이동체 데이터베이스를 위한 R-tree 기반 메인 메모리 색인의 설계 및 구현†

Design and Implementation of a Main Memory Index based on the R-tree for Moving Object Databases

안성우* / Sung-Woo Ahn 홍봉희*** / Bong-Hee Hong
안경환** / Kyoung-Hwan An 이창우**** / Chang-Woo Lee

요 약

최근 PDA, 휴대폰, GPS와 같은 모바일 기기의 발달로 인하여 이동체에 대한 위치 기반 서비스의 요구가 증대되고 있다. 위치 기반 서비스 기술의 핵심은 이동체로부터 획득된 위치를 효율적으로 저장하고 처리하기 위한 이동체 데이터베이스이며 이동체의 빈번한 보고 데이터를 처리하기 위해서는 서버에서 메인 메모리 DBMS를 유지하는 것이 필요하다. 그러나, 기존 연구에서는 대부분 디스크 기반 환경에서의 이동체 색인을 연구하였으며 이러한 색인은 메인 메모리의 특성을 고려하지 않기 때문에 메인 메모리 DBMS에서는 효율적인 동작을 보장할 수 없다. 따라서, 메인 메모리 환경에 적합한 이동체 색인에 대한 연구가 필요하다. 이 논문에서는 메인 메모리 DBMS에서 이동체의 빈번한 보고 데이터를 처리하기 위한 R-tree 기반의 메인 메모리 색인을 제시한다. 제안한 색인에서는 성장 노드 구조를 사용함으로써 노드 오버플로우 시 노드 분할을 지연하여 노드 분할에 의한 분할 비용이 증가하는 것을 방지한다. 또한, 노드 간의 중첩을 줄이기 위한 합병 후 재분할 정책과 노드 MBR이 차지하는 영역 크기 비율을 줄이기 위한 큰 영역을 가진 노드에 대한 분할 정책을 제안함으로써 검색 성능을 향상시킨다. 성능 평가를 통해서 이 논문에서 제안한 색인은 기존의 색인에 비해서 영역 질의 수행 시 최대 30% 정도의 성능 향상을 보여주고 있다.

Abstract

Recently, the need for Location-Based Services (LBS) has increased due to the development of mobile devices, such as PDAs, cellular phones and GPS. As a moving object database that stores and manages the positions of moving objects is the core technology of LBS, the scheme for maintaining the main memory DBMS to the server is necessary to store and process frequent reported positions of moving objects efficiently. However, previous works on a moving object database have studied mostly a disk based moving object index that is not guaranteed to work efficiently in the main memory DBMS because these indexes did not consider characteristics of the main memory. It is necessary to study the main memory

† 본 논문은 교육인적자원부 지방연구중심대학육성사업(차세대물류 IT 기술연구사업단)의 지원에 의하여 연구되었음.
■ 논문접수 : 2006.7.21 ■ 심사완료 : 2006.8.25

* 부산대학교 컴퓨터공학과 박사과정(swan@pusan.ac.kr)
** 한국전자통신연구원 텔레매틱스·USN연구단 연구원(mobileguru@etri.re.kr)
*** 교신저자 부산대학교 컴퓨터공학과 교수(bdhong@pusan.ac.kr))
**** 삼성전자 무선사업부 연구원(andy38.lee@samsung.com)

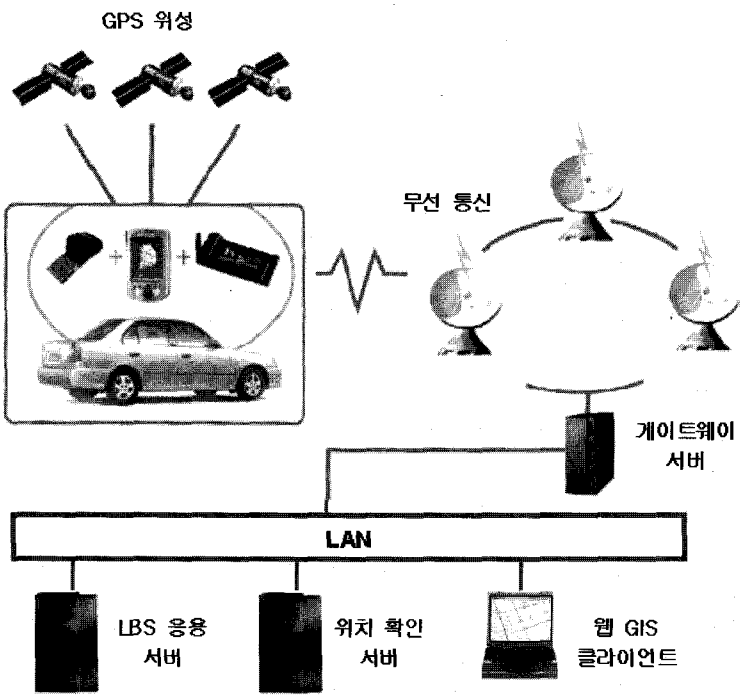
index scheme for a moving object database. In this paper, we propose the main memory index scheme based on the R-tree for storing and processing positions of moving objects efficiently in the main memory DBMS. The proposed index scheme, which uses a growing node structure, prevents the splitting cost from increasing by delaying the node splitting when a node overflows. The proposed scheme also improves the search performance by using a *MergeAndSplit* policy for reducing overlaps between nodes and a *LargeDomainNodeSplit* policy for reducing a ratio of a domain size occupied by node's MBRs. Our experiments show that the proposed index scheme outperforms the existing index scheme on the maximum 30% for range queries.

주요어 : 이동체 데이타베이스, 메인 메모리 색인, R-tree

Keyword : Moving Object Database, Main Memory Index, R-tree

1. 서 론

이동 통신 기술의 발달로 인하여 무선 이동 기기의 사용이 보편화되면서 위치 기반 시스템 (LBS : Location Based Service)의 요구가 나날이 증대되고 있다. 이러한 위치 기반 서비스를 받는 차량이나 개인을 이동체[1]라고 하는데, 이동체들은 GPS 수신기,



<그림 1> 이동체 데이터베이스 환경

무선 모뎀, PDA를 결합한 단말기나 텔레메틱스(telematics) 전용 단말기가 GPS 위성을 이용하여 이동체의 현재 위치 정보를 획득한 다음, 무선 통신을 이용하여 관제 서버에 이동체의 위치를 그림 1과 같이 위치 확인 서버에 전송한다. 서버에서는 각 이동체로부터 받은 위치 정보를 저장할 뿐만 아니라 이동체 검색 요청을 처리하기 위해서 저장된 위치 정보를 이용한다. 이동체 위치 정보를 저장, 검색하기 위해서 서버에서는 이동체 데이터베이스를 포함한다. 따라서, 다수의 사용자들이 요청하는 질의를 효율적으로 처리하기 위해서는 이동체 데이터베이스에서의 빠른 검색이 필요하다.

이동체 데이터베이스의 질의 종류는 크게 영역 질의, 타임스탬프(timestamp) 질의, 궤적 질의, 복합 질의로 나누어진다[2]. 영역 질의는 주어진 시간 간격 동안에 공간 윈도우(spatial windows)에 속하는 이동체들을 검색하는 질의이며, 타임스탬프 질의는 특정 시간에 주어진 공간 윈도우에 속하는 이동체들을 검색하는 질의이다. 그리고, 궤적 질의는 특정한 이동체의 궤적을 검색하는 질의이고, 복합 질의는 시공간 도메인의 영역 질의와 궤적 질의가 복합된 질의이다. 이러한 이동체 데이터베이스에서의 질의를 효율적으로 처리하기 위해서 기존의 연구[2][3][4]에서는 대상으로 하는 질의를 효율적으로 처리하기 위한 색인 기법을 제안하고 있다. 이 논문에서는 "5월 10일 10시부터 12시 사이에 A 영역을 지나간 이동체를 검색하라." 와 같은 영역 질의를 효율적으로 지원하기 위한 색인에 초점을 맞춘다.

유비쿼터스 컴퓨팅 환경에서는 많은 이동체들이 존재할 수 있으며, 이러한 많은 이동체들이 빈번한 위치 보고를 서버에 할 경우 기존의 디스크 기반 데이터베이스로는 디스크 처리 속도의 한계로 인해 갱신 및 검색 연산을 제대로 처리 못하는 경우가 발생할 수 있다. 즉, 디스크 I/O가 허용하는 한계를 초과하여 이동체의 위치 보고 및 위치 추적 질의가 수행될 경우 디스크의 병목현상으로 인해 이동체의 보고를 처리하지 못하는 문제점이 발생하게 되므로 많은 양의 이동체를 관리하고자 할 경우, 메인 메모리를 주기억 장치로 이용하는 메인 메모리 데이터베이스의 사용이 필수적이며 이를 위한 효율적인 색인에 관한 연구가 필요하다.

기존의 디스크 기반 색인[5][6]은 디스크 페이지에 대한 디스크 I/O를 최소한으로 하기 위한 목적으로 만들어졌기 때문에 이를 그대로 메인 메모리 환경에 적용할 경우 성능 결정 요소의 차이로 인해 좋은 성능을 보장하지 못한다. 메인 메모리 색인으로는 하나의 노드 안에 여러 개의 엔트리들을 가지고 있으면서 이진 트리인 T-tree[7]가 가장 널리 알려져 있다. T-tree는 binary-tree와 B+-tree의 특징을 모두 가지고 있다. 다른 색인으로는 캐쉬 라인(cache line)의 활용도를 높이기 위해 노드 내 포인터들을 제거한 CSB+-tree[8]가 있다. 그러나, T-tree와 CSB+-tree는 1차원 데이터를 위한 색인으로 다차원 데이터를 처리하는 이동체 데이터베이스 환경에서는 적합하지 않다. 다차원 데이터를 위한 색인으로는 노드 내에 최소 경계사각형(MBR : Minimum Bounding Rectangle)을 압축하여 L2 캐쉬 미스를 줄인 CR-tree[9][10]가 있다. 다차원의 경우 포인터에 비해 키의 크기가 크기 때문에 키에 대한 압축을 수행함으로써 캐쉬 라인의 활용도를 높일 수 있다. 그러나, 디스크 기반에서는 디스크 I/O가 가장 큰 성능 결정 요소이지만 메인 메모리의 경우 캐쉬 미스 이외에도 많은 요소들이 영향을 미치므로 기존의 방식대로 노드의 크기를 결정하는 것이 항상 좋은 성능을 보장하지는 못한다[11]. 따라서, 캐쉬 활용도 이외에 이동체 데이터베이스 환경에서의 성능 결정 요소를

고려하여 이동체를 효율적으로 저장하고 빠른 검색을 지원하기 위한 메인 메모리 색인에 대한 연구가 필요하다.

이 논문에서는 디스크 기반의 다차원 색인으로 널리 알려진 R-tree[5]를 메인 메모리에 적재했을 경우에 발생하는 문제점에 대해서 분석을 하고 이동체 데이터베이스 환경에 적합한 R-tree 기반 메모리 색인을 제시한다. 제안된 색인에서는 성장 노드 구조를 사용하여 노드 분할을 최소화 함으로써 고정된 노드 크기를 사용할 경우 노드 오버플로우 시 노드 분할에 따른 분할 비용이 증가하는 것을 방지한다. 또한, 분할 연기를 통해 이동체들이 계속해서 위치 데이터를 보고할 경우 근접한 궤적들이 같은 노드에 들어갈 수 있도록 한다. 노드의 분할이 일어날 때 노드 간의 중첩을 줄이기 위해서 합병 후 재분할(*MergeAndSplit*) 정책과 색인에서 노드가 차지하는 사장 영역을 줄이기 위해서 큰 영역을 가진 노드 분할(*LargeDomainNodeSplit*) 정책을 제안한다. 제시된 색인 구조는 실험 평가를 통하여 메인 메모리에 적재한 기존의 3DR-tree[3] 보다 영역 질의 시 성능이 최대 30% 개선됨을 알 수 있다.

이 논문의 구성은 다음과 같다. 2장에서는 관련 연구와 문제점에 대해서 언급하고, 3장에서는 문제 정의를 기술한다. 4장에서는 성장 노드 구조를 사용한 R-tree 기반 메인 메모리 색인 기법을 제시한다. 5장에서는 성능 평가 및 결과에 대한 분석을 하고, 6장에서는 결론 및 향후 연구를 기술한다.

2. 관련연구

이 장에서는 메인 메모리 기반 색인과 디스크 기반 색인에 관한 기존 연구를 살펴본다. 먼저 메인 메모리 기반 색인으로 널리 알려진 T-tree, CSB+-tree, CR-tree, DR-tree에 대해서 살펴보고, 디스크 기반의 색인으로는 다차원 색인으로 널리 알려진 R-tree 계열에 대해서 살펴본다.

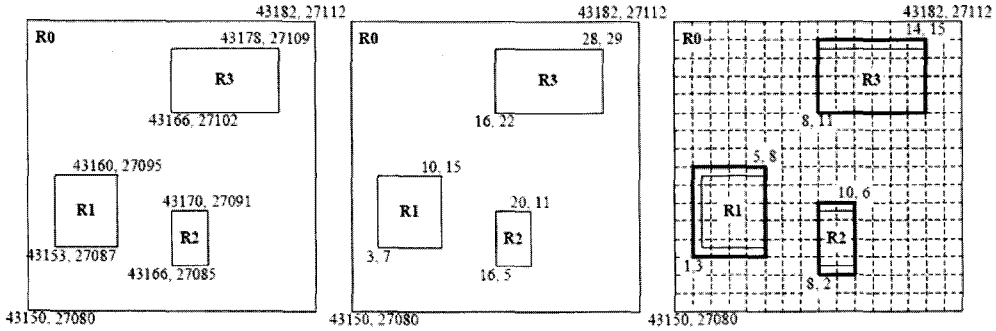
2.1 메인 메모리 기반 색인에 관한 연구

메인 메모리 기반의 색인으로는 1차원 데이터를 위한 T-tree, CSB+-tree가 있고, 다차원 데이터에 대한 색인으로는 CR-tree 계열과 DR-tree가 있다.

T-tree[7]는 기존의 B-tree와 AVL-tree로부터 진화되어 나온 대표적인 메인 메모리 색인 구조이다. T-tree는 이진 검색과 높이 균형을 가지는 AVL-tree의 성질을 가지고 있고, 한 노드 안에 여러 개의 데이터를 가지는 B-tree의 성질을 가지고 있다. T-tree는 빠른 처리 속도와 메모리 사용의 최적화라는 메인 메모리의 특성에 적합한 구조로 알려져 있지만, 빈번한 데이터 갱신 및 다차원 데이터를 대상으로 하는 이동체 데이터베이스에 적용하기에는 어려움이 있다.

CSB+-tree[8]는 자식 노드들을 연속적으로 저장하고 부모 노드는 하나의 포인터만을 저장함으로써 캐쉬 라인의 활용도를 높이는 색인으로 포인터를 가지지 않는 자식 노드는 계산에 의해서 구해질 수 있다. 실험에서 T-tree에 비해 높은 검색 성능을 보였으나 자식 노드들이 연속적으로 저장되어 있기 때문에 갱신이 빈번할 경우 복사가 많이 일어나므로 이동체 데이터베이스 환경에는 적용하기 어려운 단점이 있다.

CR-tree[9][10]는 R-tree에 기반을 둔 색인으로서 CSB+-tree와는 달리 키에 대한 압축을 수행함으로써 캐쉬 라인의 활용도를 높이는 색인이다. 다차원 색인의 경우 일차원 색인과는 달리 포인터에 비해 키의 크기가 더 크므로 포인터의 제거보다는 키에 대한 압축이 더 많은 효과를 가져올 수 있다. CR-tree에서는 그림 2와 같이 일반 좌표에



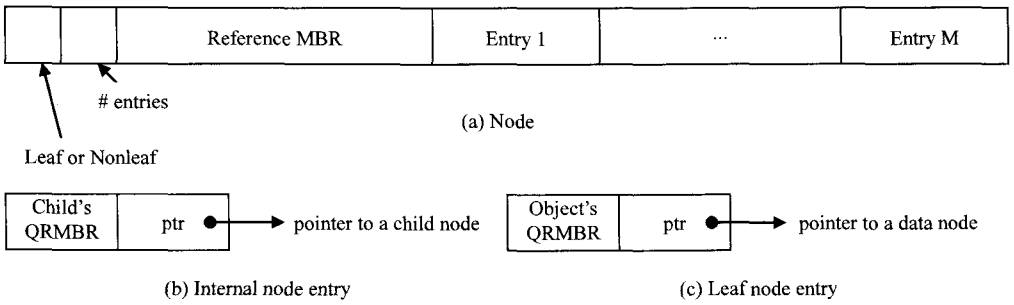
<그림 2> CR-tree의 키 압축 방식

대해 먼저 상대 좌표 값을 구하고 이를 양자화(quantization) 방법을 통하여 키를 압축하고 있다.

그림 3은 노드 구조를 나타내고 있는데 원래 좌표 값을 구하기 위한 Reference MBR을 노드에 저장하는 점이 기존의 노드 구조와는 차이가 나며, 검색 시에 양자화로 인한 적중 오류(false hit)가 기존의 방법보다 조금 더 증가할 수 있다. CR-tree는 기존의 R-tree의 노드 삽입 및 분할 정책을 그대로 사용하며 고정된 노드 크기를 사용하기 때문에 빈번한 위치 보고를 하는 이동체 환경에 적용할 경우 노드 분할이 빈번히 발생하며 그에 따른 비용이 증가하는 문제가 있다.

R-tree 알고리즘이 데이터 삽입 시에 노드 간의 중첩이 심하고, 많은 사장 공간을 유발시키는 문제점을 가지고 있기 때문에 기존의 연구에서는 이러한 문제점을 해결하기 위해서 다양한 R-tree 기반의 색인 구조를 제시하고 있다. 따라서, CR-tree에서 사용하는 노드 MBR의 양자화 방법에 대한 성능을 극대화 시키기 위해서는 개선된 R-tree 알고리즘의 적용이 필요하다. 이러한 요구 사항을 반영하여, [12]에서는 CR-tree에서 적용하고 있는 QRMBR 방법을 다양한 R-tree 계열의 색인에 적용하여 메인 메모리 환경에서의 성능 개선 사항에 대해서 기존의 R-tree 알고리즘과 비교 평가하고 있다.

QRMBR 기법을 적용하고 실험 평가를 수행한 색인으로는 R-tree 알고리즘을 그대로 적용한 CR-tree와 R*-tree 알고리즘을 적용한 CR*-tree, 그리고 Hilbert R-tree[13] 알



<그림 3> CR-tree의 노드 구조

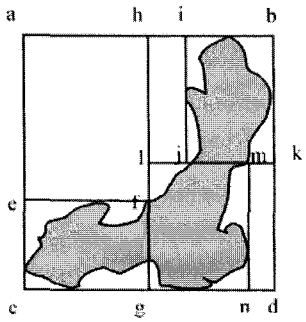
고리즘을 적용한 Hilbert CR-tree를 사용하였다. 또한, 다양한 환경에 대한 실험을 수행하기 위해서 균등 분포(uniform distribution), 편향 분포(skewed distribution) 데이터에 대해서 색인에 대한 순차 접근(sequential access) 환경과 동시 접근(concurrent access) 환경으로 구분하여 검색 및 갱신 성능을 측정하고 있다.

실험 결과 순차 접근 환경에서는 CR*-tree가 가장 좋은 검색 성능을 보였으며 Hilbert R-tree가 가장 좋은 갱신 성능을 보였다. 동시 접근 환경에서는 데이터의 분포가 균등 분포일 때 Hilbert R-tree가 가장 좋은 검색 성능을 보였으며 편향 분포일 때 R*-tree가 가장 좋은 검색 성능을 보였다. 또한, 순차 접근과 동일하게 갱신 성능은 Hilbert R-tree가 가장 좋은 것으로 나타났다. 결과적으로, 공간 영역에 대한 순서화와 검색 시 캐쉬 라인의 활용도를 높이기 위한 QRMBR을 적용한 Hilbert CR-tree가 모든 환경에서 가장 만족할 만한 성능을 보이는 것으로 나타났다.

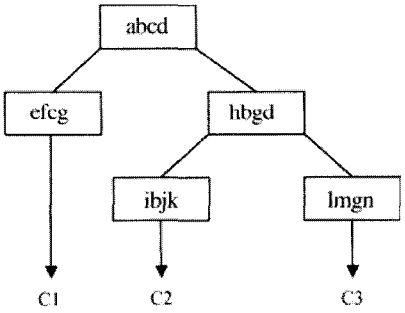
그러나, [12]에서 보여주는 각 색인의 성능 평가를 통해서 다음과 같은 결론을 얻을 수 있다. 첫째, 노드 MBR의 양자화 방법은 색인의 검색 성능을 기존의 알고리즘보다 향상시키고 있지만 QRMBR의 유지에 따른 비용 때문에 갱신 성능이 상대적으로 떨어지는 문제를 가지고 있다. 둘째, 기존의 R-tree 계열 색인 알고리즘에 QRMBR을 적용시켜 성능을 향상시키고 있지만 각 색인 알고리즘이 가지고 있는 문제점도 그대로 반영되어 상대적으로 색인 성능을 감소시키는 문제점을 가지고 있다. 따라서, 메인 메모리 환경에서 다차원 색인에 대해서 캐쉬 라인 활용도 이외에 색인의 검색 및 갱신 성능을 높일 수 있는 기법에 대한 연구가 필요하다.

DR-tree[14]는 R-tree 기반의 기존 다차원 색인에서 MBR이 객체 영역에 대한 대략적인 정보만을 제공함으로써 질의 수행 시 여과 단계(filtering step)에서 적중 오류가 발생한 객체를 많이 포함하며, 정제 단계(refinement step)에서 객체가 복잡한 모양을 가지고 있을 때 질의 조건을 만족하는지 검사하는 비용이 큰 단점을 해결하기 위해서 제안된 색인이다. 여과 단계와 정제 단계에서 발생하는 두 가지 문제점을 해결하기 위해서 DR-tree는 객체를 모두 포함하는 MBR을 여러 개의 작은 MBR로 변환하여 DMBR(Decomposed MBR)로 구성하며, DMBR을 이용하여 복잡한 공간 객체를 단순한 구조로 분할(decomposition)하여 정제 단계에서의 질의 영역과의 겹침 비교에 대한 비용을 줄이고자 한다.

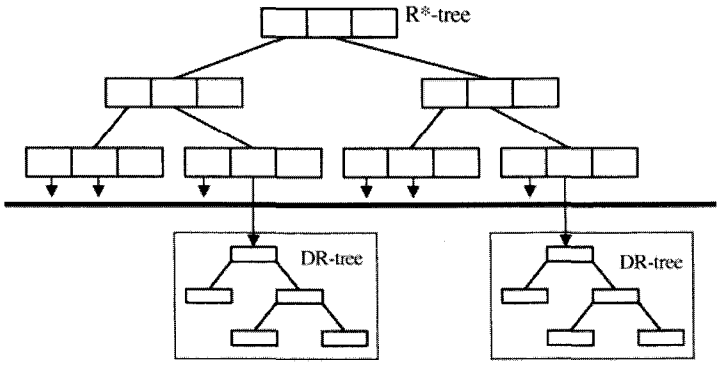
그림 4의 (a)는 공간 객체를 여러 개의 DMBR로 분할하여 표현하는 방법에 대한 예시를 보여주며 (b)는 (a)의 공간 객체를 색인으로 구성한 예시를 보여주고 있다. DMBR을 구성하기 위해 n차원의 공간 객체에 대해 해당 MBR을 각 차원을 순회하면서 MBR 영역의 중심을 이등분하여 새로운 MBR(DMBR)을 만들어 낸다. 미리 정해놓은 임계치에 도달하게 되면 DMBR을 생성하는 작업을 마치게 된다. (b)는 (a)의 공간 객체에 대한 DR-tree의 구조로써 DR-tree는 각 공간 객체에 대해서 구성된 DMBR을 기반으로 LSB-tree와 유사한 구조로써 생성된다. (c)는 DR-tree를 이용하여 공간 객체를 저장하기 위해서 기존의 R*-tree와 연동하여 색인을 구성한 모습이다. R*-tree는 디스크에 저장되어 있는 색인으로 원본 공간 객체에 대한 색인을 구성하고 있다. DR-tree는 메인 메모리에 상주하고 있는 색인으로 각 공간 객체에 대해서 DMBR을 이용하여 분할된 공간 객체를 색인하고 있으며 R*-tree의 단말 노드에서 참조하고 있다. 질의 수행 시 질의 영역과 겹치는 공간 객체를 찾기 위해서 R*-tree에서 여과 단계를 거쳐 후보 객체



(a) DMBR 집합 구성의 예제



(b) DMBR을 이용한 DR-tree 구성 예제



(c) 공간 객체를 저장하기 위한 두 단계의 색인 구조

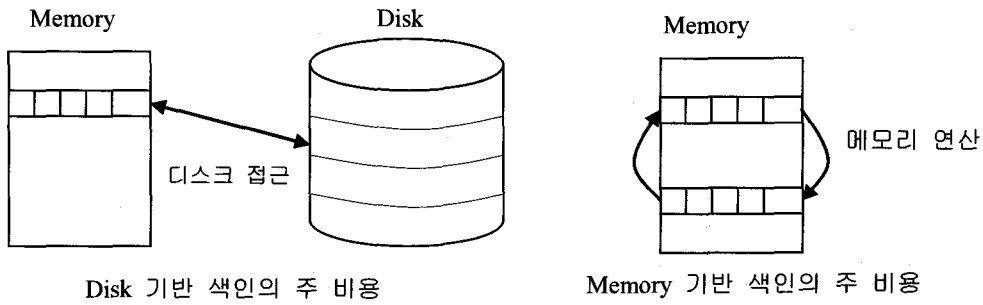
<그림 4> DR-tree를 이용한 공간 객체의 색인 방법

를 찾으며 정제 단계에서 메인 메모리에 구성된 DR-tree를 조사하여 최종 결과를 찾아낸다.

메인 메모리에 구성되어 있는 DR-tree를 정제 단계에서 검색함으로써 DMBR을 이용하여 단순한 모양으로 분할된 복잡한 공간 객체에 대한 비교 연산의 비용을 크게 줄일 수 있는 장점이 있다. 그러나, DR-tree는 정적인 공간 객체에 대해서 미리 색인을 구성함으로써 검색 성능을 높일 수 있으나 이동체 데이터베이스 환경과 같이 빈번한 데이터의 삽입 및 갱신이 일어나는 환경에서는 DMBR 구성 비용이 크기 때문에 성능을 보장하지 못한다. 그리고, 복잡한 모양을 가진 공간 객체가 다수 존재하는 환경에서는 DR-tree를 이용한 색인이 효과적으로 적용될 수 있지만 이동체가 보고하는 데이터는 색인에서 점 또는 선분으로 표현되기 때문에 정제 단계에서의 비교 연산 비용을 크게 줄일 수 없다.

2.2 디스크 기반 색인에 관한 연구

R-tree[4]는 데이터 분할 방법의 대표적인 디스크 기반 공간 색인으로서 공간 객체를 MBR을 사용하여 표현하고 저장하는 균형 트리 구조의 색인으로서 B-tree에 대해서 n 차원으로 확장한 모델이다. 공간 객체를 색



<그림 5> 디스크 기반 색인과 메모리 기반 색인의 주 비용

인에 삽입 시에 최소 영역 확장 정책(Least Area Enlargement Policy)를 사용하여 노드를 선택하며, 노드 오버플로우가 발생했을 시에는 분할되는 2개의 노드가 최소 영역을 갖도록 분할한다. R-tree는 단말 노드와 비단말 노드로 구성되며, 단말 노드의 엔트리는 <*oid*, *MBR*>을 가지고 비단말 노드의 엔트리는 <*child pointer*, *MBR*>을 가지고 있다.

그러나, R-tree는 노드 간의 중첩이 심하고, 많은 사장 공간을 가지고 있기 때문에 이러한 문제를 해결하기 위하여 새로운 삽입 및 분할 정책을 사용한 R*-tree[6]가 제안되었다. R*-tree는 R-tree의 변형으로서 삽입 시 단말 노드를 선택할 때 R-tree와는 달리 최소 중첩 확장 정책(Least Overlap Enlargement Policy)를 사용하며, 노드 분할 시 둘레(perimeter)와 중첩(overlap)을 고려한 새로운 분할 정책을 제안하였다. 또한, 사장 공간을 최소화하기 위한 재삽입 전략(Reinsertion Strategy)를 제안하였다. R*-tree는 R-tree에 비해서 성능의 향상을 가져왔지만 제안한 알고리즘들은 복잡도가 더 증가했다. 따라서, CPU 계산 비용이 중요한 메모리 기반 이동체 데이터베이스 환경에서는 삽입 성능이 매우 저하되기 때문에 사용하기 어렵다.

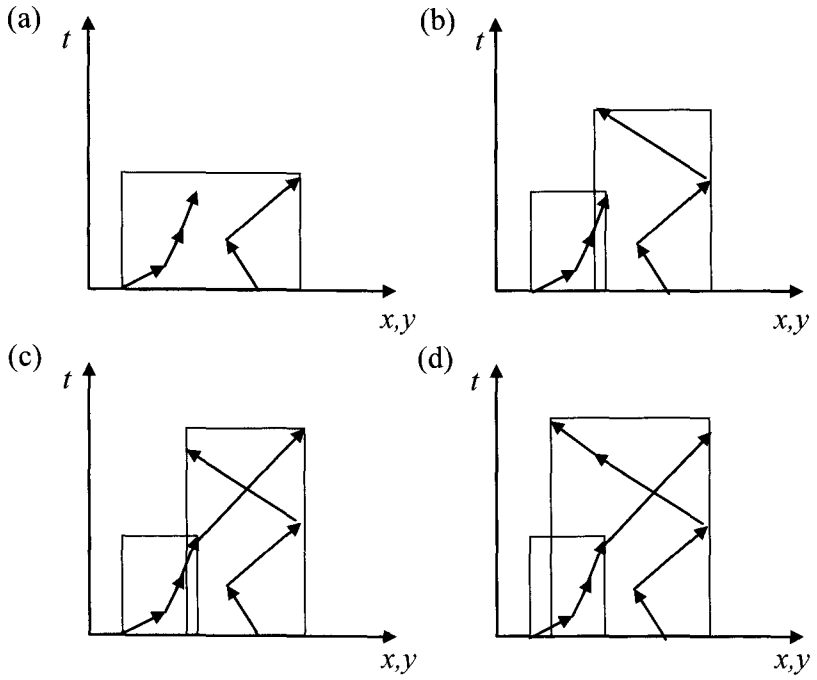
이러한 R-tree계열의 색인은 디스크의 특성을 고려한 디스크 기반의 색인으로서, 디스크의 I/O를 최소화하는 데 초점이 맞추어져 있다. 따라서, 메인 메모리 환경에서 사용하였을 경우에는 최적의 성능을 보장할 수 없다.

3. 문제정의

2장에서 살펴본 바와 같이 R-tree는 디스크 기반 색인으로 메모리 기반 이동체 데이터베이스에 적용했을 때 최적의 성능을 보장할 수 없다. 이 장에서는 메인 메모리 기반 이동체 데이터베이스에 디스크 기반 색인인 R-tree를 적용했을 경우 발생하는 문제점 및 메인 메모리 환경에서의 성능 결정 요소에 대해서 살펴본다.

3.1 삽입 성능 결정 요소

R-tree는 디스크 환경을 고려하여 설계되었기 때문에, R-tree의 분할 방법도 디스크의 I/O를 최소화하는데 초점이 맞추어져 있다. 디스크에서는 디스크에 저장되어 있는 데이터에 접근하기 위해서 페이지 단위로 접근하기 때문에 R-tree에서 최적의 성능을 얻기 위해서는 노드의 크기를 디스크 페이지의 크기에 맞추고, 노드의 용량이 가득 차



<그림 6> 삽입 순서에 따른 색인 노드 간의 중첩 비교

면 분할을 실시한다.

그러나, 그림 5와 같이 메인 메모리 기반의 색인에서는 디스크의 I/O가 존재하지 않고 메모리에서의 전체 수행 시간이 중요하기 때문에 기존의 R-tree의 분할 방법은 더 이상 메인 메모리 환경에 적합하지 않다. 고정된 노드의 크기를 사용하게 되면 검색 성능과는 상관없이 최대 엔트리 수에 의해서 분할 시점이 결정됨으로 인해 분할하지 않았을 때 보다 분할했을 때 노드들의 MBR 크기가 커지게 되는 경우 결과셋 데이터를 가지고 있지 않으면서 노드가 선택될 확률이 높아져서 검색 성능이 더 나빠질 수 있다. 메인 메모리 기반 색인에서는 노드를 접근하는 비용이 노드의 메모리 주소에 대한 포인터를 할당하고 획득하는 비용이기 때문에 다른 연산에 비해서 상대적으로 작다 [15]. 따라서, 디스크 기반 색인과 달리 메인 메모리 색인에서는 노드의 크기가 전체 비용에 영향을 많이 미치지 않으며, 필요 없는 노드 분할은 분할 비용이 발생되어 삽입 성능을 저하시키기 때문에 데이터 삽입 시 노드의 분할을 최소화하여 CPU 사이클에 따른 전체 수행 시간을 최소화 하는 것이 필요하다.

3.2 검색 성능 결정 요소

R-tree는 삽입 순서에 의해 검색 성능이 크게 영향을 받는다. 즉, 삽입 순서에 따라 색인의 성능이 아주 나빠지게 되는 경우가 발생하게 된다. 이동체 데이터베이스에서는 이동체들이 계속해서 위치 데이터를 보고하므로, 분할 연기를 통해서 근접한 궤적들이 같은 노드에 들어갈 수 있도록 하는 것이 필요하다. 또한, 노드의 분할이 발생할 때

노드 간의 중첩 영역을 최소화해야 하며 노드가 차지하는 영역 중 사장 영역을 줄이는 방법이 필요하다. 그림 6은 시간에 지나면서 노드의 용량이 5인 색인에서 삽입 순서에 따라 a)에서 d)로 가면서 노드 간의 중첩이 커지고 색인의 성능이 나빠지는 변화를 나타낸 것이다.

R*-tree에서는 이러한 문제점을 해결하기 위해서 2장에서 살펴본 바와 같은 새로운 삽입 및 분할 정책을 사용하여 성능 향상을 가져왔다. 그러나, 기존의 디스크 기반 R-tree의 성능에 있어 알고리즘의 복잡도는 크게 고려하지 않기 때문에 R*-tree가 좋은 성능을 낼 수 있지만, 메인 메모리 기반 환경에서 알고리즘의 복잡도가 성능에 가장 큰 영향을 주기 때문에 R*-tree의 삽입 및 분할 정책을 적용하기 힘들다.

4. 성장 노드 정책

이 장에서는 본 논문에서 제안하고자 하는 메인 메모리 환경에 적합한 R-tree 기반 이동체 색인을 구성할 때 색인에 데이터 삽입 시 발생하는 분할 비용을 최소한으로 하기 위해 노드 분할을 지연시키는 성장 노드 구조를 제안한다. 또한, 분할이 발생하였을 경우 노드 간의 중첩을 줄이기 위한 합병 후 재분할 정책과 노드의 사장 영역을 최소한으로 하기 위한 큰 영역을 가진 노드 분할 정책을 제시한다.

4.1 성장 노드

메인 메모리 기반에서는 노드의 용량(capacity)에 제약이 없다. 디스크 기반에서는 디스크 I/O를 줄이는 것이 가장 큰 목표이기 때문에 디스크 페이지 크기에 따라 노드의 용량이 정해지는 반면에, 메인 메모리에서는 디스크 I/O가 없기 때문에 노드의 용량에 아무런 제약이 없다. 이 논문에서는 단말 노드에서 그림 7과 같이 성장 노드(*Growing Node*)를 제안한다.

기존 R-tree에서는 노드의 용량이 가득 차면 분할을 실시하지만, 이 논문에서 제안하는 색인에서는 노드 간의 중첩이 심하지 않는 경우와 노드의 크기가 아주 크지 않는 경우에는 분할을 하지 않고, 새로운 노드를 만들어 기존의 일반 노드에 연결을 시키는 방법을 사용하여 성장 노드로 전환한다. 즉, 검색 성능을 저하시키는 노드 간의 중첩과

```
Algorithm OverflowProcessing(Node node)
1   BEGIN
2   // if a normal node or a growing node overflows,
3   // the index applies to the node split policy based on a delayed node split
4   IF node overflows THEN
5           IF node.MBR > MBR_threshold THEN
6                   reconstruct the index based on the R  tree split policy
7           ELSE IF OverlapRatio(node.MBR) > Overlap_threshold THEN
8                   reconstruct the index based on MergeAndSplit(node)
9           ELSE
10                  create a new growing node and connect it to the overflowed node
11          END IF
12  END IF
13  AdjustTree()
14  END
```

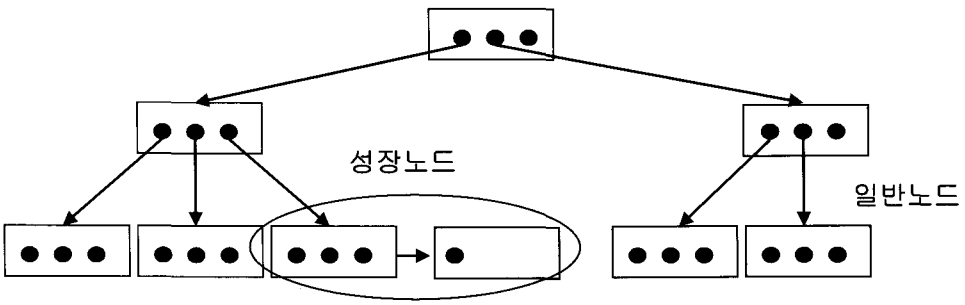
알고리즘1. 노드 오버플로우 발생 시 성장노드 적용 알고리즘

사장 영역이 커지지 않으면 노드는 계속 성장(growing)함으로써 검색 성능을 보장하도록 하고 있다. 노드의 오버플로우가 발생했을 때 성장 노드에 기반한 분할 정책을 적용한 알고리즘은 알고리즘 1과 같다.

알고리즘 1에서 노드의 분할 조건을 만족하지 않으면 오버플로우가 발생한 노드에 새로운 성장 노드를 생성하고 연결한다. 노드 분할 조건을 체크하기 위한 *MBR_hreshold*와 *Overlap_threshold*는 4.2절과 4.3절에서 설명하고 있는 분할 정책에 사용되는 임계값이다. 즉, 입력된 노드의 *node_MBR*이 다른 노드와 겹치는 비율이 *Overlap_threshold*를 초과할 경우 4.2절에서 설명하는 합병 후 재분할 정책에 따라서 노드를 분할하며 *node_MBR*이 *MBR_threshold*를 넘어설 경우 4.3절에서 설명하고 있는 큰 영역을 가진 노드 분할 정책에 따라서 노드를 분할한다. 알고리즘 1에서 제시한 것과 같이 특정 노드의 오버플로우가 발생했을 때 분할을 수행하지 않고 성장 노드를 유지하기 위해서 필요한 속성은 표 1과 같다.

표 1에서 *nodeStatus*는 해당 노드의 상태를 나타낸다. *nodeStatus*가 0이면 일반 노드를 나타내고, 2인 경우는 성장 노드의 마지막 노드를 나타내며 1인 경우는 2인 경우를 제외한 모든 성장 노드를 나타낸다. *nextNode*는 성장 노드에서 다음 노드를 가리키는 포인터로서 삽입 시나 검색 시에 성장 노드를 검사할 때 사용된다. *initialArea*는 일반 노드가 성장 노드로 전환하기 직전에 일반 노드의 영역의 크기를 가진다. 이것은 4.3절에서 설명하고 있는 큰 영역을 가진 노드 분할 정책에서 사용된다.

성장 노드 구조를 단말 노드 뿐만 아니라 비단말 노드에서도 적용하는 것을 고려할 수 있다. 그러나, 비단말 노드에서 성장 노드를 사용하는 경우에는 노드 당 엔트리들의 수가 많아지게 되어 데이터 삽입 시에 삽입할 단말 노드를 찾기 위해서 비단말 노드에 있는 엔트리 수를 비교하는 Choose



〈그림 7〉 성장 노드를 통한 오버플로우 시의 노드 분할 지연
알고리즘1. 노드 오버플로우 발생 시 성장 노드 적용 알고리즘

〈표 1〉 성장 노드의 속성

속성 타입	속성명	설 명
Integer	*nodeStatus*	성장 노드인지 일반 노드인지를 나타냄
Node	*nextNode*	성장 노드에서 다음 노드를 가리킴
Float	*initialArea*	성장 노드로 전환하기 전의 일반 노드의 영역의 크기

Subtree를 수행하게 되면 비교하는 엔트리의 개수가 많아지기 때문에 삽입 성능이 저하되는 문제가 있다. 따라서, 성장 노드는 단말 노드에서만 사용하는 것이 적절하다.

성장 노드의 삽입은 일반 노드의 삽입과 비교했을 때 한 가지를 제외하고 동일하다. 일반 노드에서는 데이터를 삽입하고 난 후에 해당 노드의 MBR을 조절하는 반면 성장 노드는 데이터를 삽입 후 성장 노드의 맨 앞에 연결되어 있는 일반 노드의 MBR을 조절한다. 이와 같이 함으로써 성장 노드와 연결되어 있는 맨 처음의 일반 노드가 성장 노드 전체의 MBR을 가지고 있기 때문에 검색 시 성장 노드를 순회하면서 질의 영역과 겹치는지에 대해서 검사하는 것을 줄일 수 있다.

성장 노드를 사용함으로써 분할 연기에 따른 비단말 노드의 엔트리 수 감소로 인한 삽입 성능의 개선이 있다. 그리고, 분할을 지연함으로써 삽입 순서에 영향을 받아 성능이 나빠지는 R-tree의 단점을 보완할 수 있기 때문에 검색 성능의 향상을 가져온다. 따라서, 검색 성능의 향상을 위한 적절한 분할 시기를 선택하는 것이 필요하다.

4.2 합병 후 재분할

3.2절에서 언급했듯이, 기존의 R-tree는 삽입 순서에 따라 노드 간의 심한 중첩이 발생할 수 있기 때문에 검색 성능이 저하될 수 있다. 따라서, 이 논문에서는 색인이 바람직하지 않는 방향으로 구축되는 것을 방지하고 노드 간의 중첩을 감소시키기 위해서 단말 노드에서 합병 후 재분할(*Merge AndSplit*) 정책을 제안한다. 제안 방법의 설명을 위해서 표 2에서 노드 간의 중첩 비율에 대한 용어 정의를 한다.

합병 후 재분할 정책은 일반 노드에서 오버플로우가 발생한 경우와 성장 노드에서 오버플로우가 발생한 경우에 수행된다. 성장 노드는 본 논문에서 제시하는 분할 조건을 만족하지 않을 경우 일반 노드에 연결되어 계속 성장하기 때문에 일반 노드에 연결되어 있는 성장 노드 중 맨 끝에 연결되어 있는 성장 노드에서 오버플로우가 발생한다. 일반 노드나 맨 마지막에 연결되어 있는 성장 노드에서 오버플로우가 발생하게 되면 해당 노드와 형제 노드 간의 중첩 비율을 계산하여 EOR 이상인 경우에는 두 노드를 합병 후 분할을 하고 나서 해당 노드에 엔트리들을 다시 할당한다.

알고리즘 2는 합병 후 재분할 정책의 알고리즘을 기술한 것이다.

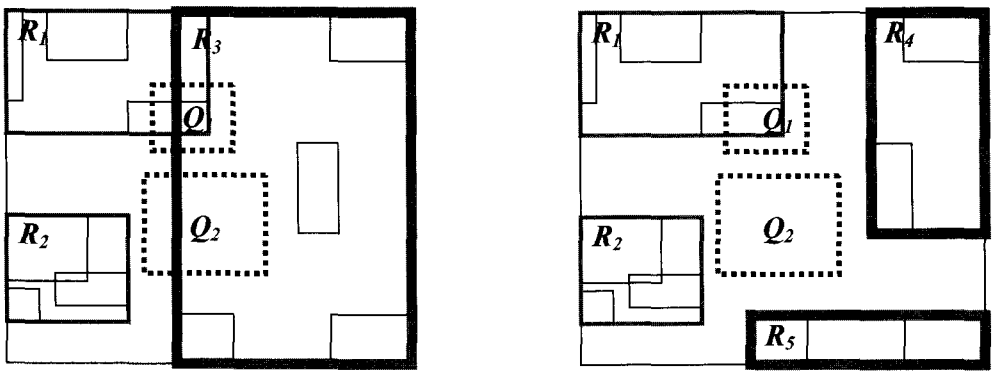
〈표 2〉 EOR (Excessive Overlap Ratio)

EOR(ExcessiveOverlapRatio)
노드 간의 심한 중첩 비율을 나타내는 용어로서 두 개의 노드 간의 중첩 비율이 EOR 이상일 경우에 이 논문에서는 심한 중첩이 발생했다고 정의한다. EOR을 30%, 40%, 50%, 60%로 변경하며 실험한 결과 40%, 즉 0.4일 때 가장 좋은 성능을 보인다.

```
Algorithm MergeAndSplit(Node node)
1  BEGIN
2  // choose the sibling node that overlaps with
   node the most
3  // and then merge it to node and split the
   merged node based on the R tree split
   policy
4  calculate an overlap ratio between sibling
   nodes of node
5  choose most overlapped sibling node mo_node
6  merge and split node with mo_node
7  allocate entries to node, mo_node
8  END
```

알고리즘2. 합병 후 재분할 정책의 알고리즘

알고리즘 1에서와 같이 *node.MBR*이 형제 노드의 MBR과 겹쳐지는 비율이 EOR 이상일 경우 알고리즘 2의 MergeAndSplit 함수에 따라 해당 노드와 형제 노드 간의 합병 후 재분할 정책을 수행한다. *node*와



(a) 큰 영역을 가진 노드가 존재할 때 (b) 큰 영역을 가진 노드가 존재하지 않을 때

<그림 8>큰 영역을 가진 노드의 문제점

영역 겹침 비율이 EOR 이상인 *mo_node*가 선택되면 두 노드를 합병한 후 기존의 선형 알고리즘에 따라 분할된 두 노드 *node*, *mo_node*에 각각 엔트리를 다시 할당한다. 기존의 선형 알고리즘에서는 노드를 분할할 때 (노드 용량 + 1)의 크기를 가지고 분할을 하지만 두 노드를 합병했을 때에는 합병 후 재분할 정책에 참여하는 노드의 종류에 따라 (노드 용량 + 1)보다 많은 엔트리를 가지고 분할을 하는 경우가 발생할 수 있다. 두 노드가 모두 일반 노드일 경우는 두 일반 노드에 모든 엔트리가 분배되어 일반 노드와 일반 노드로 될 수도 있지만 엔트리 분배 시 하나의 일반 노드 용량을 초과하여 일반 노드와 성장 노드의 구성이 될 수 있다. 합병 후 재분할 정책에 참여하는 노드 중 하나 이상이 성장 노드일 경우에는 일반 노드와 성장 노드, 성장 노드와 성장 노드의 형태로 구성될 수 있다. 따라서, 합병 후 재분할 정책에 참여하는 노드의 종류에 따라서 분할 후 생성되는 두 노드는 성장 노드와 성장 노드, 성장 노드와 일반 노드, 일반 노드와 일반 노드의 모든 경우가 발생될 수 있다.

이와 같은 합병 후 재분할 정책을 사용함으로써 중첩 비율이 심한 두 노드 간의 중첩을 감소시켜 색인 검색 시에 노드 방문 횟수를 줄여 검색 성능 향상을 가져올 수 있다. 합병 후 재분할 정책은 분할 시에 기존의 R-tree에서 사용하고 있는 선형 알고리즘과 동일한 방법을 사용하기 때문에 복잡도는 선형 알고리즘과 마찬가지로 $O(n)$임을 알 수 있다.

4.3 큰 영역을 가진 노드 분할

색인을 구성하고 있는 노드 중 큰 영역을 가진 노드는 포함하고 있는 데이터가 차지하는 영역보다 빈 영역이 상대적으로 많아질 가능성이 크기 때문에 다른 노드에 비해 많은 사장 영역을 유발할 수 있다. 이는 질의 시에 불필요한 노드 방문을 발생시켜 검색 성능을 저하시키는 요인이 된다.

그림 8에서는 큰 영역을 가진 노드로 인한 검색 시 노드 방문 횟수의 증가에 대한 예를 보여주고 있다. 그림 8에서 보듯이 (a)는 주위 노드(R_1, R_2)들의 영역 크기에 비해서 큰 영역을 가진 노드(R_3)가 존재하는 경우이고, (b)는 (a)에서 R_3에 포함되어 있던 객체들을 다른 노드와 중첩되지 않게 상대

적으로 작은 크기를 가진 노드(R_4, R_5)로서 구성한 경우이다. 두 가지 경우의 색인에 대하여 질의 영역이 Q_1, Q_2와 같은 영역으로 설정되어 질의를 할 경우 (a)는 (b)보다 상대적으로 큰 영역을 가진 노드인 R_3가 질의 영역에 포함되지만 질의 영역에 포함되는 객체가 존재하지 않기 때문에 불필요한 노드 방문을 초래한다. 이와 같이 큰 영역을 가진 노드가 색인에 많이 존재하게 되면 영역 질의 시에 불필요한 노드 방문 횟수를 증가시켜 검색 성능을 떨어뜨리는 문제가 발생한다.

이 절에서는 검색 시 노드의 불필요한 방문으로 인한 검색 성능 저하를 방지하기 위하여 큰 영역을 가진 노드의 분할(*Large DomainNodeSplit*) 정책을 제안한다. 제안 방법의 설명을 위해서 표 3과 같은 용어 정의를 한다.

<표 3> REAR (Relatively Excessive Area Ratio)와 AEAR (Absolutely Excessive Area Ratio)

REAR (Relatively Excessive Area Ratio)
노드에 대한 심한 영역 크기 비율을 나타낸 용어로서 색인의 같은 레벨의 노드들의 평균 영역 크기와 해당 노드의 영역 크기의 비율이 REAR 이상이면 심한 사장 공간을 나타낸다고 정의한다. 같은 레벨 노드들의 평균 크기의 2, 3, 4, 5, 10배로 REAR 값을 변경하면서 실험한 결과 REAR가 3일 때 가장 좋은 성능을 보인다.
AEAR (Absolutely Excessive Area Ratio)
한 성장 노드에 대한 심한 영역 크기의 비율을 나타낸 용어로서 성장하기 전의 일반 노드에서의 영역 크기와 성장 노드의 영역 크기의 비율이 AEAR 이상이면 심한 사장 공간을 나타낸다고 정의한다. 성장하기 전 일반 노드에서의 크기의 2, 3, 4, 5, 10배로 AEAR 값을 변경하면서 실험한 결과 AEAR가 3일 때 가장 좋은 성능을 보인다.

이 논문에서는 큰 영역을 가진 노드를 분할함으로써 영역의 크기와 사장 공간을 줄이고자 한다. 큰 영역을 가진 노드 분할 정책도 합병 후 재분할 정책과 마찬가지로 일반 노드와 성장 노드가 오버플로우가 발생 시에 사용한다. 일반 노드나 성장 노드가 오버플로우가 발생 시에 영역 크기 비율이 REAR이나 AEAR 이상인 경우에는 큰 영역을 가진 노드 분할을 실시한다. 즉, 노드의 크기가 큰 경우 분할을 함으로써 노드의 크기를 줄여서 검색 성능을 향상 시키고자 하는 것이다. 두 개의 영역 크기 비율을 사용하는 이유는 REAR만 사용하는 경우는 형제 노드들이 다 같이 성장하는 경우에 그 효과를 발휘하지 못하고, AEAR만 사용하는 경우는 해당 노드가 형제 노드들에 비해 아주 큰 경우에는 그 효과를 발휘하지 못하기 때문이다.

분할 방법은 일반 노드인 경우에는 기존의 선형 알고리즘을 사용하여 노드를 분할한다. 그러나, 성장노드의 분할은 기존 선형 알고리즘과 동일한 방법으로 수행되지만 선형 알고리즘이 (노드 용량 + 1)의 크기를 가지고 분할을 하는 반면에, 성장 노드의 분할은 성장 노드의 크기, 즉 일반 노드보다 많은 수의 엔트리를 가지고 분할을 한다. 하나의 성장 노드를 두 개의 노드로 나눌 때, 두 개의 노드의 크기에 따라서 하나의 성장 노드는 두 개의 성장 노드와 성장 노드, 성장 노드와 일반 노드, 일반 노드와 일반 노드로 나누어 질 수 있다. 성장 노드 분할의 복잡도는 R-tree의 선형 알고리즘과 같은 $O(n)$임을 알 수 있다. 이러한 큰 영역을 가진 노드 분할 정책을 사용함으로써 영역 질의에 성능 향상을 가져 온다.

본 논문에서는 데이터 삽입 시 노드 오버플로우가 발생하면 노드 간의 중첩 및 사장 영역의 발생으로 검색 시 성능이 저하되는 것을 방지하기 위해서 합병 후 재분할 정책과 큰 영역을 가진 노드 분할 정책을 제시하고 있다. 또한, 해당 분할 정책의 수행 또는 성장 노드의 사용을 판단하기 위해서 EOR, REAR, AEAR과 같은 임계값을 사용

하고 있다. 각 임계값은 분할 시 상호 연관성을 가지고 있으며 노드 분할을 수행할 때 두 분할 정책 중 어떤 분할 정책을 먼저 사용하는지에 따라서 삽입 성능에 영향을 미칠 수 있다. 본 논문에서는 실험을 통하여 분할 정책의 순서와 각 임계값에 대한 다양한 변경값을 적용하여 알고리즘 1과 같이 큰 영역을 가진 노드 분할 정책을 먼저 수행한 후 합병 후 재분할 정책을 수행하고 각 임계값의 정의에 나타낸 수치를 적용하는 것이 가장 좋은 성능을 보여주는 것으로 확인하였다. 그러나, 최적의 노드 분할 조건을 선정하기 위해서는 분할 정책 적용 순서 및 각 임계값에 대한 비용 모델을 제시함으로써 실험 평가 결과에 대한 이론적인 검증이 차후 필요하다.

5. 성능 평가

이 장에서는 메인 메모리를 기반으로 하여 3DR-tree의 알고리즘을 사용한 색인(MM3DR-tree)과 이 논문에서 제시한 성장 노드 정책을 사용한 색인(GN3DR-tree)을 비교 실험한다. R*-tree의 경우 디스크 기반 환경에서는 R-tree 알고리즘보다 좋은 성능을 보여주고 있지만 알고리즘의 복잡도로 인하여 이 논문에서 가정하고 있는 높은 삽입 속도 조건을 만족시키지 못하므로 고려 대상에서 제외한다. 이동체 데이터베이스 환경에서는 삽입 연산이 색인에서의 주요한 연산 중의 하나이므로 삽입 성능과 검색 성능을 평가의 기준으로 한다. 실험을 위해서 Intel Pentium IV 1.7GHz, 512MB 메모리를 가진 컴퓨터를 사용하였으며, 운영체제로는 Windows XP를 사용하였다.

GN3DR-tree는 기존의 R-tree 색인에서의 삽입 및 노드 분할 정책이 메인 메모리 기반 이동체 데이터베이스 환경에 적용했을 때 빈번한 데이터 삽입으로 인한 상당한 노드 갱신 비용을 발생시키는 점을 해결하기 위해서 성장 노드 구조 및 분할 정책 알고리즘을 제시하고 있다. 따라서, 본 논문에서는 메인 메모리 환경에서 기존 R-tree 알고리즘과 제시된 방법의 비교 평가를 통하여 성능의 우수성을 입증하고자 한다. 메인 메모리 환경에서 검색 시 노드 접근 비용을 줄이기 위하여 본 논문에서 제안한 방법과 캐쉬 활용도를 높이기 위하여 기존에 연구된 CR-tree 등의 기법이 결합된다면 기존 R-tree 알고리즘을 적용한 색인보다 더욱 좋은 검색 성능을 보일 수 있을 것이다.

5.1 실험 환경

현재 이동체 데이터베이스 환경에서는 데이터 수집의 어려움으로 인해 실험에서 사용 가능한 실제 데이터가 존재하지 않는다. 그러므로 이 논문에서는 이동체 데이터베이스를 위한 실험 데이터셋으로 잘 알려진 GSTD 데이터셋[16]과 네트워크 기반 데이터셋[17]을 이용하여 실험을 수행하였다. 네트워크 데이터는 주어진 도로 네트워크를 따라 이동체가 이동하므로 GSTD 데이터보다는 좀더 현실적인 이동체의 움직임을 보이는 데이터셋이다.

먼저 GSTD 데이터는 표 4와 같이 데이터

<표 4> GSTD 데이터의 실험 요소

총 데이터 수(개)	이동체 수(개)	보고 회수(번)	비 고
10만	1,000	100	1,000개의 이동체가 100번 보고
50만	1,000	500	1,000개의 이동체가 500번 보고
100만	1,000	1,000	1,000개의 이동체가 1,000번 보고

<표 5> 네트워크 데이터의 실험 요소

초기 이동체(개)	시간 주기(회)	이동체 속도	비 고
1,500	100	Middle	한 시간 주기당 100개의 새로운 이동체 생성

를 생성하였다. 이동체 위치 데이터를 생성하기 위한 매개 변수로, 이동체의 초기 분포는 가우시안 분포이며 이동체의 이동은 랜덤이다. 표 4에서 볼 수 있듯이 이동체의 보고는 10만개, 50만개, 100만개의 데이터를 가지고 실험을 하였다.

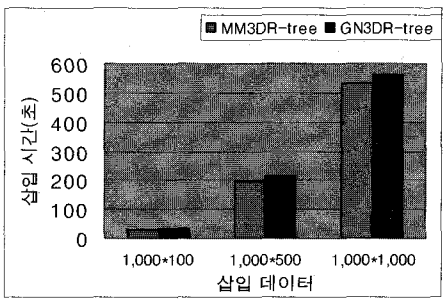
네트워크 데이터는 표 5와 같은 파라미터로 데이터를 생성하였다. 네트워크 데이터는 주어진 영역을 벗어나는 이동체는 보고가 끝나고, 시간 주기 당 새로운 이동체가 들어온다는 특성이 있다. 따라서, 더 이상 보고가 들어 오지 않는 데이터와 새로 생성되는 데이터로 인해 전체 삽입 데이터의 크기를 예상하기 어려운 점이 있다.

이 논문에서는 색인의 영역 질의에 대한 성능 평가를 위해서 각 시공간 영역에 대해서 1%, 5%, 10%, 20%의 크기를 갖는 질의 1,000개를 랜덤하게 생성하여 각 색인에 대해 사용하였다.

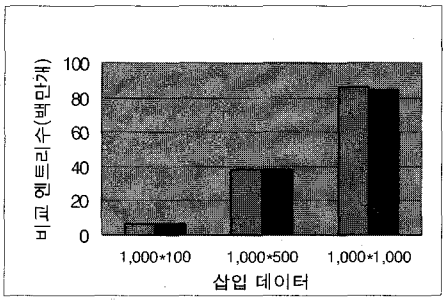
5.2 GSTD 데이터

5.2.1 삽입 성능

그림 9는 10만개(1,000 * 100), 50만개(1,000 * 500), 1,000만개(1,000 * 1,000)의 데이터를 가지고, GN3DR-tree와 MM3DR-tree를 구축 할 때 소요되는 삽입 시간 및 ChooseSubtree 함수 수행 시 비교 엔트리 수를 각각 측정한 것이다. (b)의 결과에서 보듯이 ChooseSubtree 함수 수행 시 GN3DR-tree가 MM3DR-tree보다 비교하는 엔트리 수가 2% 정도 감소하였다. 본 논문에서 제시하는 성장 노드 구조를 사용하는 GN3DR-tree는 성장 노드가 생성됨에 따라 비단말 노드의 엔트리 수는 증가되지 않으므로 분할 연기의 효과가 발생하며 결과적으로 기존의 R-tree 알고리즘보다 Choose Subtre 함수 수행 시 비교하는 엔트리 수가 감소하는 것을 알 수 있다. 그러나, 비교 엔트리 수가 감소함에도 (a)의 결과와 같이 GN3DR-tree의 삽입 시간이 MM3DR-tree에 비해서 3%에서 10% 정도 증가한 것을 알 수 있다. 이것은 성장 노드 정책을 사용하게 되면 분할 연기와 비단말 노드의 엔트리 수 감소로 인해 삽입 성능 향상의 효과가 있으나, 합병 후 재분할 정책 및 큰 영역을 가진 노드 분할 알고리즘으로 인해 비용이 추가 발생함으로 인하여 나타나는 결과이다.

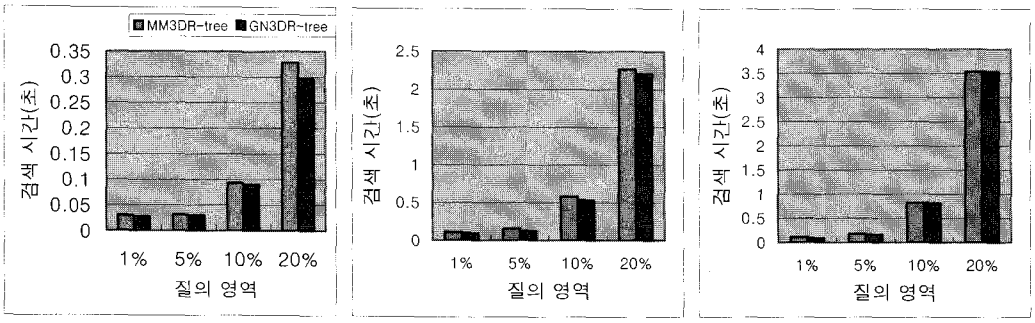


(a) 삽입 시간



(b) ChooseSubtree 수행 시 비교 엔트리 수

<그림 9> GSTD 데이터의 삽입 성능



(a) 10만개 검색시간 (b) 50만개 검색시간 (c) 100만개 검색시간

<그림 10> GSTD 데이터의 검색 시간

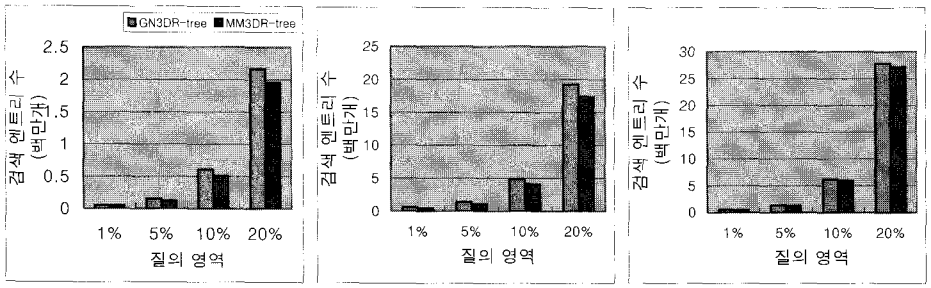
5.2.2 검색 성능

그림 10은 10만개, 50만개, 100만개를 삽입하여 구축한 색인에 대해서 1%, 5%, 10%, 20%의 영역을 가진 1,000개의 질의를 각각 수행한 후에 검색 시간을 측정한 것이다.

(a)는 10만개가 삽입된 색인의 검색 시간을 측정한 것이다. 각각의 영역 질의는 MM3DR-tree에 비해서 4%에서 10% 정도의 검색 성능의 향상을 가져 왔다. (b)는 50만개가 삽입된 색인의 검색 시간을 측정한 것이다. 각각의 영역 질의는 작게는 3%, 많게는 20% 정도의 검색 성능을 향상시켰다. (c)는 100만개가 삽입된 색인의 검색 시간을 측정한 것이다. 각각의 영역 질의는 1%에서 15% 정도의 검색 성능의 향상을 가져왔다.

그림 11은 10만개, 50만개, 100만개를 삽입하여 구축한 색인에 대해서 1%, 5%, 10%, 20%의 영역을 가진 1,000개의 질의를 각각 수행한 후에 영역 질의 시에 비교하는 엔트리 수를 측정한 것이다.

(a)는 10만개가 삽입된 색인에 영역 질의 시 비교 엔트리 수를 측정한 것이다. 각각의 영역 질의는 MM3DR-tree에 비해 10%에서 15%정도의 성능의 향상을 가져왔다. (b)는 50만개가 삽입된 색인의 영역 질의 시 검색 엔트리 수를 측정한 것으로 각각의 영역 질의에 대해서 10%에서 25% 정도의 성능 향상이 있음을 알 수 있다. (c)는 100만개가



(a) 10만개 검색 시 비교 엔트리 수 (b) 50만개 검색 시 비교 엔트리 수 (c) 100만개 검색 시 비교 엔트리 수

<그림 11> GSTD 데이터의 색인 검색 시 비교 엔트리 수

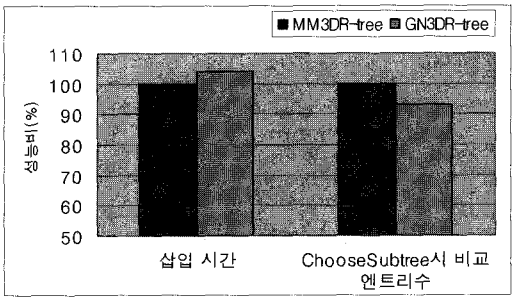
삽입된 색인의 영역 질의 시 검색 엔트리 수를 측정한 것으로 3%에서 7% 정도의 성능 향상을 보여준다.

실험 결과에서 알 수 있듯이 GN3DR-tree는 MM3DR-tree에 비해서 영역 질의 시 비교 엔트리 수는 약10%에서 25% 정도, 검색 시간은 약 1%에서 20%정도 단축됨으로 영역 질의에 대한 성능 향상을 가져왔다. 전체적으로 검색 시간이 단축되었다는 것은 제안한 성장 노드 구조와 그에 따른 합병 후 재분할, 큰 영역을 가진 노드 분할 정책이 색인의 검색 성능을 향상 시켰음을 알 수 있다. 질의 영역이 20%일 때 두 색인 모두 검색 시간이 급격히 증가하는 것은 검색해야 하는 영역의 범위가 상대적으로 크기 때문에 질의 결과에 포함되지 않지만 질의 영역에 겹치는 노드들의 수가 검색 범위가 작은 질의 영역보다 크게 늘어남으로써 전체적인 검색 성능이 떨어지기 때문이다.

5.3 네트워크 데이터

5.3.1 삽입 성능

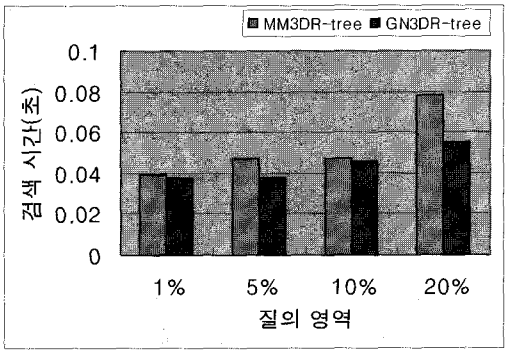
그림 12는 기존의 MM3DR-tree의 성능을 100%로 잡았을 때, GN3DR-tree의 데이터 삽입 시간 및 ChooseSubtree 함수 호출 시 비교하는 엔트리 수에 대한 성능 비를 나타낸 것이다. 그림에서 보는 바와 같이 GSTD 데이터 실험과 비슷한 결과가 나오는 것을 알 수 있다. GN3DR-tree가 성장 노드를 사용한 분할 연기 효과로 인해 ChooseSubtree 함수 호출 시 비교하는 엔트리의 수가 7% 정도 감소하였지만 이 논문에서 제안한 분할 정책의 알고리즘으로 인하여 전체 삽입 성능은 MM3DR-tree에 비해서 4% 정도 증가하였다.



<그림 12> 네트워크 데이터의 삽입 성능

5.3.2 검색 성능

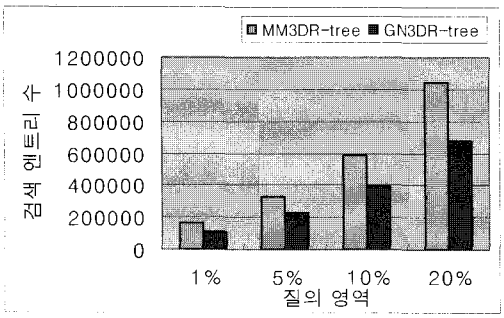
그림 13은 구축된 색인에 대해서 1%, 5%, 10%, 20%의 영역을 가진 1,000개의 질의를 각각 수행한 후에 검색 시간을 측정한 것이다. 그림 13에서 각각의 영역 질의는 MM3DR-tree에 비해서 3%에서 30% 정도 검색 시간이 단축됨으로써 검색 성능의 향상을 가져왔다. 5.2.2절에서 설명한 것과 같이 전체적으로 검색 시간이 단축되었다는 것은 제안한 성장 노드 구조와 그에 따른 합병 후 재분할, 큰 영역을 가진 노드 분할 정책이 색인의 검색 성능을 향상 시켰음을 알 수 있다.



<그림 13> 네트워크 데이터가 삽입된 색인의 검색 시간

그림 14는 그림 13과 동일한 환경에서 검색 시 비교하는 엔트리 수를 측정한 것이다. 그림

14에서 각각의 영역 질의에서 GN3DR-tree가 MM3DR-tree보다 비교 엔트리가 32%에서 35%정도 감소하는 것을 알 수 있다. 이것은 5.2.2 절에서도 설명하였듯이 성장 노드 구조로 인해 검색 시 비교해야 하는 비단말 노드가 감소하였으며, 이 논문에서 제시한 분할 정책의 사용으로 노드 간의 겹침이 감소함으로써 결과적으로 불필요한 노드의 접근을 줄였기 때문이다.



<그림 14> 네트워크 데이터가 삽입된 색인의 검색 시 비교 엔트리 수

실험 결과 GSTD 데이터와 네트워크 데이터에 대해서 비슷한 실험 결과가 나왔으며 MM3DR-tree에 비해서 GN3DR-tree의 검색 성능이 향상됨을 알 수 있었다. 네트워크 데이터의 경우 이동체가 정해진 도로를 따라서 이동하기 때문에 삽입되는 데이터 간의 중첩이 많이 발생함으로써 결과적으로 이 논문에서 제안한 성장 노드 구조에 의한 노드 분할 지연, 노드 간의 중첩 및 사장 영역을 줄이기 위한 분할 정책이 검색 성능 향상에 더 큰 영향을 준다. 따라서, GSTD 데이터를 이용한 실험 결과보다 성능 향상이 많은 것을 알 수 있다.

6. 결론 및 향후 연구

위치 기반 서비스에서는 빈번하게 보고되는 이동체의 위치에 대한 검색 성능을 보장하기 위해서는 이동체의 위치를 효율적으로 저장하고 처리해야 한다. 그러나, 기존의 시공간 색인은 디스크 기반의 색인이 대부분이므로 이동체의 수가 많아지거나 보고 회수가 많아지는 경우 이를 실시간으로 처리하기 어려운 문제가 있다. 따라서, 메인 메모리를 기반으로 효율적으로 이동체의 위치를 저장하기 위한 이동체 색인에 대한 연구가 필요하다.

이 논문에서는 이동체 데이터베이스 환경에서 이동체의 위치를 효율적으로 저장 및 검색하기 위해서 기존의 대표적인 이동체 색인인 R-tree를 기반으로 한 메인 메모리 색인 기법을 제안하고 있다. 제안한 색인 기법에서는 색인의 분할에 따른 처리 시간 복잡도가 높아지는 문제를 해결하기 위해서 성장 노드 구조를 제안하고 있으며 성장 노드 구조를 유지하기 위해서 합병 후 재분할(*MergeAndSplit*) 정책과 큰 영역을 가진 노드 분할(*LargeDomainNodeSplit*) 정책을 제안하였다. 단말 노드의 분할 시점을 결정하기 위해서 EOR(Excessive Overlap Ratio), REAR(Relatively Excessive Area Ratio), AEAR(Absolute Excessive Area Ratio) 값을 정의하였으며 실험 평가를 통해서 값들의 변화에 따른 색인 성능을 측정하여 성장 노드를 분할하기 위한 최적의 시점을 결정할 수 있었다. 실험 평가 결과 기존의 3DR-tree의 알고리즘을 메인 메모리에 적재했을 때보다 본 논문에서 제시한 성장 노드 구조 알고리즘을 적용한 색인이 영역 질의 수행 시 최대 30% 정도의 검색 시간의 향상이 있음을 알 수 있었다.

향후 연구로서 제안한 노드 분할 정책의 순서 및 성장 노드의 분할 시점을 결정하기 위한 EOR, READ, AREA 값에 대한 비용 모델을 제시함으로써 실험 평가 결과에 대한 이론적인 검증을 하는 것이 필요하다. 또한, 기존의 캐쉬 라인 활용도 향상을 위한

색인인 CR-tree와 본 논문에서 제시하는 색인 구조를 결합하여 다른 CR-tree 계열과의 성능에 대한 비교 평가가 필요하다.

참고문헌

1. O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. "Moving Objects Database: Issues and Solutions," Proc. of Int' l Conf. on Scientific and Statistical Database Management, 1998, pp. 111-122.
2. D. Pfoser, C. S. Jenson, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Objects," Proc. Of Int' l Conf. on Very Large Data Bases, 2000, pp. 395-406.
3. Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis, "Spatio-temporal Indexing for Large Multimedia Applications," IEEE Int' l Conf. on Multimedia Computing and Systems, 1996, pp. 441-448.
4. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," In Proc. of the ACM SIGMOD Int' l Conf. on Management of Data, 2000, pp.331-342.
5. A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. of the ACM SIGMOD Int' l Conf. on Management of Data, 1984, pp. 47-54.
6. N. Beckmann and H. P. Kriegel, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," Proc. of the ACM SIGMOD Int' l Conf. on Management of Data, 1990, pp. 332-331.
7. T. J. Lehman and M. J. Carey, "A Study of Index Structures for Main Memory Management Database Systems," Proc. of Int'l Conf. on Very Large DataBases, 1986, pp. 294-303.
8. J. Rao and K. A. Ross, "Making B+-trees Cache Conscious in Main Memory," Proc. of ACM SIGMOD Int' l Conf., 2000, pp. 475-486.
9. K. H. Kim, S. K. Cha, and K. J. Kwon, "Optimizing Multidimensional Index Trees for Main Memory Access," Proc. of ACM SIGMOD Int' l Conf., 2001, pp. 139-150.
10. 김기홍, 차상균, 권근주, "CR-tree: 캐쉬에 최적화된 R-tree," 한국정보과학회 데이터베이스 연구회(KDBC) 학술대회 논문집, 2001, pp. 7-12.
11. R. A. Hankins, and J. M. Patel, "Effect of Node Size on the Performance of Cache-Conscious B+-trees," Proc. of the 2003 ACM SIGMETRICS Int' l Conf. on Measurement and Modeling of Computer Systems, 2003, pp. 283-294.
12. S. Y. Hwang, K. J. Kwon, S. K. Cha, and B. S. Lee, "Performance Evaluation of Main-Memory R-tree Variants," Proc. of Int' l Symposium on Spatial and Temporal Databases, 2003, pp.10-27.
13. I. Kamel, and C. Faloutsos, "Hilbert R-tree: An Improved R-tree using Fractals," In Proc. of Int' l Conf. on Very Large DataBases, 1994, pp.500-509.
14. Y. J. Lee, and C. W. Chung, "The DR-tree: A Main Memory Data Structure for Complex Multi-dimensional Objects," GeoInformatica 5(2), 2001, pp.181-207.
15. H. Lu, Y. Y. Ng, and Z. Tian, "T-tree or

B-tree: Main Memory Database Index Structure Revisited," Australasian Database Conference, 2000, pp. 65-73.
16. Y. Theodoridis, J. R. O Silva, and M.A Nascimento, "On the Generation of Spatiotemporal Datasets," Proc. of Int' l Symposium on Spatial Databases, 1999, pp. 147-164.
17. T. Brinkhoff, "A Framework for Generating Network-Based Moving Objects," Proc. of the Int' l Conf. on Scientific and Statistical Database Management, 2000, pp.253-255.

안성우
1999년 부산대학교 컴퓨터공학과 졸업(공학사)
2001년 부산대학교 대학원 컴퓨터공학과 졸업(공학석사)
2001년~현재 부산대학교 대학원 컴퓨터공학과 박사과정
관심분야 : LBS, RFID 미들웨어, 모바일 GIS, 이동체 색인

안경환
1997년 부산대학교 컴퓨터공학과 졸업(공학사)
1999년 부산대학교 대학원 컴퓨터공학과 졸업(공학석사)
2004년 부산대학교 대학원 컴퓨터공학과 졸업(공학박사)
2004년~현재 한국전자통신연구원 LBS연구팀 연구원
관심분야 : LBS, 이동체 데이터베이스, 스트림데이터처리기

홍봉희
1982년 서울대학교 컴퓨터공학과 졸업(공학사)
1984년 서울대학교 대학원 컴퓨터공학과 졸업(공학석사)
1988년 서울대학교 대학원 컴퓨터공학과 졸업(공학박사)
1987년~현재 부산대학교 컴퓨터공학과 교수
관심분야 : 이동체 데이터베이스, 모바일 데이터베이스, 공간 데이터베이스, RFID 미들웨어

이창우
2002년 부산대학교 컴퓨터공학과 졸업(공학사)
2004년 부산대학교 대학원 컴퓨터공학과 졸업(공학석사)
2004년~현재 삼성전자 정보통신총괄 선임연구원
관심분야 : LBS, 이동체 데이터베이스