TPKDB 트리 : 이동 객체의 효과적인 미래 위치 검색을 위한 색인구조

(TPKDB-tree : An Index Structure for Efficient Retrieval of Future Positions of Moving Objects)

서 동 민 † 복 경 수 † 유 재 수 †† 이 병 엽 †††
(Dong Min Seo) (Kyoung Soo Bok) (Jae Soo Yoo) (Byoung Yup Lee)

요 약 최근 위치 기반 기술에 대한 활용이 증가하면서 이동 객체를 효율적으로 관리하기 위한 색인 구조의 필요성이 증가하고 있다. 본 논문에서는 미래 위치 검색 및 갱신 비용을 최소화하기 위한 새로운 시공간 색인 구조를 제안한다. 제안하는 색인 구조는 갱신 비용을 최소화하기 위해 이동 객체의 현재 위치를 직접 접근하기 위한 보조 색인 구조와 공간 분할 기반의 KDB-트리를 결합한 색인 구조이다. 제안하는 색인 구조에서는 이동 객체의 미래 위치 검색 및 갱신 비용을 줄이기 위해 KDB-트리의 중간 노드에 시간에 대한 파라미터를 유지한다. 또한 제안하는 색인 구조에서는 공간활용도 및 검색 효율을 극대화하기 위한 새로운 갱신 및 분할 기법을 제안한다. 제안하는 색인 구조의 우수성을 입증하기 위해 다양한 실험을 통해 성능 평가를 수행한다.

키워드 : KDB-트리, 이동 객체 색인, 미래 위치

Abstract Recently, with the rapid development of location-based techniques, index structures to efficiently manage moving objects have been required. In this paper, we propose a new spatio-temporal index structure that supports a future position retrieval and minimizes a update cost. The proposed index structure combines an assistant index structure that directly accesses current positions of moving objects with KDB-tree that is a space partitioning access method. The internal node in our proposed index structure keeps time parameters in order to support the future position retrieval and to minimize a update cost. Moreover, we propose new update and split methods to maximize the space utilization and the search performance. We perform various experiments to show that our proposed index structure outperforms the existing index structure.

Key words : KDB-tree, Moving Object Indexing, Future Position

1. 서 론

무선 통신 및 통신 기기의 발전과 더불어 위치 기반 서비스에 대한 많은 응용들이 개발되고 있다. 위치 기반 서비스(LBS : Location Based Services)는 핸드폰, PDA와 같은 무선 통신 기기를 통해 언제 어디에서나 서버에 연결된 객체들의 위치 정보를 활용하는 서비스이다. 위치 기반 서비스의 급속한 발전으로 시간의 변화에 따라 공간 위치를 변화하는 이동 객체에 대한 많은 관심이 집중되고 있다. 이와 함께 이동 객체를 효과적으로 저장하고 관리하기 위한 이동 객체 데이타베이스(MOD : Moving Object Databases)에 대한 많은 연구들이 진행되고 있다.

이동 객체 데이타베이스에서는 동적인 속성을 가지는 이동 객체에 대한 데이타 유형을 과거, 현재 그리고 미래 위치로 구분하여 표현한다. 대부분 이동 객체에 대한 연구들은 과거의 궤적 또는 현재 위치에 대한 데이타를 관리하는데 중점을 두고 많은 연구가 진행되었다. 최근 미래 위치에 대한 예측과 검색의 중요성이 증가됨에 따라 이동 객체의 현재 위치를 통해 미래 위치를 예측하

· 본 연구는 한국과학재단 목적기초연구(R01-2003-000-10627-0)지원으로 수행되었음
† 비 회 원 : 충북대학교 정보통신공학과
dmseo@netdb.chungbuk.ac.kr
ksbok@netdb.chungbuk.ac.kr
†† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@cbucc.chungbuk.ac.kr
††† 비 회 원 : 배재대학교 전자상거래학부 교수
bylee@pcu.ac.kr
논문접수 : 2003년 12월 23일
심사완료 : 2004년 9월 14일

기 위한 연구들이 진행 중이다[1].

지속적으로 움직이는 이동 객체의 위치를 추적하고 저장하기 위해서는 이동 객체의 연속적인 변화를 효율적으로 관리하기 위한 새로운 기술의 색인 구조가 요구된다. 기존의 색인 구조는 특정 시점의 객체 위치를 관리하기 때문에 이동 객체의 위치 정보와 같이 동적인 속성을 가지는 정보를 표현, 저장 그리고 질의하는데 있어서 비효율적이다. 기존에 제안된 R-트리[2] 기반의 공간 색인 구조는 지속적으로 변경되는 이동 객체의 위치 정보를 색인하기 위해서 많은 갱신 비용을 요구하고 이로 인해 성능을 크게 감소시키는 문제점을 가진다.

이러한 문제점을 해결하기 위해 새로운 시공간 색인 구조들이 제안되었다. LUR-트리[3]에서는 이동 객체의 현재 위치를 색인할 때 발생하는 갱신 비용을 감소시키기 위한 기법을 제안하였다. 그러나 현재 위치에 대해서만 색인을 구성하기 때문에 미래 위치 검색을 제공하지 못하고 이동 객체의 동적인 변화나 새로운 객체의 삽입이나 삭제가 발생할 때 전체 색인 구조를 순회하면서 갱신 작업을 수행하기 때문에 많은 갱신 비용을 필요로 할 뿐만 아니라 이로 인해 검색 성능을 저하시키는 문제점이 있다. VCI-트리[4]는 다수의 연속적인 범위 질의 및 미래 위치 검색을 처리하기 위해 기존 R-트리의 각 노드에 v_{max} 라는 이동 객체의 최대 속도 값을 유지하는 색인 구조를 제안하였다. TPR-트리[5]는 이동 객체의 미래 위치에 대한 질의를 효율적으로 처리하기 위해 시간에 대한 MBR과 속도 벡터를 사용함으로서 이동 객체를 주기적으로 갱신하는데 드는 높은 비용을 줄이는 색인 구조를 제안하였다. 그러나 이러한 색인 구조들 또한 객체의 동적 변화에 따라 많은 갱신 비용을 소요할 뿐만 아니라 이로 인해 검색 성능을 저하시키는 문제점이 있다.

본 논문에서는 이동 객체의 현재 및 미래 위치 검색을 효율적으로 수행하기 위한 TPKDB-트리(Time Parameterized KDB-tree)라는 새로운 색인 구조를 제안한다. 제안하는 색인 구조는 R-트리의 빈번한 갱신 비용을 최소화하기 위해 KDB-트리[6]와 이동 객체의 현재 위치를 직접 연결할 수 있는 보조 인덱스로 구성된다. 제안하는 색인 구조는 이동 객체의 미래 위치를 효과적으로 검색하기 위해 이동 객체의 현재 위치와 속도 정보를 노드에 표현한다. 그리고 미래 위치를 예측할 때 발생하는 불필요한 영역의 확장을 최소화하기 위해 노드에 존재하는 객체들이 특정 영역을 벗어나는 시간 정보를 함께 표현한다. 또한, 공간활용도 및 검색 성능을 향상시키기 위한 새로운 분할 기법을 제안한다.

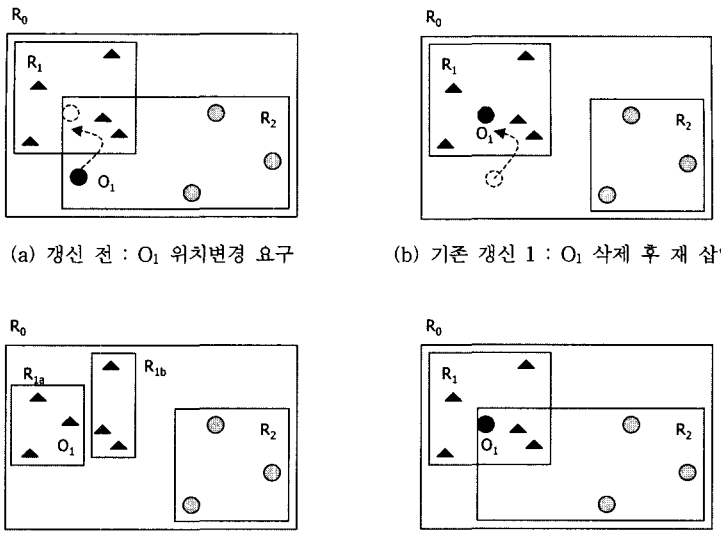
본 논문의 구성은 다음과 같다. 2장에서는 이동 객체를 위해 기존에 제안된 색인 기법들에 대해 서술한다. 3장에서는 제안하는 TPKDB-트리의 구조 및 갱신 기법을 기술하고 4장에서는 현재와 미래에 대한 검색 기법에 대해 기술한다. 5장에서는 다양한 실험을 통해 기존에 제안된 색인 구조와의 성능 평가를 수행하고 마지막 6장에서는 결론 및 향후 연구 방향에 대해 기술한다.

2. 관련 연구

기존의 공간 색인 기법에서는 위치가 고정된 객체의 위치를 저장하고 관리하기 위한 많은 연구들이 진행되었다. 그러나 최근에는 이동 객체를 효율적으로 저장하고 관리하기 위한 많은 연구들이 진행되고 있다. 이 장에서는 최근에 이동 객체의 미래 위치 검색을 지원하거나 갱신 비용을 줄이기 위해 제안된 LUR 트리[3], Q+R-트리[7], VCI-트리[4] 그리고 TPR-트리[5]와 본 논문에서 제안하는 색인 구조의 기본 색인 구조인 KDB-트리[6]에 대해 기술한다.

LUR-트리는 지속적으로 움직이는 이동 객체의 현재 위치를 색인할 경우 발생하는 갱신 비용을 줄이기 위해 제안된 색인 구조이다[3]. LUR-트리에서는 이동 객체의 위치가 갱신되는 동안 불필요한 노드의 분할과 합병을 줄이기 위해서 갱신되는 객체의 정보가 동일 노드 안에서의 변경이면 단말 노드에서 갱신된 정보만을 변경한다. 그림 1은 기존 R-트리와 LUR-트리의 갱신에 대한 예를 보여주고 있다. 그림 1의 (a), (b) 그리고 (c)는 노드가 수용할 수 있는 최대 엔트리 수가 5인 R-트리에서 노드 R_2의 객체 O_1이 갱신되는 것을 보여준다. 기존 R-트리에서는 객체의 갱신은 해당 객체의 이전 위치를 삭제하고 색인 구조 전체를 순회하면서 새로운 위치를 삽입하는 과정을 수행한다. 이러한 갱신 과정은 갱신을 요청한 객체를 포함하고 있던 기존 노드의 영역과 새로 삽입될 노드에 대한 최소 경계 영역을 만들기 위해 각 노드에 대한 MBR 변경 비용을 요구한다. 이런 문제를 해결하기 위해 LUR-트리의 지연 갱신 방법이 제안되었다. 그림 1의 (d)는 지연 갱신의 예를 보여주고 있다. 지연 갱신 방법은 갱신되는 객체의 정보가 동일 노드의 MBR 내에서의 갱신인 경우 갱신 객체의 정보만을 변경함으로서 갱신 객체의 삭제와 재삽입 그리고 MBR의 변경 연산을 줄임으로서 기존 R-트리의 갱신 비용을 감소시킨다. 그러나 새로 삽입된 객체의 위치가 기존 영역 내에 포함되지 않을 경우 MBR 영역을 확장하기 위해 많은 갱신 비용을 발생하며 이동 객체의 미래 위치 검색을 색인하지 못하는 문제점이 있다.

Q+R-트리는 이동 객체를 색인하는 기존의 R-트리 기반의 색인 구조가 가지는 많은 갱신 비용을 줄이기 위해 R*-트리와 사분 트리(Quad-tree)로 구성된 혼합형 구조이다[7]. Q+R-트리는 이동 객체를 이동 패턴과 지



(a) 갱신 전 : O_1 위치변경 요구 (b) 기존 갱신 1 : O_1 삭제 후 재 삽입

(c) 기존 갱신 2 : 오버플로우 노드 분할 (d) 지연 갱신

그림 1 LUR-트리 갱신

리적 특성에 따라 동적 이동 객체(Fast-moving object)와 정적 이동 객체(Quasi-static object)로 구분하여 색인하는 것이다. 동적 이동 객체는 GPS를 장착한 자동차와 같이 빠른 속도를 가지고 움직이며, 지속적으로 위치의 변경을 요구하는 특성을 가진 이동 객체를 의미한다. 정적 이동 객체는 장시간 일정 영역 공원이나 사무실에서와 같이 제한된 영역에서 느린 속도를 가지고 움직이는 특성을 가진 이동 객체를 의미한다. 갱신 비용을 많이 필요로 하는 동적 객체는 R-트리 기반의 색인 구조에 비해 갱신 비용이 적은 사분 트리에 색인하고 갱신 비용을 많이 필요로 하지 않는 정적 객체는 검색 성능을 향상시키기 위해서 R*-트리에 색인한다. 이와 같은 방법을 사용하여 갱신과 검색 성능을 향상시켜 색인 구조 전체의 성능을 향상시켰다.

R*-트리의 중간 노드는 정적 이동 객체를 표현하기 위해 사전에 조사된 지리적 특성을 기반으로 구성된다. Q+R-트리에 구성되는 중간 노드는 지리적 특성에 의해 구성되므로 겹침이 발생하지 않는 특성을 가진다. R*-트리의 단말 노드는 많은 양의 이동 객체를 색인할 수 있도록 동적 배열을 사용하는 특성을 가진다. 이로 인해 많은 양의 정적 이동 객체를 색인 구조의 변경 없이 색인하여 갱신 비용을 줄인다. 사분 트리를 구성하는 방법은 기존 사분 트리에 객체를 삽입하는 방법과 동일하다. 그러나 색인 구조의 갱신 성능을 향상시키기 위해서 R*-트리의 중간 노드와 겹침이 발생하는 사분 트리의 단말 노드에 대해 포인터를 연결한다.

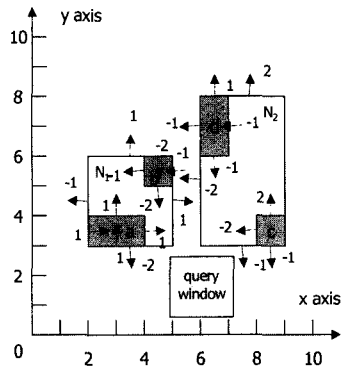
그러나 사분 트리의 특성으로 인해 색인되는 객체가 사분 트리에 편향될 경우 색인 구조의 크기가 커지고 불균형 트리가 되어 검색 성능이 저하되는 문제를 가진다. 또한 이동 객체의 다양한 동적 변화가 있을 때 사분 트리와 R*-트리의 갱신 비용을 증가시키는 문제를 가진다.

VCI-트리는 R-트리 기반의 색인 구조에 갱신 비용을 줄이고 미래 위치 검색을 지원하기 위해 제안된 색인 구조이다[4]. VCI-트리는 이동 객체를 색인하는 색인 구조에서 다수의 연속적인 범위 질의를 효율적으로 처리하기 위해 질의 인덱스와 함께 제안되었다. VCI-트리의 각 노드는 노드 내에 포함되어 있는 이동 객체들 중 가장 큰 속도 값을 취하는 v_{max} 필드를 유지한다. 각 노드의 v_{max} 필드는 노드에 포함되는 객체의 갱신이 발생하였을 때, 해당 노드의 MBR을 변경하지 않고 갱신된 객체의 정보만을 변경할 수 있도록 하여 이동 객체를 주기적으로 갱신하는데 드는 높은 비용과 많은 갱신 비용으로 검색 성능이 저하되는 기존 색인 구조의 문제를 해결하였다. 그리고 이동 객체의 미래 위치 검색 시 각 노드의 v_{max} 을 이용하여 질의 시간까지 각 노드를 확장하여 질의 영역과 겹침이 발생하는 노드에 대한 검색을 수행함으로서 이동 객체에 대한 미래 위치 검색을 제공한다. 그러나 이동 객체의 동적인 위치 변화를 갱신하기 전까지는 이동 객체에 대한 질의를 기존에 구성된 색인 구조를 사용하여 처리하기 때문에 정확한 검색 결과를 얻을 수 없는 문제를 가진다. 또한 이 문제를

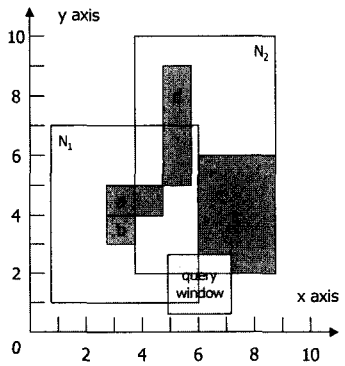
해결하기 위해 주기적으로 색인 구조를 재구성해야 한다.

TPR-트리는 VCI-트리와 유사한 방법으로 갱신 비용을 줄임으로서 이동 객체를 효율적으로 색인하고 미래 위치 검색을 지원하는 색인 구조이다[5]. TPR-트리는 R-트리를 기반으로 하고 이동 객체는 색인 구조가 구성된 시간에 대한 MBR과 속도 벡터로 표현한다. 그림 2의 (a)는 이동 객체 a, b, c 그리고 d를 포함하는 각각의 MBR을 나타내고 각 노드는 자신의 MBR에 위치하는 모든 이동 객체를 포함한다. 각각의 MBR과 함께 표시되는 숫자는 이동 객체들의 속도를 나타낸다. 음의 값을 가지는 속도는 색인 구조에 대응되는 좌표축에 대해, 절대 속도 값을 가지고 반대 방향으로 이동하는 것을 나타낸다. 각 노드 MBR의 에지(edge)는 노드에 포함되어 있는 이동 객체에 의해 각기 다른 속도를 가질 수 있다. 속도는 MBR의 에지 방향에 대응하여 이동하는 이동 객체들 중에서 가장 큰 속도 값을 취한다. 중간 노드에는 하위 노드를 포함하는 MBR과 속도 벡터로 나타낸다. 중간 노드의 MBR은 기존 R-트리와 동일하게 MBR이 구성될 때 MBR 안에 포함되어 있는 모든 자식 노드들을 포함하며 최소 경계 영역을 가진다. 그림 2의 N_1과 N_2는 중간 노드의 MBR을 나타낸다. 중간 노드의 위쪽과 오른쪽 에지는 각 에지에 대응되는 자식 노드의 가장 빠른 속도를 취하고 중간 노드의 아래쪽과 왼쪽 에지는 가장 작은 속도를 취한다. 그러나 중간 노드의 MBR은 노드에 포함되어 있는 이동 객체들의 속도에 종속되어 변경되기 때문에 최소 경계 영역의 MBR을 보장하지는 않는다. 이와 같은 문제로 색인 구조가 재구성되기 전까지 색인 구조의 정확성을 보장하지 못하는 문제가 발생한다. TPR-트리는 이동 객체의 미래 위치에 대한 질의가 가능하다. 그림 2의 (a)는 시간이 0일 때, 구성된 색인 구조에 시간 1에 대한 질의가 요청된 것을 보여주고 있다. 그림 2의 (b)는 미래 위치 질의를 처리하기 위해서 모든 노드의 MBR을 시간 1에 대해 각 노드의 에지가 가지는 속도 값만큼 확장한 것을 나타낸다. 그리고 확장된 MBR과 질의 영역의 교차되는 모든 MBR 안에 있는 이동 객체를 조사함으로서 질의를 처리할 수 있다. 그러나 R-트리를 기반으로 하기 때문에 공간 색인의 문제점인 갱신 비용 문제와 노드의 MBR을 시간의 함수로 표현함으로서 발생하는 노드들 간의 겹침으로 검색 시간이 저하되는 문제가 있다.

KDB-트리(K Dimensional Balanced tree)는 범위 질의를 통해 다중 키 레코드를 효율적으로 검색하기 위한 기법으로서 B-트리와 KD-트리의 특징을 결합한 것이다[6]. KDB-트리는 고정 크기의 노드들로 구성된 다원 트리로서 B-트리와 같이 루트 노드로부터 각 단말



(a) t=0 일 때, 노드구조 예



(b) t=1 일 때, 노드구조 예

그림 2 TPR-트리의 엔트리

노드까지의 경로 길이가 모두 같은 완전 균형 트리이다. KDB-트리에서 각 노드는 페이지에 저장되는데 이는 페이징 기법을 이용하여 보조 기억 장치를 효율적으로 사용하기 위한 것이다. 노드 구조는 단말 노드와 중간 노드로 구성된다. 점 페이지라고 불리는 KDB-트리의 단말 노드는 영역 안의 점 데이타를 포함한다. 영역 페이지라고 불리는 KDB-트리의 중간 노드는 <MBR descriptor, node pointer>의 쌍으로 표현되고, node pointer는 자식 노드를 가리킨다. KDB-트리도 KD-트리와 마찬가지로 탐색 공간을 분할한다. 즉, K 차원의 영역은 K-1 차원의 분할 요소들에 의해 겹치지 않는 사각형으로 나뉜다. KDB-트리는 데이타의 분포를 고려하여 분할하고 개념적으로 간단한 검색 스키마를 가지고 대용량 공간 데이타베이스를 위해 설계되었기 때문에 다차원 공간 색인에 있어서 성능이 좋은 것으로 연구된바 있다[8]. 그러나 KDB-트리는 공간 분할 방식이기 때문에 시공간에서 이동 객체의 위치데이타가 계속적으로 입력될 경우 시간 축의 특성상 분할 문제로 인해 공간활용도가 낮은 단말 노드와 재분할을 요구하는

중간 노드를 생성하는 문제가 발생한다.

3. TPKDB-트리 : 제안하는 색인구조

제안하는 TPKDB-트리는 갱신 비용을 최소화하면서도 이동 객체의 미래 위치를 효과적으로 검색하기 위한 색인 구조이다. TPKDB-트리에서는 그림 3과 같이 KDB-트리와 보조 색인 구조의 혼합형 색인 구조로 구성되어 있다. 제안하는 색인 구조는 R-트리 계열의 색인 구조가 가지는 문제점인 데이타의 삽입과 삭제로 발생하는 빈번한 색인 구조의 변경을 피하고 노드들의 겹침으로 인한 검색 성능 저하를 개선하기 위해 KDB-트리를 기반으로 한다. 제안하는 색인 구조에서 사용하는 보조 색인 구조는 이동 객체의 식별자와 이동 객체의 정보를 직접 접근할 수 있도록 이동 객체를 포함하는 KDB-트리의 단말 노드에 대한 정보를 포함하고 있다. 이를 통해 색인 구조 전체를 순회하지 않고 객체의 정보를 직접 접근하여 객체의 갱신을 수행할 수 있도록 한다.

TPKDB-트리에 삽입되는 이동 객체는 $\langle oid, t_{ref}, p_{ref}, v_{ref}\rangle$ 와 같이 표현되고 각각의 의미는 다음과 같다.

- oid : 이동 객체에 대한 식별자
- t_{ref} : 이동 객체의 정보가 삽입 또는 갱신된 시간
- p_{ref} : t_{ref} 시간에서 객체의 위치로 각 차원에 대한 위치 정보 $(p_1, p_2, \ldots, p_n)$.
- v_{ref} : t_{ref} 시간에서 객체의 속도로 각 차원에 대한 속도 $(v_1, v_2, \ldots v_n)$

미래 시점 t에서 객체의 위치는 식 (1)과 같이 t_{ref} 시점의 객체 위치와 속도를 통해 계산한다.

$$p_t = p_{ref} + v_{ref}(t - t_{ref}) \quad (1)$$

TPKDB-트리의 단말 노드 구조는 그림 4와 같이 헤더(Header)와 실제 이동 객체의 위치 정보를 기록하는 엔트리로 구성되어 있다. 단말 노드에 존재하는 헤더는 보조 색인 구조를 통해 객체의 현재 위치에 대한 갱신이 필요한지를 부모 노드에 접근하지 않고 계산할 수 있도록 하기 위해 존재한다. 즉, 단말 노드에 존재하는 객체의 위치에 대한 갱신이 부모 노드에 반영되어야 할지를 판단하기 위해 필요한 정보를 기록한다. 헤더에 존재하는 RS는 단말 노드에 존재하는 객체를 포함하는 공간 영역 정보 $(RS_1, RS_2, \ldots, RS_n)$ 이다. 이때, $RS_i = [p_i^-, p_i^+]$ 로 p_i^- 와 p_i^+ 는 i번째 차원의 시작 위치와 끝 위치를 나타낸다. t_{upd}는 단말 노드의 갱신 시간으로 단말 노드 내에 존재하는 객체 중에서 가장 오래된 갱신 시간을 기록한다. t_{upd}을 가장 오래된 시간으로 기록하는 것은 미래 위치 검색 과정에서 확장되는 노드 안에 포함되어 있는 모든 객체의 정확한 미래 위치 정보를 획득하기 위해서이다. $v_{\max}$ 는 단말 노드에 존재하는 객체의 속도 값 $(v_{\max_1}, v_{\max_2}, \ldots v_{\max_n})$로 각 축의 양 방향의 최대 속도 값을 나타낸다. 이때, $v_{\max_i}$는 $[v_i^-, v_i^+]$로 v_i^- 와 v_i^+는 i번째 차원의 왼쪽 및 오른쪽 방향으로 이동하는 객체의 최대 속도를 나타낸다. 또한 t_{esc}는 단말 노드에 존재하는 객체들이 RS 영역을 벗어나는 최소 시간으로 미래 위치 검색 과정에서 불필요한 영역 확장을 없애기 위해 사용한다. KDB-트리 기반의 색인 구조는 영역을 분할하여 색인을 구성하기 때문에 사장 공간(dead space)이 존재한다. 한 사장 공간 내에서 이동하는 객체에 대한 미래 위치를 검색하는데 불필요한 영역 확장을 없애기 위해 t_{esc}을 사용한다.

만약 미래 위치 검색이 t_{upd}을 기준으로 t_{esc} 시간 내에서 수행된다면 TPR-트리나 VCI-트리처럼 영역을 확장하는 것이 아니라 현재 영역을 미래 위치까지 확장된 영역으로 판별하여 검색을 수행한다. 이동 객체의 실제 위치를 나타내는 단말 노드 엔트리는 $\langle oid, t_{ref}, p_{ref}, v_{ref}\rangle$ 이다. $oid, t_{ref}, p_{ref}, v_{ref}$는 삽입된 객체의 정보와 동일하다.

그림 5는 단말 노드에 존재하는 이동 객체와 헤더 정보를 나타낸 것이다. 그림 5에서 N_1은 이동 객체를 포함하는 TPKDB-트리의 단말 노드를 나타내며 O_1, O_2,

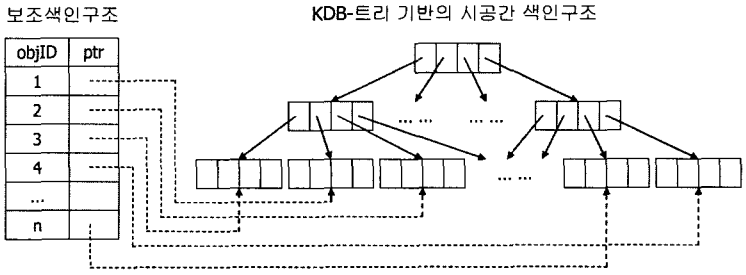


그림 3 TPKDB-트리의 구조

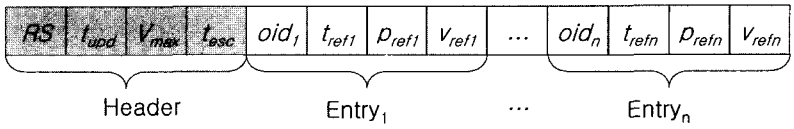


그림 4 단말 노드 구조

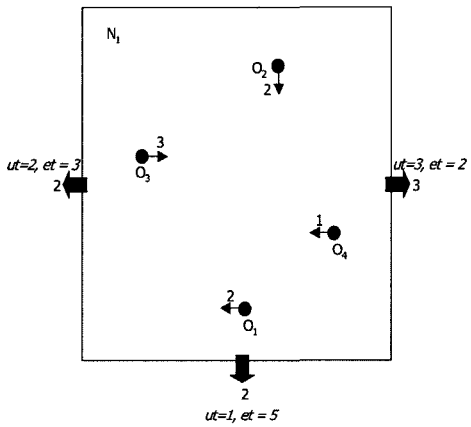


그림 5 TPKDB-트리의 단말 노드

O_3, O_4는 객체를 나타낸다. 점과 함께 표현되는 화살표는 이동 객체의 이동 방향을 나타내며 화살표 옆에 표현되는 숫자는 이동 객체의 속도를 나타낸다. 예를 들어, 이동 객체 O_3은 속도 3을 가지고 동쪽으로 이동하고 있는 것을 나타낸다.

TPKDB-트리의 각 노드들은 미래 위치 검색을 지원하고 검색 비용을 최소화하기 위해서 영역 내에 있는 이동 객체나 노드들과의 관계를 시간에 대한 파라미터로 유지한다. 단말 노드의 헤더 정보는 객체의 삽입, 삭제 그리고 노드 안에 있는 이동 객체들의 속도와 방향의 변경으로 변경될 수 있다. N_1은 이동 객체 O_1, O_2, O_3, O_4를 포함하고 있는 단말 노드를 나타내고 단말 노드 N_1에서 서쪽으로 향하는 화살표는 N_1 노드 안에 서쪽으로 향하는 이동 객체가 하나 이상 있다는 것을 의미한다. 화살표와 함께 표현되는 숫자는 동일한 단말 노드 안에 있으면서 화살표와 동일한 방향으로 이동하는 객체들 중에서 가장 빠르게 움직이는 이동 객체의 속도를 의미한다. 즉, N_1에는 서쪽으로 속도 2를 가지고 이동하는 이동 객체 O_1과 속도 1을 가지고 이동하는 이동 객체 O_4가 있다. 그러므로 N_1의 헤더 정보 중에서 서쪽으로 향하는 이동 객체에 대한 v_{max} 의 값은 2가 된다. 그리고 화살표와 함께 표현되는 ut와 et는 화살표와 동일한 방향으로 움직이는 객체에 대해 단말 노드가 가지는 t_{upd} 와 t_{esc} 을 의미한다. t_{upd} 와 t_{esc} 는 이동 객체들이 가지는 t_{upd} 와 t_{esc} 중 가장 작은 값을 취한다. 예를 들어, 이동 객체 O_1이 단말 노드 N_1의 영역을 벗어나는데 단위 시간 3이 요구되고 O_4는 단위 시간 8이 요구되면 서쪽으로 향하는 객체에 대한 N_1의 t_{esc} 는 3의 값을 취한다. 즉, N_1에서 서쪽으로 향하는 객체에 대한 단말 노드의 $up(t_{upd})$=2와 $et(t_{esc})$=3은 미래 질의 처리를 위해 단말 노드 확장시 N_1은 질의 시간 5까지는 서쪽으로 노드 확장이 불필요하다는 것을 나타낸다.

TPKDB-트리의 중간 노드 구조는 그림 6과 같이 헤더(Header)와 중간 노드에 포함되는 단말 노드의 헤더 정보를 기록하는 엔트리로 구성되어 있다. 중간 노드의 헤더 정보는 단말 노드의 헤더 정보와 같은 의미를 가진다. RS는 중간 노드에 존재하는 단말 노드를 포함하는 공간 영역 정보를 나타내고 t_{upd}는 중간 노드의 갱신 시간으로 중간 노드 내에 존재하는 단말 노드의 헤더 정보 중에서 가장 과거의 갱신 시간을 기록한다. 그리고 v_{max} 는 중간 노드에 존재하는 단말 노드의 최대 속도 값을 기록하고 t_{esc} 는 중간 노드에 존재하는 단말 노드들이 RS 영역을 벗어나는 최소 시간을 기록한다. 단지, 단말 노드는 객체의 실제 값을 이용하여 헤더를 구성하지만 중간 노드는 하위 노드들에 존재하는 헤더 정보를 이용하여 헤더 정보를 구성한다. 그리고 중간 노드에 존재하는 엔트리는 하위 노드에 존재하는 자식 노드의 헤더를 이용하여 구성된다. 또한, 엔트리의 ptr_i은 단말 노드 N_i가 저장되어 있는 곳의 포인터를 나타낸다.

그림 7은 제안하는 색인 구조의 중간 노드 구조를 나타낸 것이다. 그림 7의 중간 노드는 단말 노드 N_1, N_2

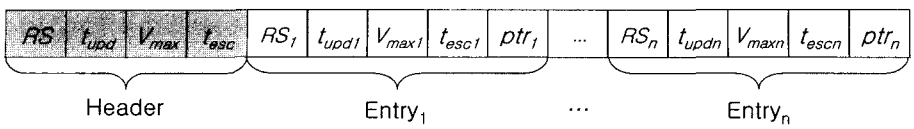


그림 6 중간 노드 구조

그리고 N_3을 포함한다. 단말 노드 N_1, N_2 그리고 N_3은 각각 해당 노드에 포함되어 있는 이동 객체들에 의해 단말 노드의 헤더 정보인 $\langle RS, t_{upd}, v_{max}, t_{esc}\rangle$ 가 구성된다. 그리고 중간 노드의 헤더 정보인 $\langle RS, t_{upd}, v_{max}, t_{esc}\rangle$ 는 중간 노드에 포함되어 있는 단말 노드의 $\langle RS, t_{upd}, v_{max}, t_{esc}\rangle$ 에 의해 구성된다. 예를 들어, 단말 노드 N_2와 N_3가 각각 동쪽으로 향하는 객체에 대해 2와 3의 v_{max} 을 가진다. 그러므로 중간 노드의 헤더 정보 중에서 동쪽으로 향하는 객체에 대한 v_{max} 는 가장 빠른 속도 3을 취한다. 그리고 t_{upd}, t_{esc} 는 가장 작은 값을 취하기 때문에 $up(t_{upd})$=3과 $et(t_{esc})$=2를 취하게 된다. 즉, 중간 노드에서 동쪽으로 향하는 객체에 대한 중간 노드의 $up(t_{upd})$=3과 $et(t_{esc})$=2는 미래 질의 처리를 위해 중간 노드 확장시 중간 노드는 질의 시간 5까지는 동쪽으로 노드 확장이 불필요하다는 것을 나타낸다.

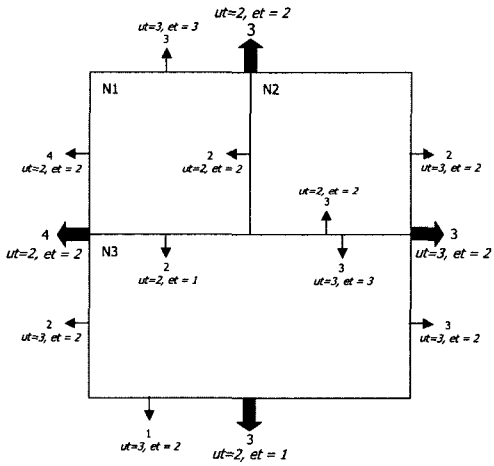


그림 7 TPKDB-트리의 중간 노드

3.1 삽입 알고리즘

TPKDB-트리에서의 삽입·삭제 알고리즘은 KDB-트리의 삽입, 삭제 알고리즘과 유사하다. 그러나 TPKDB-트리는 이동 객체의 갱신 처리를 위해 수반되는 삽입, 삭제 연산시 보조 색인 구조를 통한 빠른 객체 검색을 지원한다. 그리고 갱신 전과 갱신 후의 이동 객체의 위치가 동일 노드이고 이동 객체의 궤적을 구성하는 파라미터들의 변경이 없는 경우 갱신을 피하는 방법을 지원한다. 이와 같이 갱신 성능을 향상시키는 방법들이 사용되므로 이와 관련된 연산이 기존 알고리즘에 추가되었다.

이동 객체에 대한 삽입은 보조 색인 구조와 KDB-트리 기반의 시공간 색인 구조에 대한 삽입 연산을 수반한다. 아래 Insert 알고리즘은 TPKDB-트리의 삽입 알고리즘을 나타낸다. 먼저, 삽입하려는 객체의 정보가 기록된 mObj라는 구조체를 통해 객체의 삽입 연산이 요청되면 SearchBucket( )를 호출해서 삽입 연산을 요청한 이동 객체의 *OID*를 갖는 버켓을 보조 색인 구조에서 찾는다. SearchBucket( )를 통해 보조 색인 구조의 버켓이 획득되면 삽입을 요청한 이동 객체는 기존 정보의 갱신을 요청하는 것이므로 객체의 갱신된 정보를 반영한다. 반면, 보조 색인 구조의 버켓이 획득되지 않으면 이동 객체의 정보가 처음 삽입되는 것이므로 삽입하려는 *OID*가 선택된 버켓에 이동 객체의 *OID*를 삽입한다. KDB-트리 기반의 시공간 색인 구조에 이동 객체를 삽입 후, LinkLeaf( )를 통해 이동 객체가 삽입된 단말 노드의 포인터를 버켓에 저장한다.

갱신을 요청한 이동 객체에 대한 보조 색인 구조의 버켓이 선택되면 FindLeafNode( )를 통해 *OID*와 연결된 KDB-트리 기반의 시공간 색인 구조의 단말 노드를 검색한다. 여기서, 버켓에 저장된 단말 노드에 대한 포인터를 이용함으로서 루트 노드부터 검색하는 방법을 피하고 보조 색인 구조를 통해 직접 단말 노드를 접근하는 방법을 사용하여 단말 노드 검색 시간을 줄인다.

KDB-트리 기반의 시공간 색인 구조의 삽입 연산은 기존 KDB-트리의 삽입 연산과 유사하다. 먼저, 검색 연산을 통해서 삽입 연산을 요청한 이동 객체가 삽입될 단말 노드를 찾는다. 이동 객체가 삽입될 단말 노드를 선택한 후, 새로 삽입될 이동 객체에 의해 선택된 단말 노드가 분할이 발생하는지 검사한다. 만약, 분할이 발생하지 않는다면 선택된 단말 노드에 삽입하려는 이동 객체를 삽입하고 삽입된 이동 객체로 인한 단말 노드의 t_{upd}, t_{esc} 그리고 v_{max}가 변경되는지 검사한다. 이동 객체 삽입 후 단말 노드의 t_{upd}, t_{esc} 그리고 v_{max}의 변경이 발생하면 변경된 값을 단말 노드의 헤더에 반영해 주고 이 변경으로 인한 해당 단말 노드의 부모 노드 변경 사항을 검사 후 반영한다. 이와 같은 노드 정보의 변경은 루트 노드까지 반영될 수 있다. 단말 노드에 이동 객체를 삽입 후 오버플로우가 발생한다면 TPKDB-트리의 분할 연산인 Split( )를 통해서 오버플로우가 발생한 단말 노드를 분할한다. 그리고 분할로 인한 t_{upd}, t_{esc} 그리고 v_{max}의 변경을 요청하는 단말 노드의 변경 사항을 반영해 주고 이 변경으로 인한 해당 단말 노드의 부모 노드 변경 사항을 검사 후 반영한다. 이와 같은 노드 정보의 변경 또한 루트 노드까지 반영될 수 있다.

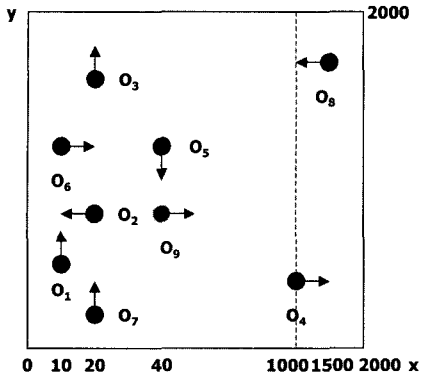
3.2 분할 알고리즘

시공간 색인으로 KDB-트리를 이용할 경우 KDB-트리가 공간분할 방식이기 때문에 계속적인 데이타의 삽입으로 인해 단말 노드에 오버플로우가 발생하고 이로

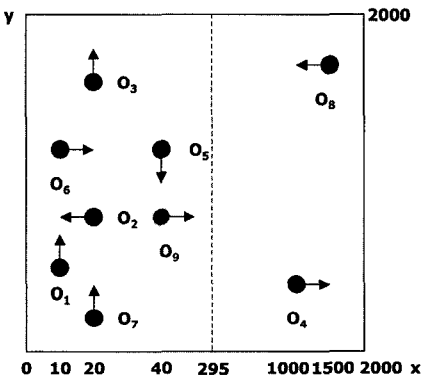
```
Algorithm Insert
입력값
    mObj : 삽입 객체의 정보를 유지하는 구조체
{
   SearchBucket(mObj, ptrBucket); /* 해쉬 버켓의 포인터 획득 */
   if(leaf = FindLeafNode(mObj, ptrBucket)) { /* 단말 노드 포인터 획득 */
      if(!(ChgLeafPos(mObj, leaf))) {  /* 객체의 변경 위치 검사 */
         Update(mObj, leaf);  /* 기존 노드 안에서의 위치 변경 */
         if(!(ChgLeaf(mObj, leaf))) { /* 단말 노드의 변경사항 검사 */
            /* 객체가 삽입될 단말 노드의 전체 경로 검색 */
            SearchNodePath(mObj, leaf, pathQueue);
            Update(leaf, pathQueue); } } /* 중간 노드의 정보 변경 */
         else {
            Delete(mObj, leaf);  /* 다른 노드로의 위치 변경 */
            SearchNodePath(mObj, leaf, pathQueue);
            if(leaf.EntryNum < MAXENTRY) /* 오버플로우 검사 */
               Insert(mObj, leaf, pathQueue); /* 단말 노드에 삽입 연산 */
            else
               Split(mObj, leaf, pathQueue); /* 분할 연산 */
            /* 보조 인덱스와 TPKDB-트리의 단말노드 연결 */
            LinkLeaf(ptrBucket, leaf); } /* 버켓과 단말 노드 연결 */
   }
   else {
      if(ptrBucket->entryNum < HASH_MAXENTRY)
         Insert(mObj, ptrBucket); /* 버켓에 OID 삽입 */
      else
         SplitBucket(mObj, ptrBucket); /* 오버플로우 발생한 버켓 분할 */
      SearchNodePath(mObj, leaf, pathQueue);
      if(leaf.EntryNum < MAXENTRY)
         Insert(mObj, leaf, pathQueue);
      else
         Split(mObj, leaf, pathQueue); /* 오버플로우 발생시 분할 처리 */
      LinkLeaf(ptrBucket, Leaf);
   }
}
```

인해 단말 노드가 분할된다. 단말 노드 분할 정책에는 데이타 종속적인 분할과 분포 종속적인 분할이 있다. 고정된 공간을 색인하는 KDB-트리가 광범위한 공간을 색인하면서 일부 영역으로 객체가 편향되어 색인되는 경우 두 분할 정책 모두 공간활용도 저하 문제를 가질 수 있다. 그림 8은 오버플로우가 발생한 단말 노드의 분할을 보여준다. 단말 노드가 수용할 수 있는 최대 엔트리 수를 8이라고 가정했을 때, 이동 객체 O9의 삽입으로 인해 단말 노드에 오버플로우가 발생하는 것을 보여준다. 그림 8의 (a)에서는 분포 종속적인 분할로 인한 단말 노드 분할 예를 보여주고 있고 편향된 객체를 포함하고 있는 단말 노드의 분포 종속적인 분할로 인해 공간활용도 저하 문제가 발생하는 것을 확인할 수 있다. 그림 8의 (b)에서는 데이타 종속적인 분할로 인해 단말 노드 분할 예를 보여주고 있다. 데이타 종속적인 분할 또한 그림 8의 (a)에서와 마찬가지로 편향된 객체를 포함하는 경우 분할로 인해 공간활용도 저하 문제가 발생하는 것을 확인할 수 있다.

계속적인 단말 노드 분할로 인해 중간 노드가 오버플로우 되면 중간 노드를 분할한다. 중간 노드 분할 정책에는 FS 분할과 FD 분할이 있다. FS 분할은 강제적 분할 전파 문제로 인해 단말 노드를 과도하게 분할한다.

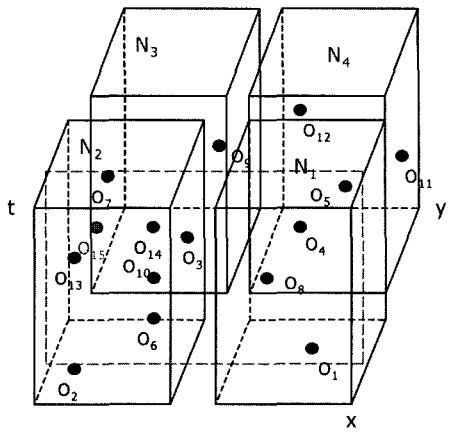


(a) 분포 종속적인 분할 예

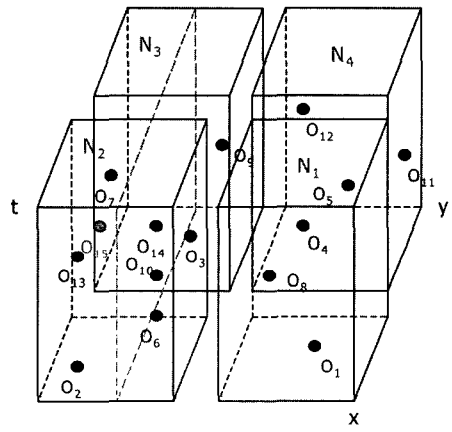


(b) 데이타 종속적인 분할 예

그림 8 오버플로우 발생한 단말 노드 분할



(a) x축 분할

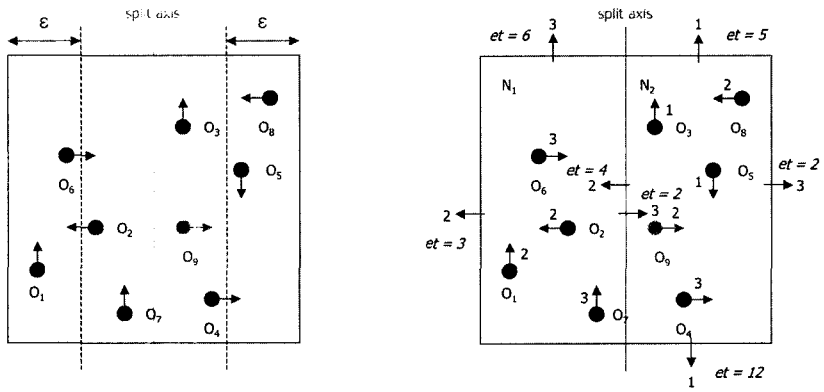


(b) y축 분할

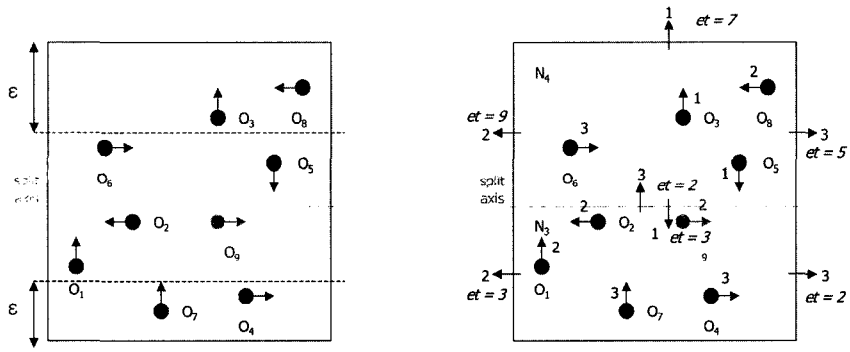
그림 9 단말 노드의 FS 분할

중간 노드가 강제적으로 하위의 단말 노드에 분할을 전파하게 되면 단말 노드들의 공간활용도가 저하되는 문제를 가진다. 이 문제는 이동 객체의 특성을 색인하는 KDB-트리의 단말 노드에서 더욱 심각한 공간활용도 저하 문제를 발생시킨다. 그림 9는 FS 분할로 인한 단말 노드들의 강제 분할 예를 보여주고 있다. 단말 노드가 수용할 수 있는 최대 단말 노드의 수를 5라고 가정했을 때 이동 객체 O_2, O_3, O_6, O_{13} 그리고 O_{14}를 수용하는 단말 노드 N_2는 이동 객체 O_{15}의 삽입으로 오버플로우가 발생한다. 오버플로우가 발생한 단말 노드 N_2는 분할을 통해 오버플로우 문제를 해결한다. 그림 9의 (a)는 단말 노드 N_2의 오버플로우로 인해 공간 도메인 x축을 기준으로 강제 분할된 예를 보여주고 그림 9의 (b)는 공간 도메인 y축을 기준으로 강제 분할된 예를 보여주고 있다. FS 분할 정책에 의해 단말 노드를 분할할 경우 오버플로우가 발생한 단말 노드 이외에도 강제적으로 분할이 발생하는 것을 확인할 수 있다. 그림 9의 (a)에서 강제 분할이 발생한 단말 노드 N_3과 그림 9의 (b)에서 강제 분할이 발생한 단말 노드 N_1과 같이 분할 시점이 아닌 다수의 단말 노드가 분할되어 색인의 크기를 증가시키고 공간활용도 저하 문제가 발생하는 것을 확인할 수 있다. FD 분할은 중간 노드를 분할할 때 하위 영역에 대해 강제적으로 분할 전파가 일어나는 것을 제거함으로서 불필요한 단말 노드의 분할을 감소시켜 다차원 색인시 FS 분할을 사용하는 것보다 성능이 좋은 것으로 연구되었다[9]. 하지만, FD 분할에서도 엔트리가 특정 영역에 편향하는 경우 처음 분할이 발생한 분할 축을 기준으로 일부 영역에 노드가 편향되므로 색인의 크기가 증가하고 공간활용도 저하 문제가 발생하는 것을 확인할 수 있다.

본 논문에서 제안하는 TPKDB-트리의 분할은 기존 KDB-트리의 분할이 가지는 공간활용도 저하 문제를 해결하기 위해서, 데이타 종속적인 분할과 FD 분할을



(a) x축에 의한 단말 노드 분할



(b) y축에 의한 단말 노드 분할

그림 10 단말 노드의 EDD 분할

기반으로 하면서 기존 색인 분할보다 공간활용도가 개선된 분할 정책을 제시한다. 본 논문에서는 단말 노드 분할 정책으로 EDD 분할(Enhanced Data Dependent splitting) 정책과 중간 노드 분할 정책으로 EFD 분할 (Enhanced First Division Splitting) 정책을 제안한다.

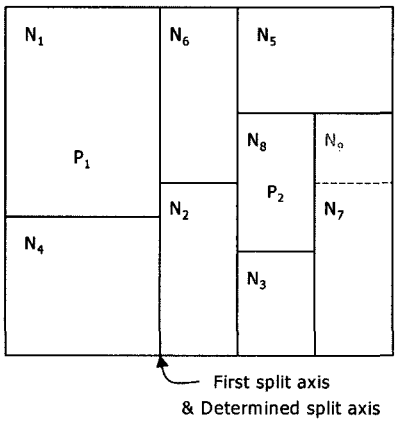
EDD 분할 정책에서는 단말 노드 오버플로우 발생시 데이타 종속적인 분할을 사용하여 모든 도메인에 대한 분할 위치를 선정한다. 분할 위치를 선정할 때는 분할되는 두 단말 노드가 수용해야 하는 최소 엔트리 수에 참여되는 데이타를 제외한 데이타들의 도메인 평균값을 분할 위치로 선정한다. 분할에 참여하는 단말 노드들의 최소 엔트리 수를 보장함으로서 편향된 데이타를 포함하는 오버플로우된 단말 노드의 공간활용도 저하 문제를 해결할 수 있다. 각 도메인에 대한 분할 위치를 선정 후, 각 단말 노드의 t_{upd}, t_{esc} 그리고 v_{max}를 구한다. 그리고 다음 미래 위치 검색시 노드의 확장을 최소화하기 위해서 임의의 t (>각 단말 노드의 $t_{esc}+t_{upd}$중 가장 큰 값)에 대해 각 단말 노드를 확장한 후, 상대적으로 작게 확장되는 단말 노드를 포함하는 분할 도메인을 분할 도메인으로 선정한다.

그림 10은 EDD 분할로 인한 단말 노드 분할을 보여주고 있다. 단말 노드가 수용할 수 있는 엔트리 수가 8이고 최소 엔트리 수를 2라고 가정한다. 이동 객체 O_9의 삽입으로 단말 노드가 오버플로우 되며 단말 노드가 분할된다. 먼저, EDD 분할은 분할 위치를 선정하고 분할된 각 단말 노드의 t_{upd}, t_{esc} 그리고 v_{max}를 구한다. 그림 10의 (a)에서는 공간 도메인 x축에 대한 단말 노드 분할 예를 보여주고 있다. 그림 10의 (b)에서는 공간 도메인 y축에 대한 단말 노드 분할 예를 보여주고 있다. 공간 도메인 x 축에 대한 분할 위치를 선정할 때 분할될 각 노드의 최소 엔트리 수를 보장하기 위해 이동 객체 O_2, O_3, O_4, O_7 그리고 O_9의 도메인 x축의 평균값을 분할 위치로 선정하고, 도메인 y축에 대한 분할 위치를 선정할 때는 이동 객체 O_1, O_2, O_5, O_6 그리고 O_9의 도메인 y축의 평균값을 분할 위치로 선정한다.

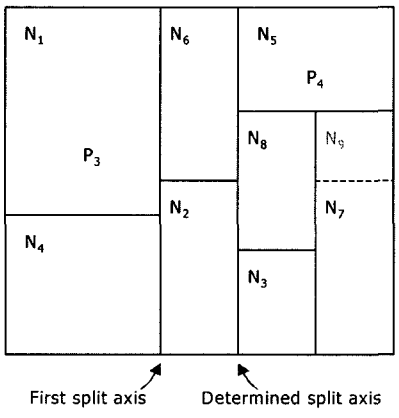
오버플로우 단말 노드를 공간 도메인 x축으로 분할할 경우 단말 노드 N_1과 N_2로 분할되고 공간 도메인 y축으로 분할할 경우 단말 노드 N_3와 N_4로 분할된다. 분할

된 각 단말 노드 N_1, N_2, N_3와 N_4 중에 단말 노드 N_2가 가장 큰 t_{upd2}-=12을 유지하므로, 단말 노드 N_1, N_2, N_3와 N_4을 임의의 t (>(t_{upd2}-=12))에 대해 확장을 한다. 확장된 각 단말 노드 N_1, N_2, N_3와 N_4의 확장 비율을 비교하면 단말 노드 N_1과 N_2의 확장 비율이 단말 노드 N_3과 N_4에 비해 작으므로 공간 도메인 x축에 대한 분할을 선택한다.

EFD 분할은 중간 노드 분할시 단말 노드의 강제 분할을 막기 위해 FD 분할을 사용하고 공간활용도를 개선하기 위해서 분할 축을 오버플로우 중간 노드의 첫 분할 도메인으로 선정하는 것이 아니라, 첫 분할 도메인을 기준으로 분할했을 때보다 공간활용도를 개선시키는 분할 도메인을 분할 축으로 선정한다. 첫 분할 도메인에 비해 공간활용도를 개선시키는 분할 도메인이 없다면 FD 분할과 마찬가지로 첫 분할 도메인을 분할 축으로 선정한다.



(a) FD 분할



(b) EFD 분할

그림 11 중간 노드의 EFD 분할

그림 11은 FD 분할과 EFD 분할로 인한 중간 노드의 분할을 보여준다. 중간 노드의 최대 수용 엔트리 수를 8이라고 가정한다. 그림 11의 (a)에서는 FD 분할로 인한 오버플로우된 중간 노드의 분할을 보여주고 있다. FD 분할은 오버플로우된 중간 노드의 첫 분할 도메인을 도메인 축으로 선정하고 분할하기 때문에 단말 노드 N_1과 N_4를 포함하는 중간 노드 P_1과 단말 노드 N_2, N_3, N_5, N_6, N_7 그리고 N_8을 포함하는 P_2로 분할된다.

첫 분할 도메인을 기준으로 일부 영역에 단말 노드가 편향한 오버플로우된 중간 노드를 FD 분할을 사용하여 분할할 경우 분할된 중간 노드로 균등하게 단말 노드가 분할되지 않아 공간활용도가 저하되는 것을 볼 수 있다. 그림 11의 (b)에서는 EFD 분할로 인한 오버플로우 중간 노드의 분할을 보여주고 있다. EFD 분할은 오버플로우된 중간 노드의 첫 분할 도메인을 분할 축으로 선정하는 것이 아니라 오버플로우된 중간 노드에 포함되어 있는 단말 노드들을 균등하게 분할하는 분할 도메인을 분할 축으로 선정한다. 그림 11의 (b)에서와 같이 오버플로우된 중간 노드를 단말 노드 N_1, N_2, N_4 그리고 N_6을 포함하는 중간 노드 P_3와 단말 노드 N_3, N_5, N_7 그리고 N_8을 포함하는 중간 노드 P_4로 균등 분할하는 분할 도메인을 분할 축으로 선정한다.

3.3 삭제 알고리즘

삭제 연산은 이동 객체의 갱신, 분할 그리고 합병에 따른 재삽입을 위해 기존 이동 객체를 삭제하는 경우에 요청된다. 아래 Delete 알고리즘은 TPKDB-트리에서의 삭제 알고리즘을 나타낸다. 삭제하려는 이동 객체를

```
Algorithm Delete
입력값
    mObj : 삭제 객체의 정보를 유지하는 구조체
    leaf : 삭제 객체를 포함하는 TPKDB-트리의
    단말 노드 포인터
{
  /* 삭제 객체를 단말 노드에서 삭제한다. */
  Delete(mObj, leaf);
  /* 언더플로우가 발생하는지 검사 */
  if(leaf_NUM >= MINENTRY) {
    /* 단말 노드의 정보가 변경되었는지 검사 */
    if(ChgLeaf(mObj, leaf)) {
      SearchNodePath(mObj, leaf, pathQueue);
      Update(leaf, pathQueue); } }
    else {
      SearchNodePath(mObj, leaf, pathQueue);
      /* 언더플로우 발생한 단말 노드 합병 */
      Merge(leaf, pathQueue); }
}
```

Delete 알고리즘을 통해 삭제 후, 삭제된 이동 객체를 포함하고 있던 단말 노드에서 언더플로우(Underflow)가 발생하는지를 검사한다. 언더플로우가 발생하지 않는다면 ChgLeaf( )를 통해 삭제된 이동 객체에 의해 단말 노드의 t_{upd}, t_{esc} 그리고 v_{max}가 변경되는지 검사한다. 삭제된 이동 객체에 의해 단말 노드의 t_{upd}, t_{esc} 그리고 v_{max}가 변경되지 않는다면 삭제 연산을 마친다. 변경이 발생한다면 Update( )를 통해 변경된 사항을 루트 노드까지 반영한다. 만약, 삭제된 이동 객체에 의해 해당 단말 노드에 언더플로우가 발생한다면 Merge( )를 통해 언더플로우가 발생한 단말 노드를 주변 노드와 합병함으로서 색인 구조의 공간활용도를 높인다.

4. 미래 위치 검색 기법

본 논문에서 제안하는 TPKDB-트리는 이동 객체의 현재 위치뿐만 아니라 미래 위치 검색을 지원하는 색인 구조이다. TPKDB-트리에서는 객체 질의와 범위 질의를 제공한다. 객체 질의 $Q=(OID,t)$는 t시점에서 OID을 갖는 객체에 대한 위치를 검색하는 것으로 t의 값에 따라 현재 또는 미래의 객체의 위치를 검색한다. 이에 반해 범위 질의 $Q=(R,t)$는 t시점에서 R 영역 내에 포함된 객체를 검색하는 것으로 t의 값에 따라 R 영역 내에 포함된 현재 또는 미래의 위치를 검색한다. R은 검색하고자 하는 공간 영역 정보로 $(R_1, R_2, \ldots, R_n)$와 같다. 이때, $R_i=[p_i^-, p_i^+]$로 p_i^-와 p_i^+는 i번째 차원의 시작 위치와 끝 위치를 나타낸다. 객체 질의는 "28번 시내버스의 현재 위치를 검색하라" 또는 "24번 시내버스의 두 시간 뒤의 위치를 검색하라"와 같이 특정 이동 객체에 대한 현재와 미래 위치를 검색하는 질의이다. 범위 질의는 "현재 운동장 내에 있는 이동 객체를 검색하라" 또는 "앞으로 세 시간 뒤에 정문 앞에 있는 이동 객체를 검색하라"와 같이 현재 또는 미래 시간에 대해 정의된 영역 안에 있는 모든 이동 객체를 검색하는 질의이다.

그림 12는 TPKDB-트리에서 제공하는 여러 종류의 질의의 예를 나타낸 것이다. TPKDB-트리는 색인된 이동 객체들의 정보를 가지고 간단한 선형 함수를 만들어 이동 객체의 궤적을 만들 수 있다. 그림 12에서 화살표는 이동 객체 O_1, O_2 그리고 O_3에 대한 정보를 가지고 이동 객체들의 궤적을 나타낸 것이고, 이동 객체 O_1과 O_2 같이 지속적으로 이동 객체의 정보가 변경될 수 있기 때문에 시간에 따라 이동 객체의 궤적은 변경될 수 있다. Q_1, Q_2 그리고 Q_3은 이동 객체 O_1에 대한 객체 질의 그리고 Q_4는 범위 질의를 나타낸 것이다. 색인 구조의 현재 시간 t_{cur}이 t_1이고 이동 객체 O_1에 대한 객체 질의를 요청하면 Q_1과 겹치는 O_1의 궤적에 대한 값을 결과로 얻을 수 있다. 또한, t_{cur}이 t_2이고 범위 질의인 Q_2를 요청하면 Q_2의 질의 영역을 나타내는 사각형 안에 포함된 모든 궤적에 대한 이동 객체를 결과로 얻을 수 있다. 여기서는, Q_2에 대한 질의의 결과로 이동 객체 O_1과 O_2를 결과로 얻는다. 이동 객체의 궤적이 이동 객체를 나타내는 t_{ref}, v_x, v_y, x_{ref} 그리고 y_{ref}에 종속적이기 때문에 질의 요청시 질의를 내린 시간에 따라 상이한 결과 값을 얻을 수 있다. 예를 들어, t_{cur}이 t_3보다 작은 시점에서 t_{cur}보다 향후 시간인 t_4에 대한 이동 객체 O_1에 대해 객체 질의시 Q_2에 대한 결과를 얻고 t_{cur}이 t_3보다 큰 시점에서 동일한 질의에 대해서는 Q_3에 대한 결과를 얻는다.

일반적으로, 객체 질의는 보조 색인 구조를 사용하여 빠르게 검색할 수 있고 현재 시간에 대한 범위 질의는 기존 KDB-트리에서 일정 질의 영역 안에 있는 객체를 검색하는 방법과 동일하다. 단지, TPKDB-트리에서는 향후 시간에 대한 범위 질의를 제공하기 때문에 기존 KDB-트리의 각 노드에 t_{upd}와 t_{esc} 같은 시간 정보와 속도 정보 v_{max}가 포함된다.

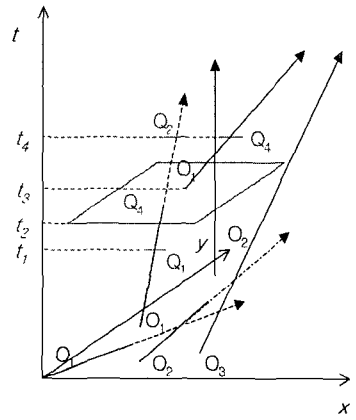
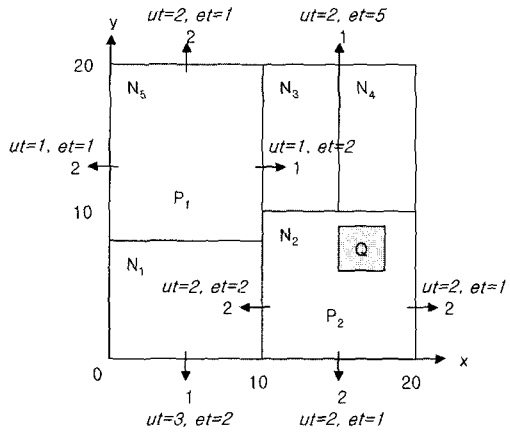


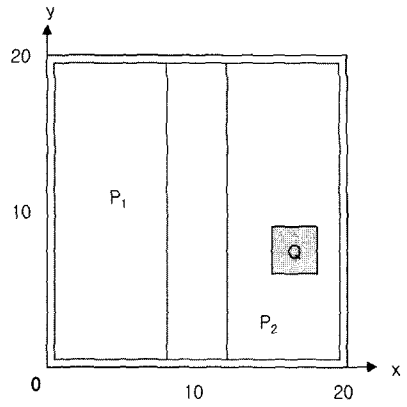
그림 12 TPKDB-트리의 질의 종류

그림 13에서는 TPKDB-트리에서 이동 객체의 향후 위치에 대한 검색 방법을 보여주고 있다. 그림 13의 (a)에서는 t_{cur}=3 일 때, TPKDB-트리의 중간 노드들의 구조와 t_{query}=5에 대한 범위 질의가 요청된 것을 나타낸다. TPKDB-트리는 요청된 질의를 처리하기 위해서 중간 노드 P_1과 P_2의 t_{upd}, t_{esc} 그리고 v를 가지고 t_{query}=5에 대해 노드를 확장한다. 각각의 노드의 확장은 식 (2)에 의해 구할 수 있다.

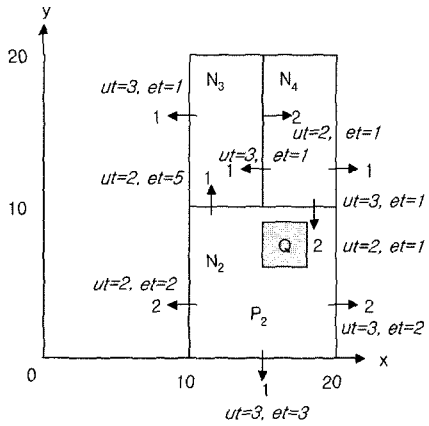
$$ES=\begin{cases} RS & , \text{ if } t_{query} \le (t_{upt}+t_{esc}) \\ RS+(t_{query}-(t_{upd}+t_{esc}))\times v_{\max} & , \text{ else} \end{cases} \quad (2)$$



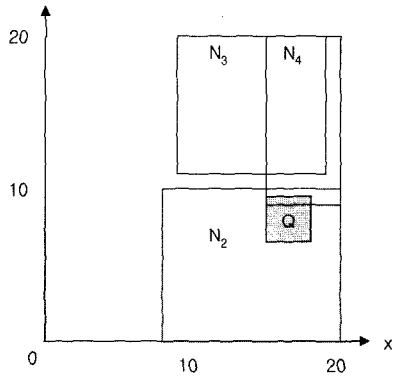
(a) t_{cur} = 3 일 때, 중간노드 구조



(b) t_{query} = 5 일 때, 중간노드 확장



(c) t_{cur} = 3 일 때, 단말노드 구조



(d) t_{query} = 5 일 때, 단말노드 확장

그림 13 TPKDB-트리의 미래 위치 검색

*ES*는 미래 위치 검색을 위해 질의된 미래 시간까지 확장된 노드의 영역 정보를 나타낸다. $t_{query} \le t_{upd} + t_{esc}$ 인 노드에 대해서는 질의된 미래 시간까지 노드를 확장하지 않아도 기존에 구성된 노드를 사용하여 해당 노드에 포함되어 있는 모든 객체를 처리할 수 있기 때문에 *ES*는 *RS*가 된다. 이로 인해 미래 위치 검색시 불필요한 노드의 확장을 피할 수 있다. 그러나 $t_{query} > t_{upd} + t_{esc}$ 인 노드에 대해서는 기존 노드의 *RS*만으로는 모든 객체를 처리할 수 없기 때문에 질의된 미래 시간까지 노드를 확장한다.

또한, TPKDB-트리가 색인하는 전체 영역 이상에 대한 확장은 무의미하므로 노드 확장시 TPKDB-트리가 색인하는 전체 영역까지만 확장을 한다. 그림 13의 (b)에서는 t_{query}=5에 대해 중간 노드를 확장한 것을 보여준다. 중간 노드 P_1과 P_2를 확장하고 범위 질의의 질의 영역과 비교하였을 때, 중간 노드 P_2만이 질의 영역과 겹침이 발생하기 때문에 중간 노드 P_2에 있는 단말 노드만을 t_{query}=5에 대해 확장을 한다. 그림 13의 (c)는 미래 시간에 대한 질의 처리를 위해 중간 노드를 확장 후 미래 질의 영역과 겹침이 발생하는 중간 노드 P_1만을 나타낸 것이다. 그리고 미래 질의 영역 안에 있는 이동 객체들을 찾기 위해 중간 노드 P_1에 포함된 단말 노드를 확장하는 과정을 보여주고 있다. 그림 13의 (d)는 t_{query}=5에 대해 확장된 단말 노드를 보여주고 있으며, 미래 질의 영역과 겹침이 발생하는 단말 노드는 N_2와 N_4인 것을 알 수 있다. 마지막으로, N_2와 N_4에 있는 이동 객체들의 궤적을 이용하여 질의 영역과 겹침이 발생하는 이동 객체들을 반환함으로서 요구된 질의를 처리할 수 있다.

5. 실험 및 성능 평가

5.1 실험 환경

성능 평가에 사용된 시스템은 펜티엄-Ⅳ 1.7㎓ 프로세서에 256Mbyte의 메모리를 가지는 윈도우 2000 운영체제를 사용하였다. 성능평가에 사용된 데이타 집합은 GiST[10]를 통해 가우시안(Gaussian), 불균형(Skew) 그리고 균등(Uniform) 분포된 최대 100,000개 객체를 이용하여 수행하였고, 색인 구조에서 노드의 크기는 4Kbytes로 한다. GiST를 통해 생성된 데이타는 이동 객체의 특성인 위치 정보와 속도 정보를 가져야 하기 때문에 1000×1000㎞의 2차원 영역을 랜덤 함수를 이용해 추출된 속도와 방향을 가지고 움직이는 이동 객체로 가정한다. 이동 객체가 가지는 속도는 1, 1.5, 2, 2.5 그리고 3km/min (60, 90, 120, 150 그리고 180 km/h)로 가정한다.

본 논문에서 제안하는 TPKDB-트리는 이동 객체의 계속되는 위치 이동으로 색인의 변경이 발생하고, 색인의 빈번한 변경으로 전체적인 색인의 성능이 저하되는 기존 R 트리 계열의 색인 구조 문제를 해결하기 위해 제안된 색인 구조이다. 제안한 색인 구조의 성능을 평가하기 위해서 본 실험에서는 R-트리 계열의 데이타 분할 방식을 사용하여 이동 객체를 색인하는 TPR-트리[5]와 비교한다. 또한 제안하는 색인 구조에서 사용되는 보조 색인 구조는 해쉬를 이용하여 구현한다. 표 1은 성능 평가에 사용된 파라미터를 나타낸다.

표 1 성능평가를 위한 파라미터

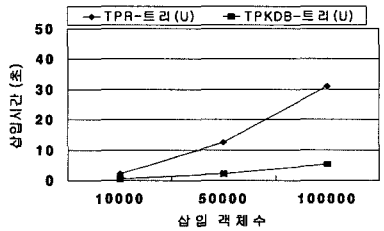
항 목	값
삽입 이동 객체 수 [개]	10,000~100,000
갱신 객체 수 [개]	500~10,000
질의 시간 간격 [분]	5~30
질의 영역 크기 [전체영역 %]	0.1~5

먼저, 최대 100,000개의 이동 객체를 색인 구조에 삽입하는 시간을 측정하였고 100,000개의 객체로 구성된 색인 구조에 최대 10,000개의 데이타가 갱신 시간을 측정하였다. 또한, 미래 위치 검색 처리 성능을 평가하기 위해서 100,000개의 데이타가 색인된 색인 구조에 5~30분 후 범위 질의 1000개를 처리하는데 소요되는 시간과 20분 후의 범위 질의에 대해 질의 영역의 크기를 변경시키면서 미래 위치 검색을 처리하는데 소요되는 시간을 측정하였다.

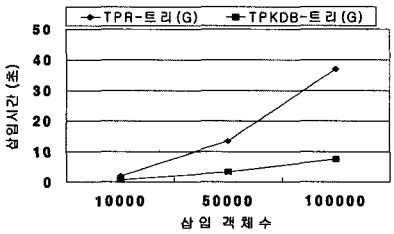
5.2 성능 평가

제안하는 TPKDB-트리는 계속적인 위치 변경이 발생하는 이동 객체에 대한 갱신 비용을 최소화하고 현재 및 미래 위치에 대한 검색 성능을 향상시키기 위한 색인 구조이다. 제안하는 색인 구조의 우수성을 입증하기 위해 기존에 제안된 TPR-트리와 제안하는 TPKDB-트리에 대한 색인 구성 시간 및 검색 시간에 대한 비교를 수행한다. 색인 구성 시간에 대한 비교는 삽입 시간과 갱신 시간 측면에서 비교를 수행하고 검색 시간에 대한 비교는 현재와 미래 위치에 대한 범위 질의를 수행한다.

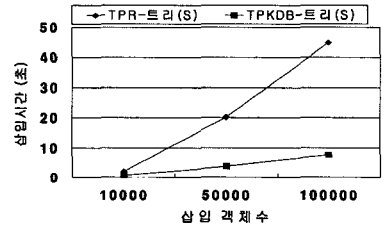
그림 14는 제안하는 TPKDB-트리와 TPR-트리에 대한 삽입 시간을 비교한 결과이다. 삽입 성능을 실험하기 위해서 균등 분포, 가우시안 분포 그리고 불균등 분포된 데이타의 개수를 10,000~100,000개까지 변경시키면서 삽입 시간을 측정하였다. 그림 14의 (a)는 균등 분포된 데이타 집합에 대한 삽입 성능 평과 결과를 (b)는 가우시안 분포된 데이타 집합에 대한 삽입 성능 평과 결과를 나타낸다. 그리고 (c)는 불균형 분포된 데이타 집합에 대한 삽입 성능 평과 결과를 나타낸다. 성능 평가 결과 TPKDB-트리가 TPR-트리 비해 300%~550%의 성능 향상을 나타내었다. 특히, 제안하는 TPKDB-트리는 불균등 데이타 집합에 대해서는 TPR-트리보다 삽입되는 객체의 수가 증가될수록 매우 향상된 삽입 성능을 보인다.



(a) 균등 분포에 대한 삽입 시간



(b) 가우시안 분포에 대한 삽입 시간

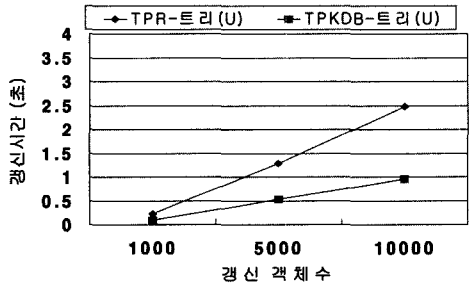


(c) 불균등 분포에 대한 삽입 시간

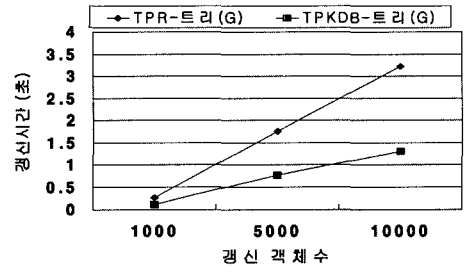
그림 14 TPKDB-트리와 TPR-트리의 삽입 시간

그림 15는 제안하는 TPKDB-트리와 TPR-트리에 대한 갱신 시간을 비교한 결과이다. 갱신 성능을 실험하기 위해서 균등 분포, 가우시안 분포 그리고 불균등 분포된 100,000개의 데이타를 색인하고 있는 각 색인 구조에 데이타의 개수를 500~10,000개까지 변경시키면서 갱신 시간을 측정하였다. 그림 15의 (a)는 균등 분포된 데이타 집합에 대한 갱신 성능 평과 결과를 (b)는 가우시안 분포된 데이타 집합에 대한 갱신 성능 평과 결과를 나타낸다. 그리고 (c)는 불균형 분포된 데이타 집합에 대한 갱신 성능 평과 결과를 나타낸다. 성능 평가 결과 TPKDB-트리가 TPR-트리 비해 200%~300%의 성능 향상을 나타내었다.

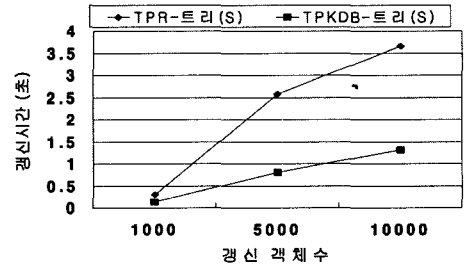
삽입과 갱신 성능 평가를 수행한 결과 TPKDB-트리는 모든 데이타 집합에 대해 TPR-트리보다 향상된 성능을 나타내었다. 특히, 삽입되는 객체의 수가 증가할수록 향상된 성능을 나타낸다. 기존의 TPR-트리는 R-트리 기반의 색인 구조이기 때문에 계속적인 객체의 삽입이나 위치 이동으로 인해 색인 구조에 빈번한 갱신이 발생하기 때문에 많은 시간이 요구된다. 이에 반해, 제안하는 TPKDB-트리는 공간 분할 방법을 사용하여 데이타를 색인하기 때문에 데이타 분할 방법을 사용하는 색인 구조의 문제점인 빈번한 색인 구조의 변경 비용을 줄일 수 있고 보조 색인 구조를 통한 빠른 객체의 검색을 지원함으로써 갱신 비용을 효율적으로 줄일 수 있다.



(a) 균등 분포에 대한 갱신 시간



(b) 가우시안 분포에 대한 갱신 시간



(c) 불균등 분포에 대한 갱신 시간

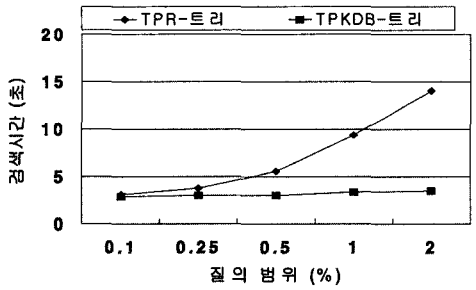
그림 15 TPKDB-트리와 TPR-트리의 갱신 시간

데이타의 분포 형태에 따른 성능을 비교하면 TPR-트리와 TPKDB-트리 모두 삽입과 갱신 연산에 있어 데이타의 분포가 균등 분포일 때, 가장 좋은 성능을 보이는 것을 볼 수 있다. TPR-트리는 데이타 분포 형태가 가우시안과 불균형 분포일 때 노드들 간에 많은 겹침 영역이 증가되어 검색 성능이 저하되고 이로 인해 삽입과 갱신 시간이 저하되는 것을 확인할 수 있다. 하지만, TPKDB-트리는 노드들 간에 겹침이 발생하지 않는 색인 구조의 특성으로 인하여 삽입과 갱신 연산의 성능이 데이타 분포 형태에 영향을 받는 것이 아니라 데이타의 수에 영향을 받기 때문에 삽입과 갱신 시간이 데이타의 수에 따라 비례적으로 증가하는 것을 확인할 수 있다.

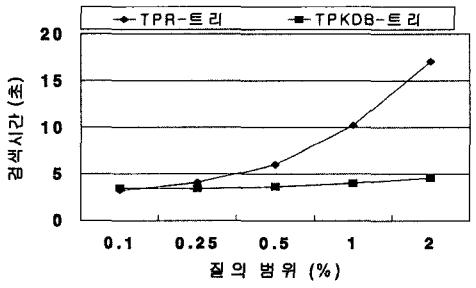
그림 16은 제안하는 TPKDB-트리에 대한 질의 범위에 따른 검색 성능에 대한 성능 평가를 수행한 결과이다. 질의 범위에 따른 검색 성능을 실험하기 위해서 균등 분포, 가우시안 분포 그리고 불균등 분포된 100,000개의 데이타를 색인하고 범위 질의 영역의 크기를 전체 영역에 대해 0.12%까지 변경시키면서 1000개의 질의를 처리하는 시간을 측정하였다. 성능 평가 결과 TPKDB-트리가 TPR-트리 비해 100%~400%의 성능 향상을 나타내었다.

그림 17은 제안하는 TPKDB-트리에 대한 미래 시간에 따른 검색 성능에 대한 성능 평가를 수행한 결과이다. 질의 범위에 따른 검색 성능을 실험하기 위해서 균등 분포, 가우시안 분포 그리고 불균등 분포된 100,000개의 데이타를 색인하고 미래 위치 검색을 위해 범위 질의 시간을 5~30분까지 변경시키면서 1000개의 질의를 처리하는 시간을 측정하였다. 성능 평가 결과 TPKDB-트리가 TPR-트리 비해 100%~300%의 성능 향상을 나타내었다.

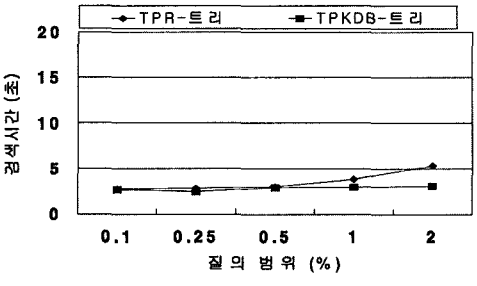
현재 위치와 미래 위치 검색 성능 평가 결과 TPKDB-트리가 TPR-트리보다 모든 경우에서 향상된 성능을 나타내었다. 특히, 삽입되는 객체의 수가 증가할수록 TPKDB-트리가 TPR-트리에 비해 매우 향상된 성능을 나타내었다. TPR-트리는 데이타를 색인하는 MBR 또는 미래 위치 검색시 확장되는 MBR의 많은



(a) 균등 분포에 대한 현재 위치 검색

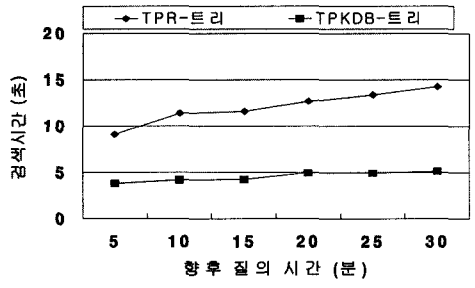


(b) 가우시안 분포에 대한 현재 위치 검색

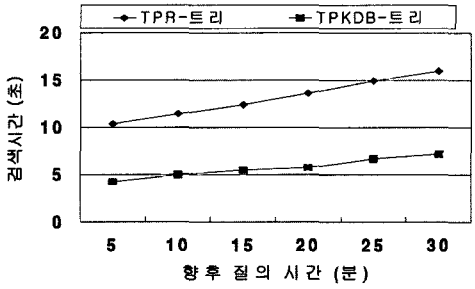


(c) 불균형 분포에 대한 현재 위치 검색

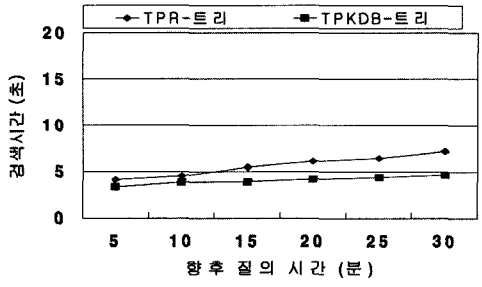
그림 16 TPKDB-트리와 TPR-트리의 현재 위치 검색



(a) 균등 분포에 대한 미래 위치 검색



(b) 가우시안 분포에 대한 미래 위치 검색



(c) 불균등 분포에 대한 미래 위치 검색

그림 17 TPKDB-트리와 TPR-트리의 미래 위치 검색

겹침과 색인의 빈번한 변경으로 전체적인 색인의 검색 성능이 저하되지만 TPKDB-트리는 데이타 색인시 영역 간에 겹침이 발생하지 않는 장점과 미래 위치 검색 시 불필요한 노드의 확장을 줄이고자 노드 내에 포함되어 있는 이동 객체의 변화를 시간에 대한 파라미터로 유지함으로서 검색 성능을 향상시킬 수 있다.

6. 결론 및 향후 연구

본 논문에서는 이동 객체에 대한 갱신 및 미래 위치 검색을 효율적으로 지원하는 시공간 색인 구조를 제안하였다. 제안한 TPKDB-트리는 이동 객체의 갱신 비용을 줄이기 위해 성능이 개선된 KDB-트리와 보조 색인 구조의 혼합형 색인 구조이다. 제안한 색인 구조에서는 빠른 이동 객체의 미래 위치 검색을 위해 노드와 노드 안에 포함되어 있는 이동 객체의 관계를 시간에 대한 파라미터로 유지하였다. 또한, 색인 구조의 공간활용도 및 검색 성능을 향상시키기 위해서 새로운 EDD 분할과 EFD 분할을 제안하였다. 성능 평가를 통해 기존에 제안된 TPR-트리에 비해 300%~500%의 삽입 성능 향상과 200%~300%의 갱신 성능 향상을 보였다. 또한 질의 범위 검색에 있어 100%~400%의 성능 향상과 이동 객체의 미래 위치 검색에 있어 100%~300%의 성능 향상을 보였다.

향후 연구 방향으로 다양한 유형의 검색 기법에 대한 연구와 함께 갱신 비용을 최소화하기 위한 기법과 벌크 연산 기법들에 대한 연구를 계속 수행할 예정이다.

참 고 문 헌

[1] R. Ding, X. Meng, and Y. Bai, "Efficient index update for moving objects with future trajectories," Proc. Eighth International Conference on Database Systems for Advanced Applications, pp.183-194, 2003.

[2] A. Guttman, "R-trees : A dynamic index structure for spatial searching," Proc. ACM SIGMOD Conference, pp.47-57, 1984.

[3] D. S. Kwon, S. J. Lee, and S. H. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," Proc. International Conference on Mobile Data Management, pp.113-120, 2002.

[4] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch, "Query indexing and velocity constrained indexing : scalable techniques for continuous queries on moving objects," IEEE Transactions on Computers, Vol.51, No.10, pp.1124-1140, 2002.

[5] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, "Indexing the Positions of Continuously Moving Objects," Proc. ACM SIGMOD Conference, pp.331-342, 2000.

[6] A. Guttman, "A dynamic index structure for spatial searching," Proc. ACM SIGMOD Conference. pp.47-57, 1984.

[7] X. Yuni and S. Prabhakar, "Q+Rtree : Efficient Indexing for Moving Object Database," Proc. Eighth International Conference on Database Systems for Advanced Applications, pp.175-182, 2003.

[8] O. Ratko and Y. Byunggu, "Implementing KDB-Trees to support High-Dimensional Data," Proc. International Database Engineering and Applications Symposium, pp.58-67, 2001.

[9] A. Henrich, H. S. Six, and P. Widmayer, "The LSD-tree : Spatial Access to Multidimensional Point and Non-point Objects," Proc. International Conference on Very Large Data Bases, pp.45-53, 1989.

[10] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer, "Generalized Search Trees for Database Systems," Proc. International Conference on Very Large Data Bases, pp. 562-573, 1995.

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

[12] J. Basch, L. J. Guibas, and J. Hershberger, "Data Structures for Mobile Data," Proc. Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, pp.747-756, 1997.

[13] H. Chon, D. Agrawal, and A. Abbadi, "Using Space-Time Grid for Efficient Management of Moving Objects," Proc. International ACM Workshop on Data Engineering for Wireless and Mobile Access, pp.59-65, 2001.

[14] C. Procopiuc, P. Agarwal, and S. Har-Peled, "STAR-Tree : An Efficient Self-Adjusting Index for Moving Objects," 4th International Workshop of Algorithm Engineering and Experimentation, pp.178-194, 2002.

[15] B. C. Ooi, K. L Tan, and C. Yu, "Frequent Update and Efficient Retrieval : an Oxymoron on Moving Object Indexes?," Proc. 3rd International Conference on Web Information Systems Engineering Workshops, pp.3-12, 2002.

[16] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Kenglik, "Supporting Frequent Updates in R-Trees : A Bottom-Up Approach," Proc. 29th International Conference on Very Large Data Bases, pp.608-619, 2003.

서 동 민

2002년 충북대학교 정보통신공학과(공학사). 2004년 충북대학교 정보통신공학과(공학석사). 2004년~현재 충북대학교 정보통신공학과 박사과정. 관심분야는 데이타베이스 시스템, 에이전트 시스템, XML, 이동 객체 데이타베이스, 시공간 색인 구조 등

복 경 수

정보과학회논문지 : 데이타베이스 제 31 권 제 3 호 참조

유 재 수

정보과학회논문지 : 데이타베이스 제 31 권 제 1 호 참조

이 병 엽

1991년 한국과학기술원 전산학과(공학사). 1993년 한국과학기술원 전산학과(공학석사). 1997년 한국과학기술원 경영정보공학(공학박사). 1997년~2003년 2월 대우정보시스템 CRM사업팀 차장. 2003년~현재 배재대학교 전자상거래학부 전임강사. 관심분야는 데이타마이닝, XML, 인공지능, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅 분야 등