DOI: 10.3745/KIPSTD.2009.16-D.4.487

영역 질의의 효과적인 처리를 위한 궤적 인덱싱

차 창 일[†]·김 상 욱[††]·원 정 임[†††]

요 약

본 연구에서는 대용량 궤적 데이터베이스에서 영역 질의를 효과적으로 처리하기 위한 인덱싱 기법에 대하여 논의한다. 먼저, 기존 인덱싱 기법의 문제점을 지적하고, 이러한 문제점을 해결하는 새로운 기법을 제안한다. 제안된 기법에서는 우선 시간 차원을 다수의 시간 구간으로 분할하고, 인덱싱의 대상이 되는 전체 라인 세그먼트들을 시간 구간별로 구분한다. 각 시간 구간에 속하는 라인 세그먼트들에 대하여 별도의 인덱스를 구축한다. 또한, 디스크에서 관리되는 과거 시간 구간에 대한 인덱스들과는 달리 최근 시간 구간에 대한 인덱스는 메인 메모리상에 관리함으로써 삽입과 검색의 성능을 크게 개선할 수 있다. 각 시간 구간에 속하는 라인 세그먼트들은 다음과 같은 방식으로 인덱스를 구축한다. 먼저, 2D-트리를 이용하여 전체 공간 차원을 유사한 수의 라인 세그먼트들이 배정되도록 다수의 셀들로 분할한다. 또한, 분할된 각 셀마다 시공간 차원 (x, y, t)에 대한 별도의 3차원 R^*-트리를 두어 보다 상세한 인덱싱을 지원한다. 이와 같은 다양한 전략을 이용함으로써 기존 기법의 문제점들을 해결 할 수 있다. 다양한 실험을 통하여 제안된 기법의 우수성을 정량적으로 검증한다. 실험 결과에 의하면, 기존 기법에 비하여 작은 인덱스 구조를 갖으면서도 검색 성능면에서 3~10배까지의 성능 향상 효과를 갖는 것으로 나타났다.

키워드 : 이동 객체, 궤적 데이터, 시공간 인덱스, 영역 질의

Trajectory Indexing for Efficient Processing of Range Queries

Chang-Il Cha[†] · Sang-Wook Kim[††] · Jung-Im Won[†††]

ABSTRACT

This paper addresses an indexing scheme capable of efficiently processing range queries in a large-scale trajectory database. After discussing the drawbacks of previous indexing schemes, we propose a new scheme that divides the temporal dimension into multiple time intervals and then, by this interval, builds an index for the line segments. Additionally, a supplementary index is built for the line segments within each time interval. This scheme can make a dramatic improvement in the performance of insert and search operations using a main memory index, particularly for the time interval consisting of the segments taken by those objects which are currently moving or have just completed their movements, as contrast to the previous schemes that store the index totally on the disk. Each time interval index is built as follows: First, the extent of the spatial dimension is divided onto multiple spatial cells to which the line segments are assigned evenly. We use a 2D-tree to maintain information on those cells. Then, for each cell, an additional 3D R^*-tree is created on the spatio-temporal space (x, y, t). Such a multi-level indexing strategy can cure the shortcomings of the legacy schemes. Performance results obtained from intensive experiments show that our scheme enhances the performance of retrieve operations by 3~10 times, with much less storage space.

Keywords : Moving Object, Trajectory Data, Spatiotemporal Index, Range Query

1. 서 론

최근, 휴대폰 및 PDA 등의 휴대용 이동 단말기의 확산과 통신 기술의 발달로 인하여 이동하는 객체의 시간 흐름에 따른 공간적 위치 정보를 활용하는 다양한 서비스들이 제공되고 있다[11]. 시공간 데이터(spatio-temporal data)란 공간 (x, y)과 시간(t)의 특성을 함께 가지고 있는 3차원 공간 (x, y, t)상의 데이터를 의미한다. 객체들의 이동 경로는 3차원의 시공간상에서 라인 세그먼트들의 집합으로 표현할 수 있으며, 이를 궤적(trajectory)이라 부른다[5, 7, 9, 10, 11]. 이동 객체의 궤적 정보를 잘 분석하면, 단순한 위치 추적 뿐만 아니라 도로 정보, 차량 정보 등과 연계하여 이동 객체와 관련된 사람, 행동 패턴 등을 파악할 수 있으며, 이를 다

※ 이 논문은 2007년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원(KRF-2007-314-D00221, KRF-2007-313-D00651), 2008년도 정부(교육과학기술부)의 재원으로 한국과학재단의 지원(R01-2008-000-20872-0)과 지식경제부 및 정보통신연구진흥원의 대학 IT연구센터 지원(IITA-2009-C1090-0902-0040)으로 수행되었습니다.
† 정 회 원 : 포인트아이(주)
†† 종신회원 : 한양대학교 정보통신학부 교수(교신저자)
††† 정 회 원 : 한양대학교 정보통신학부 연구교수
논문접수 : 2009년 1월 15일
수 정 일 : 1차 2009년 4월 1일
심사완료 : 2009년 4월 1일

양한 비즈니스에 활용할 수 있다.

이동 객체에 대한 사용자 질의는 크게 이동 객체의 미래 위치를 예측하여 검색하는 미래 예측 질의(future query)와 이동 객체의 과거에 움직인 위치를 검색하는 과거 이력 질의(historical query)로 구분된다. 미래 예측 질의의 경우 이동 객체의 현재 위치, 이동 속도, 이동 방향 등을 이용하여 미래 위치를 예측하는 것이 일반적이다[14, 11, 17]. 과거 이력 질의는 영역 질의(range query), 궤적 질의(trajectory query), 복합 질의(complex query)로 구분된다[9]. 영역 질의는 주어진 질의 영역 내에 존재하는 이동 객체를 검색하는 질의이며, 궤적 질의는 주어진 시간 간격 동안에 이동 객체가 움직인 경로를 검색하는 질의이다. 복합 질의는 영역 질의와 궤적 질의를 결합한 형태로 주어진 시간 간격 동안에 특정 영역에 있었던 이동 객체의 궤적을 검색하는 질의이다. 본 논문에서는 이들 중 영역 질의를 연구의 대상으로 한다.

이동 객체의 궤적 정보는 다음과 같은 특징을 갖는다. (1) 시간의 흐름에 따라 공간적 위치가 지속적으로 변화하므로 갱신 비용이 크다. (2) 지속적으로 축적되어야 하므로 저장 공간 오버헤드가 매우 크다. (3) 축적된 대용량의 궤적 정보를 대상으로 하므로 검색 비용이 매우 크다. 이동 객체의 이러한 특징들을 고려하여 대용량 궤적 정보를 신속하게 저장, 관리, 검색할 수 있는 기술이 요구된다.

미래 예측 질의를 위한 인덱스 구조로는 VCI-트리[14], TPR-트리[11], TPR*-트리[17] 등이 제안된 바 있다. 이 중 가장 널리 이용되고 있는 TPR*-트리는 이동 객체들의 현 위치들에 대한 MBR(minimum bound rectangle)과 이동 속도를 함께 저장하는 CBR(conservative bounding rectangle)을 R*-트리[1] 내에 저장함으로써 이동 객체의 미래 위치를 빠르게 검색할 수 있도록 한다. 과거 이력 질의를 위한 인덱스 구조로는 3DR-트리[13], HR-트리[13], STR-트리[9], TB-트리[9], MV3R-트리[15], SETI[4] 등이 있다. 3DR-트리, HR-트리, SETI는 영역 질의, STR-트리, TB-트리, MV3R-트리는 궤적 질의를 그 주요 대상으로 한다. 본 연구의 대상인 영역 질의에서는 SETI가 인덱스 크기, 갱신 비용, 검색 성능 면에서 우수한 구조로 알려져 있다[8, 12].

본 연구에서는 대용량 궤적 데이터베이스에서 영역 질의를 효과적으로 처리하기 위한 새로운 인덱싱 기법을 제안한다. 제안된 기법은 다음과 같은 이동 객체 데이터베이스의 두 가지 특성에 대한 관찰에 기반을 둔다. 첫째, 이동 객체의 궤적을 구성하는 라인 세그먼트들은 수집된 시간 순서대로 저장 및 인덱싱 된다. 따라서 저장되는 라인 세그먼트들의 시간 차원 t의 값은 단순히 증가되는 특성을 갖는다. 둘째, 영역 질의의 대상은 오래 전의 과거 세그먼트들 보다는 최근의 세그먼트들이 될 가능성이 높다.

제안된 기법에서는 우선 시간 차원을 다수의 시간 구간으로 분할하고, 인덱싱의 대상이 되는 전체 라인 세그먼트들을 구간별로 구분한다. 각 시간 구간에 속하는 라인 세그먼트들을 대상으로 별도의 인덱스를 사용한다. 다수의 최근 시간 구간과 대응되는 인덱스는 메인 메모리 내에서 관리되며, 나머지 시간 구간과 대응되는 인덱스들은 디스크 내에서 관리된다. 모든 라인 세그먼트의 삽입은 메인 메모리 내에서 관리되는 최근 시간 구간 인덱스 내에서 이루어지므로 빠른 처리가 가능하다. 또한, 최근 라인 세그먼트들에 대한 질의 특성상 빠른 질의 처리가 가능하게 된다.

각 시간 구간에 속하는 라인 세그먼트들에 대한 인덱싱은 다음과 같이 처리된다. 먼저, 전체 공간 차원을 다수의 셀들로 분할한다. 전체 공간에 대한 분할 정보는 2D-트리[2]를 통하여 관리되며, 각각의 셀에 유사한 수의 라인 세그먼트들이 배정되도록 하는 분할 정책을 사용한다. 이를 통하여 라인 세그먼트 수에 차이가 없으므로 일정한 검색 성능을 제공할 수 있다. 또한, 분할된 각 셀마다 시공간 차원 (x, y, t)에 대한 별도의 3차원 R*-트리를 두어 보다 상세한 인덱싱을 지원한다.

다양한 실험에 의한 성능 평가를 통하여 제안된 기법의 우수성을 규명한다. 실험 결과에 의하면 제안된 기법은 SETI에 비하여 작은 인덱스 구조를 갖으면서도 검색 성능 면에서 3~10배까지의 성능 향상 효과를 갖는 것으로 나타났다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 관련 연구로서 SETI에 대하여 설명하고, 그 장단점을 지적한다. 제 3 장에서는 새로운 인덱스 구조를 제안하고, 이를 기반으로 하는 질의 처리 알고리즘을 제시한다. 제 4 장에서는 실험을 통하여 제안된 기법의 우수성을 입증한다. 마지막으로, 제 5 장에서는 본 논문을 요약하고, 결론을 내린다.

2. 관련 연구

본 장에서는 본 연구에서 비교 대상으로 하는 SETI에 대하여 간략히 요약하고, 그 문제점을 지적한다.

2.1 SETI의 특성

이동 객체의 궤적 T_i는 (moId, tId, <seg_1, seg_2, ..., seg_k>)로 구성된다. 여기서, moId는 이동 객체의 식별자이며, tId는 궤적 식별자이다. $seg_j(0 < j \le k)$ 는 궤적 T_i을 구성하는 라인 세그먼트들이며, (sId, (x_{start}, y_{start}, t_{start}), (x_{end}, y_{end}, t_{end}))로 구성된다. 여기서 sId는 세그먼트의 식별자이며, 해당 객체가 시간 구간 t_{start}와 t_{end}에서 (x_{start}, y_{start})으로부터 (x_{end}, y_{end})로 이동하였음을 의미한다. 본 논문에서는 시간이 흐름에 따라 발생된 객체의 라인 세그먼트가 인덱스에 지속적으로 삽입된다고 가정한다.

SETI[4]는 궤적들을 인덱싱하여 주어진 질의 영역 내에 존재하는 이동 객체를 신속하게 검색할 수 있도록 한다. SETI에서는 궤적의 시간 차원 값은 지속적으로 증가하는 반면, 공간 차원 값은 거의 변화하지 않는다는 특징을 이용한다. 먼저, 전체 공간 영역을 논리적인 셀(cell) 단위로 분할한 후, 각 셀 영역마다 희소 시간 인덱스(sparse time index)를 개별적으로 구성한다. 즉, 몇 개의 라인 세그먼트

를 하나의 그룹으로 묶어서 데이터 페이지에 저장하고, 데이터 페이지 내의 라인 세그먼트들의 시간 차원 값들 중 최소 시간과 최대 시간을 회소 시간 인덱스의 엔트리로 저장한다. 이때 데이터 페이지에는 (tid, sid, x_{start}, y_{start}, t_{start}, x_{end}, y_{end}, t_{end})의 구조를 갖는 아이템을 저장하는 방식을 사용한다. 회소 시간 인덱스 구조로는 1차원 R^*-트리를 사용한다. 이러한 접근 방식은 기존의 3DR-트리의 단점인 사장 영역과 영역 중첩 문제점을 해결함으로써 인덱스 크기를 감소시키고, 검색 성능을 향상시킬 수 있다.

SETI는 이동 객체의 궤적에 추가되는 새로운 라인 세그먼트를 인덱스에 보다 빠르게 삽입하기 위하여 해쉬 테이블을 이용한다. 해쉬 테이블에는 이동 객체가 움직인 가장 최근 위치가 저장된다. 만약, 삽입하려는 이동 객체의 궤적 T_i의 위치가 A(x_{start}, y_{start}, t_{start})에서 A′(x_{end}, y_{end}, t_{end})로 변경되었다면 해쉬 테이블에서 A를 검색 한 후, A와 A′를 이용하여 새로운 라인 세그먼트를 구성한다. 이 라인 세그먼트가 속하는 셀 영역을 찾은 후, 이와 대응되는 회소 시간 인덱스를 검색하여 해당 데이터 페이지에 이 라인 세그먼트를 삽입한다. 이때, 해쉬 테이블의 궤적 T_i의 위치는 A에서 A′으로 갱신된다. (그림 1)은 SETI의 전체 구조를 나타낸 것이다.

SETI에서의 검색 연산은 여과 단계, 정제 단계, 제거 단계로 나누어진다. 여과 단계(filtering step)에서는 주어진 질의 내 공간 조건을 만족하는 셀 영역을 검색한 후, 이와 대응되는 회소 시간 인덱스를 검색하여 질의 내 시간 조건을 만족할 가능성이 큰 후보 라인 세그먼트들이 저장되어 있는 데이터 페이지를 파악한다. 정제 단계(refinement step)에서는 여과 단계에서 얻어진 이 데이터 페이지를 액세스하여 후보 라인 세그먼트 중 질의내 시공간 조건을 모두 만족하는 라인 세그먼트들의 궤적 식별자를 얻는다. 제거 단계(elimination step)에서는 정제 단계에서 얻어진 궤적 식별자들 중 중복되는 식별자들을 제거하여 최종 결과를 얻는다.

2.2 SETI의 문제점

본 연구에서는 먼저 SETI가 가지는 다음과 같은 문제점을 지적한다.

(1) 회소 시간 인덱스의 사용

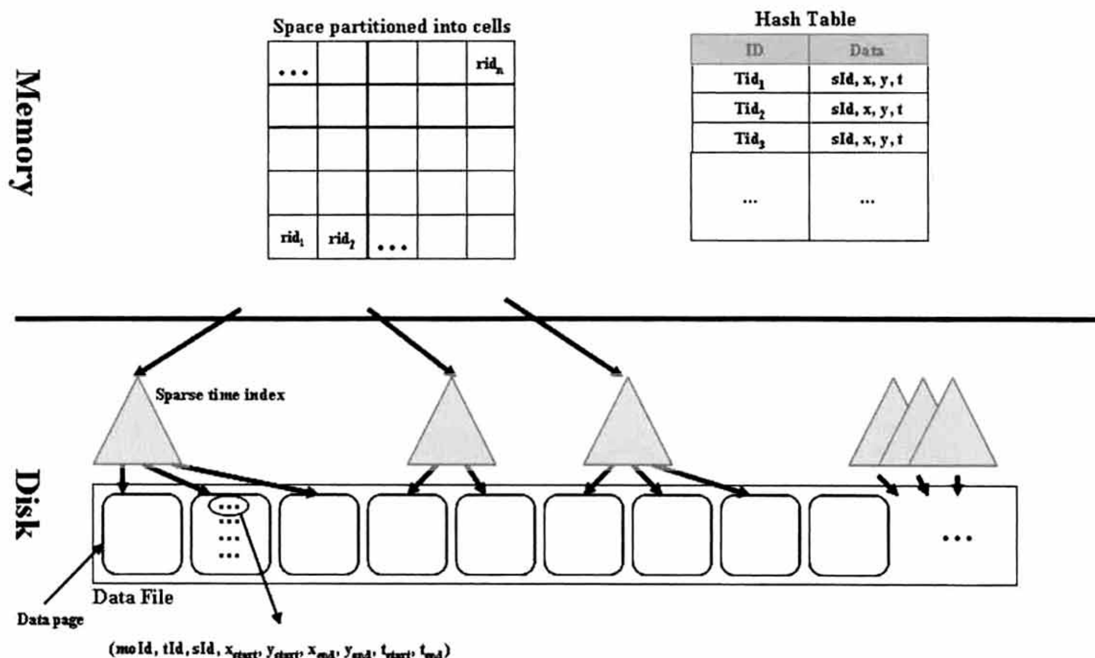
회소 시간 인덱스를 사용하기 때문에 여과 단계에서는 엔트리가 가리키는 페이지 내의 라인 세그먼트들이 질의 내 시간 조건을 만족하는지의 여부를 파악할 수 없다. 따라서 정제 단계를 통하여 해당 데이터 페이지 내의 모든 후보 라인 세그먼트들을 대상으로 질의 내 시간 조건의 만족 여부를 따져야하는 오버헤드가 발생한다.

(2) 라인 세그먼트 집중

이동 객체들은 특정 영역에 집중되는 경향이 있으므로 라인 세그먼트들의 삽입 역시 이동 객체의 궤적이 특정 셀 영역에 집중되는 현상이 발생할 수 있다. 이로 인하여 라인 세그먼트들이 집중된 셀과 대응되는 회소 시간 인덱스에는 과부하가 걸릴 가능성이 크다. 이 결과, 일정한 검색 성능을 제공하기에는 어려움이 있다.

(3) 공간 인덱스의 미사용

분할된 셀 내부에 대하여는 별도의 공간 인덱스를 사용하지 않으므로 후보 라인 세그먼트의 수가 크게 증가하게 된



(그림 1) SETI의 구조

다. 예를 들어, 주어진 질의 내 공간 조건 영역이 셀 영역의 공간 범위보다 매우 작은 경우, 희소 시간 인덱스에 대한 탐색 결과 질의 조건 영역 외부에 해당되는 다수의 라인 세그먼트들이 후보로 추천되며, 이 결과 검색 성능이 저하된다.

3. 제안하는 기법

본 장에서는 전술한 SETI의 문제점을 해결할 수 있는 새로운 인덱싱 기법을 제안한다. 제 3.1절에서는 제안하는 기법의 기본 개념을 설명하고, 제 3.2절에서는 제안된 인덱스의 구조를 설명한다. 제 3.3과 제 3.4절에서는 각각 인덱스 구조를 기반으로 하는 삽입 알고리즘과 질의 처리 알고리즘을 설명한다.

3.1 기본 개념

이동 객체의 궤적을 구성하는 라인 세그먼트들은 수집된 시간 순서대로 저장 장치에 저장 된다. 또한, 오래전의 과거 세그먼트보다는 현재 이동 중인 세그먼트나 최근에 이동이 완료된 세그먼트가 검색 대상이 될 가능성이 높다. 본 논문에서는 이동 객체의 이러한 특징을 이용하여 라인 세그먼트를 시간 흐름에 따라 효과적으로 삽입 및 검색할 수 있는 인덱싱 기법을 제안한다.

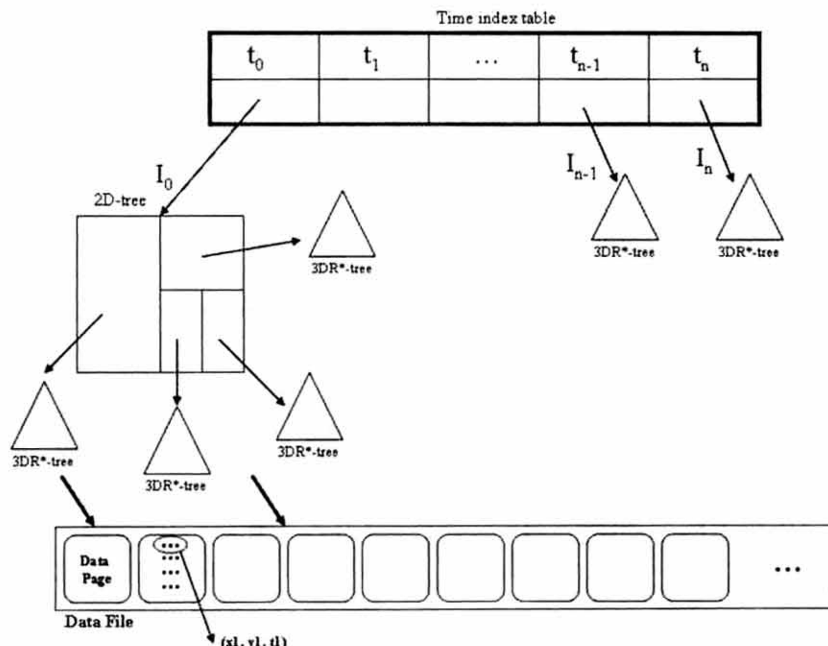
제안된 기법에서는 SETI와는 반대로 라인 세그먼트들을 우선 시간 차원으로 분할하고, 분할된 각 시간 구간에 속하는 라인 세그먼트들을 다시 공간 차원으로 분할하는 기법을 사용한다. 이와 같이, 최상단에서 시간 차원을 기반으로 분할하면 현재 이동 중인 이동 객체에 대한 라인 세그먼트의 삽입 연산은 마지막 시간 구간에서만 발생하게 된다. 제안하는 기법에서는 대부분의 인덱스는 디스크 내에서 관리하는 반면, 삽입 연산이 발생하는 마지막 시간 구간에 대한 인덱스는 메인 메모리 내에 상주시킨다. 이 결과 SETI에서와 같이 삽입 연산을 위하여 디스크 내의 희소 시간 인덱스에 대한 랜덤 액세스 문제를 해결할 수 있다.

삽입 연산을 담당하는 인덱스는 메인 메모리의 크기를 고려하여 미리 정해진 k개의 라인 세그먼트만을 삽입할 수 있도록 제한한다. 즉, 인덱스에 k번째의 라인 세그먼트에 대한 삽입이 이루어지면, 해당 인덱스를 메인 메모리에서 제거하고 디스크에 저장한다. 이후, 이 인덱스에 대하여는 검색 연산만이 발생하게 된다. 이 인덱스는 해당 시간 구간에 대하여 생성된 메인 메모리 내의 인덱스를 공간 차원으로 재분할하여, 분할된 m개의 공간 영역마다 서브 인덱스를 갖는 다단계 다중 인덱스 구조(mutilevel multiple indexes)이다. 제안된 기법에서는 분할된 각 셀이 $\lceil k/m \rceil$ 개의 라인 세그먼트를 저장하는 비 균일 셀 분할을 지원하며, 전체 공간 영역에 대한 분할 정보를 2D-트리[2]를 이용하여 관리한다. 이로써 SETI의 라인 세그먼트 집중, 공간 인덱스 미사용 등의 문제점을 해결할 수 있다.

또한, 제안된 기법에서는 검색 성능을 보다 향상시키기 위하여 분할된 공간 셀 영역 마다 작은 크기를 갖는 (x, y, t)의 3차원 R-트리를 두어 세부 인덱싱을 지원한다.

3.2 인덱스 구조

(그림 2)는 제안하는 인덱스의 구조를 나타낸 것이다. 먼저, 시간 차원을 분할 한 정보를 관리하는 시간 인덱스 테이블을 구성한다. 시간 인덱스 테이블은 n+1개의 엔트리로 구성되며, 각 엔트리는 대응되는 인덱스의 디스크 상의 위



(그림 2) 제안하는 인덱스 구조

치를 저장한다. 즉, n+1개의 서로 다른 인덱스 $I_0, I_1, ..., I_n$를 인덱스 테이블과 함께 관리한다. 시간 인덱스 테이블은 크기가 작으므로 메인 메모리에 상주시킬 수 있다. 이때, 각 인덱스 I_i는 미리 정해진 k개의 라인 세그먼트만을 저장한다.

여기서 주의할 점은 시간 구간 $t_0, t_1, ..., t_{n-2}$과 대응되는 인덱스 $I_0, I_1, ..., I_{n-2}$은 이동이 완료된 과거 궤적을 관리하기 위한 인덱스로 검색 연산만이 지원된다. 인덱스 I_i ($0 \le i \le n-2$)는 시간 구간 t_i를 2차원의 공간 차원을 기반으로 m개의 작은 셀 영역으로 재분할하여 저장한 인덱스로, 분할된 각 셀은 (그림 2)에서 나타난 바와 같이 $\lceil k/m \rceil$ 개의 라인 세그먼트를 저장하는 가변 크기를 갖는다. 인덱스 I_i는 해쉬 테이블이나 2D-트리 구조를 사용하여 구현될 수 있으며, 시공간 데이터 (x, y, t)에 대하여 해당 셀은 해당 영역 내에 존재하는 라인 세그먼트들을 대상으로 구축한 3차원 R^*-트리 인덱스의 저장 위치를 엔트리로 갖는다. 또한, 3차원 R^*-트리는 $\lceil k/m \rceil$ 개의 라인 세그먼트들에 대한 궤적 식별자와 세그먼트 식별자를 엔트리로 갖으며, 하나의 데이터 페이지에 동일 궤적의 세그먼트가 시간순으로 저장되므로 tid를 데이터 페이지의 헤더에 한번만 저장하고, (x, y, t)의 구조를 갖는 아이템을 저장하는 방식을 사용한다.

반면, 시간 구간 t_{n-1}과 t_n에 대응되는 인덱스 I_{n-1}과 I_n은 이동 중인 객체의 최근 세그먼트에 대한 삽입 연산과 검색 연산을 담당하며, 시공간 데이터 (x, y, t)에 대하여 구성된 3차원 R^*-트리로 구성된다. 본 연구에서는 삽입 및 검색 연산의 효율성을 증가시키기 위하여 이들 인덱스 I_{n-1}과 I_n를 미리 정해진 k개의 라인 세그먼트만을 저장하도록 하여 메인 메모리 내에 상주시킨다. 여기서 시간 구간 t_{n-1}의 해당 인덱스 I_{n-1}을 이동이 완료된 과거 궤적을 위한 인덱스로 구성하여 디스크 내에서 관리하지 않고 메모리에 상주시켜 관리하는 이유는 시간 간격이 긴 세그먼트가 삽입될 때 세그먼트가 쪼개져서 일부는 디스크 내의 인덱스로, 나머지 일부는 메모리에 구성 중인 인덱스로 삽입되어 삽입 성능을 저하시키는 것을 방지하기 위함이다.

3.3 삽입 알고리즘

Algorithm 1은 삽입 알고리즘 InsertSegment를 나타낸 것이다. 알고리즘은 라인 세그먼트 seg와 인덱스에 저장될 수 있는 최대 라인 세그먼트의 개수 k개를 입력 받아 $3DR^*$-트리를 이용하여 구성되는 current-index와 prev-index, 그리고 공간 분할 정보를 관리하는 2D-트리, 시간 인덱스 테이블 IT를 출력한다. 라인 4~7에서 현재 시간 구간의 최소 시간과 최대 시간 정보를 유지하기 위해 이동 객체의 라인 세그먼트 seg의 시간 정보를 이용하여 이를 갱신하고 insertToIndex() 함수를 이용하여 해당 인덱스에 삽입한다. 그 후, 인덱스 내에 존재하는 라인 세그먼트의 개수(numSeg)를 증가시킨다. 만약, 현재 numSeg의 값이 미리 지정된 k보다 크고, n 값이 2보다 크다면 메인 메모리 내에 구성 중인 시간 구간 t_{n-1}에 해당하는 인덱스 prev-index를 writeToDisk() 함수에 의하여 디스크에 저장한다. 이때, 검색 연산의 효율성을 고려하여 각 인덱스를 공간 차원으로 분할하여 재구성하고, 분할된 공간 영역에 대한 정보를 2D-트리에 저장한다(라인 10~11). 다음, add2DIndex()함수를 이용하여 시간 인덱스 테이블 IT의 엔트리에 디스크에 저장된 2D-트리의 저장 위치와 삽입된 라인 세그먼트들 내에서의 최소 시간과 최대 시간을 저장한다(라인 12). 다음, 시간 구간 t_n에 해당하는 인덱스 current-index를 prev-index로 변경한다. 끝으로, 현재 삽입되지 않은 라인 세그먼트를 삽입하기 위하여 메모리 내에 $3DR^*$-트리를 새로이 생성하고, 삽입 한다(라인 13~19).

Algorithm 1: InsertSegment

Input: Segment Info seg=(tId, (x_{start}, y_{start}, t_{start}), (x_{end}, y_{end}, t_{end})), k
Output: prev-index, current-index, 2D-tree, time index table IT

```
1. n = 1; IT = NULL; numSeg = 1;
2. createEmptyIndex(current-index);
3. if(numSeg < k)
4.      if(numSeg == 1) CurrentMinTime = t_start; CurrentMaxTime = t_end;
5.      else(t_end >= CurrentMaxTime) CurrentMaxTime = tend;
6.      insertToIndex(current-index, seg);
7.      numSeg++;
8. else
9.      if(n >= 2)
10.         create2DIndex(2D-tree[n-1]);
11.         writeToDisk(prev-index, 2D-tree[n-1]);
12.         add2DIndex(IT, PrevMinTime, PrevMaxTime, 2D-tree[n-1]);
13.      prev-index = current-index;
14.      PrevMinTime = CurrentMinTime;
15.      PrevMaxTime = CurrentMaxTime;
16.      current-index = NULL;
17.      n = n + 1; numSeg = 1;
18.      createEmptyIndex(crrent-index);
19.      insetToIndex(current-index, seg);
20. return;
```

3.4 검색 알고리즘

Algorithm 2는 영역 질의 알고리즘 RangeSearch를 나타낸 것이다. 알고리즘은 입력으로 $3DR^*$-트리를 사용하여 구성된 시간 구간 t_n에 해당하는 current-index, t_{n-1}에 해당하는 prev-index, t_0부터 t_{n-2}까지에 해당하는 disk-index 그리고 공간 분할 정보를 저장하고 있는 2D-트리와 시간 인덱스 테이블 IT, 마지막으로 질의 Q가 주어진다. 출력으로는 주어진 질의 Q를 만족하는 결과 집합 A를 출력한다. 검색은 먼저 시간 인덱스 테이블 IT에서 주어진 질의의 시간 구간에 해당하는 2D-트리를 선택하고(라인 2), 선별된 |N|개의 2D-트리에 대하여 라인 3~13의 과정을 반복 수행한다. 이때, 질의의 시간 구간이 prev-index의 시간 구간에 해당한다면 메인 메모리상의 prev-index를 검색하고, current-index의 시간 구간에 해당한다면 current-index를 검색한다(라인 5~8). 질의 시간 구간이 prev-index 시간 구간의 최소 시간보다 작다면 다음의 검색 과정을 따른다. 선별된 2D-트리에서 질의의 공간 영역과 중첩되는 셀을 getOverlappingCell()함수를 이용하여 검색한 후, 검색된 셀과 대응되는 디스크 상의 disk-index에 대한 탐색을 진행한다. 질의의 공간 영역에

포함되는 모든 셀에 대하여 이 과정을 반복 수행한다(라인 10~11).

Algorithm 2: RangeSearch

```
Input: prev-index, current-index, disk-index, 2D-tree, time index table IT,
       Query Q((x_start, y_start, t_start), (x_end, y_end, t_end))
Output: Set of answers A

1. A = { };
2. N = searchIT(IT, Q(t_start, t_end));
3. for(i = 0; i < |N|; i++) do
4.     if(N[i].MinTime > PrevMinTime)
5.         if(N[i].MinTime > CurrentMinTime)
6.             S = searchIndex(current-index, Q((x_start, y_start, t_start), (x_end,
                   y_end, t_end)));
7.         else
8.             S = searchIndex(prev-index, Q((x_start, y_start, t_start), (x_end,
                   y_end, t_end)));
9.     else
10.        while(cid = getOverlappingCell(2D-tree[N[i]])) do
11.            S = searchIndex(disk-index[N[i]][cid], Q((x_start, y_start, t_start),
                   (x_end, y_end, t_end)));
12.    end if
13.    addAnswer(A, S(sId, tId));
14. return A;
```

4. 성능 평가

본 장에서는 실험에 의한 성능 분석을 통하여 제안하는 기법의 우수성을 규명한다. 제 4.1절에서는 실험 환경을 설명하고, 제 4.2절에서는 실험 결과를 분석한다.

4.1 실험 환경

실험에서는 객체 유형과 객체들의 분포 유형을 사용자 정의 파라미터로 설정하여 시공간 데이터를 생성하는 GSTD (Generate SpatioTemporal Data)[16] 알고리즘의 시나리오 4(rectangles moving)에 의하여 생성된 [0, 1]사이에서 균일한 분포를 취하는 1,000개의 궤적 데이터를 사용한다. 여기서 시나리오 4는 초기에 이동 객체가 공간상의 중앙에 분포해 있다가 랜덤하게 균일 분포로 이동하는 알고리즘이다. 다양한 세그먼트 개수에 따른 실험을 수행하기 위하여 각 궤적 데이터에 대하여 100만개(1M), 200만개(2M), 400만개(4M), 800만개(8M), 1,600만개(16M)의 세그먼트로 구성된 다섯 개의 실험 데이터 셋을 사용한다.

질의 영역으로는 실험 데이터로 사용된 궤적 데이터로부터 추출된 임의의 점에 대하여 0.01%, 0.1%, 1%의 영역 허용 범위를 갖는 질의 영역을 각각 1,000개씩 생성하여 사용한다. 각 실험 데이터와 질의 영역에 대하여 1,000개의 영역 질의를 수행한 후, 나타난 평균 질의 처리 시간을 성능 평가 지수로 사용한다.

성능 평가는 다음 두 가지의 서로 다른 기법을 대상으로 한다. Ours는 본 논문에서 제안된 기법으로 라인 세그먼트들을 우선 시간 차원으로 분할하고, 분할된 각 시간 구간에 속하는 라인 세그먼트들을 다시 공간 차원으로 분할하는 방식이다. 성능 비교를 위한 기존의 기법으로서 SETI를 사용하며, 공간 차원에 대한 셀의 개수는 400(20×20)개이다. 이들 두 방식 모두에서 사용되는 다차원 인덱스는 GIST[3, 6]에서 제공하는 R^*-트리를 사용하며, 데이터 페이지 크기로는 2KB를 사용한다.

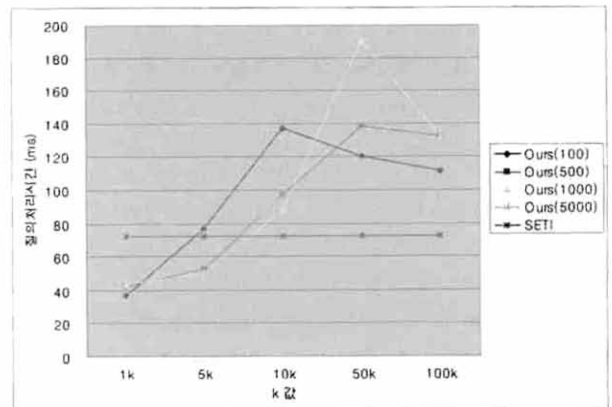
본 실험을 위한 플랫폼으로는 Windows 2003 Server를 운영 체제로 사용하고, 3GB의 주기억 장치를 갖는 Intel Xeon 노코다 3.0GHz의 Intel Server를 사용한다.

4.2 실험의 기본 파라미터 값 설정

먼저, 시간 차원을 위한 k값과 공간 차원을 위한 d값을 결정한다. k는 하나의 시간 구간내에 저장되는 세그먼트의 개수이며, d는 하나의 셀내에 들어가는 세그먼트의 개수이다. (그림 3)에 400만개의 세그먼트로 구성된 1,000개의 궤적 데이터에 대하여 k와 d값을 변화시키면서 질의 처리 시간을 측정한 결과를 보인다. 이때 질의 영역의 허용 범위는 0.1%이다. 가로축은 k의 크기로 1,000개부터 100,000개까지의 값을 사용한다. 세로축은 평균 질의 처리 시간을 나타내며 측정 단위는 msec이다. 실험 결과에 따르면, Ours는 대체적으로 k가 커질수록 질의 처리 시간이 증가하였다. 이는 k값의 증가는 하나의 시간 구간내에 속한 인덱스의 크기를 증가시키게 되므로 인덱스 검색 시간이 증가하기 때문이다.

또한, k값이 10,000개 이하에서는 d가 커질수록 질의 처리 시간이 감소하였다. 이는 d가 커지면 공간 차원에 해당하는 셀의 개수가 작아지게 되고, 따라서 액세스되는 인덱스의 개수가 작아지기 때문이다. 그러나 k가 증가하는 경우, 인덱스의 개수는 작아지는 반면에 인덱스의 크기는 오히려 커지게 되므로 이러한 성능 향상을 보장할 수는 없다.

SETI는 k와 상관없이 거의 일정한 질의 처리 시간을 보였다. 이는 SETI는 공간 차원에 대하여 정적 분할 방식을 사용하므로 셀의 크기만을 파라미터로 결정하고, 셀내에 저장되는 세그먼트의 개수 k는 설정하지 않기 때문이다. 이후



(그림 3) k값의 변화에 따른 질의 처리 시간 비교

실험에서는 k와 d의 기본 값을 각각 1,000개와 500개로 설정하여 실험한다.

4.3 실험 결과 및 분석

[실험 1] 인덱스 크기 비교

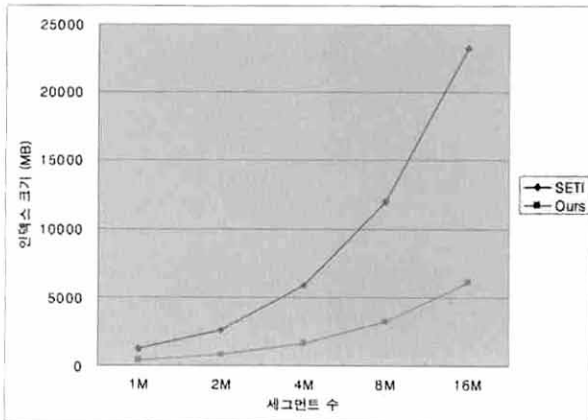
실험 1에서는 전체 인덱스 크기를 기준으로 하여 각 기법의 성능을 비교한다. 전체 인덱스의 크기는 인덱스 파일과 데이터 페이지 파일을 모두 합한 크기이다.

(그림 4)는 데이터의 크기 증가에 따르는 Ours와 SETI의 인덱스 크기 변화를 나타낸다. 가로축의 실험 데이터 크기는 궤적내 세그먼트의 개수를 나타낸다.

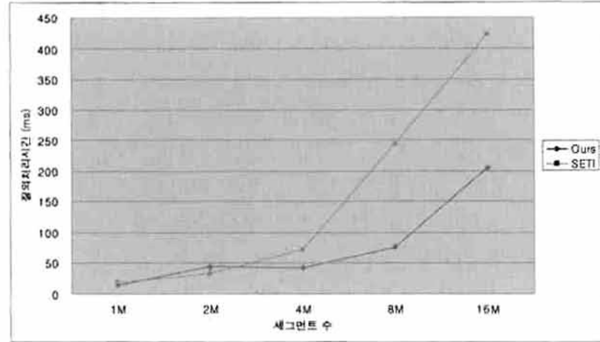
실험 결과로부터, 두 방식 모두 데이터의 크기 증가에 따라 인덱스 크기가 증가함을 알 수 있다. 그러나 절대 크기를 비교했을 때 Ours는 SETI에 비하여 3~4배까지의 저장 공간 감소 효과를 갖는 것으로 나타났다. 이는 SETI가 데이터 페이지에 (tid, sid, x_1, y_1, t_1, x_2, y_2, t_2)의 구조를 갖는 아이템을 저장하는 것과 달리 Ours는 하나의 데이터 페이지에 동일 궤적의 세그먼트가 시간순으로 저장되므로 tid를 데이터 페이지의 헤더에 한번만 저장하고, (x, y, t)의 구조를 갖는 아이템을 저장하는 방식을 사용하기 때문이다.

[실험 2] 질의 처리 시간 비교

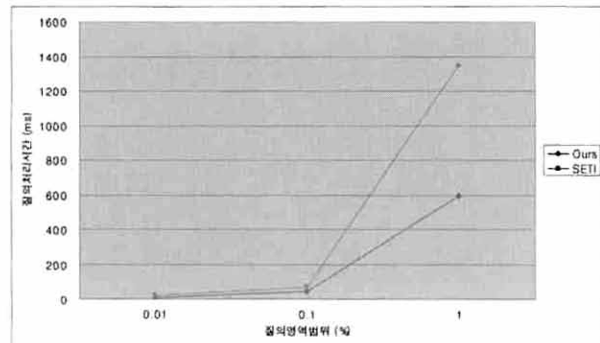
실험 2에서는 질의 처리 시간을 기준으로 각 기법의 성능을 비교한다. (그림 5)는 데이터의 크기 증가에 따르는 각 기법의 질의 처리 시간의 변화를 나타낸다. 가로축은 실험 데이터인 궤적내 세그먼트의 개수를 나타낸다. 질의 영역 허용 범위는 0.1%로 고정하였다. 세로축은 평균 질의 처리 시간을 나타내며 측정 단위는 msec이다. 실험 결과로부터 데이터의 크기 증가에 따라 두 방식 모두 질의 처리 시간이 증가하지만, Ours가 SETI에 비하여 질의 처리 성능이 매우 우수함을 알 수 있다. 이는 Ours는 시간 차원 뿐만 아니라 공간 차원에 대한 인덱스를 사용하므로, SETI에 비하여 작은 크기의 인덱스를 사용하게 되며, 이 결과 인덱스 검색 시간이 감소하기 때문이다. 또한, 데이터 페이지도 궤적별로 시간순으로 클러스터되어 저장되는 방식을 사용하기 때문에 버퍼링 효과를 극대화 할 수 있다. (그림 5)의 실험 결과로부터 Ours는 SETI에 비하여 약 30%~100%까지의 성능 향상 효과를 가지는 것으로 나타났다.



(그림 4) 데이터 크기 변화에 따른 인덱스 크기 비교



(그림 5) 데이터 크기 변화에 따른 질의 처리 시간 비교



(그림 6) 질의 영역 허용 범위에 따른 질의 처리 시간 비교

(그림 6)은 질의 영역 허용 범위의 변화에 대한 각 방식의 질의 처리 시간의 변화를 나타낸다. 실험에는 세그먼트의 개수가 400만개(4M)인 데이터를 사용한다. 가로축은 질의 영역 허용 범위를 나타내며, 그 범위로서 0.01%, 0.1%, 1%의 값을 각각 사용한다. 세로축은 평균 질의 처리 시간을 나타내며 측정 단위는 msec이다. 실험 결과로부터 질의 영역 허용 범위가 증가함에 따라 두 방식 모두 검색 대상이 되는 인덱스의 수가 증가하기 때문에 질의 처리 시간이 증가한다. 그러나 Ours가 SETI에 비하여 질의 처리 성능이 매우 우수함을 알 수 있다. 그 이유는 SETI는 희소 시간 인덱스를 사용하므로 질의 영역이 증가하게 되면 해당 데이터 페이지 내의 모든 후보 라인 세그먼트들을 대상으로 질의 내 시간 조건의 만족 여부를 판별하는 오버헤드가 급속하게 증가하기 때문이다. Ours는 SETI에 비하여 약 50%~120%까지의 성능 향상 효과를 가지는 것으로 나타났다.

[실험 3] 최근 질의에 따른 질의 처리 시간 비교

실험 3에서는 최근 질의에 대한 질의 처리 시간의 변화를 비교 및 평가한다. (그림 7)은 전체 질의에 대하여 최근 질의의 비율이 10%부터 90%까지 변화하는 상황에서의 질의 처리 시간을 나타낸다. 실험에는 세그먼트 개수가 400만개

(4M)인 궤적 데이터를 사용하며, 질의 영역 허용 범위는 0.1%로 설정한다. 최근 시간 구간내에 존재하는 궤적 데이터로부터 임의 추출하여 최근 질의를 생성한다. 실험 결과에 의하면, SETI는 최근 질의의 비율에 거의 영향을 받지 않고 일정한 질의 처리 시간을 보이는 반면, Ours는 최근 질의의 비율이 증가할수록 질의 처리 시간이 감소함을 알 수 있다. 그 이유는 SETI는 모든 세그먼트들을 디스크상에 인덱스로 구성하여 저장하는 반면, Ours는 최근에 삽입된 세그먼트들은 최근 시간 구간에 대한 인덱스로 구성하여 메인 메모리상에 관리하기 때문이다. (그림 7)의 결과에 따르면 Ours는 SETI에 비하여 100%~1000%까지의 성능 개선 효과를 갖는 것으로 나타났다.

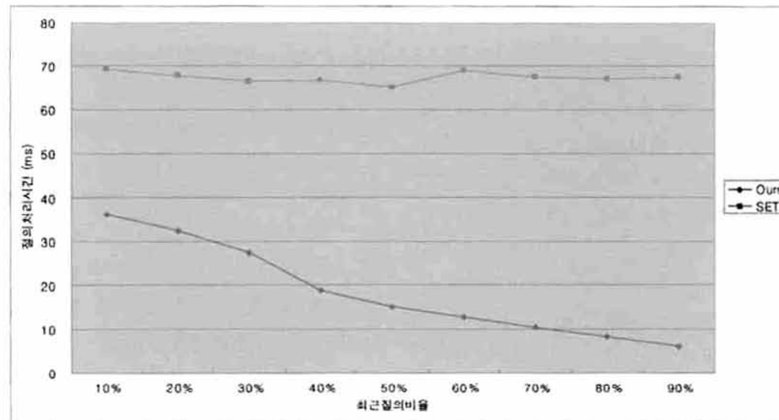
[실험 4] 삽입 시간 비교

실험 4에서는 삽입 시간을 비교, 분석한다. 실험에서는 세그먼트 개수가 400만개(4M)인 데이터를 사용하여 구성된 인덱스에 새로이 10,000개의 세그먼트를 삽입하는데 걸리는 평균 시간을 측정한다. 실험 결과에 의하면, SETI가 Ours에 비하여 전체적으로는 좋은 삽입 성능을 보이는 것으로 나타났다. 그 이유는 Ours는 하나의 시간 구간내에 저장되는 세그먼트의 개수인 k=1,000이 될 때마다 메모리에 구성된 인덱스를 디스크에 기록해야 하는데, 이 과정에서 인덱스를 매번 공간 차원으로 분할하여 재구성하기 때문에 발생하는 오버헤드가 전체 평균 삽입 시간에 반영되어 성능이 저하되기 때문이다. 반면 SETI는 공간 차원에 대한 분할을 인덱스를 구성하기 전에 미리 결정하여 해쉬 테이블로 관리하고, 이를 이용하여 세그먼트가 속하는 공간 셀 영역의 인덱스를 액세스하여 삽입하는 방식으로 공간 분할에 대한 오버헤드가 발생하지 않는다. 그러나 (그림 8)과 같이 k=1,000이 되는 시점을 제외하면 Ours는 SETI에 비하여 약 60%~590%까지의 좋은 삽입 성능을 보였다.

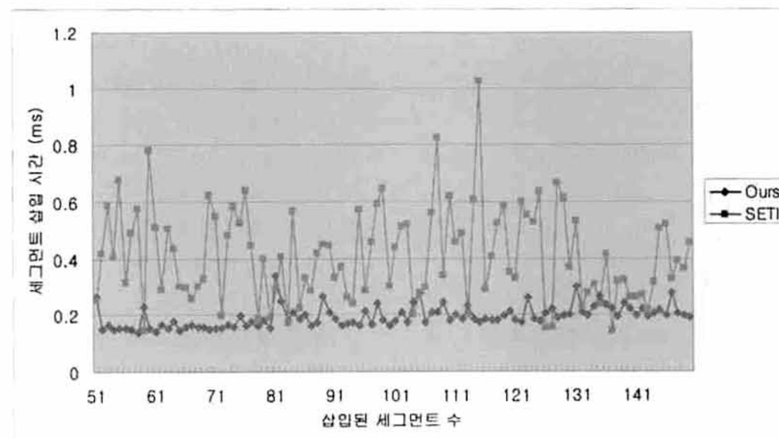
5. 결 론

본 논문에서는 대용량 이동 객체의 궤적 데이터베이스를 위한 효율적인 인덱스 구조를 제안하고, 이를 기반으로 하는 영역 질의 알고리즘을 제안하였다.

제안된 기법에서는 이동 객체의 궤적에 속하는 라인 세그먼트들이 시간 흐름에 따라 순서적으로 삽입되며, 저장이



(그림 7) 최근 질의의 비율 변화에 따른 질의 처리 시간 비교



(그림 8) 세그먼트 삽입 시간 비교(디스크 저장 시간 제외)

오래된 과거 궤적보다는 최근 궤적이 질의의 검색 대상이 될 가능성이 높다는 특징을 이용하여 삽입과 검색 효율성을 극대화시킬 수 있도록 다음과 같은 전략을 사용한다. 먼저, 라인 세그먼트들을 시간 구간별로 구분함으로써 현재와 과거 라인 세그먼트들을 분리한다. 높은 액세스가 빈번하게 발생하는 최근 라인 세그먼트들에 대하여는 메인 메모리상에 인덱스를 관리하며, 대부분의 과거 라인 세그먼트들에 대해서는 디스크 상에 인덱스들을 관리한다. 각 시간 구간에 속하는 라인 세그먼트들은 다음과 같은 방식으로 인덱스를 구축한다. 먼저, 2D-트리를 이용하여 전체 공간 차원을 유사한 수의 라인 세그먼트들이 배정되도록 다수의 셀들로 분할한다. 또한, 분할된 각 셀마다 (x, y, t)에 대하여 작은 크기를 갖는 3차원 R^*-트리를 두어 보다 상세한 인덱싱을 지원한다.

다양한 분포와 크기를 갖는 궤적 데이터 집합을 대상으로 실험을 통한 성능 평가를 수행하였다. 실험 결과에 의하면, 본 논문에서 제안한 다단계 다중 인덱스 기법이 기존의 SETI를 이용한 기법에 비해 작은 크기의 인덱스 구조를 사용하면서도, 검색 연산면에서 3~10배까지의 성능 향상이 있음을 확인하였다.

참 고 문 헌

[1] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R^*-tree: An Efficient and Robust Access Method for Points and Rectangles," In Proc. Int'l. Conf. on Management of Data, pp.322-331, 1990.

[2] J.L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," Comm. of the ACM, 18(9), 1975.

[3] Berkeley University, The GiST Indexing Project, http://gist.cs.berkely.edu, 2007.

[4] V. P. Chakka, A. C. Everspaugh, J. M. Patel, "Indexing Large Trajectory Data Sets With SETI," In Proc. Int'l. Conf. on Biennial Conference on Innovative Data Systems Research, CIDR, 2003.

[5] R. H. Guting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Schneider, and M. VazirGiannis, "A Foundation for Representing and Quering Moving Objects," ACM Transactions on Database Systems, Vol.25, No.1, pp.1-42, 2000

[6] J. M. Hellersten, J. F. Naughton, and A. Pfeffer, Guttman, "Generalized Search Trees for Database Systems," In Proc. Int'l Conf. on Very Large Data Bases, VLDB, 1995.

[7] G. Kollios, D. Gunopulos, and V. J. Tsotras, "On Indexing Mobile Objects," In Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, pp.261-272, 1999.

[8] Z. -H. Liu, X. -L. Liu, J. -W. Ge and H. -Y. Bae, "Indexing Large Moving Objects from Past to Future with PCFI+-Index," In Proc. Int'l Conf. on Mangement of Data, COMAD, pp.131-137, 2005.

[9] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Object Trajectories," In Proc. Int'l Conf. on Very Large Data Bases, VLDB, pp.395-406, 2000.

[10] E. Pitoura, and G. Samaras, "Locating Objects in Mobile Computing," IEEE Transactions on Knowledge and Data Engineering, Vol.13, No.4, pp.571-592, 2001.

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

[12] Z.Song and N.Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects, In Proc. Int'l Conf. on Mobile Data Management, MDM, pp.340-344, 2003.

[13] Y. Theodoridis, M. Vazirgiannis, and T. K. Sellis, "Spatio-Temporal Indexing for Large Multimedia Applications," In Proc. Int'l Conf. on Multimedia Computing and Systems, pp.441-448, 1996.

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

[15] Y. Tao and D. Papadias, "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries," In Proc. Int'l Conf. on Very Large Data Bases, VLDB, pp.431-440, 2001.

[16] Y. Theodoridis, J. R. O. Silva and M. A. Nascimento, "On the Generation of Spatiotemporal Datasets," In Proc. Int'l Symp. on Large Spatial Databases, SSD, pp.147-164, 1999.

[17] Y. Tao, D. Papadias, and J. Sun, "The tpr*-tree: An optimized spatio-temporal access method for predictive queries," In Proc. Int'l Conf. on Very Large Data Bases, VLDB, pp.790-801, 2003.

차 창 일

e-mail : charose@pointi.com

2005년 2월 성결대학교 컴퓨터공학과(학사)

2007년 2월 한양대학교 전자컴퓨터통신학과(석사)

2007년~현 재 포인트아이(주)

관심분야 : 데이터베이스 시스템, 저장 시스템, 공간 데이터베이스/GIS, 주기억장치 데이터베이스, 이동 객체 데이터베이스/텔레매틱스

김 상 욱

e-mail : wook@hanyang.ac.kr

1989년 2월 서울대학교 컴퓨터공학과(학사)
1991년 2월 한국과학기술원 전산학과(석사)
1994년 2월 한국과학기술원 전산학과(박사)
1991년 7월~8월 미국 Stanford University, Computer Science Department 방문 연구원
1994년 2월~1995년 2월 KAIST 정보전자연구소 전문연구원
1999년 8월~2000년 8월 미국 IBM T.J. Watson Research Center Post-Doc.
1995년 3월~2000년 8월 강원대학교 컴퓨터정보통신공학부 부교수
2003년 3월~현 재 한양대학교 정보통신학부 교수
2009년 1월~현 재 미국 Carnegie Mellon University, Visiting Scholar
관심분야 : 데이터베이스 시스템, 저장 시스템, 트랜잭션 관리, 데이터 마이닝, 멀티미디어 정보 검색, 공간 데이터베이스/GIS, 주기억장치 데이터베이스, 이동 객체 데이터베이스/텔레매틱스, 사회 연결망 분석, 웹 데이터 분석

원 정 임

e-mail : jiwon@hanyang.ac.kr

1992년 2월 한림대학교 전자계산학과(학사)
1997년 8월 한림대학교 전자계산학과(석사)
2004년 2월 한림대학교 전자계산학과(박사)
2000년 3월~2004년 2월 한림대학교 교양교육부 강의전담교수
2004년 3월~2006년 2월 연세대학교 컴퓨터과학과 연구교수
2006년 3월~6월 서울대학교 유전자이식연구소 선임연구원
2006년 7월~현 재 한양대학교 정보통신학부 연구교수
관심분야 : 데이터베이스 시스템, 데이터 마이닝, XML 응용, 바이오 정보공학, 데이터베이스 보안, 이동객체 데이터베이스/ 텔레매틱스