HBR-tree : 위치 기반 서비스를 위한 효과적인 현재 위치 인덱싱 기법†

(HBR-tree : An Efficient Current Location Data Indexing Mechanism for Location Based Services)

윤재관, 홍동숙, 한기준
(Jae-Kwan Yun, Dong-Suk Hong, Ki-Joon Han)
건국대학교 컴퓨터공학부
{jkyun, dshong, kjhan}@db.konkuk.ac.kr

초록

최근 PDA와 같은 모바일 장치와 무선 인터넷의 사용이 확대되고, GPS의 개발로 인하여 위치 기반 서비스가 활발히 연구되고 있다. 그러나, 위치 기반 서비스의 중요한 요소인 이동 객체는 이동에 따른 갱신 비용이 높기 때문에 이전의 디스크 기반의 GIS에서 사용되던 인덱스를 이용하는 것은 효과적이지 못하다.

본 논문에서는 위치 기반 서비스를 위한 효과적인 현재 위치 데이타 처리를 위해 공간 해쉬 인덱스와 R-tree 인덱스를 결합한 형태인 HB(Hash Based)R-tree 인덱스를 개발하였다. HBR-tree 인덱스는 위치 기반 서비스에서 이동 객체의 위치 데이타가 빈번하게 갱신된다는 특징을 이용하여 갱신 작업은 HBR-tree 인덱스의 공간 해쉬 테이블 내에서 처리하고, 생성된 공간 해쉬 테이블을 이용하여 R-tree 인덱스를 구성함으로써 빠른 검색 질의 처리가 가능하고 갱신 비용을 줄일 수 있다는 장점이 있다.

1. 서론

최근에 핸드폰, PDA, HPC 등과 같은 모바일 장치가 많이 사용되고 있고, 무선 네트워크의 발전으로 인하여 실생활에서도 무선 네트워크가 많이 보급되어 있다. 또한, 소형의 GPS 장치의 모듈식 개발로 인하여 개인이 사용하고 있는 모바일 장치에도 GPS 장치가 부착되고 있다. 이러한 기술의 발전으로 인하여 이동 통신망을 기반으로 하여 사람이나 사물의 위치를 정확하게 파악하고 이를 활용하는 응용 시스템 및 서비스인 위치 기반 서비스(LBS: Location Based Service)가 점차 부각되고 있다[2,5,6].

일반적으로 위치 기반 서비스에서는 시간이 흐름에 따라 연속적으로 변화하면서 발생되는 이동 객체의 위치 데이타 처리가 중요하다. 즉, 이동 객체의 위치 데이타는 시간의 흐름에 따라 지속적으로 변화하기 때문에 이동 객체의 위치 데이타를 처리하기 위해서는 갱신 연산이 주로 사용된다. 그러므로, 이동 객체의 갱신 연산을 효과적으로 처리하기 위해서는 이동 객체의 특성을 반영한 별도의 현재 위치 데이타 인덱스가 필요하다[3,4].

기존의 이동 객체의 위치 데이타를 처리하기 위한 인덱스 기법으로는 R-tree 인덱스

† 본 연구는 한국과학재단 목적기초연구(과제번호: R01-2001-000-00540-0) 지원으로 수행되었음.

를 확장하는 방법과 1차원의 해쉬 함수를 2차원의 공간으로 확장한 공간 해쉬 인덱스를 사용하는 방법이 있다. 그러나, 공간 해쉬 인덱스는 데이타 집합이 비정규 분포일 경우 지속적인 오버플로우가 발생하고 해쉬 인덱스의 전체 영역을 설정하기 어려운 문제점이 있으며, R-tree 인덱스는 인덱스의 빈번한 변경으로 인하여 성능 저하가 발생하는 단점이 있다[6]. 본 논문에서는 이러한 기존의 위치 데이타 인덱스들의 단점을 극복하는 효율적인 이동 객체의 현재 위치 데이타 인덱스 방법인 HBR-tree 인덱스를 제시하고 성능 평가를 수행하였다.

본 논문의 구성은 다음과 같다. 제 2 장에서는 공간 해쉬와 R-tree 인덱스를 분석하고, 문제 해결 방법을 제시한다. 제 3 장에서는 HBR-tree 인덱스의 구조 및 알고리즘에 대해 언급하고, 제 4 장에서는 HBR-tree 인덱스의 성능 평가 결과를 살펴본다. 마지막으로, 제 5 장에서는 결론 및 향후 연구 과제에 대해 언급한다.

2. 관련 연구

본 장에서는 공간 해쉬 인덱스와 R-tree 인덱스를 분석하고 기존 위치 데이타 인덱스의 문제 해결 방법에 대하여 설명한다.

2.1 공간 해쉬와 R-tree 인덱스 분석

공간 해쉬 인덱스는 이동 객체의 위치를 키 값으로 해쉬하는 방법을 사용하여 비교적 간단한 과정으로 인덱스가 가능하다는 장점이 있다[5]. 그러나, 데이타 집합이 비정규 분포(특정 지역에 밀집)일 경우 특정 영역 셀에 지속적인 오버플로우가 발생하여 인덱스의 성능이 저하되는 문제가 발생한다. 이동 객체는 주기적으로 이동하므로 밀집 지역을 빈번하게 발생시키기 때문에 공간 해쉬 인덱스의 성능 저하가 발생된다.

R-tree 인덱스는 높이 균형 트리 구조이기 때문에 데이타의 분포와 관계없이 일반적으로 검색에 우수한 성능을 나타낸다. 그러나, 이동 객체의 계속되는 위치 이동으로 인해 인덱스의 변경이 발생하고, 인덱스의 빈번한 변경으로 전체적인 인덱스의 성능 저하가 발생한다[1,2]. 이와 같은 문제의 원인은 R-tree 인덱스가 변경이 극히 적은 정적 데이타를 기반으로 설계되었기 때문에 검색에는 효과적이지만 삽입이나 갱신이 빈번한 위치 데이타의 연산에는 적합하지 않은 구조이기 때문이다.

2.2 기존 인덱스의 문제 해결 방법

일반적으로 이동 객체에 대한 현재 위치 데이타 인덱스를 구성할 경우에 이동 객체의 특성에 따라서 그룹을 지을 수 있다. 즉, 지하철을 타고 가는 경우, 버스를 타고 가는 경우, 고속도로를 가는 경우, 인도를 걸어가는 경우 등 대부분의 경우는 동일한 영역을 일정하게 이동하다가 새로운 이동 객체의 추가 및 다른 영역으로 이동하는 이동 객체의 삭제가 일어나게 된다.

그러므로, 그룹에 대한 MBR을 이용하여 공간 해쉬 테이블을 구성하고 이에 대한 MBR 정보를 R-tree 인덱스에 추가하게 되면 기존의 공간 해쉬 인덱스의 단점을 해결할 수 있고, 또한 R-tree 인덱스의 문제점인 인덱스의 빈번한 변경으로 인한 전체적인 인덱스의 성능 저하가 발생하는 문제점도 해결할 수 있다.

3. HBR-tree 인덱스

본 장에서는 이동 객체의 현재 위치 데이타 인덱싱 방법인 HBR-tree 인덱스에 대하여 설명한다.

3.1 HBR-tree 인덱스의 구조

HBR-tree 인덱스는 기존의 R-tree 인덱스의 장점과 공간 해쉬 인덱스의 장점을 동시에 수용하고 있다. HBR-tree 인덱스는 R-tree 인덱스와 공간 해쉬 인덱스에서 확장된 개념으로 그 구조는 그림 1과 같다.

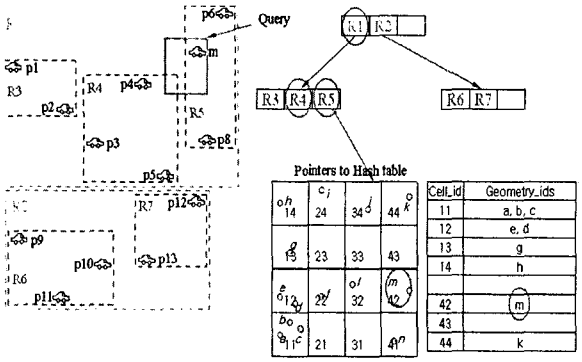

그림 1. HBR-tree 인덱스의 구조

HBR-tree 인덱스에서 영역에 대한 질의가 들어오면 R-tree 인덱스에서 해당하는 MBR을 검색하게 된다. 그림 1에서와 같이 R1의 전체 영역 중 질의에 포함되는 R4와 R5를 검색한 후 공간 해쉬 테이블에서 m을 구하게 된다. 일반적인 위치 데이타의 갱신은 공간 해쉬 테이블에서 일어나고, 이에 대한 전체적인 MBR의 검색은 R-tree 인덱스에서 일어나게 되므로 빈번한 이동 객체의 변경에도 빠른 처리가 가능하게 된다.

3.2 공간 해쉬 테이블의 생성

HBR-tree 인덱스에서는 전체 공간 도메인을 공간 해쉬 함수를 통해 작은 범위로 축소시키게 된다. 예를 들면, 서울의 지도를 가로, 세로의 일정한 영역으로 나누었을 때 각각의 격자는 공간 해쉬 테이블의 하나의 셀(cell)에 매핑될 수 있다. 즉, d-차원을 구성하는 각 i축(단, $1 \le i \le d$)을 m_i 구간으로 나누어 전체 공간 도메인을 $k = m_1 \times m_2 \times \ldots \times m_d$개 셀들의 집합으로 구성되는 공간으로 변환한다. d-차원 공간의 임의의 좌표는 $(v_1, v_2, \ldots, v_d)$로 표현된다. 공간 해쉬 테이블에서는 각 축의 값에 대한 공간 해쉬 함수 값들인 $(f_1(v_1), f_2(v_2), \ldots, f_d(v_d))$을 해당 좌표의 셀 주소(cell address)로서 사용한다. HBR-tree에서 사용하는 i축을 위한 공간 해쉬 함수 $f_i(v_i)$는 다음과 같다 (단, $1 \le i \le d$).

$$f_i(v_i) = \frac{v_i - \min_i}{\left\lceil \frac{\max_i - \min_i + 1}{m_i} \right\rceil}$$

$\max_i$와 $\min_i$는 각각 공간 객체들의 i 축의 값에 대한 최대값과 최소값이다. m_i는 i축을 구성하는 구간의 수이다. 즉, i축은 m_i개로 나뉘어진다. 그러므로, 셀 주소는 $(0,0,\ldots,0) \sim (m_1, m_2, \ldots, m_d)$의 범위에 존재하게 된다. m_i의 값이 Δi $(\max_i - \min_i + 1)$에 근접한다면, 즉 m_i의 값이 커지면 셀의 크기가 작아지기 때문에 이동 객체는 밀집 상태가 되어 오버플로우가 많이 발생하게 된다. m_i의 값이 1에 근접한다면, 즉 m_i의 값이 작아지면 셀의 크기가 커져서 셀에 속하게 되는 이동 객체가 많아지게 되므로 인접한 이동 객체를 찾는 것이 쉽지만 메모리 공간을 많이 차지하게 된다. 그러므로, m_i는 이동 객체의 밀집도에 따라서 결정하는 것이 바람직하다.

3.3 HBR-tree 인덱스의 생성

본 논문에서는 이동 객체가 일종의 그룹을 이룬 형태로 입력이 되는 것을 가정하고 있다. 실세계에서의 이동 객체는 일반적으로 그룹을 이루면서 동일한 궤적으로 이동한다. 즉, "2호선 지하철을 이용하는 시민들"은 지하철이 이동하는 궤적에 따라 위치 데이타가 발생되게 된다. 그러므로, 이러한 이동 객체의 위치 데이타는 위치 획득 단계에서 그룹으로 나누어 HBR-tree 인덱스에 입력된다.

특정 시간에 그림 2(a)와 같은 새로운 위치 데이타 그룹이 삽입이 되면 그림 2(b)와 같이 이 위치 데이타 그룹에 해당하는 공간 해쉬 테이블이 만들어진다. 위치 데이타 그룹에 의해 생성된 공간 해쉬 테이블 h14는

그림 2(c)와 같이 HBR-tree 인덱스에 삽입된다. 삽입된 h14는 R7의 자식 노드에 삽입이 된다. 만약 h14에 소속된 위치 데이타를 검색하고자 한다면 루트 노드에서 R2를 검색하여 R6, R7을 찾고, R6, R7을 검색하여 h14를 찾은 후 공간 해쉬 함수를 이용하여 위치 데이타를 검색하면 된다.

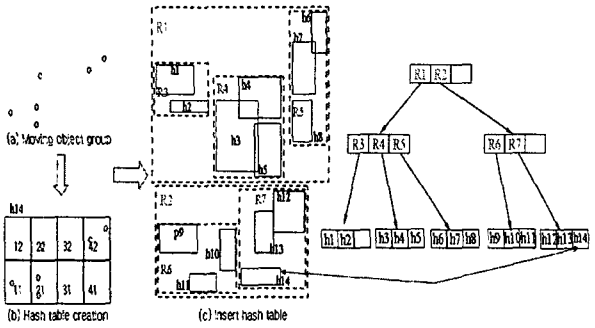

그림 2. 이동 객체의 추가

3.4 HBR-tree 인덱스의 알고리즘

이동 객체의 위치 데이타 획득은 특정 시간의 간격에 따라 연속적으로 발생한다. 즉, 특정 시간에서의 동일한 시간 정보를 가지는 이동 객체들을 획득하여 삽입한다. 그리고 나서, 다시 특정 시간에서 동일한 시간 정보를 가지는 이동 객체들을 획득하여 삽입하는 작업을 반복하게 된다.

본 논문에서는 이동 객체를 삽입할 때 중간 노드의 사각형 영역이 최소로 증가하도록 하여 간접적으로 겹치는 영역을 줄이고, 삽입할 리프 노드에 오버플로우가 발생하였을 때 리프 노드에서 분할이 일어나게 하였다. 새로운 이동 객체를 삽입하고자 할 경우 가장 먼저 삽입되는 객체는 공간 해쉬 테이블에 저장된다.

두 번째 이동 객체를 삽입할 때, 공간 해쉬 함수를 이용하여 이전 이동 객체의 MBR 정보와 동일한 시공간 정보를 가지고 있을 경우 이전에 저장된 객체와 새로 삽입하고자 하는 이동 객체를 동일한 공간 해쉬 테이블에 저장한다. 만약 이동 객체를 삽입할 때 공간 해쉬 함수가 설정한 영역과 다른 영역에 저장되어야 한다면 별도의 공간 해쉬 함수를 이용해 다른 MBR을 구성한다. 그리고 나서, 구성된 MBR을 R-tree 인덱스에 추가한다. 그림 3은 공간 해쉬 함수에서 이동 객체가 삽입될 때의 알고리즘을 보여준다.

```
Algorithm InsertData(p)
Input  p : array of points
Output The new HBR-tree that results after insertion of p
Begin
1.          Calculate MBR(p)
2.          Search R-tree to find predefined spatial hash
table.
3.          If previous hash table is found,
              call InsertIntoSpatialHashTable(p),
            else call MakeNewSpatialHashTable(p)
4.          e = MBR of p
5.          cn  = root
6.          If cn is leaf, stop.
7.          From all entries in cn, choose the one e with
the smallest room.
8.          cn = e.ptr, go to 6.
9.          Insert e into cn. Call SplitAndAdjust (cn).
            SplitAndAdjust(cn)
1.          If cn is overfull, call Split(cn) to produce cn1
and cn2, replace cn's old entry in its parent by e1 =
Union(cn1), e2 = Union(cn2), call SplitAndAdjust on cn's
parent.
2.          Otherwise, if e = Union(cn) is different from
cn's old entry in its parent, replace the old entry with e, call
SplitAndAdjust on cn's parent.
End
```

그림 3. 삽입 알고리즘

HBR-tree 인덱스에서 이동 객체를 검색하기 위한 알고리즘은 루트부터 시작해서 트리의 아래 방향으로 검색하며 질의 영역과 겹치는 중간 노드의 사각형들에 대해 대응하는 자식 노드들을 루트로 하여 재귀적으로 검색하는 방법을 사용한다. 이동 객체에 대한 점 질의 수행 순서 및 과정은 그림 4와 같다.

```
Algorithm SearchData(W)
Input W : search window(MBR)
Output All objects overlapping W
Begin
  1. Start at the root
  2. If current node is non-leaf, for each entry <MBR,
     ptr>, if Consistent(MBR,W) , search subtree
     identified by ptr
  3. If current node is leaf, for each entry <E, hid>, if E
     overlaps W, hid identifies a pointer of spatial hash
     table MBR that overlaps W.
  4. Find matched objects in spatial hash tables.
End
```

그림 4. 검색 알고리즘

HBR-tree 인덱스에서 이동 객체를 삭제하기 위한 알고리즘은 루트부터 시작해서 트리의 아래 방향으로 검색하며 질의 영역과

겹치는 중간 노드의 사각형들에 대해 대응하는 자식 노드들을 루트로 하여 재귀적으로 삭제하는 방법을 사용한다. 이동 객체에 대한 삭제 수행 순서 및 과정은 그림 5와 같다.

```
Algorithm DeleteData(T, e)
Input T : HBR-tree rooted at node T, e : Index entry(id, hRect)
Output The new HBR-tree that results after the deletion of e
Begin
  1. Using the search procedure, find a leaf cn where entry e is located
  2. Remove e from cn. Call SplitAndAdjust(cn).
       SplitAndAdjust(cn)
  1. If cn is underfull, deallocate the node cn remove cn's entry in its parent, call SplitAndAdjust on cn's parent, and reinsert all cn's entries or merge them into some other node
  2. Otherwise, if e = Union(cn) is different from cn's old entry in its parent, replace the old entry with e, call SplitAndAdjust on cn's parent.
End
```

그림 5. 삭제 알고리즘

4. HBR-tree 인덱스의 성능 평가

본 장에서는 다음과 같은 실험적 방법을 사용하여 HBR-tree 인덱스의 성능 평가를 수행하였다. 우선, 경로 탐색 알고리즘을 반복하여 수행함으로써 이동 객체가 출발지와 목적지 사이에서 움직이면서 생성되는 실제 도로에서의 위치 데이타를 추출하였다. 그리고, 경로 탐색 시에 사용된 이동 객체에 ID를 부여하고 ID별 이동 객체의 위치 데이타를 그림 6과 같이 메모리 테이블에 저장하였다. 또한 저장된 메모리 테이블의 데이타를 쓰레드(Thread)를 이용하여 이동 객체 ID별로 시작 지점부터 종료 지점까지 일정 시간 동안 위치 데이타를 지속적으로 발생시켰다.

objid	objx	objy
1	211874.631156317	449626.016665557
1	211874.633238916	449626.019207253
1	211874.633238916	449626.019207253
1	211878.82626292	449621.902215249
1	211896.717206936	449606.024263234
1	211911.86626295	449589.245255219
1	211924.758230962	449577.537223208
1	211924.758230962	449577.537223208

그림 6. 메모리 테이블

그림 7은 이동 객체의 개수에 따라 삽입을 수행한 결과이다.

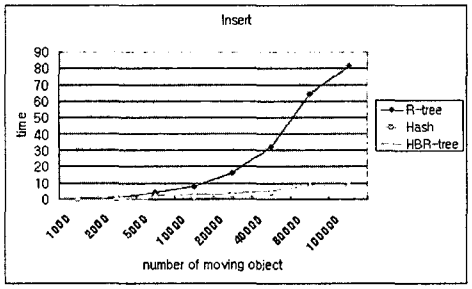

그림 7. 삽입에 대한 성능 평가

HBR-tree 인덱스는 삽입의 처음에는 공간 해쉬 테이블을 이용하여 트리를 구성하기 때문에 시간이 많이 걸리지만 공간 해쉬 테이블을 이용한 트리가 점차적으로 구성이 되고 나면 그 이후에는 갱신을 위한 비용이 적기 때문에 공간 해쉬 인덱스만 사용하였을 경우의 속도에 근접하게 된다.

갱신 연산은 이동 객체의 수가 증가함에 따라 검색을 수행한 후에 그 값을 변경하는 방법을 사용하였다. 갱신에 대한 성능 평가 결과는 그림 8과 같다. 갱신 연산은 이동 객체의 위치 획득이 5분 단위로 발생한다고 가정하고 성능 평가를 수행하였다.

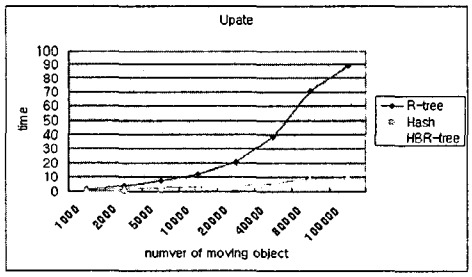

그림 8. 갱신에 대한 성능 평가

HBR-tree 인덱스는 트리를 검색하여 이동 객체의 위치 데이타가 포함된 공간 해쉬 테이블을 찾고 나면 공간 해쉬 테이블에서 검색 연산을 하기 때문에 이동 객체의 위치 데이타가 포함된 공간 해쉬 테이블을 찾는 시간을 제외하고는 공간 해쉬 인덱스와 유사한 성능을 보여주고 있다.

검색은 이동 객체가 1,000~100,000개만큼 삽입된 상태에서 동일한 검색 영역을 설

정한 후 그 영역에 해당하는 이동 객체를 검색하는데 소요되는 시간을 측정하였다. 그림 9는 검색에 대한 성능 평가 결과이다.

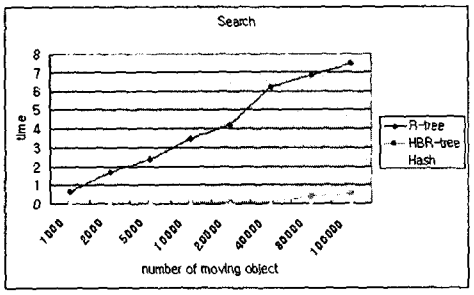

그림 9. 검색에 대한 성능 평가

HBR-tree 인덱스는 이동 객체가 포함되어 있는 공간 해쉬 테이블을 트리에서 찾게 되면 공간 해쉬 함수를 이용하여 이동 객체가 소속되는 셀을 바로 찾을 수 있기 때문에 빠른 검색이 가능하다. 공간 해쉬 인덱스는 공간 해쉬 테이블에서만 검색을 하면 되기 때문에 가장 빠른 성능을 보이고 있다.

제 5 장 결 론

본 논문에서 제안한 HBR-tree 인덱스는 기존의 GIS에서 사용되고 있는 R-tree 인덱스의 특성과 일차원 비공간 데이타 검색에서 사용되던 해쉬 인덱스를 다차원 공간으로 확장시킨 공간 해쉬 인덱스의 특성을 모두 반영하고 있다. 본 논문에서 제시하는 HBR-tree 인덱스의 장점은 다음과 같다.

첫째, 셀 내에서의 위치 데이타의 변경은 공간 해쉬 테이블 내부에서만 일어나므로 전체적인 인덱스의 재구성이 필요 없다. 둘째, 이동 객체의 위치가 이웃한 다른 셀로 이동하더라도 공간 해쉬 기반의 인덱스 구조에서는 삭제와 재삽입 연산 비용이 다른 트리 계열보다 작다.

셋째, 공간 해쉬 테이블을 사용하여 위치 인덱스를 구성할 경우 삽입, 검색, 삭제, 갱신 작업이 신속하게 이루어질 수 있다. 넷째, 빈번한 인덱스 변경은 초기에 이동 객체에 대한 인덱스가 구성될 때만 자주 일어나게 되고, HBR-tree 인덱스가 일단 구성되게 되면 모두 공간 해쉬 테이블을 이용하기 때문에 시간이 지날수록 빠른 인덱스 구성 및 검색이 가능하다.

본 연구의 향후 연구 방향으로는 현재 위치 데이타에 대한 인덱스뿐만이 아니라 HBR-tree 인덱스를 기반으로 하는 과거 위치 인덱스, 미래 위치 인덱스에 대한 연구가 필요하다.

참고문헌

[1] Guttman, A., "R-trees: A Dynamic Index Structure for Spatial Searching," Proc. of ACM SIGMOD Conference, 1984, pp.47-54.

[2] Kwon, D., and Lee, S., "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," Proc. of International Conference on Mobile Data Management, 2002, pp.113-120.

[3] Pfoser, D., Theodoridis, Y., and Jensen, C. S., *Indexing Trajectories in Query Processing for Moving Objects*, Chorochronos Technical Report, 1999.

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

[5] Song, Z., and Roussopoulos, N., "Hashing Moving Object," Proc. of International Conference on Mobile Data Management, 2001, pp.161-172.

[6] Yun, J.K., Kim, D.O., and Han, K.J., "Development of a Real-Time Mobile GIS supporting the Open Location Service," Proc. of Geotec Event Conference, Canada, 2003.