유비쿼터스 센서 네트워크에서 불확실한 데이타의 효율적인 처리를 위한 인덱스

An Index for Efficient Processing of Uncertain Data in Ubiquitous Sensor Networks

김동오* / Dong-Oh Kim **강홍구**** / Hong-Koo Kang
홍동숙*** / Dong-Suk Hong **한기준****** / Ki-Joon Han

요 약

유비쿼터스 센서 네트워크 관련 기술의 급속한 발전으로 센서가 여러 응용 분야에서 널리 활용되고 있다. 일반적으로, 유비쿼터스 센서 네트워크에서 각 센서 노드로부터 센싱되는 데이타는 검색의 효율성을 위해 중앙 서버에 저장된다. 이러한 환경에서 센싱된 데이타의 갱신 비용을 줄이기 위한 갱신 지연 등으로 인해 중앙 서버에 불확실한 데이타가 저장되며, 이로 인해 질의 처리 시 잘못된 결과를 야기할 수 있다.

본 논문에서는 유비쿼터스 센서 네트워크에서 불확실한 데이타 처리 방법에 대해서 살펴보고, 불확실한 데이타를 효율적으로 처리하기 위한 인덱스를 제시한다. 이 인덱스는 불확실한 데이타가 실제 존재할 가능성이 있는 영역인 불확실성 영역 내에서 갱신을 지연시킴으로써 갱신 비용을 감소시킨다. 특히, 갱신 지연은 특정 갱신 영역 내에서만 수행되도록 제한함으로써 갱신 지연으로 인해 검색의 정확성이 감소되는 문제를 해결한다. 마지막으로, 성능 평가를 통해 이 인덱스의 성능을 분석하여 우수성을 입증한다.

Abstract

With the rapid development of technologies related to Ubiquitous Sensor Network (USN), sensors are being utilized in various application areas. In general, the data sensed by each sensor node on ubiquitous sensor networks are stored into the central server for efficient search. Because update is delayed to reduce the cost of update in this environment, uncertain data can be stored in the central server. In addition, Uncertain data make query processing produce wrong results in the central server.

Thus, this paper examines how to process uncertain data in ubiquitous sensor networks and suggests a new index for efficient processing of uncertain data. The index reduces the cost of update by delaying update in uncertainty areas. Uncertainty areas are areas where uncertain data are likely to exist. In addition, it solves the problem of low accuracy in search

■ 논문접수 : 2006.10.24 ■ 심사완료 : 2006.12.18

* 교신저자 건국대학교 컴퓨터공학부 강의교수 (dokim@db.konkuk.ac.kr)
** 건국대학교 대학원 컴퓨터공학과 박사과정 (khhang@db.konkuk.ac.kr)
*** 건국대학교 대학원 컴퓨터공학과 박사과정 (dshong@db.konkuk.ac.kr)
**** 건국대학교 컴퓨터공학부 교수 (kjhan@db.konkuk.ac.kr)

resulting from update delay by delaying update only for specific update areas. Lastly, we analyze the performance of the index and prove the superiority of its performance by comparing its performance evaluation.

주요어 : 유비쿼터스 센서 네트워크, 불확실한 데이타, 인덱스, 불확실성
Keyword : Ubiquitous Sensor Network, Uncertain Data, Index, Uncertainty

1. 서 론

최근 온도 센서, RFID, GPS 등과 같은 다양한 데이타를 센싱하는 센서 기술과 CDMA, WiFi, WiBro 등과 같은 무선 통신 기술 등이 발전함에 따라 환경 모니터링, 차량 도난 감지 등과 같은 유비쿼터스 센서 네트워크(ubiquitous sensor network) 관련 기술에 대한 관심과 연구가 증대되고 있다[1,2]. 유비쿼터스 센서 네트워크란 많은 종류의 센서 노드들이 무선 통신 방식으로 연결되어 구성되는 네트워크를 말하는 것으로써, 일반적으로 이러한 환경에서는 검색의 효율성을 위해 유비쿼터스 센서 네트워크의 각 센서 노드에서 센싱된 데이타는 중앙 서버(또는 외부 서버)에 저장되고 검색된다[3,4].

또한, 중앙 서버에서는 데이타 갱신으로 인한 갱신 비용을 줄이기 위해 각 센서 노드에서 중앙 서버로 데이타를 보고하는 간격을 늘리게 된다. 그러나 이러한 갱신 지연과 같은 측정 방법의 부적합에서 발생하는 랜덤 오차와 센싱된 데이타의 계통적 오차 때문에 센서 노드에서 센싱된 데이타는 불확실성을 가질 수 있는데, 이러한 데이타 불확실성(data uncertainty)으로 인하여 질의 처리 시 잘못된 결과를 야기할 수 있다[5,6]. 따라서 유비쿼터스 센서 네트워크의 데이타를 효율적으로 처리하기 위해서는 불확실한 데이타(uncertain data)를 효율적으로 처리하기 위한 방법이 필요하다.

유비쿼터스 센서 네트워크에서 발생되는 불확실한 데이타를 효율적으로 처리하기 위해서 불확실성을 고려한 다양한 인덱스가 연구되었다[2,7,8]. 그러나 이러한 인덱스들은 데이타 불확실성을 고려하여 검색의 성능은 개선하였으나, 자원이 제한된 유비쿼터스 센서 네트워크에서 고려해야 할 갱신 성능을 고려하지 못했다. 따라서 유비쿼터스 센서 네트워크에서 센싱된 불확실한 데이타를 효율적으로 처리하기 위해서는 갱신 비용을 최소화할 수 있는 인덱싱에 대한 연구가 필요하다.

따라서 본 논문에서는 유비쿼터스 센서 네트워크에서 불확실한 데이타 처리 방법에 대해서 살펴보고, 불확실성 영역(uncertainty area) 내에서 센싱된 데이타의 갱신을 지연시킴으로써 갱신 비용을 줄일 수 있는 불확실한 데이타 처리를 위한 인덱스인 UR-Tree(Uncertainty R-Tree)를 제시한다. UR-Tree는 불확실한 데이타가 실제 존재할 가능성이 있는 영역인 불확실성 영역을 이용해 각 센서 노드에서 센싱된 데이타를 인덱싱하며, 또한 불확실성 영역 내에서 갱신을 지연시킴으로써 갱신 비용을 감소시킨다. 특히, 갱신 지연으로 인해 검색의 정확도가 감소하는 문제를 해결하기 위해 특정 영역 내에서만 갱신을 지연시킨다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 데이타 불확실성과 불확실성을 고려한 데이타 처리에 대하여 살펴본다. 제 3 장에서는 본 논문에서 제시한 UR-Tree의 인덱스 구조와 상세 알고리즘에 대해 기술하고, 제 4 장에서는 UR-Tree의 성능 평가를 위해 사용된 성능 평가 환경과 성능 평가에 대해 설명한다. 마지막으로, 제 5 장에서는 본 논문의 결론에 대해 언급한다.

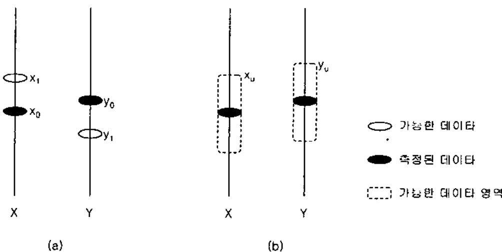
2. 관련 연구

본 장에서는 관련 연구로써 데이타 불확실성과 불확실성을 고려한 데이타 처리에 대해 살펴본다.

2.1 데이타 불확실성

데이타 불확실성은 데이타가 측정 방법의 부적합에서 발생하는 랜덤 오차와 계통적 오차로 인해 확실하지 않은 성질을 말한다. 본 논문에서 불확실성이라 함은 데이타 불확실성을 의미하며, 이처럼 불확실성을 가지는 데이타를 불확실한 데이타라고 한다. 데이타 불확실성은 데이타의 정확성(accuracy), 타당성(validity), 품질(quality), 에러(error)나 노이즈(noise), 신빙성(reliability) 등의 특성과 밀접한 관련을 가진다[9,10].

일반적으로 센서 네트워크에서의 불확실성이 발생하는 요인은 센서 자체의 정밀도, 오차 범위 등의 문제로 인해 발생하는 센서 자체 요인, 센서의 위치, 방향 등에 의해 발생하는 데이타 수집에 따른 요인, 데이타 처리 및 전송으로 인한 시간 지연 등으로 인해 발생하는 데이타 처리에 따른 요인 등이 있다. 그림 1은 데이타 불확실성의 예를 보여준다.



<그림 1> 데이타 불확실성의 예

그림 1은 두 개의 센서 X와 Y에서 측정된 데이타를 보여준다. 그림 1(a)에서 x_0, y_0은 중앙 서버에 보고된 데이타이고, x_1, y_1은 센서 노드에서 현재 측정된 데이타라고 가정하자. 이 때, 두 개의 센서에서 더 높은 데이타를 측정한 센서는 중앙 서버에 저장된 데이타에서 검색한 경우 센서 Y가 되지만, 실제 현재 센서 노드에서 센싱된 데이타를 검색한 경우 센서 X가 된다. 즉, 현재 중앙 서버에 보고된 데이타만으로 판단하였을 경우 잘못된 결과가 나올 수 있다. 데이타 불확실성은 이러한 경우와 같이 현재 보고된 데이타와 실제 데이타가 다를 수 있는 것을 의미한다. 그림 1(b)의 x_u, y_u와 같은 가능한 데이타 영역은 불확실성을 가지는 데이타가 실제 존재할 수 있는 영역을 의미하고, 일반적으로 이러한 가능한 데이타 영역은 불확실성 영역 또는 불확실성 범위(uncertainty interval)라고 부른다. 유비쿼터스 센서 네트워크에서 불확실성을 가지는 데이타에 대한 질의를 효율적으로 처리하기 위해서는 불확실성 영역을 고려할 필요가 있다.

2.2 불확실성을 고려한 데이타 처리

불확실성을 고려한 데이타 처리는 이동 객체의 위치를 표현하기 위한 다양한 데이타 모델 분야에서 먼저 연구되기 시작하였다[6]. 또한, 이동 객체의 위치에 대한 효율적인 질의 처리 알고리즘과 빈번히 발생하는 위치 변경을 반영하기 위한 중앙 서버로의 데이타 전송을 최소화하는 방법에 대한 연구도 진행되었다[11]. 특히, 도로와 같이 제한된 장소에서 이동하는 이동 객체는 도로 데이타를 이용함으로써 위치에 대한 불확실성을 최소화하고 갱신 비용을 감소시킨다[12].

유비쿼터스 센서 네트워크가 발전하면서 이동 객체의 위치를 획득하기 위한 센서를 포함한 다양한 센서에서 데이타가 센싱되고 있다. 따라서 다양한 센서에서 센싱된 데이타의 불확실성을 처리하기 위한 데이타 모델과 불확실한 데이타에 대한 질의 타입 및 각 질의 타입에 맞는 질의 처리 전략이 연구되었다. 불확실한

데이타에 대한 질의 타입으로는 확률 질의와 may/must 질의가 있다[2].

확률 질의는 질의 조건으로 특정 데이타의 범위와 확률을 지정하는 질의로써, 예로는 "현재의 온도가 30℃ 이상인 확률이 50%인 센서의 아이디는 무엇인가?" 등이 있다. May/must 질의는 질의 조건으로 특정 데이타의 범위와 may/must 키워드를 지정하는 질의로써, 예로는 "현재의 온도가 30℃ 이상인(또는, 일 수 있는) 센서의 아이디는 무엇인가?" 등이 있다. 본 논문에서 불확실한 데이타에 대한 질의로써 may/must 질의를 이용하였다.

또한, 불확실한 데이타에 대한 nearest neighbor 검색에 대한 효율적인 알고리즘과 확률 범위 검색에 대한 효율적인 인덱스가 연구되었다[13,14]. 그 외에도 센싱된 데이타별로 확률을 가지는 불확실한 데이타의 검색을 효율적으로 처리하기 위한 인덱스(PTI)[7]나 다각형 형태의 불확실성 영역에 대한 처리를 효율적으로 하기 위해 probabilistically constrained region을 이용하여 검색을 처리하기 위한 인덱스가 제시되었다[8]. 그러나 이러한 알고리즘이나 인덱스는 불확실한 데이타에 대한 다양한 검색 성능을 개선하였으나, 자원이 제한된 유비쿼터스 센서 네트워크에서 고려해야 할 갱신 성능을 고려하지 못했다.

이동 객체 분야에서는 갱신 성능을 개선하기 위해 이동 객체의 아이디로 바로 접근하기 위한 보조 인덱스를 사용하거나 MBR을 확장하고 확장된 MBR 내에서 갱신을 지연시킴으로써 갱신 성능을 개선하기 위한 인덱스가 제시되었다[15]. 또한, 유비쿼터스 센서 네트워크의 각 센서 노드에서 센싱된 데이타는 매우 유동적이지만, 긴 시간 동안에 그 값의 편차가 매우 작음을 이용해 평균에서 편차 범위를 벗어난 경우만 갱신하는 인덱스가 제시되었다[16]. 그러나 이러한 인덱스에서는 불확실성에 대한 고려보다는 갱신 성능 개선에만 초점을 두었다. 따라서 유비쿼터스 환경에서 다양한 센서에서 센싱된 데이타의 불확실성을 처리할 수 있으면서 갱신 성능 및 검색 성능이 개선된 인덱스가 필요하다.

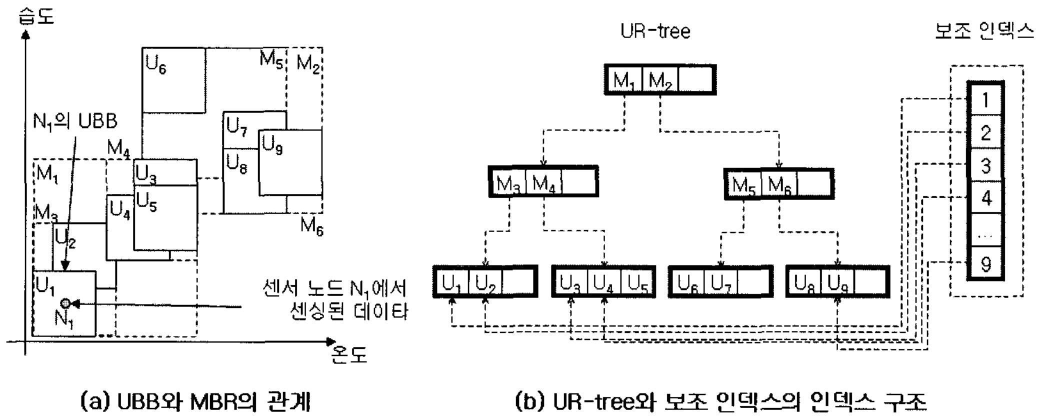
3. UR-Tree

본 장에서는 본 논문에서 제시한 불확실한 데이타의 효율적인 처리를 위한 인덱스로써 불확실성 영역과 갱신 영역을 이용하는 UR-Tree의 인덱스 구조와 상세 알고리즘에 대해 기술한다.

3.1 인덱스 구조

UR-Tree는 불확실성 영역 내에서 센싱된 데이타의 갱신을 지연시킴으로써 갱신 성능을 개선한 불확실한 데이타를 위한 센서 데이타 인덱스이다. UR-Tree의 인덱스 구조는 R-Tree와 유사하나 검색 시에 불확실성 영역을 이용하는 점과 갱신 시에 갱신을 지연시키기 위한 갱신 영역을 이용하는 점에서 구분된다. UR-Tree에서 갱신 요청은 센서 노드의 아이디와 새로 센싱된 데이타로 구성된다. 여기서, 해당 아이디를 가지는 리프 노드에 빠르게 접근하기 위한 방법으로 보조 인덱스를 사용한다. 보조 인덱스의 각 노드는 아이디와 리프 노드 포인터를 가진다. 보조 인덱스는 Hash나 B-Tree로 구현할 수 있다.

유비쿼터스 센서 네트워크를 구성하는 각 센서 노드 N은 센서 노드를 식별할 수 있는 아이디와 센싱을 담당하는 센서 R과 O를 가진다고 가정하자. 이때, 아이디가 i인 센서 노드를 Ni라고 하고, 센서 노드 N_i의 센서 R과 O에서 센싱된 데이터는 $(N_{i.r}, N_{i.o})$이며, 센서 R과 O의 불확실성 영역은 각각 $(-U_r, U_r)$, $(-U_o, U_o)$이라고 하자. 이때, 센서 노드 Ni의 불확실성 영역을 위한 인덱스의 엔트리는 $(N_i, (N_{i.r}-U_r, N_{i.o}-U_o), (N_{i.r}+U_r, N_{i.o}+U_o))$이다. 즉, 센싱된 데이타는 1차원 점 $(N_{i.r}, N_{i.o})$으로 표현



(a) UBB와 MBR의 관계 (b) UR-tree와 보조 인덱스의 인덱스 구조

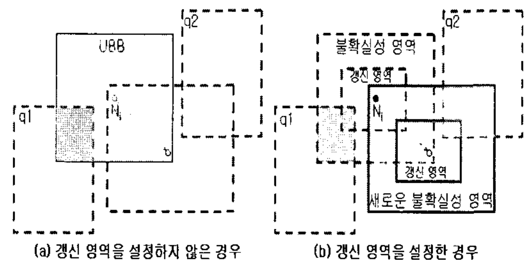
<그림 2> UR-Tree와 보조 인덱스의 구성 예

되며, 해당 데이타의 불확실성 영역은 2차원 사각형 $((N_{i.r}-U_r,\ N_{i.o}-U_o),\ (N_{i.r}+\ U_r,\ N_{i.o}+U_o))$으로 표현된다. 이때, 센싱된 데이타에 대한 불확실성 영역을 Uncertainty Bounding Box (UBB)라고 한다. 그림 2는 UR-Tree와 보조 인덱스를 구성한 예를 보여준다.

그림 2(a)는 센서 노드 N_i의 UBB인 U_i 상호간, 트리 노드의 MBR인 M_j 상호간 또는 Ui와 M_j상호간의 contain 및 overlap 관계를 보여준다. 예를 들어, U_1은 센서 노드 N_1에서 센싱된 데이타의 UBB를 나타내며, M_3은 UR-Tree에서 U_1과 U_2를 포함하는 트리 노드의 MBR을 나타낸다. 그림 2(b)는 그림 2(a)를 한 노드가 최대 3개의 엔트리를 가지는 UR-Tree로 구성한 그림으로써, UR-Tree의 리프 노드는 각 센서 노드의 UBB로 구성된다. 그림 2(b)에서 보조 인덱스는 아이디 기반 접근 시의 속도를 개선하기 위해 UR-Tree에서 해당 노드 아이디를 가지는 리프 노드를 가리키고 있다.

그림 3은 UR-Tree에서 데이타를 갱신하는 모습을 보여준다. UBB 내에서 갱신을 지연시킬 경우 그림 3(a)와 같이 갱신 지연으로 인해 실제로 검색되지 않아야 하는데도 질의 윈도우 q1과 UBB가 Overlap되어 검색되고, 실제로 검색되어야 하는데도 질의 윈도우 q2와 UBB가 Overlap되지 않아 검색되지 않음을 보여준다. 이러한 이유로 갱신을 지연시킬 경우 검색의 정확도를 감소시키게 된다.

따라서 검색의 정확도를 증가시키기 위해 UR-Tree는 갱신 지연을 제한하기 위한 갱신 영역을 가지게 된다. 갱신 영역은 UBB 내부에 존재하며, 데이타의 갱신이 이루어지지 않도록 설정된 영역이다. UR-Tree에서 센싱된 데이타가 갱신 영역을 벗어나지 않았을 경우에는 인덱스 갱신이 일어나지 않으며, 센싱된 데이타가 갱신 영역을 벗어난 경우에는 UBB가 재구성되고, UBB가 인덱스 노드의 MBR을 벗어난 경우에는 인덱스가 재구성된다. 그림 3(b)는 갱신 영역을 설정한 경우 UBB가 재구성됨으로써, 질의 윈도우 q1과 UBB가 Overlap되지 않고, 질의 윈도우 q2와 UBB가 Overlap됨을 보여준다.



(a) 갱신 영역을 설정하지 않은 경우 (b) 갱신 영역을 설정한 경우

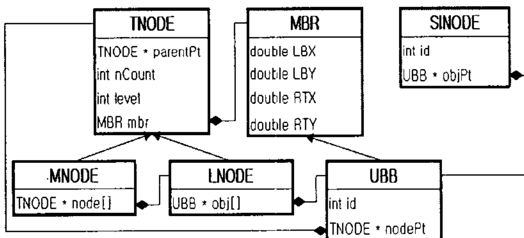
<그림 3> UBB에서의 데이타 갱신

3.2 알고리즘

본 절에서는 UR-Tree에서 사용되는 자료 구조와 삽입, 갱신, 삭제, 검색 알고리즘에 대해 기술한다.

3.2.1 자료 구조

UR-Tree에서 사용되는 자료 구조는 UR-Tree에서 중간 노드 정보를 저장하기 위한 MNODE, 리프 노드 정보를 저장하기 위한 LNODE, 리프 노드에 저장되는 객체로써 센싱된 데이타의 불확실성 영역을 표현하기 위한 UBB, 보조 인덱스의 노드인 SINODE가 있다. MNODE와 LNODE는 UR-Tree의 트리의 구조를 유지하기 위한 자료 구조이다. 그림 4는 UR-Tree를 구성하는 트리 노드들의 자료 구조를 구체적으로 보여준다.



<그림 4> UR-Tree 노드의 자료 구조

그림 4에서 TNODE는 UR-Tree 노드의 공통 속성을 가지는 부모 클래스로써, MNODE와 LNODE는 TNODE로부터 트리 노드가 가지는 기본 속성을 상속받는다. TNODE는 부모 노드의 주소를 저장하기 위한 parentPt, 자식 노드의 개수를 저장하기 위한 nCount, 해당 노드의 레벨을 저장하기 위한 level, 해당 노드의 MBR을 저장하기 위한 mbr로 구성된다. MNODE는 자식 노드의 주소를 저장하기 위한 배열인 node를 가지며, LNODE는 자식 UBB의 주소를 저장하기 위한 배열인 obj를 가진다. MBR은 좌하점의 x, y좌표와 우상점의 x, y좌표로 구성된다. UBB는 센싱된 데이타의 불확실성 영역을 저장하기 위한 자료 구조로써 MBR로부터 상속받으며, 데이타를 센싱하는 센서 노드를 식별하기 위한 id, 해당 UBB의 주소를 저장하고 있는 리프 노드의 주소를 저장하기 위한 nodePt로 구성된다. 마지막으로, SINODE는 보조 인덱스에서 사용하는 자료구조로써, 객체를 식별하기 위한 id, 해당 id에 속하는 UBB의 주소를 저장하기 위한 objPt로 구성된다.

3.2.2 삽입 알고리즘

UR-Tree에서 삽입 알고리즘은 해당 아이디를 가지는 센서 노드에서 센싱된 데이타가 처음 입력될 경우 수행된다. 알고리즘의 입력은 센싱된 데이타의 아이디인 id, 센싱된 데이타 x, y를 가진다. 그리고 출력은 삽입 성공 실패 여부를 반환한다. 또한 삽입될 위치를 찾기 위한 FindLeaf(node, ubb) 함수, 노드 분할을 위한 Split(cnode,ubb) 함수 등으로 구성된다. 그림 5는 UR-Tree의 삽입 알고리즘을 보여준다.

ALGORITHM : Insert(ID *id*, DOUBLE *x*, DOUBLE *y*)

```
    BEGIN
1:   IF( SECONDARY INDEX have a id )
2:     RETURN ERROR
     END IF
3:   UBB ubb ← MakeUBB( id, x, y )
4:   TNODE cnode ← FindLeaf( node, ubb )
5:   IF( cnode.nCount equal NODEORDER )
6:     Split ( cnode, ubb )
     ELSE
7:     insert ubb in cnode
     END IF
    END
```

<그림 5> 삽입 알고리즘

그림 5에서 보듯이 삽입 알고리즘의 수행 과정은 다음과 같다. 먼저, 보조 인덱스를 통해 삽입하려는 id를 가지는 UBB가 있는지 검사하여 해당 아이디가 존재하는 경우 에러를 반환한다. 검사한 결과 해당 아이디가 존재하지 않

는 경우, MakeUBB(id,x,y) 함수를 호출하여 ubb를 생성한다. 다음 FindLeaf(node,ubb) 함수를 호출하여 해당 데이타가 입력될 트리의 리프 노드를 검색하여 cnode에 할당한다. 마지막으로, cnode의 자식 노드가 리프 노드의 최대 자식의 개수인 NODEORDER와 같다면 Split(cnode,ubb) 함수를 호출해 해당 노드를 분할하고, 같지 않다면 cnode에 ubb를 추가한다.

그림 6과 그림 7은 각각 센싱된 데이타가 입력될 트리의 리프 노드를 검색하기 위한 FindLeaf(node,ubb) 함수와 해당 노드의 자식 노드에 오버플로우가 발생했을 경우 노드를 분할하기 위한 Split(node,ubb) 함수를 보여준다.

```
ALGORITHM : FindLeaf( TNODE node, UBB ubb )
    BEGIN
1:    TNODE node ← root node of UBR-tree
2:    WHILE ( node isn't leaf node )
3:      TNODE tnode ← find child node which contain ubb, in node
4:      IF ( tnode is NULL )
5:        FOR EACH child node child of node
6:          MBR newmbr ← ubb + child.mbr
7:          IF( newmbr is minimum )
8:            tnode ← child
          END IF
        END FOR
      END IF
9:    IF( tnode is leaf node )
10:     RETURN tnode
      END IF
11:   node ← tnode
    END WHILE
  END
```

<그림 6> FindLeaf 함수

```
ALGORITHM : Split( TNODE node, UBB ubb )
    BEGIN
1:    TNODE basenode
2:    insert all child object of node and ubb to basenode
3:    sort basenode
4:    TNODE node1, node2
5:    INTEGER i=0
6:    WHILE( i <= basenode.nCount )
7:      IF( i < NODEORDER/2 )
8:        insert basenode[i] to node1
        ELSE
9:        insert basenode[i] to node2
        END IF
10:     i++
      END WHILE
11:   delete node from node.parentPt
12:   insert node1,node2 to node.parentPt
    END
```

<그림 7> Split 함수

그림 6에서 보듯이 FindLeaf(node,ubb) 함수의 수행 과정은 다음과 같다. 먼저, UR-Tree의 루트 노드를 node에 할당한다. 다음 node가 리프 노드일 때까지 WHILE 구문을 반복한다. WHILE 구문의 수행 과정은 다음과 같다. 먼저, node의 자식 노드 중에서 ubb를 포함하는 노드를 tnode에 할당한다. 만약 tnode가 NULL이라면, 즉 자식 노드 중에서 ubb를 포함하는 노드가 존재하지 않는다면, node의 모든 자식 노드 중에서 child.mbr과 ubb를 더한 newmbr이 최소인 자식 노드를 tnode에 할당한다. 마지막으로, tnode가 리프 노드이면 tnode를 반환하고, 그렇지 않으면 tnode를 node에 할당하고 WHILE 루프를 수행한다.

그림 7에서 보듯이 Split(node,ubb) 함수의 수행 과정은 다음과 같다. 먼저, basenode를 생성하고, node의 모든 자식 객체와 ubb를 입력한 후 정렬한다. 다음, 분할될 트리 노드인 node1과 node2를 생성한 후 basenode의 처음부터 NODEORDER/2개까지의 자식 노드를 node1에 입력하고, 그 이후의 자식 노드는 node2에 입력한다. 마지막으로, node의 부모 노드에서 node를 삭제하고 node1과 node2를 입력한다.

3.2.3 갱신 알고리즘

UR-Tree에서는 센싱된 데이타가 갱신 영역을 벗어나지 않았을 경우에는 인덱스 갱신이 일어나지 않는다. 그러나, 센싱된 데이타가 갱신 영역을 벗어난 경우에는 UBB가 재구성되고, UBB가 인덱스 노드의 MBR을 벗어난 경우에는 인덱스가 재구성된다. 그림 8은 UR-Tree의 갱신 알고리즘을 보여준다. 그림 8에서 보듯이 갱신 알고리즘은 Search(id) 함수를 이용하여 해당 아이디의 노드를 검색한다. 다음, 센싱된 데이타가 해당 노드의 갱신 영역에 포함되면 갱신을 끝낸다. 포함되지 않을 경우 MakeUBB() 함수를 이용해 아이디와 센싱된 데이타로 부터 불확실성 영역인 ubb로 작성한다. 그

리고 갱신 전과 후의 MBR이 부모 노드의 MBR에 포함되는지를 검사하여, 포함되는 경우 ubb만 갱신하고, 포함되지 않는 경우 해당 노드를 삭제 후 삽입하는 과정을 통해 재삽입한다.

3.2.4 삭제 알고리즘

UR-Tree에서의 삭제는 특정 아이디에 관한 데이타를 삭제하는 경우에 사용된다. 그림 9는 UR-Tree의 삭제 알고리즘을 보여준다. 그림 9에서 보듯이 삭제 알고리즘은 Search(id) 함수를 이용하여 해당 아이디의 노드를 검색한다. 그리고 해당 노드의 부모 노드에서 해당 노드를 삭제한 후, 해당 노드의 부모 노드 mbr이 변경되는지를 검사한다. 만약 변경되지 않았다면 삭제를 끝내고, 만약 변경되었다면 부모 노드의 mbr을 수정한다. 이 과정은 mbr을 수정하는 노드의 부모 노드 mbr이 변경되지 않을 때까지 반복 수행된다.

```
ALGORITHM : Update( ID id, DOUBLE x, DOUBLE y )
    BEGIN
1:    TNODE leaf ← Search( id )
2:    IF( UPDATEAREA of leaf contain x and y )
3:      RETURN   // update is not need
      ELSE
4:      UBB ubb ← MakeUBB( id, x, y )
5:      MBR newmbr ← add ubb to leaf.mbr
6:      IF( leaf.parentPt.mbr contain leaf.mbr and newmbr )
7:        leaf.mbr ← newmbr
        ELSE
8:        Delete( id )
9:        Insert( id, x, y )
        END IF
      END IF
    END
```

<그림 8> 갱신 알고리즘

```
ALGORITHM : Delete( ID id )
    BEGIN
1:    TNODE leaf ← Search( id )
2:    delete leaf in leaf.parent
3:    IF( leaf.parent.mbr don't change )
4:      RETURN
      ELSE
5:      leaf.parent.mbr update until node.parent.mbr don't change
      END IF
    END
```

<그림 9> 삭제 알고리즘

3.2.5 검색 알고리즘

UR-Tree는 아이디 기반 검색과 윈도우 기반 검색을 지원한다. 윈도우 기반 검색은 may/must 키워드를 사용할 수 있다. 그림 10은 UR-Tree의 아이디 기반 검색 알고리즘을 보여준다.

```
ALGORITHM : Search( ID id )
    BEGIN
1:    SINODE sinode ← search SECONDARY INDEX to find a SINODE with id
2:    IF( sinode is NULL )
3:      RETRUN NULL
      ELSE
4:      UBB ubb ← sinode.objpt
5:      RETURN ubb.nodept
      END IF
    END
```

<그림 10> 아이디 기반 검색 알고리즘

그림 10에서 보듯이 아이디 기반 검색 알고리즘에서는 보조 인덱스를 이용하여 해당 id를 가지는 보조 인덱스의 노드를 검색하여 sinode에 할당 한다. 다음, 검색된 sinode가 NULL일 경우, 즉 해당 id가 존재하지 않을 경우 NULL을 반환하고, 검색된 sinode가 NULL이 아닐 경우 sinode.objpt를 ubb에 할당한 후, ubb를 포함하는 UR-Tree의 리프 노드인 ubb.nodept를 반환한다. 그림 11은 UR-Tree의 윈도우 기반 검색 알고리즘을 보여준다.

그림 11에서 보듯이 윈도우 기반 검색 알고리즘에서는 먼저 해당 노드가 윈도우 영역에 포함되는지 겹치는지를 검사한다. 해당 노드가 윈도우 영역에 포함될 경우 해당 노드의 자식 노드의 객체를 추가 검사 없이 모두 검색 결과에 추가한다. 해당 노드가 윈도우 영역과 겹칠 경우 해당 노드가 리프 노드인지를 검사한다. 해당 노드가 리프 노드가 아닐 경우 Search_Window (mbr,node) 함수를 재귀적으로 수행한다. 그러나 해당 노드가 리프 노드일 경우에는 질의 타입에 따라 구분되어 처리된다. 따라서 keyword가 "MAY" 일 때, 즉 may 질의의 경우 자식 노드가 윈도우 영역에 겹치면 검색 결과에 추

```
ALGORITHM : Search_Window( MBR window, TNODE node, String Keyword )
      BEGIN
        result ← Ø
1:      IF( window contain node.mbr )
2:        add object of all leaf node in node to result
3:      ELSE IF( window overlap node.mbr ) // Search Subtrees
4:        FOR EACH child node child of node
5:          IF( child is leaf node )
6:            IF(( keyword is "MAY") AND ( window contain child.mbr ))
7:              add object in child to result
            END IF
8:            IF(( keyword is "MUST") AND ( window overlap child.mbr ))
9:              add object in child to result
            END IF
          ELSE
10:           Search_Window( window, child, keyword )
          END IF
        END FOR
      END IF
11:     RETURN result
      END
```

<그림 11> 윈도우 기반 검색 알고리즘

가하며, keyword가 “MUST” 일 때, 즉 must 질의의 경우 자식 노드가 윈도우 영역에 포함되면 검색 결과에 추가한다. 마지막으로 검색 결과를 반환한다.

4. 성능 평가

본 장에서는 UR-Tree의 성능 평가를 위해 사용한 성능 평가 환경에 대해서 설명한다. 그리고 UR-Tree에 대한 성능 평가를 수행하고 실험 결과를 통해 본 논문에서 제시한 인덱스의 효율성을 입증한다.

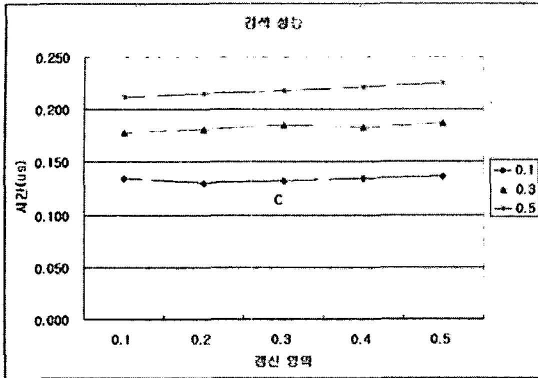
4.1 성능 평가 환경

본 논문에서 성능 평가에 사용된 성능 평가 환경은 Intel Pentium 4 2.6GHz 프로세서와 1GByte 메인 메모리를 사용하였다. 그리고 운영체제는 Windows XP를 사용하였다. 실험을 보다 객관화하기 위해 실험 프로그램을 제외한 모든 프로그램은 가동하지 않은 상태에서 실험을 수행하였다. 본 논문에서 성능 평가에 사용한 데이타는 표 1과 같다.

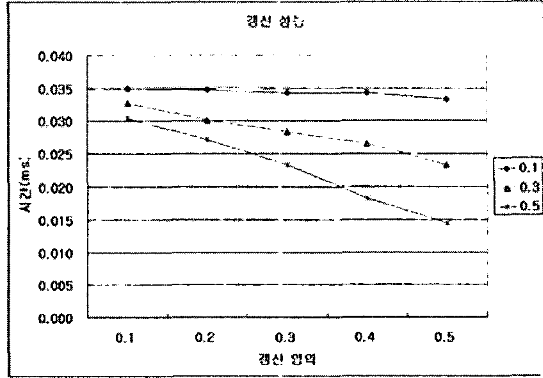
<표 1> 성능 평가 데이타

이름	센서 노드 개수	분포 함수	시작 데이타 범위	평균 이동 범위	최소/최대 이동 범위	전체 데이타 범위	센서 개수
DataSet1	500	Gaussian	10,20	0.2	-0.5,0.5	10,20	2
DataSet2	500	Gaussian	15,20	0.2	-0.5,0.5	10,20	2

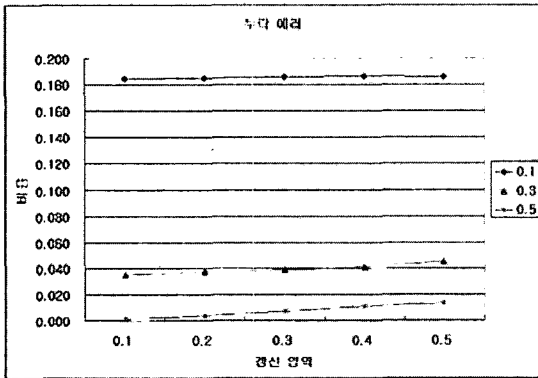
본 논문에서 성능 평가는 갱신 지연을 불확실성의 원인으로 보고, 임의의 시간 T_i에 해당하는 데이타를 불확실한 데이타로 설정하여 인덱스를 구성하였으며, T_{i+1}에 해당하는 데이타를 확실한 실제 데이타로 간주하여 비교하였다. T_i의 데이타에서 검색한 결과의 개수는 R_i, T_{i+1}의 데이타에서 검색한 결과의 개수는 R_{i+1}, T_i와 T_{i+1}의 검색 결과에 공통으로 나타난 결과의 개수는 R_e라고 할 때, 다음 두 가지는 검색의 정확성을 측정하기 위한 성능 지표이다.



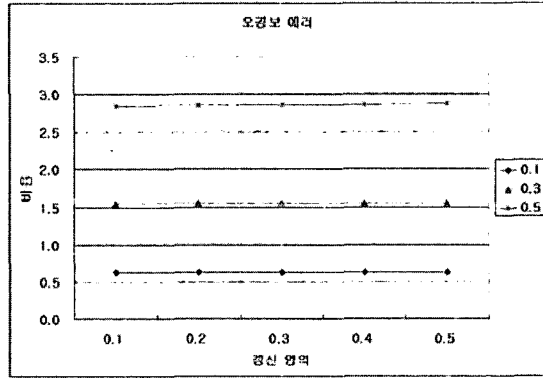
(a) 갱신 영역에 따른 평균 검색 시간



(b) 갱신 영역에 따른 평균 갱신 시간



(c) 갱신 영역에 따른 평균 누락 에러



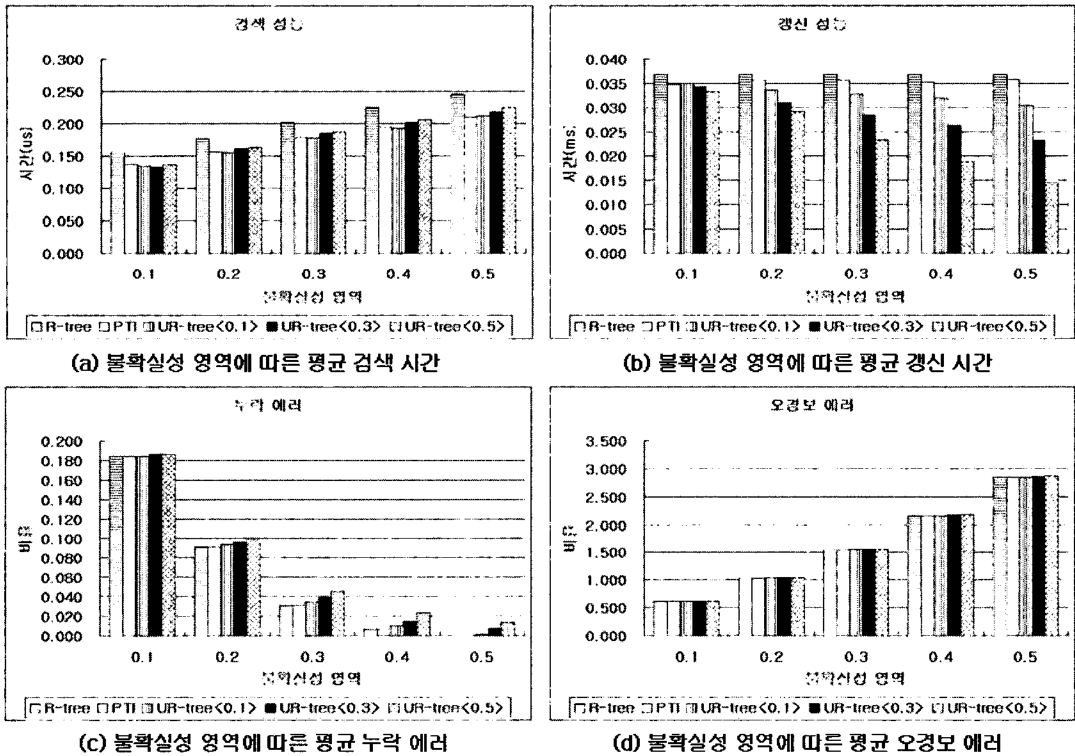
(d) 갱신 영역에 따른 오경보 에러

<그림 12> 갱신 영역에 따른 성능 실험 결과

$$\text{누락 에러} = \frac{\text{검색되어야 할 객체 중 실제로 검색되지 않은 객체의 수}}{\text{검색되어야 할 객체의 수}} = \frac{(R_{i+1} - R_e)}{R_{i+1}} = 1 - \frac{R_e}{R_{i+1}}$$

$$\text{오경보 에러} = \frac{\text{검색된 객체 중 실제로 검색되지 않아야 할 객체의 수}}{\text{검색되어야 할 객체의 수}} = \frac{(R_i - R_e)}{R_{i+1}}$$

누락 에러(Omission error)는 검색되어야 할 객체의 수와 검색되어야 할 객체 중 실제로 검색되지 않은 객체의 수에 대한 비율로써 비율이 낮을수록 좋다. 오경보 에러(Commission error)는 검색되어야 할 객체의 수와 검색된 객체 중 실제로 검색되지 않아야 할 객체의 수에 대한 비율로써 비율이 낮을수록 좋다. 예를 들어, 시간 T_5에서 질의 윈도우 q로 검색한 결과, 즉 인덱스에서 검색된 객체의 아이디가 {3,5,8,9,11,15}이고, T_6에서 질의 윈도우 q로 검색한 결과, 즉 실제로 검색되어야 할 객체의 아이디가 {5,8,12}이라고 하면, R_5, R_6, R_e는 각각 6, 3, 2가 된다. 따라서 누락 에러는 1−2/3 = 0.33이고, 오경보 에러는 (6−2)/3 = 1.33이다. 누락 에러가 0.33이므로 검색되어야 할 아이디가 검색이 되지 않고, 오경보 에러가 1.33이므로 검색되지 않아야 할 아이디가 검색이 됨으로써 에러가 발생하게 되어 검색의 정확도가 감소함을 알 수 있다.



(a) 불확실성 영역에 따른 평균 검색 시간

(b) 불확실성 영역에 따른 평균 갱신 시간

(c) 불확실성 영역에 따른 평균 누락 에러

(d) 불확실성 영역에 따른 평균 오경보 에러

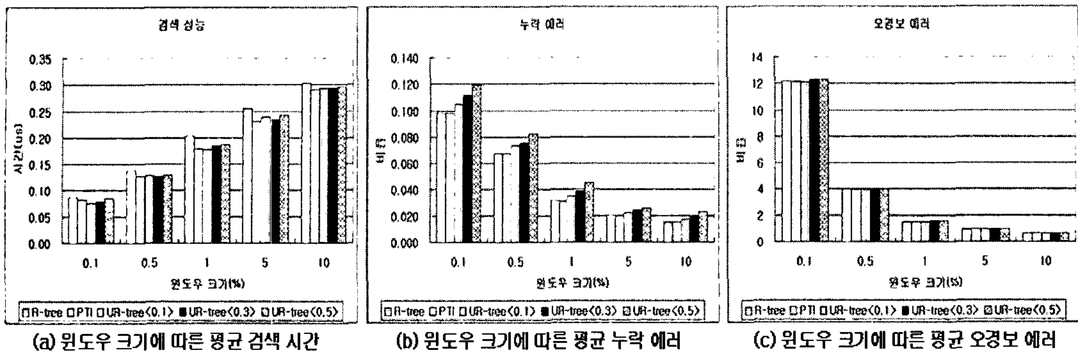
<그림 13> 불확실성 영역에 따른 성능 실험 결과

4.2 UR-Tree의 성능 평가

본 절에서는 UR-Tree의 성능 평가를 위한 실험 및 결과에 대해서 설명한다. UR-Tree의 성능 평가를 위해 불확실성 영역, 갱신 영역, 검색 윈도우의 크기를 변화시키면서 R-Tree 및 PTI와 UR-Tree의 성능을 비교하였다. 성능 평가에 사용한 R-Tree는 불확실성 질의를 처리할 수 없는 인덱스이다. 따라서 질의 영역을 데이타 불확실성을 고려하여 확장함으로써 불확실성 질의를 처리할 수 있도록 하였다. PTI는 인덱스 노드에 확률 질의를 효율적으로 처리하기 위한 확률 테이블을 구성함으로써 불확실성 질의를 처리할 수 있는 인덱스이다.

그림 12는 갱신 영역과 불확실성 영역을 변화시키며 UR-Tree의 성능을 실험한 결과로써, 불확실성 영역이 0.1, 0.3, 0.5일 때, 갱신 영역을 0.0에서 0.5까지 변화시키며 성능을 실험하였다. 그림 12(a)에서 보듯이 불확실성 영역이 커질수록 검색 시간이 증가하고, 그림 12(b)에서 보듯이 불확실성 영역과 갱신 영역이 커질수록 갱신 시간이 감소한다. 그림 12(c)와 그림 12(d)에서 보듯이 에러는 갱신 영역의 변화보다 불확실성 영역의 변화에 크게 영향을 받는다. 이는 불확실성 영역의 변화에 따라 검색되어야 할 데이타가 검색되지 않거나 검색되지 않아야할 데이타가 검색되기 때문이다.

그림 13은 갱신 영역과 불확실성 영역을 변화시키며 UR-Tree의 성능을 실험한 결과로써, 불확실성 영역을 0.0에서 0.5까지 변화시키며 R-Tree, PTI, UR-Tree의 성능을 비교하였다. 여기서 UR-Tree<p>는 갱신 영역이 p%인 UR-Tree를 나타낸다. 그림 13(a)에서 보듯이 검색 성능은 UR-Tree와 PTI가 유사하



(a) 윈도우 크기에 따른 평균 검색 시간 (b) 윈도우 크기에 따른 평균 누락 에러 (c) 윈도우 크기에 따른 평균 오경보 에러

<그림 14> 윈도우의 크기에 따른 성능 실험 결과

나, 그림 13(b)에서 보듯이 불확실성 영역과 갱신 영역이 커질수록 UR-Tree의 갱신 시간이 급격히 감소한다. 그러나 그림 13(c)와 그림 13(d)에서 보듯이 검색 시의 error는 크게 차이가 나지 않으며, 불확실성 영역이 클수록 UR-Tree의 누락 에러가 갱신 지연으로 인해 검색되어야 할 데이타가 검색되지 못하기 때문에 증가한다.

그림 14는 검색 윈도우의 크기를 변화시키며 UR-Tree의 성능을 실험한 결과로써, 검색 윈도우의 크기가 전체의 0.1, 0.5, 1, 5, 10%일 때 R-Tree, PTI, UR-Tree의 성능을 비교하였다. 이 때, 갱신 성능은 질의 윈도우 크기와 무관하기 때문에 성능 평가에서 제외하였다. 그림 14(a)와 그림 14(c)에서 보듯이 검색 윈도우가 커지더라도 UR-Tree와 R-Tree 및 PTI의 검색 성능 및 오경보 에러는 유사하다. 그러나 그림 14(b)에서 보듯이 검색 영역이 매우 작을 수록 경우 UR-Tree가 불확실성 영역을 고려한 검색으로 인해 누락 에러가 커진다.

5. 결 론

유비쿼터스 센서 네트워크에서 일반적으로 검색의 성능을 높이기 위해 센싱된 데이타를 중앙 서버(또는 외부 서버)에 저장하고 검색하는 외부 저장 방식을 사용한다. 이 때, 갱신 비용의 감소를 위한 갱신 지연 시간과 중앙 서버까지 데이타를 전송하는 전송 시간 등으로 인해 데이타는 불확실성을 가지게 된다. 이러한 불확실한 데이타에 대한 검색의 효율성을 위해 불확실성을 고려한 다양한 인덱스에 대한 연구가 진행되었다. 그러나 이러한 인덱스는 불확실성을 고려한 질의 처리와 함께 검색의 효율성은 개선되지만, 잦은 갱신으로 인해 인덱스 갱신 비용이 증가하고 노드 간의 중복 및 검색의 비효율성이 발생하였다.

이러한 문제점을 해결하기 위해 본 논문에서는 유비쿼터스 센서 네트워크에서 불확실한 데이타 처리 방법에 대해서 살펴보고, 센싱된 데이타의 정확도 감소를 최소화하면서 갱신 비용을 줄일 수 있는 불확실한 데이타를 위한 인덱스인 UR-Tree를 제시하였다. UR-Tree는 불확실성 영역을 이용해 데이타를 검색하고, 갱신 영역 내에서 센싱된 데이타의 갱신을 지연시킴으로써 인덱스의 갱신 성능을 개선하였다. 특히, 갱신 영역을 두어 갱신 지연을 제한함으로써, 검색 지연으로 인한 검색의 정확도 감소를 최소화하였다. 마지막으로, 성능 평가를 통해 UR-Tree에서는 불확실성 영역이 커지고 갱신 영역이 커질수록 검색 성능 및 검색의 정확도는 크게 떨어지지 않으나 갱신 성능이 월등히 좋아짐을 입증하였다.

참고문헌

1. Bonnet, P., Gehrke, J., Seshadri, P., "Towards Sensor Database Systems," Proc. of the 2th Intl. Conf. on Mobile Data Management, 2001, pp.3-14.
2. Cheng, R., Prabhakar, S., "Managing Uncertainty in Sensor Databases," SIGMOD Record, Vol.32 No.4, 2003, pp.41-46.
3. Tubaishat, M., Madria, S., "Sensor Networks: An Overview," IEEE Potential, Vol.22 No.2, 2003, pp.20-23.
4. Wong, J. L., Potkonjak, M., "Search in Sensor Networks: Challenges, Techniques and Applications," Proc. of the Intl. Conf. on Acoustics Speech and Signal, 2002, pp.3752-3755.
5. Deshpande, A., Guestrin, C., Madden, S. R., Hellerstein, J. M., Hong, W., "Model-driven Data Acquisition in Sensor Networks," Proc. of the 30th Intl. Conf. on Very Large Databases, 2004, pp.588-599.
6. Trajcevski, G., Wolfson, O., Hinrichs, K., Chamberlain, S., "Managing Uncertainty in Moving Objects Databases," ACM Transactions on Database Systems, Vol.29 No.3, 2004, pp.463-507.
7. Dai, X., Yiu, M.L., Mamoulis, N., Tao, Y., Vaitis, M., "Probabilistic Spatial Queries on Existentially Uncertain Data," Proc. of the 9th Intl. Symposium on Spatial and Temporal Databases, 2005, pp.400-417.
8. Li, X., Kim, Y. J., Govindan, R., Hong, W., "Multi-Dimensional Range Queries in Sensor Networks," Proc. of the Embedded Networked Sensor Systems, 2003, pp.63-75.
9. Goodchild, M., Buttenfield, B., Wood, J., Visualization in Geographical Information Systems, John Wiley & Sons, 1994.
10. Hill, J., Horton, M., Kling, R., Krishnamurthy, L., "The Platforms Enabling Wireless Sensor Networks," Communications of the ACM, Vol.47 No.6, 2004, pp.41-46.
11. Hosbond, J.H., Saltenis, S., Ørtoft, R., "Indexing Uncertainty of Continuously Moving Objects," Proc. of the DEXA Workshops, 2003, pp.911-915.
12. Civilis, A., Jensen, C.S., IEEE, Pakalnis, S., "Techniques for Efficient Road-Network-Based Tracking of Moving Objects," IEEE Transactions on Knowledge and Data Engineering, Vol.17 No.5, 2005, pp.698-712.
13. Cheng, R., Xia, Y., Prabhakar, S., Shah, R., Vitter, J.S., "Efficient Indexing Methods for Probabilistic Threshold Queries over Uncertain Data," Proc. of the 30th Intl. Conf. on Very Large Databases, 2004, pp.876-887.
14. Yu, X., Mehrotra, S., "Capturing Uncertainty in Spatial Queries over Imprecise Data," Proc. of the 14th Intl. Conf. on Database and Expert Systems Applications, 2003, pp.192-201.
15. Kwon, D.S., Lee, S.J., Lee, S.H., "Indexing the Current Positions of Moving Objects Using the Lazy Update R-Tree," Proc. of the 3th Intl. Conf. on Mobile Data Management, 2002, pp.113-120.
16. Yuni, X., Sunil, P., Shan, L., Reynold, C., Rahul, S., "Indexing Continuously Changing Data with Mean-Variance Tree," Proc. of the Mobile Computing and Applications, 2005, pp.1125-1132.

김동오
2000년 건국대학교 컴퓨터공학과(공학사)
2002년 건국대학교 대학원 컴퓨터공학과(공학석사)
2006년 건국대학교 대학원 컴퓨터공학과(공학박사)
2006년~현재 건국대학교 컴퓨터공학부 강의교수
관심분야: MODBMS, LBS, 스트림 데이타베이스,
유비쿼터스 센서 네트워크, GML

강홍구
2002년 건국대학교 컴퓨터공학과(공학사)
2004년 건국대학교 대학원 컴퓨터공학과(공학석사)
2004년~현재 건국대학교 대학원 컴퓨터공학과
박사과정
관심분야: 공간 데이타베이스, GIS, LBS, USN,
센서 데이타베이스

홍동숙
1999년 건국대학교 컴퓨터공학과(공학사)
2001년 건국대학교 대학원 컴퓨터공학과(공학석사)
2000년~2003년 쌍용정보통신 모바일/GIS 기술팀
2003년~현재 건국대학교 대학원 컴퓨터공학과
박사과정
관심분야: 데이터베이스, 이동체 데이터베이스,
유비쿼터스 컴퓨팅

한기준
1979년 서울대학교 수학교육학과(이학사)
1981년 한국과학기술원(KAIST) 전산학과(공학석사)
1985년 한국과학기술원(KAIST) 전산학과(공학박사)
1990년 Stanford 대학 전산학과 Visiting Scholar
1985년~현재 건국대학교 컴퓨터공학부 교수
2004년~현재 한국정보시스템감리사협회 회장
관심분야: 공간 데이타베이스, GIS, LBS,
텔레매틱스, 정보시스템 감리