이동 객체의 이동성을 고려한 공간 분할 색인 기법
(A Space Partitioning Based Indexing Scheme Considering the Mobility of Moving Objects)

복 경 수 † 유 재 수 ††
(Kyoung Soo Bok) (Jae Soo Yoo)

요 약 최근 다양한 응용 분야에서 이동 객체의 현재 위치를 기반으로 미래 위치를 검색하기 위한 필요성이 증가되고 있다. 이와 함께, 대용량의 이동 객체를 빠르게 검색하기 위한 색인 구조의 필요성이 증가되고 있다. 기존에 제안된 색인 구조들은 이동 객체의 위치를 검색하는 과정에서 불필요한 노드의 확장을 유발시켜 검색 성능이 저하되는 문제점이 있다. 이러한 문제점을 해결하기 위해 본 논문에서는 객체의 이동성을 고려한 공간 분할 방식의 색인 구조를 제안한다. 제안하는 색인 구조는 노드의 오버플로우를 처리하기 위해 형제 노드와 병합 분할을 수행하고 형제 노드와 병합을 수행하여 오버플로우를 처리할 수 없을 경우에는 이동성을 고려하여 분할을 수행한다. 제안하는 색인 구조는 분할된 영역들 사이에 겹침 영역이 발생하지 않으며 속도, 이동 객체가 노드의 영역을 벗어나는 시간, 노드의 갱신 시간과 같은 파라미터를 고려하여 분할 위치를 판별한다. 중간 노드에서는 공간 분할 방식의 색인 구조에서 발생하는 연속 분할을 방지하기 위한 분할 위치를 판별한다. 제안하는 색인 구조의 우수성을 입증하기 위해 이동 객체에 대한 검색 성능을 비교 분석한다. 성능 평가 결과 제안하는 색인 구조는 현재 위치 검색에 대해서는 17%~264% 그리고 미래 위치 검색에 대해서는 107%~191% 성능 향상을 나타낸다.

키워드 : 이동 객체, 미래 위치, 공간 분할 전략, 색인 구조

Abstract Recently, researches on a future position prediction of moving objects have been progressed as the importance of the future position retrieval increases. New index structures are required to efficiently retrieve the consecutive positions of moving objects. Existing index structures significantly degrade the search performance of the moving objects because the search operation makes the unnecessary extension of the node in the index structure. To solve this problem, we propose a space partition based index structure considering the mobility of moving objects. To deal with the overflow of a node, our index structure first merges it and the sibling node. If it is impossible to merge them, our method splits the overflow node in which moving properties of objects are considered. Our index structure is always partitioned into overlap free subregions when a node is split. Our split strategy chooses the split position by considering the parameters such as velocities, the escape time of the objects, and the update time of a node. In the internal node, the split position is determined from preventing the cascading split of the child node. We perform various experiments to show that our index structure outperforms the existing index structures in terms of retrieval performance. Our experimental results show that our proposed index structure achieves about 17%~264% performance gains on current position retrieval and about 107%~191% on future position retrieval over the existing methods.

Key words : Moving Object, Future Position, Space Partition Strategy, Index Structure

· 본 논문은 2006년도 교육인적자원부 지방연구중심대학 육성사업과 한국과학재단 목적기초연구(특정기초연구 과제번호 R01-2006-000-1080900) 지원에 의해 연구되었음
† 정 회 원 : 한국과학기술원 전산학과 Postdoc 연구원
ksbok@netdb.chungbuk.ac.kr
†† 종신회원 : 충북대학교 정보통신공학과 교수
yjs@cbucc.chungbuk.ac.kr
(Corresponding author임)
논문접수 : 2005년 8월 1일
심사완료 : 2006년 8월 6일

1. 서 론

다양한 응용 분야에서 위치 기반 서비스에 대한 활용이 증가되면서 위치 기반 서비스의 대상이 되거나 서비스를 요청하는 이동 객체에 대한 관심이 집중되고 있다. 이동 객체는 시간의 변화에 따라 공간적인 위치 및 모양이 연속적으로 변하는 시공간 데이타(spatio-temporal

data)로 위치 획득 장치를 통해 자신의 위치를 파악하고 위치 변화를 서버로 전송한다[1]. 90년대 중반 이후 이동 객체를 효과적으로 저장, 관리하기 위해 이동 객체 데이타베이스(MOD : Moving Object Database)에 대한 많은 연구들이 진행되고 있다. MOD에서 대용량의 이동 객체를 효과적으로 검색하기 위해서는 연속적인 이동 객체의 변화를 색인하고 이를 통해 사용자의 요청을 빠르게 처리하기 위한 색인 구조가 필수적이다[2]. 초기에 이동 객체에 대한 검색은 대부분 과거에서 현재까지의 위치 즉, 이동 객체의 궤적을 검색하는데 초점이 맞추어져 있었다[3,4]. 그러나 최근 다양한 응용 분야에서 이동 객체의 현재 위치를 기반으로 미래 위치를 예측하기 위한 필요성이 증가되고 있다. 예를 들어, 교통 통제 시스템, 도로 안내 서비스 등에서 도로의 상태를 예측하거나 최적의 경로를 선택하기 위한 목적으로 현재 사용자의 위치를 기준으로 미래의 위치를 검색할 수 있다. 이러한 상황에서 이동 객체의 계속적인 위치 변화를 통해 미래 위치를 효과적으로 검색하기 위한 색인 구조는 필수적이다[5].

이동 객체의 미래 위치를 검색하기 위한 색인 구조는 이동 객체들이 존재하는 시공간 영역을 다른 공간 영역으로 변환하여 색인을 구성하는 변환 기반 색인 구조와 영역 내에 포함된 이동 객체에 대한 파라미터를 이용하여 색인을 구성하는 파라미터 기반 색인 구조로 구분된다[2]. 변환 기반 색인 구조는 실제 이동 객체들의 시공간 특성이 변환된 공간에서 손실될 수 있으며 변환된 공간에서는 이동 객체들 사이의 거리가 유지되지 못하는 문제점이 있다[6,7]. 이에 반해, 파라미터 기반 색인 구조는 이동 객체들이 존재하는 공간 영역과 미래 위치를 검색하기 위한 파라미터를 유지하여 색인을 구성하기 때문에 실제 이동 객체의 시공간 특성이 손실되지 않는다. 따라서, 변환 기반 색인 구조에서 발생되는 문제점을 해결할 수 있다[5]. 파라미터 기반 색인 구조는 단말 노드에 이동 객체를 저장하고 중간 노드에는 자식 노드에 포함된 이동 객체를 포함하는 영역과 미래 위치 검색을 위해 필요한 파라미터를 함께 저장한다[5,8,9,10]. 예를 들어, VCI-트리는 기존 R-트리 기반 색인 구조에 이동 객체의 미래 위치를 검색할 수 있도록 자식 노드에 존재하는 이동 객체의 가장 빠른 속도 v_{max}을 함께 저장한다[8]. 이에 반해 TPR-트리 계열의 색인 구조들은 각 차원의 상한 값과 상한 값을 기준으로 자식 노드에 포함되는 이동 객체의 속도를 저장한다[5,9]. 이러한 속도 정보는 이동 객체의 미래 위치를 검색하기 위한 정보로 이용한다.

이동 객체의 미래 위치를 검색하기 위해 제안된 기존 색인 구조들은 R-트리 기반 색인 구조를 변형하여 사용하기 때문에 분할된 영역들 사이에 겹침이 증가되어 이동 객체를 삽입하기 위한 최적의 노드를 판별하기 위해 많은 시간을 소요할 뿐만 아니라 검색 성능이 저하되는 문제점이 있다. 또한, 분할된 영역 내에 존재하는 이동 객체의 최대 속도를 중간 노드에 유지하기 때문에 미래 위치를 검색하는 과정에서 실제 이동 객체의 위치보다 더 많은 영역을 확장하여 검색 성능이 저하되는 문제점이 있다[11]. 예를 들어, 가장 빠르게 이동하는 객체가 영역에 중앙이나 반대편에서 이동하고 있을 때 이러한 객체는 특정 시간 동안 주어진 영역을 벗어나지 않는다. 그러나 미래 위치 위치를 검색하기 위해 중간 노드에서는 자식 노드에 존재하는 이동 객체의 최대 속도를 이용하여 노드를 확장하기 때문에 실제 객체들이 존재하는 영역보다 많은 영역을 확장하여 검색을 수행한다.

본 논문에서는 기존 파라미터 기반 색인 구조의 문제점을 해결하고 이동 객체의 현재 및 미래 위치 검색을 효과적으로 처리하기 위한 공간 분할 방식의 색인 구조를 제안한다. 제안하는 색인 구조는 노드에 오버플로우가 발생할 경우 분할로 인해 색인 구조의 높이가 증가되는 문제점을 해결하기 위해 오버플로우가 발생한 노드에 존재하는 일부 객체들을 형제 노드에 삽입하는 병합 분할(merge and split)을 수행한다. 병합 분할이 가능하지 않을 때에는 이동 객체의 이동성을 고려하여 미래 위치 검색을 향상시킬 수 있는 분할 위치를 선택하고 분할을 수행한다. 제안하는 색인 구조의 중간 노드는 공간 분할 방식의 색인 구조에서 발생하는 연속적인 분할(cascading split)을 방지하기 위해 연속 분할이 발생하지 않은 위치를 판별하고 이동 객체의 위치 검색을 효과적으로 수행하기 위한 분할 위치를 선택하여 분할을 수행한다. 또한, 이동 객체의 위치를 검색하는 과정에서 불필요한 노드의 확장을 감소시키기 위해 노드 내에 존재하는 이동 객체들이 영역을 벗어나는 시간을 유지하여 불필요한 노드의 확장을 제거한다.

본 논문의 나머지 구성은 다음과 같다. 2절에서는 이동 객체의 위치를 검색하기 위한 기존의 색인 구조를 살펴보고 3절에서는 기존 색인 구조의 문제점을 해결하기 위한 새로운 색인 구조를 제안한다. 4절에서는 제안하는 색인 구조의 검색 기법을 제시한다. 5절에서는 제안하는 색인 구조의 우수성을 입증하기 위해 기존 색인 구조와 다양한 성능 평가를 수행한다. 마지막 6장에서는 본 논문의 결론 및 향후 연구에 대해 기술한다.

2. 관련연구

초기에 제안된 색인 구조는 대부분 과거에서 현재까지 객체의 이동 경로 즉, 궤적을 관리하거나 현재의 위치만을 저장한다. 3DR-트리[12], HR-트리[13], HR$^+$-트

리[14], MVR3R-트리[15] 등은 과거에서 현재까지의 위치 변화를 색인하고 시공간 범위 검색을 지원한다. Y. Tao에 의해 제안된 HR-트리는 현재시점에서 하나의 객체가 변하게 되면 변경된 객체를 저장하기 위한 R-트리의 노드를 생성하여 변경된 객체를 삽입하고 과거시점에 변경되지 않은 객체는 이전 노드에 그대로 유지하는 방법을 취한다. 그러나 이러한 색인 구조들은 이동 객체의 궤적을 효과적으로 표현하지 못하기 때문에 이를 해결하기 위한 TB-트리[3], SETI-트리[16], SEB-트리[4] 등이 제안되었다. TB-트리는 기존 R-트리를 확장하여 이동 객체의 궤적을 보존하기 위해 왼쪽에서 오른쪽으로 확장되며 궤적을 보존한다. 이동 객체의 궤적을 보존하려는 TB-트리의 양방향 연결 리스트는 동일한 이동 객체의 궤적을 빠르게 검색할 수 있다는 장점이 있다.

최근 다양한 응용 분야에서 이동 객체에 대한 미래 위치를 검색하기 필요성이 증가되면서 이동 객체의 미래 위치를 검색하기 위한 색인 구조들이 제안되었다. S. Prabhakar는 이동 객체의 이동 변화에 따른 색인 구조의 갱신 비용을 줄이기 위해 이동 객체의 최대 속도를 이용하여 색인을 구성하는 VCI-트리를 제안하였다[8]. 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 나타낸 것으로 중간 노드에 v_{max}는 자식 노드에 포함된 v_{max}의 최대 값을 나타낸다. 이러한 VCI-트리는 색인을 구성하는 시점 t_0에서는 모든 이동 객체에 대한 위치를 정확하게 유지하지만 객체들이 임의 방향으로 이동할 수 있기 때문에 특정 시점 t에서는 정확한 위치를 유지하지 못하는 문제점이 있다. VCI-트리는 색인을 유지하는 특정 시간 동안 갱신을 수행하지 않고 v_{max}을 통해 *MBR*을 확장하여 검색을 수행하기 때문에 검색 성능이 저하될 수 있다.

S. Saltenis는 이동 객체의 현재 및 미래 위치를 검색하기 위한 기존 R*-트리 기반의 색인 구조를 변형한 TPR-트리를 제안하였다[5]. TPR-트리의 중간 노드에는 미래 위치를 검색하기 위해 각 차원의 상한 영역과 상한 영역으로 이동하는 객체의 속도를 표현한다. 즉, 상한 영역에 표현되는 속도는 자식 노드에 포함된 최대 속도를 유지하고 하한 영역에 표현되는 속도는 자식 노드에 포함된 최소 속도를 유지한다. 이러한 속도는 미래 위치를 검색하는 과정에서 객체들이 이동할 수 있는 최대 영역으로 계산된다. TPR-트리의 중간 노드에 저장되는 영역 정보는 항상 이동 객체를 포함하는 최소의 영역을 표현하지 못하며 기존 R*-트리의 삽입 기법을 사용하기 때문에 이동 객체의 미래 위치 검색에 영향을 미치는 속도를 고려하지 않는다.

TPR-트리에서 삽입과 삭제는 다차원 색인 구조로 제안된 R*-트리의 알고리즘을 사용하고 있다. 그러나 R*-트리는 고정된 객체들을 색인하기 때문에 이동 객체를 색인하는데는 적합하지 않다. Y. Tao는 검색 과정에서 접근하는 노드의 수를 감소시키기 위해 기존 TPR-트리의 삽입과 삭제 알고리즘을 개선한 TPR*-트리를 제안하였다[9]. 기존 TPR-트리는 새로운 이동 객체의 위치를 삽입하기 위해 하나의 노드에서 최적의 조건을 만족하는 하나의 자식 노드를 선택한다. 그러나 TPR*-트리는 새로운 이동 객체를 삽입할 위치를 판별하기 위해 큐를 이용하여 현재 노드와 자식 노드 중에서 객체 삽입을 위한 최적의 조건을 만족하는 노드를 판별한다. 그러나 분할된 영역들 사이에 겹침이 증가될수록 이동 객체를 삽입할 최적의 노드를 판별하기 위해 많은 시간을 소요하며 분할 과정에서 이동 객체의 동적인 변화 특성을 고려하지 못한다.

최근 이동 객체를 위한 색인 구조로 이동 객체의 갱신을 빠르게 처리하기 위해 그리드 기반 또는 메모리 기반 색인 구조들이 연구되고 있다. D. Kwon은 그리드(grid) 기법을 이용하여 이동 객체를 색인하기 위한 다중 레벨 해쉬 기법을 제안하였다. [17]에서는 빠르게 이동하는 객체에 대한 갱신을 효과적으로 처리하기 위해 상위 레벨에는 그리드를 크게 분할하여 색인을 구성하고 하위 레벨에는 느리게 이동하는 객체를 위해 그리드를 작게 분할하여 색인을 구성한다. X. Wang은 이동 객체의 갱신으로 발생되는 색인 구조의 변화를 감소시키기 위해 GTree라는 그리드 기반 색인 구조를 제안하였다[18]. *GTree*는 객체들이 존재하는 영역을 동일한 크기의 그리드로 분할하고 각 그리드를 하나의 노드에 매핑하여 디스크에 저장한다. GTree는 이동 객체가 존재하는 단말 노드를 빠르게 접근하기 위해 보조 색인 구조를 이용한다. 또한, X-트리의 슈퍼 노드 개념을 이용하여 형제 노드(sibling node)라는 개념을 사용한다. 이동 객체에 대한 갱신을 빠르게 처리하기 위해 GTree는 Median-Down 갱신 기법을 사용한다.

3. SPI-트리

3.1 구조

제안하는 SPI(Space Partitioned Indexing)-트리는 객체의 이동성을 고려하여 다차원 데이타를 위해 제안된 공간 분할 방식을 변형한 색인 구조이다. SPI-트리는 전통적인 다차원 색인 구조인 KDB-트리와 유사한 구조를 가지며 높이 균형 트리이다[19,20]. SPI-트리의 단말 노드는 서버로 전송된 실제 이동 객체의 위치 정

보를 저장하고 중간 노드에는 기존 KDB-트리의 중간 노드를 변형하여 분할된 영역 정보와 함께 미래 위치를 검색하기 위한 부가적인 파라미터들을 저장한다.

단말 노드에는 이동 객체의 계속적인 위치 변화에 따라 갱신된 가장 최신의 위치를 표현한다. 즉, 단말 노드에 저장되는 이동 객체 o_i의 정보 $<id, t_{upt}, p_{upt}, v_{upt}>$을 저장한다. 이때, id는 이동 객체 o_i의 식별자를 나타내며 t_{upt}는 이동 객체 o_i의 갱신 시간을 나타낸다. 단말 노드에는 서로 다른 갱신 시간을 갖는 이동 객체들을 포함하기 때문에 이동 객체는 갱신 시간 t_{upt}을 함께 표현한다. 이동 객체 o_i의 위치를 나타내는 p_{upt}는 이동 객체가 존재하는 공간 상의 위치로 $<p_1, p_2, ..., p_d>$와 같다. 이와 유사하게 이동 객체 o_i의 속도를 나타내는 v_{upt}는 $<v_1, v_2, ..., v_d>$와 같다.

그림 1은 SPI-트리의 구조를 나타낸 것이다. SPI-트리의 단말 노드는 그림 1의 (b)와 같이 동일한 계층에 존재하며 단말 노드에 저장된 이동 객체들은 보조 색인 구조에 의해 연결되어 있다. 보조 색인구조는 단말 노드에 저장된 이동 객체의 위치를 직접 접근하기 위해 사용한다. 그림 1의 (a)와 같이 4개의 단말 노드가 존재한다고 가정할 때 색인을 구성하면 그림 1의 (b)와 같다. 보조 색인 구조는 단말 노드에 저장되어 있는 이동 객체의 위치를 직접 접근하기 위해 해쉬테이블로 구성한다. 이동 객체의 위치를 갱신하기 위해서는 보조 색인 구조를 통해 이동 객체가 저장된 단말 노드를 검색하고 단말 노드를 직접 접근하여 이동 객체의 위치를 갱신한다. 그러나 새로운 이동객체의 삽입은 전통적인 삽입 기법에 의해 색인 구조를 순회하면서 이동 객체를 삽입한다. 새로운 이동 객체를 색인 구조에 삽입하면 삽입된 이동 객체에 대한 갱신을 빠르게 수행하기 위해 보조 색인 구조에 이동 객체가 삽입된 단말 노드의 정보와 이동 객체 식별자를 보조 색인 구조에 삽입한다.

SPI-트리는 이동 객체의 계속적인 위치 변화에 따른 *MBR* 영역의 재구성 시간을 제거하기 위해 공간 분할 방식에 의해 색인을 구성한다. SPI-트리의 중간 노드에는 이동 객체의 위치를 검색하기 위해 분할 영역과 함께 현재 및 미래 위치를 검색할 수 있는 정보를 표현한다. 또한, 검색 과정에서 자식 노드를 접근할 수 있도록 자식 노드에 대한 포인터를 포함한다. SPI-트리의 중간 노드는 $<sp,pv,ptr>$로 구성되어 있다. 이때, sp는 자식 노드를 포함하는 영역, pv는 위치 검색을 위해 필요한 파라미터, ptr는 자식 노드에 대한 포인터를 나타낸다. 만약 분할된 영역 sp가 존재한다고 할 때 pv는 $<sp_{vec}, t_{esp}, t_{upt}>$로 구성된다. 이때, sp_{vec}는 분할 영역에 대한 속도 정보, t_{upt}는 분할 영역에 대한 갱신 시간, t_{esp}는 이동 객체들이 분할된 영역을 벗어나는 시간을 나타낸다. 예를 들어, 그림 2와 같이 2차원의 공간에서

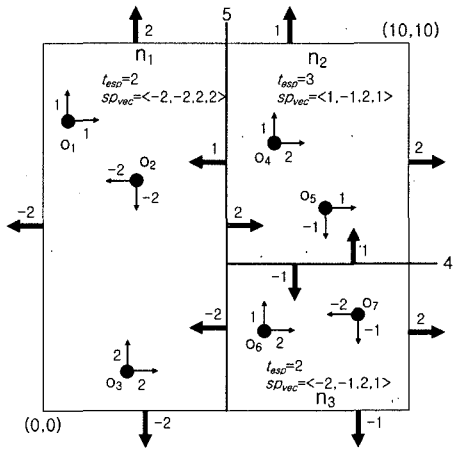
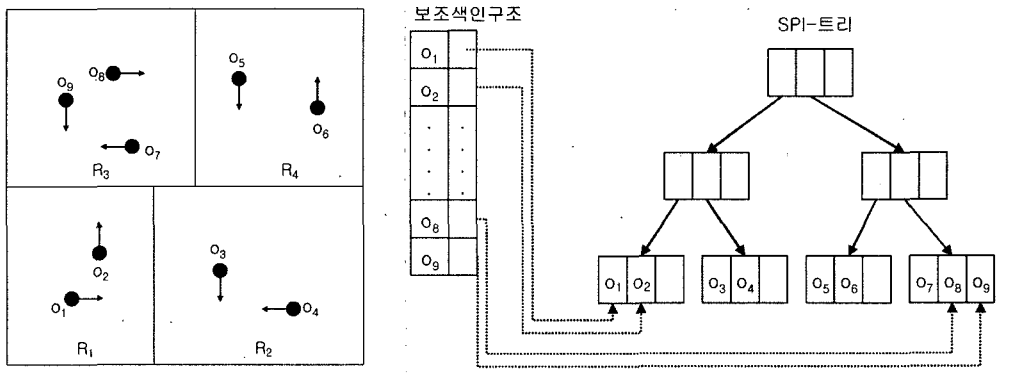


그림 2 중간 노드



(a) 단말 노드 (b) 색인 구조

그림 1 제안하는 SPI-트리 구조

1차원에 대해 5인 위치와 2차원에 대해 4인 위치에서 분할되어 n_1에서 n_3까지 3개의 영역을 생성하였다고 가정하자. 그림 2에서 o_1에서 o_7는 이동 객체를 나타내며 이동 객체에 표시된 화살표는 객체의 속도를 나타내며 분할된 외곽선에 표시된 굵은 화살표는 분할된 노드의 sp_{vec}을 나타낸다.

3.1.1 분할 영역 및 속도

중간 노드에 존재하는 sp는 공간 분할에 의해 생성된 자식 노드의 영역으로 동일 레벨에 존재하는 sp들은 겹침이 발생하지 않는다. sp는 자식 노드에 포함된 이동 객체들을 포함하기 위한 영역 $<sp^-, sp^+>$이다. 이때, sp^-는 d 차원 공간에 대해 각 차원의 하한 영역 값 $<sp_1^-, sp_2^-, ..., sp_d^->$이고 sp^+는 각 차원의 상한 영역 값 $<sp_1^+, sp_2^+, ..., sp_d^+>$이다. 이동 객체의 미래 위치 검색을 위해 sp_{vec}는 분할된 자식 노드에 포함된 이동 객체들을 포함하는 속도 정보이다. d 차원의 공간에서 분할된 자식 노드에 대한 속도 sp_{vec}는 $<sp_{vec}^-, sp_{vec}^+>$와 같다. sp_{vec}^-는 분할된 d 차원 공간에서 각 차원의 하한 영역으로 이동하는 최소 속도 $<v_1^-, v_2^-, ..., v_d^->$을 나타내고 sp_{vec}^+는 분할된 d 차원 공간에서 각 차원의 상한 영역으로 이동하는 최대 속도 $<v_1^+, v_2^+, ..., v_d^+>$이다. 각 차원 k에 대한 $v_k^- = Min\{v_k(o_i)\}$, $v_k^+ = Max\{v_k(o_i)\}$이다.

3.1.2 갱신 시간

제안하는 색인 구조는 이동 객체의 계속적인 위치 변화를 색인 구조에 반영하기 위해 갱신을 수행한다. 노드의 갱신 시간을 중간 노드에 반영하지 않을 경우 이동 객체의 위치를 검색하는 과정에서 불필요한 영역의 확장을 발생시킬 수 있다. 이로 인해, 검색해야 할 노드의 수를 증가시켜 검색 성능을 저하시킨다. 따라서, 중간 노드에는 자식 노드에 대한 갱신 시간 t_{upt}을 표현해야 한다. 이러한 t_{upt}는 단말 노드에 존재하는 이동 객체의 위치가 갱신되면 노드의 갱신 시간을 부모 노드에 반영한다. 그러나, 이러한 t_{upt}의 반영은 갱신 성능에 많은 영향을 미칠 수 있다. 따라서, 제안하는 색인 구조는 t_{upt}의 반영으로 검색 성능에 아무런 영향을 미치지 않을 경우 갱신을 수행하지 않는다. 제안하는 색인 구조는 미래 위치를 검색하기 위해 노드에 존재하는 t_{upt}과 t_{esp}을 이용한다. 만약 단말 노드의 갱신 시간을 부모 노드에 반영하지 않아도 갱신 시간 t_{upt}과 노드를 벗어나는 시간 t_{esp}의 합이 검색에 영향을 미치지 않으면 더 이상 부모 노드에 t_{upt}을 반영하지 않는다.

3.1.3 노드를 벗어나는 시간

기존에 제안된 색인 구조에서는 중간 노드에는 자식 노드에 존재하는 이동 객체들을 포함하는 속도 정보를 저장하여 미래 위치를 검색할 수 있도록 한다. 이러한 속도 정보는 분할된 영역 sp에 포함된 이동 객체들이 최대로 이동할 수 있는 속도를 저장한다. 따라서, 미래 위치를 검색하는 과정에서 중간 노드의 영역이 실제 이동 객체들의 위치보다 더 큰 영역으로 확장되어 검색 성능이 저하되는 문제점이 있다. 예를 들어, 그림 3과 같이 분할된 영역 sp가 존재하고 분할된 영역 내에 o_1에서 o_4까지 4개의 이동 객체가 존재한다고 하자. 그림 3에서 sp_1^-의 속도 v_1^-과 sp_2^+의 속도 v_2^+는 이동 객체 o_4에 의해 결정되며 sp_1^+의 속도 v_1^+과 sp_2^-의 속도 v_2^-의 속도는 이동 객체 o_1에 의해 결정된다. 따라서, 분할된 노드 sp에 대해 sp_{vec}은 $<3,-3,3,-2>$이다. 그러나, 현재 시점 t에서 sp_2^- 방향으로 이동하는 o_1의 위치가 $<4,8>$이고 o_2의 위치가 $<4,4>$라 한다면 $[t,t+4]$ 동안은 o_2가 o_1 보다 sp_2^- 방향으로 더 많은 이동을 한다. 또한, o_4의 위치가 $<9,2>$이고 o_3의 위치가 $<6,6>$라면 $[t,t+2]$ 동안은 o_3가 o_4 보다 sp_2^+ 방향으로 더 많은 이동을 한다. 즉, 이동 객체 o_1과 o_4가 분할된 영역에 대한 속도 sp_{vec}을 결정하지만 실제 이동 객체의 위치를 고려할 때 특정 시간 동안은 o_2, o_3가 특정 방향으로 더 많은 이동한다. 따라서, 이동 객체의 위치를 검색하는 과정에서 o_1과 o_4의 속도를 이용하여 sp_{vec}을 사용할 경우 실제 이동 객체의 위치보다 더 많은 영역을 확장하여 검색을 수행한다.

SPI-트리에서는 이동 객체의 미래 위치를 검색하는 과정에서 불필요한 노드의 확장으로 인해 검색 성능이

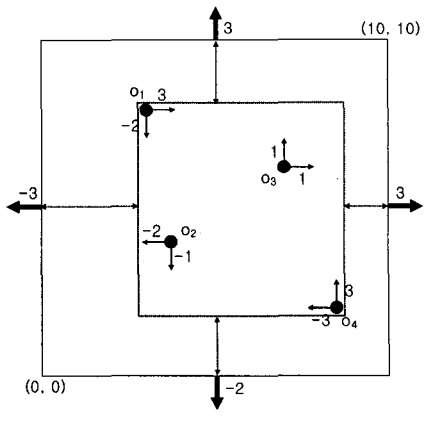


그림 3 불필요한 노드의 확장

저하되는 문제점을 해결하기 위해 분할 영역 sp에 존재하는 이동 객체들이 노드를 벗어나는 시간 t_{esp}을 함께 저장한다. 분할된 노드에 대한 갱신 시간이 t_{upt}이고 분할된 영역을 벗어나는 시간이 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 $t_{upt}+t_{esp} \geq t$인 경우에는 자식 노드에 존재하는 이동 객체들이 분할된 노드의 영역을 벗어나지 못하기 때문에 검색을 수행하는 시점 t로 영역을 확장하지 않는다. 예를 들어, 노드의 갱신 시간 t_{upt}가 5이고 노드 내에 존재하는 객체들이 노드를 벗어나는 시간 t_{esp}가 4라 가정하자. 이러한 상황에서 시점 $t=7$에서 검색을 수행한다고 하자. 이때, 노드 내에 존재하는 모든 객체들은 $t_{upt}+t_{esp}=9$일 때까지 노드를 벗어나지 않기 때문에 $t=7$ 시점에서 검색을 위해 중간 노드의 영역을 확장하지 않는다.

이동 객체 o_i가 분할된 영역 sp을 벗어나는 시간을 $t_{esp}(o_i)$라 할 때 분할된 노드에 대한 t_{esp}는 $t_{esp}=Min\{t_{esp}(o_i)\}$과 같다. 이때, $t_{esp}(o_i)$는 이동 객체 o_i가 각 차원 k에 대해 분할된 영역 sp을 벗어나는 시간 $t^k_{esp}(o_i)$의 최소 값 $Min\{t^k_{esp}(o_i)\}$과 같다. 이때, $t^k_{esp}(o_i)$는 식 (1)을 이용하여 계산한다. 식 (1)에서 $p^k_{upt}(o_i)$과 $v^k_{upt}(o_i)$는 이동 객체 o_i에 대한 k번째 차원의 위치와 속도를 나타낸다.

$$t^k_{esp}(o_i)=\begin{cases}\dfrac{sp^+_k-p^k_{upt}(o_i)}{v^k_{upt}(o_i)}+t_{upt}, & v^k_{upt}(o_i)>0\\ \dfrac{sp^-_d-p^k_{upt}(o_i)}{v^k_{upt}(o_i)}+t_{upt}, & v^k_{upt}(o_i)<0\\ \infty, & v^k_{upt}(o_i)=0\end{cases} \quad (1)$$

3.2 삽입

SPI-트리는 공간 분할 방식을 이용하여 색인을 구성하기 때문에 분할된 영역들 사이에 겹침이 발생하지 않으며 중간 노드에는 자식 노드에 대한 속도 정보를 저장하여 이동 객체의 미래 위치를 검색할 수 있다. 또한, 이동 객체의 미래 위치를 검색하는 과정에서 불필요한 영역의 확장을 방지하기 위해 t_{esp}와 t_{upt}을 중간 노드에 유지한다. SPI-트리의 삽입은 기존에 제안된 공간 분할 방식과 유사하게 수행한다. 그림 4는 SPI-트리의 삽입 알고리즘을 나타낸 것이다. 현재 서비스를 수행 중인 이동 객체의 위치는 주기적으로 서버에 전송된다. 그러나 이동 객체의 전원이 갑작스럽게 소모되거나 기기의 파손 등이 발생할 경우 이동 객체는 명시적으로 서비스를 중지한다는 것을 서버에 전송할 수 없다. 또한 이동 객체를 활용하는 대부분의 응용 서비스들은 명시적으로 이동 객체의 생명 주기를 관리하는 것이 불가능하다. 본 논문에서는 특정 시간 동안 서버로 위치를 전송하지 않는 이동 객체는 장애가 발생하거나 서비스를 중지한 것으로 판별한다. 제안하는 SPI-트리는 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 위치를 갱신하기 때문에 이동 객체의 삽입이나 갱신을 수행하기 위해 단말 노드를 접근하면 DeleteExpObjects()를 수행하여 서비스를 중지한 이동 객체를 삭제한다. DeleteExpObjects()는 단말 노드에 존재하는 이동 객체에 대해 현재 서버로 전송된 이동 객체의 갱신 시간을 비교하여 $MaxT$을 초과하도록 자신의 위치를 전송하지 않는 이동 객체 o_i는 서비스를 중지한 것으로 판별하고 삭제한다. 식 (2)는 단말 노드에서 서비스를 중지한 이동 객체를 삭제하는 조건을 나타낸 것이다.

```
Algorithm Insert(e, root)
/* 이동 객체 e를 삽입할 단말 노드를 검색하여 객체를 삽입 */
{
  stack=InitStack(); /* 탐색 경로를 저장할 스택을 초기화 */
  leaf=FindNode(e, root, stack); /* 객체 e를 삽입할 단말 노드를 탐색 */
  DeleteExpObject(leaf); /* 서비스를 중지한 객체를 삭제 */
  if(CheckOverflow(leaf)) /* 객체의 삽입으로 오버플로우가 발생 */
    TreatOverflow(e, leaf, stack); /* 오버플로우를 처리 */
  else{
    newinfo=WriteNode(e, leaf); /* 단말 노드에 이동 객체 e를 삽입 */
    AdjustNode(node, newinfo, stack); /* 변경된 내용을 부모 노드에 반영 */
  }
}
```

그림 4 삽입 알고리즘

$$(t_{upt}(e) - t_{upt}(o_i)) > MaxT \quad (2)$$

이동 객체의 삽입으로 오버플로우가 발생하여 분할을 수행할 경우 계속적인 위치 변화에 따라 분할되는 노드의 수가 증가되어 색인 구조의 크기를 증가시킬 수 있다. 또한, 이동 객체의 위치 변화가 클 경우 분할로 인해 생성된 일부 노드에 언더플로우가 발생하여 노드의 재구성 시간이 필요하다. 제안하는 SPI-트리에서는 이러한 문제점을 해결하기 위해 노드에 오버플로우가 발생하면 직접 분할을 수행하는 것이 아니라 오버플로우가 발생한 형제 노드를 검색하여 병합 분할(merge and split)을 수행한다. 병합 분할은 오버플로우가 발생한 노드와 형제 노드를 병합하고 오버플로우가 발생한 노드에 존재하는 일부 객체들을 형제 노드에 삽입한다. 만약 병합 분할이 가능한 형제 노드가 존재하지 않는 경우에는 객체의 이동성을 고려하여 분할을 수행한다. 오버플로우가 발생한 노드에 대해 병합 분할이 가능한 노드는 다음과 같다.

- 오버플로우가 발생한 노드와 이웃하게 존재하는 노드
- 병합을 수행하여도 다른 형제 노드들과 겹침이 발생하지 않는 노드
- 오버플로우가 발생하지 않는 노드

예를 들어, 2차원 데이타 공간에 o_1에서 o_8의 8개의 객체가 존재하고 하나의 단말 노드에 최대 3개의 객체를 저장할 수 있다고 하자. 그림 5와 같이 새로운 객체 e가 삽입된다고 가정하면 노드에 오버플로우가 발생한다. 현재 객체 e가 삽입된 노드와 병합이 가능한 형제 노드가 존재하지 않는다. 따라서, 이러한 노드에 대해서는 그림 6과 같이 분할을 수행한다. 이에 반해, 그림 7과 같이 객체 e가 삽입되는 경우에는 이웃한 형제 노드에 오버플로우가 발생하지 않으면서 병합 분할이 가능한 형제 노드가 존재한다. 이러한 경우에는 그림 8과 같이 오버플로우가 발생한 노드에 존재하는 객체의 일부를 형제 노드에 재삽입하여 오버플로우를 처리할 수 있다.

3.3 분할

이동 객체의 삽입으로 노드에 오버플로우가 발생하면 오버플로우가 발생한 노드의 형제 노드를 검색하여 병합 분할을 수행한다. 그러나 병합 분할이 가능한 형제 노드가 없는 경우에는 분할을 수행한다. 전통적인 다차원 색인 구조에서 공간 분할 기법은 데이타 기반 분할 기법(data dependent split strategy)과 분포 기반 분할 기법(distribution dependent split strategy)으로 구분된다[19,20]. 그러나 기존의 공간 분할 방식은 고정된 객

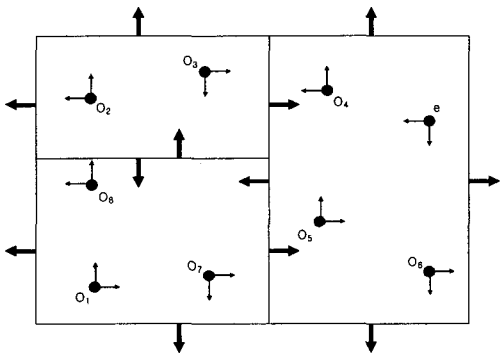


그림 5 병합 분할이 불가능 경우

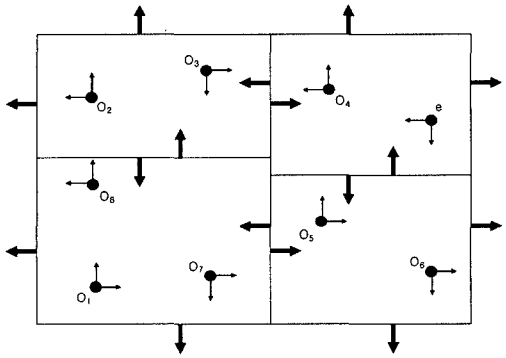


그림 6 분할을 수행

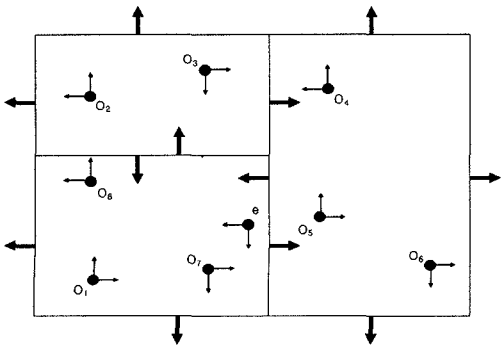


그림 7 병합 분할이 가능한 경우

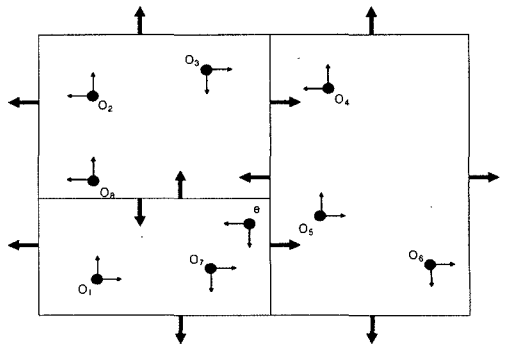


그림 8 노드의 재조정

체들만을 고려하기 때문에 동적인 이동 변화를 갖는 이동 객체를 위한 분할 기법으로 적합하지 못하다. SPI-트리는 이동 객체의 동적 변화 특성을 고려하여 현재 및 미래 위치를 검색할 때 탐색해야 할 노드의 수를 감소시킬 수 있도록 분할 위치를 선택한다. 오버플로우가 발생한 노드에서 분할을 수행하기 위한 분할 위치를 선택 기준은 다음과 같다.

첫째, 분할된 영역에 대한 미래 위치에 대한 검색을 효과적으로 수행하기 위해 분할 영역에 대한 속도 sp_{vec}을 최소로 생성할 수 있는 위치를 선택한다. 이동 객체의 미래 위치를 검색하기 위해서는 노드에 대한 영역 sp을 기준으로 자식 노드에 대한 속도 sp_{vec}에 의해 미래 시점에 확장될 영역을 계산한다. 따라서, 미래 위치에 대한 검색하는데 sp_{vec}가 가장 많은 영향을 미친다. 따라서, 오버플로우가 발생한 노드에 대한 분할을 수행하기 위한 최우선적인 기준은 분할된 영역들에 대한 속도 sp_{vec}을 최소로 생성할 수 있는 위치를 선택한다. d 차원의 공간에서 각 차원 i에 대해 분할된 영역을 low_i와 upp_i라 할 때 분할된 영역에 대한 sp_{vec}을 최소로 생성하기 위한 조건은 식 (3)과 같다. 이때, $ev()$는 각 차원에 대해 분할을 수행하여 생성된 영역에 대한 속도의 합으로 식 (4)와 같다.

$$Min\{ev(low_i)+ev(upp_i)\} \quad (3)$$

$$ev() = \sum_{i=1}^{d} |Max(v_i) - Min(v_i)| \quad (4)$$

둘째, 미래 위치를 검색하는 과정에서 불필요한 노드의 확장을 감소시키기 위해 분할된 노드에 대한 t_{esp}을 최대로 생성할 수 있는 위치를 선택한다. 이동 객체의 미래 위치를 검색하기 위해서 노드의 속도 sp_{vec}을 통해 미래 시점에 확장될 영역을 계산한다. 이때, 미래 시점이 노드의 갱신 시간 $t_{upt}+t_{esp}$을 초과하지 않는 경우에는 속도 정보를 통해 영역을 확장하지 않는다. sp_{vec}와 함께 이동 객체의 미래 위치를 검색하기 위해 많은 영향을 미치는 것은 t_{upt}와 t_{esp}이다. 일반적으로 동일한 노드에 존재하는 객체들에 대한 갱신 시간은 거의 유사한 값을 갖는다. 따라서, 분할을 수행할 경우 각 영역에 대한 t_{esp}의 차이가 t_{upt}의 차이보다 큰 값을 갖는다. 따라서, t_{esp}을 최대 값으로 생성할 수 있는 위치에 의해 분할을 수행할 경우 불필요한 영역의 증가를 제거할 수 있다. d 차원의 공간에서 각 차원 i에 대해 분할된 영역을 low_i와 upp_i라 할 때 분할된 영역에 대한 t_{esp}을 최대로 생성하기 위한 조건은 식 (5)과 같다.

$$Max\{t_{esp}(low_i)+t_{esp}(upp_i)\} \quad (5)$$

셋째, 분할된 영역 내에 존재하는 객체들의 갱신 시간 t_{upt}의 값을 최대로 생성할 수 있는 위치를 선택한다. 노드에 존재하는 갱신 시간 t_{upt}는 항상 객체의 갱신 시간보다 작거나 같다. 따라서, t_{upt}가 클수록 중간 노드의 갱신과 자식 노드에 존재하는 객체의 갱신 시간이 거의 유사한 값을 갖는다. 이로 인해 검색 과정에서 확장될 영역의 크기를 감소시킬 수 있다. d 차원의 공간에서 각 차원 i에 대해 분할된 영역을 low_i와 upp_i라 할 때 분할된 영역에 대한 t_{upt}을 최대로 생성하기 위한 조건은 식 (6)과 같다.

$$Max\{t_{upt}(low_i)+t_{upt}(upp_i)\} \quad (6)$$

넷째, 분할을 수행한 영역의 길이가 유사한 위치를 선택한다. 분할을 수행한 영역에 대한 길이가 동일할 경우 분할된 영역들에 다시 오버플로우가 발생할 때 병합이 가능한 노드를 선택할 가능성이 증가된다.

3.3.1 단말 노드 분할

단말 노드에 오버플로우가 발생하면 먼저 오버플로우가 발생한 노드에 대해 분할을 수행하기 전에 다른 노드와 병합이 가능한지를 확인한다. 만약 병합이 가능한 단말 노드가 존재할 경우에는 병합을 수행한다. 그러나 병합이 가능한 단말 노드가 존재하지 않을 경우에는 단말 노드에 분할을 수행한다. 단말 노드에는 실제 이동 객체의 위치와 속도 정보를 저장하고 있다. 이러한 이동 객체의 위치 정보와 속도 정보를 이용하여 이동 객체의 위치를 검색하기 위해 불필요한 노드의 확장을 최소로 수행할 수 있는 분할 위치를 결정한다. 그림 9는 단말 노드의 분할 알고리즘을 나타낸다.

단말 노드에 대한 분할 위치를 판별하기 위해서는 먼저 각 차원 i에서 분할된 영역에 대한 sp_{vec}을 최소로 생성할 수 있는 위치를 계산한다. 그림 10과 그림 11은 차원 i에 대해 최소 속도를 갖는 분할 위치를 판별하는 과정을 나타낸 것이다. 차원 i에 대해 정렬된 객체 o_1에서 o_6까지 6개의 객체가 존재하고 분할로 생성된 영역을 low_i와 upp_i라 하자. 분할 위치를 판별하기 위해 먼저 분할된 영역에 대해 최소로 포함되어야 할 객체 수 m을 만족하면서 분할된 영역에 대해 sp_{vec}을 최소로 생성할 위치를 계산한다. 그림 10은 m이 2라고 가정할 때 노드에 포함되어야 할 최소의 객체 수 m을 만족하면서 분할된 영역에 대한 $ev(low_i)+ev(upp_i)$가 최소가 되도록 두 개의 그룹으로 분할한 것이다. 그림 10에서 i 차원에서 low_i에 o_1, o_2, o_5을 포함하고 upp_i에 o_3, o_4, o_6을 포함할 때 분할된 영역에 대해 $ev(low_i)+ev(upp_i)$가 최소가 된다.

단말 노드에 대한 공간 분할을 수행할 실제 위치는

```
Algorithm SplitLeaf(e, leaf, leafinfo)
/* e의 삽입으로 node에서 오버플로우가 발생할 경우 분할을 수행 */
{
   node에 최소로 포함되어야 할 객체 수 m을 계산;
   node에 존재하는 객체와 새로 삽입하려는 객체 o를 list에 저장;
   for(i=1; ;i++){ /* 각 차원에 대해 */
      sortlist=SortObject(list, i) /* 각 차원에 대해 정렬 */
      splitpos[i]=LeafMinVec(sortlist, m, i); /* 최소 속도를 갖는 분할 위치 */
   }
   leafpos=LeafSplitPos(splitpos); /* 최적의 분할 위치를 선택 */
   newleaf=CreateLeaf(); /* 새로운 단말 노드를 생성 */
   /* leafpos를 기준으로 leaf와 newleaf에 객체들을 삽입 */
   AllocateObject(leaf, newleaf, leafpos);
   WriteLeafHeader(leaf, newleaf); /* 단말 노드에 헤더를 기록 */
   return newleaf;
}
```

그림 9 단말 노드 분할 알고리즘

low_i와 upp_i로 분할된 위치 r_i을 u_i을 기준으로 공간을 수행할 실제 위치 p_i을 선택한다. p_i는 분할로 인해 생성된 노드에 대해 t_{esp}와 t_{upt}을 최대로 생성할 수 있는 위치를 선택한다. 이러한 p_i는 식 (7)과 식 (8)를 만족하도록 분할한다. 이때, $1 \leq m \leq 2^k - 1$이다. 그림 11은 r_i와 u_i을 포함하는 영역 spb_i 내에서 실제 분할 위치를 선택한 것을 나타낸다. 분할 위치를 판별하기 위해 식 (8)를 사용하는 것은 분할된 영역에 대해 오버플로우가 발생할 경우 병합이 가능하거나 중간 노드에 분할을 수행할 수 있는 다수의 위치를 생성하기 위해서이다.

$$r_i \leq p_i \leq u_i \tag{7}$$

$$p_i = sp_i^- + m\frac{sp_i^+ - sp_i^-}{2^k} \tag{8}$$

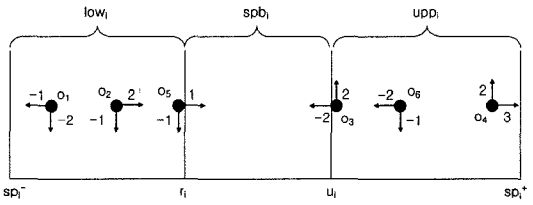


그림 10 최소의 속도로 분할

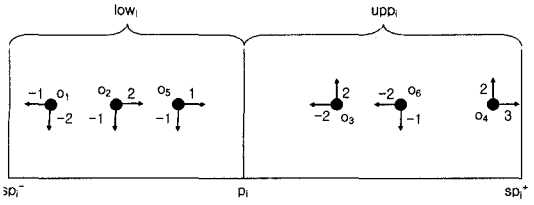
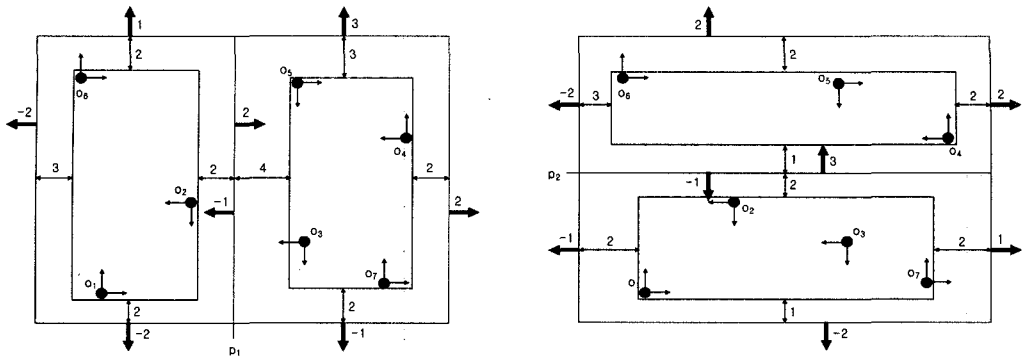


그림 11 각 차원에 대한 단말 노드의 분할 위치

각 차원 i에 대해 분할된 영역에 대한 최소 속도를 포함하는 분할 위치 p_i를 판별하면 각 차원에 대해 식 (3)를 이용하여 분할된 영역에 대한 sp_{vec}가 최소인 위치를 선택하여 분할한다. 만약 식 (3)을 만족하는 다수의 차원이 존재할 경우에는 t_{esp}, t_{upt}을 최대로 생성할 수 있는 차원을 선택하여 분할을 수행한다. 그림 12는 2차원 공간에서 단말 노드에 대한 최적의 분할 위치를 선택하는 과정을 나타낸 것이다. 각 차원 i에 대해 분할된 영역의 sp_{vec}가 최소인 분할 위치 p_i라 하자. 그림 12의 (a)는 첫 번째 차원에 대해 $ev(low_1)+ev(upp_1)$가 최소인 위치 p_1에 의해 분할된 영역을 나타낸 것이다. p_1에서 분할된 영역에 대한 $ev(low_1)+ev(upp_1)$는 14이고 $t_{esp}(low_1)+t_{esp}(upp_1)$는 4이다. 그림 12의 (b)는 두 번째 차원에 대해 $ev(low_2)+ev(upp_2)$가 최소인 위치 p_2에 의해 분할된 영역을 나타낸 것이다. p_2에서 분할된 영역에 대한 $ev(low_2)+ev(upp_2)$는 14이고 $t_{esp}(low_2)+t_{esp}(upp_2)$는 2이다. 따라서, 첫 번째 차원에 의해 분할된 영역과 두 번째 차원에 의해 분할된 영역 모두 식 (3)에 대해 동일한 값을 갖는다. 따라서, 분할된 영역에 대해 식 (5)를 만족하는 첫 번째 차원에 의해 분할을 수행한다.

3.3.2 중간 노드 분할

공간 분할 방식 색인 구조에서는 중간 노드의 분할로 인해 자식 노드에 대한 연속 분할을 유발하는 문제점이 있다[19,20]. 즉, 중간 노드의 분할로 인해 하나의 자식 노드가 두 개의 자식 노드로 분할되는 문제점이 있다. 이러한 연속 분할은 자식 노드에 대한 계속적인 분할 수행하기 때문에 많은 시간이 소요될 뿐만 아니라 분할



(a) 첫 번째 차원에 의한 분할 (b) 두 번째 차원에 의한 분할

그림 12 단말 노드의 분할

로 인해 색인 구조의 크기를 증가시킬 수 있다. 또한, 객체들이 존재하지 않는 빈 노드를 생성할 수도 있다. 전통적인 다차원 색인 구조에서 이러한 연속 분할을 제거하기 위해 자식 노드들을 분할한 첫 번째 위치에 의해 중간 노드를 분할한다[21]. 그러나 이러한 분할 방식은 연속 분할은 방지할 수 있지만 분할된 자식 노드에 존재하는 이동 객체의 동적 특성을 고려하지 못하기 때문에 검색 성능을 저하시킬 수 있다. 제안하는 색인 구조에서는 전통적인 다차원 색인 구조의 분할 기법에 대한 문제점을 해결하기 위해 새로운 분할 기법을 제안한다. 제안하는 중간 노드의 분할 기법은 먼저 중간 노드에 존재하는 분할 정보를 이용하여 연속 분할을 수행하지 않는 위치를 선택한다. 연속 분할이 발생하지 않는 위치에 대해 자식 노드에 대한 sp_{vec}, t_{esp}, t_{upt}을 이용하여 이동 객체의 검색을 향상시킬 수 있는 분할 위치를 판별한다. 연속 분할을 수행하지 않는 위치를 선택하면 연속 분할을 수행하지 않은 위치에 대해 분할로 생성되는 노드에 대한 식 (3), 식 (5) 그리고 식 (6)를 이용하여 분할된 영역에 대한 sp_{vec}을 최소로 생성하고 t_{esp}와 t_{upt}을 최대로 생성할 수 있는 위치를 분할 위치로 선택한다. 그림 13은 중간 노드의 분할 알고리즘을 나타낸다.

중간 노드에 대한 분할 위치는 중간 노드에 존재하는 분할 정보를 통해 연속 분할이 발생하지 않으면서 sp_{vec}을 최소로 t_{esp}과 t_{upt}을 최대로 생성할 수 있는 위치를 선택해야 한다. 그림 14에서 그림 16까지는 2차원 공간에서 중간 노드에 대한 분할을 수행하는 과정을 나타낸 것이다. 그림 14와 같이 분할된 6개의 영역이 존재한다고 하자. 그림 14의 분할 영역에 대해 s_3와 s_4는 중간 노드의 분할 위치를 선택할 경우 자식 노드에 대한 연속 분할을 발생시킨다. 이에 반해 s_1와 s_2는 중간 노드에 대한 분할 위치로 선택할 경우 자식 노드에 대한 연속 분할을 발생시키지 않는다. 따라서, 자식 노드에 대

```
Algorithm SplitInternal(e, node, nodeinfo)
/* e의 삽입으로 node에서 오버플로우가 발생할 경우 분할을 수행 */
{
    node에 존재하는 영역 정보와 newe를 list에 저장;
    /* 자식 노드에 대한 연속 분할이 발생하지 않는 분할 위치를 판별 */
    interpos[]=NonCascadingPos(e, list);
    for(i; ; i++){ /* interpos에 존재하는 위치에 대해 */
        splitpos[]=InterMinVec(interpos, i); /* 최소 속도를 갖는 분할 위치 */
    }
    interpos=InterSplitPos(splitpos); /* 최적의 분할 위치를 선택 */
    newnode=CreateInternal(); /* 새로운 단말 노드를 생성 */
    /* interpos를 기준으로 node와 newnode에 객체들을 삽입 */
    AllocateObject(node, newnode, interpos);
    return newnode;
    }
}
```

그림 13 중간 노드 분할 알고리즘

한 연속 분할을 발생시키지 않는 s_1와 s_2에 대해 실제 분할을 수행하기에 위한 위치를 선택한다. 그림 15는 s_1을 분할 위치로 선택하여 분할을 수행한 결과이다. 그림 15와 같이 s_1에 의해 중간 노드를 분할할 경우 $ev(low_2)+ev(upp_2)$는 19이다. 그림 16과 같이 s_2에 의해 중간 노드를 분할할 경우 $ev(low_1)+ev(upp_1)$는 18이 된다. 따라서, 첫 번째 차원 s_2을 분할 위치를 판별하여 분할을 수행한다.

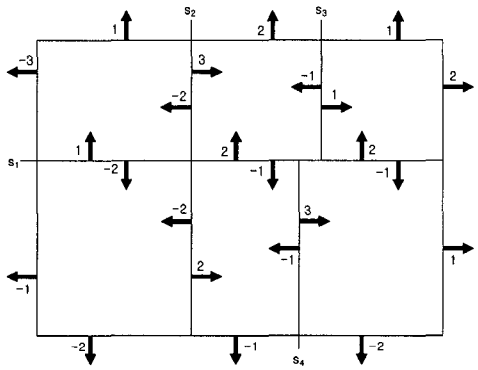


그림 14 중간 노드의 오버플로우

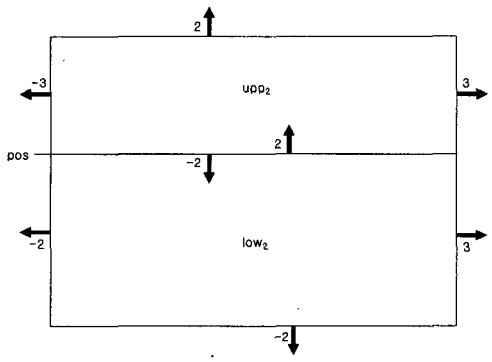


그림 15 s_1에 의한 중간 노드 분할

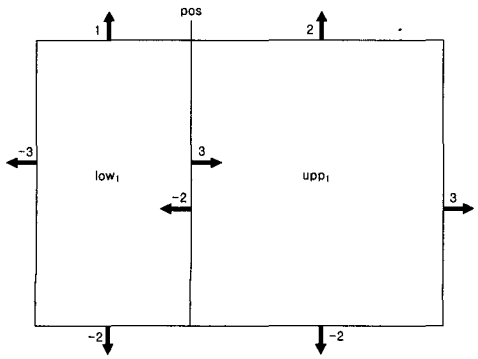


그림 16 s_2에 의한 중간 노드 분할

4. 범위 검색

제안하는 색인 구조는 이동 객체의 현재 및 미래 위치를 검색할 수 있도록 객체의 위치와 속도 정보를 이용하여 색인을 구성한다. 이로 인해, 현재 또는 미래 시점에서 특정 범위 내에 존재하는 이동 객체를 검색하는 범위 검색을 지원한다. 범위 검색은 이동 객체에 검색 중에서 가장 많이 사용되는 검색 유형으로 현재 또는 미래 시점에서 주어진 공간적인 범위 내에 존재하는 이동 객체의 검색한다. 범위 검색을 수행하기 위해서는 검색 범위 q와 함께 검색을 수행할 시간 t을 지정하여 사용자가 지정한 검색 시점 t에 주어진 범위 q 내에 존재하는 이동 객체들을 검색한다. 이동 객체에 대한 범위 검색을 수행하기 위해서는 각 노드에 존재하는 분할 정보를 통해 주어진 범위 내에 존재하는 자식 노드들을 검색한다. 그림 17은 범위 검색을 수행하는 알고리즘을 나타낸 것이다.

중간 노드에서 ReadRegion()을 통해 읽은 영역을 r_i라 할 때, OverlapRegion()은 검색 시점 t에서 검색 범위 q와 겹침이 발생할 경우 자식 노드를 탐색한다. 자식 노드 n_i의 영역 정보를 sp라 할 때, 자식 노드가 검색 시점 t에서 범위 q 내에 존재하는지는 판별하기 위해 자식 노드의 영역을 식 (9)와 식 (10)을 이용하여 검색 시점의 영역으로 노드를 확장한다. $sp_i^-(t_{upt})$와 $sp_i^+(t_{upt})$는 중간 노드에 존재하는 분할 영역의 하한 영역과 상한 영역이고 $sp_i^-(t)$와 $sp_i^+(t)$는 t에서 중간 노드의 하한 영역과 상한 영역이다. 만약 식 (9)와 식 (10)을 이용하여 검색 시점 t로 확장한 영역이 검색 범위와 겹침이 발생할 경우에는 자식 노드에 대한 포인터 ptr을 이용하여 자식 노드를 탐색한다.

$$sp_i^-(t)=\begin{cases} sp_i^-(t_{upt}) & , (t_{upt}+t_{esp}) \geq t \\ sp_i^-(t_{upt})+(t-(t_{upt}+t_{esp}))v_{i_{upt}}^-, & (t_{upt}+t_{esp})<t \end{cases} \quad (9)$$

$$sp_i^+(t)=\begin{cases} sp_i^+(t_{upt}) & , (t_{upt}+t_{esp}) \geq t \\ sp_i^+(t_{upt})+(t-(t_{upt}+t_{esp}))v_{i_{upt}}^+, & (t_{upt}+t_{csp})<t \end{cases} \quad (10)$$

예를 들어, 그림 18의 (a)와 같이 분할된 세 개의 노드 n_i가 존재하고 t가 3인 시점에서 범위 검색을 위한 q가 주어져 있다고 하자. 분할된 n_i에 대한 t_{upt}와 t_{esp}가 표 1과 같다. 분할된 노드 n_1는 $t_{upt}+t_{esp}>t$이기 때문에 t 시점에서 범위 검색을 수행하기 위해 노드를 확장할 필요가 없다. 이에 반해, n_2와 n_3는 $t_{upt}+t_{esp}<t$이기 때문에 t 시점에서 범위 검색을 수행하기 위해 노드를 확장해야 한다. 그림 18의 (b)는 t 시점에서 범위 검색을 수행하기 위해 확장된 영역을 나타낸 것이다. n_2

```
Algorithm RangeSearch(q, t)
/* 객체 o에 대한 t 시점의 위치를 검색 */
{
  node=ReadNode(root); /* 루트 노드를 읽음 */
  while(1){
    if(node==LEAF){
      objlist[]=ReadObject(leaf); /* 객체의 위치 정보를 읽음 */
      for(i=0; ;i++){/* objlist에 존재하는 객체들에 대해 */
        p=CalculatePos(objlist[i], t); /* 이동 객체의 위치 계산 */
        if(ObjectContains(p, q)) /* 검색 범위 q 내에 존재하는 경우 */
          result[]=objectlist[i];
      }
    }
    else{
      sp[]=ReadRegion(node); /* 노드에 대한 영역을 읽음 */
      for(i=0; ; i++){
        reg=CalculateRegion(sp[i], t); /* 검색 시점으로 영역을 확장 */
        if(OverlapRegion(reg, q)) /* 검색 범위 q와 겹침이 발생 */
          RangeSearch(q, t, sp[i].ptr);
      }
    }
    return result;
  }
}
```

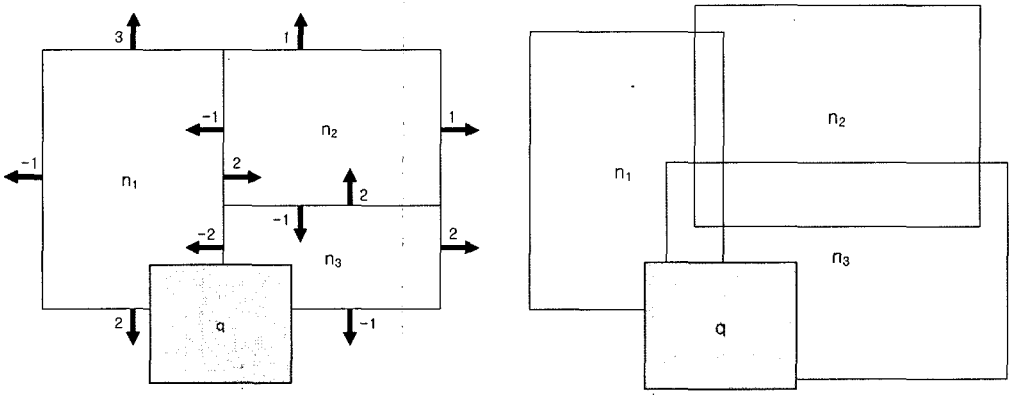
그림 17 범위 검색 알고리즘

표 1 분할 영역에 대한 정보

분할 영역	t_{upt}	t_{esp}
n_1	2	2
n_2	1	1
n_3	1	0

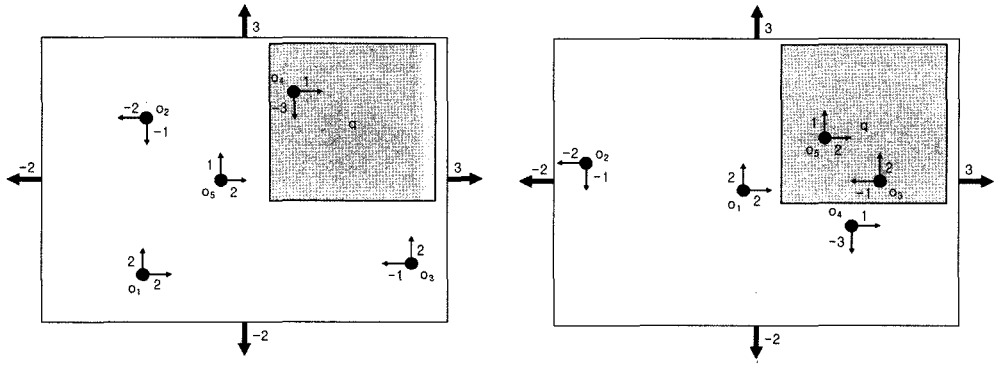
는 t 시점에서 범위 검색을 수행할 때 겹침을 발생하지 않지만 n_1과 n_3는 겹침이 발생한다. 따라서, n_1과 n_3 영역에 해당하는 자식 노드를 탐색하면서 t 시점에서 범위 검색을 만족하는 이동 객체를 검색한다.

t 시점에서 범위 검색을 만족하는 자식 노드를 계속 탐색하여 단말 노드에 도달하면 단말 노드에 존재하는 이동 객체 o_i에 대해 t 시점 위치를 계산하여 질의 범위 q 내에 존재하는지를 판별한다. 예를 들어, 그림 19의 (a)와 같이 질의 범위 q가 존재하고 t가 3이라고 할 때 t_{upt}가 1인 5개의 이동 객체가 단말 노드에 존재한다고 하자. t 시점에서 질의 범위 q 내에 존재하는지를 계산하기 위해 단말 노드에 존재하는 5개의 이동 객체에 대한 위치를 계산한다. 그림 19의 (b)는 t 시점에서 이동 객체의 위치를 나타낸 것이다. 그림 19의 (b)에서



(a) 중간 노드의 분할 영역 (b) 범위 검색을 위한 노드의 확장

그림 18 중간 노드에서 미래 위치 검색



(a) 단말 노드의 이동 객체 (b) t 시점에서 이동 객체의 위치

그림 19 단말 노드에서 미래 위치 검색

보는 것과 같이 t 시점에서 질의 조건을 만족하는 이동 객체는 o_3와 o_5이다.

5. 실험 및 성능평가

성능 평가를 위해 제안하는 색인 구조, TPR-트리 그리고 TPR*-트리를 펜티엄 Ⅳ 2.8GHz 프로세서와 256 Mbyte의 메모리를 가지는 시스템에서 C 언어를 통해 구현한다. 성능평가에 사용된 데이타 집합은 GSTD [22]를 통해 1000×1000km의 2차원 공간에서 임의의 속도를 갖는 이동 객체를 생성한다. 다양한 실험 데이타를 사용하기 위해 GSTD을 통해 이동 객체 위치가 가우시안(gaussian), 스큐(skew) 그리고 균등(uniform) 분포로 존재하는 100,000개 이동 객체를 생성한다. 또한, 각 이동 객체에 대해 1,000개의 이동 변화를 갖도록 한다. 성능 평가를 위해 구현한 색인 구조에서 노드의 크기는 4Kbyte로 구성하며 10,000개~100,000개의 이동 객체를 삽입하고 1,000개~10,000개의 객체에 대한 갱신을 수행한 후 현재 및 미래 위치에 대한 범위 검색을 수행한다. 서비스를 중지한 객체를 판별하기 위해 사용되는 $MaxT$는 이동 객체의 갱신 시간의 3배로 설정하여 실험을 수행한다.

5.1 분할 기법에 따른 검색 성능

범위 검색에 대한 성능 평가를 수행하기 위해 먼저 제안하는 색인 구조에서 제안된 분할 기법과 기존 공간 분할 기반 색인 구조에서 사용된 분할 기법에 대해 성능 평가를 수행한다. 분할 정책에 대한 성능 평가를 위해 제안하는 분할 기법(MOS: MObility dependent Split)과 기존에 제안된 데이타 기반 분할 기법(DAS: DAta dependent Split), 분포 기반 분할 기법(DIS: DIstribution dependent Split)을 비교 분석한다. 그림 20에서 그림 22는 현재 시점에서 검색 범위를 전체 영역에 대해 0.1에서 2로 변경하면서 분할 정책에 대한 검색 시간을 나타낸 것이다. 그림 23에서 그림 25는 검색을 수행하는 미래 시간을 5에서 30분까지 변화하면서 분할 정책에 대한 검색 시간을 나타낸 것이다. 성능 평가 결과 기존에 제안된 분할 기법에 비해 제안하는 분할 기법에 이동 객체의 현재 시점에서의 범위 검색과 미래 시점에서의 범위 검색 모두에 대해 성능 향상됨을 알 수 있다. 제안하는 분할 기법은 공간 분할 방식에서 사용되는 분할 기법에 대해 현재 위치 검색은 약 15%의 성능 향상이 보인다. 또한, 미래 위치 검색은 약 28%의 성능 향상을 보임을 알 수 있다.

5.2 기존 색인 구조와의 검색 성능

기존에 제안된 색인 구조와의 범위 검색에 대한 성능

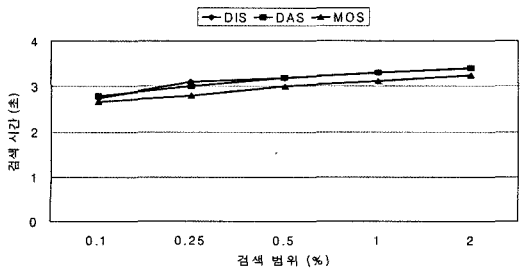


그림 20 분할 정책에 따른 균등 분포의 현재 위치 검색

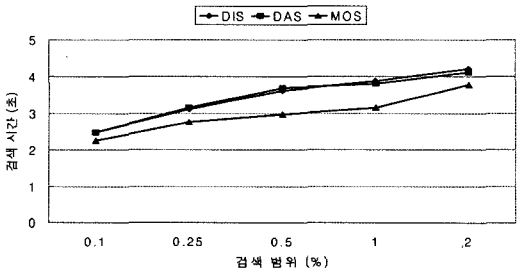


그림 21 분할 정책에 따른 가우시안 분포의 현재 위치 검색

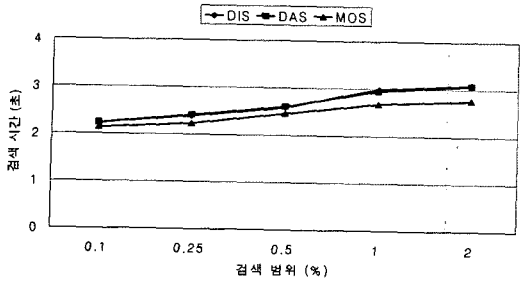


그림 22 분할 정책에 따른 스큐 분포의 현재 위치 검색

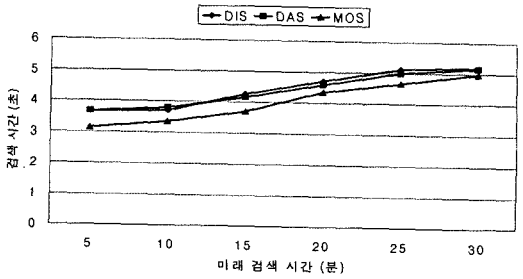


그림 23 분할 정책에 따른 균등 분포의 미래 위치 검색

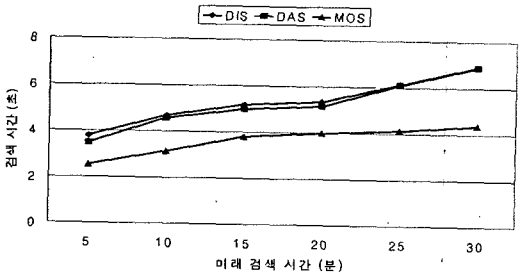


그림 24 분할 정책에 따른 가우시안 분포의 미래 위치 검색

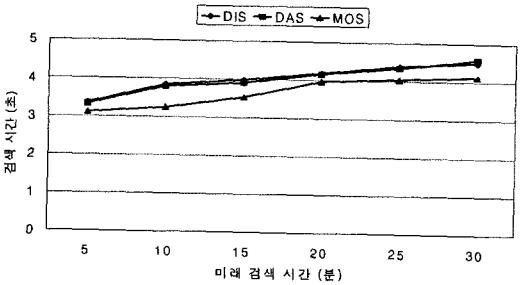


그림 25 분할 정책에 따른 스큐 분포의 미래 위치 검색

평가를 수행하기 위해 기존에 제안된 TPR-트리, TPR*-트리에 대한 검색 시간을 비교한다. 그림 26에서 그림 28은 현재 시점에서 검색 범위를 전체 영역에 대해 0.1에서 2로 변경하면서 검색을 수행한 시간을 나타낸 것이다. 성능 평가 결과 제안하는 색인 구조는 현재 시점에 대한 범위 검색 성능에 대한 성능이 향상됨을 알 수 있다. 기존에 제안된 색인 구조는 검색 범위가 증가할수록 검색 시간을 급격히 증가되지만 제안하는 색인 구조는 검색 범위가 증가되어도 검색 시간에 큰 영향을 미치지 않는다. 제안하는 색인 구조는 기존 TPR-트리에 비해 약 190%의 성능 향상을 보이며 TPR*-트리에 비해서는 135%의 성능 향상을 보인다. 특히, 검색 범위가 증가할수록 제안하는 색인 구조는 기존 색인 구조에 비해 많은 성능 향상을 보인다.

그림 29에서 그림 31은 검색을 수행하는 미래 시간을 5분에서 30분까지 변화하면서 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 유사하

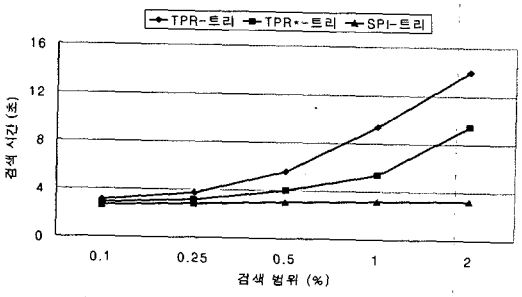


그림 26 균등 분포의 현재 위치 검색

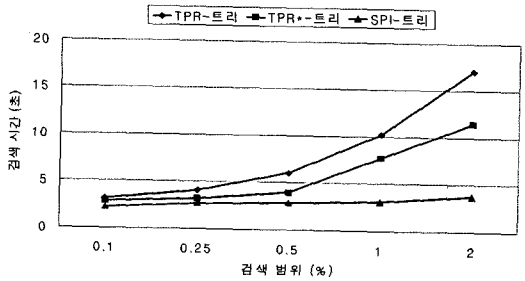


그림 27 가우시안 분포의 현재 위치 검색

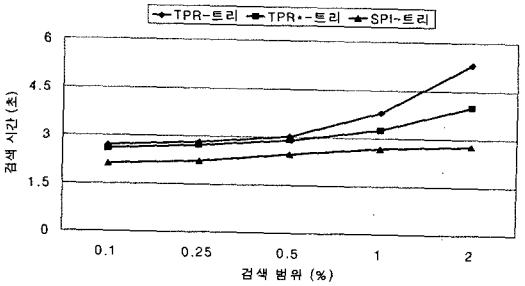


그림 28 스큐 분포의 현재 위치 검색

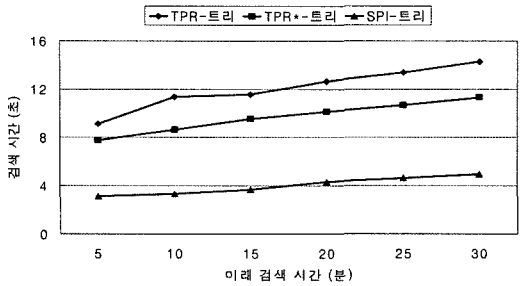


그림 29 균등 분포의 미래 위치 검색

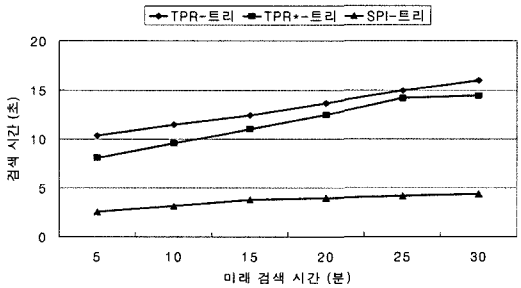


그림 30 가우시안 분포의 미래 위치 검색

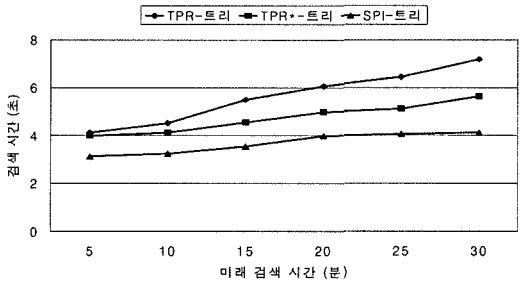


그림 31 스큐 분포의 미래 위치 검색

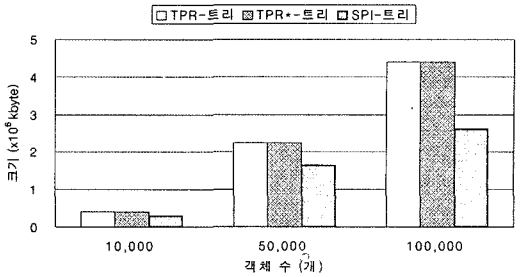


그림 32 균등 분포의 색인 구조 크기

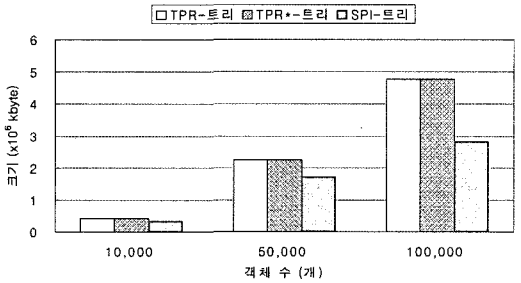


그림 33 가우시안 분포의 색인 구조 크기

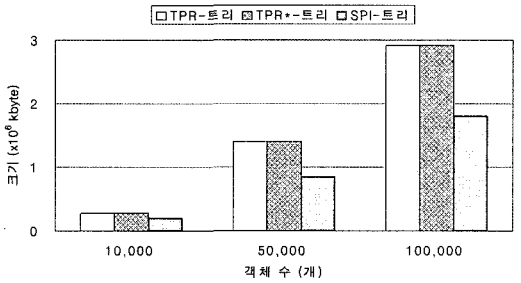


그림 34 스큐 분포의 색인 구조 크기

게 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 성능의 저하가 거의 없는 것을 알 수 있다. 제안하는 색인 구조는 TPR-트리에 비해 약 185%의 성능 향상을 보이며 TPR*-트리에 비해서는 약 140%의 성능 향상을 보인다.

기존에 제안된 TPR*-트리는 TPR-트리에 비해 이동 객체의 현재 위치 검색이나 미래 위치 검색에 대해 우수한 성능을 보인다. 기존 색인 구조들은 자식 노드에 존재하는 이동 객체의 속도를 포함하는 속도를 유지하고 이를 통해 이동 객체에 대한 검색을 수행한다. 따라서, 실제 노드 내에서 최대 속도로 이동하는 객체가 실제 영역을 벗어나지 않는 시간에도 노드를 확장하기 때문에 검색 성능이 저하된다. 제안하는 색인 구조는 자식 노드에 존재하는 이동 객체의 속도 뿐만 아니라 노드 내에 존재하는 이동 객체들이 영역을 벗어나는 시간 등을 표현한다. 또한, 오버플로우가 발생한 노드에 대한 분할을 수행할 때 노드에 존재하는 이동 객체들에 대한 이동성을 고려하기 때문에 미래 시점에 대한 검색을 수행하는 과정에서 불필요한 노드의 확장을 감소시킬 수 있다. 따라서, 미래 위치 검색을 수행하는 과정에서 불필요한 노드의 확장을 감소시킬 수 있어 검색 성능이 향상된다.

5.3 색인 구조 크기

이동 객체의 새로운 위치를 삽입하거나 이동 객체의 위치를 탐색하기 위해 접근해야 할 노드의 수를 감소시키기 위해서는 색인 구조의 높이를 감소시키는 것이 필요하다. 제안하는 색인 구조에서는 노드에 대한 분할 영역을 표현하기 위해 실제 영역을 표현하는 것이 아니라

자식 노드를 생성하기 위해 필요한 분할 정보를 kd-트리로 구성한다. 따라서, 노드의 팬아웃을 증가시킬 수 있고 이로 인해 색인 구조의 높이를 감소시킬 수 있다. 제안하는 색인 구조와 기존에 제안된 색인 구조에 대해 이동 객체를 삽입하여 구성된 색인 구조의 크기를 실험을 통해 비교 분석한다. 그림 32에서 그림 34는 이동 객체의 삽입으로 생성된 색인 구조의 크기를 나타낸 것이다. 이동 객체의 삽입으로 생성되는 색인 구조의 크기를 비교하기 위해 10,000개에서 100,000개까지의 이동 객체를 삽입한다. 성능 평가 결과 제안하는 색인 구조는 기존에 제안된 TPR-트리와 TPR*-트리에 비해 색인 구조의 크기가 감소되는 것을 알 수 있다. 10,000개의 이동 객체를 삽입하였을 때, 생성된 색인 구조의 크기는 기존 색인 구조의 크기와 큰 차이를 보이지 않는다. 그러나, 삽입되는 이동 객체의 수가 증가될수록 색인 구조의 크기는 큰 차이를 보인다. 제안하는 SPI-트리는 기존에 제안된 TPR-트리와 TPR*-트리에 비해 약 40%의 성능 향상을 보인다.

제안하는 색인 구조는 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, SPI-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.

5.4 삽입 성능

이동 객체에 대한 삽입 성능을 평가하기 위해 10,000개에서 100,000개의 이동 객체를 삽입하는 시간을 분석한다. 그림 35에서 그림 37은 이동 객체에 대한 삽입 시간을 나타낸 것이다. 성능 평가 결과 제안하는 색인 구조는 기존 색인 구조에 비해 삽입 성능이 매우 우수한 것을 알 수 있다. 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. 제안하는 SPI-트리는 기존에 제안된 TPR-트리에 비해 285%~710%의 성능 향상을 보이며 TPR*-트리에 비해서는 293%~850%의 성능 향상을 보인다.

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

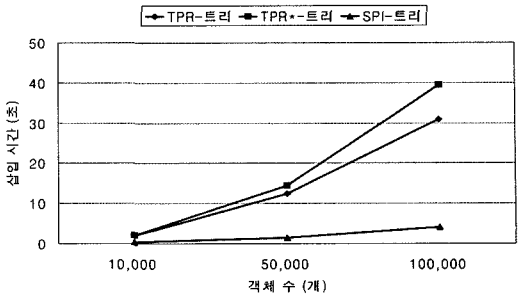


그림 35 균등 분포의 삽입 성능

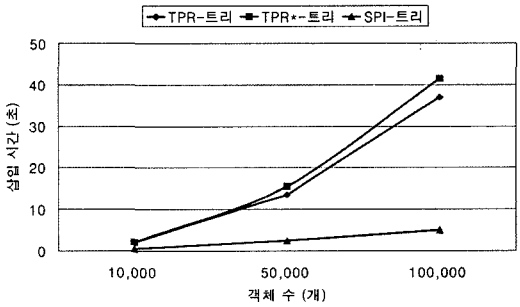


그림 36 가우시안 분포의 삽입 성능

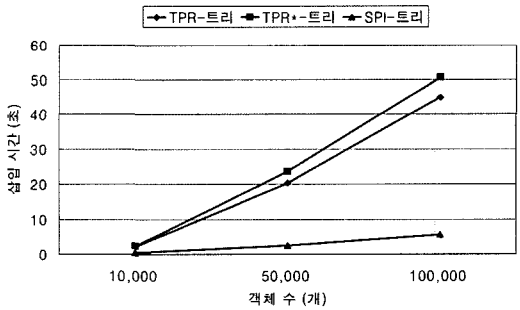


그림 37 스큐 분포의 삽입 성능

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.

6. 결론 및 향후 연구

본 논문에서는 계속적인 위치를 변화에 따른 이동 객체의 현재 및 미래 위치 검색을 지원하기 위해 공간 분할 방식의 색인 구조를 제안하였다. 제안하는 색인 구조는. 기존 공간 분할 방식 색인 구조를 확장하여 이동 객체의 미래 위치를 검색할 수 있도록 노드 내에 존재하는 이동 객체들에 대한 속도를 정보를 유지한다. 또한,

이동 객체의 위치를 검색하는 과정에서 불필요한 노드의 확장을 감소시키기 위해 노드 내에 존재하는 이동 객체들이 영역을 벗어나는 시간과 갱신 시간을 유지하여 불필요한 노드의 확장을 제거한다. 또한, 노드에 오버플로우가 발생할 경우 직접 분할을 수행하지 않고 형제 노드와 병합 분할을 수행한다. 병합 분할이 가능한 노드가 존재하지 않을 경우에는 이동 객체의 이동성을 고려하여 분할을 수행한다. 성능 평가 결과 제안하는 색인 구조는 기존에 제안된 TPR-트리와 TPR*-트리에 비해 매우 우수한 성능을 보임을 알 수 있다. 제안하는 색인 구조는 전통적인 분할 전략에 비해 약 24%의 성능 향상을 보이며 기존에 제안된 색인 구조에 비해 약 160% 성능이 향상되었다.

향후 연구 방향으로 이동 객체에 대한 k-최근접 검색 및 모양 기반 검색을 효과적으로 처리하기 위한 질의 처리 기법과 함께 특정 시간 동안 동일한 검색을 수행하는 연속 질의 처리 기법에 연구를 수행할 예정이다.

참 고 문 헌

[1] D. L. Lee, J. Xu, B. Zheng and W. C. Lee, "Data Management in Location-Dependent Information Services," IEEE Pervasive Computing, Vol. 1, No. 3, pp.65-72, 2002.

[2] 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.

[3] D. Pfoser, C. Jensen and Y. Tehodoridis, "Novel Approaches to the Indexing of Moving Object Trajectories," Proc. the 26th International Conference on Very Large Data Bases. pp.395-406, 2000.

[4] Z. Song and N. Roussopoulos, "SEB-tree : An Approach to Index Continuously Moving Objects," Proc. the 4th International Conference on Mobile Data Management, pp.340-344, 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] P. K. Agarwal, L. Arge and J. Erickson, "Indexing Moving Points," Journal of Computer and System Sciences, Vol.66, No.1, pp.207-243, 2003.

[7] D. Papadopoulos, G. Kollios, D. Gunopulos and V. J. Tsotras, "Indexing Mobile Objects on the Plane," Proc. the 13th International Workshop on Database and Expert Systems Applications, pp.693-697, 2002.

[8] 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.

[9] 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.

[10] C. M. Procopiuc, P. K. Agarwal and S. H. Peled, "STAR-Tree : An Efficient Self-Adjusting Index for Moving Objects," Proc. the 4th International Workshop on Algorithm Engineering and Experiments, pp.178-193, 2002.

[11] 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.

[12] Y. Theodoridis, R. Silva and T. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," Proc. the 3rd IEEE International Conference on Multimedia Computing and Systems, pp.441-448. 1996.

[13] M. A. Nascimento and J. R. O. Silva, "Towards Historical R-trees," Proc. ACM symposium on Applied Computing, pp.235-240, 1998.

[14] Y. Tao and D. Papadias, "Efficient Historical R-trees," Proc. IEEE International Conference on Scientific and Statistical Database Management, pp.223-232, 2001.

[15] Y. Tao and D. Papadias, "MV3R-Tree : A Spatio-Temporal Access Method for Timestamp and Interval Queries," Proc. 27th International Conference on Very Large Data Bases, pp.431-440, 2001.

[16] V. Prasad Chakka, A. Everspaugh, J. M. Patel, "Indexing Large Trajectory Data Sets With SETI," Proc. the First Biennial Conference on Innovative Data Systems Research, 2003.

[17] D. Kwon, S. Lee, W. Choi, S. Lee, "Adaptive Multi-level Hashing for Moving Objects," Proc. International Conference on Database Systems for Advanced Applications, pp.920-925, 2005.

[18] X. Wang, Q. Zhang, W. Sun, "GTree: An Efficient Grid-Based Index for Moving Objects," Proc. International Conference on Database Systems for Advanced Applications, pp.914-919, 2005.

[19] A. Henrich, "A Hybrid Split Strategy for k-d-Tree Based Access Structures," Proc. the fourth ACM workshop on Advances on Advances in Geographic Information Systems, pp.1-8, 1996.

[20] A. Henrich, "The LSDh-Tree : An Access Structure for Feature Vectors," Proc. the Fourteenth International Conference on Data Engineering, pp.362-369, 1998.

[21] R. Orlandic and B. Yu, "Implementing KDB-Trees to Support High-Dimensional Data," Proc. the International Database Engineering and Applications Symposium, pp.58-67, 2001.
[22] 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.

복 경 수

1998년 충북대학교 수학과(이학사). 2000년 충북대학교 정보통신공학과(공학석사). 2005년 충북대학교 정보통신공학과(공학박사). 2005년 3월~현재 한국과학기술원 전산학과 Postdoc. 관심분야는 자료 저장 시스템, 이동 객체 데이타베이스, 시공간 색인 구조, 센서 네트워크 및 RFID 등

유 재 수

1989년 전북대학교 컴퓨터공학과(공학사). 1991년 한국과학기술원 전산학과(공학석사). 1995년 한국과학기술원 전산학과(공학박사). 1995년~1996년 8월 목포대학교 전산통계학과 전임강사. 1996년 8월~현재 충북대학교 정보통신공학과 교수. 관심분야는 데이타베이스 시스템, 정보검색, 멀티미디어 데이타베이스, 분산 객체 컴퓨팅 등