연속적인 이동 객체의 효과적인 갱신을 위한 색인 구조

복 경 수[†] · 윤 호 원[††] · 김 명 호[†††] · 조 기 형[††††] · 유 재 수[†††††]

요 약

기존에 제안된 색인 구조는 연속적 이동 객체를 갱신하기 위해 이동 객체의 기존 위치를 삭제하고 새로운 위치를 삽입하는 과정을 반복하기 때문에 많은 갱신 비용을 소요한다. 본 논문에서는 연속적 이동 객체의 갱신 비용을 감소시키기 위한 새로운 색인 구조를 제안한다. 제안하는 색인 구조는 이동 객체의 위치를 저장한 공간 분할 방식의 색인 구조와 이동 객체의 위치를 직접 접근하기 위한 보조 색인 구조로 구성된다. 노드의 팬아웃을 증가시키기 위해 자식 노드에 대한 정보는 실제 분할 영역을 저장하는 것이 아니라 kd-tree로 저장한다. 또한, 이동 객체의 위치 갱신을 빠르게 처리하기 위해 색인 구조 전체를 순회하지 않고 단말 노드를 직접 접근하여 상향식으로 갱신을 수행한다. 제안하는 색인 구조의 우수성을 입증하기 위해 다양한 분포 특성에 따라 이동 객체를 생성하고 이동 객체에 대한 삽입, 갱신, 검색 성능을 비교 분석한다.

키워드 : 이동 객체, 색인 구조, 갱신, 공간 분할, 미래 위치

An Index Structure for Updating Continuously Moving Objects Efficiently

Kyoung Soo Bok[†] · Ho Won Yoon[††] · Myoung Ho Kim[†††] · Ki Hyung Cho[††††] · Jae Soo Yoo[†††††]

ABSTRACT

Existing index structures need very much update cost because they repeat delete and insert operations in order to update continuously moving objects. In this paper, we propose a new index structure which reduces the update cost of continuously moving objects. The proposed index structure consists of a space partitioning index structure that stores the location of the moving objects and an auxiliary index structure that directly accesses to their current positions. In order to increase the fanout of the node, it stores not the real partitioning area but kd-tree as the information about the child node of the node. In addition, we don't traverse a whole index structure, but access the leaf nodes directly and accomplish a bottom-up update strategy for efficiently updating the positions of moving objects. We show through the various experiments that our index structure outperforms the existing index structures in terms of insertion, update and retrieval.

Key Words : Moving Object, Index Structure, Update, Space Partition, Future Position

1. 서 론

무선 통신 기술 및 위치 획득 기술의 발전과 더불어 이동성에 기반한 다양한 서비스가 개발되고 있다. 이와 함께, 이동 객체 데이터베이스에 대한 많은 연구들이 진행되고 있다. 이동 객체 데이터베이스는 시간에 따라 연속적인 위치 변화를 갖는 이동 객체를 효과적으로 저장하고 관리하기 위한 데이터베이스이다[1]. 전통적인 데이터베이스 시스템은 한번 삽입이 수행되면 갱신이 자주 일어나지 않는 정적인 객체들을 저장하고 관리한다. 따라서, 이동 객체에 대해서는 많은 갱신 비용을 초래하여 데이터베이스 시스템의 성능이 급격히 저하되는 문제점이 있다[2].

대용량의 이동 객체에 대한 효과적인 검색을 수행하기 위해서는 시간의 변화에 따른 공간적인 위치를 빠르게 색인하고 다양한 유형의 검색을 처리할 수 있는 색인 구조가 필수적이다[3, 4]. 이동 객체의 미래 위치를 검색하기 위해 제안된 색인 구조들은 단말 노드에 이동 객체의 위치와 속도 정보를 저장하고 중간 노드에는 자식 노드에 포함된 이동 객체를 포함하는 영역과 미래 위치를 검색하기 위한 파라미터를 저장한다[5-8]. 미래 위치 검색을 지원하는 대표적인 색인 구조인 TPR-트리는 다차원 색인 구조의 대표적인 R*-트리를 기반으로 각 차원의 상한 값과 하한 값을 기준으로 자식 노드에 포함되는 이동 객체의 속도 정보를 중간 노드에 저장하고 이를 통해 미래 위치를 검색한다[5]. VCI-트리

※ 본 과제(결과물)는 한국과학재단 특정기초 연구(과제번호R01 2006-000-10809 0) 지원 및 정보통신부의 IT기초기술연구지원사업(정보통신연구진흥원), 산업자원부 지역혁신인력양성사업의 연구결과물 입니다.
† 준 회 원 : 한국과학기술원 전산학과 Postdoc
†† 준 회 원 : 충북대학교 정보통신공학과 석사과정
††† 정 회 원 : 한국과학기술원 전산학과 교수
†††† 정 회 원 : 충북대학교 정보통신공학과 교수
††††† 종신회원 : 충북대학교 정보통신공학과 교수
논문접수 : 2005년 9월 1일, 심사완료 : 2006년 6월 7일

는 TPR-트리와 유사하게 R*-트리를 기반으로 자식 노드에 포함된 최대 속도를 중간 노드에 저장하여 미래 위치를 검색한다[6].

기존에 제안된 색인 구조들은 이동 객체의 위치를 갱신하기 위해 색인 구조 전체를 순회하면서 이동 객체의 이전 위치를 삭제하고 새로운 위치를 삽입하는 과정을 수행한다. 이러한 갱신 과정은 추가적인 연산을 필요로 하며 이와 함께 많은 I/O를 발생시켜 색인 구조의 성능을 저하시키는 문제가 발생한다[9, 10]. 또한, 이러한 갱신 과정에서 분할된 영역들 사이에 겹침이 증가하거나 객체의 위치 변화로 인해 분할된 영역을 재조정하기 위한 작업을 수행한다. 따라서, 분할된 영역들 사이에 선별력이 저하되어 검색 성능이 저하되는 문제점이 있다. R-트리 기반 색인 구조는 노드에 오버플로우가 발생할 경우 노드에 존재하는 일부 객체들에 대해 재삽입을 수행한다. 이러한 재삽입은 노드에 존재하는 일부 객체들을 삭제하고 색인 구조를 순회하면서 삽입을 수행하기 때문에 많은 시간을 소요하게 된다. 이러한 상황에서 재삽입을 수행하기 위해 선택된 객체에 대한 새로운 위치를 갱신해야 할 경우 재삽입되는 이동 객체의 위치는 무의미하게 된다. 또한, 기존에 제안된 색인 구조는 객체의 이동 방향 및 속도가 일정 시간 동안 고정되어 있다고 가정하고 있다. 그러나 실제 응용 분야에서 활용되는 이동 객체들의 이동은 언제나 변화될 수 있다. 따라서, 실제 객체들에 대한 정확한 위치를 고려하지 않기 때문에 잘못된 검색 결과를 생성할 수 있다. 이에 따라, 이동 객체의 계속적인 위치 변화에 따른 갱신을 효과적으로 수행하기 위한 색인 구조에 대한 필요성이 증가되고 있다.

본 논문에서는 이동 객체의 미래 위치 검색을 지원하기 위해 기존 R-트리 기반 색인 구조에서 발생하는 갱신 문제점을 해결하기 위한 새로운 색인 구조와 갱신 기법을 제안한다. 제안하는 색인 구조는 이동 객체의 위치 변화를 빠르게 갱신하기 위해 공간 분할 방식 색인 구조를 변형한 주 색인 구조(primary index structure)와 이동 객체가 존재하는 단말 노드를 직접 접근할 수 있는 보조 색인 구조(secondary index structure)로 구성된다. 제안하는 색인 구조는 분할된 영역들 사이에 겹침이 발생하지 않도록 전체 공간을 분할하기 때문에 객체들이 존재하는 최소 영역을 생성하기 위한 시간을 제거할 수 있다. 또한, 새로운 이동 객체를 삽입하기 위한 위치를 판별하는 것이 단순하기 때문에 이동 객체를 삽입하는 시간을 절약할 수 있다. 이동 객체의 갱신 비용을 감소시키기 위해 실제 이동 객체의 위치를 저장하는 단말 노드를 직접 접근하기 위한 보조 색인 구조를 사용하고 각 노드에는 부모 노드에 대한 포인터를 유지하여 이동 객체의 갱신으로 노드의 변경이 발생할 경우 색인 구조 전체를 순회하지 않고 상향식으로 갱신을 수행한다. 또한, 노드의 팬아웃을 증가시키기 위해 중간 노드에는 분할된 실제 영역을 표현하는 것이 아니라 필요한 분할 정보를 kd-트리로 표현한다. 분할로 인해 색인 구조의 크기가 증가되는 문제점을 해결하기 위해 오버플로우가 발생한 노드에 존재하는 일부 객체들을 형제 노드에 삽입하는 병합 분할을 수행한다.

논문의 나머지 구성은 다음과 같다. 2장에서는 미래 위치 검색을 위해 기존에 제안된 색인 구조와 색인 구조의 갱신을 효율적으로 처리하기 위한 색인 구조에 대해 기술한다. 3장에서는 기존 연구의 문제점을 해결하기 위한 새로운 색인 구조 및 갱신 기법을 제안한다. 4장에서는 제안하는 색인 구조 및 갱신 기법의 우수성을 입증하기 위해 성능 평가를 수행한 결과를 기술하고 5장에서는 논문의 결론 및 향후 연구에 대해 기술한다.

2. 관련 연구

대용량의 이동 객체에 대한 빠른 검색을 지원하기 위해서는 이동 객체의 계속적인 위치 변화를 갱신하면서 검색 유형에 적합한 색인 구조에 대한 연구가 필수적이다[9, 11]. S. Prabhakar는 이동 객체의 변화에 따른 색인 구조에 대한 갱신 비용을 줄이기 위해 자식 노드에 포함된 이동 객체의 최대 속도를 이용하여 색인을 구성하는 VCI-트리를 제안하였다[6]. VCI-트리는 기존의 R-트리 기반 색인 구조에 v_{max}라는 값을 부가적으로 사용하여 이동 객체를 색인한다. v_{max}는 하나의 노드에 포함되는 모든 객체가 가질 수 있는 최대 속도를 나타낸다. 중간 노드의 v_{max}는 자식 노드에 대한 v_{max}의 최대 값을 나타낸다. VCI-트리는 색인을 구성하는 시점에서는 모든 객체에 대한 위치를 정확하게 유지하지만 객체들이 임의 방향으로 이동할 수 있기 때문에 특정 시점에서는 정확한 위치를 유지하지 못한다. 만약 특정 시점에서 범위 질의를 수행할 경우 객체에 대한 갱신을 수행하지 않고도 생성된 VCI-트리의 중간 노드에 존재하는 MBR을 확장하여 질의에 겹침이 발생하는 자식 노드들을 탐색할 수 있다. VCI-트리는 색인을 유지하는 특정 시간 동안 갱신을 수행하지 않고 v_{max}을 통해 MBR을 확장하여 색인을 구성하기 때문에 많은 질의 결과를 생성할 수 있다.

TPR-트리는 이동 객체의 현재 및 미래 위치를 검색하기 위해 기존 R*-트리 기반의 색인 구조를 변형한 색인 구조이다[5]. TPR-트리의 단말 노드에는 이동 객체의 위치를 저장하고 중간 노드에는 자식 노드에 포함된 모든 이동 객체를 포함하는 영역과 속도 정보를 표현한다. TPR-트리의 중간 노드에 저장되는 속도는 각 차원의 상한 영역과 하한 영역으로 이동하는 객체의 속도를 표현한다. TPR-트리는 이동 객체의 계속적인 갱신이 발생할 경우 많은 비용을 소요하며 기존 MBR 영역의 외부에 이동 객체가 삽입될 경우 영역을 재구성하기 위해 많은 시간을 소요한다. 또한, 이동 객체의 삽입은 기존 R*-트리에서 사용되는 삽입 기법을 사용하기 때문에 이동 객체의 이동성 즉, 이동 객체의 미래 위치 검색에 영향을 미치는 속도를 고려하지 않는다. 따라서, 미래 위치를 검색하는 과정에서 검색 성능을 저하시킬 수 있다.

이동 객체의 미래 위치를 검색을 지원하기 위해 제안된

TPR-트리에서 삽입과 삭제는 다차원 색인 구조로 제안된 R*-트리의 알고리즘을 사용하고 있다. 그러나 R*-트리는 고정된 객체들을 색인하기 때문에 이동 객체를 색인하기 위해서는 적합하지 못하다. Y. Tao는 검색 과정에서 접근하는 노드의 수를 감소시키기 위해 기존 TPR-트리의 삽입과 삭제 알고리즘을 개선한 TPR*-트리를 제안하였다[8]. 기존 TPR-트리는 새로운 이동 객체의 위치를 삽입하기 위해서 하나의 노드에서 최적의 조건을 만족하는 하나의 자식 노드를 선택한다. 그러나 TPR*-트리는 새로운 이동 객체를 삽입할 위치를 판별하기 위해 큐를 이용하여 루트 노드에서부터 자식 노드를 삽입하기에 적절한 노드의 우선 순위에 따라 노드를 기록하여 새로운 이동 객체를 삽입할 자식 노드를 판별한다.

이동 객체의 미래 위치를 검색하기 위해 제안된 색인 구조들은 이동 객체의 계속적인 위치 변화를 갱신하기 위해 많은 시간을 소요하여 색인 구조의 성능이 급격히 저하되는 문제점이 있다. 이러한 문제점을 해결하기 위해 Y. Xia는 R*-트리와 사분 트리를 결합한 Q+R-트리를 제안하였다[12]. 사분 트리는 R*-트리 기반 색인 구조에 비해 검색 성능은 저하되지만 갱신을 빠르게 처리할 수 있다는 장점이 있다. Q+R-트리는 빠르게 이동하는 객체들은 사분 트리에 저장하고 특정 지역에서 느리게 이동하는 객체들을 R*-트리에 저장한다. Q+R-트리는 사분 트리를 이용하여 이동 객체에 대한 갱신을 효과적으로 수행할 수 있지만 이동 객체의 위치가 특정 영역 내에 편향된 경우 색인 구조의 크기가 커지고 불균형 트리가 되어 검색 성능이 저하되는 문제점을 가진다. 또한, 이동 객체의 위치 변화가 매우 클 경우 R*-트리와 사분 트리에서 삭제와 삽입 과정을 반복하기 때문에 갱신 비용을 증가시킬 수 있다는 문제점이 있다.

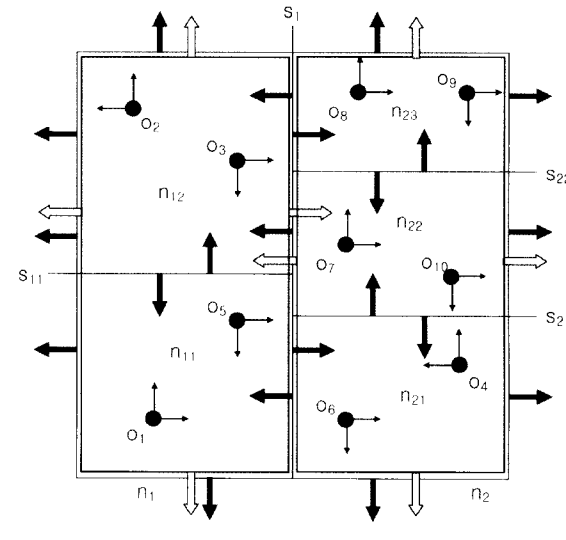
3. 이동 객체 갱신을 위한 시공간 색인 구조

3.1 제안하는 색인 구조

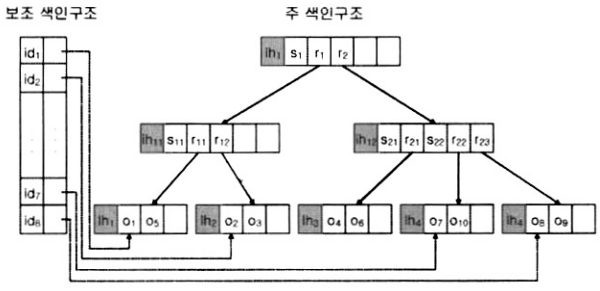
제안하는 색인 구조는 이동 객체의 계속적인 위치 변화에 대한 갱신을 효과적으로 수행하면서 현재 및 미래 위치를 검색하기 위한 색인 구조이다. 기존 R-트리 기반의 색인 구조는 이동 객체가 삽입되거나 이동 객체의 위치 변화에 따라 갱신을 수행할 때 이동 객체들을 포함하는 최소 영역을 생성하기 위해 많은 시간을 소요하며 노드들 사이에 겹침 영역이 발생하기 때문에 이동 객체를 삽입하거나 갱신하기 위한 최적의 노드를 선택하기 위해 많은 시간을 소요한다. 이러한 문제점을 해결하기 위해 제안하는 색인 구조는 분할된 영역들 사이에 겹침이 발생하지 않도록 공간 분할 방식으로 색인을 구성한다. 따라서, 분할된 노드에 존재하는 객체들을 포함하는 최소 영역을 생성하기 위한 비용을 제거할 수 있다. 뿐만 아니라, 새로운 이동 객체를 삽입하기 위한 위치를 판별하는 것이 단순하기 때문에 이동 객체를 삽입하는 시간을 절약할 수 있다.

이동 객체의 위치 변화를 갱신하기 위해 이동 객체의 이전 위치를 삭제하고 새로운 위치를 삽입하는 과정을 수행하기 때문에 많은 갱신 비용을 소요한다. 만약 이동 객체의 이전 위치가 저장되어 있는 단말 노드를 직접 접근할 수 있다면 이동 객체의 이전 위치를 검색하기 위한 시간을 절약할 수 있다. 따라서, 제안하는 색인 구조에서는 이동 객체의 계속적인 위치 변화에 따른 갱신 비용을 감소시키기 위해 실제 이동 객체의 위치를 저장하는 단말 노드를 직접 접근하기 위한 보조 색인 구조를 사용한다.

제안하는 색인 구조는 공간 분할 방식의 색인 구조로써 (그림 1)과 같은 분할 영역에 대해 (그림 2)와 같이 구성된다. (그림 1)에서 s_i는 분할 위치, 분할 영역에 표시된 검은색의 화살표는 단말 노드에 대한 속도 그리고 흰색 화살표는 중간 노드에 대한 속도를 나타낸다. (그림 1)의 분할 영역은 2차원 데이터 공간에 대해 s_1을 기준으로 분할된 중간 노드 n_1과 n_2을 생성한다. 또한, n_1는 s_{11}에 의해 분할된 단말 노드 n_{11}과 n_{12}을 생성하고 n_2는 s_{21}과 s_{22}에 의해 분할된 단말 노드 n_{21}, n_{22}, n_{23}을 생성한다. 제안하는 색인 구조는 (그림 2)와 같이 주 색인 구조와 보조 색인 구조로 구성되어 있다. 주 색인 구조는 위치 변화에 따른 노드의 재구성 시간을 감소시키기 위해 공간 분할 방식 색인 구조를 변형한 높이 균형 트리로 이동 객체의 위치 검색을 위해 실제 이동 객체의 위치를 저장한다. 주 색인 구조의 중간 노드에는 자식 노드에 존재하는 이동 객체를 포함하는 영역과 함께 이동 객체의 위치 위치를 검색하기 위한 부가적인 정보를 표현한다. (그림 2)에서 lh_i과 ih_i는 단말 노드와 중간 노드의 헤더를 나타내며 r_i는 자식 노드 n_i에 대한 위치 검색을 위한 부가적인 정보를 나타낸다. 주 색인구조는 각각의 중간 노드를 자식 노드에 대한 분할 영역을 나타내기 위해 kd-트리 형태로 구성된다. 보조 색인 구조는 갱신 비용을 감소시키기 위해 이동 객체의 위치가 저장되어 있는 주 색인 구조의 단말 노드를 직접 접근하기 위해 색인 구조이며 해쉬 테이블로 구성된다.



(그림 1) 분할 영역의 예

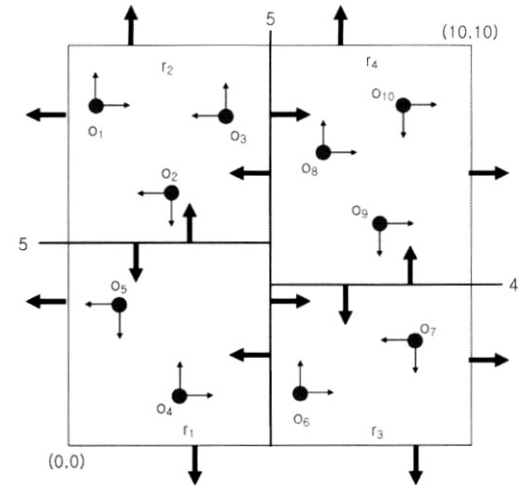


(그림 2) (그림 1)에 대한 제안된 색인 구조

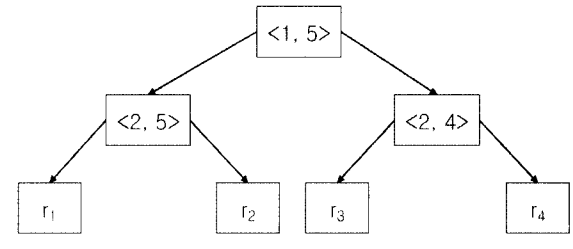
3.2 노드 구조

각 노드에 존재하는 헤더는 공간 분할에 의해 생성된 노드의 영역과 이동 객체의 갱신을 상향식으로 수행하기 위해 $\langle sp, pptr \rangle$로 구성되어 있다. sp는 공간 분할 방식에 의해 생성된 각 노드의 영역 $\langle sp^-, sp^+ \rangle$이다. 이때, sp^-는 d 차원 공간에 대해 각 차원의 하한 값 $< sp_1^-, sp_2^-, ..., sp_d^- >$이고 sp^+는 각 차원의 상한 값 $< sp_1^+, sp_2^+, ..., sp_d^+ >$이다. $pptr$은 부모 노드에 대한 포인터로 노드에 존재하는 이동 객체의 갱신으로 노드에 대한 변경이 발생할 경우 부모 노드를 직접 접근하여 상향식으로 갱신을 하기 위해 사용된다. 단말 노드에 존재하는 엔트리는 이동 객체 o_i의 실제 위치 정보 $\langle id, t_{upd}, p_{upd}, v_{upd} \rangle$을 저장한다. 이때, id는 이동 객체 o_i에 부여된 고유한 식별자, t_{upd}는 이동 객체의 갱신 시간, p_{upd}는 t_{upd} 시점의 이동 객체의 위치 정보, v_{upd}는 t_{upd} 시점의 속도이다.

중간 노드의 헤더를 제외한 나머지 부분은 자식 노드를 접근하기 위한 분할 영역을 표현하기 위해 하나의 kd-트리로 구성된다. 기존 색인 구조는 중간 노드에 d 차원 공간에서 분할된 영역을 표현하기 위해 각 차원 i의 상한 값과 하한 값에 의해 표현되기 때문에 $2d$의 크기를 갖는다. 만약 중간 노드에 분할된 영역이 m개가 존재한다고 할 때 중간 노드에 존재하는 영역 정보를 표현하기 위해서는 $2dm$ 만큼의 공간을 소요한다. 그러나 제안하는 색인 구조는 중간 노드에 실제 분할된 영역을 표현하는 것이 아니라 분할을 수행한 정보를 kd-트리로 표현하므로 d 차원의 데이터 공간에서 m개의 분할 영역이 존재한다고 할 때 $m-1+\alpha$의 공간을 소요한다. 이때, α는 분할 차원을 표현하기 위해서 사용되는 정보를 나타낸다. 예를 들어, (그림 3)의 (a)와 같이 각 노드에 최대 3개의 객체를 저장되고 o_1에서 o_{10}까지 이동 객체에 의해 4개의 분할된 노드 $n_i\ (i=1,2,3,4)$가 생성되었다고 가정하자. (그림 3)의 분할된 노드들은 먼저 1차원의 5인 위치에 의해 분할되고 각 분할된 노드들은 다시 2차원의 5와 4인 위치에 의해 분할되었다. 이때, 중간 노드의 헤더에 존재하는 sp에는 (0, 0, 10, 10)이 저장되어 있다. 분할된 4개의 노드 n_i에 대한 분할 정보를 표현하면 (그림 3)의 (b)와 같다. (그림 3)의 (b)에서 $r_i\ (i=1,2,3,4)$는 kd-트리의 단말 노드로 분할된 노드 n_i을 접근하기 위한



(a) 중간 노드의 분할 영역



(b) 분할 정보

(그림 3) 중간 노드의 분할 정보

포인터와 현재 및 미래 위치를 접근하기 위한 정보로 구성되어 있다. 중간 노드에 저장되는 kd-트리는 (그림 3)에서 보는 것과 같이 kd-트리는 이진 트리 형태로 구성되며 이러한 이진 트리의 자식 노드를 접근하기 위해서는 포인터가 필요하다. 이러한 포인터를 제거하기 위해 제안하는 색인 구조에서는 kd-트리를 깊이 우선 또는 넓이 우선 방식에 의해 트리를 순회한 순서대로 저장한다.

kd-트리의 중간 노드는 $\langle \dim, pos \rangle$로 구성되며 $\dim$는 분할 차원을 나타내고 pos는 분할 위치를 나타낸다. kd-트리의 단말 노드는 분할된 영역에 대한 위치를 검색하기 정보를 표현한다. 만약 kd-트리에 의해 분할된 영역 sp가 존재한다고 할 때, kd-트리의 단말 노드 r_i는 kd-트리에 의해 분할된 영역 sp의 정보 $\langle sp_{vec}, t_{esp}, t_{upd}, ptr \rangle$로 구성된다. 이때, sp_{vec}는 분할 영역에 대한 속도 정보, t_{upd}는 분할 영역에 대한 갱신 시간, t_{esp}는 이동 객체들이 분할된 영역을 벗어나는 시간, ptr은 자식 노드에 대한 포인터이다. 제안하는 색인 구조에서는 이동 객체의 미래 위치를 검색하는 과정에서 불필요한 노드의 확장으로 인해 검색 성능이 저하되는 문제점을 해결하기 위해 분할 영역 sp에 존재하는 이동 객체들이 노드를 벗어나는 시간 t_{esp}을 함께 저장한다. 분할된 노드에 대한 갱신 시간이 t_{upd}이고 분할된 영역을 벗어나는 시간 t_{esp}일 때, 시점 t에서 이동 객체에 대한 검색을 수행하기 위해서는 이동 객체에 대한 속도 정보

sp_{vec}을 이용하여 시점 t에서 자식 노드에 존재하는 이동 객체들이 이동할 수 있는 영역으로 노드를 확장한다. 그러나, $t_{upd}+t_{esp}\geq t$인 경우에는 자식 노드에 존재하는 이동 객체들이 분할된 노드의 영역을 벗어나지 못하기 때문에 검색을 수행하는 시점 t로 영역을 확장하지 않는다. 이로 인해 이동 객체를 검색하는 과정에서 발생하는 불필요한 노드의 확장을 감소시킬 수 있다.

3.3 삽입

제안하는 색인 구조는 공간 분할 방식을 이용하여 색인을 구성하기 때문에 분할된 영역들 사이에 겹침이 발생하지 않으며 주 색인 구조의 중간 노드에는 자식 노드에 대한 속도 정보를 저장하기 때문에 이동 객체의 미래 위치를 검색할 수 있다. 또한, 이동 객체의 미래 위치를 검색하는 과정에서 불필요한 영역의 확장을 방지하기 위해 t_{esp}와 t_{upd}을 이용하여 중간 노드에 유지한다. 제안하는 색인 구조의 삽입은 기존에 제안된 공간 분할 방식과 유사하게 수행한다. (그림 4)는 제안하는 색인 구조의 삽입 알고리즘을 나타낸 것이다.

현재 서비스를 수행 중인 이동 객체는 위치 변화를 주기적으로 전송한다. 특정 시간 동안 위치 변화를 전송하지 않는 객체는 장애가 발생하거나 서비스를 중지한 것으로 판별한다. 서비스를 중지한 이동 객체를 삭제하기 위해 명시적으로 삭제 연산을 수행하지 않는다. 그러나 서비스를 중지한 이동 객체를 삭제하지 않을 경우 검색 성능이 저하될 수 있다. 이동 객체는 주기적으로 위치 변화를 갱신하기 때문에 이동 객체의 삽입이나 갱신을 수행하기 위해 단말 노드를 접근하면 *DeleteExpObjects*()를 수행하여 서비스를 중지한 이동 객체를 삭제한다. 이러한 삭제 과정은 독립적인 연산으로 수행되는 것이 아니라 삽입이나 갱신 과정에서 수행되기 때문에 삭제 연산을 위한 색인 구조의 접근을 감소시킬 수 있다. *DeleteExpObjects*()는 단말 노드에 존재하는 이동 객체 o_i에 대해 현재 서버로 전송된 이동 객체 e의 갱신 시간을 비교하여 $MaxT$을 초과하도록 자신의 위치를 전송하는지 않는 이동 객체 o_i는 서비스를 중지한 것으로 판별하고 삭제한다. 이때, $MaxT$는 일정 시간 동안 위치 정보를 전송하지 않는 객체에 대해 실제 서비스를 중지한 객체를 판별하기 위한 시간이다. (식 1)는 단말 노드에서 서비스를 중지한 이동 객체를 삭제하는 조건을 나타낸 것이다.

$$(t_{upd}(e)-t_{upd}(o_i))>MaxT \qquad \text{(식 1)}$$

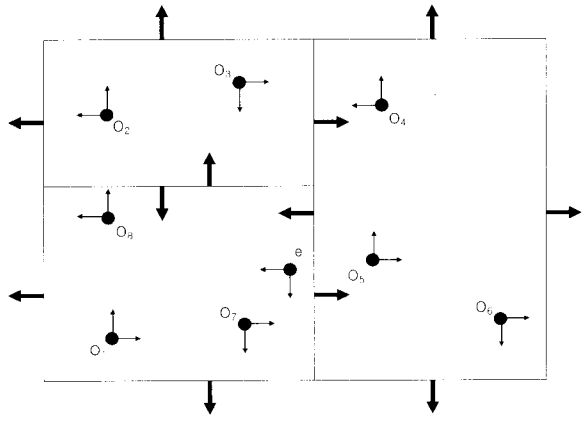
R-트리 기반 색인 구조에서는 노드에 오버플로우가 발생할 경우 분할로 인해 색인 구조의 높이가 증가되는 문제점을 해결하기 위해 오버플로우가 발생한 노드에 존재하는 일부 객체들에 대해 재삽입을 수행한다. 그러나 재삽입 연산은 많은 시간을 소요할 뿐만 아니라 재삽입 대상이 되는 이동 객체에 대해 새로운 위치를 갱신해야 할 필요가 있는 경우 재삽입 연산은 무의미해진다. 공간 분할 색인 구조는 재삽입을 수행하는 과정에서 자식 노드에 대한 분할이 발생하지 않는다면 재삽입되는 이동 객체는 다시 이전 노드에 삽입되게 된다. 따라서, 제안하는 색인 구조에서는 노드에 오

```
Algorithm Insert(e, root)
/* 이동 객체 e를 삽입할 단말 노드를 검색하여 객체를 삽입 */
{
    if(node=SearchHashTable(e)){ /* 해쉬 테이블에 객체가 존재 */
        leaf=ReadNode(node); /* 단말 노드를 읽음 */
        DeleteExpObjects(leaf); /* 서비스를 중지한 객체를 삭제 */
        UpdateNode(e, leaf, root); /* 단말 노드에서 객체의 위치를 갱신 */
    }
    else{ /* 해쉬 테이블에 이동 객체가 존재하지 않는 경우 */
        leaf=FindNode(e, root); /* 객체 e를 삽입할 단말 노드를 탐색 */
        DeleteExpObject(leaf); /* 서비스를 중지한 객체를 삭제 */
        if(CheckOverflow(leaf)){ /* 객체의 삽입으로 오버플로우가 발생 */
            pnode=ReadNode(node.pptr); /* 부모 노드를 읽음 */
            if(sibling=SearchSibling(pnode, leaf)){ /* 병합이 가능한 노드 검색 */
                sleaf=ReadNode(silbing);
                MergeSplitNode(e, leaf, sleaf); /* 병합을 수행하여 분할 */
            }
            else
                SplitNode(e, leaf); /* 분할을 수행 */
        }
        else{
            leaf=InsertObject(e, node); /* 단말 노드에 이동 객체 e를 삽입 */
            InsertHashTable(e, leaf); /* 해쉬 테이블에 e를 삽입 */
        }
    }
}
```

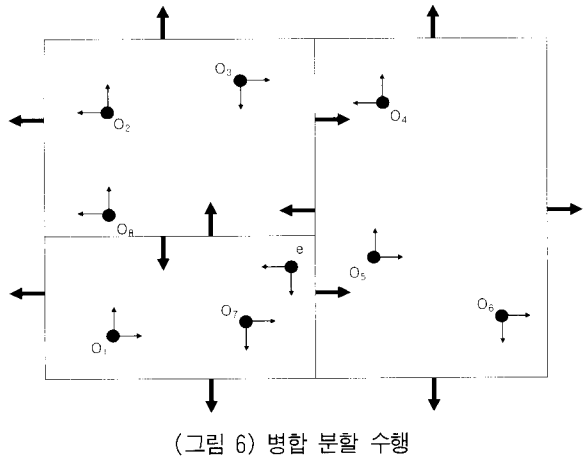
(그림 4) 삽입 알고리즘

버플로우가 발생할 경우 오버플로우가 발생한 노드와 형제 노드를 병합하고 오버플로우가 발생한 노드에 존재하는 일부 객체들을 형제 노드에 삽입하는 병합 분할(merge and split)을 수행한다. 오버플로우가 발생한 노드에 대해 병합 분할이 가능한 노드를 검색하는 *SearchSibling*()은 부모 노드에 존재하는 자식 노드들에 대한 분할 정보를 이용하여 형제 노드를 검색한다. 병합 분할이 가능한 형제 노드의 조건은 다음과 같다. 첫째, 오버플로우가 발생한 노드와 이웃하게 존재하는 노드, 둘째, 병합을 수행하여도 다른 형제 노드들과 겹침이 발생하지 않는 노드, 셋째, 객체의 삽입으로 오버플로우가 발생하지 않는 노드이다. 예를 들어, 2차원 데이터 공간에 o_1에서 o_8의 8개의 객체가 존재하고 하나의 단말 노드에 최대 3개의 객체를 저장할 수 있다고 하자. (그림 5)와 같이 새로운 객체 e가 삽입된다고 가정하면 노드에 오버플로우가 발생한다. 이러한 경우 (그림 6)과 같이 오버플로우가 발생한 노드에 존재하는 객체의 일부를 형제 노드에 재삽입하여 오버플로우를 처리할 수 있다.

제안하는 색인 구조는 이동 객체의 동적 변화 특성을 고려하여 현재 및 미래 위치를 검색할 때 탐색해야 할 노드의 수를 감소시킬 수 있도록 분할 위치를 선택해야 한다. 특히, 이동 객체의 미래 위치를 검색할 때 중간 노드에 존재하는



(그림 5) 노드에 오버플로우 발생



(그림 6) 병합 분할 수행

sp_{vec}, t_{esp}, t_{upd}을 이용하여 노드에 대한 영역을 확장시킨다. 따라서, 이동 객체의 검색에 영향을 미치는 sp_{vec}, t_{esp}, t_{upd}을 고려하여 분할을 수행한다.

단말 노드에 대한 분할 위치를 판별하기 위해서는 먼저 각 차원 i에서 분할된 영역에 대한 sp_{vec}을 최소로 생성할 수 있는 위치를 계산한다. 공간을 수행할 실제 위치 p_i를 선택은 실제 분할로 인해 생성된 노드에 대한 족하면서 t_{esp}와 t_{upd}를 최대로 생성할 수 있는 위치를 선택한다. 이러한 p_i는 (식 2)를 만족하도록 분할한다. 이때, $1 \leq m \leq 2^k - 1$이다. 분할 위치를 판별하기 위해 (식 2)를 사용하는 것은 분할된 영역에 대해 오버플로우가 발생할 경우 병합이 가능하거나 중간 노드에 분할을 수행할 수 있는 다수의 위치를 생성하기 위해서이다.

$$p_i = sp_i^- + m\frac{sp_i^+ - sp_i^-}{2^k} \qquad \text{(식 2)}$$

공간 분할 방식에서 발생할 수 있는 연속 분할(cascading split)을 수행하지 않는다. 공간 분할 방식의 색인 구조에서는 중간 노드의 분할로 인해 자식 노드에 대한 연속적인 분할을 유발할 수 있다. 이러한 문제를 해결하기 위해 가장 단순한 방법은 자식 노드들을 분할한 첫 번째 위치에 의해 중간 노드를 분할하는 것이다. 그러나 이러한 분할 방식은 노드의 공간 활용률을 저하시킬 수 있기 때문이다. 제안하는 색인 구조에서는 중간 노드의 분할을 수행하기 위해 먼저 중간 노드에 존재하는 분할 정보를 이용하여 연속 분할을 수행하지 않는 위치를 선택한다. 자식 노드에 대한 연속 분할이 발생하지 않는 위치를 선택하면 자식 노드에 대한 sp_{vec}, t_{esp}, t_{upd}를 이용하여 이동 객체의 검색을 향상시킬 수 있는 분할 위치를 판별한다.

3.4 갱신

이동 객체는 시간의 변화에 따라 계속적인 위치 변화를 갖기 때문에 이동 객체에 대한 갱신이 빠르게 처리되지 못하거나 지연될 경우 검색의 정확성을 보장할 수 없다. 뿐만 아니라, 다수 사용자를 지원하는 환경에서는 이동 객체에 대한 검색을 지연시켜 검색 성능을 저하시킨다. 제안하는 색인 구조에서는 보조 색인 구조를 통해 이동 객체가 저장된 단말 노드에 직접 접근하고 위치 변화를 갱신한다. 본 절에서는 이동 객체의 위치 변화에 대한 갱신을 효과적으로 처리하기 위해 부분 상향식 갱신(PBU : Partial Bottom-up Update) 기법과 완전 상향식 갱신(FBU : Full Bottom-up Update) 기법을 제안한다. 먼저, 부분 상향식 갱신 기법에 대해 기술하고 부분 상향식 갱신 기법을 문제점을 해결하기 위한 완전 상향식 갱신 기법을 기술한다.

3.4.1 부분 상향식 갱신 기법

부분 상향식 갱신 기법은 이동 객체의 새로운 위치를 갱신하기 위해 접근한 단말 노드의 영역 내에 이동 객체의 새

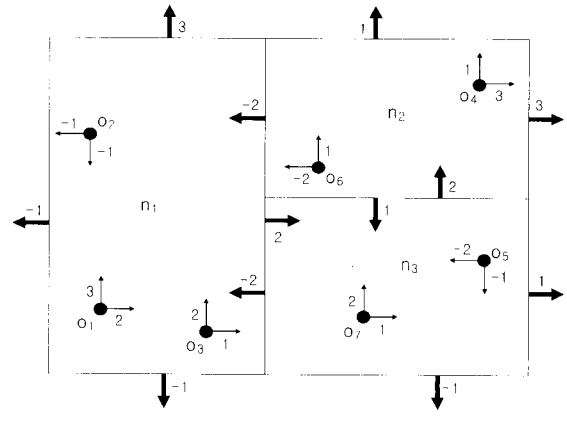
```
Algorithm PartialUpdateNode(e, leaf, root)
/* 기존 객체에 존재하는 단말 노드 node에서 객체 e의 위치를 갱신 */
{
   leafhd=ReadHeader(leaf); /* 단말 노드에 대한 헤더를 읽음 */
   if(CaluateContains(e, leafhd.sp){ /* 영역 내에 포함되는 경우 */
      newinfo=ReplacePosition(e, leaf); /* 객체의 새로운 위치를 갱신 */
      AdjustNode(leafhd.pptr, newinfo); /* 변경된 내용을 반영 */
   }
   else{ /* 객체 e의 위치가 기존 영역 내에 포함되지 않는 경우 */
      newinfo=DeleteObject(e, node); /* 객체의 이전 위치를 삭제 */
      AdjustNode(node, newinfo); /* 변경된 내용을 반영 */
      leaf=FindNode(e, root); /* 객체 e를 삽입할 단말 노드를 탐색 */
      leaf=WriteNode(e, leaf);   /* 단말 노드에 객체 e를 삽입 */
      UpdateHashTable(e, leaf); /* 해쉬 테이블을 변경 */
   }
}
```

(그림 7) 부분 상향식 갱신 알고리즘

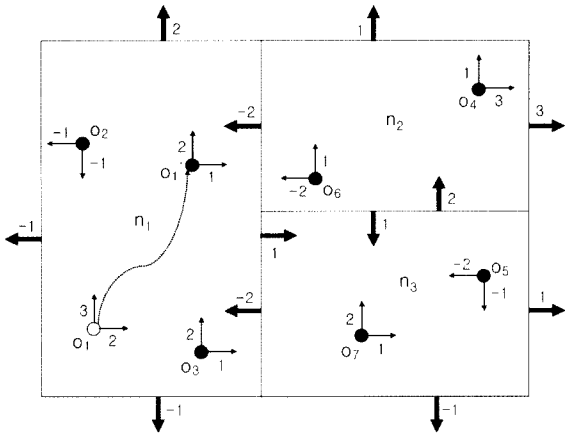
로운 위치가 존재하는 경우에는 이동 객체의 새로운 위치를 삭제하고 새로운 위치를 단말 노드에 삽입한다. 만약 이동 객체의 갱신으로 단말 노드에 대한 변경이 발생할 경우 색인 구조를 전체를 순회하지 않고 노드의 헤더에 존재하는 부모 노드에 대한 포인터를 이용하여 상향식으로 변경된 내용을 반영한다. 이에 반해, 이동 객체의 새로운 위치가 이동 객체의 이전 위치가 존재하는 단말 노드의 영역 내에 존재하지 않는 경우에는 단말 노드에 존재하는 이동 객체의 위치를 삭제하고 상향식으로 갱신을 수행한다. 즉, 주 색인 구조 전체를 순회하면서 이동 객체의 새로운 위치를 삽입하기 위한 단말 노드를 검색하고 새로운 위치를 삽입한다.

(그림 7)은 부분 상향식 갱신 알고리즘을 나타낸 것이다. 보조 색인 구조를 통해 이동 객체의 이전 위치가 저장되어 있는 단말 노드에 접근하면 객체의 새로운 위치가 단말 노드의 영역 내에 포함되는지를 판별한다. 단말 노드의 헤더에는 공간 분할에 의해 생성된 영역 정보를 포함하고 있다. *ReadHeader*()를 통해 단말 노드의 헤더를 읽어 새로운 객체의 위치가 단말 노드에 포함되는지를 판별한다. 이동 객체의 새로운 위치가 단말 노드 내에 포함되는 경우에는 *ReplacePosition*()을 수행하여 객체의 이전 위치를 이동 객체의 새로운 위치로 변경한다. 만약 객체의 갱신으로 단말 노드에 대한 sp_{vec}, t_{esp}, t_{upd}가 변경되는 경우 *AdjustNode*()를 수행하여 부모 노드에 변경된 내용을 반영한다. 만약 이동 객체의 새로운 위치가 단말 노드의 영역 내에 존재하지 않는 경우에는 *DeleteObject*()를 수행하고 이동 객체의 삭제로 단말 노드의 대한 sp_{vec}, t_{esp}, t_{upd}가 변경되는 경우 *AdjustNode*()를 수행한다. t_{upd}는 자식 노드에 대한 모든 노드가 갱신될 경우에만 변경되며 sp_{vec}과 t_{esp}는 객체의 갱신으로 인해 변경이 발생할 수 있다. *AdjustNode*()는 자식 노드에 대한 sp_{vec}, t_{esp}, t_{upd}가 변경되더라도 검색 성능에 아무런 영향을 미치지 않을 경우 갱신을 수행하지 않는다. 또한, 중간 노드에 t_{upd}는 자식 노드들의 갱신 시간 중 최소값을 유지하기 때문에 자식 노드에 존재하는 모든 객체들이 갱신되기 전까지는 t_{upd}을 부모 노드에 반영하지 않는다.

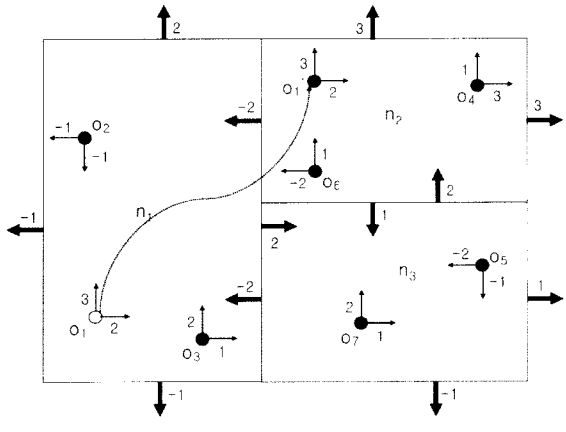
(그림 8)에서 (그림 10)은 부분 상향식 갱신을 수행하는 과정을 나타낸 것이다. (그림 8)과 같이 2차원 공간에 o_1에서 o_7까지 7개의 이동 객체가 존재하고 분할된 3개의 단말 노드 n_i가 존재한다고 가정하자. 이때, 기존에 삽입된 이동 객체 o_1가 o_1'로 갱신을 수행한다고 하자. (그림 9)와 같이 이동 객체 o_1에 대한 새로운 위치 o_1'가 이동 객체의 이전 위치가 존재하는 단말 노드 n_1에 존재하는 경우에는 이동 객체의 이전 위치를 새로운 위치를 변경하고 이동 객체의 갱신으로 변경된 노드의 정보를 부모 노드 포인터를 이용하여 부모 노드에 반영한다. (그림 9)는 이동 객체가 존재하는 n_1에 대해 이동 객체의 갱신으로 변경된 노드 정보를 반영한 것이다. 이에 반해, (그림 10)과 같이 이동 객체의 새로운 위치 o_1'가 기존 단말 노드 n_1에 존재하지 않는 경우에



(그림 8) 이동 객체에 대한 갱신



(그림 9) 이전 노드에 존재하는 경우



(그림 10) 다른 노드에 존재하는 경우

는 이동 객체의 이전 위치 o_1을 삭제하고 변경된 노드의 정보를 부모 노드에 반영한다. 또한, 색인 구조 전체를 순회하면서 새로운 위치 o_1'을 삽입할 단말 노드 n_2을 검색하여 단말 노드에 o_1'을 삽입한다. 만약 n_2에 이동 객체의 새로운 위치 o_1'의 삽입으로 노드의 정보가 변경된 경우에는 부모 노드에 반영한다. 또한, 이동 객체가 존재하는 단말 노드가 n_1에서 n_2로 변경되었기 때문에 해쉬 테이블에 이에 대한 내용을 반영한다. (그림 10)은 n_1에서 o_1을 삭제와 n_2에 o_1'을 삽입하여 변경된 노드의 정보를 반영한 것을 나타낸다.

3.4.2 완전 상향식 갱신 기법

부분 상향식 갱신 기법은 이동 객체의 새로운 위치가 기존 단말 노드 영역 내에 존재하는 경우에는 각 노드의 헤더에 존재하는 부모 노드 포인터를 이용하여 상향식 갱신을 수행한다. 그러나, 이동 객체의 새로운 위치가 기존 단말 노드 내에 존재하지 않는 경우에는 색인 구조 전체를 순회하면서 이동 객체의 새로운 위치를 삽입한다. 이러한 과정은 색인 구조 전체를 순회하기 때문에 많은 갱신 시간을 소요한다. 공간 분할 방식의 색인 구조에서는 이동 객체의 갱신되는 위치는 대부분 기존 단말 노드에 존재하거나 이웃한 형제 노드 내에 존재한다. 따라서, 색인 구조 전체를 순회하지 않고 현재 갱신을 수행하는 부모 노드를 접근하면 이동 객체의 새로운 위치를 삽입하기 적절한 노드를 검색할 가능성이 매우 높다.

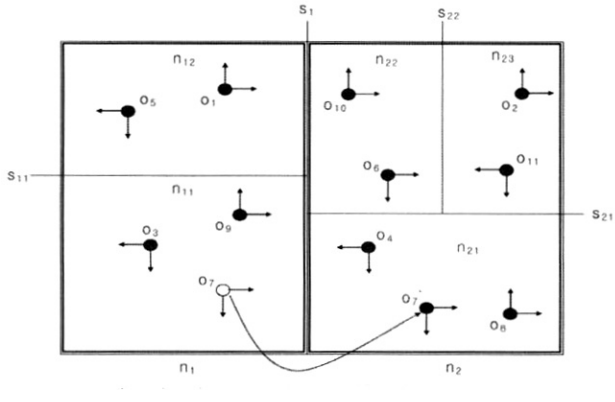
```
Algorithm FullUpdateNode(e, leaf, root)
/* 부모 노드를 읽어 객체 e를 삽입할 새로운 단말 노드를 검색 */
{
    leafhd=ReadHeader(leaf); /* 단말 노드에 대한  헤더를 읽음 */
    if(CaluateContains(e, leafhd.sp)){ /* 영역 내에 포함되는 경우 */
        newinfo=ReplacePosition(e, leaf); /* 객체의 새로운 위치를 갱신 */
        AdjustNode(leafhd.pptr, newinfo); /* 변경된 내용을 반영 */
    }
    else{ /* 객체 e의 위치가 단말 노드 영역 내에 포함되지 않는 경우 */
        newinfo=DeleteObject(e, leaf); /* 이동 객체의 위치를 삭제 */
        node=ReadNode(e, leafhd.pptr); /* 부모 노드를 읽음 */
        newinfo=ChangeNode(leaf, newinfo); /* 자식 노드의 변경된 내용을 반영 */
        while(1){
            nodehd=ReadHeader(node); /* 단말 노드에 대한  헤더를 읽음 */
            if(CaluateContains(e, nodehd.sp)){ /* 영역 내에 포함되는 경우 */
                newleaf=FindNode(e, node); /* 단말 노드를 검색 */
                leaf=WritetNode(e, newleaf); /* 단말 노드에 이동 객체 e를 삽입 */
                UpdateHashTable(e, leaf); /* 해쉬 테이블을 변경 */
                break; /* 갱신을 종료 */
            }
            node=ReadNode(e, nodehd.pptr);
            newinfo=ChangeNode(node, newinfo);
        }
        AdjustNode(nodehd.pptr, newinfo);
    }
}
```

(그림 11) 완전 상향식 갱신 알고리즘

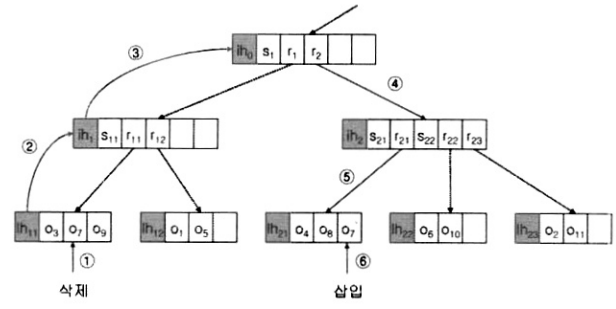
완전 상향식 갱신 기법은 이동 객체의 새로운 위치가 기존 단말 노드에 존재하지 않는 경우에도 상향식 갱신을 수행한다. 각 노드의 헤더에는 부모 노드를 접근하기 위한 헤더 정보를 포함하고 있다. 완전 상향식 갱신 기법은 이동 객체의 새로운 위치가 기존 단말 노드에 존재하지 않는 경우 노드의 헤더에 존재하는 부모 노드 포인터를 이용하여 부모 노드를 접근하고 이동 객체의 새로운 위치를 포함하는 부모 노드를 검색한다. 이동 객체의 새로운 위치를 포함하는 부모 노드를 검색하면 부모 노드의 자식 노드에 존재하는 단말 노드를 검색하여 이동 객체의 새로운 위치를 삽입한다.

(그림 11)은 완전 상향식 갱신 알고리즘을 나타낸 것이다. 완전 상향식 갱신에서는 이동 객체의 새로운 위치 e가 기존 단말 노드 영역 내에 존재하지 않는 경우에는 먼저 *DeleteObject*()를 수행하여 이동 객체의 이전 위치를 삭제한다. 이동 객체의 삭제로 노드의 정보가 변경되면 *AdjustNode*()를 수행하여 변경된 노드의 정보를 부모 노드에 반영한다. 이동 객체의 이전 위치를 삭제하면 헤더에 존재하는 부모 노드에 포인터를 통해 부모 노드를 읽고 *CaluateContains*()를 수행하여 부모 노드의 영역 내에 이동 객체의 새로운 위치가 포함되는지를 판별한다. 이동 객체의 새로운 위치가 부모 노드 내에 포함되면 *FindNode*()를 수행하여 부모 노드를 기준으로 이동 객체의 새로운 위치를 삽입할 단말 노드를 검색하고 이동 객체를 삽입한다.

예를 들어, (그림 12)와 같이 2차원 공간에 분할된 영역들이 존재할 때 이동 객체 o_7이 o_7'로 갱신을 수행한다고 하자. 이때, s_1에 의해 분할된 두 개의 중간 노드 n_1과 n_2가 존재하고 n_1 노드의 자식 노드로 n_{11}과 n_{12} 노드가 존재한다고 하자. 또한 n_2 노드의 자식 노드로 n_{21}, n_{22}, n_{23} 노드가 존재한다고 하자. (그림 13)은 완전 상향식 갱신을 수행하는 과정을 나타낸 것이다. (그림 13)에서 각 번호 ①는 완전 상향식 갱신을 처리하는 순서를 나타낸 것이다. 완전 상향식 갱신 기법에서는 o_7의 새로운 위치 o_7'가 기존 단말 노드 n_{11}에 존재하지 않는 경우에는 이동 객체의 이전 위치 o_7을 삭제하고 n_{11} 노드의 부모 노드 n_1을 접근



(그림 12) 이동 객체에 대한 갱신이 발생



(그림 13) 완전 상향식 갱신 처리 과정

하여 이동 객체의 새로운 위치 o_7'을 포함하는 노드가 존재하는지 판별한다. 그러나, 이동 객체의 새로운 위치 o_7'가 n_1 내에 존재하지 않기 때문에 부모 노드를 계속 접근하여 o_7'의 포함하는 노드를 판별한다. 이때, n_1에서 o_7과 서비스를 중지한 객체들을 삭제하여 노드의 정보가 변경되면 접근한 부모 노드에 이를 함께 반영한다. o_7'이 n_1의 부모 노드의 영역 내에 포함되기 때문에 부모 노드를 기준으로 o_7'을 삽입할 단말 노드를 검색하고 이동 객체의 위치를 삽입한다.

4. 실험 및 성능평가

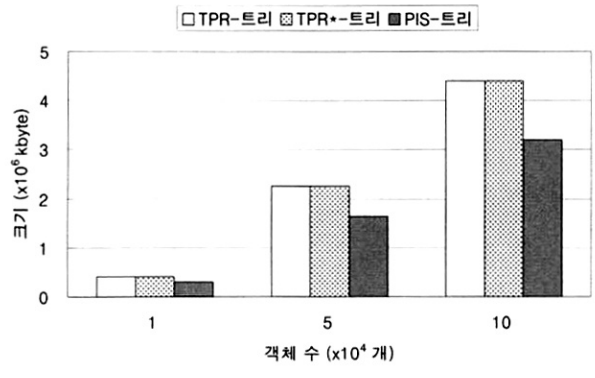
4.1 실험 환경

성능 평가를 위해 제안하는 색인 구조, TPR-트리 그리고 TPR*-트리를 펜티엄 Ⅳ 2.8GHz 프로세서에 256Mbyte의 메모리를 가지는 시스템에서 구현한다. 구현한 색인 구조에서 노드의 크기는 4kbyte이며 C언어를 통해 구현한다. 성능평가에 사용된 데이터 집합은 GSTD[13]을 통해 1000×1000㎞의 2차원 공간에서 균등(uniform), 가우시안(gaussian) 그리고 스큐(skew) 분포로 존재하는 100,000개 이동 객체를 생성한다. 또한, 각 이동 객체에 대해 1,000개의 이동 변화를 갖도록 한다. 제안하는 색인 구조의 우수성을 입증하기 위해 이동 객체의 삽입 성능, 갱신 성능 그리고 검색 성능을 비교한다. 먼저 삽입 성능은 10,000개에서 100,000개의 이동 객체를 삽입하면서 색인을 구성하는 시간을 비교한다. 갱신 성능은 100,000개의 이동 객체를 삽입하여 색인을 구성하고 1,000개에서 10,000개의 이동 객체에 대한 갱신을 수행하는 시간을 비교한다. 본 논문에서 제시하는 부분 상향식 갱신과 완전 상향식 갱신에 대한 비교도 수행한다. 마지막으로 검색 성능은 검색해야 할 미래 시간을 5에서 20분까지 변화하면서 범위 검색을 수행한 시간을 비교한다. 편의상 제안하는 색인 구조는 PIS-트리(Proposed Index Structure-tree)라 한다.

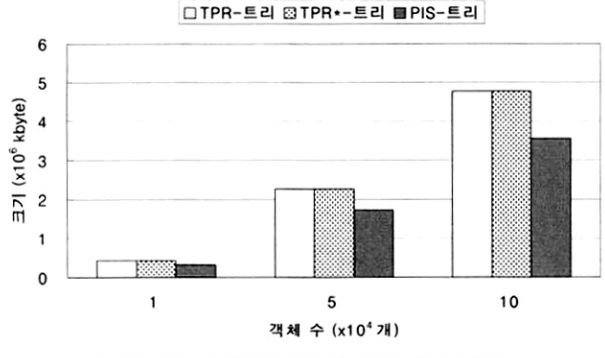
4.2 색인 구조 크기

이동 객체의 위치를 검색하는 과정에서 색인 구조의 크기가 증가할수록 탐색해야할 노드의 수가 증가되어 검색 성능

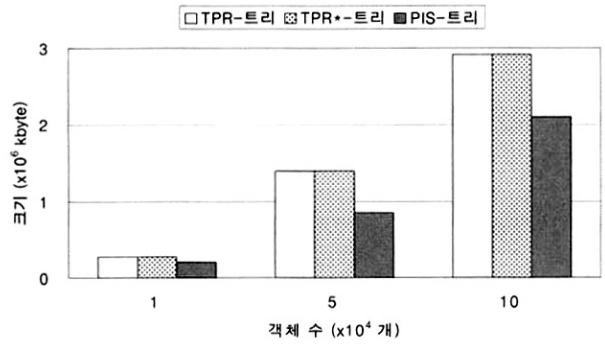
을 저하시킨다. 또한, 색인 구조의 크기가 증가할수록 이동 객체의 삽입이나 갱신 과정에서 탐색해야 할 노드의 수가 증가된다. 먼저 제안하는 색인 구조와 기존에 제안된 색인 구조에 대해 이동 객체를 삽입하여 구성된 색인 구조의 크기를 실험을 통해 비교 분석한다. 해쉬 구조로 구성된 보조 색인 구조의 크기는 주 색인 구조의 크기에 비해 상대적으로 작아 실험 평가에서 보조 색인 구조의 크기는 고려하지 않는다[10]. (그림 14)에서 (그림 16)은 이동 객체의 삽입으로 생성된 색인 구조의 크기를 나타낸 것이다. 이동 객체의 삽입으로 생성되는 색인 구조의 크기를 비교하기 위해10,000개에서 100,000개까지의 이동 객체를 삽입한다. (그림 14)는 GSDT을 통해 균등 분포 형태로 생성한 이동 객체를 삽입하여 생성한 색인 구조의 크기를 나타낸 것이다. (그림15)와



(그림 14) 균등 분포의 색인 구조 크기



(그림 15) 가우시안 분포의 색인 구조 크기



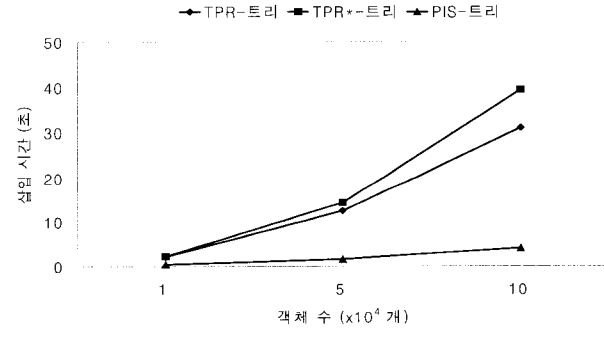
(그림 16) 스큐 분포의 색인 구조 크기

(그림 16)은 가우시안 분포와 스큐 분포 형태로 생성한 이동 객체를 삽입하여 생성한 색인 구조의 크기를 나타낸 것이다.

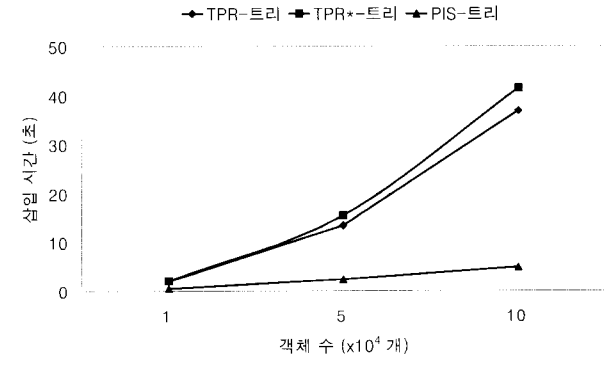
제안하는 PIS-트리에서는 이동 객체의 삽입이나 갱신으로 인해 노드에 오버플로우가 발생할 경우 재삽입을 수행하지 않고 오버플로우가 발생한 노드에 대해 부모 노드를 직접 접근하여 형제 노드와 병합을 수행한다. 이로 인해 노드의 재삽입으로 인해 소요되는 시간을 단축시킬 수 있고 분할로 인해 노드의 크기가 증가되는 문제점을 해결한다. 또한, PIS-트리는 중간 노드에 분할된 노드에 대한 실제 영역 정보를 저장하지 않고 분할 정보만을 유지하기 때문에 노드의 팬아웃을 증가시킬 수 있다. 이로 인해 색인 구조의 크기를 감소시킬 수 있다. 성능 평가 결과 제안하는 색인 구조는 기존에 제안된 TPR-트리와 TPR*-트리에 비해 색인 구조의 크기가 감소되는 것을 알 수 있다. 제안하는 PIS-트리는 기존에 제안된 색인 구조에 비해 29~58%의 성능 향상을 보인다.

4.3 삽입 성능

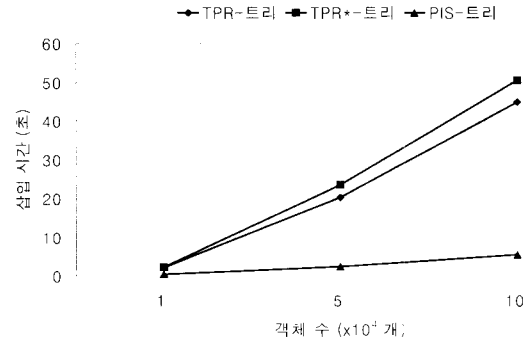
이동 객체에 대한 삽입 성능을 평가하기 위해 10,000개에서 100,000개의 이동 객체를 삽입하는 시간을 분석한다. (그림 17)에서 (그림 19)는 이동 객체에 대한 삽입 시간을 나타낸 것이다. 기존에 제안된 TPR-트리와 TPR*-트리는 삽입되는 이동 객체의 수가 증가할수록 삽입 시간이 급격히 증가되지만 제안하는 색인 구조는 삽입되는 이동 객체의 수가 증가되어도 삽입 시간에 많은 영향을 받지 않는다. 제안하



(그림 17) 균등 분포의 삽입 시간



(그림 18) 가우시안 분포의 삽입 시간



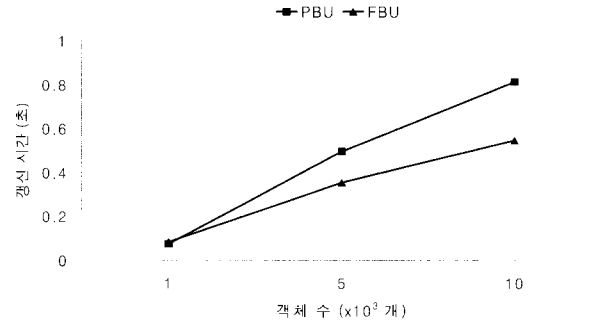
(그림 19) 스큐 분포의 삽입 시간

는 PIS-트리는 기존에 제안된 TPR-트리에 비해 450%의 성능 향상을 보이며 TPR*-트리에 비해서는 약 520%의 성능 향상을 보인다.

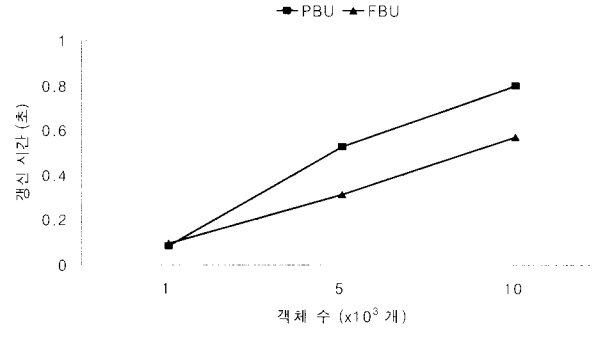
기존에 제안된 색인 구조에서는 분할된 노드에 존재하는 이동 객체를 포함하는 최소 영역을 생성하기 때문에 새로운 객체의 삽입으로 노드의 영역 정보가 변경될 경우 변경된 영역 정보를 부모 노드에 반영하기 위해 많은 시간을 소요한다. 또한, 이동 객체의 삽입으로 노드에 오버플로우가 발생할 경우 오버플로우가 발생한 노드에 존재하는 일부 객체들을 재삽입하는 과정을 수행한다. 그러나 재삽입 과정은 색인 구조를 순회해야 하기 때문에 많은 I/O를 발생시킨다. 이로 인해, 많은 삽입 시간을 소요한다. 또한, 노드들 사이에 겹침 영역이 발생하기 때문에 이동 객체를 삽입하기 위한 최적의 단말 노드를 검색하기 위해서는 많은 비용을 소요한다. 제안하는 색인 구조는 공간 분할 기반의 색인 구조이기 때문에 자식 노드에는 이동 객체의 포함하는 하나의 자식 노드만이 존재한다. 따라서, 이동 객체를 삽입하기 위한 자식 노드를 판별하기 위해 많은 시간을 소요하지 않는다. 또한, 노드의 오버플로우가 발생할 경우 재삽입을 수행하는 것이 아니라 형제 노드와의 병합을 수행을 하여 오버플로우가 발생한 노드에 존재하는 일부 객체를 형제 노드에 저장하기 때문에 오버플로우를 빠르게 처리할 수 있다.

4.4 갱신 성능

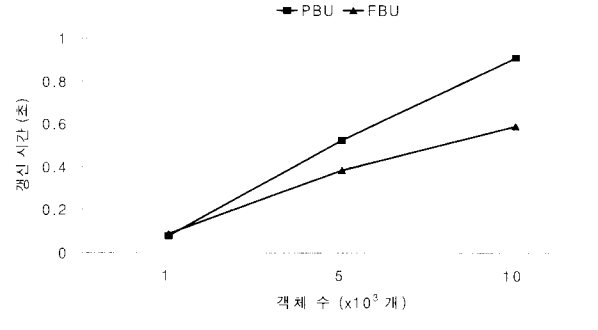
이동 객체의 계속적인 위치 변화에 따른 갱신을 효과적으로 수행하기 위해서는 이동 객체의 위치 변화에 따라 탐색해야 할 노드의 수를 감소시켜야 한다. 제안하는 색인 구조에서는 이동 객체의 갱신을 수행하기 위해 부분 상향식 갱신 PBU와 완전 상향식 갱신 FBU을 제안하였다. 먼저 제안하는 색인 구조에서 제안한 부분 상향식 갱신 기법과 완전 상향식 갱신 기법에 대한 성능 평가를 수행한다. 이동 객체에 대한 갱신 비용을 비교하기 위해 100,000개 이동 객체를 삽입하고 1,000개에서 10,000개의 이동 객체에 대한 갱신을 수행한다. (그림 20)에서 (그림 22)는 이동 객체의 분포 특성에 따른 갱신 시간을 비교한 것이다. 성능 평가 결과 1,000개의 이동 객체에 대한 갱신을 수행할 경우에는 완전 상향식 갱신이 부분 상향식 갱신 기법보다 성능 약간 저하



(그림 20) 균등 분포의 갱신 기법 비교



(그림 21) 가우시안의 갱신 기법 비교



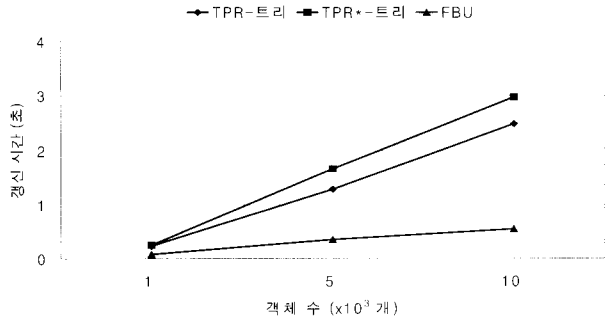
(그림 22) 스큐의 갱신 기법 비교

되는 것을 알 수 있다. 그러나 갱신되는 이동 객체의 수가 증가할수록 완전 상향식 갱신 기법이 부분 상향식 갱신 기법보다 우수하다. 상향식 갱신 기법은 1,000개의 이동 객체를 갱신할 때에는 약 10% 성능이 저하되지만 갱신되는 이동 객체의 수가 증가할수록 약 45% 성능이 향상을 보인다.

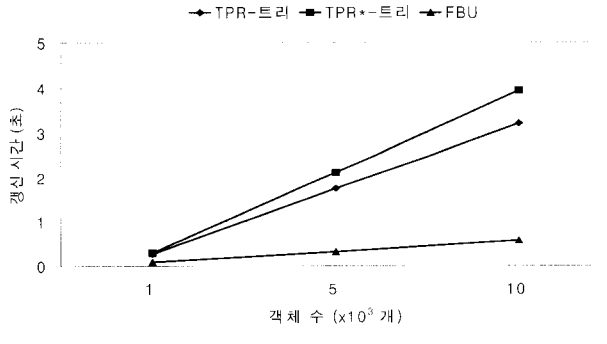
부분 상향식 갱신은 이동 새로운 위치 단말 노드의 영역 내에 포함되어 있는 경우에는 부모 노드를 직접 접근하여 상향식 갱신을 수행한다. 그러나 이동 객체의 새로운 위치가 단말 노드의 영역 내에 존재하지 않는 경우에는 색인 구조 전체를 순회하면서 하향식 갱신을 수행한다. 이에 반해, 완전 상향식 갱신은 이동 객체의 새로운 위치가 단말 노드의 영역 내에 존재하지 않는 경우에는 부모 노드를 접근하여 이동 객체의 새로운 위치를 포함할 수 있는 노드를 검색하여 갱신을 수행한다. 공간 분할 방식의 색인 구조에서는 이동 객체의 갱신을 수행한 위치는 대부분 이동 객체가 존

재하는 단말 노드의 형제 노드에 존재한다. 따라서, 색인 구조의 높이가 증가할수록 부분 상향식 갱신 기법보다는 완전 상향식 갱신 기법보다 탐색해야 할 노드의 수를 감소시킨다. 이로 인해 완전 상향식 갱신 기법이 부분 상향식 갱신 기법보다 우수한 성능을 보인다.

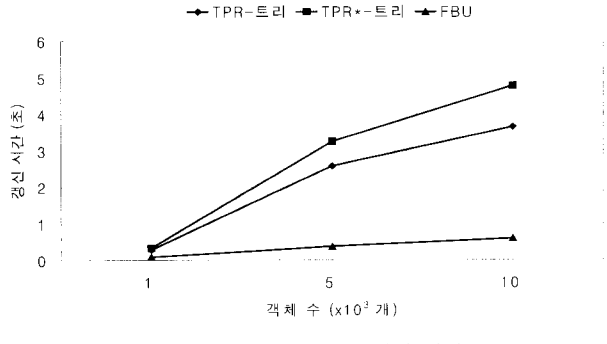
기존에 제안된 색인 구조와 갱신 비용에 대한 성능 평가를 수행하기 위해 제안하는 색인 구조와 기존에 제안된 TPR-트리 그리고 TPR*-트리와의 성능 평가를 수행한다. 제안하는 색인 구조에 대한 갱신은 완전 상향식 갱신 기법을 사용한다. 이동 객체의 갱신 기법을 비교하기 위해 100,000개의 이동 객체의 삽입하고 1,000개에서 10,000개의 이동 객체를 갱신하기 위해 소요되는 시간을 비교 분석한다. (그림 23)에서 (그림 25)는 이동 객체의 분포 특성에 따른 갱신 시간을 비교한 것이다. 기존에 제안된 색인 구조는 갱신되는 이동 객체의 수가 증가될수록 갱신 시간이 급격히 증가된다. 이에 반해, 제안하는 색인 구조는 갱신되는 이동 객체의 수가 증가되어도 갱신 시간에 많은 영향을 받지 않는다. 제안하는 PIS-트리의 완전 상향식 갱신 기법은 기존에 제안된 TPR-트리에 비해 약 340%의 성능 향상을 보이며 TPR*-트리에 비해서는 약 460%의 성능 향상을 보인다.



(그림 23) 균등 분포의 갱신 시간



(그림 24) 가우시안 분포의 갱신 시간

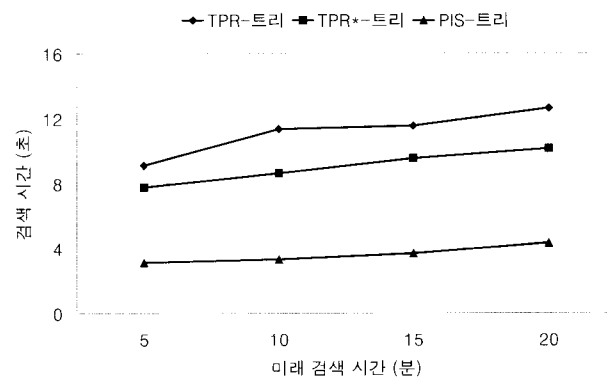


(그림 25) 스큐의 갱신 기법 시간

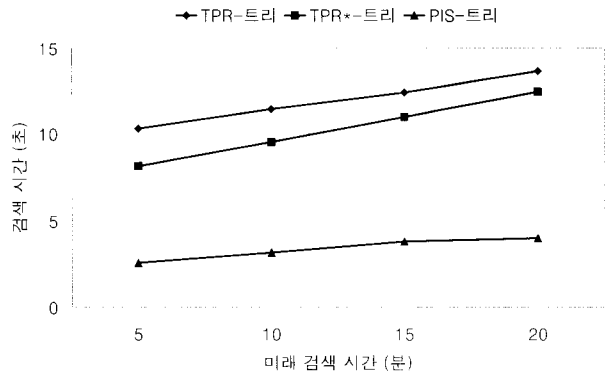
기존에 제안된 색인 구조들은 이동 객체의 이전 위치가 저장되어 있는 단말 검색하여 이전 위치를 삭제하고 이동 객체의 새로운 위치를 삽입하는 과정을 수행하면서 이동 객체의 위치를 갱신한다. 또한, 이동 객체의 갱신으로 이동 객체의 포함하는 영역이나 속도 정보가 변경될 경우 단말 노드에서부터 변경된 내용을 부모 노드에 반영하는 과정을 수행해야 한다. 이로 인해, 이동 객체의 위치를 갱신하기 위해 많은 시간을 소요한다. 제안하는 색인 구조는 이동 객체의 새로운 위치를 갱신하기 위해 이동 객체의 이전 위치가 저장되어 있는 단말 노드를 직접 접근한다. 이동 객체의 갱신으로 노드의 정보가 변경될 경우 노드의 헤더에 존재하는 부모 노드에 대한 포인터를 통해 상향식 갱신을 수행한다. 따라서, 이동 객체의 위치를 갱신하기 위해 색인 구조의 일부 노드만을 탐색하기 때문에 기존 색인 구조에 비해 갱신 비용을 감소시킬 수 있다.

4.5 검색 성능

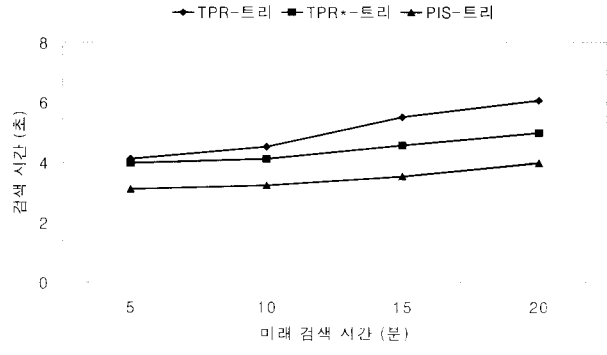
이동 객체에 대한 검색 중에서 가장 많이 사용되는 검색 유형은 범위 검색이다. 범위 검색은 현재 및 미래 시점에서 주어진 공간적인 범위 내에 존재하는 이동 객체를 검색한다. 실험에서 사용한 검색은 공간 분할 기반 색인 구조로 제안된 기존에 기법을 이용한다[14]. 제안하는 색인 구조의 검색 성능을 비교하기 위해 검색해야 할 미래 시간을 5에서 20분까지 변경하면서 범위 검색을 수행한 시간을 측정한다. (그림 26)에서 (그림 28)은 범위 검색을 수행한 시간을 나타낸 것이다. 미래 시점에 범위 검색을 수행하기 위해 검색 범위를 전체 영역에 대해 0.1%~2% 범위에 존재하는 1,000개의 검색을 수행한 평균 시간을 사용한다. 성능 평가 결과 현재 시점에서 검색한 결과와 유사하게 기존에 제안된 색인 구조들은 미래 시간이 증가될수록 검색해야 할 노드의 수가 증가되어 검색 시간이 저하됨을 알 수 있다. 이에 반해 제안하는 색인 구조는 검색 해야할 미래 시간이 증가되어도



(그림 26) 균등 분포의 미래 위치 검색



(그림 27) 가우시안 분포의 미래 위치 검색



(그림 28) 스큐 분포의 미래 위치 검색

성능의 저하가 거의 없는 것을 알 수 있다. 제안하는 색인 구조는 TPR-트리에 비해 약 180% 성능 향상을 보이며 TPR*-트리에 비해서는 약 140% 성능 향상을 보인다.

기존에 제안된 색인 구조들은 미래 시간이 증가할수록 검색 성능이 급격히 저하되는 문제점이 있다. 제안하는 색인 구조는 자식 노드에 존재하는 이동 객체의 속도 뿐만 아니라 노드가 갱신된 시간과 노드 내에 존재하는 이동 객체들이 영역을 벗어나는 시간 등을 표현하기 때문에 탐색해야 할 노드의 수를 감소시킬 수 있다. 또한, 오버플로우가 발생한 노드에 대한 직접적인 분할을 수행하지 않기 때문에 색인 구조의 크기를 감소시킬 수 있다. 이로 인해 미래 위치 검색에 대한 성능을 향상시킬 수 있다.

5. 결론 및 향후 연구

본 논문에서는 연속적인 위치 변화에 따른 이동 객체의 미래 위치 검색을 위한 색인 구조에 대한 갱신을 효과적으로 수행하기 위한 새로운 색인 구조를 제안하였다. 제안하는 색인 구조에서는 공간 분할 방식으로 색인을 구성하며 계속적인 위치 변화에 대한 갱신을 효과적으로 수행하기 위해 보조 색인 구조를 이용하여 이동 객체의 이전 위치가 저장되어 있는 노드를 직접 접근할 수 있도록 한다. 각 노드에는 부모 노드를 직접 접근하기 위한 포인터를 유지하고 이동 객체의 갱신으로 노드의 변경이 발생할 경우 색인 구조 전체를 순회하지 않고 상향식으로 갱신을 수행한다. 또한, 노드에 오버플로우가 발생할 경우 직접 분할을 수행하지 않고 형제 노드와 병합 분할을 수행한다. 성능 평가 결과 제안하는 색인 구조는 삽입 성능, 갱신 성능, 검색 성능 모두에 대해 우수함을 보였다.

향후 연구 방향으로 제안하는 색인 구조를 통해 이동 객체의 연속 질의 처리 기법에 대한 연구를 수행할 예정이며 다수의 연속 질의를 효과적으로 처리하기 위한 메모리 기반 색인 구조에 대한 연구를 수행할 예정이다.

참 고 문 헌

[1] O. Wolfson, "Moving Objects Information Management : The Database Challenge", Proc. the 5th Workshop on Next Generation Information Technologies and Systems, pp.75-89, 2002.

[2] G. Trajcevski, O. Wolfson, B. Xu and P. Nelson, "Real-Time Traffic Updates in Moving Objects Databases", Proc. the 13th International Workshop on Database and Expert Systems Applications, pp.698-704, 2002.

[3] 전봉기, 임덕성, 홍봉희, "이동체 데이터베이스를 위한 색인 기법", 데이터베이스연구회지, Vol.18, No.04 pp.23-35, 2002.

[4] M. F. Mokbel, T. M. Ghanem and W. G. Aref, "Spatio-Temporal Access Methods", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol.26, No.2, pp.40-49, 2003.

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

[6] 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.

[7] S. Saltenis and C. S. Jensen, "Indexing of Moving Objects for Location-Based Services", Proc. the 18th International Conference on Data Engineering, pp.463-472, 2002.

[8] Y. Tao, D. Papadias and J. Sun, "The TPR*-Tree : An Optimized Spatio-Temporal Access Method for Predictive Queries", Proc. the 29th International Conference on Very Large Data Bases, pp.790-801, 2003.

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

[10] M. L. Lee, W. Hsu, C. S. Jensen, B. Cui and K. L. Teo, "Supporting Frequent Updates in R-Trees : A Bottom-Up Approach", Proc. the 29th International Conference on Very

Large Data Bases, pp.608-619, 2003.

[11] M. F. Mokbel, T. M. Ghanem and W. G. Aref, "Spatio-Temporal Access Methods", Bulletin of the IEEE Computer Society Technical Committee on Data Engineering, Vol.26, No.2, pp.40-49, 2003.

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

[13] Y. Theodoridis, R. Silva and M. Nascimento, "On the Generation of Spatiotemporal Datasets", Proc. the 6th International Symposium on Spatial Databases, pp.147-164. 1999.

[14] K. S. Bok, D. M. Seo, S. S. Shin, J. S. Yoo, "TPKDB-Tree: An Index Structure for Efficient Retrieval of Future Positions of Moving Objects", Proc. Conceptual Modeling for Advanced Application Domains, pp.67-78, 2004.

복 경 수

e-mail : ksbok@dbserver.kaist.ac.kr
1998년 충북대학교 수학과(이학사)
2000년 충북대학교 정보통신공학과(공학석사)
2005년 충북대학교 정보통신공학과(공학박사)
2005. 3~현재 한국과학기술원 전산학과 Postdoc
관심분야 : 자료 저장 시스템, 이동 객체 데이터베이스, 시공간 색인 구조, 센서 네트워크 및 RFID 등

윤 호 원

e-mail : hwyoon@netdb.chungbuk.ac.kr
2005 충북대학교 정보통신공학과(공학사)
2005. 3~현재 충북대학교 정보통신공학과 석사과정
관심분야 : 이동객체 데이터베이스, 시공간 색인구조, 센서 네트워크 등

김 명 호

e-mail : mhkim@dbserver.kaist.ac.kr
1982년 서울대학교 컴퓨터공학과(공학사)
1984년 서울대학교 컴퓨터공학과(공학석사)
1989년 미국 Michigan 주립대학교 전산학과(공학박사)
1989년~현재 한국과학기술원 전산학과 교수
관심분야 : 데이타베이스, 센서네트워크, XML, 분산시스템, 워크플로우 등

조 기 형

e-mail : khjoe@cbucc.chungbuk.ac.kr
1966. 2 인하대학교 전기공학과(공학사)
1984. 8 청주대학교 산업공학과(공학석사)
1992. 2 경희대학교 전자공학과(공학박사)
1981~1988 충주공업전문대학 조교수
1996~1999 전기전자공학부장
1988~현재 충북대학교 정보통신공학과 교수
관심분야 : 데이터베이스시스템, 화상처리 및 통신, 통신 프로토콜, 분산 객체 컴퓨팅 등

유 재 수

e-mail : yjs@cbucc.chungbuk.ac.kr
1989 전북대학교 컴퓨터공학과(공학사)
1991 한국과학기술원 전산학과(공학석사)
1995 한국과학기술원 전산학과(공학박사)
1995~1996.8 목포대학교 전산통계학과 전임강사
1996.8~현재 충북대학교 정보통신공학과 교수
관심분야 : 데이터베이스 시스템, 정보검색, 멀티미디어 데이터베이스, 분산 객체 컴퓨팅 등