R-tree에서 Seeded 클러스터링을 이용한 다량 삽입

(Bulk Insertion Method for R-tree using Seeded Clustering)

이 태 원 † 문 봉 기 †† 이 석 호 †††
(Taewon Lee) (Bongki Moon) (Sukho Lee)

요 약 지구 관측 시스템(EOSDIS)나 많은 수의 클라이언트를 추적하는 이동전화 서비스 등 많은 응용에서는 지속적으로 생겨나는 대량의 복잡한 데이타들을 보관하고 인덱싱하는 것이 매우 어려운 일이다. 다차원 데이타를 효과적으로 관리하기 위해 R-tree에 기반한 인덱스 구조가 널리 사용되어 왔다. 본 논문에서는 빠른 데이타 생성 속도를 따라잡으면서 대량 삽입을 통해 R-tree를 관리할 수 있는 seeded clustering이라는 확장성있는 기법을 제안한다. 이 기법에서는 삽입할 대상 R-tree의 상위 k레벨의 구조를 활용하여 시드 트리를 만들어 삽입 데이타를 분류해 클러스터를 생성한다. 그리고 각 클러스터로부터 삽입 R-tree를 생성하고 이를 대상 R-tree에 한 번에 하나씩 삽입한다. 논문에서는 자세한 알고리즘과 함께 다양한 실험 결과를 보여준다. 실험 결과를 통해 seeded clustering을 이용한 대량 삽입이 기존의 대량 삽입 기법들과 비교해 삽입이나 질의 처리 모두에서 우수함을 알 수 있다.

키워드 : R-tree, 대량 삽입, seeded clustering, 대량 연산

Abstract In many scientific and commercial applications such as Earth Observation System (EOSDIS) and mobile phone services tracking a large number of clients, it is a daunting task to archive and index ever increasing volume of complex data that are continuously added to databases. To efficiently manage multidimensional data in scientific and data warehousing environments, R-tree based index structures have been widely used. In this paper, we propose a scalable technique called seeded clustering that allows us to maintain R-tree indexes by bulk insertion while keeping pace with high data arrival rates. Our approach uses a seed tree, which is copied from the top k levels of a target R-tree, to classify input data objects into clusters. We then build an R-tree for each of the clusters and insert the input R-trees into the target R-tree in bulk one at a time. We present detailed algorithms for the seeded clustering and bulk insertion as well as the results from our extensive experimental study. The experimental results show that the bulk insertion by seeded clustering outperforms the previously known methods in terms of insertion cost and the quality of target R-trees measured by their query performance.

Key words : R-tree, bulk insertion, seeded clustering, bulk operation

1. 서 론

지금까지 많은 응용 분야에서 다량의 공간 데이타를 효과적으로 다루기 위한 연구가 활발히 이루어졌다. R-tree는 공간 데이타를 관리하는 가장 널리 알려진 인덱스 구조이다. 요즈음과 같이 대량의 데이타가 지속적으로 생겨나는 환경에서는 새로 수집된 데이타를 기존에 존재하는 데이타베이스에 빠르게 추가하는 것이 매우 중요하다. 매번 전체 데이타로부터 새로이 인덱스를 생성하는 것은 매우 비효율적이며 확장성이 떨어진다. 이를 위해 이미 존재하는 R-tree 인덱스에 다량의 새로운 공간(다차원) 데이타를 빠르게 추가하는 대량 삽입(bulk insertion) 문제의 효율적인 해결이 요구되고 있다.

본 논문에서는 이 문제를 풀기 위해 seeded clustering을 이용한 대량 삽입 기법을 제안한다. 이 기법은 기존 대량 삽입 기법들에 비해 삽입 비용과 질의 처리

· 본 연구는 BK21의 지원을 받았음
† 학생회원 : 서울대학교 전기컴퓨터공학부
warrior@db.snu.ac.kr
†† 비 회 원 : 아리조나주립대학 컴퓨터학과 교수
bkmoon@cs.arizona.edu
††† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
shlee@cse.snu.ac.kr
논문접수 : 2003년 6월 14일
심사완료 : 2003년 9월 26일

성능 모두에서 우수하다. 기존 기법들은 모두 비슷한 방법을 따라 대량 삽입을 수행한다[1, 2, 3]. 먼저 새로 추가할 데이타 집합을 몇 개의 클러스터로 분류한 뒤 각 클러스터를 하나의 단위로 해서 한 번에 대상 R-tree에 대량으로 삽입하는 방법을 이용한다. 이 방법에서 하나의 클러스터는 공간상에서 가까운 데이타들의 집합으로 이루어진다. 각각의 클러스터가 공간 상에서 작은 영역을 차지하도록 하여 대상 R-tree에 삽입이 되었을 때 삽입이 된 노드의 MBR 영역의 확장을 줄여보려는 시도이다. 그러나 이것이 삽입된 다른 노드들과 겹치는 영역이 늘어나지 않게 한다고 보장할 수는 없다. 새로 추가할 데이타들만을 이용해 클러스터링한 경우 이미 존재하는 R-tree의 구조를 전혀 고려하지 않기 때문에 클러스터가 삽입된 노드의 MBR이 확장할 가능성이 높으며 결과적으로 다른 노드들과 겹치는 영역이 증가하여 질의 성능 저하로 이어진다.

본 연구에서는 이 점에 착안하여 seeded clustering을 통해 대상 R-tree의 구조를 활용하여 새로 추가되는 데이타들을 클러스터링한다. 이 방법에서는 대상 R-tree의 상위 k레벨의 MBR 정보를 이용하여 입력 데이타를 효과적으로 분류한다. 클러스터링이 끝나면 각 클러스터로부터 작은 입력 R-tree를 구성하고 각 입력 R-tree들을 대상 R-tree에 한 번에 하나씩 삽입한다. 여기서 입력 R-tree를 대상 R-tree에 삽입하는 것은 대상 R-tree가 삽입이 완료된 후에도 R-tree로서의 특징을 만족해야 하기 때문에 단순한 작업이 아니다. 일부 기존 연구들에서는 이것이 보장되지 않는다.

대량 삽입을 함에 있어서 삽입 속도가 데이타의 생성 속도를 충분히 따라잡을 수 있어야 하는 것은 물론이고 그 결과로 생성되는 대상 R-tree가 질의를 효과적으로 수행할 수 있어야 한다. Seeded clustering을 통한 대량 삽입 기법은 대량 삽입 도중에 계속해서 노드 간에 겹치는 영역을 줄이려는 시도를 하기 때문에 질의 성능이 향상됨을 실험을 통해 볼 수 있다. 성능 평가를 위해서 본 논문에서는 실제 데이타와 다양한 가상 데이타를 이용해 실험을 수행하였다. 실제 데이타로는 지리정보시스템(GIS)에서 가장 널리 사용되는 TIGER/Line 데이타 집합을 사용하였고, 가상 데이타로는 TPC/H에서 사용되는 데이타와 다양한 분포를 가진 집합들을 이용하였다.

본 논문은 다음과 같이 구성된다. 2장에서는 관련 연구에 대해 간략히 살펴본다. 3장에서는 시드 트리(seed tree)의 구조에 대해 살펴보고 4장에서는 자세한 대량 삽입 알고리즘을 제시한다. 그리고 5장에서 성능 평가 결과를 제시하며 마지막으로 6장에서 결론을 맺는다.

2. 관련 연구

R-tree는 공간 데이타나 다차원 데이타를 효율적으로 검색하기 위해 만들어진 다차원 인덱스 구조이다. 트리 구조로 이루어지며 모든 리프 노드가 같은 레벨에 위치하는 특징을 가지고 있다. 한 노드는 (ptr, mbr) 형태의 엔트리를 포함하게 되는데, 여기서 ptr은 자식 노드를 가리키는 포인터이고 mbr은 자식 노드에 존재하는 모든 엔트리를 포함하는 최소의 사각형(minimum bounding *rectangle*)으로 정의한다[4]. 한 노드는 디스크 기반의 R-tree에서 하나의 디스크 페이지로 표현된다. R-tree의 하나의 노드는 최대 M개의 엔트리를 포함할 수 있고 루트 노드를 제외한 모든 노드는 최소 $m(2 \le m \le M/2)$개의 엔트리를 포함해야 하는 특징이 있다. 이렇게 구성된 R-tree에 대해 공간상의 점(point query)이나 영역(range query)을 질의로 입력하게 되면 해당 점을 포함하거나 해당 질의 영역과 겹치는 모든 데이타를 검색해 결과로 돌려주게 된다. R-tree에서 노드의 MBR 간에는 서로 겹치는 영역이 존재할 수 있고, 이 영역의 넓이에 따라 질의 수행 능력에 차이를 보이게 된다. 겹치는 영역이 넓으면 질의 수행 중 리프 노드까지 따라가야 하는 경로가 많아져 그만큼 질의 수행 능력이 저하된다. 그러므로 노드 간에 겹치는 영역이 적을수록 좋은 구조를 가진 R-tree라 할 수 있다.

R-tree에서의 대량 삽입에 관한 초기 연구에서는 삽입할 데이타를 먼저 공간 상에서의 근접성에 의해 정렬한 후(예: Hilbert 값에 의해) B개씩 묶어서 블럭을 구성한다[3]. 그런 후에 이 블럭들을 한 번에 하나씩 변형된 표준 삽입 알고리즘을 이용해 삽입한다. 직관적으로 이 기법은 한 번에 B개의 데이타가 삽입되므로 B배의 삽입 속도 향상을 가져온다는 것을 알 수 있다. 그러나 이 블럭과 기존 R-tree의 노드 사이의 겹치는 면적은 증가할 가능성이 매우 높고 결과적으로 질의 성능 저하를 가져올 수 있다. 이는 대상 R-tree의 입장에서 봤을 때 삽입될 각 블럭은 랜덤하게 생성된 것과 같기 때문에 블럭이 작은 영역을 차지하더라도 블럭이 삽입되는 노드가 확장될 가능성이 높은 것이다. 이는 실험 결과를 통해서도 확인할 수 있다[3].

또 다른 대량 삽입에 대한 연구로 STLT(small-tree-large-tree) 기법을 이용한 연구가 있다[1]. STLT에서는 입력 데이타로부터 하나의 입력 R-tree(small tree)를 만들고 이것을 대상 R-tree (large tree)에 삽입한다. 이 방법은 입력 R-tree가 넓은 영역을 포함하는 경우 이것이 삽입되는 대상 R-tree의 노드의 MBR이 매우 커져서 노드간 겹침이 많이 발생하게 된다. 따라서 데이타가 좁은 영역에 치우쳐 있는 경우에만 효과적으로 동작한다[2]. 그리고 입력 R-tree의 높이가 반드시 대상 R-tree보다 낮아야 한다는 제약이 있다.

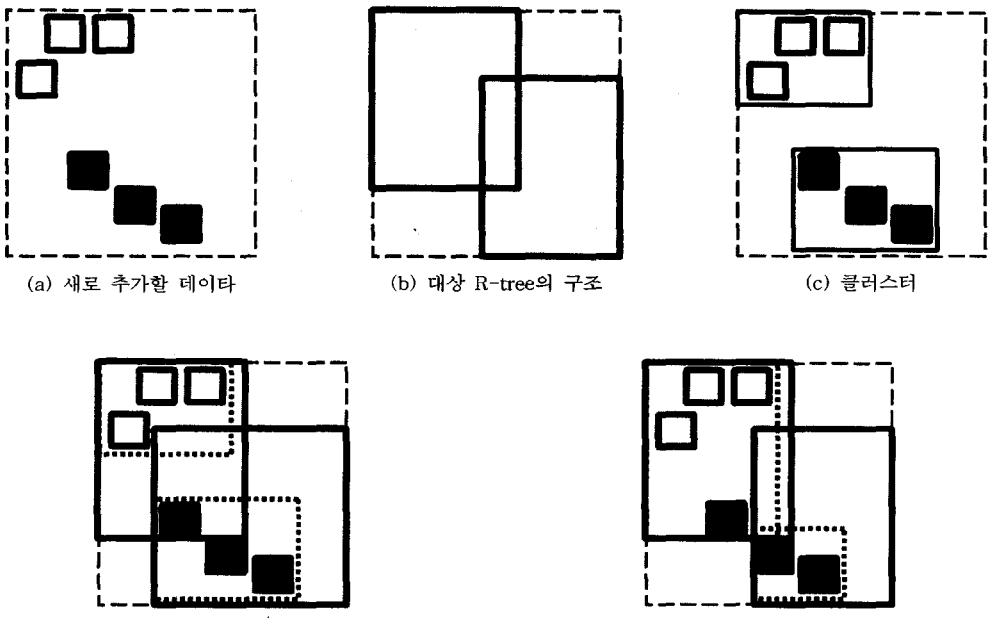
STLT에서 발전된 것으로 GBI(generalized bulk insertion) 기법이 있다[2]. 여기서는 입력 데이타 집합을 공간 상에서 근접한 데이타끼리 분류하여 여러 개의 클러스터를 생성한다. 각 클러스터로부터 R-tree를 생성하고 마지막으로 이 R-tree들을 대상 트리에 한 번에 하나씩 삽입한다. 어떤 클러스터에도 포함되지 않는 데이타들은 열외자(outlier)로 분류하여 일반적인 R-tree 삽입 알고리즘에 의해 하나씩 삽입한다. 이 기법은 STLT의 문제점들을 많이 보완했으나 역시 블럭 단위로 삽입하는 경우에 발생하는 문제와 같이 대상 R-tree 트리의 노드들과 새로 삽입되는 R-tree들이 많이 겹칠 수 있는 특징이 있다. 또한 STLT나 GBI는 최종적으로 생성되는 R-tree가 올바른 R-tree가 아닐 수 있다는 문제가 존재한다. R-tree의 정의에 따르면 루트 노드는 m(한 노드가 포함할 수 있는 최소의 엔트리 개수)보다 적은 수의 엔트리를 가질 수 있지만, 입력 R-tree를 대상 R-tree에 삽입한 후에는 입력 R-tree의 루트 노드는 대상 R-tree의 중간 노드가 되어 버리므로 m보다 적은 엔트리를 포함하는 중간 노드가 생기게 된다. 이는 R-tree의 기본적인 조건에 맞지 않으므로 본 논문에서는 5장에서 이 문제를 해결하는 방법을 제시한다.

이와는 다르게 버퍼를 이용한 접근 방법을 이용한 연구가 있다[5]. 이는 메인 메모리의 여유 공간과 운영체제의 디스크 페이지 크기를 이용하는 버퍼 트리 아이디어를 사용한 기법이다. 이 방법은 일반적인 삽입 알고리즘에 비해서 향상된 삽입 성능을 보이지만 개념적으로 하나씩 반복해서 삽입하는 알고리즘과 동일하기 때문에 최종 생성되는 트리는 일반적인 삽입으로 생성되는 트리보다 향상된 질의 처리 성능을 보이지는 못한다.

3. Seeded Clustering

기존 연구에서 대량 삽입을 수행하는 일반적인 방법은 다음과 같다[1-3]. 먼저 입력 데이타 집합을 클러스터링 기법 등을 통해 분할한다. 각 분할로부터 하나씩의 입력 R-tree가 생성되고 이 입력 R-tree를 변형된 삽입 알고리즘에 의해 대상 R-tree에 삽입한다. 입력 R-tree 내의 데이타가 한꺼번에 삽입되므로 대량 삽입이 가능해진다.

입력 데이타가 그림 1(a)와 같고 대상 R-tree의 구조가 그림 1(b)와 같이 주어졌다고 하자. 대부분의 클러스터링 기법을 적용하면 그림 1(c)와 같이 데이타를 분류하게 된다. 각 클러스터로부터 입력 R-tree를 생성한 후에 대상 R-tree에 삽입을 하면 대상 R-tree를 고려하지 않고 클러스터링을 했기 때문에 그림 1(d)에서와 같이 대상 R-tree 노드의 영역이 기존보다 확장될 가능성이 크다. 그러나 그림 1(e)에서처럼 대상 R-tree의 구조



(a) 새로 추가할 데이타 (b) 대상 R-tree의 구조 (c) 클러스터

(d) 삽입 후 대상 R-tree의 구조(MBR이 증가했음) (e) 대상 R-tree의 구조를 고려해 삽입했을 때의 이상적인 모습

그림 1 시드 트리의 제안 동기

를 고려하여 클러스터링을 수행을 하는 경우, 입력 데이타 중 하단 세 개가 같은 노드로 들어가지 않는 것이 바람직함을 미리 알 수 있고 따라서 더 좋은 결과를 가져올 수 있다.

시드 트리는 대상 R-tree의 상위 k레벨을 복제하여 생성되며, 입력 데이타를 분류하기 위해 사용된다. 여기서 k는 입력 데이타의 수를 고려하여 결정한다. 여기서는 입력 데이타로부터 대량 로딩(bulk loading)을 통해 하나의 R-tree를 생성한다고 했을 때 이것이 삽입될 수 있는 레벨까지를 복제한다.

시드 트리는 데이타를 단지 분류하는 목적으로만 사용한다는 점에서 초기 구조는 R-tree와 동일하지만 R-tree는 아니다. 시드 트리를 통해 데이타를 분류하는 과정은 다음과 같다. 먼저 입력 데이타의 MBR이 루트 노드의 MBR에 완전히 포함되면 루트 노드의 엔트리들 중 입력 데이타의 MBR을 완전히 포함하는 엔트리를 선택한다. 다시 이 엔트리가 가리키는 하위 노드를 기준으로 동일한 작업을 반복적으로 수행한다. 만일 이 노드가 시드 트리의 리프 레벨 노드인 경우에는 더 이상 진행하지 않고 해당하는 클러스터에 입력 데이타를 추가한다. 이렇게 함으로서 시드 트리는 입력 데이타가 정상적으로 R-tree에 삽입이 되었을 경우에 선택이 될 노드로 데이타를 유도하게 된다.

입력 데이타의 분류 과정에서 시드 트리의 엔트리 중에 입력 데이타의 MBR을 완전히 포함하는 엔트리가 하나 이상일 수 있다. 이 중 하나를 선택하는 가장 단순한 방법으로는 엔트리들 중 첫번째로 조건을 만족하는 것을 선택하는 것으로 이는 가장 빠른 속도를 제공한다. 이외에도 해당 엔트리들 중 가장 면적이 작은 것을 선택하는 방법이 있다. 이 경우 클러스터가 차지하는 영역이 작아지는 효과가 있다. 또 다른 방법으로는 입력 데이타의 중심과 엔트리의 중심이 가장 가까운 그런 엔트리를 선택할 수 있다. 이는 입력 데이타와 삽입이 될 엔트리와의 관계에 초점을 맞춘 것이다. 이처럼 여러 가지의 엔트리를 선택하는 방법이 있으나 여기서는 구현을 간단하게 하기 위해 조건을 만족하는 첫번째 엔트리를 선택하도록 한다.

분류 과정 중 어떤 중간 노드에서 입력 데이타의 MBR을 완전히 포함하는 엔트리를 발견하지 못할 경우 이 데이타는 열외자(outlier)로 분류되어 추후에 일반적인 삽입 방식을 이용해 한 번에 하나씩 삽입된다. 성능 평가에서 사용한 모든 종류의 데이타 집합에 대해 열외자의 비율은 전체 삽입 데이타의 0.1% 이내이므로 전체적인 성능에 크게 영향을 끼치지 않음을 알 수 있다.

다음 절에서는 알고리즘의 삽입 단계에 대해 자세하게 설명한다. 이 단계에서는 각 클러스터로부터 입력 R-tree가 생성되고 이 R-tree들을 대상 R-tree에 하나씩 삽입을 한다.

4. 대량 삽입

이 절에서는 seeded clustering을 이용한 대량 삽입에서의 삽입 단계에 대해 설명한다. 설명을 위해 R-tree의 리프 레벨은 레벨 0이라 정의한다. 따라서 R-tree의 높이는 1+(루트 노드의 레벨)이 된다.

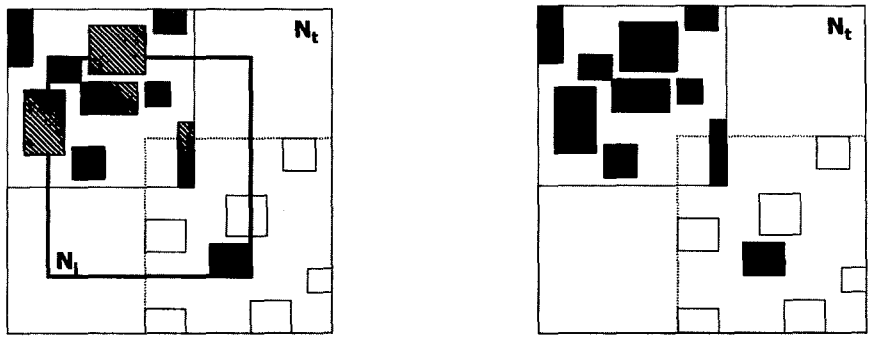
4.1 입력 R-tree를 대상 R-tree에 삽입

3장에서 기술했듯이 분류 작업을 마치면 여러 개의 클러스터와 열외자들을 얻을 수 있다. 대량 삽입을 위해서는 먼저 각각의 클러스터로부터 입력 R-tree를 생성한 후에 이를 대상 R-tree에 삽입을 한다. 입력 R-tree는 빠르고 효율적인 구조를 갖도록 하기 위해 대량 로딩 기법을 이용하여 생성한다[6,7]. 입력 R-tree를 대상 R-tree에 삽입하는 작업은 데이타 아이템을 삽입하는 과정과 비슷하지만 후처리 작업이 필요한 점 등 몇 가지 차이점이 있다. 입력 R-tree를 삽입하기 위해서는 입력 R-tree를 MBR이 루트 노드의 MBR인 하나의 데이타 아이템으로 간주를 하고 삽입한다. 단, 모든 리프 노드는 같은 레벨에 존재해야만 하는 R-tree의 특징을 만족시키기 위해서 이 데이타 아이템은 대상 R-tree의 리프 노드에 삽입하면 안 된다.

예를 들어 입력 R-tree의 높이가 h_i라 가정해 보자. 대상 R-tree가 입력 R-tree를 삽입한 후에도 올바른 R-tree이기 위해서는 입력 R-tree의 루트 노드인 N_i가 대상 R-tree의 레벨 h_i에 있는 노드에 삽입이 되어야 한다. 시드 트리의 정의에 의해 이 입력 R-tree가 삽입될 대상 R-tree의 노드는 이미 알고 있다.

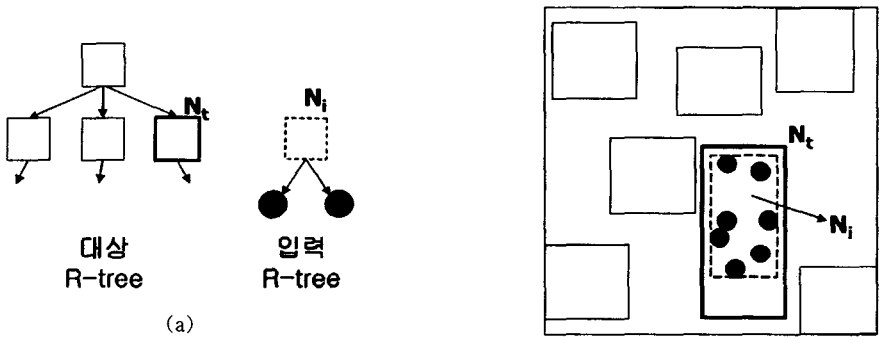
삽입 후에 대상 R-tree가 올바른 R-tree이기 위해 지켜져야 할 또다른 중요한 성질이 있는데 이는 루트 노드를 제외한 모든 노드는 최소한 m개의 엔트리를 포함해야 한다는 것이다[4]. 노드 언더플로우(node underflow)는 m보다 적은 수의 엔트리를 갖는 노드의 상태를 나타내는 용어이다[8]. 입력 R-tree의 루트 노드인 N_i가 노드 언더플로우 상태라 가정하자. R-tree의 정의에 의해 루트 노드는 언더플로우 상태일 수 있으므로 이 입력 R-tree는 올바른 R-tree이다. 그러나 이것을 대상 R-tree에 삽입하게 되면 입력 R-tree의 루트 노드 N_i는 대상 R-tree의 중간 노드가 되므로 대상 R-tree는 올바르지 않은 R-tree가 된다.

우리는 이 문제를 N_i의 모든 엔트리를 대상 R-tree의 대상 노드인 N_t의 엔트리 중 겹치는 엔트리들에 분산시켜 해결한다. 즉, N_i의 엔트리를 N_t에 재삽입해 문제를 해결한다. 그림 2는 노드 언더플로우의 한 예와



(a) (b)

그림 2 노드 언더플로우 예(m=5). (a) N_i의 엔트리 수가 m보다 적다. (b) N_i의 엔트리들을 삽입될 노드인 N_t의 엔트리에 분산시킨 후의 결과



(a) (b)

그림 3 (a) 입력 데이타가 한 공간에 편중되어 입력 R-tree의 루트 노드인 N_i가 N_t와 같은 레벨이 된 경우 (b) 시드 트리의 정의에 의해 N_t가 N_i를 완전히 포함한다.

그 처리 결과를 나타낸다.

만일 N_i가 노드 언더플로우가 발생하지 않는 경우에는 입력 R-tree를 N_t에 바로 삽입한다. 그러나 이 경우엔 다음과 같은 복잡한 문제가 발생할 가능성이 있다. 먼저 N_t의 레벨이 l_t라고 가정하자. 입력 R-tree가 노드 N_t에 삽입되기 위해서는 높이가 l_t여야 한다. 그러나 입력 데이타가 한 곳에 편중되어 분포하는 경우, 한 클러스터 내의 데이타가 너무 많아져서 그로부터 생성된 입력 R-tree의 높이가 l_t보다 높아질 수 있다. 이 경우 입력 R-tree는 N_t에 맞지 않는다. 또 이와는 반대로, 클러스터 내의 데이타가 너무 적어서 입력 R-tree의 높이가 l_t보다 낮을 수도 있다. 이와 같이 입력 R-tree의 높이가 삽입될 노드에 맞지 않는 경우에는 각각 다음과 같이 해결을 한다.

4.1.1 입력 R-tree가 N_t보다 상위 레벨에 삽입되어야 하는 경우

N_i는 대상 R-tree의 레벨 h_i에 위치한 노드에 삽입이 되어야 한다. 이 경우는 $h_i > l_t$이므로 N_i는 N_t의 부모나 선조 노드에 삽입이 되어야만 한다. Seeded clustering의 정의에 의해 입력 R-tree는 그림 3(b)에서처럼 N_t에 완전히 포함되게 된다. 만일 $l_t = h_i - 1$이라 가정을 하면, N_i는 N_t의 부모 노드에 삽입이 되어야 하므로 결국 N_i와 N_t가 같은 부모 노드를 가지게 된다. 그런데 N_i가 공간상으로 N_t에 완전히 포함되어 있기 때문에 겹치는 영역을 줄이기 위해 4.2에서 설명할 후처리 작업인 재포장 기법을 통해 N_i와 N_t의 엔트리들을 포함하는 새로운 MBR을 구성하여 삽입을 수행할 수 있다. $l_t < h_i - 1$인 경우에도 마찬가지로 N_i가 N_t에 완전히 포함되기 때문에 N_i가 삽입될 N_t의 적절한 레벨의 선조 노드를 찾아 후처리 작업을 수행함으로서 삽

입을 처리할 수 있다.

4.1.2 입력 R-tree가 N_t보다 하위 레벨에 삽입되어야 하는 경우

이 경우는 앞 절에서보다 더 복잡한 문제가 발생한다. N_i는 입력 데이타 중에서 N_t에 완전히 포함되는 데이타만으로 만들어진 입력 R-tree의 루트 노드이기 때문에 N_t의 많은 영역을 차지할 가능성이 크다. 그런데 그림 4 (a)에서처럼 입력 R-tree가 높이가 낮아 N_t에 바로 삽입이 될 수 없는 상황이기 때문에 N_i는 N_t의 엔트리들 중 적합한 엔트리에 삽입이 되어야 한다. 이 때 N_i를 포함하게 되는 엔트리는 MBR이 매우 커져서 N_t의 기존의 다른 엔트리들과 겹치는 영역이 커져 결과적으로 질의 성능을 저하시키게 된다.

이를 해결하기 위해서는 N_i의 모든 엔트리를 N_t에 재삽입하는 방법이 있다. 이 경우 N_t의 엔트리가 크게 확장되지 않아 질의 성능 저하를 줄일 수 있다. 이 방법은 루트 노드로부터 N_t에 이르는 경로에 대해서만 한 레벨 더 확장한 시드 트리를 이용해 클러스터링을 수행하는 것과 유사하다. 본 논문에서는 이 문제를 해결하기 위해 시드 트리를 동적으로 확장하는 기법을 제안한다.

N_t는 정의에 의해 시드 트리의 리프 레벨 노드이다. 시드 트리의 높이를 어떻게 결정해야 적절한지 미리 예측하기 어려우므로 3장에서 설명했던 방법을 통해 적절한 레벨의 시드 트리를 구성해 클러스터링을 수행한다. 그런 후에 각 클러스터 내의 데이타 수를 이용, 필요한 경우 해당 경로에 대해서만 시드 트리를 한 단계 더 확장해 클러스터링을 수행한다. 입력 R-tree는 대량 로딩 기법을 통해 생성되기 때문에 입력 R-tree를 생성하지 않고도 클러스터 내의 데이타 수를 토대로 높이를 예측하는 것이 가능하다. 따라서 1차 클러스터링 후에 입력 R-tree를 생성하는 비용을 들이지 않고 확장해야 하는 경로를 확인할 수 있고, 해당 경로에 대해서는 시드 트리를 확장한 후에 해당 클러스터의 입력 데이타만을 대상으로 클러스터링을 다시 수행한다. 이 때는 다시 시드 트리의 루트 노드로부터 입력 데이타를 분류하는 것이 아니라 해당 리프 노드로부터 확장된 경로를 따라 분류를 수행할 수 있다. 동적으로 확장된 시드 트리의 예가 그림 5에 제시되어 있다.

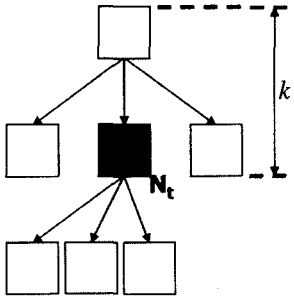
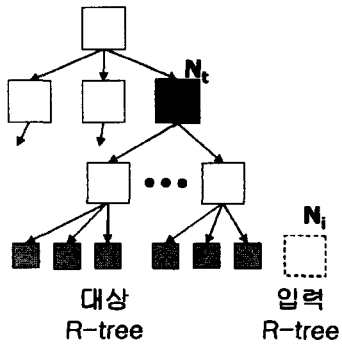


그림 5 동적으로 확장된 시드 트리

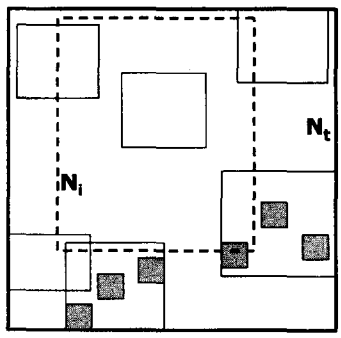
4.2 재포장 기법

입력 R-tree를 대상 R-tree의 노드 N_t에 삽입할 때, N_i와 N_t의 엔트리들 사이에 겹치는 영역의 넓이가 클 수 있다. 효율적인 질의 처리를 위해서는 삽입 과정에서 노드들간의 겹치는 영역을 줄일 필요가 있다. 여기서 제안하는 재포장 기법은 지역적으로 겹치는 영역을 줄여주는 대량 삽입의 후처리 과정이다.

기본적인 아이디어는 다음과 같다. N_i와 많이 겹치는 N_t의 엔트리들이 가리키는 하위 노드의 엔트리들을 추



(a)



(b)

그림 4 (a) N_t에 클러스터된 입력 데이타가 너무 적어 입력 R-tree가 N_t의 직속 엔트리가 될 수 없는 경우
(b) N_i는 N_t의 자식 노드의 엔트리와 같은 레벨

출해 해당 엔트리들을 포함하는 최적의 MBR을 다시 구성하는 것이다. 이렇게 함으로써 겹치는 영역을 감소시킬 수가 있다.

알고리즘 1 후처리 : 대량 삽입 과정 중 노드간의 겹침을 줄이는 작업

```
Function post-process(N_t, N_i)
N_t : N_i가 삽입될 노드
N_i : 입력 R-tree의 루트 노드
begin
        NoOvlp ← N_t와 겹치지 않는 N_t의 엔트리들
        Ovlp ← N_t와 겹치는 N_t의 엔트리들
        RepackedNodes ← REPACK(Ovlp, N_i)
        return pack(NoOvlp ∪ RepackedNodes)
end
```

자세한 알고리즘은 알고리즘 1~3에 제시되어 있고 여기서는 간단하게 알고리즘에 대해 설명을 한다. N_t의 엔트리들은 2개의 그룹으로 나눌 수 있는데, 하나는 N_i와 겹치는 엔트리 그룹이고, 또 하나는 겹치지 않는 엔트리 그룹이다. 이 중 겹치는 엔트리 그룹과 N_i를 다시 포장해 새로운 노드(들)을 구성한다. 이 때 재포장을 하기 전에 먼저 하위 레벨에서 재포장을 완료해야만 한다. 이는 N_t의 엔트리들이 가리키는 하위 노드들 각각과 재구성할 N_t의 엔트리들이 가리키는 하위 노드들 사이에서 동일한 과정을 통해 겹쳐지는 것들끼리 재포장을 함으로써 가능하다. 이것은 알고리즘 2에 있는 것처럼 재귀적으로 정의할 수 있다.

알고리즘 2 N_{input}와 N_{set}의 노드간의 재포장

```
Function REPACK(N_set, N_input)
N_set : N_input와 겹치는 부모 노드를 갖는 노드들의 집합
N_input : 입력 R-tree의 한 노드
begin
   ent ← N_set의 엔트리가 가리키는 하위 노드의 엔트리들
   NoOvlp ← N_input의 엔트리와 겹치지 않는 ent의 부분집합
   Repacked ← Φ
   if N_input이 리프 노드 then
      return pack(ent∪ N_input의 엔트리)
   else
      foreach e ∈ N_input의 엔트리 do
         entOvlp ← e와 겹치는 ent의 원소들
         if entOvlp ≠ Φ then
            Repacked ← Repacked ∪ REPACK(entOvlp, e)
         else
            Repacked ← Repacked ∪ e
         end
      end
   end
   return pack(NoOvlp ∪ Repacked)
end
```

알고리즘 3 노드들을 포장해 새로운 노드들을 구성

```
Function pack(N_set)
N_set : 포장할 노드들의 집합
begin
   return N_set의 원소들을 포함하는 대량 로딩을 통해
   생성된 노드들
end
```

5. 실 험

이 절에서는 본 논문에서 제시한 접근방법이 타당하고 효과적이라는 것을 증명하기 위해 다양한 실험을 수행하였다. 실험을 위해 리눅스 기반에서 C++를 이용해 디스크 기반의 R-tree를 작성하였다. 한 노드는 4KB의 디스크 블록에 해당하며 100여개의 엔트리를 포함할 수 있다. 성능 비교는 한 번에 하나의 데이타를 삽입하는 R-tree의 기본적인 삽입 방식(OBO; One by one)과 GBI 기법에 대해 수행하였다. 삽입 비용과 질의 처리 비용은 평균적인 디스크 입출력 회수로 측정하였다. 본 논문에서 제시한 seeded clustering을 이용한 대량 삽입 기법은 간략히 SCB라 표시하겠다.

실험에 사용할 데이타로는 실데이타 집합과 서로 다른 특성을 지닌 여러 개의 합성 데이타 집합을 이용하였다. 실데이타로는 공간 데이타베이스 분야에서 널리 사용되는 표준 벤치마크 데이타인 TIGER/Line 데이타 중 2,600,934개의 개체 정보를 추출하여 사용하였다[9]. 합성 데이타로는 의사결정 지원 시스템 벤치마킹에서 사용하는 TPC-H 데이타와 세 개의 서로 다른 분포를 가지는 데이타 집합을 이용하였다[10]. 세 가지 분포는 각각 균등분포, 편중분포(zipfian) 그리고 군집분포(clustered)이다. 각 데이타 집합은 1,000,000개의 데이타를 포함한다. TPC-H 데이타는 합성 데이타 중에서도 상업 데이타베이스의 성능을 비교하기 위해 실제로 사용되는 데이타이므로 실험에서 사용하는 데이타가 조작되지 않고 공정성을 가질 수 있어서 실험에서 이용을 하였다.

5.1 서로 다른 데이타 집합에 대한 삽입 비용 및 질의 처리 비용

표 1 데이타 집합별 삽입 비용(디스크 I/O 수)

	SCB	GBI	OBO
TIGER/Line	36,263	80,757	151,689
TPC-H	18,752	53,021	108,018
균등분포	16,443	49,855	103,809
편중분포	13,247	43,724	98,096
군집분포	11,607	25,351	56,945

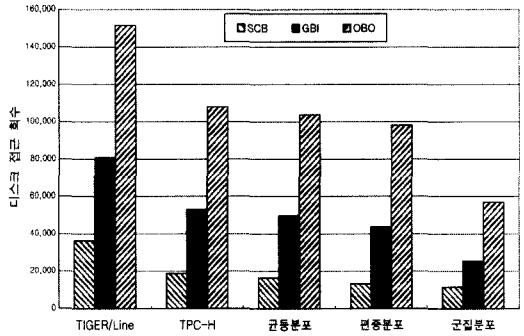


그림 6 데이타 집합별 삽입 비용(디스크 I/O 수)

여기서는 위에서 설명한 각 데이타 집합에 대해 SCB, OBO, GBI 기법의 삽입 및 질의 처리 비용을 실험하였다. 입력 데이타 집합은 대상 R-tree의 5% 크기로 하였고 시드 트리는 대상 R-tree의 상위 2개 레벨을 이용해 구성하였다. 대상 R-tree는 모든 데이타 집합에 대해 높이가 4였다. 대상 R-tree는 한 번에 하나씩 삽입하는 OBO 방법에 의해 생성되었다. 질의 처리 비용은 단위 사각형 내에서 5,000번의 임의 포인트 질의를 수행한 평균 디스크 접근 회수를 측정하였다. 여기에는 실제 데이타를 추출하는 비용이 아닌 R-tree 노드의 접근 비용을 비교하였다. 삽입 비용은 표 1/그림 6에, 질의 처리 비용은 표 2/그림 7에 제시되어 있다. 표에서 SCB와 GBI의 삽입 비용에는 clustering에 필요한 디스크 I/O 수가 포함된 결과이다.

결과에서 볼 수 있듯이 본 논문에서 제안하는 SCB가 GBI나 OBO에 비해 삽입이나 질의 처리 모두에서 크게 앞서는 것을 알 수 있다. GBI의 경우 클러스터링에 드는 비용이 커서 삽입 비용도 뒤떨어지는 현상을 보이고 있음을 알 수 있다. 삽입 비용 면에서는 압도적인 우위를 보이고 있지는 않지만 여전히 향상된 성능을 보이고 있음을 알 수 있다.

삽입을 빠르게 수행하는 것도 중요하지만 인덱스는 결국 질의 처리를 효과적으로 하기 위한 것이므로 질의 처리 비용에서도 성능을 크게 향상시킨다는 점이 본 연구가 크게 기여한 점이다. 본 연구가 목표로 하는 환경은 인덱스를 재구성하기 어려운 응용 환경이므로 대량

표 2 각 데이타 집합별 질의 처리 비용(디스크 I/O 수). 포인트 질의

	초기상태	SCB	GBI	OBO
TIGER/Line	2.665	2.496	3.034	3.215
TPC-H	2.111	1.547	2.234	2.561
균등분포	5.446	4.313	5.678	5.941
편중분포	4.442	3.733	4.645	5.018
군집분포	4.234	3.471	4.523	4.833

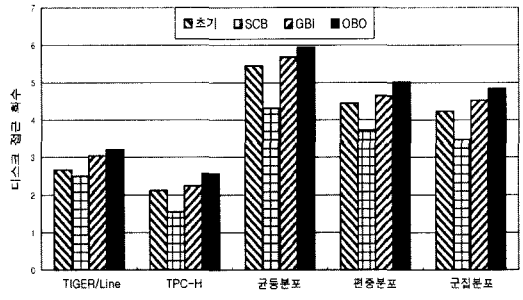


그림 7 데이타 집합별 질의 처리 비용(디스크 I/O 수). 포인트 질의

삽입 이후에 질의 성능이 유지되거나 향상될 수 있다는 점은 매우 중요하다. 지금까지는 OBO가 질의 처리 면에서 가장 좋은 성능을 보여왔고, 기존 대량 삽입에 관련된 연구들은 OBO보다 향상된 결과를 보이지 못했다[1-3]. 그러나 SCB는 대량 삽입 후에 오히려 OBO보다 향상된 질의 처리 성능을 보임을 알 수 있다. 이는 SCB가 질의 성능에 가장 큰 영향을 끼치는 노드 간의 겹치는 영역을 삽입 과정에서 노드의 재구성을 통해 크게 줄이기 때문이다.

5.2 반복되는 대량 삽입에 따른 성능

여기서는 대상 R-tree가 SCB기법을 이용한 반복적인 대량 삽입 이후에도 충분히 효과적인 질의 처리 성능을 보이는지를 검증하기 위한 실험을 수행하였다. 이는 인덱스를 재구성하는 비용이 너무 큰 환경에서는 매우 중요한 문제라고 할 수 있다. 실험을 위해 초기에 1,000,000개의 데이타를 갖는 R-tree를 일반적인 삽입 기법을 통해 구성한 후 100,000개씩의 데이타를 총 6번 다량 삽입하면서 매 삽입 후의 질의 처리 비용을 측정하였다.

실험 결과는 그림 8에 제시되어 있다. 결과를 살펴보면 예상했듯이 SCB의 경우 계속해서 비슷한 질의 수행 시간을 유지함을 알 수 있다. OBO의 경우는 대상 R-tree내의 데이타 수가 계속해서 증가하게 되면 질의 수행 시간이 조금씩 증가하는 반면에 SCB의 경우에는 대상 R-tree를 지속적으로 재구성하여 질의 성능에 중

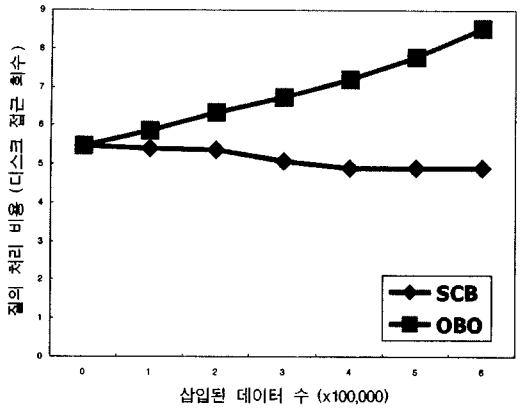


그림 8 반복적인 대량 삽입 시의 질의 처리 비용 비교

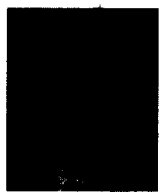
요한 요소인 노드 간의 겹치는 영역을 감소시켜 주므로 데이타 수가 증가함에도 불구하고 질의 성능은 오히려 좋아지는 결과를 보이고 있음을 알 수 있다.

6. 결 론

본 논문에서 우리는 데이타가 지속적으로 대량으로 추가되는 환경에서 효율적으로 대량 삽입을 통해 인덱스를 유지할 수 있는 기법을 제안하였다. 입력 데이타를 분류하는 과정에서 대상 R-tree의 구조를 이용해 빠르고 효과적으로 입력 데이타를 분류하는 seeded clustering 기법을 제시하였다. Seeded clustering을 통해 입력 데이타가 여러 개의 클러스터로 분류되고 각 클러스터로부터 입력 R-tree가 생성되며 이 입력 R-tree를 한 번에 하나씩 대상 R-tree에 삽입하여 대량 삽입이 이루어진다. 이 과정에서 질의 성능을 향상시키기 위해 지역적으로 노드 간의 겹치는 영역을 줄여주는 재포장 기법을 제안하였다. 이런 특징으로 인해 본 논문에서 제시한 알고리즘은 확장성을 가진다. 또한 데이타가 지속적으로 추가됨에 따라 지속적으로 잘 구성된 트리 구조를 유지해 기존 대량 삽입 기법에 비해 향상된 질의 처리 성능을 보인다.

참 고 문 헌

[1] L. Chen, R. Choubey and E. A. Rundensteiner, "Bulk-insertions into R-trees using the small- tree-large-tree approach," ACM GIS, pp. 161~162, 1998.
[2] R. Choubey, L. Chen and E. A. Rundersteiner, "GBI: A Generalized R-tree Bulk-Insertion Strategy," Advances in Spatial Databases, pp. 91~108, 1997.
[3] I. Kamel, M. Khalil and V. Kouramajian, "Bulk insertion in dynamic R-trees," SDH '96, pp. 3B.3 1~3B.42, 1996.
[4] A. Guttman, "R-trees: a dynamic index structure for spatial searching," ACM SIGMOD, pp. 47~57, 1984.
[5] L. Arge, K. H. Hinrichs, J. Vahrenhold and J. S. Vitter, "Efficient Bulk Operations on Dynamic R- Trees," Algorithmica, Vol. 33, No. 1, pp. 104~128, 2002.
[6] I. Kamel and Christos Faloutsos, "On packing R-trees," CIKM, pp. 490~499, 1993.
[7] S. T. Leutenegger, J. M. Edgington and M. A. Lopez. "STR: A Simple and Efficient Algorithm for R-Tree Packing," ICDE, pp. 497~506, 1997.
[9] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," ACM SIGMOD, pp. 322~331, 1990.
[9] TIGER/Line Files, 2000 Technical Documentation, U.S. Bureau of Census, Washington DC, accessible via URL http://www.census.gov/geo/www/tiger/tigerua/ua_tgr2k.html
[10] TPC-H, Transaction Processing Performance Council, accessible via URL, http://www.tpc.org/tpch/



이 태 원

1997년 서울대학교 컴퓨터공학과 졸업(학사). 1999년 서울대학교 전기컴퓨터공학부 졸업(공학석사). 1999년~현재 서울대학교 전기컴퓨터공학부 박사과정. 관심분야는 다차원 인덱스, XML 등임

문 봉 기

1996년 미국 매릴랜드 대학 졸업(박사) 현 University of Arizona, Computer Science 부교수. ACM SIGMOD (2003), EDBT(2002), ISDB(2002), WWW(2002), VLDB (2001), ACM SIGMOD(20001) 위원회 위원 역임, 2002년 ACM SIGMOD Proceedings Chair 역임. 관심분야는 XML indexing, data mining, data warehousing과 parallel & distributed processing

이 석 호

1964년 연세대학교 정치외교학과 졸업 1975년, 1979년 미국 텍사스대학교 전산학 석사와 박사학위 취득. 1979년~1982년 한국과학원 전산학과 조교수. 1982년~1986년 한국정보과학회 논문 편집위원장. 1986년~1988년 한국정보과학회 부회장. 1988년~1989년 미국 IBM T.J. Watson연구소 객원교수. 1988년~1990년 데이타베이스연구회 운영위원장 1989년~1991년 서울대학교 중앙교육연구전산원 원장. 1994년 한국정보과학회 회장. 1997년~현재 한국학술진흥재단 부설 첨단학술정보센터 소장. 1982년~현재 서울대학교 컴퓨터공학부 교수. 관심분야는 데이타베이스, 멀티미디어 데이타베이스