 응용의 경우 데이타의 변경이 자주 일어나지 않으므로 이러한 문제가 심각하지 않지만, 연속적으로 움직이는 이동 객체의 위치나 스트림으로부터 끊임없이 입력되는 값과 같은 빈번한 변경을 요구하는 데이타를 저장하게 될 경우 R-트리의 삭제, 삽입 연산만으로 이러한 변경 연산을 처리하는 것은 시스템의 성능에 크게 영향을 미치게 되는 문제를 지닌다.

예를 들어, 수십, 수백만 명의 사용자가 휴대폰이나 PDA와 같은 휴대용 정보기기와 GPS를 이용하여 서버에 자신의 위치를 계속적으로 보고하고, 데이타베이스에서는 수집된 위치를 계속적으로 저장 추적하는 이동 객체 데이타베이스 응용을 생각해 보자. 그리고 사용자나 관리자는 이렇게 추적되고 있는 위치에 대하여 실시간으로 특정 범위에 존재하는 사용자의 위치를 찾기 위한 간단한 범위 질의나 특정 위치에 가장 가까운 k명의 사용자를 찾기 위한 k-최근접 검색 질의[17]와 같은 다양한 질의를 수행한다고 가정해보자. 이 경우 사용자들의 위치를 전통적인 R-트리에 저장할 경우 사용자의 위치가 변경될 때마다 끊임없이 트리에 삭제, 삽입 연산을 수행하여야 할 것이다. 그리고 이러한 삭제 삽입 연산은 시스템의 성능을 크게 저하시킬 수밖에 없다. 하지만, 그렇다고 해서 인덱스를 전혀 사용하지 않고 사용자의 위치를 저장한다면 복잡한 질의를 처리하기 위해 항상 전체 데이타를 읽어야 하므로 사용자의 질의를 효율적

으로 처리할 수 없을 것이다.

스트림 데이타베이스 분야에서도 비슷한 문제가 발생할 수 있다. 예를 들어, 수십만 개의 다양한 센서로부터 계속적으로 기온, 습도 등의 여러 가지 값들이 실시간으로 측정되고 이러한 측정값들이 데이타 스트림을 통하여 서버로 전달된다고 가정하자. 서버에서는 센서들을 모니터링[7]하고 여러 가지 문제들을 해결하기 위하여 특정 조건을 만족하는 센서들을 찾거나 현재 측정된 값들을 분석하기 위하여 다양한 질의를 수행할 수 있을 것이다. 이러한 센서의 측정값을 저장하기 위해서 전통적인 인덱스를 사용하게 될 경우 마찬가지로 변경 연산이 가지는 문제 때문에 시스템의 성능이 저하될 수 있다.

이와 같이 새롭게 중요성이 대두되고 있는 다양한 분야에서 데이타의 빈번한 변경을 효율적으로 처리하는 인덱스 구조에 대한 필요성이 증대되고 있다. 본 논문은 이러한 문제점을 해결하기 위하여 이동 객체의 위치나 스트림 데이타와 같은 변경이 매우 빈번한 데이타의 색인을 위한 R-트리의 새로운 갱신 기법을 제안한다. 본 논문에서 제안하는 기법은 리프 접근 해시 테이블을 이용하여 리프 노드를 직접 접근하여 불필요한 루트로부터의 트리 순회를 줄이고, 새롭게 변경될 값이 이전에 저장된 값과 많이 달라지지 않아서 이전 값이 저장되어 있는 리프 노드의 MBR(Minimum Bounding Rectangle) 속에 저장될 수 있는 경우, 이 갱신 연산을 전체 트리 구조에 영향을 미치지 않도록 해당 리프 노드에서만 갱신한다. 이러한 리프 갱신이 일어날 경우 변경은 트리 전체에 영향을 주지 않고, 해당 리프 노드에서만 지역적으로 발생한다. 따라서 전체 트리에 대하여 삽입 삭제를 수행하였을 때와 비교하여 최적화된 트리를 만들어 낼 수는 없기 때문에 질의시 효율성이 떨어질 수 있지만, 지역적 변경을 하더라도 R-트리의 정확성은 보장되므로 대량의 데이타 변경을 수행하여야 하는 경우에도 이전의 기법에 비해 전체 수행 비용을 크게 줄일 수 있다. 본 논문에서는 제안 기법의 성능을 이론적으로 분석하였고, 제안기법을 R^*-트리[4] 상에 구현하여 기존의 기법과의 실험을 통하여 성능을 비교 분석하였다. 그리고 이를 통하여 제안 기법이 다양한 환경에서 매우 효율적인 성능을 보여줌을 제시한다.

본 논문은 기본적으로 디스크 기반의 색인 구조를 대상으로 하고 있다. 물론, 데이타의 개수가 많지 않고, 주기억장치의 용량이 충분할 경우 동적인 데이타의 저장을 위해 주기억장치 데이타베이스를 이용하는 것이 효율적일 수 있으나, 모든 응용 환경에서 항상 주기억장치 데이타베이스를 이용할 수 있는 것이 아닐 뿐더러 전통적인 디스크 기반 데이타베이스에서도 색인 구조의 변경 처리에 관하여 고려할 필요가 있으므로 본 논문에서는 디스크 기반의 색인 구조를 대상으로 하여 보다 효율적으로 변경 연산을 처리하는 방법을 살펴 본다.

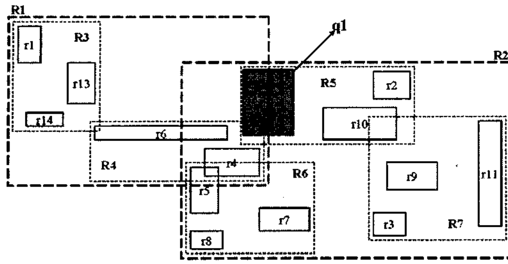
본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로서 R-트리를 간략히 소개하고, 이동 객체 데이타베이스 분야에서 이러한 문제를 풀기 위해 제안된 기법들을 간단히 살펴본다. 3장에서는 본 논문에서 풀고자 하는 문제를 정의하고, 4장에서는 본 논문에서 제안하는 기법인 리프 갱신 기법을 소개한다. 그리고 5장에서는 제안 기법의 비용을 기존의 기법을 사용하였을 때와 비교하여 성능의 이점을 이론적으로 분석하고, 6장에서는 실험을 통하여 제안 기법의 성능을 분석한다. 마지막으로 7장에서 결론을 맺는다.

2. 관련 연구

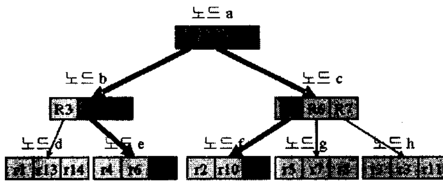
2.1 R-트리 기반 인덱스

R-트리[10]는 점이나 선, 면과 같은 다양한 공간 데이타를 저장하기 위한 디스크 기반의 인덱스 구조이다. R-트리와 관련된 다양한 연구들에 대한 조사 논문[9][15]에서 찾을 수 있다. 기본적인 R-트리의 구조는 일차원 데이타의 색인을 위하여 가장 널리 이용되는 인덱스 구조인 B^+-트리와 유사하다. R-트리는 구조가 단순하고 쉬우면서도 k-최근접 검색 질의[17], 범위 검색 질의, 공간 조인 질의[6] 등 다양한 질의를 효율적으로 지원하므로 다양한 응용분야에 널리 이용되고 있고, 상용 데이타베이스에도 다차원 데이타를 위한 인덱스 구조로서 채택되어 있다.

R-트리는 기본적으로 모든 리프 노드의 높이가 동일한 균형 트리이다. R-트리는 점이나 선, 면 등의 다양한 공간 데이타를 처리하기 위하여 최소 경계 사각형, 즉 MBR(Minimum Bounding Rectangle)을 이용하여 데이타를 표현한다. 그리고 각 노드들 역시 해당 노드가 가지고 있는 데이타나 자식 노드들을 모두 포함하는 MBR에 의하여 표현된다. 일반적으로 R-트리의 노드의 크기는 B^+-트리와 마찬가지로 디스크 페이지 크기의 배수로 정하여진다. R-트리의 리프 노드는 해당 리프 노드가 포함하고 있는 각 데이타 객체의 MBR과 데이타의 id의 쌍들을 저장하고 있고, 리프 노드를 제외한 R-트리의 노드들은 해당 노드의 자식 노드들의 MBR과 자식 노드들의 포인터의 쌍들을 저장하고 있다. B^+-트리와 마찬가지로 R-트리는 디스크 저장 공간의 효율적인 이용을 위하여 루트 노드를 제외한 노드들은 최소 m개(일반적으로 최대 저장 개수의 1/2개)의 엔트리를 지닌다. 그리고 트리의 노드가 저장할 수 있는 최대 자식 노드의 개수를 팬아웃(fanout)이라고 한다. B^+-트리는 일반적으로 하나의 데이타 검색을 처리하기 위하여 루트 노드부터 리프 노드까지 하나의 패스만 방문하면



(a)



(b)

그림 1 R-트리 예

되지만, R-트리에서는 노드들의 MBR이 겹쳐질(overlap) 수 있으므로 하나의 검색을 처리하기 위하여 여러 패스에 있는 트리 노드들을 방문하게 될 수도 있다. 이 점이 B-트리 계열의 인덱스와 가장 큰 차이점이다.

그림 1은 팬아웃이 3인 2차원 R-트리의 예이다. 루트 노드인 노드 a에는 각각 하위 중간 노드인 노드 b와 노드 c의 영역을 나타내는 MBR R1, R2와 노드 b, 노드 c를 가리키는 포인터가 저장되어 있다. 일반적으로 R-트리는 디스크에 저장되므로 이러한 경우 포인터는 디스크 페이지 번호가 된다. 마찬가지로 각 중간노드에는 중간노드에 포함된 하위 노드들의 MBR과 해당 노드를 가리키는 포인터가 저장된다. 위의 R-트리에서 q1 영역과 겹치는 데이타를 찾기 위한 범위 질의를 수행할 경우 루트 노드에서부터 자식 노드의 각 MBR과 질의 영역을 비교하여 만족하는 노드들을 찾아나가기 때문에 실제 만족하는 데이타는 r12뿐이지만, 노드 a, b, c, e, f의 5개 노드를 검사해야만 한다.

R-트리의 기본 구조는 다양한 응용 환경의 인덱스 구조에서 이용되고 있다. 많은 다차원 인덱스 기법들(R^{+}-tree[20], R^{*}-tree[4], Hilbert R-tree[11], SR-tree[12], X-tree[5] 등)이 R-트리를 기반으로 하여 제안되어 왔다. 뿐만 아니라, 시공간 데이타베이스나 이동 객체 데이타베이스 응용을 위해서도 다양한 R-트리 기반 인덱스 구조들(TPR-tree[19], R^{exp}-tree[18] 등)이 제안되어 왔다. 하지만, 이러한 인덱스 구조에서도 데이타의 변경은 기본적으로 이전 데이타의 삭제 후 새로운 데이타를 삽입하는 방법을 이용하고 있다.

2.2 이동 객체를 위한 인덱스 구조

빈번한 데이타의 변경을 고려하고 있는 가장 대표적인 응용이 이동 객체 데이타베이스[8] 분야이다. 이동 객체의 위치를 저장하기 위한 인덱스 구조에 대한 연구가 활발히 진행 중이다. 이러한 연구들은 저장할 이동 객체의 위치의 종류에 따라 2가지로 나눌 수 있다. 하나는 이동 객체의 이동한 모든 경로를 저장하는 방법[16,22]이고, 다른 하나는 이동 객체의 현재 위치를 저장하는 기법이다. 본 논문은 후자의 문제를 다룬다. 이동 객체의 현재 위치를 저장하는 기법에 대한 대부분의 연구[2,13,18,19]는 객체의 위치를 단순한 선형 함수로 가정하고 객체의 속도와 특정 시점의 위치만을 저장하는 기법들이다. 그리고 객체의 속도가 변경되었을 때만 데이타베이스를 변경하게 된다.

이동객체의 현재 위치를 검색하기 위한 R-트리 기반 인덱스 구조로 가장 대표적인 것이 TPR-tree[19]이다. TPR-tree는 트리의 각 노드와 MBR에 공간상의 위치를 저장하는 대신 시간에 따른 위치를 표현하는 함수를 저장한다. 이러한 함수는 두개의 파라미터 (x, v)로 표현된다. x는 특정 기준 시간의 위치이고, v는 속도이다. 따라서 특정 시간 t의 위치 p는 $x+v \cdot t$와 같이 계산된다. 따라서 TPR-tree는 객체가 같은 방향으로 동일한 속력으로 움직일 경우, 즉 등속도 운동을 할 경우, 파라미터의 값이 변경되지 않으므로 R-트리의 변경 없이 색인을 유지할 수 있는 장점이 있다. 하지만, MBR의 크기는 객체가 이동함에 따라 매우 커질 수 있기 때문에 색인의 검색 성능이 떨어질 수 있는 단점이 있다. 따라서 주기적으로 MBR의 값을 조정해줄 필요가 있다.

보다 근본적인 문제점은 TPR-tree를 포함한 대부분의 기존 기법들이 객체의 위치를 예측할 수 있는 것으로 가정하고 있다는 점이다. 객체의 움직임을 단순한 선형 함수로 가정하고 있기 때문에, 이러한 기법에서는 객체의 속도를 저장하고, 속도가 변경되지 않는 동안은 변경이 필요가 없는 점을 이용하고 있다. 이러한 방법은 객체가 항상 같은 속력을 가지고 같은 방향으로 이동하게 되는, 예를 들어 고속도로를 이동하고 있는 자동차의 움직임이나 비행기나 선박 등과 같은 특수한 경우에는 데이타의 변경을 줄일 수 있어 효과적일 수 있다. 하지만, 실제 복잡한 응용 환경에서는 객체가 등속도 운동을 하는 경우가 거의 없으므로 실용적이지 못하다. 결과적으로 이러한 선형 함수로의 근사 기법은 일반적인 변경이 빈번히 발생하는 데이타의 색인을 위해서 적용될 수는 없다. 본 논문에서는 이렇게 선형 함수로의 근사기법을 적용할 수 없는 경우에 R-트리 색인의 변경 연산을 효율적으로 처리할 수 있는 방법을 제안한다.

공간 객체에서 변경연산의 비용을 줄이기 위하여 R-트리를 이용하지 않고 해싱을 이용하여 극복하려는 시도

[21]도 있었다. 이 방법은 전체 공간을 일정한 개수의 격자로 자르고 객체가 속한 격자의 번호만을 데이타베이스에 저장한 후 객체가 다른 객체로 이동한 경우만 데이타베이스에 변경 연산을 수행한다. 이 방법은 위의 연구들과는 달리 복잡한 객체의 움직임에도 적용이 가능하지만, 객체의 위치가 편중된 경우 오버플로에 대한 처리에 대한 고려가 되어있지 않으므로 객체들의 위치 분포에 따라 성능이 급격히 나빠질 수 있는 단점이 있다.

[14]는 본 논문에서 제안하고 있는 리프 갱신 기법의 기본적인 아이디어를 제시하고 있다. 본 논문은 [14]에서 제안했던 리프 갱신 기법의 아이디어를 이론적으로 구체화하여 보완, 개선하였고, 리프 갱신 기법의 비용을 수학적으로 분석, 제시하였다.

3. 문제 정의

이 장에서는 본 논문에서 다루고 있는 문제를 정의하고, 기본적인 가정을 소개한다. 본 논문에서 풀고자 하는 문제는 빈번히 일어나는 변경을 효율적으로 처리해 줄 수 있는 R-트리의 갱신 알고리즘을 고안하는 것이다. 이를 위하여 다음과 같이 데이타 모델을 정의한다.

데이타베이스의 인덱스에 저장되는 데이타셋 D는 다음과 같이 정의된다.

$$D=\{o_1, o_2, \ldots, o_n\}$$

여기서 n은 데이타의 개수이고, o_i는 i번째의 데이타 객체를 지칭한다. 본 논문에서는 데이타의 변경을 처리하는 알고리즘에 대해서 논의하는 것이고, 데이타의 삽입이나 삭제 알고리즘은 기존의 R-트리와 동일하게 이용하므로 데이타의 삽입 삭제는 알고리즘의 성능에는 영향을 미치지 않는다. 따라서 이후 논문에서는 데이타의 개수는 n개로 고정된 것으로 가정하도록 하겠다.

하나의 데이타 객체 o_i는 다음과 같이 정의된다.

$$o_i=(id_i, v_i)$$

하나의 객체는 객체를 구별하는 유일한 식별자인 id_i와 객체의 값 v_i로 표현된다. 본 논문에서는 다차원 데이타를 대상으로 하므로 v_i는 다음과 같이 d차원의 한 점으로 정의된다.

$$v_i=(a_1, a_2, \ldots a_d)$$

예를 들어, 등록된 자동차들의 위치를 추적하는 응용 환경이라면 id_i는 자동차 번호와 같은 고유한 등록번호가 되고, v_i는 자동차에 설치된 GPS에서 측정한 자동차의 현재 위치로서 2차원 상의 한 점 (x, y)의 형태로 나타난다. 만약 응용 환경이 공장에 설치된 수많은 센서에서 압력, 온도, 습도의 값이 계속적으로 측정되고, 이를 모니터링하는 시스템이라면 id_i는 센서의 고유번호가 되고, v_i는 (압력, 온도, 습도)의 3차원 상의 점으로 나타난다.

이러한 정의 아래에서 본 논문에서 이야기하는 변경 요청은 객체의 id와 객체의 새로운 값의 쌍인 (id, v_{new})로 표현될 수 있다.

이 때 변경될 새로운 값 v_{new}가 이전의 값 v_{old}와 전혀 관계없이 d차원 공간내의 임의의 한점이 된다면, 이러한 변경은 이전의 데이타를 삭제하고 전혀 새로운 하나의 데이타를 입력하는 것과 다르지 않다. 하지만 실제 응용 환경에서는 v_{new}는 v_{old}와 연관성이 존재한다. 예를 들어 실제 많은 응용 환경에서 v는 연속적인 값을 불연속적으로 측정한 값이므로 새로운 값 v_{new}는 이전의 값인 v_{old}의 값에 임의의 변량 δ가 더해진 $v_{old}+\delta$의 형태가 된다. 예를 들어, 자동차의 위치를 추적하는 응용의 경우 새로운 위치인 v_{new}는 이전의 위치 v_{old}에서 이전 측정시간 이후 새롭게 측정된 시간까지의 물체의 이동 거리 δ를 더한 값이 된다. 그리고 위치 추적이라는 목적을 달성하기 위해서는 측정 시간 간격이 그리 크지 않을 것이므로 변량인 δ는 d차원의 전체 크기와 비교하였을 때는 충분히 작은 값을 가진다. 본 논문은 변경 요청에서 발견할 수 있는 이러한 경향을 변경 요청의 지역성이라고 다음과 같이 정의한다.

정의 1. 변경 요청의 지역성

변경 요청에 의해 새롭게 변경될 값과 이전의 값과의 관련성을 변경 요청의 지역성이라고 정의한다. 새롭게 변경될 값과 이전의 값이 가까울수록 변경 요청의 지역성이 높고, 이전의 값과 가깝지 않을수록 변경 요청의 지역성이 낮다. 이러한 성질에 따라 변경 요청의 지역성 L은 다음과 같이 정의할 수 있다.

$$L=1-\frac{\delta}{\delta_{MAX}}$$

여기서 δ는 새롭게 변경될 값 v_{new}와 이전의 값 v_{old}의 거리이다. 각 값들이 존재할 수 있는 d차원 의 전체 공간을 U라고 하면, v_{new}와 v_{old}는 U속의 한 점으로 표현된다. 따라서 δ는 다음과 같이 정의 된다.

$$\delta=|v_{new}-v_{old}|(v_{new}\in U, n_{old}\in U)$$

그리고 δ_{MAX}는 d차원 공간 U상에서 임의의 두 점이 가질 수 있는 거리의 최대 값으로 다음과 같이 정의된다.

$$\delta_{MAX}=\max_{\forall p\in U, \forall q\in U}|p-q|$$ □

실제 많은 응용 환경을 살펴보면 변경이 빈번하게 일어나고 중요한 경우들은 이동 객체나 스트림 데이타 등과 같이 연속적으로 변하는 값을 계속적으로 시스템이 필요로 하는 것이므로 변경 요청에 지역성이 높은 경우

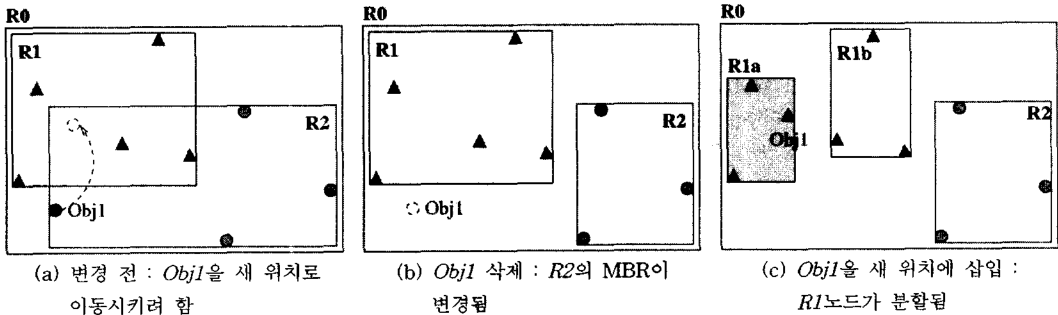
가 많다. 따라서 본 논문에서 다루는 변경 요청은 이러한 지역성이 높다고 가정하고, 변경 요청의 지역성이 높은 경우 이를 이용하여 R-트리의 갱신 요청을 효율적으로 처리할 수 있는 방법을 고안하고자 한다.

4. 리프 갱신 기법

4.1 기본 아이디어

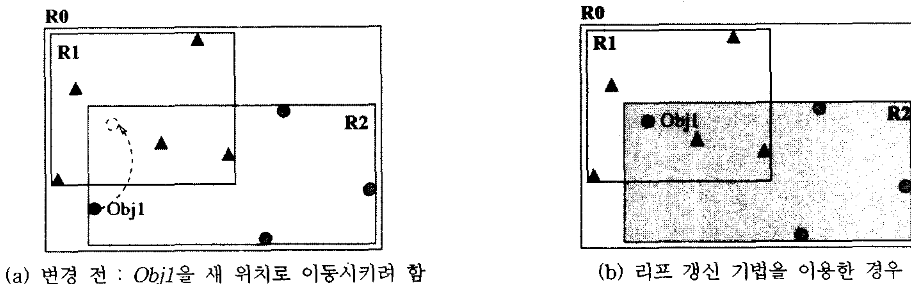
2.1절에서 언급했듯이 기존의 R-트리는 특별한 변경 연산이 존재하지 않는다. 따라서 변경 요청이 들어오면 이전의 데이타를 인덱스에서 삭제하고, 새로운 데이타를 삽입하여야 한다. 하지만, 이러한 방법은 R-트리와 같은 트리 구조의 인덱스 구조에서는 큰 문제를 야기할 수 있다. 예를 들어, 그림 1(a)와 같은 상황에서 *Obj1*의 값이 현재 위치에서 그림과 같이 변경되었다고 가정하면 기존의 R-트리에서는 이전 값을 인덱스에서 삭제하고 새로운 값을 인덱스에 삽입하여야만 한다. 이때 리프 노드의 최대 저장 가능 객체의 수를 5라고 가정하자. 위와 같은 경우 *Obj1*을 우선 삭제하면 객체를 삭제한 노드 *R2*는 그림 1(b)와 같이 MBR이 변경되어야 한다. 그리고 이는 *R0* 노드에도 이러한 MBR의 조정이 반영되어야 한다. 실제 이러한 MBR의 조정은 트리의 루트까지 연속적으로 파급될 수 있다. 뿐만 아니라, 경우에 따라서는 노드에 언더플로가 발생할 수 있고, 이러한 경우 트리 노드의 병합이 발생한다. 최악의 경우에는 노드의 병합이 루트까지 파급되어 전체 트리의 높이가 1 줄어들 수도 있다. 그리고 여기서 새로운 값을 트리에 삽입하면 현재 트리 구조에서 적당한 하부 트리를 계속 따라가서 결국 새로운 값은 *R1* 노드에 삽입되어야 하는데 이 경우 *R1* 노드에 오버플로가 발생하므로 노드가 그림 1(c)와 같이 분할되어야 한다. 이러한 분할 역시 트리의 루트까지 파급될 수도 있고, 최악의 경우에는 새로운 루트가 만들어질 수도 있다. 위의 예제의 경우 위치 변경을 위하여 최소 4번의 디스크 쓰기 접근(노드 *R2, R1a, R1b, R0*)이 필요하다. 그리고 트리 노드의 분할이나 병합은 트리의 루트까지 파급될 수 있으므로 이러한 경우에는 훨씬 많은 수의 디스크 접근이 필요하고, 전체 트리의 구조가 변경될 수 있다.

하지만, 위의 경우는 *Obj1*의 변경된 값은 여전히 *R2* 노드 내부에 존재하므로 이 경우 *R2*노드 내부의 *Obj1*의 위치만 변경하더라도 그림 2(b)와 같이 R-트리의 구조적 정확성에는 아무런 이상이 없다. 물론 MBR이 최소 경계 사각형이 되지 않지만, 노드 *R2*의 MBR내에 *Obj1*이 존재하므로 실제 트리를 검색하거나 조작하는 데는 아무런 문제가 없다. 이와 같이 새로운 위치가 기존의 리프 노드의 MBR의 범위를 벗어나지 않는 경우 노드 내부의 위치만을 변경하고, 트리의 구조 변경을 하지 않고 리프 노드만을 변경하는 방법이 본 논문에서 제안하는 리프 갱신 기법이다. 리프 갱신 기법으로 위의



(a) 변경 전 : *Obj1*을 새 위치로 이동시키려 함 (b) *Obj1* 삭제 : *R2*의 MBR이 변경됨 (c) *Obj1*을 새 위치에 삽입 : *R1*노드가 분할됨

그림 2 기존의 R-트리 갱신 기법의 수행 예



(a) 변경 전 : *Obj1*을 새 위치로 이동시키려 함 (b) 리프 갱신 기법을 이용한 경우

그림 3 리프 갱신 기법의 수행 예

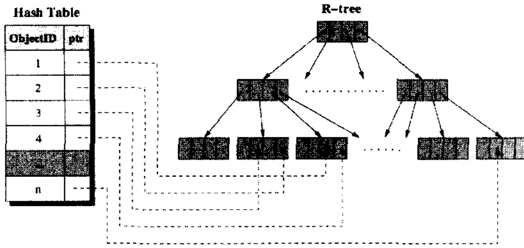


그림 4 리프 접근 해시 테이블의 예

예를 처리하는 경우 *R2*노드에 대한 디스크 쓰기 접근 1회로 처리가 가능하다. 따라서 이동 객체의 위치 변경 비용을 크게 줄 일 수 있다. 특히 이동 객체의 경우와 같이 변경 요청에 지역성이 큰 경우 리프 갱신 기법의 효용성은 더욱 커지게 된다.

자세한 리프 갱신 기법의 알고리즘은 4.3절에서 설명한다.

4.2 리프 접근 해시 테이블

기존의 R-트리를 이용하여 변경 요청을 처리하는 방법이 가지는 또 하나의 문제점은 변경 요청이 3.1절의 문제 정의에서 언급했듯이, (id, v_{new})의 형태로 나타나는데 반하여 R-트리의 삭제 알고리즘은 삭제할 값을 인자로 주어야 하므로 이전 값 v_{old}를 알아야 한다는 점이다. R-트리의 검색 역시 *id*에 의하여 검색하는 것이 아니고, 저장된 값 *v*에 의하여 검색할 수밖에 없기 때문에, v_{old} 값을 찾기 위해서는 전체 R-트리를 순회하며 *id* 값이 일치하는 객체를 찾을 수밖에 없고, 실제 응용 환경에서 이러한 방법을 적용할 수는 없다. 따라서 실제 응용 환경에서는 *id* 값에 해당하는 값 *v*를 찾기 위하여 (id, v)의 쌍을 해시 테이블과 같은 별도의 저장 구조를 이용하여 저장하고 있어야 한다.

뿐만 아니라, 해시 테이블을 이용하여 v_{old} 값을 찾았다고 하더라도 기존 R-트리의 삭제 알고리즘을 이용하게 되면 v_{old} 값을 가지고 트리의 루트에서부터 리프 노드를 찾아나가야 한다. 2.1절에서 설명했듯이 R-트리의 검색은 B-트리와는 달리 노드의 MBR 간에 겹침이 있는 영역을 검사해야 할 경우 해당하는 모든 노드를 방문하여야 하므로 많은 노드를 방문하여야 하는 문제점이 있다. 트리의 루트부터 리프 노드를 찾아 나가게 되므로 최소 트리의 높이만큼의 디스크 접근이 필요하고, 최악의 경우 찾고자하는 점이 있는 영역에 트리의 모든 노드 MBR이 겹쳐있다면 트리의 모든 노드를 방문하여야 하는 문제점을 지닌다.

이러한 문제점을 해결하기 위하여 본 논문에서 제안하는 리프 갱신 기법은 리프 접근 해시 테이블이란 구조를 이용한다. 기존의 R-트리의 삭제, 삽입 기법의 경우 트리의 루트부터 리프 노드로 트리를 따라 순회하는 부분이 필요하지만, 리프 갱신 기법은 우선 트리의 리프 노드의 MBR과 새로운 위치 v_{new}를 비교하고 리프 갱신에 의해 처리할 수 있는 경우는 리프 노드만을 접근하면 되기 때문에 처음부터 트리의 루트부터 트리를 순회할 필요가 없다. 앞에서 언급했듯이 기존의 R-트리의 삭제 알고리즘을 전체 트리 순회 없이 이용하기 위해서는 해시 테이블 등을 이용하여 (id, v_{old})의 값을 유지하고 있어야한다. 리프 갱신 기법에서는 트리의 루트부터 따라 내려오지 않아도 갱신을 처리할 수 있으므로 객체의 *id*와 객체가 실제 저장된 리프 노드의 주소를 해시 테이블에 유지하고 *id*값에 의해 직접 트리의 리프 노드에 접근하면 된다. 이를 이용하면 루트 노드부터 리프 노드까지의 불필요한 순회를 제거할 수 있으므로 갱신 연산 처리 비용을 줄일 수 있다. 본 논문에서는 이와 같이 특정 객체를 저장하고 있는 트리의 리프 노드에 직접 접근하기 위해 객체의 *id*와 객체가 저장된 리프 노드의 주소를 저장하고 있는 해시 테이블을 리프 접근 해시 테이블이라 명명한다. 그림 3은 이러한 리프 접근 해시 테이블의 예이다.

4.3 리프 갱신 기법 알고리즘

알고리즘 1은 리프 갱신 기법의 구체적인 알고리즘이다. 리프 갱신 기법의 구체적인 알고리즘은 알고리즘 1과 같다. 알고리즘의 입력은 변경할 객체의 *id*와 새로운

알고리즘 1 리프 갱신 알고리즘

```
Procedure LeafUpdate( id, v_new)
Input:
    id      // 변경할 객체의 ID
    v_new   // 변경될 값
begin
    // 리프 접근 해시 테이블을 이용하여 객체를 저장하고 있
       는 리프 노드를 찾는다.
    leafID ← FindLeafNode( id);
    N ← ReadNode(leafID);   // 리프 노드를 읽는다.
    if v_new ∈ N.MBR then
        // 새로운 위치가 리프 노드의 MBR내에 존재하면 리
           프 갱신으로 처리한다.
        N.Entry[ id].pos ← v_new;
        WriteNode(N);
    else
        // 새로운 위치가 MBR을 벗어나면 기존 방법을 이용
           하여 갱신한다.
        v_old ← N.Entry[ id].pos;
        Delete( id, v_old);
        newLeafID ← Insert( id, v_new);
        // 리프 접근 해시 테이블을 변경한다.
        UpdateLeafAccessHashTable( id, newLeafID);
    end
end
```

값의 쌍인 (id, v_{new})이다. 우선 리프 접근 해시 테이블을 이용하여 id에 해당하는 객체가 저장된 리프 노드를 찾는다. 그리고 리프 노드의 MBR이 새로운 값을 포함할 수 있으면 이 갱신을 리프 갱신의 방법으로 처리하여 리프 노드에 저장되어 있는 v_{old}의 값만 v_{new}로 변경하고 갱신을 종료한다. 만일 v_{new}가 MBR을 벗어나면 리프 갱신 기법을 적용할 수 없으므로 기존의 변경 방법을 이용하여 이전 값을 삭제하고, 새로운 값을 삽입한다. 리프 갱신 기법을 이용하게 되면 변경이 단지 리프 노드에만 지역적으로 영향을 주게 되므로 전체 트리 구조의 변경을 수행하지 않는다. 기존의 삭제, 삽입 기법을 이용하여 변경을 처리하면 삭제를 수행하고 새로 삽입을 하는 과정에서 R-트리의 노드 간 MBR 겹침을 최소화하려는 최적화 시도를 수행하게 되므로 삽입 결과가 리프 갱신을 이용하였을 때와는 다를 수 있고 더 최적화되었을 수가 있지만, 이러한 시도는 3.1절의 예에서 살펴봤듯이 노드 간의 병합이나 합병을 유발하므로 변경 비용을 크게 증가시킨다.

정리 1. 리프 갱신 기법은 R-트리의 정확성을 보장한다.

증명 :

R-트리가 정확하려면 모든 부모 노드의 MBR은 자신의 모든 자식 노드 혹은 리프 노드의 경우 자신의 모든 자식 개체의 MBR을 포함하고 있어야한다. 따라서 R-트리의 정확성을 보장하기 위해서는 변경을 수행한 후에도 이 포함 조건이 만족하여야만 한다. 리프 갱신 기법에서 이 조건이 만족하는 지 살펴보자. 리프 갱신 기법이 변경 요청을 처리하는 방법은 경우에 따라 두 가지로 나뉜다. 첫 번째, 새로운 값이 리프 노드의 MBR을 벗어나는 경우, 기존의 R-트리의 삭제, 삽입 알고리즘을 그대로 이용하므로 R-트리의 정확성을 보장한다. 두 번째, 새로운 값이 리프 노드의 MBR 속에 포함하는 경우, 리프 갱신의 방법으로 갱신을 처리하면 해당 리프 노드 내부의 객체의 값만 변경하게 되는데, 이 경우 리프 노드를 제외한 모든 노드는 변경된 사항이 없으므로 위의 포함 조건을 만족한다. 그리고 리프 노드의 경우 새로운 위치가 MBR 속에 포함되는 경우에 해당하므로 리프 노드가 가진 모든 객체는 역시 리프 노드의 MBR에 포함된다. 따라서 위의 포함 조건을 만족한다. 그러므로 리프 갱신 기법은 R-트리의 정확성을 보장한다. □

4.4 확장 리프 MBR

리프 갱신 기법의 경우 객체가 어떠한 MBR의 가장 자리에서 있는 경우 쉽게 MBR 밖으로 이동할 수 있다. 만일 응용 환경이 변경 연산의 효율적인 처리가 아주 중요하고 R-트리의 노드간의 겹치는 영역이 증가하는 것이 크게 문제가 되지 않는다면, 처음 트리를 생성할 때부터 항상 리프 노드의 MBR을 인위적으로 일정한 크기만큼 확장하여 사용할 수도 있다. 이렇게 리프 노드의 MBR을 일정한 크기 ε만큼 확장한 것을 확장 리프 MBR이라 정의한다. 그림 4는 확장 리프 MBR의 예이다. 확장 리프 MBR은 다음과 같이 정의할 수 있다.

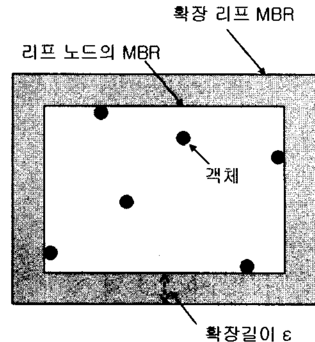


그림 5 확장 리프 MBR의 예

정의 2. 확장 리프 MBR (Extended leaf MBR)

O를 확장 리프 MBR이 포함하고 있는 객체의 집합이라 가정하자. 이때, O가 n개의 객체를 가지고 있다면 다음과 같이 정의할 수 있다.

$$O=\{o_1, o_2, \ldots, o_n\}$$

각 객체의 값은 d차원의 한 점으로 나타날 수 있다. 따라서 o_i의 값 v_i는 다음과 같이 정의된다.

$$v_i=(p_{i_1}, p_{i_2}, \ldots, p_{i_d}),\ where\ \ 1\le i\le n$$

d차원 상의 확장 리프 MBR은 다음과 같이 각 차원 상의 범위 $EI_k(1\le k\le d)$에 의해 정의될 수 있다.

$$EMBR=(EI_1, EI_2, \ldots, EI_d)$$

여기서 확장 리프 MBR의 확장 길이를 ε이라 하면 EI_k는 다음과 같이 정의된다.

$$EI_k=\left[\min_{1\le i\le n}(p_{i_k}-\varepsilon), \max_{1\le i\le n}(p_{i_k}+\varepsilon)\right]$$ □

확장 범위 ε는 인덱스가 처음 생성될 때 주어지는 상수 값이다. 만일 ε가 0이면 이는 확장 리프 MBR을 사용하지 않고, 기존 R-트리의 MBR을 이용한다는 의미가 된다. ε가 커지면 새로운 값이 이전 MBR 내에 존재하는 확률이 커지므로 갱신을 리프 갱신을 이용 처리할 수 있게 되어 갱신 연산의 비용을 줄일 수 있다. 하지만, 리프 노드 간에 겹치는 영역이 넓어지므로 검색 성능이 저하될 수 있다.

5. 리프 갱신 기법의 수행 비용 분석

이 장에서는 리프 갱신 기법의 수행 비용을 기존의 R-트리의 삭제, 삽입 기법을 이용했을 때와 비교 분석

해보겠다.

기존 R-트리의 경우 갱신 연산은 삭제, 삽입 연산을 수행하는 것으로 이루어진다. 그리고 삭제 연산을 수행하기 위해서는 3.2절에서 언급했듯이 미리 주어진 *id*를 이용하여 이전의 값 v_{old}를 찾아야 한다. 따라서 기존의 R-트리에서의 갱신 연산 수행 비용 $C_{original}$은 다음과 같이 계산될 수 있다.

$$C_{original} = C_{hash} + C_{delete} + C_{insert}$$

여기서 C_{hash}는 주어진 *id*로부터 이전의 값 v_{old}를 찾기 위해 해시 테이블을 검색하는 데 필요한 비용으로 1회의 디스크 읽기 연산 비용으로 볼 수 있다. C_{delete}와 C_{insert}는 각각 R-트리로부터 데이타 하나를 삭제하는 비용과 삽입하는 비용이다.

마찬가지의 방법으로 리프 갱신 기법을 사용했을 때의 갱신 연산 수행 비용을 계산해보자. 리프 갱신의 경우 우선 해시 테이블로부터 *id*에 해당하는 객체를 지니고 있는 리프 노드를 찾은 다음 새로운 값 v_{new}가 리프 노드의 MBR에 포함되면 리프 갱신으로 처리하고, 그렇지 않으면 기존의 R-트리와 마찬가지로 삭제, 삽입 연산을 통해서 갱신 연산을 수행한다. 따라서 새로운 위치가 리프 노드의 MBR에 포함될 확률을 $p_{include}$라고 하고, 리프 갱신의 방법을 사용하였을 경우의 비용을 $C_{leafUpdate}$라고 하면 리프 갱신 기법의 총 수행 비용 $C_{leafUpdateTotal}$은 다음과 같다.

$$C_{leafUpdateTotal} = C_{hash} + p_{include} \cdot C_{leafUpdate} + (1 - p_{include}) \cdot (C_{delete} + C_{insert})$$

마찬가지로 여기서 C_{hash}는 디스크 읽기 1번이고, $C_{leafUpdate}$는 리프 노드를 한번 읽고 쓰면 되므로 2번의 디스크 접근 비용이 된다.

이때, 리프 갱신을 이용하였을 때 얻을 수 있는 이득은 $C_{original} - C_{leafUpdateTotal}$을 계산하면 구할 수 있다. 위의 두 식을 이용하여 $C_{original} - C_{leafUpdateTotal}$를 계산하면 결과는 다음과 같다.

$$C_{original} - C_{leafUpdateTotal} = p_{include} \cdot (C_{delete} + C_{insert} - C_{leafUpdate})$$

여기서 $C_{leafUpdate}$는 리프 노드를 한번 읽고 쓴 비용이므로 2 I/O이다.

$$C_{leafUpdate} = 2$$

그리고 기존의 R-트리에서 데이타를 삭제하기 위해서는 노드의 MBR 간에 겹침이 한번도 없는 최상의 경우를 가정하더라도 트리의 루트로부터 리프까지 노드들을 한번씩 디스크에서 읽어야 하고, 데이타를 삭제하였을 때 단지 리프 노드만 변경하면 되는 최상의 경우에도 1번의 디스크 쓰기를 수행하여야 한다. 따라서 트리의 높이를 *h*라고 했을 때 최상의 경우 데이타를 삭제하는데 드는 비용 C_{delete}는 최소 $h+1$이다. 따라서 다음 부등식이 성립한다.

$$C_{delete} \geq h+1$$

마찬가지로 데이타를 삽입하기 위한 비용인 C_{insert}역시 최상의 경우 $h+1$의 비용이 필요하므로 다음 부등식이 성립한다.

$$C_{insert} \geq h+1$$

따라서 위의 결과들을 이용하여 수행 비용의 이득을 계산하면 아래와 같은 결과를 얻을 수 있다.

$$C_{original} - C_{leafUpdateTotal} = p_{include} \cdot (C_{delete} + C_{insert} - 2) \geq 2 \cdot p_{include} \cdot h$$

그러므로 리프 갱신 기법은 최악의 경우에도 기존의 기법에 비해 $2 \cdot p_{include} \cdot h$ 만큼의 수행 비용을 줄일 수 있다. 또한 새로운 값이 이전 리프 노드의 MBR에 포함될 확률 $p_{include}$는 변경 요청에 지역성이 클수록 커지므로 변경 요청에 지역성이 높으므로 리프 갱신 기법을 이용했을 때 이득이 커진다. 예를 들어, $p_{include} = 0.6$ 이라고 가정하고, 즉, 전체 갱신 요청의 60%를 리프시켜 처리할 수 있다고 가정하고, 트리의 높이가 4라고 가정하면 MBR 간에 겹치는 영역이 전혀 없고, 트리의 삽입 삭제 시 노드간의 병합이나 분할이 전혀 일어나지 않는 기존 기법에 가장 유리한 환경을 가정하더라도 기존 기법의 디스크 접근 회수는 11회($C_{hash}+2h+2$)이고 리프 갱신 기법의 경우 6.2회($C_{hash}+0.6\times2+2h+2$)에 불과하므로 디스크 접근 회수를 1/2가까이 줄일 수 있다. 물론, 디스크의 노드들이 버퍼링되는 상황에서는 결과가 약간 다를 수 있지만, 버퍼링에 의해서 줄어드는 비용은 기존의 기법과 리프 갱신 기법이 유사하게 나타나게 되고, 일반적인 경우 트리의 노드간의 MBR이 겹치고 삽입이나 삭제 시 노드의 분할이나 병합이 일어나게 되므로 리프 갱신 기법에 의해 얻을 수 있는 비용은 일반적으로 더 커진다.

6. 실험

이 장에서는 실험을 통하여 본 논문에서 제안한 리프 갱신 기법을 R^*-트리에 구현하여 기존의 기법과 비교 분석한다.

6.1 실험 환경

실험은 Pentium 1GHz CPU, 512MB의 메모리, 40GB E-IDE HDD를 가진 Linux 기계에서 수행되었다. 디스크 페이지의 크기는 4KB로 가정하였다. 실험을 위해서는 이동 객체 분야에서 실험을 위해 널리 사용되는 데이타 생성기인 [23]를 이용하여 데이타를 생성하여

사용하였다. 데이타 분포에 따른 성능을 비교하기 위하여 표 1과 같이 4가지의 데이타를 생성, 실험하였다. 실험은 2차원 점 데이타를 대상으로 이루어졌고, 주어진 분포에 따라 $[0,1]^2$의 2차원 공간 속의 점을 생성하여 사용하였다.

표 1 실험 데이타

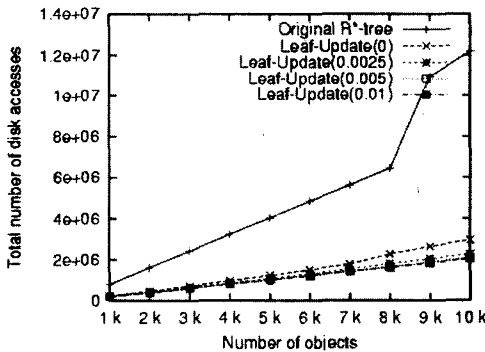
	초기 분포	이동방향
U-R dataset	Uniform 분포	무작위
U-D dataset	Uniform 분포	방향성 지님
G-R dataset	Gaussian분포	무작위
G-D dataset	Gaussian분포	방향성 지님

실험 결과에서 'Original R^*-tree'는 전통적인 기존의 R^*-트리이고, 'Leaf-Update'는 리프 갱신 기법을 적용한 R^*-트리를 의미한다. 그리고 'Leaf-Update' 뒤의 숫자는 확장 리프 MBR의 확장길이 ε를 의미한다. 확장 리프 MBR의 효과를 비교 분석하기 위하여 ε의 크기에 따라 각 0, 0.0025, 0.005의 세 가지의 확장 길이를 가지는 리프 갱신 기법을 이용하였다. 여기서 ε가 0인 것은 확장 리프 MBR을 이용하지 않고 기존의 MBR을 이용한 R-트리이다.

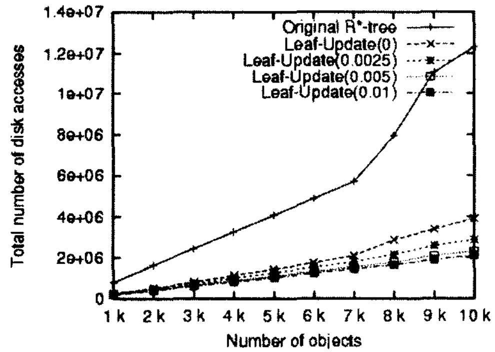
모든 실험에서 실험이 수행된 시스템의 하드웨어 성능에 실험 결과가 영향을 받지 않도록 하기 위해 디스크 접근 회수를 측정하였다. 이는 기존의 R-트리에서 리프 갱신 기법을 지원하기 위해 추가된 알고리즘이 단순하므로 리프 갱신 기법에 의한 성능의 차이는 전적으로 디스크 접근 회수에 연관이 되기 때문이다. 그리고 리프 갱신 기법과 기존 R-트리의 갱신 기법 모두 알고리즘의 수행을 위해서는 동일하게 한번의 해시 테이블 접근이 필요하기 때문에 해시 테이블에 의하여 발생하는 디스크 접근 회수는 실험 결과에서 제외하였다.

6.2 갱신 질의 성능

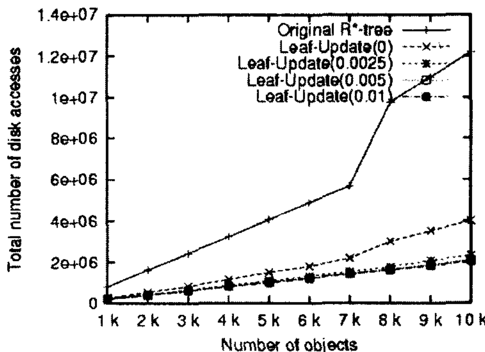
첫 번째 실험에서는 리프 갱신 기법과 기존 기법에 대하여 각 객체 당 100번의 갱신 연산을 수행하였을 때 총 디스크 접근 회수를 비교 분석하였다. 그림 5는 데이타 객체의 수를 1,000개에서 10,000개까지 변화시켜가며, 이 때의 총 디스크 접근 회수를 측정한 결과이다. 실험 결과 데이타의 개수가 증가함에 따라 기존 기법의 총



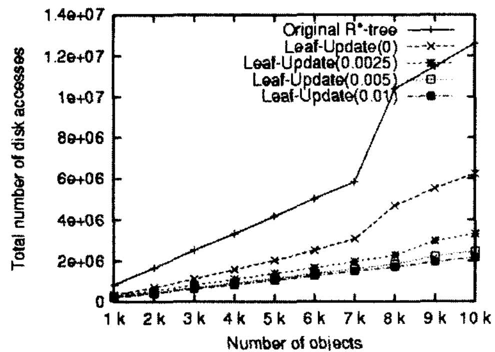
(a) U-R 데이타셋



(b) U-D 데이타셋

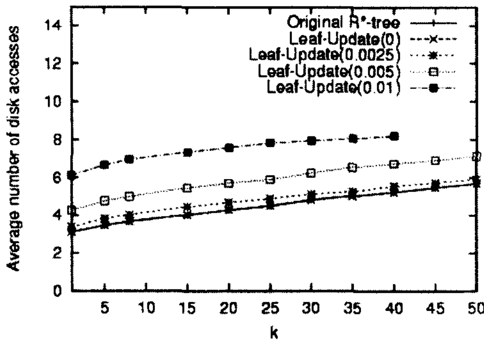


(c) G-R 데이타셋

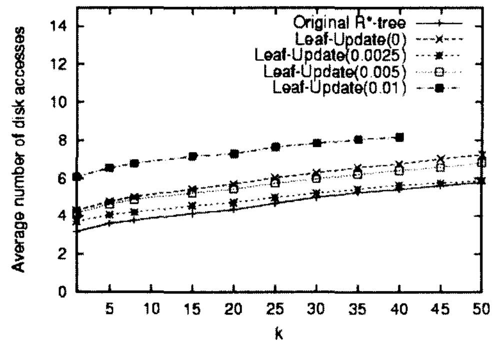


(d) G-D 데이타셋

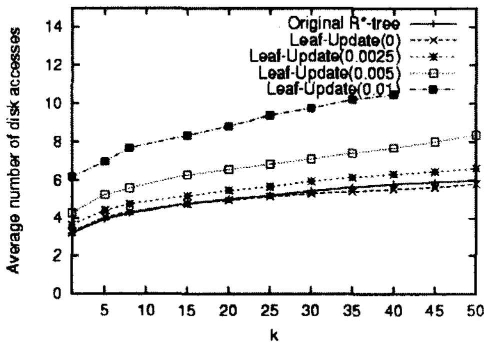
그림 6 k-최근접 질의 수행 시 평균 디스크 접근 회수



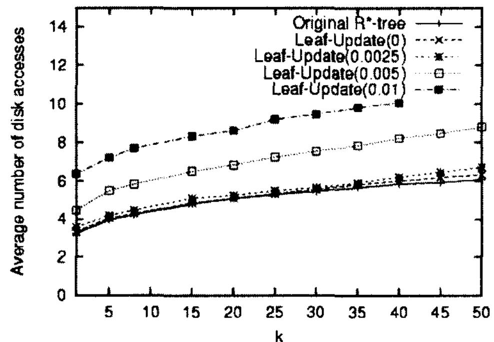
(a) U-R 데이타셋



(b) U-D 데이타셋



(c) G-R 데이타셋



(d) G-D 데이타셋

그림 7 변경 질의 수행 시 총 디스크 접근 회수

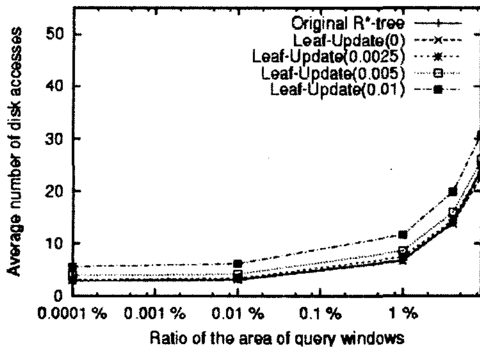
디스크 접근 회수가 더 빨리 증가함을 알 수 있었다. 그리고 확장 리프 MBR을 사용하는 경우가 디스크 접근 횟수를 더 줄일 수 있었고, ε를 증가시킴에 따라 그 효과는 더 커졌다. 이 결과는 5장에서 분석된 결과와 일치하는 것으로 ε가 증가함에 따라 새로운 값이 MBR속에 존재할 확률 $p_{include}$가 높아지기 때문에 리프 갱신 기법으로 얻을 수 있는 성능 향상이 커짐을 보여준다.

6.3 검색 질의 성능

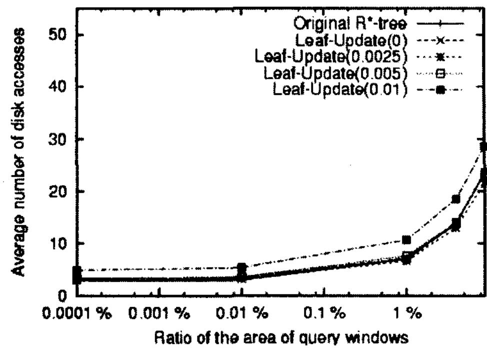
그림 7은 범위 검색 질의를 수행했을 때의 평균 디스크 접근 회수이다. 이 실험에서는 질의 범위를 전체 공간 크기의 0.00001%에서 1%까지 변경시켜가며 각 100회의 질의를 수행했을 때의 평균 디스크 접근 회수를 측정했다. 그림 7에서 x축은 로그 스케일이다. 그림 8은 k-최근접 검색 질의의 결과이다. k를 1부터 50까지 변경해가며 각 100회의 질의를 수행했을 때의 평균 디스크 접근 회수를 측정하였다. 실험 결과 리프 갱신 기법을 이용하는 경우 MBR간에 겹치는 영역이 증가하므로 검색 질의 수행 성능이 약간 나빠지지만 질의 수행 성능은 크게 차이가 나지 않았다. 특히 확장 리프 MBR의 확장 길이를 적당히 작게 잡은 경우나 확장 리프 MBR을 사용하지 않은 경우에는 그림 7과 그림 8에서 알 수 있듯 리프 갱신 기법이 기존 R-트리의 성능과 거의 같은 성능을 나타내었다. 그리고 예상했던 바와 같이 확장 리프 MBR의 ε가 커짐에 따라 MBR간에 겹치는 영역이 증가하므로 접근 횟수가 증가하였다. 하지만 ε를 0.01까지 증가시킨 경우를 살펴보면, 전체 실험 공간의 한 축 길이가 1이므로 0.01이라는 길이는 전체 길이의 1%에 해당하는 매우 큰 값으로서 실험을 위하여 극단적인 환경을 가정한 것이므로 실제 응용 환경에서 이러한 값을 ε로 이용하여야 하는 경우는 거의 없을 것이다.

6.4 실험 결과 분석

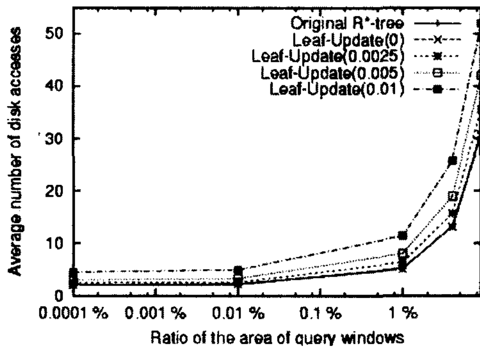
실험 결과, 리프 갱신 기법은 갱신 연산의 비용을 크게 감소시켜주는 장점이 있었다. 물론, 검색 질의 처리 시에는 약간의 오버헤드를 가짐을 확인하였다. 하지만, 확장 리프 MBR을 사용하지 않는 경우나 충분히 작은 값의 ε를 가지는 확장 리프 MBR을 가진 경우의 검색 질의 처리 성능은 기존 R-트리와 거의 동일하였다. 따라서 리프 갱신 기법은 변경 요청이 빈번한 이동 객체의 위치 추적이나 스트림 데이타 등의 응용분야에서 이용될 경우 시스템의 성능을 개선 할 수 있을 것으로 기



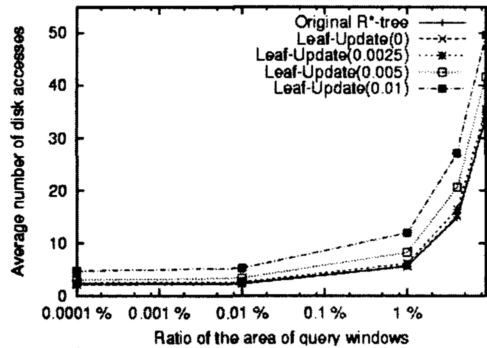
(a) U-R 데이타셋



(b) U-D 데이타셋



(c) G-R 데이타셋



(d) G-D 데이타셋

그림 8 범위 검색 질의 수행 시 평균 디스크 접근 회수

대된다. 그리고 확장 리프 MBR을 이용할 경우 변경 요청의 지역성을 더 많이 용할 수 있으므로 갱신 연산 비용을 더욱 줄일 수 있음을 확인하였다. 결과적으로, ε의 값이 커짐에 따라 변경 성능은 향상되지만, 검색 성능은 저하되므로 ε의 값은 갱신 요청의 비율이나 갱신 요청의 지역성의 정도에 따라 적정한 값을 결정하여야 한다.

7. 결 론

R-트리는 공간 데이타베이스와 같은 응용에서 가장 널리 이용되는 다차원 인덱스 구조이다. 하지만, 기존의 R-트리가 다루던 데이타는 상대적으로 정적인 데이타를 대상으로 하므로 데이타의 변경에 대한 고려가 없었다. 따라서 이동 객체의 위치 추적이나, 스트림 데이타베이스에서의 다차원 데이타 색인과 같은 데이타의 변경이 끊임없이 매우 빈번하게 일어나는 환경에서는 이러한 데이타의 변경을 효율적으로 처리해줄 수 없는 문제점이 존재한다. 이러한 문제를 해결하기 위하여, 본 논문에서는 이러한 끊임없는 변경의 경우 변경된 값과 이전의 값이 연관성을 지니는 변경 요청의 지역성이 존재함을 이용한 리프 갱신 기법을 제안한다. 제안 기법은 이동 객체의 새로운 위치가 이전에 속해 있던 MBR에 속해 있는 경우 리프 노드만을 변경하여 트리 전체 구조의 변경을 제거한다. 뿐만 아니라 리프 접근 해시 테이블을 이용하여 갱신 질의 처리 시 루트로부터의 불필요한 트리의 순회를 제거하므로 기존 기법에 비하여 갱신 연산의 비용을 크게 줄일 수 있다. 또한 제안 기법은 R-트리의 구조를 동일하게 이용하고 인덱스의 정확성을 보장하므로 기존 알고리즘을 그대로 이용할 수 있어서 기존 응용 환경에 쉽게 적용할 수 있는 장점이 있다. 본 논문에서는 제안 기법이 기존 기법에 대하여 가지는 갱신 연산의 비용 이득을 수학적으로 분석하였고, 실험을 통하여 제안 기법의 우수성을 확인하였다.

추후 연구로서, 본 논문에서 제시하고 있는 기법 및 기존의 디스크 기반 이동 객체 색인 기법을 주 기억 장치 기반 환경에 적용하였을 경우의 성능의 변화 및 주기억 장치 상에서의 효율적인 변경 연산 지원 방법에 관한 연구를 진행 중이다.

참 고 문 헌

[1] Abraham, T. and Roddick, J. F., "Survey of Spatio-Temporal Databases," GeoInformatica, Vol. 3, No. 1, pp. 61-99, 1999.

[2] Agarwal, P. K., Arge, L. and Erickson, J., "Indexing Moving Points," In Proc. of Int'l Symposium on Principles of Database Systems, ACM PODS,pp. 175-186, 2000.
[3] Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J., "Models and Issues in Data Stream Systems," In Proc. of Int'l Symposium on Principles of Database Systems, ACM PODS, pp. 1-16, 2002.
[4] Beckmann, N., Kriegel, H., Schneider, R. and Seeger, B., "The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles," In Proc. of the 1990 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 322-331, 1990.
[5] Berchtold, S., Keim, D. A. and Kriegel, H., "The X-tree : An Index Structure for High-Dimensional Data," In Proc. of 22th Int'l. Conf. on Very Large Data Bases, pp. 28-39, 1996.
[6] Brinkhoff, T., Kriegel, H. and Seeger, B., "Efficient Processing of Spatial Joins Using R-Trees," In Proc. of the 1993 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 237-246, 1993.
[7] Carney, D., Çetintemel, U., Cherniack, M., Convey C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N. and Zdonik, S. B., "Monitoring Streams - A New Class of Data Management Applications," In Proc. of 28th Int'l. Conf. on Very Large Data Bases, pp. 215-226, 2002.
[8] Forlizzi, L., Güting, R. M., Nardelli, E. and Schneider, M., "A Data Model and Data Structures for Moving Objects Databases," Proc. of the 2000 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 319-330, 2000.
[9] Gaede, V. and Günther, O., "Multidimensional Access Methods," ACM Computing Surveys, Vol. 30, No. 2, pp. 170-231, 1998.
[10] Guttman, A., "R-Trees: A Dynamic Index Structure for Spatial Searching," In Proc. of the 1984 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 47-57, 1984.
[11] Kamel, I. and Faloutsos, C., "Hilbert R-tree: An Improved R-tree using Fractals," In Proc. of 20th Int'l. Conf. on Very Large Data Bases, pp. 500-509, 1994.
[12] Katayama, N. and Satoh, S., "The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," In Proc. of the 1997 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 369-380, 1997.
[13] Kollios, G., Gunopulos, D. and Tsotras, V. J., "On Indexing Mobile Objects," In Proc. of Int'l Symposium on Principles of Database Systems, ACM PODS, pp. 261-272, 1999.
[14] Kwon, D., Lee, S., and Lee, S., "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," In Proc. of the 3rd Int'l. Conf. on Mobile Data Management, pp.113-120, 2002.
[15] Manolopoulos, Y., Nanopoulos, A., Papadopoulos, A. N. and Theodoridis, Y., "R-trees Have Grown Everywhere," Submitted to ACM Computing Surveys, http://www.rtreeportal.org/pubs/MNPT03.pdf, 2003.
[16] Pfoser, D., Jensen, C. S. and Theodoridis, Y., "Novel Approaches in Query Processing for Moving Object Trajectories," In Proc. of 26th Int'l. Conf. on Very Large Data Bases, pp. 395-406, 2000.
[17] Roussopoulos, N., Kelley, S. and Vincent, F., "Nearest Neighbor Queries," In Proc. of the 1995 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 71-79, 1995.
[18] Saltenis, S. and Jensen, C. S., "Indexing of Moving Objects for Location-Based Services," In Proc. of the 18th Int'l. Conf. on Data Engineering, pp. 463-472, 2002.
[19] Saltenis, S., Jensen, C. S., Leutenegger, S. T. and Lopez, M. A., "Indexing the Positions of Continuously Moving Objects," In Proc. of the 2000 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 331-342, 2000.
[20] Sellis, T. K., Roussopoulos, N. and Faloutsos, C., "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects," In Proc. of 13th Int'l. Conf. on Very Large Data Bases, pp. 507-518, 1987.
[21] Song, Z. and Roussopoulos, N., "Hashing Moving Objects," In Proc. of the 2nd Int'l. Conf. on Mobile Data Management, pp. 161-172, 2001.
[22] Tao, Y. and Papadias, D., "MV3R-Tree: a spatio-temporal access method for timestamp and interval queries," In Proc. of 27th Int'l. Conf. on Very Large Data Bases, pp. 431-440, 2001.
[23] Theodoridis, Y., Silva, J. R. O. and Nascimento, M. A., "On the Generation of Spatiotemporal Datasets," In Proc. of the 6th Int'l. Symp. on Spatial Databases, pp. 147-164, 1999.

권 동 섭
1998년 서울대학교 컴퓨터공학과 졸업(공학사), 2000년 서울대학교 대학원 컴퓨터공학과 졸업(공학석사), 2000년~현재 서울대학교 전기.컴퓨터공학부 박사과정. 관심분야는 시공간 데이타베이스, 이동 객체, 다차원 인덱스, XML 등

이 상 준
정보과학회논문지 : 데이타베이스
제 31 권 제 1 호 참조

이 석 호
정보과학회논문지 : 데이타베이스
제 31 권 제 1 호 참조