비디오에서 이동 객체의 궤적 검색을 위한 시공간 색인구조

이 낙 규[†] · 복 경 수[††] · 유 재 수[†††] · 조 기 형[††††]

요 약

이동 객체는 시간이 변화함에 따라 공간적인 위치나 모양, 크기 등이 변화하는 특징을 가지고 있다. 이런 객체의 변화는 연속적인 움직임을 수반하고 있는데, 이것을 궤적이라 한다. 본 논문에서는 한번의 노드 접근으로 이동 객체의 궤적을 검색할 수 있는 색인구조를 제안한다. 또한 시공간 범위검색은 물론 궤적검색에 효율적인 다중복합 검색을 제안한다. 제안된 방법의 우수성을 보이기 위해 실험을 통하여 검색시간과 저장 공간에 대한 성능을 여러 환경에서 비교 분석하여 기존의 색인구조들에 비해 이동 객체의 시공간 궤적검색이 우수함을 보인다.

Spatio-Temporal Index Structure for Trajectory Queries of Moving Objects in Video

Nak Gyu Lee[†] · Kyoung Soo Bok[††] · Jae Soo Yoo[†††] · Ki Hyung Cho[††††]

ABSTRACT

A moving object has a special feature that it's spatial location, shape and size are changed as time goes. These changes of the object accompany the continuous movement that is called the trajectory. In this paper, we propose an index structure that users can retrieve the trajectory of a moving object with the access of a page. We also propose the multi-complex query that is a new query type for trajectory retrieval. In order to prove the excellence of our method, we compare and analyze the performance for query time and storage space through experiments in various environments. It is shown that our method outperforms the existing index structures when processing spatio-temporal trajectory queries on moving objects.

키워드 : 이동 객체(Moving Object), 궤적 검색(Trajectory Retrieval), 비디오 데이터(Video Data), 시공간 색인구조(Spatio-temporal Index Structure)

1. 서 론

최근 멀티미디어 데이터에 대한 관심이 매우 높아지고 있으며 특히 멀티미디어 데이터의 대표적인 비디오 데이터에 대한 연구가 활발히 진행되고 있다. 비디오 데이터는 대용량, 비정형화된 데이터의 특성뿐만 아니라, 시청각 정보, 시공간 정보를 복합적으로 포함하고 있다. 비디오 데이터는 시간의 변화에 따라 공간적인 위치나 크기 등이 변화하는 특징을 가지는 이동 객체를 포함하고 있다. 이러한 이동 객체의 변화는 연속적인 움직임을 수반하게 되는데 이를 궤적(trajectory)이라 한다.

도로 교통 통제 시스템, 동물 군 관찰 시스템, 스포츠 등과 같은 응용 분야에서는 비디오에서 나타나는 객체를 공간상의 하나의 점으로 표현하고 이러한 점들의 변화를 통해 궤적을 표현한다. 비디오에 나타나는 객체의 궤적은 공간상의 위치, 이동 방향 그리고 이동 거리 등을 통해 표현하고 이를 통해 유사한 움직임을 갖는 객체의 이동 경로를 추적하거나 유사한 패턴으로 움직이는 객체를 검색한다[1, 2].

비디오에 나타나는 이동 객체를 효과적으로 검색하기 위해서는 시간의 변화에 따른 공간적인 위치뿐만 아니라 객체의 전체적인 이동 경로 즉, 궤적을 저장하고 검색할 수 있는 색인 구조가 필요하다. 기존에 제안된 색인구조는 시간 또는 공간을 기준으로 색인을 구성한다. [3, 4]에서는 객체의 공간적인 위치는 고려하지 않고 시간을 기준으로 색인을 구성하는 기법들을 제안하였다. 이에 반해 [5-8]에서는 객체의 공간적인 위치를 이용하여 색인을 구성하는 기법들을 제안하였다. 그러나 이러한 색인구조들은 시간 또는 공간적인 하나의 특징만을 이용하여 색인을 구성하기 때문에 연속적으로 변하는 객체에 대한 궤적 변화를 효과적으로 검색하지 못한다는 문제점이 있다.

이러한 문제점을 해결하기 위해 시간의 변화와 공간적인

※ 본 논문은 2003년도 정보통신진흥연구원 정보통신기초기술연구사업 연구비 지원에 의해 수행되었음.
† 준 회 원 : 충북대학교 대학원 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소
†† 준 회 원 : 충북대학교 대학원 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소
††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소 교수
†††† 정 회 원 : 충북대학교 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소 교수
논문접수 : 2003년 3월 25일, 심사완료 : 2003년 10월 29일

변화 모두를 고려한 시공간 색인구조가 제안되었다[9-12]. 시공간 색인구조들은 객체의 유효 시간구간과 영역정보로 표현하여 객체에 대한 시공간 범위검색을 가능하게 하였다. 그러나 이러한 시공간 색인구조들은 이동 객체를 시간구간과 영역정보로만 표현하기 때문에 궤적검색을 효과적으로 수행하지 못한다는 문제점이 있다. 또한 시간구간과 영역정보로만 이동 객체를 표현할 때 객체가 없음에도 불구하고 있는 것처럼 저장하기 때문에 많은 저장공간과 사장공간(dead space)을 요구하게 된다[13]. 특히 색인구조의 중간노드(internal node)에서 사장공간은 저장공간과 검색성능의 성능을 저하시키는 요인이 된다.

이런 문제점을 해결하기 위해서 이동 객체의 궤적을 검색하기 위한 색인구조가 제안되었다[14]. [14]에서는 연속적인 이동 객체의 궤적을 가능한 하나의 노드에 유지하기 위해 궤적 보존 방법을 이용하여 분할한다. 궤적 보존 방법은 연속적인 궤적이 서로 다른 노드에 저장 될 때 각 노드를 양방향 연결 리스트 형태로 연결시키는 것이다. 이렇게 함으로써 이동 객체에 대한 이전 또는 이후의 궤적은 쉽게 접근할 수 있다. 그러나 다른 노드에 존재하는 궤적을 검색하기 위해 양방향 연결 리스트를 따라 이동하게 될 때마다 단말노드에 대해 빈번한 I/O를 수행해야 한다는 문제점이 있다.

본 논문에서는 이러한 문제점들을 해결하기 위해 MVTB (Multi-Version Trajectory Bundle)-트리를 제안한다. 제안하는 MVTB-트리는 유효시간 구간에 나타나는 이동 객체의 궤적을 미리 할당된 연속적인 공간에 유지함으로써 한 번의 I/O만으로도 이동 객체의 궤적검색을 효율적으로 수행할 수 있게 한다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구로 기존에 연구되었던 시공간 색인구조에 대해 기술하고 3장에서 제안하는 MVTB-트리의 구조와 삽입 및 검색 방법을 기술한다. 4장에서는 성능평가를 통해 제안하는 색인구조가 시공간 궤적검색에 우수함을 증명한다. 마지막 5장에서는 결론 및 향후 연구방향에 대해 기술한다.

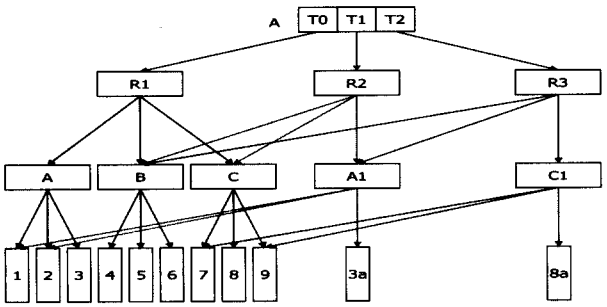
2. 관련 연구

최근 비디오 데이터에 대한 연구가 활발히 진행되면서 비디오 데이터에 나타나는 이동 객체에 대한 시공간 검색의 중요성이 증가되고 있다. 또한 무선 통신과 모바일 장치들의 개발과 함께 GPS 기술에 기반한 위치 기반 서비스를 효과적으로 처리하기 위한 시공간 색인 구조들이 제안되었다. 이 장에서는 기존에 제안된 시공간 색인구조의 특징과 문제점을 기술한다.

2.1 HR(Historical R)-트리

R-트리[5]는 시간에 따라 변화하는 데이터 객체를 유지하려 할 때 많은 저장공간을 요구한다는 단점이 있다. 이런 문제점을 해결하기 위해 HR-트리가 제안되었다[9]. (그림 1)은 HR-트리의 구조를 보여준다. HR-트리는 현재시점에서 하나의 데이터 객체가 변하게 되면 변경된 객체를 저장하기 위한 R-트리의 노드를 생성하여 변경된 데이터 객체를 삽입하고 과거시점에 변경되지 않은 공간 데이터 객체는 그대로 유지하는 방법을 취한다.

예를 들어 (그림 1)과 같이 T0인 시점에서 최초의 R-트리가 구성되어있다고 하자. T1인 시점에서 엔트리 *3*이 *3a*로 변경되었다면 A와 동일한 부모노드 A1을 생성한다. 엔트리 *3*을 가리키던 A의 포인터 대신 A1의 포인터는 변경된 엔트리 *3a*를 가리키게 하며 변경되지 않은 엔트리 *1*과 *2*를 가리키게 한다. 엔트리 *3a*를 포함하는 부모노드의 정보가 변경되었기 때문에 부모노드의 상위노드 역시 부모노드에서 변경되지 않은 노드를 가리키는 포인터는 그대로 두고 변경된 노드의 포인터 정보를 가지는 새로운 부모노드 R2를 생성한다. 이런 방법을 최상위 레벨까지 반복하여 HR-트리를 구성한다.



(그림 1) HR-트리의 구조

HR-트리는 각각의 단위시간에 따라 R-트리를 구분해서 사용하므로 단위시간에 따른 검색에는 매우 효과적이다. 그러나 시간에 대한 범위검색 수행에는 적합하지 못한 색인구조이다. HR-트리를 이용하여 범위검색을 수행하게 된다면 시간 구간으로 들어오는 검색형태를 단위시간으로 나누어서 각각의 단위시간에 맞는 R-트리를 모두 검색해야 한다. 따라서 서로 다른 R-트리의 부모노드에서 같은 노드를 여러 번 중복 방문한다는 문제점이 있다. 또한 같은 정보를 중복 저장하여 유지하게 되므로 많은 저장공간을 요구한다는 문제점도 있다. 이를 해결하기 위한 방법으로 제안된 것이 positive와 negative 포인터이고 HR-트리를 개선시킨 색인 구조로 HR^+-트리[12]가 있다.

2.2 3DR(3 Dimension R)-트리

[13]에서는 시간차원 색인구조와 공간차원 색인구조를 하나의 색인구조로 하여 시공간 검색 검색을 제공해 주는 것과, 시공간차원 특성을 동시에 고려하여 하나의 색인구조로 구성하고 범위검색을 제공해 주는 두 가지 색인구조로 제안되었다.

전자의 경우는 하나의 색인구조에서 객체의 시간적 특성 정보(id, t_1, t_2)를 유지하고 다른 하나의 색인 구조에서는

동일한 객체의 공간적 특성정보(id, x_1, x_2, y_1, y_2)를 유지한다. 즉, 공간 색인구조를 유지하는 2DR-트리와 시간 색인구조를 유지하는 1DR-트리를 이용하여 하나의 시공간 색인구조처럼 사용하는 것이다. 이러한 색인구조를 사용하는 시스템에서의 시공간 검색 처리단계를 살펴보면, 우선 검색 연산을 수행하기 위해 시간검색과 공간검색을 적절히 나누어 각 색인구조에 검색을 수행하게 한다. 이때 각 색인구조에서는 검색에 적합한 결과집합을 응답하게 되는데 응답되어진 결과집합에서 다시 한번 최초 검색에 적합한 데이터 객체가 있는지를 계산하게 된다.

이런 문제점을 해결하기 위해 [13]에서는 두 개의 서로 다른 색인구조를 동시에 고려한 3DR-트리를 제안하고 있다. 3DR-트리는 객체의 시간차원과 공간차원을 동시에 고려한 색인구조로서 R-트리로 표현되는 X, Y 공간 좌표축 외에 시간 차원을 표현하는 T축이 추가되어 데이터 객체에 대한 시간 정보를 함께 유지하는 방법을 취한다. 데이터 객체에 대한 시공간 정보를 해당 데이터 객체가 나타난 시점부터 사라진 시점까지의 시간구간 정보와, 유효한 시간구간에서 가장 큰 X, Y 공간영역을 포함하는 육면체로 모든 이동 객체를 표현하고 있다. 이렇게 하나의 색인구조에서 시공간차원을 동시에 고려함으로써 시공간 데이터를 표현하는데 적합하며 시공간 검색을 보다 효율적으로 수행할 수 있다.

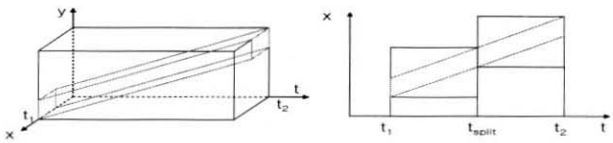
그러나 3DR-트리는 궤적을 가지며 움직이는 데이터 객체를 단순히 나타난 시점에서의 공간좌표와 사라진 시점에서의 공간좌표만을 가지고 하나의 육면체로 표현하게 될 경우 많은 사장공간이 발생한다는 문제점이 있다. 많은 사장공간은 인근에 있는 다른 노드와의 겹침을 유발시키며 검색수행 속도를 저하시킨다는 문제점을 가지고 있다. 또한 3DR-트리는 시간 차원이 증가함에 따라 시간구간 검색은 다른 색인구조에 비해 성능이 썩 나쁘지는 않지만 단위시간 검색은 모든 시간차원을 검색해야 하므로 성능이 크게 떨어지게 되는 문제점이 있다.

2.3 SR(Spatiotemporal R)-트리

[15]에서는 효율적인 시공간 검색을 위해 사장공간을 줄이는 방법을 제안하였다. 본 논문에서는 [15]에서 제안한 색인을 편의상 SR-트리라 한다. 기존에 제안된 색인구조에서 시공간 객체들은 시간차원 T축을 따라 2차원 공간인 X, Y 평면에 영역정보를 가지고 표현되는데, 이렇게 유효한 시간 구간동안 새로운 영역정보와 위치정보를 가지는 이동 객체를 시간구간과 영역정보로 표현하다보면 두 가지 문제점이 발생한다. 우선 많은 사장공간이 발생한다는 것이다. 다음으로는 이동 객체의 유효 시간구간이 상당히 길어지면 이 이동 객체가 차지하는 영역크기도 길게 나타나며 색인구조의 많은 단말노드에서 같은 객체정보를 유지하고 있어야 한다는 것이다. 이런 문제로 노드의 겹침이 많이 발생하고 결국 검색 수행능력이 감소하게 된다.

이러한 문제점들을 해결하기 위해서 [15]에서는 이동 객체의 사장공간을 감소시키기 위해 해당 시공간 객체의 시간영역을 분할하는 방법을 취한다. (그림 2)에서 이동 객체의 시간영역을 분할하여 사장공간을 감소시키는 모습을 보여준다.

(그림 2)(a)에서와 같이 많은 사장공간을 포함하고 있는 시공간 객체를 연속되는 작은 영역정보를 가지게끔 분할하면 (그림 2)(b)처럼 기존의 영역에 포함된 사장공간 보다 적은 사장공간을 포함하게 된다. 즉, 사장공간을 줄임으로써 불필요한 영역이 서로 다른 노드에 겹치지 않게 하여 검색 수행능력을 향상시킬 수 있다.



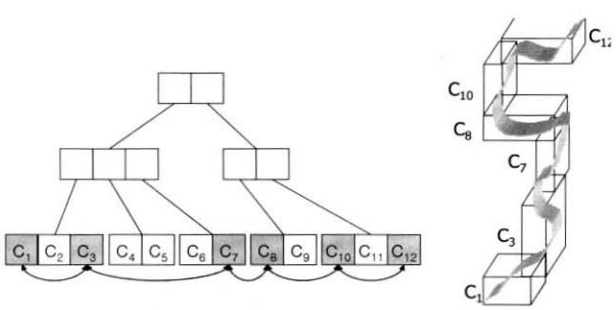
(a) 감소 전 (b) 감소 후

(그림 2) 사장공간의 감소

2.4 TB(Trajectory Bundle)-트리

TB-트리는 R-트리를 확장하여 이동 객체의 궤적을 보존하기 위해 제안된 색인 구조이다[14]. TB-트리에서 새로운 세그먼트가 삽입되면 R-트리의 *FindNode*()와 유사한 알고리즘을 통해 궤적의 이전 세그먼트가 저장된 단말 노드를 찾는다. 만약 단말 노드에 새로운 세그먼트를 삽입할 수 없다면 새로운 단말 노드를 생성하여 새로운 노드를 삽입할 수 있는 부모 노드를 탐색한다. TB-트리는 왼쪽에서 오른쪽으로 확장되며 가장 오른쪽에 있는 노드가 가장 처음에 삽입된 궤적의 세그먼트가 된다.

(그림 3)은 궤적의 세그먼트가 삽입되어 TB-트리를 구성한 예를 나타낸다. 하나의 객체에 대한 궤적이 (그림 3)(b)와 같이 나타낼 때 TB-트리는 처음 삽입된 세그먼트를 저장한 C_1노드에서부터 마지막에 삽입된 C_{12}까지 양방향 연결리스트 하나의 객체에 대한 궤적을 연결하고 있다. 이를 통해 하나의 객체에 대한 궤적을 검색할 때 양방향 연결리스트 연결된 노드를 순회하면서 조건에 만족하는 노드를 검색한다.



(a) TB-트리 (b) 궤적

(그림 3) TB-트리 및 궤적 보존 방법

TB-트리의 단말 노드는 하나의 궤적에 속하는 세그먼트만을 포함하고 있다. 따라서 다른 궤적에 속하는 세그먼트가 시간 또는 공간적으로 인접하게 존재할 때에도 다른 노드들에 저장된다는 문제점이 있다. 또한 이로 인해 노드들 사이의 공간적인 겹침을 증가되어 범위 질의에 대한 성능이 저하되는 문제점이 있다.

이동 객체의 궤적을 보존하려는 TB-트리의 양방향 연결 리스트는 동일한 이동 객체의 궤적을 빠르게 검색할 수 있다는 장점이 있다. 그러나 단말노드에서의 양방향 연결 리스트는 많은 노드 I/O를 발생시킨다. (그림 3)의 경우 C_1에서 C_{12}을 검색하기 위해 순차적으로 6번의 노드 I/O를 해야한다. 이렇게 많은 수의 노드 I/O는 검색 수행속도를 감소시키는 문제점 중의 하나이다.

3. MVTB-트리

3.1 개 요

본 논문에서 제안하는 색인구조는 비디오 데이터 안에 포함된 이동 객체에 대한 효과적인 시공간 궤적 검색을 할 수 있도록 이동 객체의 궤적 존속구간을 명확하게 구분하여 표현한다. 이동 객체의 궤적 존속구간을 명확하게 표현하기 위해 동일한 이름을 갖는 이동 객체가 연속적인 움직임이 없을 경우 서로 다른 객체 식별자를 부여한다.

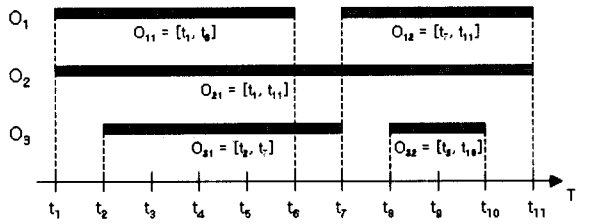
비디오에 데이터에 나타나는 이동 객체에 대한 궤적 및 시공간 범위 검색을 수행하기 위해서는 이동 객체에 대한 연속적인 움직임 정보를 표현한다. 이동 객체에 대한 궤적을 생성하기 위해 먼저 비디오에서 키 프레임을 추출하고 추출된 키 프레임 정보를 이용하여 이동 객체의 연속적인 궤적을 생성한다. 비디오 데이터에 나타나는 이동 객체는 식 (1)과 같이 표현한다. 이때, OID는 객체의 식별자, NA는 객체의 이름, D는 객체의 존속 구간으로 객체가 나타나는 시작 프레임과 종료 프레임으로 나타낸다.

$$(\text{OID}, \text{NA}, \text{D}, \text{TRA}) \quad (1)$$

TRA는 키 프레임에 포함된 객체의 정보를 공간상의 점으로 표현하고 이를 통해 생성한 궤적 정보이다. 객체가 n번의 움직임을 포함할 때 객체의 궤적 TRA는 식 (2)와 같이 표현된다. 이때, SP_i는 객체가 움직이는 시작 위치, R_i은 객체가 움직이는 방향, V_i는 객체가 움직이는 속도, D_i는 객체의 이동 거리, I_i는 객체의 이동시간을 나타낸다.

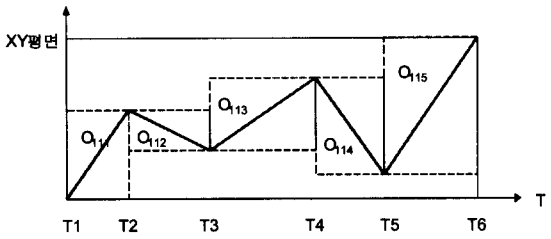
$$<(SP_1, R_1, V_1, D_1, I_1), (SP_2, R_2, V_2, D_2, I_2), \cdots, (SP_n, R_n, V_n, D_n, I_n)> \quad (2)$$

동일한 이름을 갖는 객체에 대해 동일한 식별자 OID를 부여할 경우 실제 객체가 존재하지 않은 구간을 동일한 객체로 인식할 수 있다. 따라서 본 논문에서는 동일한 이름을 갖는 객체에 대해 서로 다른 식별자를 부여한다. (그림 4)는 이동 객체에 대한 식별자를 부여하는 예를 나타낸 것이다. 비디오에 객체 O_1, O_2, O_3가 나타난다고 가정하자. 객체 O_2는 t_1에서 t_{11}까지 연속된 시간동안 한번 나타난다. 이에 반해 객체 O_1과 O_3의 경우 동일한 객체가 불연속적인 시간동안 나타나고 있다. 따라서 제안하는 색인구조에서는 객체 O_1과 같이 t_1에서 t_{11}동안 불연속적으로 나타난 하나의 객체에 대해 O_{11}(=[T_1, T_6]), O_{12}(=[T_7, T_{11}])와 같이 서로 다른 객체 식별자를 부여하여 객체의 궤적 존속구간을 명확히 하는 방법을 취한다. 객체 O_3도 이와 동일한 방법으로 이동 객체의 궤적을 구분한다.



(그림 4) 객체 식별자 부여

효과적인 궤적검색을 제공하기 위해서는 명확한 객체의 궤적 존속구간의 표현뿐만 아니라 사장공간의 문제도 함께 고려해야만 한다. (그림 5)는 (그림 4)에서의 유효 시간구간 [T_1, T_6]에서 객체 식별자 O_{11}을 부여받은 이동 객체의 궤적의 예를 나타내고 있다. 기존에 제안되었된 R-트리, HR-트리, 3DR-트리와 같은 색인구조는 (그림 5)와 같이 이동 객체 O_{11}을 저장하기 위해 실선으로 만들어진 사각영역 전부를 삽입하는 방법을 사용하였다. 이렇게 이동 객체의 시작 시간과 종료 시간, 영역정보만으로 객체를 표현하여 삽입을 수행하면 사장공간이 많이 발생한다는 문제점이 있다.



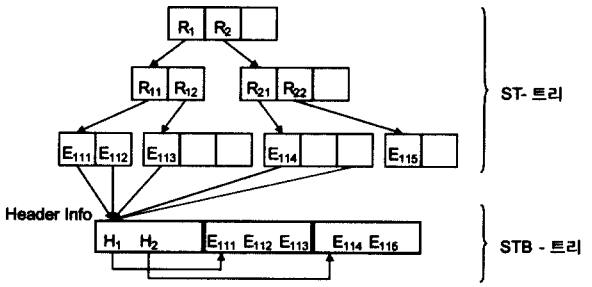
(그림 5) 이동 객체의 궤적

제안하는 색인구조에서는 사장공간을 감소시키며 궤적정보를 효율적으로 표현하기 위해 유효 시간구간 안에서 이동 객체의 궤적 방향이 바뀔 때를 구분하여 저장한다. 이렇게 나누어진 궤적의 일부분을 세그먼트라 한다. (그림 5)에서 O_{111}은 이동 객체 O_{11}의 처음 세그먼트를 의미하고 O_{112}는 O_{11}의 두 번째 세그먼트를 의미한다. 제안하는 색인구조에서는 (그림 5)에서 보는 것과 같이 이동 객체 O_{11}을 점선으로 만들어진 서로 다른 다섯 개의 궤적, 즉 다섯 개의 세그먼트로 나누어 저장하는 방법을 사용한다. 이렇게 함으로

써 기존에 제안되었던 방법처럼 전체적인 시공간 정보를 가지는 이동 객체를 저장할 때 보다 세분화된 궤적에 따라 세그먼트화 하여 저장하는 방법이 사장공간의 문제를 해결할 뿐만 아니라 궤적 검색에도 효과적이다.

3.2 MVTB-트리 구조

제안하는 MVTB-트리는 비디오 데이터가 가지는 이동 객체에 대한 시공간 범위검색 및 궤적검색을 효과적으로 수행하기 위한 색인구조로서 구조로 (그림 6)과 같이 두 개의 색인구조로 구성된다. 하부 구조는 STB(Simple Trajectory Bundle)-트리로서 동일한 식별자를 갖는 객체의 궤적 정보를 물리적으로 연속된 공간에 저장하여 이동 객체에 대한 빠른 궤적검색을 수행한다. 궤적의 전후 위치를 빠르게 검색하기 위해 STB-트리는 *Header Info* 노드와 객체의 실제 궤적을 저장한 노드들로 구성한다. *Header Info* 노드는 세그먼트 정보가 삽입되었을 때 삽입된 노드의 위치와 시공간 정보를 유지한다. 상부구조는 ST(Spatio-Temporal)-트리로서 3DR-트리와 유사한 구조이며 STB-트리의 이동 객체에 대한 세그먼트를 삽입 단위로 하며 단말노드에서는 STB-트리의 *Header Info*를 가리키는 포인터를 두어 이동 객체에 대한 빠른 시공간 범위질의 및 다중 복합질의를 수행할 수 있게 한다.



(그림 6) MVTB-트리의 구조

이동 객체를 삽입하기 위해서는 먼저 시공간 궤적정보를 이용하여 STB-트리의 연속적인 공간에 시공간 궤적정보를 기록한다. STB-트리가 구성되면 기록된 시공간 궤적정보를 이용하여 ST-트리를 구성한다. 객체의 세그먼트를 ST-트리에 기록하면 범위검색 및 궤적검색을 효과적으로 수행할 수 있는 MVTB-트리가 구성된다.

(그림 7)은 MVTB-트리의 삽입 알고리즘이다. MVTB-트리를 구성하기 위해 먼저 *STB_Insert*() 함수를 호출하여 동일한 식별자를 갖는 객체에 대한 STB-트리를 구성한다. *STB_Insert*() 함수는 미리 할당된 연속적인 공간에 세그먼트 정보의 기록을 수행한다. 연속된 공간에 STB-트리를 기록하면 ST-트리를 구성하기 위해 *ST_Insert*() 함수를 호출하게 되는데 *ST_Insert*() 함수는 세 개의 매개변수를 가진다. getSegmentInfo는 *getSegmentInfo*() 함수를 이용하여 i번째 할당된 노드의 세그먼트 정보와 이 노드를 가리키는 *Header info* 노드의 주소를 함께 설정한다. k와 child_ptr은 *ST_Insert*() 함수의 재귀호출시 사용되며 초기 값은 0을 가진다. *ST_Insert*() 함수를 *STB_Insert*()에서 구해진 연속적으로 할당된 노드의 수만큼 호출하게 된다.

```
함수명 : insert ( )
1 : {
2 :     STB_Insert() ;
3 :     for (i = 0부터 연속적으로 할당된 노드 수까지) {
4 :         getSegmentInfo = getSegmentInfo(i) ;
5 :         ST_Insert(getSegmentInfo, k, child_ptr) ;
6 :     }
7 : }
```

(그림 7) MVTB-트리 삽입 알고리즘

3.3 STB-트리

STB-트리는 이동 객체의 세그먼트들을 물리적으로 연속된 공간에 기록함으로 궤적검색을 빠르게 수행하는 것을 목적으로 한다. 빠른 궤적검색을 보장하기 위해 STB-트리는 *HeaderInfo*라는 노드구조를 가지며 다중복합검색 처리에 사용된다. *HeaderInfo*는 삽입되는 세그먼트가 미리 할당된 연속적인 노드들 중 어느 노드에 기록되는지의 정보와 그 노드의 유효 시간구간 및 영역정보를 유지하는 기능을 한다.

3.3.1 STB-트리의 엔트리 구조

STB-트리는 이동 객체 정보를 물리적으로 연속된 공간에 세그먼트 단위로 저장한다. 이동 객체의 세그먼트도 시공간 정보를 가지고 있으므로 세그먼트에 대한 시공간 정보를 함께 유지해야 한다. 식 (3)은 STB-트리의 엔트리 구조를 나타낸 것이다. MBR(Minimum Bounding Rectangle)과 I는 세그먼트의 시공간 정보 유지를 위한 영역정보와 유효 시간구간을 의미하며 TRA는 MBR과 I 구간에서 표현되는 객체의 궤적정보로서 I 구간에서 이동 객체가 가지는 방향 정보이다.

$$(\text{MBR}, \text{I}, \text{TRA}) \tag{3}$$

물리적으로 연속된 공간을 미리 확보하기 위해 식 (4)를 이용하여 동일한 이동 객체의 모든 세그먼트 수 *total_segment_num*을 하나의 노드에 들어갈 수 있는 최대 엔트리 수 *MAX_ENTRY_NUM*로 나누어 물리적으로 연속된 공간이 몇 개가 필요한지 계산한다.

$$pre\_Node\_num = \left\lceil \frac{total\_segment\_num}{MAX\_ENTRY\_NUM} \right\rceil \tag{4}$$

3.3.2 STB-트리의 Header Info 구조

연속적으로 할당된 공간의 노드정보를 유지하는 식 (5)은 *Header info* 구조를 보여준다. *Header Info*의 각 엔트리는 연속적으로 할당된 공간의 노드를 가리키는 주소와 그 노드의 시공간 정보를 함께 유지하여 다중복합검색을 효율적

으로 처리하게 한다. *Header Info* 노드의 MBR과 I는 미리 할당된 하나의 노드에 대한 영역정보와 유효 시간구간을 의미한다. 마지막으로 POS는 이동 객체의 세그먼트 정보가 기록된 위치를 나타낸다.

$$(MBR, I, POS) \quad (5)$$

식 (6)을 이용하여 식 (4)에서 계산된 결과를 *MAX_HEADER_NUM*로 나누어 물리적으로 연속된 공간을 관리할 *Header Info* 노드가 몇 개가 필요한지를 계산한다.

$$head\_Info\_num = \lceil \frac{pre\_Node\_num}{MAX\_HEADER\_NUM} \rceil \quad (6)$$

3.3.2 STB-트리의 삽입

STB-트리는 연속적인 공간에 객체의 궤적정보를 유지하여 다중복합검색을 처리한다는 특징이 있다. STB-트리의 삽입 알고리즘은 삽입하려는 이동 객체의 세그먼트들은 맨 처음에 할당된 노드의 엔트리부터 먼저 채워가면서 저장한다. 따라서 엔트리가 비어있는 노드의 생성빈도가 낮아지게 되어 물리적 저장공간을 절약할 수 있다. 또한 *Header info*에서 세그먼트가 어느 노드에 저장되었는지를 유지하므로 효과적인 다중복합검색을 수행할 수 있다. 이렇게 세그먼트 정보를 유지하는 방법은 한번 색인이 구성되면 새로운 삽입이나 삭제 등 변경 연산이 거의 없는 비디오 데이터 특성상 효과적인 결과를 나타낸다.

```
함수명 : STB_Insert ()
 1 : {
 2 :     getSegment = getMovingObject() ;
 3 :     tempNode = createTempNode() ;
 4 :     for (i = 0 ; i < total_segment_num ; i++){
 5 :         tempNode[entry] = getSegment[i] ;
 6 :         compSpatioTemporal(getSegment[i]) ;
 7 :         if (entry == MAX_ENTRY_NUM ||
                 i == total_segment_num){
 8 :             writeSegment(tempNode) ;
 9 :             entry = 0 ;
10 :             node_count+= 1 ; }
11 :     }
11 : }
```

(그림 8) STB-트리 삽입 알고리즘

(그림 8)은 STB-트리의 삽입 알고리즘을 나타낸다. 동일한 객체 식별자를 가지는 이동 객체의 궤적정보를 데이터 파일로부터 읽어서 세그먼트화하기 위해 *getMovingObject*() 함수를 이용한다. 또한 물리적으로 연속된 공간을 미리 확보하기 위해 식 (4)와 식 (6)를 이용하여 필요한 만큼의 노드 수와 *Header info* 노드의 수를 미리 계산하여 할당한다. 3행에서 최대 엔트리 수만큼 세그먼트를 저장할 수 있는 크기의 노드인 tempNode를 생성한다. 5행에서 getSegment의 정보를 tempNode에 채우고 6행의 *compSpatioTemporal*() 함수를 이용하여 tempNode에 채워지는 모든 getSegment의 시공간 정보를 포함하는 영역정보 값과 시간구간 값을 설정한다. 7행에서 tempNode가 다 찼거나 더 이상 tempNode를 채울 세그먼트 정보가 없다면 8행의 *writeSegment*() 함수를 이용하여 미리 할당된 공간의 node_count번째에 tempNode의 내용을 기록하고 맨 처음 *Header info* 노드의 node_count번째 엔트리에 6행에서 설정된 영역정보 값과 시간구간 값을 설정하게 된다. 여기에서 node_count는 미리 할당된 노드의 순번 및 *Header info*의 엔트리 순번을 의미한다. 9행에서 tempNode의 엔트리 값을 초기화시키며 10행의 node_count는 미리 할당된 다음 노드의 위치 및 *Header info* 노드의 다음 엔트리를 가리키게 1증가시킨다.

3.4 ST-트리

ST-트리는 3DR-트리를 기반한 균형 트리로 시공간 범위 검색을 처리하기 위한 색인구조이다. ST-트리는 STB-트리에 저장된 이동 객체의 세그먼트들을 삽입단위로 한다. 범위 검색을 처리하기 위해 각 내부 노드는 이동 객체의 시공간 정보를 유지한다. 또한 단말노드에서는 이동 객체의 세그먼트 정보를 유지하고 있는 STB-트리를 가리키게 하여 범위 검색 뿐만 아니라 다중복합검색 수행을 가능하게 한다.

3.4.1 ST-트리의 엔트리 구조

제안하는 색인구조 MVTB-트리의 상부구조인 ST-트리의 내부노드 엔트리 구조는 식 (7)과 같다. OID는 이동 객체에 대한 식별자이며 PTR은 하위레벨 노드를 가리키는 포인터이고, MBR과 I는 PTR이 가리키는 하위레벨 노드를 포함하는 영역정보와 유효 시간구간이다.

$$(OID, PTR, MBR, I) \quad (7)$$

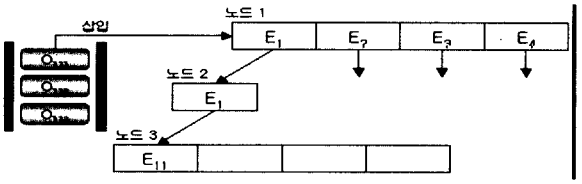
ST-트리의 단말노드 엔트리 구조는 식 (8)과 같이 이동 객체의 식별자인 OID와 궤적정보를 유지하는 STB-트리의 *Header info*를 가리키는 포인터 STB_PTR을 가진다. 내부노드와 마찬가지로 MBR과 I는 삽입되는 세그먼트의 영역정보와 유효 시간구간을 의미한다.

$$(OID, STB\_PTR, MBR, I) \quad (8)$$

3.4.2 ST-트리의 삽입

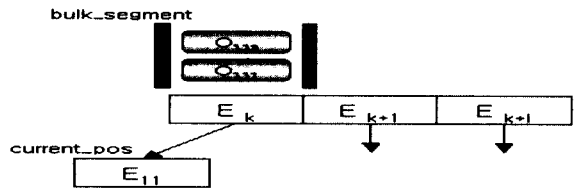
ST-트리는 3DR-트리를 기반으로 한다. 기존에 제안되었던 R-트리 계열의 색인구조는 데이터를 하나씩 삽입하면서 트리를 구성한다. 예를 들어 (그림 9)와 같이 세 개의 세그먼트 O_{111}, O_{112}, O_{113}를 삽입한다고 하자. 만약 세 개의 세그먼트가 노드 *1*의 서로 다른 엔트리에 삽입된다면, 서로 다른 단말노드에 저장 될 것이기 때문에 해당되는 모든 노드에 접근해야 하므로 문제가 되지 않는다. 그런데 이 세 개의 세그먼트가 삽입되기에 적절한 단말노드가 노드 *3*이라면 문제점이 발생한다. 기존에 제안되었던 색인구조의 단말노드 검색방법을 사용하게 되면 세그먼트 O_{111}을 삽입하기 위해 노

드 *1*에 있는 모든 엔트리와 비교연산을 수행하여 그 중 가장 적절한 엔트리가 가리키는 포인터를 따라 하위레벨로 이동한다. 하위레벨에 있는 노드 *2*에서도 같은 방식의 비교연산을 수행하게 된다. 마지막으로 노드 *3*이 선택되고, 노드 *3*에 공간이 있으면 삽입 하게된다. 이런 단계를 나머지 삽입 세그먼트 O_{112}, O_{113}에도 동일하게 적용한다. 이러한 탐색기법은 각 세그먼트를 저장하기 위해서 동일한 노드에 대한 I/O연산을 빈번하게 수행해야 한다는 문제가 발생한다. 이 예의 경우는 세 개의 세그먼트 O_{111}, O_{112}, O_{113}를 삽입하기 위해 노드 *1*과 노드 *2*, 노드 *3*에 대해 각각 세 번씩의 노드 I/O가 발생하여 전체적으로 9번의 노드 I/O가 발생한다. 이는 삽입연산에 많은 시간이 소요된다는 문제점이 있다.



(그림 9) 기존 삽입의 예

본 논문에서 제안하는 삽입의 목적은 이런 많은 수의 노드 I/O 연산을 최소화하여 삽입성능을 향상시키는데 있다. (그림 10)에서는 *ST_insert*() 함수의 매개변수에 대한 예를 나타내고 있다. bulk_segment에는 삽입될 세그먼트 정보가 들어있고 k는 bulk_segment에 저장되어있는 세그먼트 정보가 삽입되기에 가장 적절한 엔트리를 가리키는 target_node에서의 엔트리 순서 정보이다. current_pos는 k번째 엔트리가 가리키는 하위레벨 노드를 가리키는 포인터이다.



(그림 10) ST-트리의 삽입 방법

(그림 11)은 ST-트리의 삽입 알고리즘을 나타낸다. 2행의 getSegmentInfo는 STB-트리에서 segment 정보를 읽어온 1차원 배열로서 getSegmentInfo가 비어있으면 의미가 없으므로 리턴하고 함수를 종료한다. 3행에서 current_pos가 0이면 현재 이동 객체의 세그먼트 정보가 처음 삽입되는 시점이므로 ST-트리의 Root 노드를 읽어서 target_node로 설정한다. 0이 아니면 재귀호출되는 경우인데 이때는 current_pos가 가리키는 주소의 노드 정보를 읽어서 target_node로 설정하게 된다. 5행과 6행에서 target_node의 상태를 체크하고 7행에서 getSegment에 있는 엔트리 수까지 for 문을 수행하며 *InsertSegment*() 함수를 호출한다. *InsertSegment*() 함수는 target_node에 세그먼트 정보를 삽입하는 함수로서 bulk_segment를 구성하게 된다. bulk_segment는 2차원 배열로서 배열의 x축 요소는 target_node의 엔트리 순서((그림 10)의 예에서 E_k, E_{k+1}, E_{k+2})를 의미하여 y축 요소는 각 엔트리에 따라 삽입되기에 적합한 segment 정보((그림 10)의 예에서 O_{111}, O_{113})를 유지하게 된다. target_node에 대한 bulk_segment가 구성되면 target_node의 첫 번째 엔트리에 삽입되기에 적합한 segment 정보를 추출한다. 11행의 for 문은 0부터 getSegmentInfo의 크기만큼 수행이 되는데 이는 최악의 경우 삽입되려고 하는 모든 segment 정보들이 target_node의 특정 엔트리에만 삽입되는게 가장 적합하다고 판단이 될 경우 target_node의 특정 엔트리를 위한 setBulk_segment를 구성할 수 있어야 하기 때문이다. 12행의 *add*(*data*) 함수는 data의 값을 add를 호출한 배열의 맨 마지막 번째 위치에 저장하는 함수이다. *add*()를 이용하여 bulk_segement[i][j]의 값을 setBulk_segment에 설정하게 되는데 i는 target_node의 엔트리에 대한 순서이고 j는 해당 엔트리 번째에 삽입되기에 적합한 segment 정보가 저장된 위치를 의미한다. 즉, 2차원 배열 형태의 bulk_segment의 내용을 1차원 배열 형태의 setBulk_segme nt로 변경하며 target_node의 i번째 위치에 삽입되기에 적합하다고 판단된 segmemt 정보들로 setBulk_segment 값을 채운다. 13행에서 setBulk_segment에 저장된 세그먼트 정보를 target_node의 k번째 엔트리가 가지는 하위레벨 포인터가 가리키는 노드에 삽입을 시도하게 된다.

```
함수명 : ST_Insert ()
 1 : {
 2 :     if (getSegmentInfo == NULL) return 실패 ;
 3 :     if (current_pos == 0) Root 노드를 읽어서 target_node로
             설정 ;
 4 :     current_pos가 가리키는 하위레벨 노드를 읽어서
         target_node로 설정 ;
 5 :     if (target_node == internal node) node_status = true ;
 6 :     else if (target_node == leaf node) node_status = false ;
 7 :     for (j = 0 부터 getSegmentInfo에 있는 엔트리 수) {
 8 :         InsertSegment(target_node, getSegmentInfo, j,
             node_status) ;
 9 :     }
10 :     for (k = 0 부터 target_node에 포함된 엔트리 수) {
11 :         for (segment_data = 0부터 getSegmentInfo.getSize()
                 만큼)
12 :             setBulk_segment.add (bulk_segment[k]
                                 [segment_data]) ;
13 :         ST_insert (setBulk_segment, target_node
                         -> entry[k].child_ptr) ;
14 :     }
15 : }
```

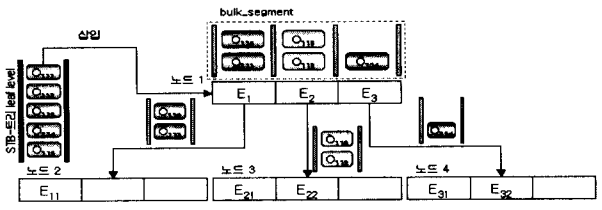
(그림 11) ST-트리 삽입 알고리즘

(그림 12)의 *InsertSegment*() 함수는 세그먼트 정보를 삽입하기 위해 호출되는 함수이다. 이 함수는 네 개의 전달인자를 가지는데 세그먼트 정보가 저장될 target_node와 STB-트리에서 읽어온 getSegmentInfo의 j번째 엔트리, target_

node의 상태를 나타내는 node_status이다. 1행에서 target_node가 full인지 아닌지를 검사하여 full이면 2행과 7행에서 target_node의 node_status를 검사한다. node_status가 true이면 target_node는 내부노드임을 의미하므로 4행에서 *compare*() 함수는 식 (10)을 이용하여 target_node의 모든 엔트리와 getSegmentInfo의 j번째 엔트리와 시공간 크기 비교를 수행하여 getSegmentInfo의 j번째 엔트리와 가장 적게 확장하는 target_node의 i번째 엔트리 값을 temp에 설정한다. temp에 설정된 값은 comp와 비교하여 둘 중 작은 값을 comp에 저장한다. 6행에서는 bulk_segment의 첫 번째 요소 중 comp값을 설정하고 두 번째 요소에 getSegmentInfo의 j번째 엔트리 정보를 설정한다. node_status가 false이면 단말노드임을 의미하며 9행에서 분할을 시도한다. 분할시 호출되는 함수 *split*()은 getSegmentInfo의 j번째 엔트리를 저장할 target_node를 전달인자로 가진다. 분할 연산을 수행하고 난 후 *Adjust*() 함수를 이용하여 색인을 재 구성하게 된다. *Split*() 함수 및 *Adjust*()에 대한 설명은 3.4.3절에 나타나있다. 아직 삽입되지 못한 getSegmentInfo에 있는 정보들을 추출하여 재구성 과정이 종료된 ST-트리를 대상으로 *ST_insert*()를 호출하여 재 삽입하게 된다. 14행의 *setSegment*() 함수는 target_node가 full이 아닌 경우 호출되며 getSegmentInfo의 j번째 엔트리 값을 target_node의 맨 처음 비어있는 곳에 기록하고 target_node의 stb_ptr이 getSegmentInfo에 설정되어있는 *header Info*의 주소를 가리키게 하여 효과적인 다중복합검색을 수행할 수 있도록 한다.

```
함수명 : InsertSegment()
 1 : if (target_node의 엔트리 수 == 최대 엔트리 수)
 2 :     if (node_status == true){
 3 :         for (i = 0 부터 최대 엔트리 수 ){
 4 :             temp = compare(target_node, i, getSegmentInfo, j) ;
 5 :             comp = greater (temp, comp) ; }
 6 :         bulk_segment[comp][getSegmentInfo.j] ; }
 7 :     else
 8 :     {
 9 :         new_node = split (target_node, getSegmentInfo[j].id,
               getSegmentInfo[j].MBR, getSegmentInfo[j].Interval) ;
10 :         Adjust (new_node, target_node) ;
11 :         getSegmentInfo에 삽입되지 않고 남아있는 정보 추출
12 :         ST_insert(추출된 정보, 0, 0) ;
13 :     }
14 : else
15 :     setSegment(target_node, getSegementInfo, j) ;
```

(그림 12) InsertSegment() 함수



(그림 13) 삽입의 예

(그림 13)에서 제안하는 삽입 알고리즘의 예를 나타낸다. 기존의 많은 수의 노드 I/O에 대한 문제점을 해결하기 위해 각 노드에는 bulk_segment라는 임시 저장공간을 설정하였다.

ST-트리의 삽입은 기본적으로 STB-트리에 유지되는 하나의 노드 안에 포함된 이동 객체에 대한 세그먼트들을 단위로 한다. 여기에서는 (그림 5)의 다섯 개의 세그먼트를 예로 들겠다. 다섯 개의 세그먼트 O_{111}, O_{112}, O_{113}, O_{114}, O_{115}를 삽입하기 위해 먼저 노드 1에 있는 모든 엔트리와 O_{111}을 비교한다. 비교연산은 다음의 식 (9)와 같이 시공간 상에 위치한 객체의 크기를 계산한다.

$$size_{3d} = (x_u - x_l) * (y_u - y_l) + (t_e - t_s)^2 \qquad (9)$$

적절한 엔트리 E_1이 선택되면, E_1이 가리키는 포인터를 따라 하위레벨로 가는 것이 아니라 E_1의 엔트리 순서 정보 k를 가진 bulk_segment[k][i]에 삽입하려는 세그먼트 O_{111}을 저장한다. 그리고 O_{111}이 삽입됨으로써 E_1의 시공간영역이 확장되는 경우 노드 1에 있는 E_1의 시공간 정보를 변경한다. 다음에 삽입되는 세그먼트 O_{112}는 시공간 정보가 변경된 E_1과 다른 모든 엔트리들과 비교연산을 수행하여 bulk_segment에 유지한다. 이 과정을 삽입하려는 세그먼트가 더 이상 없을 때까지 수행한다. 삽입하려는 세그먼트가 더 이상 없으면 E_1이 가리키는 하위레벨 노드 2로 가서 E_1에 삽입되는 것이 적절하다고 판단된 O_{111}과 O_{113}을 노드 1에서와 같은 방법으로 삽입을 한다. E_1의 삽입이 완료되면 E_2에 삽입되기에 적절한 세그먼트를 삽입한다. 이러한 과정을 bulk_segment안에 있는 모든 세그먼트가 삽입될 때까지 수행한다.

제안하는 삽입방식은 하나의 노드에 대해 한번의 I/O연산으로 삽입연산 수행이 가능해 진다. 제안한 방법을 (그림 14)에 적용해 보면 노드 1과 노드 2, 노드 3에 대해 각각 1번씩의 노드 I/O를 수행하면 되고 전체적으로는 3번의 노드 I/O이면 삽입연산을 수행하는데 충분하다.

3.4.3 ST-트리의 분할

분할기법은 3DR-트리와 유사하며 ST-트리의 분할 기준은 이웃한 노드들 사이의 겹침이 가장 적은 위치를 선택한다. 분할 기준으로 사용되는 겹침 영역에 대한 연산은 식 (10)과 같이 그룹 O_1과 O_2의 영역이 겹치는 크기와 시간구간이 겹치는 크기의 제곱으로 겹침 영역의 크기를 계산한다.

$$overlap(O_1, O_2) = \{spatial(O_1) \cap spatial(O_2)\} + \{temporal(O_1) \cap temporal(O_2)\}^2 \qquad (10)$$

(그림 14)에 분할 알고리즘이 나타나있다. 2행에서 삽입 대상의 id, mbr, interval의 정보를 가지고 있는 엔트리 e를 설정한다. 3행의 삽입 대상 노드인 target에 있는 모든 각 엔트리와 삽입 대상이 되는 엔트리 e와 시공간 크기를 비교하기 위한 구문이다. 5행의 *space*(a, b)는 전달인자로 주어지는 엔트리 a, b의 시공간 정보를 이용하여 a와 b를 하나로 그룹화 하였을 때의 크기를 리턴하며 6행의 *area*(a)는

전달인자로 주어지는 엔트리 a의 시공간 정보를 이용하여 자신의 크기를 리턴하는 기능을 가진 함수이다. 5행에서 target 노드에 들어있는 i번째 엔트리와 삽입 대상 엔트리 e를 하나로 그룹화 하였을 경우의 크기를 Joinspace에 저장한다. 6행에서는 Joinsapce에서 각각의 크기를 뺀 형태로 사장공간이 가장 큰 경우의 엔트리가 구해지면 Group1에는 target 노드의 i번째 엔트리를 설정하고 Group2에는 삽입 대상이 되는 엔트리 e를 설정한다. 여기서 worst의 초기값은 0이다. 이렇게 3행부터 9행까지의 for문을 통하여 삽입 대상이 되는 엔트리 e와 그룹되기에 가장 적합하지 않은 target 노드의 i번째 엔트리를 구할 수 있다. 10행에서 target 노드의 전체 엔트리 E에서 Group1과 Group2에 있는 엔트리를 제거한 나머지 엔트리를 E'에 설정한다. 15, 16행에서 E'에 속해있는 i번째의 엔트리와 Group1, Group2를 이용하여 서로 그룹화 되었을 때의 크기를 d1, d2에 설정한다. 17행 및 19행에서 d1, d2의 크기 관계를 비교하여 18행 및 20행의 변수를 설정한다. 여기서 expansion의 초기값은 0이다. 설정된 엔트리들을 이용하여 22행에서 best_group를 구할 수 있으며 이 best_group이 서로 그룹되기에 적절한 엔트리들이라는 의미를 가진다. 23행에서는 E'에 있는 엔트리 중 best_group에 속하는 best_entry를 제거한 E'을 설정하고 24~27행에서 Group1 및 Group2에 있는 엔트리의 수를 한 노드에 저장될 수 있는 엔트리의 최소수 m에서 E'에 포함된 엔트리 수를 뺀 값이 같으면 25행 및 27행에서 Group1 및 Group2와 E'을 그룹화한다. 이러한 과정을 target 노드의 모든 각 엔트리가 그룹화될 때까지 계속되어진다.

```
함수명 : Split()
 1 : {
 2 :     e = id, mbr, interval 정보를 가지고 있는 엔트리 ;
 3 :     for (i = 0부터 target 노드에 있는 모든 엔트리에 대해)
 4 :     {
 5 :         Joinspace = space(target -> ei, e) ;
 6 :         if ((area(Joinspace) - area(target -> ei)
                 - area(e)) > worst){
 7 :             worst = (area(Joinspace) - area(target -> ei)
                         - area(e))
 8 :             Group1 = target -> ei ; Group2 = e ; }
 9 :     }
10 :     E' = E - {Group1 ∪ Group2} ;
11 :     while (E'이 빌 때 까지)
12 :     {
13 :         for (int j = 0부터 E'에 있는 모든 엔트리에 대해)
14 :         {
15 :             d1 = area(space(ei, Group1) - area(ei)) ;
16 :             d2 = area(space(ei, Group2) - area(ei)) ;
17 :             if (d1 - d2 > expansion)
18 :                 best_group = Group1 ; best_entry = ei ;
                                 expansion = d1-d2 ;
19 :             else if (d2 - d1 > expansion)
20 :                 best_group = Group2 ; best_entry = ei ;
                                 expansion = d1-d2 ;
21 :         }
22 :         best_group = best_group ∪ best_entry ;
23 :         E' = E' - {best_entry} ;
24 :         if (Group1에 있는 엔트리 수 = m - E'안에 있는
               엔트리 수)
25 :             Group1 = Group1 ∪ E' ;          E' = null ;
26 :         if (Group2에 있는 엔트리 수 = m - E'안에 있는
               엔트리 수)
27 :             Group2 = Group2 ∪ E' ;          E' = null ;
28 :     }
29 :     Group화 되지 않은 엔트리가 target에 존재하면 goto 3 ;
30 : }
```

(그림 14) Split() 알고리즘

```
함수명 : Adjust()
 1 : {
 2 :     if (node == isRoot())
 3 :         CreateNewRoot(node, nnode) ;
 4 :     else
 5 :     {
 6 :         parent = GetParent(node) ;
 7 :         AdjustEntry(parent, node -> id, node -> MBR,
                         node->Interval) ;
 8 :         InsertInNode(parent, nnode -> id, nnode -> MBR,
                         nnode -> Interval) ;
 9 :         if (parent가 overflow되면)
10 :             Split (parent, nnode -> id, nnode -> MBR,
                         nnode -> Interval) ;
11 :         else
12 :             AdjustPath(parent) ;
13 :     }
14 : }
```

(그림 15) Adjust() 알고리즘

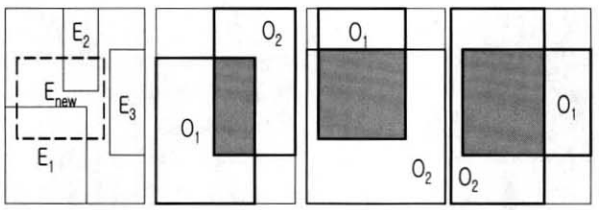
분할 후 트리를 조정할 필요가 있는데 이때 사용되는 Adjust 알고리즘이 (그림 15)와 같다. *Adjust*()는 두 개의 전달인자를 가지며 하나는 분할 후 새로 생긴 new_node이고 다른 하나는 분할 대상 노드인 target_node가 된다. 2행에서 분할 대상이 되었던 노드가 Root 노드인지를 판별하여 Root 노드이면 새로운 Root노드를 만들기 위해 *CreateNewRoot*() 함수를 호출한다. 이 *CreateNewRoot*() 함수는 분할 대상이 되었던 노드와 분할 후 새로 생긴 노드를 전달인자로 받으며 Root노드를 대신할 새로운 노드를 생성한다. 새로운 노드가 생성이 되면 *AdjustEntry*() 함수를 이용하여 분할 대상이 되었던 노드 및 분할 후 새로 생긴 노드의 시공간 영역 정보를 포함할 수 있는 엔트리를 구성한다. 6행에서 만일 node가 Root 노드가 아니면 *GetParent*() 함수를 이용하여 부모 노드의 정보를 얻어와서 parent에 설정한다. *GetParent*() 함수는 하나 이상의 부모노드가 있는 자식 노드를 전달인자로 가진다. 10행에서 *AdjustEntry*() 함수를 이용하여 부모노드에 있는 엔트리 정보를 조정하게 된다. 전달인자로는 조정될 대상이 되는 parent와 parent 노드에 새로 조정될 node의 id, MBR, Interval을 가진다. 이 세 개의 정보를 이용하여 기존 parent 노드에서 node를 가리키고있던 엔트리의 시공간 정보를 변경하게 된다. 또한 10행에서는 *InsertInNode*() 함수가 사용되는데 이 함수는 분할 후 새로 생성된 nnode의 시공간 정보를 parent 노드에 설정하기 위한 함수이다. 10행을 수행하고 나면 parent 노드에 새로운 엔트리가 삽입되는 결과를 가지게 되므로 parent 노드에 overflow가 발생할 가능성이 있다. 11행에서는 overflow가 발생하면 overflow가 발생된 parent 노드와 overflow

를 발생시킨 엔트리 정보를 이용하여 *Split*() 함수를 호출한다. 만일 overflow가 발생되지 않았다면 14행에서 *AdjustPath*() 함수를 parent 노드를 전달인자로 하여 호출한다. *AdjustPath*()는 현재 노드에 대한 엔트리의 시공간 정보 변경으로 인해 Root 노드 방향으로 해당되는 엔트리들의 시공간 정보를 변경하는 함수이다. 이에 대한 설명은 (그림 16)에서 한다.

(그림 16)에는 *AdjustPath*()에 대해 나타나있다. *AdjustPath*()는 node를 전달인자로 받으며 node는 분할로 인해 새로운 엔트리 정보가 삽입된 노드의 부모노드를 가리킨다. 2행에서 node가 Root이면 더 이상 엔트리 정보를 변경할 필요가 없으므로 리턴하고 그렇지 않은 경우 6행에서 현재 node의 부모 노드정보를 가져와서 *AdjustEntry*() 함수를 이용하여 부모노드에 현재 노드의 시공간 정보 등을 조정하게 된다. 이 과정이 성공적으로 종료가 되면 다시 *AdjustPath*() 함수를 호출하여 *AdjustEntry*() 함수의 기능을 Root까지 진행한다.

```
함수명 : AdjustPath()
 1 : {
 2 :     if (node == isRoot())
 3 :         return ;
 4 :     else
 5 :     {
 6 :         parent = GetParent(node) ;
 7 :         if (AdjustEntry(parent, node -> id, node -> MBR,
                             node -> Interval))
 8 :             AdjustPath(parent) ;
 9 :     }
10 : }
```

(그림 16) AdjustPath() 알고리즘



(a) E_{new} 삽입 (b) E_{new} + E_1 (c) E_{new} + E_2 (d) E_{new} + E_3

(그림 17) 분할의 예

(그림 17)은 분할 알고리즘의 예이다. 한 노드에 삽입될 수 있는 최대 엔트리 수를 3개로 가정하였을 때 (그림 17)(a)는 이미 세 개의 엔트리로 노드가 다 찼는데 E_{new}가 새로 삽입이 되어 오버플로가 발생하는 경우를 나타내고 있다. (그림 17)(b)에서는 E_{new}가 E_1과 함께 그룹 O_1을 구성하고 나머지 엔트리 E_2와 E_3가 그룹 O_2를 구성하는 모습이다. 또한 (그림 17)(c)에서는 E_{new}가 E_2와 함께 그룹 O_1을 구성하는 모습이며 (그림 17)(d)에서는 E_3와 함께 그룹 O_1을 구성하는 모습으로서, 그룹 O_1과 O_2가 겹치는 부분을 검게 칠해진 영역으로 확인할 수 있다. (그림 17)의 경우만 보면 단순히 공간 측면의 겹침만을 고려하여 설명되었지만 식(10)에서처럼 시간 측면을 함께 고려하여 분할되어질 그룹을 선택하여야 한다.

3.5 검 색

3.5.1 검색 유형

기존에 제안된 검색은 크게 범위검색과 궤적검색으로 나눌 수 있다. 먼저 범위검색은 주어지는 시공간 범위 안에서 검색을 수행한다. 대표적인 시공간 검색형태는 "2시에서 3시 사이에 인천공항에 나타난 객체를 검색하라"이다. 궤적검색은 주어지는 시공간 범위 안에서 특정 이동 객체의 궤적을 검색하는 검색이다. 궤적검색의 대표적인 형태는 "2시에서 3시 사이에 인천공항에 나타난 객체의 궤적을 검색하라"이다. 궤적검색의 형태를 보면 시공간 검색과 특정 객체의 궤적정보를 나타내는 검색이 결합된 형태를 나타낸다. 이런 형태의 검색을 결합 검색이라 한다.

기존의 시공간 색인구조들은 결합검색을 처리하기 위해 범위검색을 수행한 후 다시 궤적검색을 처리하는 형태를 가진다. 이동 객체의 궤적 보존 방법을 사용하는 TB-트리의 경우 동일한 이동 객체의 궤적정보를 가지는 단말노드들이 양방향 연결 리스트 형태로 연결 되어있기 때문에 먼저 범위검색을 처리하고 여기에 속하는 이동 객체의 단말노드에 대해 포인터를 따라 궤적을 검색하는 방법을 취한다. 그러므로 TB-트리는 결합검색을 처리하기 위해 재 검색을 수행하는 기존의 시공간 색인 구조보다는 적은 노드 접근으로 결합검색을 처리할 수 있다는 장점이 있다. 그러나 검색된 이동 객체의 과거 혹은 미래의 위치정보, 궤적정보 등을 알고 싶으면 검색을 다시 수행해야 한다는 문제점이 있다.

본 논문에서는 비디오 데이터는 현재시점으로부터 과거뿐만 아니라 비디오 데이터의 논리적 이야기 전개가 끝나는 시점까지의 모든 객체정보를 알고 있다는 특징을 이용하여 재검색 과정에 소요되는 비용을 줄이고 효과적으로 궤적검색을 수행할 수 있도록 다중복합검색을 제안한다. 다중복합검색은 기존에 제안된 결합검색을 확장한 형태이다. 다중복합검색의 검색형태는 "2시부터 3시 사이에 인천공항에 나타난 객체의 궤적과 30분전(후)의 궤적 및 위치를 검색하라"이다.

3.5.2 검색 알고리즘

(그림 18)에 검색 알고리즘이 나타나있다. 검색은 크게 범위검색과 궤적 검색로 나누어서 처리한다. 검색을 처리하기 위한 *Search*() 함수는 네 개의 매개변수를 갖는다. 첫 번째 매개변수 *N*은 현재 노드정보이다. *range1*과 *range2*는 범위검색을 처리하기 위한 시간 및 공간범위이며 마지막 매개변수 *range3*는 다중복합검색을 처리하기 위한 궤적 정보이다. 예를 들어 "2시부터 3시 사이에 인천공항에 나타난 객체의 궤적과 30분전의 위치를 검색하라"라는 검색에서 '2시부터 3시 사이'가 *range1*에 속하며, '인천공항'이 *range2*에 속하게 된다. 마지막으로 *range3*는 '30분전의 위치'를 의미한다.

(그림 18)의 2행과 8행에서 현재 노드 *N*이 단말노드인지 아닌지를 검사한다. 단말노드가 아니면 범위검색을 처리하기

위하여 5행에서 N안에 있는 모든 엔트리와 *range1*, *range2*가 겹치는 엔트리가 있는지 검사를 한다. 여기에서 *range1*과 *range2*를 만족하는 i번째 엔트리를 찾으면 6행에서 i번째 엔트리가 가리키는 하위레벨 노드 N'을 서브 루트 노드로 하여 *range1*, *range2*를 가지고 N'이 단말노드일 때까지 검사를 반복하며 8행에서 만일 N이 단말노드이면 N에 포함된 모든 엔트리에 대해 *range1*, *range2*가 겹치는지와 range3이 null인지 아닌지를 검사한다. 12행에서처럼 만일 range3이 null이면 단순 시공간 질의로서 겹침을 가지는 N의 엔트리는 *range1*, *range2*의 조건에 맞는 데이터를 가진 검색 결과가 된다. 또한 14행에서처럼 *range3*가 null이 아니면 겹침을 가지는 N의 엔트리가 가리키는 stb_ptr의 위치에 있는 노드에 *range3*를 이용하여 궤적 검색을 수행한다. *Search_Trajectory*() 함수는 (그림 16)에 나타나 있다. 마지막으로 시공간 질의처리에서 최악의 경우 실제 데이터가 존재하지 않음에도 불구하고 실제 데이터가 있는 것처럼 인식되어 검색을 수행하게 되는 경우가 존재하는데 16행에서처럼 단말노드까지 *range1*, *range2*의 질의 조건을 만족하여 검색을 수행했으나 실제 검색 대상이 존재하지 않는 경우 return하게 된다.

```
함수명 : Search()
 1 : {
 2 :   if (N != 단말노드)
 3 :   {
 4 :       for (i = 0부터 N에 포함된 엔트리 수){
 5 :          range1, range2가 겹치는 엔트리가 있는지 검사 ;
 6 :          if (존재하면)
                엔트리가 가리키는 하위레벨 노드 N'을 루트로
                하여 다시 검사 ; }
 7 :   }
 8 :   else if (N == 단말노드)
 9 :   {
10 :       for (j = 0부터 N에 포함된 엔트리 수){
11 :          range1, range2가 겹치는 엔트리가 있는지 검사 ;
12 :          if (겹침이 발생하면서 range3가 NULL)
13 :             해당 엔트리가 시공간 검색의 결과 ;
14 :          else if (겹치면서 range3가 NULL이 아니면)
15 :             Search_Trajectory() 함수를 이용하여 range3를
                 만족하는 궤적검색 ;
16 :          else return ;
17 :       }
18 :   }
19 : }
```

(그림 18) 검색 알고리즘

(그림 19)의 *Search_Trajectory*() 함수는 두 개의 매개변수를 가진다. 첫 번째 매개변수 ptr은 *range1*과 *range2*를 만족하는 엔트리가 STB-트리의 *Header info* 노드를 가리키는 정보이며 두 번째 매개변수는 이동 객체의 궤적정보이다. 궤적검색 알고리즘의 수행은 2행에서 ptr이 가리키는 곳에 있는 *Header info* 노드 정보를 읽어들인다. 3행에서 Header Info의 모든 각 엔트리와 *range3*가 겹치는 엔트리가 있는지 검사한다. 5행에서 겹치는 엔트리가 있으면 이 엔트리가 가리키는 단말 노드의 위치 POS를 offset에 설정하고 for문을 빠져 나온다. 3행부터 7행까지는 Header Info에서 range3와 겹치는 맨 처음의 엔트리를 찾아내서 해당 엔트리가 가리키는 단말노드의 위치를 알고자 하는 것이다. 다음 8행과 9행에서는 일단 offset의 위치에 있는 단말노드의 정보를 읽어들이고 10행과 11행에서 읽어들인 단말노드의 모든 각 엔트리가 *range3*와 겹침이 있는지를 검사한다. 12행에서 겹치는 엔트리가 있으면 해당 엔트리의 궤적정보를 유지한다. 13행에서는 10행의 for문이 종료시점인지를 판단한다. 즉 10행의 for문을 통해 현재의 단말노드의 모든 각 엔트리와 *range3*와 비교를 마쳤으면 offset의 위치를 바로 인접한 다음노드를 가리키게 한다. 15행에서 while()문은 단말노드가 *range3*와 더 이상 겹침이 없을 때까지이며, do~while()문을 탈출할 조건이 되면 12행에서 유지시킨 궤적정보를 16행에서 반환하고 종료한다. 이렇게 함으로써 *Header Info*에서 *range3*와 겹침을 가지는 단 하나의 첫 번째 엔트리를 구하게 되면, 이 엔트리의 ptr를 따라 단말노드를 읽어서 *range3*와 겹침이 있는 엔트리가 있는지를 비교한다. 이때 만일 *Header Info*에 *range3*와 겹치는 엔트리가 다수 존재하더라도 첫 엔트리가 가리키는 ptr의 단말노드의 위치만 알면 나머지 단말노드들은 물리적으로 인접한곳에 위치하므로 다시 *Header Info*에서 검색하지 않아도 쉽게 접근할 수 있다.

```
함수명 : Search_Trajectory()
 1 : {
 2 :    headerInfo = getHeaderInfo(ptr) ;
 3 :    for (i = 0부터 headerInfo에 포함된 엔트리 수만큼){
 4 :          headerInfo의 엔트리와 range3가 겹치는 엔트리가
              있는지 검사 ;
 5 :          만일 range3와 겹치는 엔트리를 찾으면
              해당 엔트리의 POS를 offset에 설정 ;
 6 :          break ;
 7 :    }
 8 :    do {
 9 :          offset의 위치에 있는 단말노드 leaf를 읽어들임 ;
10 :          for (j = 0부터 leaf에 포함된 엔트리 수만큼){
11 :               range3와 겹치는 엔트리가 있는지 검사 ;
12 :               if (겹치는 엔트리가 있으면) 해당 엔트리의
                        궤적 정보를 유지 ;
13 :               if (j가 leaf에 포함된 엔트리 수이면)
14 :                    offset은 NextLeaf 노드 ; }
15 :    } while (range3의 범위를 벗어날 때까지)
16 :    return 궤적 정보 ;
17 : }
```

(그림 19) 궤적검색 알고리즘

4. 성능 평가

4.1 실험 환경

본 장에서는 제안된 MVTB-트리를 구현하여 실험한 성능평가 결과를 기술한다. MVTB-트리의 검색성능을 평가하기 위한 방법으로 범위검색 처리를 위해 접근하는 노드 수와 다중복합검색을 처리하기 위해 접근하는 노드 수를 측정하였다. MVTB-트리의 검색성능을 평가한 환경은 Pentium-Ⅲ 500㎒ CPU와 메인 메모리 256MB, 리눅스 커널 2.4.9를 가지는 시스템을 이용하여 C언어로 구현하였다. 제안한 알고리즘을 구현하여 성능 평가를 하기 위해 고려한

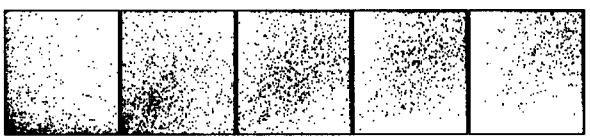
파라미터들은 <표 1>과 같다.

〈표 1〉 실험 데이터

파라미터	값
최초 노드 크기(K)	1
이동 객체 수	10~1000
궤적 / 이동 객체	100~1000
ST-트리 내부노드 팬 아웃	18
ST-트리 단말노드 팬 아웃	18
STB-트리 팬 아웃	15

4.2 실험 데이터

실험 데이터는 MVTB-트리의 비교 평가대상인 TB-트리와 동일하게 GSTD(Generate SpatioTemporal Data)를 이용하여 데이터 집합을 생성하였다[16]. 생성된 이동 객체의 궤적분포는 (그림 20)과 같이 이동 객체는 전반적으로 남서쪽에서 북동쪽으로 이동하고 있다. 이동 객체는 10~1000개까지 생성하였고 각 이동 객체당 100~1000개의 궤적을 가지도록 하였다.



(그림 20) 실험 데이터

색인구조를 위해 사용되는 노드의 크기는 1KB이며 ST-트리의 단말노드와 비단말노드의 팬아웃은 18이고, STB-트리는 하나의 노드에 15개의 궤적정보를 저장하도록 하였다. 모든 검색성능 측정은 실험 데이터를 이용해 색인을 구성한 후 실험 데이터 중 100개를 검색 대상으로 사용하여 평균 노드 접근 수를 측정한 것이다.

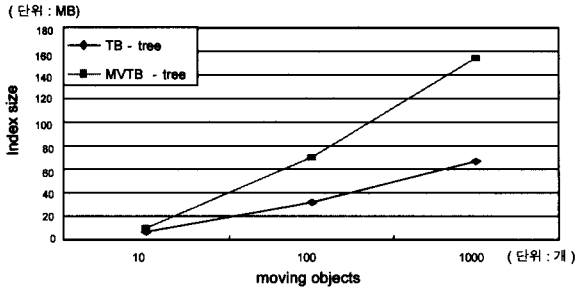
4.3 색인 크기

〈표 2〉 색인의 크기 및 특성

색인의 크기 및 특성	MVTB-트리		TB-트리
	ST-트리	STB-트리	
공간 활용률 (%)	70	80	90
객체당 궤적 수 (개)	1000		1000
색인 크기 (MByte)	71	83	67

GSTD를 통해 생성한 데이터를 통해 생성한 제안하는 색인 구조와 TB-트리의 특성은 <표 2>와 같다. <표 2>에서 보는 것과 같은 제안하는 색인 구조는 STB-트리라는 보조적인 색인 구조를 사용하기 때문에 TB-트리에 비해 공간 활용률(space utilization)은 저하되고 색인의 크기는 대략 2배 정도 증가된다.

(그림 21)은 TB-트리와 제안하는 MVTB-트리의 색인크기를 나타낸 것이다. 이동 객체의 수가 적으면 색인구조의 크기는 거의 차이가 없다. 그러나 이동 객체의 수가 1000개인 경우를 보면 약 20MB정도의 크기차이가 발생한다. MVTB-트리의 색인구조 크기가 더 큰 이유는 보다 효과적인 시공간 궤적검색을 위해 STB-트리와 ST-트리에 동일한 이동 객체의 시공간 정보를 유지하기 때문이라고 판단된다.



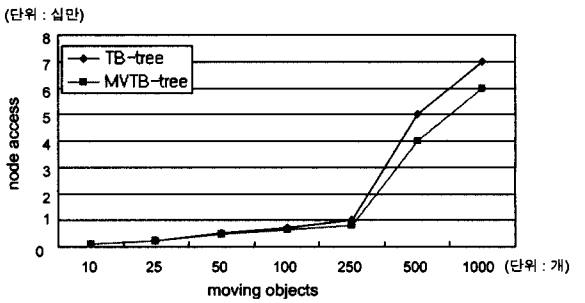
(그림 21) 색인 크기 비교

4.4 범위검색

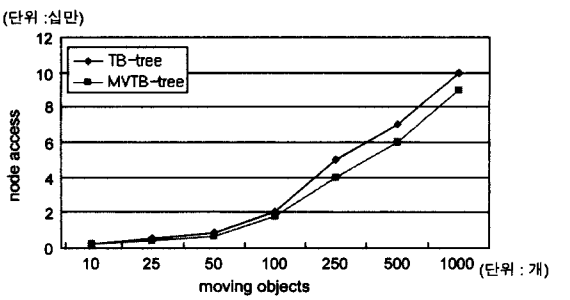
범위검색은 시공간 데이터에서 중요한 검색형태 중 하나이다. 범위검색에 대한 성능평가는 고정된 공간차원에서 시간차원이 변경되는 경우와 고정된 시간차원에서 공간차원이 변경되는 경우 두 가지 방향으로 평가하였다.

4.4.1 고정된 공간차원에서 시간차원의 변경

(그림 22)와 (그림 23)은 시간차원이 전체 시간구간 중 10%, 20%의 범위를 가지는 경우에 대한 결과이다. (그림 22)에서는 이동 객체가 250개일 때까지 TB-트리와 MVTB-트리가 접근하는 노드 수가 근사하다. 이동 객체가 500개일 때부터 MVTB-트리가 TB-트리 보다 노드 접근 수를 작다



(그림 22) 시간축에 대해 10% 검색



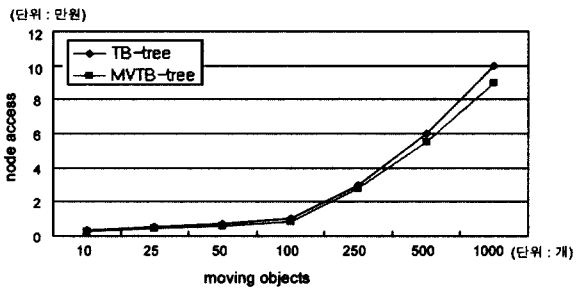
(그림 23) 시간축에 대해 20% 검색

는 것을 확인할 수 있다. 또한 (그림 23)은 근사하게 MVTB-트리의 노드 접근 수가 적다.

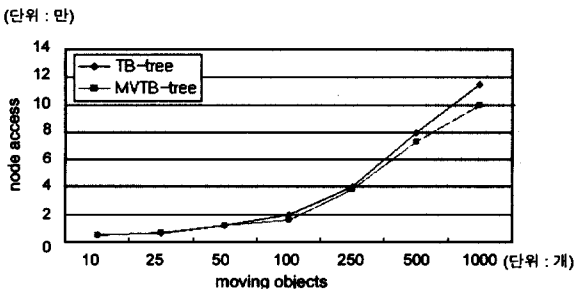
TB-트리의 경우 이동 객체가 위치하기에 적절한 노드이더라도 궤적을 강제로 유지하기 위해서 새로운 단말노드를 생성하고 생성된 단말노드에 이동 객체의 궤적정보를 위치시키는 궤적 보존 방법을 사용한다. 이때 동일한 이동 객체의 궤적을 보존하기 위해서 새로운 단말노드를 생성하게 되며 MVTB-트리는 시공간 객체의 궤적을 삽입할 때 서로 인접한 시공간범위에 삽입하기 때문에 범위질의 연산시 TB-트리 보다는 적은 수의 노드에 접근한다고 판단된다. 또한 MVTB-트리는 이동 객체의 궤적을 STB-트리의 연속된 공간에 할당하고 각각의 궤적을 ST-트리에 삽입하는 형태로 구성된다. 이 ST-트리에서는 하나의 이동 객체에 대한 전체 궤적을 저장한 STB-트리를 직접 접근할 수 있도록 포인터를 연결이 되어있다. 따라서 하나 이상의 노드에 저장된 세그먼트를 한번의 노드 접근으로 접근이 가능한 것도 MVTB-트리가 적은 수의 노드에 접근하는 이유라고 판단된다.

4.4.2 고정된 시간차원에서 공간차원의 변경

(그림 24)와 (그림 25)는 전체 공간영역 중 10%와 20%를 검색 범위로 설정했을 경우의 결과이다. 이 경우도 TB-트리보다 MVTB-트리가 노드 접근을 적게 하고있다. 주된 이유는 4.4.1절에서도 언급한 TB-트리의 궤적 보존 방법에 의한 새로운 단말노드의 생성과 MVTB-트리의 Header Info 구조 때문이라고 판단된다. 또한 제안하는 MVTB-트리에서는 사장공간을 감소시켜 색인구조를 구성하기 때문에 범위검색 처리에서 TB-트리보다 적은 수의 노드 접근을 한다고 판단된다.



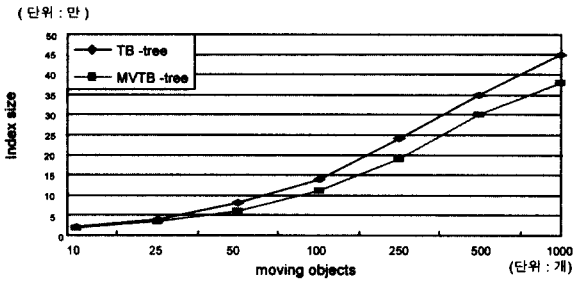
(그림 24) 공간축에 대해 10% 검색



(그림 25) 공간축에 대해 20% 검색

4.5 다중복합검색

Header Info 구조를 이용한 다중복합검색 처리에 대한 결과가 (그림 26)에 나타나있다. 다중복합검색이란 앞에서도 언급했듯이 이동 객체에 대한 궤적을 검색하고 검색된 이동 객체를 대상으로 임의시간 전·후에 나오는 시공간 궤적을 검색하는 검색형태이다. (그림 23)을 보면 이동 객체의 수가 250일 때부터 MVTB-트리가 다중복합검색 수행시 TB-트리보다 훨씬 적은 노드를 접근하는 것을 볼 수 있다. TB-트리는 양방향 연결 리스트를 사용하여 이동 객체의 궤적검색을 수행하는데 이동 객체의 궤적이 여러 노드의 단말노드에 걸쳐 존재한다면 양방향 연결 리스트를 따라 순차적으로 이동 객체의 궤적이 존재하는 모든 노드를 I/O해야 한다. MVTB-트리의 경우 STB-트리의 연속적으로 할당된 공간에 이동 객체의 궤적정보를 유지하는 방법을 취하였다. 또한 *Header Info*에서 연속적으로 할당된 공간의 정보를 별도로 유지한다. 그러므로 다중복합검색 처리시 TB-트리처럼 포인터를 따라 이동하는 것이 아니라 *Header Info*에 한번만 접근하여 요구하는 궤적이 저장된 노드를 쉽게 찾을 수 있으므로 TB-트리보다 훨씬 적은 노드를 접근한다고 판단된다.



(그림 26) 다중복합검색 처리

5. 결론 및 향후 연구

본 논문에서는 비디오 데이터에 나타나는 이동 객체의 시공간 궤적을 효과적으로 검색할 수 있는 MVTB-트리를 제안하였다. 제안된 색인구조는 이동 객체의 궤적별로 세그먼트화 하여 저장함으로써 사장공간을 감소시켜 검색처리 성능을 향상 시켰다. MVTB-트리는 *Header Info* 및 연속된 공간에 이동 객체의 궤적정보를 유지하는 특징이 있다. 이런 특징은 이동 객체의 과거나 미래의 시점에 대한 시공간 궤적검색인 다중복합검색에 대해 노드 I/O에 대한 성능을 약 18%정도 향상시키게 하였다.

향후 연구로는 벌크 로딩 기법을 이용하여 색인을 구성하는 기법에 대한 연구와 공간 활용률을 좀더 개선하고 다양한 검색 유형을 지원하기 위한 연구를 계속 수행할 예정이다.

참 고 문 헌

[1] J. Z. Li, M. T. Ozsu and D. Szafron, "Modeling of Moving

Objects in a Video Database," Proc. IEEE International Conference on Multimedia Computing and Systems, pp.336-343, 1997.
[2] M. Nabil, A. H. H. Ngu and J. Shepherd, "Modeling and Retrieval of Moving Objects," Multimedia Tools and Applications Vol.13, No.1, pp.35-71, 2001.
[3] Ramez Elmasri, Gene T. J. Wuu and Yeong-Joon Kim, "The time index : An access structure for temporal data," Proc. 16th International Conference on VLDB, pp.1-12, August, 1990.
[4] Vram Kouramajian, Ibrahim Kamel, Ramez Elmasri and Syed Waheed, "The time index+ : An Incremental access structure for temporal database," Proc. Third International Conference on Information and Knowledge Management, pp.296-303, 1994.
[5] A. Guttman, "R-trees : A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD International Conference on Management of Data, pp.47-57, 1984.
[6] 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 International Conference on Management of Data, pp.322-331, 1990.
[7] I. Kamel and C. Faloutsos, "Hilbert R-tree : an Improved R-tree Using Fractals," Proc. 20th International Conference on VLDB, pp.500-509, 1994.
[8] T. Selis, N. Roussopoulos and C. Faloutsos, "The R+ tree : A Dynamic Index for Multi-Dimensional Object," Proc. 13rd International Conference on VLDB, pp.507-518, 1987.
[9] Mario A. Nascimento and Jefferson R. O. Silva, "Towards Historical R-trees," Proc. ACM symposium on Applied Computing, pp.235-240, 1998.
[10] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An Asymptotically Optimal Multi-version B-tree," VLDB Journal, Vol.5, No.4, pp.264-275, 1996.
[11] Y. Tao and D. Papadias, "MV3R-Tree : A Spatio-Temporal Access Method for Timestamp and Interval Queries," Proc. 27th International Conference on VLDB, pp.431-440, 2001.
[12] Y. Tao and D. Papadias, "Efficient Historical R-trees," Proc. IEEE International Conference on Scientific and Statistical Database Management, pp.223-232, 2001.
[13] Y. Theodoridis, R. Silva and T. Sellis, "SpatioTemporal Indexing for Large Multimedia Applications," Proc. the 3rd IEEE International Conference on Multimedia Computing and Systems, pp.441-448. 1996.
[14] D. Pfoser, C. Jensen and Y. Tehodoridis, "Novel Approaches to the Indexing of Moving Object Trajectories," Proc. 26th International Conference on VLDB, pp.395-406, 2000.
[15] G. Kollios, D. Gunopulos, V. Tsotras, A. Delis and M. Hadjieleftheriou, "Indexing Animated Objects Using Spatio-Temporal Access Methods," Proc. IEEE Trans. Knowledge and Data Engineering, pp.742-777, 2001.
[16] Y. Theodoridis, R. Silva and M. Nascimento, "On the Generation of Spatiotemporal Datasets," Proc. 6th International Symposium on Spatial Databases, pp.147-164, 1999.

이 낙 규

e-mail : true01@netdb.chungbuk.ac.kr
2001년 충북대학교 정보통신공학과 공학사
2003년 충북대학교 전기전자 컴퓨터공학부 공학석사
2003년~현재 충북대학교 전기전자컴퓨터 공학부 및 컴퓨터정보통신연구소 박사과정
관심분야 : XML, 멀티 미디어 검색, 자료 저장 시스템, 시공간 색인 구조

복 경 수

e-mail : ksbok@netdb.chungbuk.ac.kr
1998년 충북대학교 수학과 이학사
2000년 충북대학교 정보통신공학과 공학석사
2000년~현재 충북대학교 전기전자컴퓨터 공학부 및 컴퓨터정보통신연구소 박사과정
관심분야 : 자료 저장 시스템, 멀티미디어 데이터베이스, 이동 객체 데이터베이스, 고차원 색인 구조, 시공간 색인 구조 등

유 재 수

e-mail : yjs@cbucc.chungbuk.ac.kr
1989년 전북대학교 컴퓨터공학과 공학사
1991년 한국과학기술원 전산학과 공학석사
1995년 한국과학기술원 전산학과 공학박사
1995년~1996년 목포대학교 전산통계학과 전임강사
1996년~현재 충북대학교 전기전자컴퓨터공학부 및 컴퓨터정보통신연구소 부교수
관심분야 : 데이터베이스 시스템, 정보검색, 멀티미디어 데이터베이스, 분산 객체 컴퓨팅 등

조 기 형

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