이동 객체의 효율적인 처리를 위한 갱신 관리 기법
(An Update Management Technique for Efficient Processing of Moving Objects)

최 용 진 † 민 준 기 † 정 진 완 ††
(Yong-Jin Choi) (Jun-Ki Min) (Chin-Wan Chung)

요 약 대부분의 시공간 데이타베이스 연구들은 색인 분야에서 진행되었다. 그러나, 색인 연구들은 시공간 색인 유지를 위해서 필요한 엄청난 오버헤드의 고려 없이, 색인 생성 후의 빠른 질의 처리에 초점이 맞추어져 있다. 이 논문에서, 우리는 이동체들의 갱신들을 시공간 색인에 반영하기 위해서 필요한 디스크 액세스 수를 줄이는 효율적인 갱신 관리 방법을 제안한다. 객체의 움직임을 잘 표현할 수 있는 현실적인 갱신 패턴을 고려하여 자주 갱신되는 소수의 객체들을 유지할 수 있는 메모리 구조를 제안한다. 실질적인 갱신 패턴을 고려한 실험 환경에서, 우리의 방법은 기존 색인들의 일반적인 갱신 방법보다 약 40%의 디스크 액세스 수를 줄인다.

키워드 : 시공간 데이타베이스, 이동체, 갱신 관리, 색인

Abstract Spatio-temporal databases have been mostly studied in the area of access methods. However, without considering an extraordinary update maintenance overhead after building up a spatio-temporal index, most indexing techniques have focused on fast query processing only. In this paper, we propose an efficient update management method that reduces the number of disk accesses required in order to apply the updates of moving objects to a spatio-temporal index. We consider realistic update patterns that can represent the movements of objects properly. We present a memory based structure that can efficiently maintain a small number of very frequently updating objects. For an experimental environment with realistic update patterns, the number of disk accesses of our method is about 40% lower than that of a general update method of existing spatio-temporal indexes.

Key words : spatio-temporal database, moving object, update management, index

1. 서 론

무선 통신이나 GPS와 같은 기술들의 발전으로 인해, 최근에 시공간 데이타베이스가 집중적으로 연구되었다. 대부분의 연구는 모델링[1,2]과 색인[3-8] 분야에서 진행되었다. 시공간 데이타베이스 분야에서 선택 질의는 두 가지 형태로 구별된다. 하나는 이동체(움직이는 객체)의 과거 위치 정보를 바탕으로 한 질의이고, 또 하나는 이동체의 미래 위치를 위한 질의이다. 이 논문은 후자에 연관되며, 미래 질의로서 언급된다[2]. 미래 질의의 예는 다음과 같다. 지금부터 10분 내에 어떤 공간 영역 안에 존재할 수 있는 비행기를 검색하라. 비행기들은 시간에 따라 이동체들로써 표현된다. 최근 갱신 정보를 가지고 객체의 미래 위치를 관리할 수 있는 모델이 제안되었다[2]. 이동체의 최근 갱신 정보는 공간 위치, 속도 벡터, 그리고 최근 갱신 시간으로 구성된다. 이러한 모델링에 의해서 이동체의 미래 위치를 시간 함수로 표현하여 객체의 갱신 수를 감소시킨다. 최근에, 시공간 데이타베이스 분야에서의 많은 연구들이 이 모델링을 바탕으로 연구되었다[3,4,6,8-10]. 본 논문도 이 모델링을 바탕으로 한, 시공간 색인과 관련된다.

시간의 흐름에 따라, 이동체들은 자주 갱신되는 특징이 있으며, 이러한 갱신들은 색인에 계속해서 반영되어야 한다. 하나의 갱신을 시공간 색인에 반영하기 위해서, 일반적으로 디스크의 여러 페이지 액세스가 요구된다. 따라서, 빈번한 갱신들을 실시간으로 시공간 색인에 반영하기 위해서 상당한 갱신 오버헤드가 존재한다. 그

· 본 연구는 정보통신부의 대학 IT연구센터(ITRC)지원을 받아 수행되었습니다.
† 비 회 원 : 한국과학기술원 전산학과
omni@islab.kaist.ac.kr
jkmin@islab.kaist.ac.kr
†† 종신회원 : 한국과학기술원 전산학과 교수
chungcw@islab.kaist.ac.kr
논문접수 : 2003년 5월 7일
심사완료 : 2003년 8월 18일

러나, 이러한 갱신 오버헤드에도 불구하고, 대부분의 시공간 색인 연구들은 시공간 질의를 빠르게 처리하는 것에 초점이 맞추어져 진행되었다. 이 논문에서 우리는 이동체의 갱신들을 빠르게 처리하기 위한 하나의 효율적인 갱신 관리 기법을 제안한다. 공간 데이타베이스에서의 공간 객체와 비교했을 때, 시공간 데이타베이스에서 객체의 가장 중요한 특징은 공간 상에서 움직인다는 것이다. 따라서, 객체의 움직임을 중요하게 다루어야만 한다. 실세계에서, 이동체는 안정된 변화와 급격한 변화의 혼합된 상태를 보인다[11]. 급격한 변화가 일정 부분 존재함에도 불구하고, 단지 안정된 변화만을 고려하여 객체의 움직임을 표현한다면, 시공간 질의 결과는 쓸모없을 수 있다. 그러므로, 질의 결과가 유용하도록 객체의 급격한 변화 또한 중요하게 다루어야 한다. 본 논문은 객체의 급격한 변화를 적절하게 묘사할 수 있는 많은 갱신들로 이루어진 현실적인 갱신 패턴을 고려한다. 이 논문에서 제안한 기본적인 개념은 빈번하게 갱신되는 소수의 객체들만을 위한 메모리 기반 구조를 유지하고, 상대적으로 덜 갱신되는 대부분의 객체들을 디스크 기반 구조에 유지하는 것이다. 제안한 메모리 기반 구조는 메모리 기반 색인과 그 색인에 유지된 객체들을 관리하는 해쉬 테이블로 구성되어 있다.

다양한 실험 결과에 의하면, 제안된 방법이 시공간 색인을 유지하기 위해 필요한 디스크 액세스 수를 줄인다. 좀더 현실적인 실험환경을 위해서, 두 가지 방법들을 사용한다. 첫째로, 객체의 초기 위치를 생성하기 위해서, 공간 데이타베이스 분야에서 널리 사용되는 실세계 데이타인 Tiger/Lines 데이타[12]를 사용한다. 둘째로, 객체의 움직임을 잘 표현하기 위해서 현실적인 갱신 패턴을 고려한다. 시공간 색인을 유지하기 위해 필요한 디스크 액세스 수를 비교할 때, 제안된 방법은 기존의 방법에 비해 약 40%의 디스크 액세스 수를 줄인다.

이 논문은 다음과 같이 구성된다. 2장에서, 시공간 색인 기법에 대한 관련 연구를 설명한다. 3장에서, 이동체의 현실적인 갱신 패턴을 설명한다. 4장에서, 이동체의 갱신을 시공간 색인에 효율적으로 반영하는 방법을 설명한다. 5장에서, 실험 결과를 보이고, 세부적으로 그 결과를 논한다. 마지막으로, 6장에서 결론을 내린다.

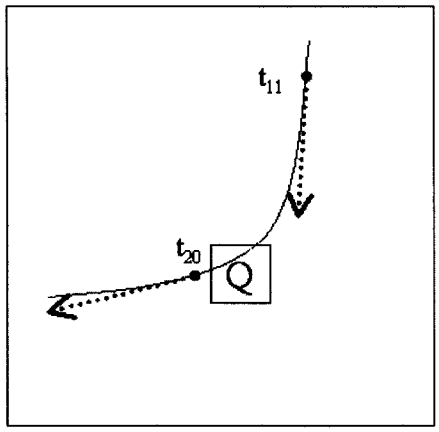
2. 관련 연구

객체의 미래 위치를 표현하는 데이타 모델이 제안되었다[2]. 이러한 모델은 이동체의 갱신 수를 감소시킨다. 시공간 색인에서의 대부분의 연구들[3,4,6,8,9,13,14]은 이 모델링을 기반으로 하고 있다. 그러나, 그 연구들은 색인 생성 후, 빠르게 질의를 처리하는 것에 초점이 맞추어져 있다. 일반적으로 객체가 갱신될 때, 기존의 시공간 색인 기법들은 간단하게 하나의 삭제연산과 하나의 삽입연산으로 이루어진 갱신연산을 사용한다. 이러한 일반적인 갱신 방법을 향상시키기 위해서, 만약 두 연산들이 동일한 단말노드에서 이루어지면, 두 연산 대신 하나의 수정(modify) 연산으로 대신 처리할 수 있다[15,16]. 그러나, 그 갱신 방법은 모든 객체들을 위한 디스크 기반의 색인(DirectLink라 불리어진다)을 사용하기 때문에, 상당한 공간 오버헤드를 추가적으로 요구한다[16]. 이러한 단순하고 일반적인 방법들은 하나의 갱신을 시공간 색인에 반영하기 위해서, 디스크의 여러 페이지 액세스를 요구한다. 따라서, 시간이 흐를 때, 빈번한 갱신들을 시공간 색인에 반영하기 위해서 상당한 갱신 비용이 요구된다. 주어진 모든 객체들에 대한 초기의 새로운 색인을 빠르게 생성하기 위한 방법으로 Bulk loading이 제안되었다[17]. Bulk insertion은 새롭게 추가된 객체들을 위해서 이미 존재하는 색인에 해당되는 엔트리들을 빠르게 추가하는 것이다. 그러나, 이 논문에서 다루는 문제는 일부 객체들의 갱신들을 이미 존재하는 색인에 빠르게 반영하는 것이다.

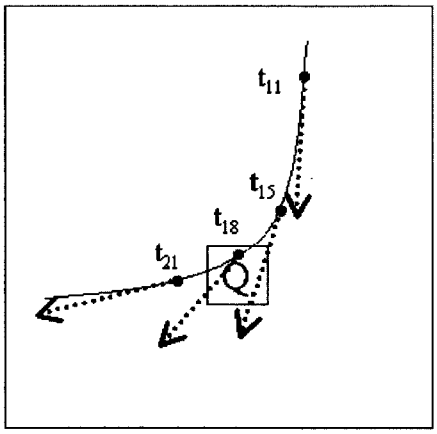
3. 현실적인 갱신 패턴

먼저, 이 장에서는 이동체의 갱신을 설명한다. 그러고 나서, 객체의 현실적인 갱신 패턴을 설명한다.

시공간 데이타베이스에서, 가장 중요한 특징 중의 하나가 객체가 공간에서 움직인다는 것이다. 이동체는 객체의 실제 위치와 그것의 데이타베이스 위치 사이의 차가 주어진 거리 임계치를 넘을 때마다 갱신된다[6,11]. 객체의 움직임은 연속적인 갱신들로 표현된다[2]. 그림 1은 방향을 바꾸는 객체의 갱신을 나타낸 것이다. 곡선은 객체의 실질적인 움직임을 나타낸다. 점은 갱신이 반영된 시간에서의 공간 위치를 의미하며, 점선의 화살표는 갱신 시간에 고려된 속도 벡터를 나타낸다. 그림 1(a)는 큰 거리 임계치의 초과에 의해 갱신된 객체의 움직임을 보인다. 유사하게, 그림 1(b)는 작은 거리 임계치의 초과에 의해 갱신된 객체의 움직임을 보인다. 객체가 가속, 감속, 방향 전환할 때 (앞으로 **활동적인**이란 용어를 사용한다), 객체의 움직임은 큰 거리 임계치를 적용하는 갱신 방법에 의해서 적절하게 표현되지 못할지도 모른다. 그림 1(a)에서 알 수 있듯이, 그림 1(b)와 비교했을 때 객체의 움직임의 갑작스러운 변화가 두 개의 연속적인 갱신 시간(t_{11}과 t_{20}) 사이에서 발생한다. 시간 16에서의 시공간 질의를 고려해보자. 이 시공간 질의는 "이동체가 현재 상태(시간 16) 이후에 그림과 같은 영역(Q)을 지날 수 있는가?"이다. 그림 1(a)의 경우에는 객체가 주어진 영역을 지나지 못하지만, 그림 1(b)의 경우에는 객체가 주어진 영역을 지날 수 있다. 거리 임계



(a) 큰 거리 임계치에 의한 갱신



(b) 작은 거리 임계치에 의한 갱신

그림 1 객체의 갱신

치의 값이 작을수록, 시공간 질의 결과는 더욱 유용해진다[10].

그림 2는 고속도로나 거리에서의 속도 변화를 나타내는 객체의 현실적인 움직임을 보인다[11]. 그림 2에서 알 수 있듯이, 객체의 실질적인 움직임은 단조로운 변화와 급격한 변화의 혼합된 상태를 보인다. 급격한 변화를 보이는 움직임이 부분적으로 존재하기 때문에, 만약 급

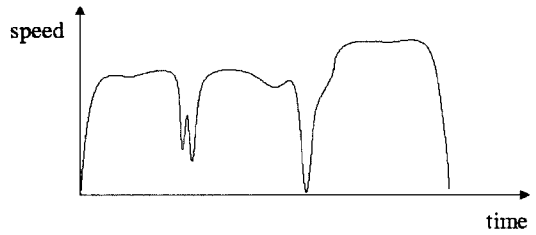


그림 2 객체의 현실적인 움직임

격한 변화를 고려하지 않고 단조로운 변화만을 고려하여 객체의 움직임이 표현된다면, 시공간 질의 결과는 쓸모없을 수 있다. 그러므로, 질의 결과를 유용하게 하려면 객체의 급격한(또는 활동적인) 변화를 중요하게 다루어야 한다. 따라서, 우리는 객체의 급격한 변화를 적절하게 묘사할 수 있는 많은 갱신들로 이루어진 현실적인 갱신 패턴을 고려해야 한다. 이를 위해서, 작은 거리 임계치를 사용할 수 있다. 또한, 객체의 갱신을 처리하기 위해서 속도 임계치를 추가적으로 고려할 수 있다. 속도 임계치에 의한 갱신 방법은 객체의 급격한 변화를 적절하게 표현할 수 있다는 장점이 있다. 거리 임계치와 속도 임계치를 동시에 고려함으로써 객체의 움직임을 잘 표현할 수 있다.

활동적인 움직임을 묘사하기 위해서 다음과 같은 정의를 소개한다. T를 하나의 객체에 갱신 시간에 대한 하나의 리스트 $<t_1, t_2, \cdots, t_n>$라 하고, L_a를 하나의 양수라고 하자. $L_a \geq t_{j+1}-t_j$ for $i \leq j \leq k-1$, $L_a < t_i - t_{i-1}$, 그리고 $L_a < t_{k+1}-t_k$를 만족하는 T의 한 서브 리스트 $<t_i, t_{i+1}, \cdots, t_k>$를 하나의 사건(event) e_a로 정의한다. 그리고, $L_a < t_i - t_{i-1}$와 $L_a < t_{i+1}-t_i$를 만족하는 T의 한 서브 리스트 $<t_i>$를 하나의 사건 $\bar{e}_a$로 정의한다. 그림 3은 이러한 정의들의 예를 설명한다. 점은 한 객체의 임계치 초과 때마다 생성된 갱신 시간들을 의미한다. L_a가 3일 때, 그림 3은 2개의 e_a와 3개의 $\bar{e}_a$를 보인다.

객체의 움직임이 활발한지를 판단하는 값 L_a가 필요하다. 시스템 관리자는 그림 2처럼 실제 상황에서의 이동체의 현실적인 움직임을 파악한 후, 시공간 질의 결과의 유용성을 고려하여 초기에 L_a값을 설정할 수 있다.

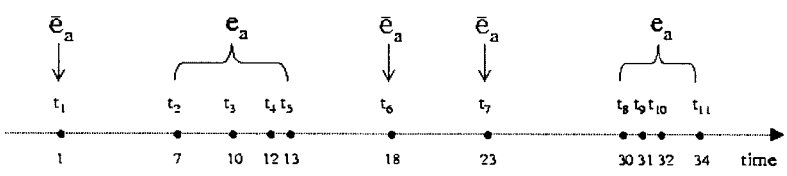


그림 3 사건

객체의 갱신이 L_a와 같거나 작은 시간 내에 이루어지면, 객체가 많은 갱신들을 유발시키는 활동적인 움직임 상태에 있다고 판단할 수 있다. 5장의 실험 평가를 위해서, 객체의 활동적인 움직임을 적절하고 현실적으로 묘사할 수 있도록 사건 $e_a<t_i, t_{i+1}, \cdots, t_k>$를 생성한다. 즉, $i \le j \le k-2$를 위한 $t_{j+1}-t_j$와 $t_{j+2}-t_{j+1}$의 차가 작도록 각 사건들을 생성한다. 이 논문의 기본적인 개념은 e_a 상태에 놓여 있는 객체들을 제안된 메모리 기반 구조 안에 유지하는 것이다.

4. 메모리 기반 구조의 효율적인 갱신 관리 기법

이 장에서는 활동적인 움직임을 갖는 소수의 객체들을 효율적으로 유지할 수 있는 메모리 기반 구조를 설명한다. 그리고 나서, 객체의 갱신을 효율적으로 관리할 수 있는 알고리즘을 상세히 설명한다.

4.1 메모리 기반 구조

이 논문에서 제안하는 전체 구조는 하나의 디스크 기반의 색인(I_d), 하나의 메모리 기반의 색인(I_m), 그리고 I_m에서 유지되는 객체들을 관리하는 해쉬 테이블(H)로 구성된다. 이동체들 중에서, 대부분의 객체들은 디스크 기반의 색인에서, 일부 소수의 객체들은 메모리 기반의 색인에서 유지된다. 이동체들은 두 색인 중에서 하나의 색인에서만 유지되기 때문에, 시공간 질의의 결과는 두 색인에서의 질의 결과들의 단순한 합이며, 중복되지 않는다. 이 장에서는 H와 I_m로 이루어진 메모리 기반 구조에 대해서 자세히 설명한다.

제안된 메모리 기반 구조(H와 I_m)에 존재할 수 있는 최대 객체 수(N_m)가 4일 때, 그림 4는 4개의 객체들이 유지된 상태를 나타낸다. H는 기본적으로 체인 구조이며, I_m에 유지된 객체들을 관리한다. H안에 객체의 식별자가 삽입, 삭제, 액세스될 때, least recently used (LRU) 알고리즘이 사용되어 H가 관리된다. 그림 4처럼 H의 한 슬롯(slot)은 객체 식별자 id, 그 객체의 최근 갱신 시간 t^u, 그 객체를 포함하는 I_m의 단말 노드를 지시하는 포인터 p, 그리고, 각 slot을 연결하는 3개의 포인터들로 구성된다. 객체 id가 H에 삽입될 때, 하나의 슬롯이 생성된다. 간단한 설명을 위해서 해쉬 함수로 mod 연산을 고려해 보자. 객체의 id는 어떤 숫자 b에 의해서 나누어지고, 그 나머지가 id를 위한 해쉬 주소로 사용된다. 즉, 해쉬 함수는 h(id) = id%b이다. 그림 4처럼, 5개의 버켓(bucket)들을 가진 H를 고려하자. 그리고, 객체 9, 4, 7, 11들이 각각 시간 t_1, t_2, t_3, t_4에 갱신되었고 I_m에 이러한 4개의 객체가 유지되었다고 하자. 객체 9는 해쉬 주소 4(=9%5)를 갖는다. 따라서, H의 버켓 4에 객체 9를 위한 하나의 슬롯이 생성된다. 객체 9의 식별자와 갱신 시간이 그 슬롯에 저장된다. 가장 먼저 갱신된 객체 9의 슬롯은 H의 front 슬롯이 된다. 그리고, 각 객체의 식별자가 I_m에 삽입될 때, 각 슬롯에서 해당 객체를 포함하는 I_m의 단말 노드를 지시하도록 한다. 각 슬롯의 I_m의 단말 노드를 지시하는 포인터는 I_m에서부터 빠르게 객체의 정보를 제거하기 위해서 사용된다.

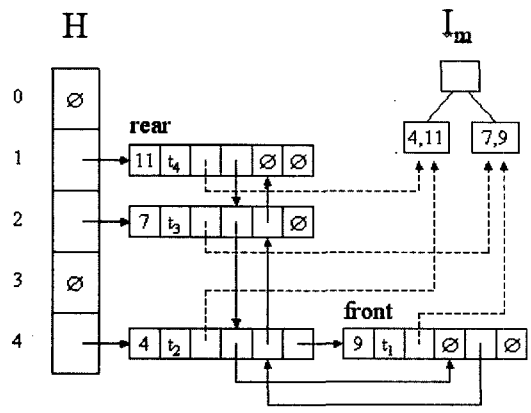


그림 4 메모리 기반 구조

이제, I_d에서 유지되었던 객체 α가 갱신되어 이미 N_m개의 객체를 유지하는 메모리 기반 구조에 삽입될 조건을 만족한다고 가정할 때, 그 삽입 과정을 자세히 살펴보자. 먼저, 그 갱신된 α를 메모리 기반 구조에 삽입하려면, H의 front 슬롯에 놓여진 객체 9를 메모리 기반 구조에서 제거해야 한다. 메모리 기반 구조에서 한 객체의 제거는 두 단계로 이루어진다. 첫 번째, H의 front 슬롯에서 I_m의 단말 노드를 지시하는 포인터를 따라, 그 슬롯에 속한 객체 엔트리를 I_m으로부터 제거하여 I_d에 삽입한다. 두 번째, H의 현재 front 슬롯을 제거한다. 이 때, H의 front 슬롯은 객체 5의 슬롯이 된다. α의 새롭게 갱신된 정보를 처리하기 위해서 α의 이전 엔트리를 I_d에서 제거한 후, 갱신된 정보를 메모리 기반 구조에 삽입한다. 먼저 α의 식별자가 위치할 새로운 슬롯이 생성되며, 그 슬롯 안에는 α의 식별자와 새로운 갱신 시간이 저장된다. 그 생성된 슬롯은 H의 rear 슬롯이 된다. 그리고, I_m에 그 갱신된 정보에 관한 객체 엔트리를 삽입한다. 마지막으로, rear 슬롯의 I_m을 지시하는 포인터가 α의 엔트리를 포함하는 단말 노드를 지시하도록 한다.

4.2 효율적인 갱신 관리 알고리즘

이 장에서는 이동체의 갱신들을 메모리 기반 구조에서 효율적으로 유지할 수 있는 갱신 관리 알고리즘을 설명한다. 먼저, 알고리즘의 이해를 돕기 위해 간단한 예를 들어 설명한다. 그리고 나서, 제안한 알고리즘을 상세히 설명한다.

이 논문에서 제안된 메모리 기반 구조(H와 I_m)는 기존의 색인 기법들의 일반적인 갱신 방법에 비해 추가의 메모리를 요구한다. S_m을 추가의 메모리 크기라 하자. 따라서, 공정한 성능 비교를 위해서, 기존의 색인 유지 기법들의 일반적인 갱신 방법(GUM)을 위한 버퍼 크기로 B가 사용된다면, 우리가 제안한 실질적인 갱신 패턴을 고려한 메모리 기반 갱신 방법(MUMA)에서 버퍼 크기로 B-S_m이 사용된다. 버퍼가 디스크 기반 색인의 노드들을 유지하는 반면에, 제안된 메모리 기반 구조는 빈번하게 갱신되는 소수의 일부 객체들을 유지한다. 그리고, 버퍼의 페이지 교환을 위해 LRU 기법을 사용한다. 그림 5(c)에서 보여지듯이, rear와 front는 각각 가장 최근에 액세스되었던 페이지와 가장 오래 전에 액세스되었던 페이지를 나타낸다. 페이지가 교체될 때, LRU 알고리즘은 가장 오래 동안 사용되지 않은 페이지를 선택한다. 또한, 제안된 메모리 기반 구조에서 유지된 객체를 관리하기 위해서 LRU 알고리즘이 사용된다. 버퍼 크기(B)는 3페이지로, 제안된 메모리 기반 구조 크기(S_m)는 1페이지로 설정된다고 하자. 그리고, 그림 5(d)에서 보여지듯이, 1페이지 크기의 메모리 기반 구조에 4개의 객체가 유지된다고 하자.

제안된 알고리즘의 직관적인 이해를 돕기 위해서, 먼저 한 가지 가정을 고려하는 간단한 예를 살펴보자. 디스크에서 이루어지는 한 객체에 대한 액세스는 하나의 읽기 액세스(하나의 페이지 I/O)라고 가정한다. 그러나, 제안된 알고리즘은 이러한 가정에 상관없이 일반적인 경우에 적용된다. 이동체들의 갱신에 대한 처리는 아래 예제에서와 비슷하게 처리된다.

그림 5(a)는 9개의 디스크 페이지들(P_1~P_9)을 보이며, 각 페이지는 8개의 객체들을 유지한다. 그리고, 그림 5(b)는 시간에 흐름에 따른 객체의 액세스를 나타낸다. 모든 i를 위한 t_{i+1}-t_i의 값은 같다. 그림 5(c)는 객체들의 액세스에 대한 GUM 방식에서의 LRU 교체 기법을 적용한 결과를 나타낸다. 초기의 3 페이지는 비어있는 상태이다. 세 객체들(9,51,48)에 대한 액세스는 페이지 결함을 일으키며, 각 해당 페이지 P_2, P_7, P_6는 비어있는 3 페이지에 할당된다. *는 객체의 액세스에 의한 페이지 결함을 나타낸다. 그 다음 객체 2에 대한 액세스는 가장 오래 전에 메모리에 유지되었던 페이지 P_2를 교체시킨다. 다음에 액세스 되는 객체 51은 이미 메모리에 있기 때문에, 페이지 결함이 발생하지 않는다. 이러한 과정은 그림 5(c)에서 보여준 것처럼 계속된다. 모두 11개의 페이지 결함이 발생된다.

그림 5(d)는 우리의 제안된 방법 MUMA를 설명한다. 하나의 객체가 액세스 될 때, 그 객체가 먼저 메모리 기반 구조에 있는지 확인한다. 만약 메모리 기반 구조에 없으면, 그 객체가 버퍼에 있는지 확인한다. 만약 그 객체가 버퍼에도 없으면, 마지막으로 디스크를 액세스한

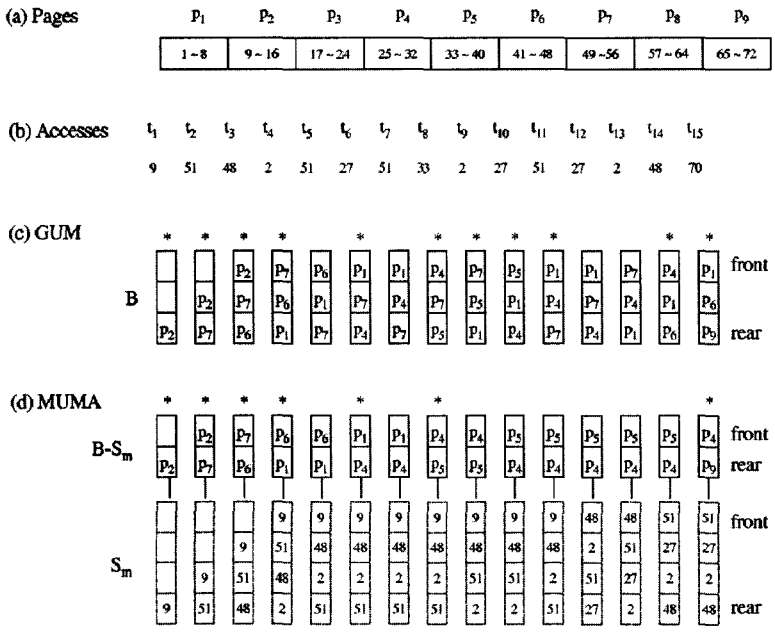


그림 5 효율적인 갱신 관리 알고리즘 예제

다. 우리는 객체의 액세스가 빈번하게 이루어지는지를 판단하기 위해서 L_a를 사용한다. 즉, 한 객체의 액세스가 이전 액세스 시간으로부터 L_a 이내에 액세스 되면, 그 객체가 빈번하게 액세스 된다고 가정한다. 이 예제를 위해서, L_a를 3으로 설정한다. 처음에 객체 9가 액세스 되면, 그 객체를 포함하는 페이지 P_2는 비어있는 페이지에 할당되고 객체 9는 제안된 메모리 기반 구조에도 할당된다. 다음 3개의 객체(51,48,2)에 대한 액세스는 모두 페이지 결함을 유발시키고, 각 객체는 메모리 기반 구조에 할당된다. 그 다음에 액세스 되는 객체 51는 이미 메모리 기반 구조에 있기 때문에, 메모리 기반 구조에서 객체들의 액세스 순서만을 변경시킨다. 그 다음 액세스 되는 객체 27은 메모리에 존재하지 않기 때문에, 페이지 결함을 유발시켜, 페이지 P_4가 페이지 P_2를 대신해서 메모리에 할당된다. 그러나, 객체 27의 바로 이전 액세스와의 시간 차가 L_a보다 크기 때문에, 이 객체는 매우 빈번하게 액세스 되는 객체가 아니라고 판단하여, 메모리 기반 구조에서 유지되지 않는다. 시간 12에 객체 27은 다시 액세스 된다. 이 객체의 바로 이전 액세스와의 시간 차는 2이기 때문에, 객체 27은 매우 빈번하게 갱신될 것이라고 판단할 수 있다. 객체 27은 가장 오래 전에 액세스 된 객체 9를 대신하여 메모리 기반 구조에서 유지된다. 이러한 과정은 그림 5(d)에서 보여준 것처럼 계속된다. 모두 7개의 페이지 결함이 발생된다.

이제, 메모리 기반 갱신 관리 알고리즘을 자세히 살펴보자. 먼저, 알고리즘을 설명하기 위해서 필요한 몇 가지 연산들과 기호들을 설명한다. o를 이동체라고 하자. 객체의 식별자 id, 그 객체의 최근 갱신 시간 t^u, 그 객체를 포함하는 I_m의 단말노드를 지시하는 포인터 p로 이루어진 하나의 컴포넌트를 c라고 하자. C를 c의 집합이라고 하자. H의 연산들은 다음과 같다. insertH(id,t^u)은 id와 t^u가 저장된 새로운 슬롯을 생성하고, 그 슬롯은 H의 rear가 된다. adjustH(id)은 id를 포함하는 슬롯이 rear 슬롯이 되도록 조정한다. setpH(c)은 s를 c.id의 슬롯이라고 할 때, c.p를 s.p에 저장한다. InsertIm(o), deleteIm(o), updateIm(o)를 객체 o에 대한 색인의 일반적인 연산들처럼 각각 I_m의 삽입, 삭제, 갱신 연산들이라고 하자. 그러나, InsertIm(o), deleteIm(o), updateIm(o)는 해당 연산에 영향받는 엔트리들을 위한 C를 돌려준다. 예를 들어, 객체 o의 삽입 연산 전에 I_m의 한 단말노드에 속한 다른 객체의 엔트리 k를 고려하자. 객체 o의 삽입 연산 후에 노드분할(split)로 인해서, k는 다른 단말노드에 포함될 수 있다. 이와 같이 객체 o뿐만 아니라, o에 대한 연산 수행 후, 영향 받은 모든 객체 엔트리들의 id와 단말 노드를 지시하는 포인터 p의 정보가 필요하다. 급격한 움직임을 보이는 객체들을 선택하기 위해서 필요한 L_a값은 객체들의 움직임을 파악하여 초기에 설정할 수 있다. 그리고, θ_o를 객체의 새로운 갱신 시간과 이전 갱신 시간과의 차이라고 하자.

그림 6은 메모리 기반 구조에 객체의 갱신들을 반영하기 위한 메모리 기반 갱신 관리 알고리즘 MUMA를 나타낸다. 첫째로, 갱신된 객체 o가 메모리 기반 구조에 존재할 경우에, 그 처리 방법은 다음과 같다(2줄~3줄). 먼저, H에 o.id를 가진 슬롯을 rear 슬롯으로 조정한다. 그리고, 그 객체의 갱신을 I_m에 반영되도록 한다. 이 때, 이 객체에 의해 단말 노드의 위치가 변경되는 엔트리들이 생기면, H에 해당 슬롯의 p를 모두 변경한다. 둘째로, 갱신된 객체 o가 메모리 기반 구조에 존재하지 않는 경우에, 4줄부터 16줄까지는 그 처리 방법을 나타낸다. H에 존재하는 객체 수가 N_m보다 작은지의 여부에 따라, 그 처리 방법은 달라진다. H에 존재하는 객체 수가 N_m보다 작을 경우에, 그 처리 방법은 다음과 같다(5줄~7줄). 그 객체의 이전 엔트리를 I_d로부터 삭제 후에, H에 o.id와 o.t^u를 삽입한다. 그리고, 갱신 정보를 I_m에 삽입하고, 이 연산에 의해 단말 노드의 위치가 변경되는 엔트리들에 대해서 처리해 준다. H에 존재하는 객체 수가 N_m보다 크거나 같을 경우에, 그 처리 방법은 다음과 같다(8줄~16줄). 객체의 갱신이 메모리 기반 구조에 반영되기 위해서는 두 가지 조건을 모두 만족해야 한다(8줄). 첫번째 조건($\theta_o \le L_a$)에서 객체가 활발하게 움직이는 상태에 있는지 판단한다. 객체의 두 연속적인 갱신 사이의 시간 차(θ_o)가 L_a보다 작으면, 이러한 객체는 활발하게 움직이는 상태에 있다. 그러나, 메모리 기반 구조에서 유지된 객체들이 모두 활발하게 움직이는 상태에 있을 수 있다. 두 번째 조건($\theta_o \le$ (현재 시간 - front 슬롯의 t^u))에서 새롭게 갱신된 객체와 메모리 기

Algorithm MUMA
Input: *o*, an updated object

```
1.  if o.id exists in H,
2.      invoke adjustH(o.id).
3.      invoke setpH(c) for all c ∈ C ← updateIm(o).
    else,
4.      if the number of slots in H is less than Nm,
5.          delete o from Id.
6.          invoke insertH(o.id, o.tu).
7.          invoke setpH(c) for all c ∈ C ← insertIm(o).
        else,
8.          if θo ≤ La and θo ≤ (current time - tu of the front slot of H),
9.              α ← an index entry in Im with id of the front slot of H.
10.             invoke setpH(c) for all c ∈ C ← deleteIm(α).
11.             insert α to Id.
12.             delete the front slot of H.
13.             delete o from Id.
14.             invoke insertH(o.id, o.tu).
15.             invoke setpH(c) for all c ∈ C ← insertIm(o).
            else,
16.             update o from Id.
```

그림 6 효율적인 갱신 관리 알고리즘

반 구조에서 가장 오래 전에 액세스 되었던 객체 사이의 활동성을 비교한다. 두 번째 조건으로 메모리 기반 구조에서 유지된 객체들이 새로이 갱신된 객체보다 매우 빈번하게 갱신될 때, 메모리 기반 구조에 계속해서 유지될 수 있음을 보장받는다. 두 가지 조건을 만족하면, 가장 오래 전에 액세스 되었던 객체를 대신하여 새로이 갱신된 객체가 메모리 기반 구조에서 유지된다(9줄~15줄). 위의 두 가지 조건을 모두 만족하지 못하면, 객체의 갱신이 메모리 기반 구조에 반영되지 않는다. 단순히 그 갱신은 디스크 기반 색인 I_d에 반영된다(16줄).

5. 실 험

기존의 연구들[4,6,11]의 실험 환경과 비슷하게 인위적인 시공간 객체를 생성하여 실험한다. 최대 속도 3.0을 가진 객체들은 10,000 * 10,000의 2차원 공간 상에서 움직이도록 설정된다. 보다 현실적인 실험 환경을 위해서 2가지 사항을 고려한다. 첫번째, 공간 데이타베이스 분야에서 널리 사용된 Tiger/Lines 데이타를 사용하여 이동체들의 초기 공간 위치를 고려한다. 객체의 수는 약 530,000개이다. 둘째로, 이동체의 현실적인 갱신 패턴을 고려한다. 그림 2에서 알 수 있듯이, 객체의 실질적인 움직임은 단조로운 움직임과 급격한 움직임의 혼합된 형태를 따른다. 유용한 질의 결과를 위해서, 우리는 객체의 급격한(또는 활동적인) 움직임을 중요하게 다루어야 한다. 활동적인 움직임을 갖는 객체를 의미하는 e_a를 생성하기 위해서, 갱신 타입과 갱신 수를 고려한다. 객체의 활동적인 움직임을 표현할 수 있는 대표적인 갱신 패턴으로부터 e_a 생성 예를 살펴보자. 그림 7은 활동적인 움직임을 표현하는 대표적인 갱신 패턴을 보인다. 연속적인 두 개의 갱신 시간 사이의 시간 차는 점진적으로 감소하며, 최대 갱신 수는 7이다. 이 대표적인 갱신 패턴으로부터 임의로 선택된 3개의 연속적인 갱신 시간 t_2,t_3,t_4으로부터 생성된 $e_a<t_\alpha,t_\beta,t_\chi>$를 고려해 보자. e_a 상태에 있는 객체의 첫번째 갱신 시간 t_α는 e_a의 생성시간이며, 두 번째 갱신 시간 t_β는 t_α로부터 5(=t_3-t_2) 단위 시간 후이며, 세 번째 갱신 시간 t_χ은 t_β로부터 4(=t_4-t_3) 시간 후이다.

우리는 갱신의 간격이 점진적으로 증가하는 경우, 갱신의 간격이 일정한 경우, 갱신의 간격이 점진적으로 감소하는 경우들과 같은 여러 가지 갱신 패턴들을 고려한다. 각 대표적인 갱신 패턴의 최대 갱신 수(N_a)는 20으로 설정한다. 그림 7와 유사하게, 갱신의 시간 차가 점진적으로 감소하는 대표적인 갱신 패턴은 초기에 갱신 시간 차(t_2-t_1)가 19(=N_a-1)이며, 다른 시간 차들은 1씩 감소한다. 점진적으로 증가하는 대표적인 갱신 패턴은 초기에 갱신 시간 차가 1이며, 다른 시간 차들은 1씩 증가한다. 갱신의 시간 차가 일정한 경우는 4 가지 패턴을 고려한다. 각 갱신의 시간 차가 4, 6, 8, 10인 일정한 갱신 패턴을 고려한다. e_a를 결정하기 위한 L_a를 최대 갱신 시간 차인 19(=N_a-1)로 설정한다. 그리고, 활동적으로 이동체의 갱신 리스트(e_a)를 생성할 때, 갱신의 시간 차가 점진적으로 감소하는 경우는 30%, 점진적으로 증가하는 경우는 30%, 일정한 4가지 경우는 각각 10%씩 선택하였다. 또한, 활동적으로 이동체의 갱신 리스트(e_a)를 위해서 최소 갱신 수(n_{min}) 5와 최대 갱신 수(n_{max}) 20사이에서 임의로 설정하였다.

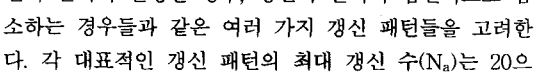
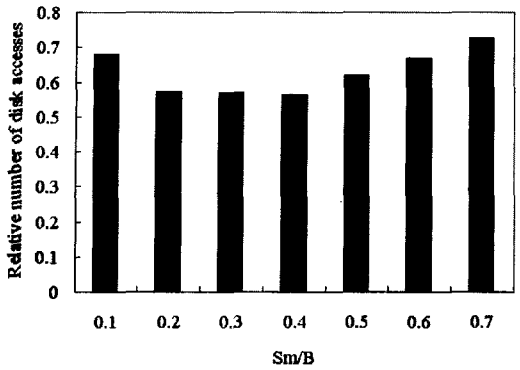


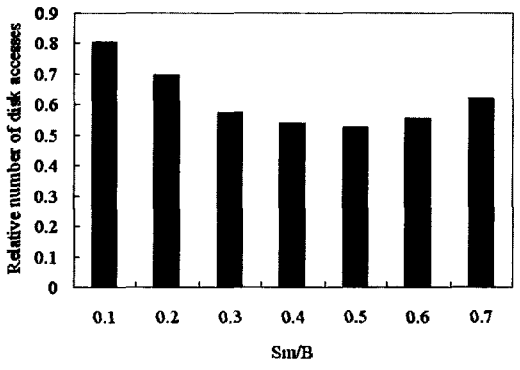
그림 7 대표적인 갱신 패턴

지금까지, 활동적으로 이동체의 갱신 리스트(e_a)를 생성을 고려하였다. 그러나, 전체적인 갱신 스트림(stream)은 e_a와 $\bar{e}_a$의 혼합된 형태이다. 추가적인 파라미터를 사용하여 전체적인 갱신 스트림을 표현한다. E를 e_a의 수와 $\bar{e}_a$의 수의 합이라고 하자. 그리고, E_a를 e_a의 수라고 하자. E/E_a는 전체적인 갱신 스트림을 나타낸다. 페이지 크기(또는 색인의 노드 크기)는 4Kbytes로 설정하였다. 우리는 I_m과 I_d를 위해서 기존에 널리 사용되고 있는 TPR-tree[13]를 사용한다. 버퍼의 크기(B)는 페이지 수가 300이 되도록 설정한다. 제안된 메모리 기반 구조는 추가의 메모리를 요구한다. 따라서, 공정한 성능 비교를 위해서, 기존의 색인 유지 기법들의 일반적인 갱신 방법을 위한 버퍼 크기로 B를 사용한다면, 제안된 방법에서 버퍼 크기로 B-S_m을 사용한다. S_m의 선택은 질의 성능 향상과 갱신 성능 향상 사이의 tradeoff와 관련한다. [4,13,16]처럼, 이동체들의 갱신들을 처리하기 위해서 필요한 I/O 비용은 CPU 비용에 비해 매우 크기 때문에, 실험 평가는 디스크 액세스 수만을 고려한다.

그림 8은 S_m/B에 관한 상대적인 디스크 액세스 수를 보여준다. 상대적 디스크 액세스 수는 기존의 시공간 색인 기법들에서의 일반적인 갱신 방법에 의한 디스크 액세스 수에 대한 제안한 메모리 기반 갱신 관리 기법에 의한 디스크 액세스 수의 비율을 의미한다. E/E_a는 10으로 설정되었다. 전체적인 갱신 스트림을 위해서, E는 약 200,000 e_a와 $\bar{e}_a$으로 설정하였다. 그림 8(a)는 N_a가 20일 때의 대표적인 갱신 패턴으로부터 생성된 e_a를 가진 이동체들에 대한 실험 결과를 보인다. S_m/B가 0.2와 0.4사이일 때, 약 40%의 디스크 액세스 수의 감소를 보인다. S_m/B의 크기가 작은 경우에, 제안된 방법의 효과는 다소 감소한다. 이러한 이유는 적은 크기의 메모리 기반 구조에서 e_a의 상태에 있는 모든 객체들을 유지하

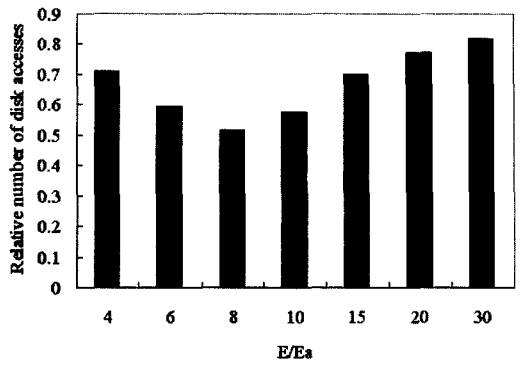


(a) Na = 20

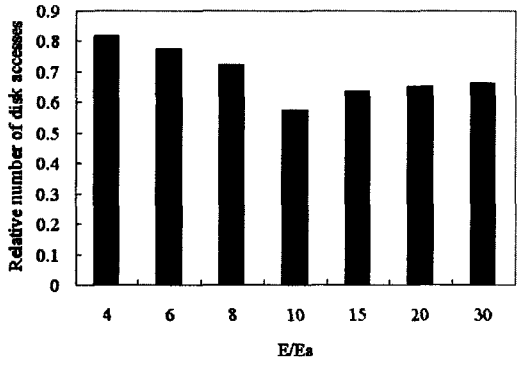


(b) Na = 40

그림 8 S_m/B에 대한 상대적인 디스크 액세스 수



(a) Na = 20



(b) Na = 40

그림 9 E/E_a에 대한 상대적인 디스크 액세스 수

지 못하고, 그러한 객체들이 서로 I_m과 I_d에서 사이에서 교환되기 때문이다. S_m/B의 크기가 클 때, 제안된 방법의 효과는 다소 반감된다. e_a에 해당되는 객체들을 모두 메모리 기반 구조에서 유지하지만, I_d에서 유지되는 객체들의 갱신을 위해서 사용되는 버퍼의 페이지 수가 상대적으로 줄어들기 때문이다. 그림 8(b)는 N_a가 40일 때의 대표적인 갱신 패턴으로부터 생성된 e_a를 가진 이동체들에 대한 유사한 실험 결과를 보인다.

추가적으로, E/E_a에 관한 실험을 평가하였다. S_m/B는 0.3으로 설정되었다. 그림 9(a)는 N_a가 20일 때의 대표적인 갱신 패턴으로부터 생성된 e_a를 가진 이동체들에 대한 실험 결과를 보인다. E/E_a가 6에서 10 사이일 때, 약 40%의 디스크 액세스 수의 감소를 보인다. E/E_a가 작을 때(예:4), 제안된 방법은 다음과 같은 2가지 이유로 디스크 액세스 수의 감소 효과가 줄어든다. 첫 번째 이유는 메모리 기반 구조에서 e_a에 해당되는 모든 객체들을 유지하지 못하기 때문이다. 두 번째 이유는 일반적인 갱신 방법에서 사용되는 버퍼 크기의 70% 버퍼크기를 사용하기 때문이다. 또한, E/E_a가 클 때(예:30), e_a에 해당되는 갱신 수가 너무 미비하기 때문에 제안된 방법의 효과는 다소 감소한다. 기대되었듯이, 그림 9(b)는 N_a가 40일 때의 대표적인 갱신 패턴으로부터 생성된 e_a를 가진 이동체들에 대한 유사한 실험 결과를 보인다.

6. 결 론

최근에, 무선 통신이나 GPS와 같은 기술들의 발전으로 시공간 데이타베이스가 많이 연구되었다. 이 논문에서, 우리는 현실적인 갱신 패턴을 가진 빈번하게 갱신되는 소수의 이동체들을 유지할 수 있는 메모리 기반 구조를 제안하였다. 객체의 실질적인 움직임은 단조로운 변화와 급격한 변화의 혼합된 형태를 따른다. 유용한 질의 결과를 위해서, 객체의 단조로운 변화뿐만 아니라 급격한 변화도 중요하게 고려되어야 한다. 현실적인 실험 환경을 위하여 실세계 공간 데이타와 여러 가지 갱신 패턴을 고려하는 이동체들을 인위적으로 생성하였다. 제안한 방법이 이동체들의 빈번한 갱신들을 시공간 색인에 반영하기 위해서 필요한 디스크 액세스 수를 상당히 줄였다. 존재하는 색인 기법들의 일반적인 갱신 방법과 비교하였으며, 제안한 방법이 일반적인 방법에 비교하여 더욱 좋은 결과를 보였다.

참 고 문 헌

[1] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Schneider, "A Data Model and Data Structures for Moving Objects Databases," Proceedings of the ACM SIGMOD Conference on Management of Data, pp.319~330, 2000.

[2] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, "Modeling and Querying Moving Objects," Proceedings of the IEEE International Conference on Data Engineering, pp.422~432, 1997.

[3] P. K. Agarwal, L. Arge, and J. Erickson, "Indexing Moving Points," Proceedings of the ACM Symposium on Principles of Database Systems, pp.175~186, 2000.

[4] G. Kollios, D. Gunopulos, and V. J. Tsotras, "On Indexing Mobile Objects," Proceedings of the ACM Symposium on Principles of Database Systems, pp.261~272, 1999.

[5] D. Pfoser, C. S. Jensen, and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Object Trajectories," Proceedings of the International Conference on Very Large Data Bases, pp.395~406, 2000.

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

[7] Y. Tao and D. Papadias, "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries," Proceedings of the International Conference on Very Large Data Bases, pp.431~440, 2001.

[8] Y. Tao and D. Papadias, "Time-Parameterized Queries in Spatio-Temporal Databases," Proceedings of the ACM SIGMOD Conference on Management of Data, pp.334~345, 2002.

[9] K. Porkaew, I. Lazaridis, and S. Mehrotra, "Querying Mobile Objects in Spatio-Temporal Databases," Proceedings of the Symposium on Spatial and Temporal Databases, pp.59~78, 2001.

[10] O. Wolfson and S. Chamberlain and S. Dao and L. Jiang and G. Mendez, "Cost and Imprecision in Modeling the Position of Moving Objects," Proceedings of the IEEE International Conference on Data Engineering, pp.588~596, 1998.

[11] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha, "Updating and querying databases that track mobile units," Distributed and Parallel Databases, pp.257~387, 1999.

[12] U. S. Bureau of Census, "Tiger/lines precensus files: 1994 technical documentation, Technical report," 1994.

[13] S. Saltenis and C. S. Jensen, "Indexing of Moving Objects for Location-Based Services," Proceedings of the IEEE International Conference on Data Engineering, pp.331~342, 2002.

[14] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," Proceedings of the International Conference on Very Large Data Bases, pp.287~298, 2002.

[15] C. S. Jensen and S. Saltenis, "Towards Increasingly Update Efficient Moving-Object Indexing," IEEE Data Engineering Bulletin, pp.35~40, 2002.

[16] D. Kwon, S. Lee, and S. Lee, "Indexing the current positions of moving objects using the lazy update R-tree," Proceedings of the International Conference on Mobile Data Management, pp.113~120, 2002.

[17] J. v. d. Bercken and B. Seeger, "An Evaluation of Generic Bulk Loading Techniques," Proceedings of the International Conference on Very Large Data Bases, pp.461~470, 2001.

최 용 진

1997년 성균관대학교 정보공학과(학사) 1999년 한국과학기술원 전산학과(석사) 2003년 8월 한국과학기술원 전자전산학과(박사). 현재 한국과학기술원 박사후과정 관심분야 공간 데이타베이스, GIS, 시공간 데이타베이스, LBS

민 준 기

1995년 숭실대학교 전자계산학과(학사) 1997년 한국과학기술원 전산학과(석사) 2002년 8월 한국과학기술원 전산학전공(박사) 2003년 8월 한국과학기술원 박사후과정. 현재 한국과학기술원 전산학 전공 초빙교수. 관심분야는 데이타베이스, XML, 시공간 DB, OLAP

정 진 완

1973년 서울대학교 공과대학 전기공학과(학사). 1983년 University of Michigan 컴퓨터공학과(박사). 1983년~1993년 미국 GM 연구소 전산과학과 선임연구원 및 책임연구원. 1993년~현재 한국과학기술원 전산학과 부교수 및 교수. 관심분야는 XML, 멀티미디어 데이타베이스, GIS, 웹 정보검색, 객체지향 데이타베이스