이동객체의 현재 위치정보 관리를 위한 셀 기반 색인 기법

이 응 재[†] · 이 양 구[†] · 류 근 호[††]

요 약

모바일 환경에서 정보 제공 및 처리의 대상이 되는 사람, 자동차, 비행기 등과 같은 이동객체는 시간이 경과함에 따라 끊임없이 자신의 위치를 변경하는 특징을 가지며, 이러한 정보들을 데이터베이스에서 효과적으로 처리하기 위해서는 연속적인 위치 변경을 수용할 수 있는 저장 공간과 색인 유지 및 관리 기술이 필요하다. 기존의 색인 기법들은 색인의 유지 성능보다 질의 처리 능력을 향상시키는데 노력을 기울여 왔기 때문에 복잡하게 이동하며 매우 빈번하게 위치 정보를 갱신하는 이동객체 정보를 관리하기 어렵다. 이 논문에서는 질의 처리 성능뿐만 아니라 이동객체의 빈번한 위치 갱신을 효율적으로 처리하기 위해 고정 그리드 방식의 색인과 R-Tree의 장점을 결합한 ACAR-Tree(Adaptive Cell index with Auxiliary R-Tree)를 제안한다. 제안된 ACAR-Tree는 R-Tree에서 색인의 재구성으로 인하여 갱신 성능이 저하되는 단점을 보완하기 위하여 고정 그리드 방법의 셀 기반의 색인 방법을 기초로 한다. 또한 고정 그리드 방법에서의 데이터 분포가 편중되었을 경우의 색인 성능 저하 문제를 해결하기 위하여, 셀과 버켓이 1 : 1로 매핑되는 셀에서 오버플로가 발생하였을 경우 해당 버켓이 부가적인 색인 구조인 보조 R-Tree로 전환하여 처리한다. 제안된 방법을 다양한 데이터 분포 및 데이터 크기에서 실험한 결과, 기존의 R-Tree 기반의 색인 방법과 비교하여 색인의 크기가 크게 감소하였으며, 질의 및 갱신 성능에 대해서도 뛰어난 성능을 보였다.

A Cell-based Indexing for Managing Current Location Information of Moving Objects

EungJae Lee[†] · YangKoo Lee[†] · KeunHo Ryu[††]

ABSTRACT

In mobile environments, the locations of moving objects such as vehicles, airplanes and users of wireless devices continuously change over time. For efficiently processing moving object information, the database system should be able to deal with large volume of data, and manage indexing efficiently. However, previous research on indexing method mainly focused on query performance, and did not pay attention to update operation for moving objects. In this paper, we propose a novel moving object indexing method, named ACAR-Tree. For processing efficiently frequently updating of moving object location information as well as query performance, the proposed method is based on fixed grid structure with auxiliary R-Tree. This hybrid structure is able to overcome the poor update performance of R-Tree which is caused by reorganizing of R-Tree. Also, the proposed method is able to efficiently deal with skewed-, or gaussian distribution of data using auxiliary R-Tree. The experimental results using various data size and distribution of data show that the proposed method has reduced the size of index and improve the update and query performance compared with R-Tree indexing method.

키워드 : 이동객체(Moving Objects), 이동객체 색인(Moving Object Indexing), 시공간 데이터베이스(Spatiotemporal Indexing), 위치기반 서비스(Location-Based Service)

1. 서 론

최근 무선 통신 기술의 발달과 초고속 인터넷의 보급은 모바일 사용자에게 다양한 응용 서비스를 가능하게 하였다[1]. 특히, 휴대폰, 노트북, PDA 등과 같은 위치 데이터의 전송이 가능한 단말기를 탑재한 자동차나 이를 휴대한 사용자들의 위치 정보를 위성항법장치(GPS : Global Positioning System)를 이용해 실시간으로 추적할 수 있게 되었다. 또한 위치 정보를 이용하여 사용자에게 다양한 서비스를 제공하는 위치기반서비스(LBS : Location Based Service)[2-4]는 현재 활발히 연구되고 있는 응용 분야이며, 교통 통제 시스템, 항법 시스템, 전장 분석 시스템 등의 분야에도 다양하게 응용되고 있다[5, 6]. 모바일 환경에서 위치 정보를 생성하고 그와 연관된 정보를 제공받는 차량, 선박, 비행기, 사람 등은 시간이 흐름에 따라 자신의 위치를 끊임없이 변경시키는 특징이 있다. 이러한 시간이 경과함에 따라 자신의 위치를 연속적으로 변경시키는 객체를 이동객체(moving objects)라고 한다[7-9].

이동 객체의 위치 정보를 관리하기 위한 기존의 색인 기법들은 주로 사용자의 질의에 대한 빠른 응답 시간에 초점

※ 이 연구는 과학기술부 RRC(청주대학교 ICRC)와 대학 IT 연구센터 육성 및 지원 사업에 의한 것입니다.
† 준 회 원 : 충북대학교 대학원 전자계산학과
†† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
논문접수 : 2004년 7월 21일, 심사완료 : 2004년 8월 13일

을 맞추어 연구되어 왔다. 하지만 모바일 환경에서 이동객체는 시간이 경과함에 따라 자신의 위치를 끊임없이 변경시키기 때문에 객체의 위치 갱신은 매우 빈번하게 발생한다. 시스템의 측면에서 이동객체의 위치 갱신은 이전의 위치 정보를 삭제하고 새로운 위치 정보를 삽입하는 연산을 통해 수행되는데, 이러한 갱신 연산은 시스템에 심각한 부담을 준다. 특히 R-Tree 기반의 색인 구조에서는 위치 정보 변경에 따른 색인 구조의 재조직화가 빈번하게 발생하며, 현재 데이터가 삽입되는 영역의 데이터들뿐만 아니라 기존의 다른 데이터 영역에 대해서도 영향을 주기 때문에 색인의 갱신 성능이 크게 저하된다. 또한 갱신 성능의 저하는 전체 시스템의 성능을 저하시키는 요인이 된다.

최근 모바일 환경에서의 이동객체 정보의 갱신 성능을 향상시키기 위하여 단말 노드로부터 루트 노드로 상향 접근을 수행하는 방법들이 제안되고 있다. 이러한 방법들은 기존의 R-Tree와 비교하여 갱신 성능은 크게 향상시켰지만, Tree의 상향 접근을 위한 부가적인 구조를 유지해야 하는 부담과 색인의 재조직화가 빈번히 발생하는 문제를 여전히 갖고 있다.

이 논문에서는 모바일 환경에서 이동하는 객체를 이동 점객체로 정의하고, 빈번한 갱신이 발생하는 모바일 환경을 효과적으로 지원할 수 있는 ACAR-Tree(Adaptive Cell index with Auxiliary R-Tree) 색인 방법을 제안한다. 제안된 색인은 이동 객체의 현재 위치 정보를 관리하기 위한 색인으로 고정 그리드 구조와 R-Tree를 결합한 형태의 구조를 갖는다. 제안된 방법에서 이동객체의 위치는 고정 그리드를 통해 셀 단위로 관리되며, 하나의 셀 내에서 오버플로가 발생할 경우에만 해당 셀 내의 데이터들에 대해서 R-Tree를 생성한다. 따라서 보조 색인 구조로써 사용되는 R-Tree는 오버플로가 발생하는 셀에 대해서만 생성되기 때문에, 고정 그리드 구조에서 발생하는 편중된 데이터 처리의 성능 저하 문제를 해결할 수 있다. 하지만 제안된 ACAR-Tree는 고정 그리드 방법과는 다르게 초기에 모든 셀에 버켓을 할당하지 않고, 데이터 밀도가 낮은 여러 개의 셀들이 하나의 버켓을 공유하도록 함으로써 저장 공간을 효율적으로 활용하도록 한다. 이러한 적응적 버켓 할당 방법은 이동객체가 데이터 밀도가 낮은 영역을 이동할 때 발생하는 셀 간의 데이터 갱신을 줄이는 효과를 얻을 수 있다.

이 논문의 구성은 다음과 같다. 먼저 2장에서는 이동객체를 효율적으로 관리하기 위해 기존에 제안된 색인 기법들에 대해 살펴보고, 문제점을 분석한다. 3장에서는 이동객체의 현재 위치 정보를 관리하기 위해 이 논문에서 제안하는 색인 기법의 구조에 대해 기술하며, 4장에서는 제안된 색인 방법을 위한 알고리즘을 제시한다. 그리고 5장에서는 기존의 색인과 제안된 색인과의 비교를 통하여 성능을 분석하고, 마지막으로 결론을 맺는다.

2. 관련 연구

기존의 이동객체의 정보를 다루는 색인 연구들은 크게 이동객체의 과거 이력 정보 및 궤적을 다루는 색인에 대한 연구와 현재 위치와 가까운 미래 위치를 다루는 색인에 대한 연구로 나눌 수 있다.

전자의 경우에 해당하는 색인 연구로는 3DR-Tree[10], STR-Tree, TB-Tree[11] 등이 있다. 3DR-Tree는 기존의 R-Tree에 시간 차원을 추가하여 3차원으로 표현한 색인으로 시간 관련 질의에는 좋은 성능을 보이지만, 궤적 관련 질의에는 성능이 떨어지는 단점이 있다. STR-Tree와 TB-Tree는 특히, 이동 객체의 궤적과 관련된 질의를 빠르게 처리를 위해 궤적을 보호하며 색인을 구성함으로써, 궤적 관련 질의에 좋은 성능을 보인다.

후자의 경우에 해당하는 연구로는 R-Tree, TPR-Tree[1], LUR-Tree[12], Bottom-Up Update[13] 등이 있다. R-Tree는 대표적인 공간색인 기법으로써 객체를 포함하는 MBR(Minimum Bouncing Rectangle)의 겹침을 최소화하여 색인을 구성한다. 이러한 구조에서 이동객체의 갱신을 수행하는 과정은 R-Tree에서 객체가 포함된 단말 노드를 검색한 후, 엔트리를 삭제하고, 객체의 새로운 위치를 삽입하기 위한 최적의 노드를 탐색함으로써 이루어지는데, 이는 모든 위치 갱신에서 노드의 분할 및 합병을 발생시킬 수 있고, 이에 따른 색인의 재구성으로 인해 Disk I/O 횟수가 현저히 증가하여 전체 색인의 성능을 크게 악화시키게 된다. 따라서 R-Tree를 이용하여 연속적인 위치 갱신이 발생하는 이동객체의 동적 속성을 관리하기는 매우 어렵다.

TPR-Tree는 이동객체의 갱신 비용을 감소시키기 위해 객체의 이동을 간단한 선형 함수로 표현하는 방법이다. 하지만 객체의 이동이 매우 복잡할 경우, 간단한 선형 함수는 객체의 이동을 저장하는데 적합하지 않고, 많은 갱신 연산을 요구하게 된다. 실제로, 객체의 위치를 서술하기에 적합한 선형 함수는 거의 존재하지 않기 때문에, 선형 함수는 객체의 위치 정보를 정확하게 저장할 수 없는 문제를 포함하고 있다.

LUR-Tree와 Bottom-Up Update는 보조 저장 구조를 통해 R-Tree의 단말 노드에 직접 접근하고, 상위 노드를 상향으로 접근하여 색인을 갱신하는 방법을 이용한다. 또한 갱신 효율을 위해 MBR을 확장하도록 하였지만, 이로 인하여 색인의 검색 성능이 R-Tree보다 오히려 나빠지는 단점이 있다. 또한 단말 노드부터 루트 노드로 상향 탐색하기 위해 항상 상위 노드 포인터를 유지해야 하는 문제점을 가지고 있다.

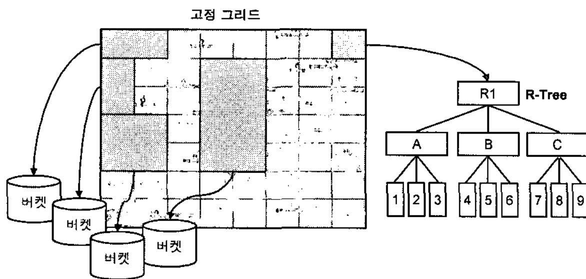
3. 제안된 색인 구조

3.1 색인의 전체 구조

제안된 방법은 초기에 전체 공간 영역을 Nx×Ny 크기의 셀로 분할한다. 초기 셀의 개수와 크기는 사용자에 의해 미리 정

의된 값이고, 셀 하나의 크기는 (Domainx/Nx, Domainy/Ny)로 정의된다. (그림 1)은 제안된 색인의 전체 구조를 나타내고 있다. 그림에서와 같이 전체 데이터 공간은 고정 그리드 방법에서처럼 일정한 크기의 셀들로 분할되고, 각각의 셀에 고유 ID를 할당하여 관리한다.

기존의 고정 그리드는 모든 셀과 버켓에 대해 1 : 1로 매핑되어 각각의 셀은 자신의 버켓을 할당 받았다. 하지만 객체가 분포되어 있지 않은 셀까지도 버켓이 할당되어 색인의 크기가 증가하며, 데이터 밀도와는 무관하게 각각의 셀에 버켓이 할당되는 문제가 발생한다. ACAR-Tree는 저장 공간을 효율적으로 관리하기 위해 객체의 밀도가 낮고 서로 인접한 여러 개의 셀들이 하나의 버켓을 공유하도록 한다. 따라서 초기에는 모든 셀들은 하나의 버켓을 공유하며, 객체의 삽입과 삭제로 인해 버켓에 오버플로나 언더플로 등이 발생하면 버켓을 공유하는 셀들이 포함하고 있는 객체의 개수에 따라 분할과 합병 과정을 거친다.



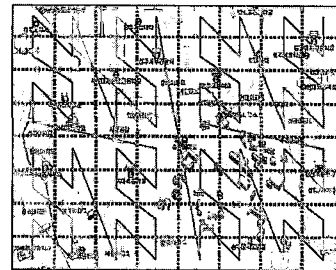
(그림 1) 전체 색인 구조

만약, 하나의 셀과 하나의 버켓이 대응되고, 버켓에 오버플로가 발생하면 기존의 고정 그리드는 새로운 버켓을 생성하며, 버켓 간에는 체인으로 연결하였다. 그러나 이런 경우, 검색이 발생했을 때 체인으로 연결된 버켓들을 순차 탐색하게 되므로 색인으로써의 기능을 잃게 된다. ACAR-Tree는 셀과 버켓이 1 : 1로 대응되는 경우 오버플로가 발생하는 버켓에 대해 셀의 크기를 전체 데이터 공간으로 하는 R-Tree를 생성하고, cell ID가 R-tree의 루트를 가리키도록 조정하여, 버켓의 순차 탐색으로 인한 부하를 해결한다.

3.2 Cell ID 관리와 버켓의 할당

ACAR-Tree에서 전체 공간은 고정 그리드를 사용하여 고정된 크기의 셀로 일정하게 분할되고, 각각의 셀은 ID를 할당하여 관리된다. cell ID의 부여는 Z-Ordering[14] 방법으로 각 셀을 순회하면서 순차적으로 생성된 고유 번호로 할당되고, 이러한 방법을 통해 모든 셀을 ID의 순서에 따라 클러스터링할 수 있게 된다. 이렇게 셀들이 순서를 갖도록 함으로써 여러 개의 셀 집합을 포함하고 있는 버켓이 분할될 때, 셀들을 적절하게 두 개의 버켓에 분배할 수 있게 된다. 이 과정에서 cell ID와 셀에 부여된 영역에 대한 매핑을 위한 저장 공간은 요구되지 않으며, 이는 Z-Ordering의 ID 부여 방법에 의하여 cell ID와 영역 좌표간의 변환이 가능하다.

(그림 2)는 전체 공간에서 모든 셀들에 대해 Z-Ordering 방법으로 ID를 부여하는 과정을 나타낸 것이다. 그림에서 보는 바와 같이 cell ID를 관리하는 Bucket Table은 ID와 ID에 할당된 Bucket Data로 구성된다. 여기서 ID는 Z-Ordering 운행으로 부여된 cell ID를 나타내고 Bucket Data의 엔트리는 <type, ID, N>으로 구성된다.



Bucket Table

ID	Bucket Data
0	<type, ID, N>
1	
2	
3	
~	
n-1	
n	

(그림 2) Z-Ordering 과정과 Bucket Table 구조

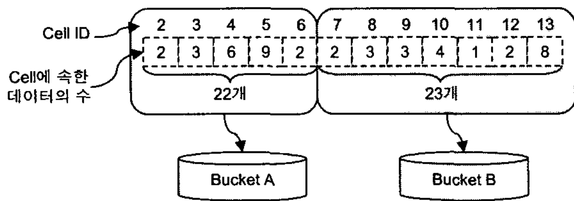
Bucket Data의 엔트리 속성 type은 셀에 대응되는 버켓이 고정 그리드 방식에서의 구조인지, R-Tree 노드인지를 나타낸다. ID 속성은 셀에 포함된 객체를 저장하고 있는 버켓의 주소를 말한다. 마지막으로 N 속성은 각 셀들이 버켓을 공유하는 것과는 상관없이 각각의 셀에 포함된 객체의 개수이다. 이 정보는 여러 셀들이 공유하는 버켓이 오버플로가 발생할 때, 어느 셀을 기준으로 분할하고, 어느 셀들을 통합할 것인가를 결정하기 위하여 사용된다.

ACAR-Tree는 여러 개의 셀들이 하나의 버켓을 공유하도록 하고, 필요에 따라서 버켓을 공유하고 있는 셀들과의 분할/합병을 수행하여 버켓을 관리한다. 여기서 버켓을 공유할 수 있는 셀은 객체의 밀도가 작고, 순차적으로 연속된 cell ID를 갖는 셀들의 집합이다. 만약, 여러 개의 셀들이 공유하고 있는 버켓에 오버플로가 발생했을 경우에는 새로운 버켓을 할당하고, 셀들의 집합을 두개의 버켓에 적절하게 분배되어 저장되도록 한다.

버켓이 분할되는 과정을 살펴보면, 최초에 전체 데이터 공간에 대해 하나의 버켓이 할당되고, 버켓에서 오버플로가 발생할 때까지 객체가 위치한 셀에 대응되는 버켓에 삽입된다. 만약, 오버플로가 발생할 경우, 새로운 버켓을 생성하고 각각의 셀들을 적절하게 나누어 저장한다. 여기서 셀들의 분할은 (그림 3)과 같이 버켓을 공유하는 각각의 셀들이 갖는 데이터 수의 분포를 분석하여, 객체의 분포에 따라 버켓을 공유하는 셀들이 적응적으로 분산되도록 하여, 두 버켓에 가장 최적으로 분할될 수 있도록 한다.

(그림 3)에서 보는 바와 같이 버켓의 분할은 셀의 개수와 상관없이 오버플로가 발생한 버켓에 포함된 셀들이 갖고 있는 객체의 개수를 고려하여 수행된다. 이때, 버켓을 공유하

는 셀들이 공간적으로 인접하거나 가까운 위치에 있는 셀들로 군집화되도록 Z-Ordering을 통해 Cell ID를 할당한다. 이렇게 함으로써, 질의 처리 시에 같은 버켓의 중복 검사를 제거할 수 있고, 또한 disk I/O 횟수를 줄일 수 있는 장점이 있다. 반대로 언더플로가 발생했을 경우에 버켓을 합병하는 방법은 간단하다. 먼저, 언더플로가 발생한 셀을 기준으로 앞과 뒤의 cell ID를 검사하여, 가장 적은 객체를 포함하고 있는 버켓을 찾아 합병을 수행한다.



(그림 3) 버켓의 분할

3.3 노드 구조

노드의 구조는 크게 버켓 형태의 구조와 R-Tree 노드 형태의 구조로 구분된다. 하나의 셀과 버켓이 1 : 1 대응되며 오버플로가 발생한 경우, 해당 셀에 포함된 객체들은 R-Tree 형태로 관리된다. 따라서 노드의 구조는 R-Tree 노드의 형태를 가지며, 그 외의 경우는 고정 그리드 방식에서의 버켓이 갖는 노드 구조를 갖는다. 먼저, 고정 그리드의 버켓 엔트리는 <x, y, moID, dataPtr>로 구성된다. 여기서 x와 y는 이동 점 객체의 좌표 정보이고, moID는 이동객체의 ID를 나타낸다. 마지막으로 dataPtr은 moID와 연관된 recordID 또는 데이터 포인터를 의미한다. (그림 4)는 고정 그리드 형태의 버켓 구조를 나타낸 것이다.

(그림 4)에서 bucketType 속성은 현재의 버켓이 일반 고정 그리드 방식의 버켓인지, R-Tree의 루트 노드인지를 의미한다. startCell과 endCell 속성은 버켓을 공유하는 연속된 셀들의 집합이 시작되는 cell ID와 끝나는 cell ID를 의미한다. 여기에 포함되는 startCell과 endCell 사이에는 생략되는 ID가 없는 연속적인 ID의 범위를 갖는다. 만약, startCell과 endCell이 같을 경우, 이 버켓은 단 하나의 셀에만 대응되는 버켓임을 의미한다. 마지막으로 used 속성은 버켓에 포함된 모든 이동객체의 개수를 나타낸다.

하나의 셀과 하나의 버켓이 1 : 1로 매핑된 셀에 오버플로가 발생하면, 하나의 셀 범위를 루트 노드의 범위로 갖는 R-Tree가 생성되고, 오버플로가 발생한 이후의 모든 갱신은 모두 셀 내부에서 R-Tree로 처리된다. 이렇게 셀 내부에서 생성된 R-Tree는 오버플로가 발생한 셀을 검색할 때, 버켓을 순차 탐색하지 않고 R-Tree를 이용해 빠르게 검색할 수 있는 장점이 있다. 또한, 셀 내부에서 생성된 R-Tree의 깊이는 전체 R-Tree와 비교하였을 경우, 매우 낮기 때문에 탐색해야 하는 노드의 수를 줄일 수 있고, 이로 인해 갱신 성능 또한 향상되는 장점이 갖는다.

Bucket

// Bucket information
bucketType : byte
startCell : int
endCell : int
used : int
// General attribute
x : int
y : int
moID : int
ptr : int

(그림 4) Bucket 클래스

R-Tree 노드의 엔트리는 <MBR, ptr>로 구성된다. (그림 6)은 R-Tree의 노드 구조를 클래스로 나타낸 것이다.

R-Tree Node

// Node information
bucketType : byte
level : int
used : int
// General attribute
x1 : int
y1 : int
x2 : int
y2 : int
ptr : int

(그림 5) R-Tree 노드 클래스

(그림 5)에서 bucketType 속성은 버켓이 고정 그리드 방식에서의 버켓인지, R-Tree의 루트 노드인지를 나타내는 상수이다. 그리고 level 속성은 R-Tree의 레벨 정보이고, used 속성은 노드에 저장된 엔트리의 개수를 나타내는 카운터이다. 나머지 x1, x1, x2, y2와 ptr은 노드의 MBR 정보와 자식 노드 또는 객체가 저장된 레코드에 대한 포인터를 나타낸다.

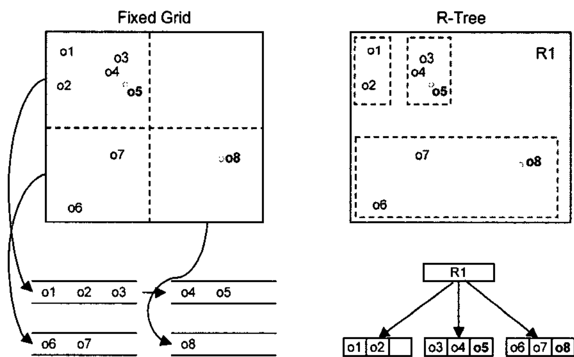
3.4 그리드 기반의 색인과 R-Tree의 고찰

기존의 R-Tree는 다차원 데이터를 관리하기 위해 사용하는 대표적인 색인 구조로써, 대부분의 동적 데이터를 처리하는 환경에서 좋은 성능을 보인다. 그러나 모바일 환경에서 이동객체는 시간의 경과에 따라 연속적으로 위치를 이동하기 때문에 색인에서 관리되어야 할 위치 정보도 매우 빈번하게 갱신된다. 이러한 이동객체 데이터를 R-Tree를 이용하여 관리할 경우, 데이터 갱신 도중에 빈번하게 발생하는 노드의 분할 및 합병에 의해 전체 색인을 재구성하여야 하며, 이로 인해 갱신 비용은 급격히 증가하게 된다. 특히, 색인 내에 저장되어야 할 데이터의 양이 커질수록 트리의 깊이는 깊어지며, 동시에 색인의 재구성 비용 또한 증가하는 단점이 있다.

이에 반해 고정 그리드 구조의 색인은 전체 공간 영역을 일정한 크기의 셀로 분할하고, 각각의 셀에 대응되는 버켓을

할당하여 색인을 구성한다. 이러한 방법은 R-Tree와 같이 노드의 탐색으로 인한 Disk I/O를 요구하지 않으므로 색인의 재구성으로 인한 갱신 성능이 저하되는 문제가 발생하지 않는다. 따라서 색인의 유지 및 관리가 쉽고, 셀에 대응되는 버켓에 간단한 연산에 의하여 직접 접근할 수 있기 때문에 갱신 비용을 크게 줄일 수 있는 장점이 있다. 그러나 고정 그리드 구조는 이동객체의 분포가 특정 지역에 편중되어 있는 경우, 특정 버켓에서만 오버플로가 발생하게 되고, 오버플로의 처리를 위하여 새로운 버켓을 할당하여 리스트로 연결하여 처리한다. 따라서 데이터가 편중된 분포를 가질 경우, 질의 처리 시 연결된 버켓들을 순차 탐색을 수행하여야 하기 때문에 질의 성능이 저하되는 문제가 발생한다.

(그림 6)은 객체의 분포에 따라 R-Tree와 고정 그리드 색인을 비교한 것이다. 고정 그리드의 경우, 객체 o5의 위치는 오버플로가 발생된 셀이고, o5를 검색하기 위해서는 두개의 버켓을 순차 탐색해야 한다. 하지만 R-Tree는 객체의 분포에 따라 색인의 구조가 결정되기 때문에 데이터 분포와는 상관없이 일정한 성능을 보인다. 이와 반대로 객체 o8의 경우, 고정 그리드는 한번의 디스크 접근으로 객체 o8의 위치를 찾는 반면, R-Tree는 객체 o8을 검색하기 위해 루트부터 노드를 탐색하기 때문에 오버플로가 발생하지 않은 경우, 고정 그리드 방식이 R-Tree보다 더 효율적이다.



(그림 6) 고정 그리드와 R-Tree의 비교

일반적으로 기존의 공간데이터베이스 분야에서 편중된 데이터 분포를 갖는 경우, 고정 그리드 방식의 단점을 해결하기 위하여 Grid File[15]이 제안되었다. 하지만 Grid File 계열의 색인 방법들은 처리하여야 할 데이터의 양이 커질수록 디렉토리를 메모리 내에서 처리하지 못하게 되며, 디렉토리의 관리도 어려워 지는 문제가 발생한다. 즉 Grid File 계열의 방법에서는 디렉토리 관리 방법에 따라 시스템 성능에 큰 영향을 준다. 특히 모바일 환경에서는 객체가 빈번하게 위치를 이동하며 위치정보를 갱신하며, 이에 따라 데이터의 분포도 시간의 흐름에 따라 지속적으로 변화한다. 따라서 어떤 특정 시점에서의 데이터 분포를 기준으로 하는 디렉토리 관리는 의미가 없어지며, 매우 빈번하게 디렉토리 정보를 변경해야 하는 문제가 발생한다.

4. 제안된 색인의 알고리즘

제안된 방법은 기본적으로 고정 그리드를 사용하여 이동객체를 관리하고 R-Tree는 오버플로가 발생한 각각의 셀에 대해서만 구성된다. 그리고 최초의 삽입이 발생하기 전에, 먼저 고정 그리드를 초기화 하는 과정이 수행된다. 색인의 초기화는 방법은 하나의 버켓을 할당하고, 전체 공간을 고정된 셀로 분할한다. 그런 다음 Z-Ordering 방법으로 cell ID를 계산하여 모든 셀이 초기화된 버켓을 가리키도록 버켓 테이블을 초기화함으로써 이루어진다.

4.1 삽입 알고리즘

먼저, 객체의 삽입이 발생하면 객체의 위치 좌표 x, y를 사용하여 객체가 위치할 cell ID를 획득하고, 버켓 테이블을 통해 획득된 cell ID에 대응하는 버켓 ID를 얻은 후 해당 버켓을 접근한다. 만약, 접근한 Bucket Type이 R-Tree라면, 주어진 셀은 오버플로 상태이고, 하나의 셀 크기를 갖는 R-Tree가 구성되어 있게 된다. 따라서 R-Tree의 삽입 알고리즘에 따라 객체를 삽입한다.

```
Algorithm Insert (e : NewEntry)
Input : NewEntry (새로운 이동객체)
Begin
  cid = Cell ID using location(e)
  bid = Bucket ID in Bucket Table pointed by cid
  bk = ReadBucket(bid)
    if(Bucket Type in bk is RTROOT) then
      InsertRtree(e)
    else
      if(bk is not overflow) then
        InsertData(bk, e)
      else
        if(the same value between startCell and endCell in bk) then
          InsertRtree(e)
        else
          nbk = CreatNewBucket()
          SplitCellSet(bk, nbk)
          UpdateBucketTable(bk, nbk)
        end if
      end if
    end if
end
```

(알고리즘 1) 삽입 알고리즘

만약, Bucket Type이 Bucket이고 오버플로 상태가 아니라면, 해당 버켓에 객체를 삽입한다. 그렇지 않고 Bucket Type이 오버플로 상태라면, 먼저, 셀과 버켓이 1 : 1로 매핑되는가를 체크하고, 만약, 매핑된다면 셀에서 오버플로가 발생한 것이기 때문에 새로운 R-Tree를 생성하여 객체를 삽입한다. 그렇지 않으면 버켓은 여러 개의 셀들을 포함하고 있는 것이므로 새로운 버켓을 할당하고 셀 집합을 적절하게 분

할한 후, Bucket Table을 갱신한다. (알고리즘 1)은 이러한 수행 순서를 바탕으로 설계된 삽입 알고리즘이다.

4.2 삭제 알고리즘

삭제 알고리즘은 삽입 알고리즘과 마찬가지로 먼저 객체의 위치 좌표 x, y를 사용하여 객체가 위치할 cell ID를 획득한 후, 버켓 테이블을 통해 획득된 cell ID에 대응하는 버켓 ID를 얻고, 해당하는 버켓을 접근한다. 버켓에 접근한 후에는 접근한 Bucket Type에 따라 두 가지의 경우로 객체의 삭제가 발생한다. 먼저, 버켓이 R-Tree의 루트 노드라면, R-Tree로부터 삭제할 객체를 찾아 삭제한다. 객체를 삭제한 후에 R-Tree가 오직 루트 노드만을 포함하고 있다면, R-Tree는 더 이상 유지되지 않는다. 따라서 이런 경우, R-Tree의 루트 노드는 Bucket page로 변환하게 된다. 만약, 객체의 삭제 이후에도 여전히 R-Tree를 유지하고 있다면, 기존의 R-Tree의 삭제 알고리즘에 따라 객체를 삭제한다.

또 다른 경우로, Bucket Type이 Bucket이라면 해당하는 버켓에서 객체를 찾아 삭제한다. 만약, 객체를 삭제한 후에 버켓에서 언더플로가 발생하면, 언더플로가 발생된 버켓의 셀은 연속적으로 인접해 있는 cell ID가 속한 버켓으로의 합병을 수행한다. 만약, 객체를 삭제한 후에 언더플로가 발생하지 않으면, 삭제 연산은 종료된다. 자세한 삭제 알고리즘은 (알고리즘 2)와 같다.

```
Algorithm Delete (e : LeafEntry)
Input : LeafEntry (삭제할 이동객체)
Begin
   cid = Cell ID using location(e)
   bid = Bucket ID in Bucket Table pointed by cid
   bk = ReadBucket(bid)
      if(Bucket Type in bk is RTROOT) then
         DeleteRtree(e)
         if(Root is the same Leaf) then
            Change Bucket Type in bk
         end if
      else
         DeleteData(bk, e)
         if(bk is Underflow) then
            Merge bucket cell IDs in bk
         end if
      end if
end
```

(알고리즘 2) 삭제 알고리즘

4.3 검색 알고리즘

영역 질의를 처리하기 위한 검색 알고리즘은 검색 영역을 포함하는 모든 cell ID를 확보하고, Bucket Table을 통해 획득된 cell ID에 대응하는 버켓 ID를 얻은 다음 각각의 모든 Bucket IDs를 검색한다. 그런 후에, 질의 영역을 포함하는 모든 Bucket IDs를 검색할 때까지 각각의 버켓에 접근하고, 만약, Bucket Type이 R-Tree이라면 R-Tree의 검색 알고리즘에 따라 질의 범위에 일치하는 객체를 검색하고, Bucket Type이 Bucket이라면 버켓 안에 있는 객체를 검색함으로써 범위 질의를 처리한다. (알고리즘 3)은 자세한 검색 알고리즘을 나타낸다.

```
Algorithm Search (R : Range)
Input : Range (검색 영역)
Output : oIDs (검색된 이동객체 IDs)
Begin
   while(Cell IDs are included Range) do
      cid = Cell ID using location(R)
      bid = Bucket ID in Bucket Table pointed by cid
      bk = ReadBucket(bid)
      if(Bucket Type in bk is RTROOT) then
         result += SearchRtree(R)
      else
         SearchData(R)
      end if
   end while
   return result
end
```

(알고리즘 3) 검색 알고리즘

5. 실험 및 평가

이 논문에서 제안하고 비교 평가된 R-Tree, ACAR-Tree는 C++ 언어를 이용하여 구현하였고, 펜티엄 IV 2.6GHz, RAM 1GB 메모리를 사용하는 MS Windows Server 2003 시스템에서 실험 평가하였다. 실험에 사용된 데이터는 이동 객체 환경에서 모의 실험 데이터로 많이 사용되고 있는 GSTD(Generator of SpatioTemporal Datasets)[16]를 이용하여 생성하였다. 데이터 셋은 GSTD에서 제공되는 기본 파라미터를 이용하였으며, 시간이 지남에 따라 다양하게 변화하는 데이터 분포에 대한 색인의 성능을 평가하기 위하여 Uniform, Gaussion, Skewed 분포를 갖는 실험 데이터를 생성하였다. 또한 모바일 정보를 생성하는 이동 객체 수의 변화에 따른 색인의 성능 변화를 확인하기 위하여 위치 정보를 생성하는 객체의 수가 1000, 5000, 10000이라고 가정하고 실험 데이터를 생성하였다. 실험에 사용한 성능 파라미터는 <표 1>과 같다.

〈표 1〉 성능 파라미터

파라미터	사용된 값
이동객체 수	1000, 5000, 10000
데이터 분포	Uniform, Gaussian, Skewed
버켓의 크기	1024

제안된 색인의 성능을 검증하기 위하여 다음과 같은 항목에 대하여 R-Tree와 비교 평가하였다. 먼저, GSTD에서 제공하는 이동객체의 세 가지 이동 패턴을 기준으로 크게 갱신과 검색에 대한 성능을 평가하였다. 갱신의 경우 색인 구축 시간에 대한 비교와 이동객체의 삽입/삭제에 따른 Disk

I/O 횟수 비교 등으로 세부적인 평가를 하였다. 검색의 경우 전체 데이터 공간의 10%, 20% 검색 영역을 임의로 생성하여 질의를 수행하였으며, 질의에 대한 Disk I/O 횟수를 비교 분석하였다.

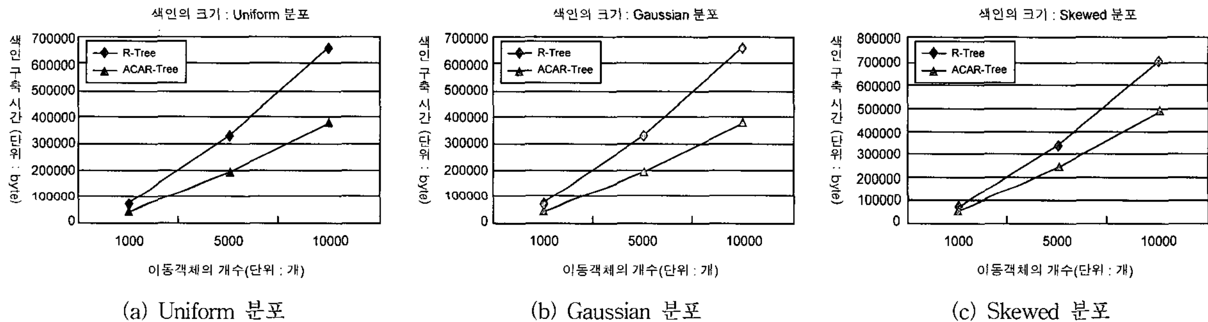
5.1 데이터 분포별 색인의 크기

색인의 크기는 특정 색인의 성능을 좌우하는 Disk I/O 비용과 밀접한 관련이 있기 때문에 반드시 고려되어야 한다. (그림 7)은 각각의 데이터가 Uniform, Gaussian, Skewed 분포를 가질 경우의 색인 크기를 비교한 그림이다. 그림에서 보는 바와 같이 제안된 색인 방법이 R-Tree보다 Uniform 분포와 Gaussian 분포에서 약 40% 정도 색인의 크기가 감소되었음을 알 수 있다. 이는 고정 그리드 형태에서의 버켓에 저장되는 이동 객체의 위치 정보는 점 데이터 형태로 저장되지만, R-Tree의 경우 MBR 형태로 저장되기 때문에 동일한 버켓 크기에서 저장할 수 있는 데이터의 수가 R-Tree가 더 적어지기 때문이다.

(그림 7)(c)는 이동객체의 분포가 Skewed일 경우의 색인 크기를 비교한 그림이다. 그림에서 보는 바와 같이 Skewed 분포에서도 ACAR-Tree가 R-Tree보다 약 30% 정도 색인의 크기가 감소되었다. 하지만, Skewed 분포의 경우, ACAR-Tree에서 하나의 셀에 대응되는 버켓이 오버플로가 될 가능성이 커지기 때문에 R-Tree 노드로 변환되는 버켓이 증가한다. 따라서 (그림 7)(a), (그림 7)(b)의 데이터 분포와 비교하여 전체 색인의 크기도 증가하게 된다.

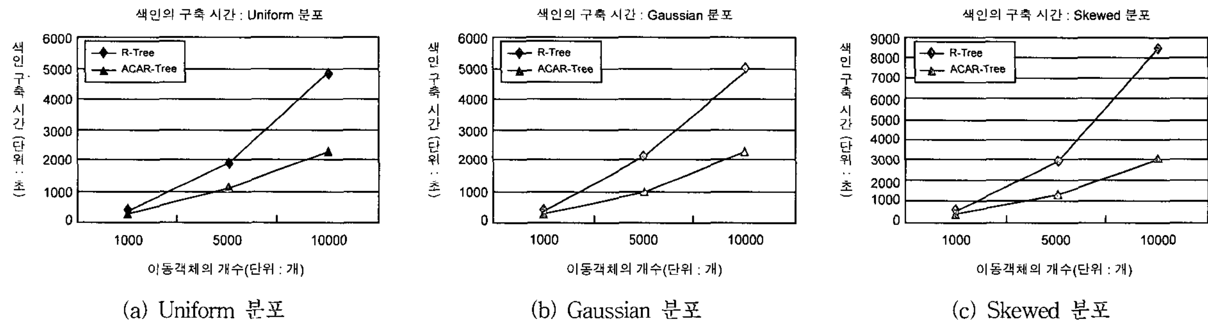
5.2 데이터 분포별 색인의 구축 시간

실험에서 비교한 색인의 구축 시간은 이동객체의 모든 위치 변경에서 발생하는 삭제/삽입 연산의 수행 시간이다. (그림 8)는 각각의 데이터 분포에 대한 색인의 구축 시간을 비교한 그림이다. 그림을 통해 알 수 있듯이, 각각의 분포에서 ACAR-Tree가 R-Tree보다 좋은 성능을 보이고, 이동객체의 수가 증가함에 따라 제안된 방법의 성능이 선형적으로 증가하는 것을 알 수 있다. 특히, R-Tree는 이동객체의 수가 많아질수록 구축 시간이 현저히 증가하는 반면, ACAR-Tree는 선형적으로 안정되게 증가한다.



(a) Uniform 분포 (b) Gaussian 분포 (c) Skewed 분포

(그림 7) 데이터 분포별 색인의 크기 비교



(a) Uniform 분포 (b) Gaussian 분포 (c) Skewed 분포

(그림 8) 데이터 분포별 색인의 구축 시간 비교

5.3 데이터 분포별 색인의 갱신 성능

이동객체의 갱신 연산 비용은 색인의 노드에 접근하는 횟수를 통해 산출할 수 있다. 갱신은 객체의 이전 위치 정보 삭제와 새로운 위치 정보의 삽입을 통해 수행된다. 따라서 전체 갱신 성능은 삽입 연산과 삭제 연산에서 접근한 Disk I/O 횟수의 합을 비교함으로써 평가된다. (그림 9)은 각각의 데이터 분포에서 이동객체의 갱신 성능을 비교한 그림이다. 그림을 통해 알 수 있듯이 제안된 ACAR-Tree가 R-Tree보다 좋은 갱신 성능을 보이고 있다.

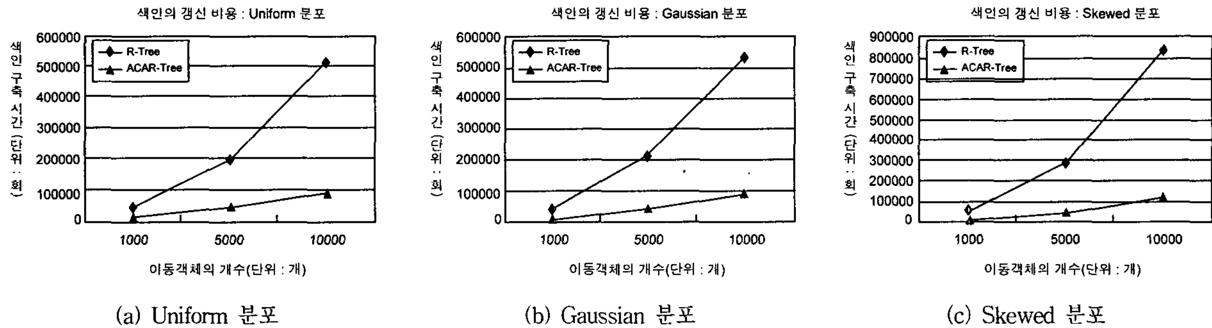
갱신 연산에서 R-Tree는 객체의 삭제/삽입 연산을 수행하기 위해 루트 노드부터 노드를 탐색하고, 갱신으로 인해 변경된 노드 정보를 반영하기 위해 색인을 재구성하는 경우가

빈번하게 발생한다. 따라서 색인의 크기가 클수록 탐색해야 하는 노드의 수가 증가하기 때문에 전체 색인의 구축 시간은 현저히 증가하게 된다. 반면에, 제안된 방법은 고정 그리드를 통해 객체의 위치에 직접 접근하므로 노드의 탐색 시간을 제거할 수 있고, 만약, 하나의 셀에 대응되는 버켓이 오버플로가 발생하여 R-Tree로 관리된다고 하더라도, 셀에 생성된 R-Tree의 크기는 매우 작다. 따라서 전체 색인의 구축 시간은 R-Tree보다 현저히 줄어드는 것을 알 수 있다.

5.4 데이터 분포별 영역 질의 성능

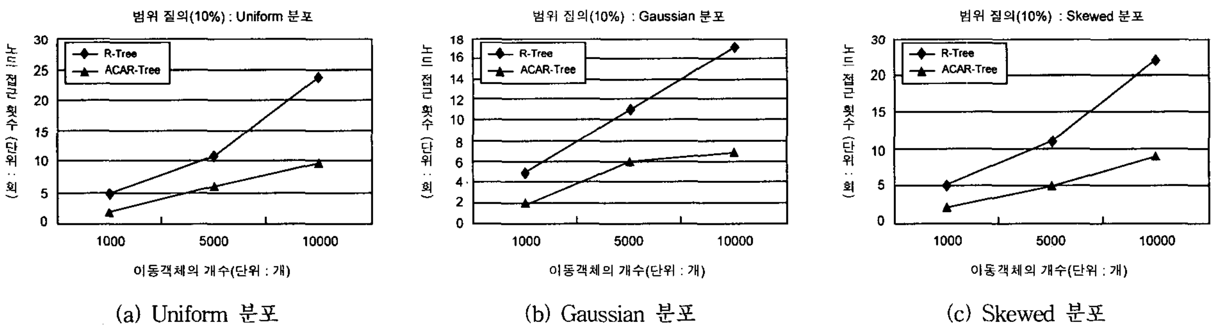
영역 질의는 "2004년 5월 13일 충북대학교에 있는 모든 차량을 검색하시오."와 같이 공간에 대한 영역을 정하여 검색하는 것이다. 이러한 영역 질의는 이동객체 환경에서도 중요한 질의 중의 하나이다. 영역 질의에 대한 실험은 전체 공간 영역의 10%, 20% 영역을 각각의 데이터 분포에 따라 임의로 생성하여 평가된다. (그림 10)은 각각의 데이터 분포에 대한 10% 영역 질의를 비교한 그림이다. 그림에서와 같이 제안된 ACAR-Tree가 R-Tree보다 좋은 성능을 보인다. 이러한 결과는 ACAR-Tree의 경우, R-Tree에서 객체를 포함하는 단말 노드를 검색하기 위해 루트 노드부터 비단말 노드를 탐색하는 과정이 제거되거나, 매우 작기 때문에 나타난다. 5.3절에서 언급한 것처럼 R-Tree의 경우 질의 처리를 위하여 트리의 루트 노드로부터 여러 노드를 탐색하여야 하기 때문에 질의 처리 비용이 증가한다. 특히 R-Tree내에 Overlap되는 데이터가 많아질 경우, 질의 처리를 위하여 방문해야할 노드가 증가하기 때문에 질의 성능은 크게 저하된다.

(그림 11)는 각각의 데이터 분포에 대한 20% 영역 질의를 비교한 그림이다. 그림에서와 같이 20% 영역 질의도 10% 질의와 마찬가지로 제안된 방법이 R-Tree보다 모든 분포에서 좋은 성능을 보이고 있다.



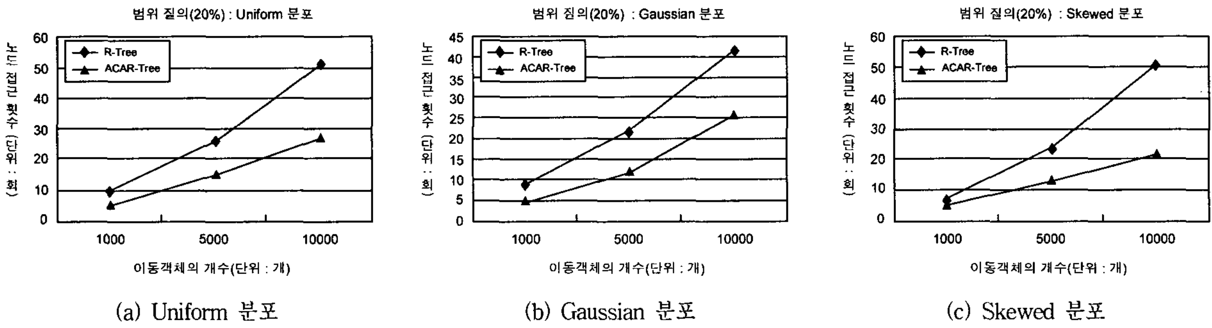
(a) Uniform 분포 (b) Gaussian 분포 (c) Skewed 분포

(그림 9) 데이터 분포별 색인의 갱신 성능 비교



(a) Uniform 분포 (b) Gaussian 분포 (c) Skewed 분포

(그림 10) 데이터 분포별 색인



(a) Uniform 분포 (b) Gaussian 분포 (c) Skewed 분포

(그림 11) 데이터 분포별 색인의 영역 질의 성능 비교(20% 영역 질의)

5.4 실험 결과 분석

지금까지 실험한 결과를 요약하면, 이 논문에서 제안된 ACAR-Tree는 모든 데이터 분포에서 R-Tree보다 색인의 크기와 구축 시간이 현저히 감소되었고, 갱신 성능 또한 매우 향상되었다. 그 이유는 ACAR-Tree가 버켓을 통해 많은 셀들을 공유하여 관리하므로 각각의 버켓에 대해 저장 효율을 최대화 할 수 있고, 오버플로가 발생한 셀에 대해서만 제한적으로 매우 작은 크기의 R-Tree가 생성되므로 색인의 재구성으로 인한 성능 저하를 최소화하기 때문으로 보인다.

특히, 제안된 ACAR-Tree는 데이터 분포가 Uniform일 때 가장 좋은 성능을 보이며 Gaussian 분포, Skewed 분포 순으로 좋은 성능을 보였다. 이는 ACAR-Tree가 이동객체의 이동 패턴에 관계없이 전체 영역에서 데이터의 분포가 규칙적이라면 좋은 성능을 보이고, 데이터의 분포가 불규칙적일수록 성능이 낮아짐을 의미한다. 그 이유는 이동 객체 데이터가 특정 셀에 집중되는 경우 R-Tree 형태의 데이터 관리를 하기 때문이다. 하지만 모든 데이터가 하나의 셀에 집중되는 최악의 경우를 제외하고는 기존의 R-Tree보다 좋은 성능을 보이는 것을 알 수 있다.

검색 성능의 경우도 모든 질의에 대해 제안된 ACAR-Tree가 R-Tree보다 매우 좋은 성능을 보이고 있다. 이는 R-Tree에서 검색 영역을 포함하는 단말 노드를 찾기 위해 R-Tree의 루트 노드부터 비단말 노드를 탐색하는 과정이 ACAR-Tree에서는 검색 영역에 해당하는 cell ID에 직접 접근함으로써 이루어지기 때문에 노드 탐색으로 인한 Disk I/O를 현저히 감소시키기 때문이다.

6. 결론 및 향후 연구

모바일 사용자에게 위치 정보와 관련된 다양한 서비스의 제공을 가능하게 하기 위해서는 이동객체의 연속적인 위치 변경을 수용할 수 있는 저장 공간과 새로운 색인 기술이 필요하다. 특히 모바일 환경에서 객체의 빈번한 위치 이동은 시스템에 잦은 갱신 연산 수행을 요청하게 되며, 이로 인하여 전체 시스템 성능을 저하시키는 문제를 발생시킨다.

이 논문에서는 이러한 이동객체의 빈번한 위치 갱신 요구를 효율적으로 처리하기 위한 구조로써 고정 그리드와 R-Tree 구조를 이용한 색인 기법인 ACAR-Tree를 제안하였다. 제안된 방법은 이동객체의 위치를 빠르게 갱신하기 위해 고정 그리드와 마찬가지로 셀 단위로 객체 정보를 저장하고, 오버플로가 발생한 셀에 대해서만 R-Tree로 관리함으로써 색인의 갱신 성능과 다양한 데이터 분포에도 적응할 수 있도록 하였다. 또한 여러 개의 셀들이 하나의 버켓을 공유할 수 있도록 함으로써, 저장 공간의 효율성과 더불어 데이터 밀도가 낮은 지역에서의 정보를 효과적으로 관리하도록 하였다. 또한 다양한 실험 및 분석을 통하여 제안된 방법이 기존의 R-Tree보다 색인의 크기, 색인 구축 시간, 갱신 비용 및 질의 처리 성능 면에서 현저히 향상됨을 확인하였다.

향후 연구로는 실세계의 데이터를 사용해서 기존에 존재하는 다양한 이동객체 색인 기법과의 평가를 통한 성능 분석이 필요하다. 또한 Hilbert Curve 등의 다른 형태의 Cell ID 부여 방법을 통하여, 버켓을 공유하는 셀들의 클러스터링과 색인의 성능과의 상관관계에 대한 연구도 진행할 예정이다.

참 고 문 헌

[1] S. Saltenis, C. Jensen, S. Leutenegger and M. Lopez, "Indexing the Positions of Continuously Moving Objects," In Proc. of the 19th ACM-SIGMOD Conference, Dallas, Texas, pp.331-342, 2000.

[2] 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.

[3] 양영규, "위치 기반 서비스(LBS : Location Based Service) 기술 현황 및 전망", 정보처리학회지, Vol.8, No.6, pp.4-6, 2001.

[4] 이상정, "위치기반서비스에서의 무선 측위 기술", 제1회 국제 LBS 기술 워크샵, pp.77-100, 2002.

[5] 배종철, 박성숭, 안윤애, 류근호, 주재우, "시공간 추론 개념을 이용한 전장분석", 한국정보과학회 추계학술발표, Oct., 2000.

[6] 신기수, 안윤애, 배종철, 정영진, 류근호, "GIS를 이용한 시공간 이동 객체 관리 시스템", 정보처리학회논문지D, 제8-D권 제2호, Apr., 2001.

[7] M. Erwig, R. H. Guting, M. Schneider and M. Vazirgiannis, "Spatio-Temporal Data Types : An Approach to Modeling and Querying Moving Object in Databases," CHOROCHRONOS Technical Report CH-97-08, December, 1997.

[8] R. H. Guting, M. H. Bohlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, M. Vazirgiannis, "A foundation for representing and querying moving objects," ACM Transactions on Database Systems, Vol.25, No.1, pp.1-42, 2000.

[9] 류근호, 안윤애, 이준욱, 이용준, "이동객체 데이터베이스와 위치기반서비스의 적용", 데이터베이스연구회지, Vol.17, No.03, pp.57-74, 2001.

[10] Y. Theodoridis, M. Vazirgiannis, T. Sellis, "SpatioTemporal Composition and Indexing for Large Multimedia Applications," ACM Multimedia Systems, pp.284-298, 1998.

[11] D. Pfoser, C. S. Jensen and Y. Theodoridis, "Novel Approaches in Query Processing for Moving Objects," CHOROCHRONOS Technical Report CH-00-3, February, 2000.

[12] D. Kwon, S. Lee and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," In Mobile Data Management, MDM, pp.113-120, 2002.

[13] M. Lee, W. Hsu, C. jensen, B. Cui and K. Teo, "Supporting Frequent Updates in R-Trees : A Bottom-Up Approach," In Proceedings of the 29th International Conference on

VLDB, September, 2003.

[14] J. A. Orenstein and T. H. Merrett, "A Class of Data Structures for Associative Searching," In Proceedings of ACM International Symposium on Principles of Database Systems(PODS), Waterloo, Ontario, Canada, pp.181-190, April, 1984.

[15] J. Nievergelt, H. Hinterberger and K. C. Sevcik, "The Grid File : An Adaptable, Symmetric Multikey File Structure," ACM Transactions on Database Systems (TODS), Vol.9, No.1, pp.38-71, March, 1984.

[16] Yannis Theodoridis, Jefferson R. O. Silva and Mario A. Nascimento, "On the Generation of Spatiotemporal Datasets," In Proceedings of the 6th Int'l Symposium on Large Spatial Databases (SSD'99), Hong Kong, China, Springer-Verlag LNCS Series, July, 1999.

이 응 재

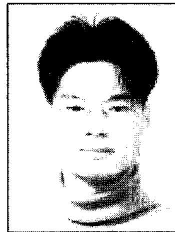
e-mail : eungjae@dblab.chungbuk.ac.kr

1994년 충북대학교 컴퓨터과학과(이학사)

1996년 충북대학교 대학원 전자계산학과 (이학석사)

2001년~현재 충북대학교 대학원 전자계산학과 박사과정

관심분야 : 시공간 데이터베이스, 이동객체 데이터베이스, LBS, 지리정보 시스템



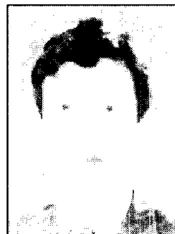
이 양 구

e-mail : leeyangkoo@dblab.chungbuk.ac.kr

2002년 청주대학교 컴퓨터정보공학과 (공학사)

2004년~현재 충북대학교 대학원 전자계산학과 석사과정

관심분야 : 시공간 데이터베이스, 이동객체 데이터베이스, 메인메모리 데이터베이스, GIS, LBS



류 근 호

e-mail : khryu@dblab.chungbuk.ac.kr

1976년 숭실대학교 전산학과(이학사)

1980년 연세대학교 대학원 전산전공 (공학석사)

1988년 연세대학교 대학원 전산전공 (공학박사)

1976년~1986년 육군군수 지원사 전산실(ROTC 장교), 한국전자통신연구원(연구원), 한국 방송대 전산학과(조교수) 근무

1989년~1991년 Univ. of Arizona Research Staff(TempIS 연구원, Temporal DB)

1986년~현재 충북대학교 전기전자컴퓨터공학부 교수

관심분야 : 시간데이터베이스, 시공간 데이터베이스, Temporal GIS 및 지식기반 정보검색 시스템, 데이터마이닝 및 데이터베이스 보안, 바이오인포매틱스