시공간 집계정보를 위한 Aggregation R-tree 기반의 하이브리드 인덱스

(A Hybrid Index based on Aggregation R-tree for Spatio-Temporal Aggregation)

유 병 섭 † 배 해 영 ††
(ByeongSeob You) (HaeYoung Bae)

요 약 교통 관리 시스템과 같은 응용에서는 공간 데이타 웨어하우스의 공간 계층을 이용한 분석을 수행하는데, 이러한 분석에서는 주로 단순한 집계정보만을 요구한다. 공간 계층 기반의 집계정보 제공을 위하여 기존의 연구들은 공간 인덱스를 사용한 해결방법을 제시하였는데, 대부분의 연구들은 공간 인덱스 중 가장 널리 이용되는 R-tree를 확장한 방법을 이용하였다. 그러나 단순히 현재 집계 정보만을 제공하여 수년에 걸친 분석을 요구하는 교통 정책에 대하여 의사결정을 지원할 수 없었다. 본 논문에서는 과거의 집계정보까지 관리할 수 있는 aR-tree(Aggregation R-tree)기반의 하이브리드 인덱스를 제안한다. 제안 기법은 aR-tree를 이용하여 공간 계층과 현재시점의 집계정보를 제공하며, 시간 구조체를 이용한 정렬 해쉬 테이블로 시간 계층과 과거의 집계정보를 제공한다. 따라서 제안기법은 시공간 분석을 통한 효율적인 의사결정을 지원하며, 이는 현재의 교통 분석 및 과거를 통한 교통 정책 결정을 가능하게 한다.

키워드 : 공간 데이타 웨어하우스, 의사결정 지원, 시공간 인덱스

Abstract In applications such as a traffic management system, analysis using a spatial hierarchy of a spatial data warehouse and a simple aggregation is required. Over the past few years, several studies have been made on solution using a spatial index. Many studies have focused on using extended R-tree. But, because it just provides either the current aggregation or the total aggregation, decision support of traffic policy required historical analysis can not be provided. This paper proposes hybrid index based on extended aR-tree for the spatio-temporal aggregation. The proposed method supports a spatial hierarchy and the current aggregation by the R-tree. The sorted hash table using the time structure of the extended aR-tree provides a temporal hierarchy and a historical aggregation. Therefore, the proposed method supports an efficient decision support with spatio-temporal analysis and is possible currently traffic analysis and determination of a traffic policy with historical analysis.

Key words : Spatial Data Warehouse, Decision support, Spatio-Temporal Index

1. 서 론

최근 급변하는 시장 변화에 대응하기 위해 정확하고 신속한 의사 결정이 기업의 중요 요소가 되었다[1]. 따라서 기업은 입지선정, 물류이동경로선택 등의 의사 결정을 하기 위해 경영에 필요한 정보와 공간 정보를 통합하고 있으며, 이로 인해 공간 의사결정 지원 시스템인 공간 데이타 웨어하우스에 관심이 집중되고 있다[2,3]. 특히 교통 관리 시스템에서는 현재의 교통량 분석에 따른 효율적인 차량 분산을 위해 공간 데이타 웨어하우스가 중요한 역할을 수행한다[4]. 고속도로의 현재 교통량에 따라 막히는 구간을 찾아내고 우회도로를 찾아 안내해 주는 서비스는 공간 의사결정의 대표적인 예라고 할 수 있다. 교통 관리를 효율적으로 하기 위해서는 공간 데이타의 계층 분석을 통해 의사결정을 지원하는 공간 데이타 웨어하우스가 매우 중요하며, 공간 의사결정 지원을 위한 분석에서는 차량의 상세한 정보보다는 특정 지역의 시간별 차량의 수와 같이 단순 집계정보만을 요구한다[5,6].

· 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 육성.지원사업의 연구결과로 수행되었음
† 학생회원 : 인하대학교 컴퓨터·정보공학과
subi@dblab.inha.ac.kr
†† 종신회원 : 인하대학교 컴퓨터·정보공학과
hybae@inha.ac.kr
논문접수 : 2006년 3월 3일
심사완료 : 2006년 8월 11일

현재까지 공간 계층을 효율적으로 지원하고 집계정보를 효율적으로 관리하기 위하여 공간 인덱스[7-10]를 이용한 연구가 진행되었다[11]. 대부분의 연구들은 공간 인덱스로 가장 널리 사용하는 R-tree를 이용하였는데, 이는 R-tree의 레벨을 이용하여 공간 계층을 지원하며 노드별 집계정보를 제공하였다. 그 중 aR-tree는 각 엔트리에 집계정보를 갖도록 확장한 인덱스로 R-tree가 관리하는 각 지역에 대한 집계정보를 효율적으로 제공할 수 있다[4]. 그러나 현재의 집계정보만 제공하여 분석을 위한 시간별 집계정보를 위해서는 모든 데이타를 읽어야 하는 문제점이 있었다. OLAP Favored Search는 aR-tree를 개선한 인덱스로 집계정보를 갖는 요약 테이블을 별도로 구축한다[12]. OLAP Favored Search는 R-tree의 구조를 변형하지 않고 요약 테이블에 인덱스 식별자와 집계정보가 하나의 레코드로 존재하는 방법을 이용하였다. 이는 R-tree를 변형하지 않고 집계정보를 제공하는 장점을 갖지만 별도의 요약 테이블에 집계정보를 유지하므로 해당 엔트리의 집계정보를 검색하기 위한 비용이 증가하였다.

따라서 본 논문에서는 시공간 계층을 지원하며, 현재 및 과거의 집계정보를 효율적으로 제공할 수 있는 하이브리드 인덱스를 제안한다. 하이브리드 인덱스란 aR-tree를 기반으로 하여 정렬 해쉬 테이블을 결합한 인덱스이다. 하이브리드 인덱스에서 확장된 aR-tree는 각 엔트리마다 기존 aR-tree에서와 마찬가지로 현재 집계정보를 갖으며, 추가적으로 정렬 해쉬 테이블 포인터 정보를 갖는다. 따라서 현재 집계값을 통하여 현재의 정보를 쉽고 빠르게 분석할 수 있으며, 정렬 해쉬 테이블 포인터를 통하여 시간의 계층구조를 이용한 과거부터 현재까지의 경향 분석을 수행할 수 있다. 정렬 해쉬 테이블은 '연(year)'을 해쉬값으로 갖는 해쉬 테이블로 각 버킷은 '월(month)' 이하의 시간을 구조체 형태로 가지고 있는다. 여기서 시간이라 함은 날짜와 시각을 모두 포함한다. 이때 '월' 이하의 단위는 그 크기가 고정되어 있기 때문에 고정 크기의 구조체를 이용한다. 단, '일(day)'의 경우 각각의 달마다 크기가 다른데, 이는 사용하지 않는 공간을 표시해두는 방법을 이용한다. 각 시간의 단위는 하위 시간 단위의 각각의 집계정보를 모두 집계한 정보를 갖는데, 이는 시간에 대한 빠른 집계 연산을 가능하게 한다. 또한 각 시간 단위의 구조화된 저장되므로 시간에 대한 계층 지원이 가능하다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구에 대하여 설명하고, 3장에서는 본 논문에서 제안하는 aR-tree 기반의 하이브리드 인덱스를 설명한다. 4장에서 제안기법의 질의 처리에 대해 설명하며, 5장에서 성능평가를 하고, 6장에서 결론을 맺는다.

2. 관련 연구

공간 데이타 웨어하우스는 공간 데이타에 대한 의사결정을 지원할 수 있도록 데이타 웨어하우스를 확장한 시스템이다[2]. 데이타 웨어하우스는 의사결정을 지원하기 위해 계층 구조를 이용한 분석을 제공하는데[13,14], 비공간 데이타의 경우 시간 데이타와 같이 의미있고 개념이 확실하게 구분되는 정보를 이용하여 계층을 나눈다[15,16]. 그러나 공간 데이타의 경우 그 자체로는 의미가 없기 때문에 계층 구조를 지원하기 어렵다. 이러한 문제를 해결하기 위하여 공간 데이타 웨어하우스에서는 공간 인덱스를 이용한 계층 구조의 지원을 연구하였다[4]. 공간 인덱스 중에서 R-tree는 널리 사용되는 공간 인덱스로 공간 데이타 웨어하우스에도 이를 이용한 계층 구조 지원이 연구되었다[11,12].

aR-tree는 R-tree를 확장한 인덱스로 각 엔트리마다 집계정보를 추가적으로 저장한다[4]. aR-tree는 R-tree 구조에서 트리 각각의 레벨을 일반화의 정도로 정의하여, 부모 노드가 자식노드들의 일반화된 노드로 정의된다. 따라서, 최상위 노드(root node)를 공간 개념의 가장 일반화된 노드로, 최하위 노드(leaf node)를 가장 상세화된 노드로 하여, 공간 개념에 대한 개념 계층을 aR-tree의 레벨과 매핑시킨다. 이렇게 구성되는 aR-tree는 집계정보 관리를 위하여 각 엔트리마다 집계정보를 갖도록 확장하였다. 즉, 각 엔트리는 자신의 공간 영역에 대하여 현재의 집계정보를 가진다. 이때 부모노드는 자식노드의 일반화된 개념으로 정의되었으므로 부모노드의 집계정보는 자식노드들의 집계정보를 모두 집계한 것과 같게 된다. 예를 들어 SUM 집계를 갖는 aR-tree에서 하나의 부모노드가 각각 2와 3을 갖는 두 자식노드를 가진다면, 부모노드의 집계정보는 2와 3을 더한 5를 갖는다. 이는 집계정보를 요구하는 공간 영역이 부모노드의 공간 영역 전부를 포함하면 자식노드까지 검색하지 않고 바로 집계정보를 제공할 수 있다는 장점을 제공한다. 그러나 aR-tree의 집계정보는 현재의 집계정보만 가지고 있어 시간별 집계정보 제공을 위해서는 저장 데이타를 모두 읽어 분석해야 하는 단점을 갖는다.

OLAP Favored Search는 R-tree를 변형하지 않고 집계연산을 제공하기 위해 제안되었다[12]. 이를 위해 OLAP Favored Search에서는 각 엔트리에 해당하는 집계정보를 저장하기 위한 별도의 요약 테이블을 둔다. 요약 테이블은 각 엔트리의 식별자와 집계값을 필드로 구성하며, R-tree의 각 엔트리에 해당하는 집계정보를 저장한다. OLAP Favored Search는 R-tree를 변형하지 않고 집계연산을 제공할 수 있다는 장점을 가지고 있다. 그러나 집계정보를 얻기 위해 R-tree 연산 후 각

엔트리 식별자를 이용한 요약 테이블 검색 비용이 증가하고, 이는 사용자 응답시간을 지연시킨다. 또한 요약 테이블의 각 집계정보는 aR-tree와 같이 현재의 집계정보를 갖기 때문에 시간별 집계정보 제공을 위해서는 저장 데이타를 모두 읽어 분석해야 하는 단점을 갖는다.

aR-tree와 OLAP Favored Search 현재의 집계정보만을 지원하기 때문에 수년간 변화를 통한 정책 수립 등을 효율적으로 지원할 수 없다. 따라서 현재시간의 분석을 위해서는 빠른 현재 집계정보의 제공이 필요하며, 의사결정 지원을 위한 시간별 경향 분석을 효율적으로 지원하기 위해서는 과거부터 현재까지의 시간별 집계정보를 제공할 수 있는 인덱스가 필요하다.

3. aR-tree 기반의 하이브리드 인덱스

aR-tree 기반의 하이브리드 인덱스는 aR-tree를 확장한 EaR-tree(Extended aggregation R-tree)와 정렬해쉬 테이블 sHT(sorted Hash Table)이 결합된 인덱스이다.

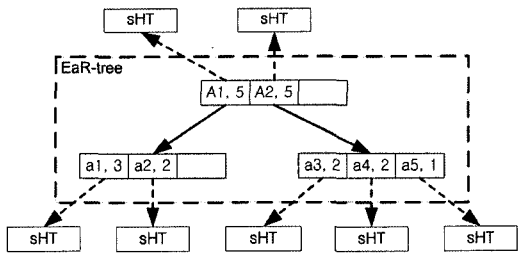


그림 1 하이브리드 인덱스의 전체 구조

그림 1은 aR-tree 기반 하이브리드 인덱스의 전체 구조를 나타낸다. EaR-tree는 aR-tree에 기반하여 공간 계층을 지원하며, sHT는 해쉬 테이블과 구조체에 기반하여 시간 계층을 지원한다. EaR-tree의 각 엔트리는 하나의 sHT를 갖고 sHT는 시간 계층을 관리한다. 본 장에서는 하이브리드 인덱스에 대해 EaR-tree와 sHT로 나누어 설명한다.

3.1 EaR-tree: Extended aggregation R-tree

EaR-tree는 기존의 aR-tree에 시간 계층을 확장시킨 인덱스이다. EaR-tree는 공간 계층을 지원하기 위하여 기존의 aR-tree와 마찬가지로 트리의 레벨을 이용하며, 현재의 집계정보를 관리하기 위하여 aR-tree와 마찬가지로 각 엔트리에 집계정보를 관리한다. 그러나 EaR-tree는 시간 계층을 지원하기 위하여 각각의 엔트리마다 시간 계층 구조의 sHT(sorted Hash Table)를 갖고, 이를 위해 각 엔트리에 sHT의 링크 정보를 추가적으로 관리한다.

EaR-tree는 트리의 구조에서 볼 때 기존 기법과 다르게 관리영역이 고정되어 있다. 또한, EaR-tree는 각각의 객체를 빠르게 검색하기 위한 것이 아니라 특정 영역에 존재하는 객체들의 집계정보를 빠르게 검색하기 위한 것이기 때문에 기존 aR-tree와 달리 단말노드에 각각의 객체정보를 갖지 않는다. 따라서 초기 생성시 최소 크기의 관리영역이 결정되면 그 영역을 단말노드로 하는 R-tree를 생성하게 되며 엔트리가 포함할 수 있는 객체의 개수는 고정되지 않는다. 이는 R-tree의 각 엔트리들 간의 겹치는 영역이 없어지고 노드의 크기 변화가 일어나지 않기 때문에 객체들에 대한 삽입 및 삭제, 갱신, 검색 연산에 대하여 성능향상을 가져온다.

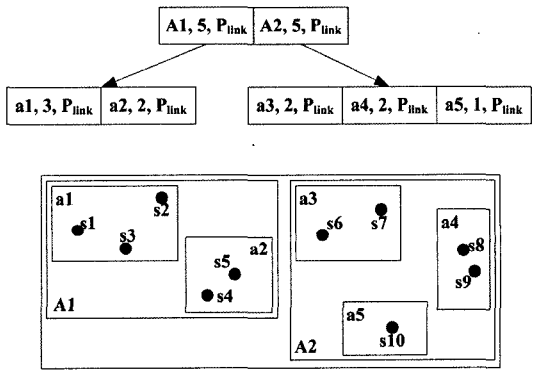


그림 2 Extended aggregation R-tree의 구조

그림 2는 EaR-tree의 구성을 나타낸다. 엔트리의 각 영역은 겹치지 않게 구성되어 있으며, 단말노드는 객체가 아닌 최소 관리영역을 갖고있다. 또한 EaR-tree의 각 엔트리는 aR-tree의 엔트리 정보 이외에 해당 엔트리의 시간 계층 및 집계정보를 갖는 sHT를 가르키는 링크 정보를 추가적으로 갖는다. 각 엔트리는 자신의 자식노드들을 포함하는 최소 크기의 사각형인 MBR (Minimum Bounding Rectangle)을 가지고 있으며, 이는 하위 개념들의 일반화가 상위 개념이 되는 계층 구조에 적합한 형태이다.

EaR-tree의 생성과정은 다음과 같다. 먼저 EaR-tree의 최소 관리 영역 a1, a2, ..., a5을 정하고 이를 기준으로 R-tree를 생성한다. 관리 집계정보가 객체들의 개수를 라고 하면, a1 엔트리의 경우 s1, s2, s3의 3개의 객체가 있으므로 집계값으로 3을 갖는다. 그리고, 현재 엔트리에 대한 sHT에 현재 시각과 집계값을 갖고 삽입을 한다. 같은 방법으로 a2 엔트리의 경우 s4와 s5의 두 객체가 있으므로 집계값으로 2를 갖는다. A1 엔트리의 경우 a1과 a2를 자식노드로 가지므로 a1과 a2의 일반화 엔트리이고 두 지역의 집계값에 대한 합인 5를 집계값

으로 갖는다. 나머지 객체 s6~s10까지 이와 같은 방법으로 EaR-tree를 구성하게 되면, 그림 2의 트리가 된다.

그림 2에서 A1 지역의 엔트리는 a1 지역과 a2 지역의 엔트리를 자식 노드로 가지고 있으며, A1 지역 엔트리의 MBR은 a1 지역 엔트리의 MBR과 a2 지역 엔트리의 MBR을 포함한다. 따라서, a1 지역과 a2 지역을 공간 계층에 의해 일반화 하면 그 두 지역을 포함하는 A1 지역이 결과가 된다. EaR-tree의 특성상 a1 지역에 객체 10개가 추가되어도 a1 지역에 대한 엔트리의 MBR의 크기는 변함이 없고, 집계값만 13으로 변경된다. 이는 새로운 데이타의 삽입 및 삭제 등에 대하여 트리 재구성이 발생하지 않게 되므로 성능이 크게 향상된다.

EaR-tree의 장점은 다음과 같다. 첫째, 각 엔트리의 관리 영역 크기가 고정되어 삽입 및 삭제 연산에 대한 트리 재구성이 나타나지 않는다. 이는 트리의 삽입 및 삭제 연산에서 가장 많은 비용을 차지하는 트리 재구성이 나타나지 않아 연산비용의 많은 감소를 가져온다. 둘째, 현재 상황에 대한 분석을 위한 집계정보를 트리의 각 엔트리마다 가지고 있어 빠른 현재 집계정보 검색이 가능하다. 이는 현재 집계정보를 이용하는 현재의 지역별 상황 분석의 성능을 향상시킨다. 셋째, 공간의 계층 개념을 지원한다. 의사결정 지원을 위한 OLAP(On-Line Analytical Processing) 연산은 데이타의 효율적인 분석을 위하여 데이타에 대한 일반화(Roll up)와 상세화(Drill down)가 필요한데, EaR-tree는 트리의 레벨을 일반화의 단계로 접목하여 일반화 및 상세화를 트리의 레벨 이동으로 쉽게 처리할 수 있다.

3.2 sHT: sorted Hash Table

sHT는 시간의 계층 구조를 지원하기 위한 해쉬 테이블이다. 이는 하나의 해쉬 테이블로 구성되며, 해쉬 테이블의 키값은 시간의 '연'에 해당하는 값을 사용한다. 이는 '연'의 경우 고정되어 있지 않고 계속 증가하기 때문이다. '연' 이하의 단위인 '월', '일', '시', '분', '초' 등은 '일'을 제외하고는 고정된 크기를 가지고 있다. '일'의 경우도 최대의 크기는 정해져 있으므로 고정된 크기의 구조체를 이용하여 관리할 수 있다. 고정된 크기의 구조체는 데이타를 하나의 구조체에서 관리를 하기 때문에 관리가 쉽고, 데이타간의 링크정보를 갖지 않아 공간 효율성이 좋은 장점이 있다.

그림 3은 sHT의 구조를 나타내고 있다. 해쉬 함수(Hash Function)는 '연'에 해당하는 값을 이용하여 해쉬값을 만든다. 이는 시간의 가장 큰 단위를 '연'으로 하였기 때문이다. 해쉬 함수를 통한 해쉬값은 1~0까지 모두 10개가 존재하며 순서대로 나열되어 있다. 각 해쉬값에는 해당하는 버켓들이 링크드 리스트로 저장되어 있으며, 각 버켓들은 연도순으로 다음 연도의 버켓을 가르키는 포인터를 갖는다. 다음 연도의 버켓을 가르키는 포인터는 "1993년에서 2000년까지"와 같이 연속된 시간에 대한 분석을 요구하는 질의를 빠르게 처리할 수 있다. sHT의 버켓은 키값인 '연'의 값과 하위레벨 시간인 '월', '일' 등의 시간에 대한 계층 구조 및 집계정보를 관리하는 타임 구조체(T_{st}: Time Structure), '연'의 집계값을 갖는 V_{tot} 등을 갖는다. 타임 구조체를 자세히 살펴보면 그림 4와 같다.

그림 4는 '시'단위까지 시간 정보를 관리하는 타임 구조체의 모습을 나타낸 것이다. 그림 4에서 '월' 단위의 크기는 12로 고정되어 있다. 1년은 12달로 '월'의 크기는 변하지 않으므로 고정된 크기의 사용이 가능하다. '시'에 대한 크기도 '월'에 대한 크기와 마찬가지로 고정되어

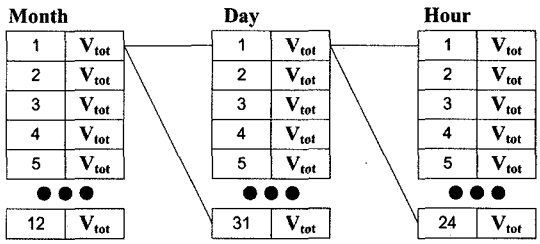


그림 4 타임 구조체의 구조

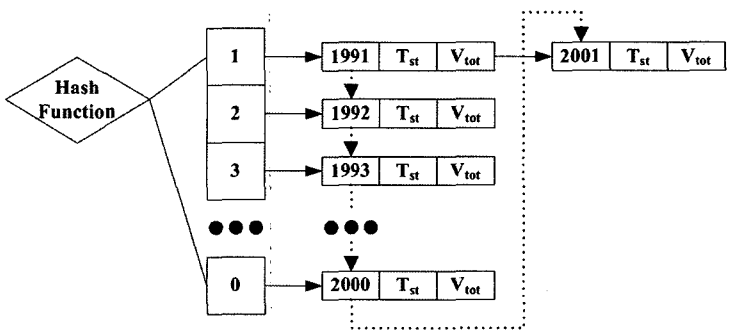


그림 3 sorted Hash Table의 구조

있다. '일'의 경우 각 달에 대하여 다른 크기를 갖는다. 그러나 모든 달은 최대 31을 넘을 수 없으므로 최대 크기인 31로 고정된 크기를 갖는다. 이때 '일'의 크기를 판단하기 위하여 '사용불가'라는 값을 미리 정의하고 이를 해당 '월'의 크기를 넘는 값에 해당하는 V_{tot}에 저장한다. 따라서 검색시 '일'에 해당하는 크기를 쉽게 판별할 수 있다.

'월'의 일반화 단위는 '연'으로 이는 sHT에서 버킷의 V_{tot}값이 된다. 예를 들어 1991년의 '월'단위 분석을 하였을 경우 일반화를 요청 받으면, 해당 '월'이 존재하는 버켓의 상위 개념인 '연' 단위 1991년의 V_{tot}값을 결과로 한다.. 이러한 시간 계층 개념의 이해를 돕기 위하여 sHT 버켓의 개념도를 그리면 그림 5와 같다.

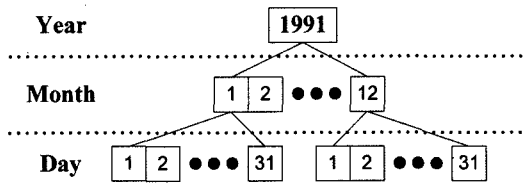


그림 5 sHT 버켓의 개념도

그림 5는 '일' 단위까지의 개념도를 그린것으로 하나의 노드가 하나의 엔트리를 갖는 형태이다. 제일 상위 루트 노드에 '연'에 해당하는 부분이 존재하고, 그 자식으로는 '월'에 해당하는 노드들을 갖는다. 마찬가지로 '월'에 해당하는 노드는 '일'에 해당하는 노드들을 자식 노드로 갖는다. 이때 각 '월'의 노드들에 대해 '일'에 해당하는 노드들의 개수는 31개로 고정된다. 만일 최하위 관리 시간 단위를 '시' 또는 '분'으로 한다면 위와 같은 개념으로 확장이 가능하다.

이렇게 구성된 sHT는 다음과 같은 장점을 갖는다. 첫째, 시간에 대한 계층을 지원한다. 그림 5의 개념도에서 알 수 있듯이 각 시간 단위의 계층 관계가 명확하게 구분되며, 이를 통하여 일반화 및 상세화 연산을 쉽고 빠르게 할 수 있다. 둘째, '월'이하의 단위에 대해서는 고정된 크기의 구조체를 이용하여 관리한다. 이는 '월' 이하의 단위에 대해서는 '일'을 제외하고는 그 크기가 고정되어 있으므로 가능하고, '일' 단위의 경우에도 최대 크기로 크기를 고정할 수 있다. 따라서 크기 변형에 따른 연산 시간의 증가를 막는다. 셋째, 해쉬테이블을 이용하여 빠른 연산이 가능하다. '연' 단위의 경우 연도를 키 값으로 해쉬 함수를 이용하여 해쉬 값을 제공하고 해쉬값들을 정렬된 상태로 제공하기 때문에 시간의 연속성을 보장하고 빠른 검색이 가능하다.

4. 하이브리드 인덱스에서의 질의 처리

본 장에서는 하이브리드 인덱스에서 질의 처리에 대하여 검색, 삽입, 삭제, 갱신 연산으로 나누어 설명한다.

4.1 검색연산

검색연산은 크게 현재 집계정보 검색과 과거 집계정보 검색으로 나누어진다. 사용자는 임의의 영역을 선택하여 해당 영역의 현재 집계정보나 과거 집계정보를 요청하게 된다. 현재 집계정보의 경우 EaR-tree의 각 엔트리에서 관리되는 집계정보를 이용하여 결과를 전송하며 과거 집계정보의 경우 EaR-tree를 통하여 해당 엔트리들을 찾고 엔트리들의 sHT 집계정보를 이용하여 결과를 전송한다.

사용자가 임의의 영역에 대한 현재 집계값을 요청할 경우 EaR-tree의 루트노드에서부터 검색 영역에 완전히 포함되는 엔트리나 부분적으로 포함되는 엔트리들을 찾는다. 만약 완전히 포함되는 엔트리일 경우 해당 엔트리의 집계정보를 가져가며, 더 이상 자식노드의 검색을 하지 않는다. 이는 해당 영역에 존재하는 자식 노드들의 집계정보에 대한 집계가 부모노드에 존재하기 때문에 가능하다. 만약 부분적으로 포함되는 엔트리일 경우 해당 엔트리의 자식 노드들에 대해서 위의 과정을 수행한다. 이때 현재 노드가 단말노드로 더 이상의 자식노드가 존재하지 않는다면 검색 영역과 현재 엔트리의 영역의 겹치는 비율을 계산하여 집계정보를 비율만큼 결과로 한다. 제안기법은 단말노드에 실제 객체 정보를 갖고 있지 않기 때문에 정확한 결과값이 아닌 근사값을 제공한다. 그러나 공간 데이타웨어하우스에서 분석을 제공하기 위하여 근사값을 사용하는 것이 허용되기 때문에 정확성이 문제되지 않는다.

사용자가 임의의 영역에 대한 과거 집계값을 요청할 경우 현재 집계정보 검색과 같은 방법으로 EaR-tree를 통하여 해당 엔트리를 찾는다. 해당 엔트리에 대해서 검색을 원하는 시간의 '연' 단위의 값을 sHT의 해쉬함수에 넣어 해쉬값을 찾은 후 해당 버켓을 찾는다. 만약 원하는 시간이 '연' 단위까지라면 버켓의 '연' 단위 집계정보를 결과로 하고, 만약 '연' 이하의 단위까지 원한다면 해당 버켓의 각 하위 단위에 대하여 해당 값의 집계정보를 결과로 한다.

알고리즘 1과 알고리즘 2는 검색연산에 대한 알고리즘을 보여준다.

알고리즘 1 하이브리드 인덱스의 검색연산 알고리즘

```
Search(N, p, T, U)
Input:  node N, search predicate p, temporal range T
        time unit that wants get U
Output: all slots of both time and aggregated value
        that satisfies p
SlotList ret = null
```

```
FOR Each entry E on N
BEGIN
  IF p.MBR contains E.MBR THEN
    // start time is same to end time
    IF T is current THEN
      add time and the aggregated value to ret
    ELSE
      add HashSearch(H, T, U, 1) to ret
      // 1 is intersected rate
  IF p.MBR is intersected with E.MBR THEN
    IF N is not a leaf node THEN
      // invoke Search on the subtree
      // whose root node is referenced by E.ptr
      add Search(E.ptr, p, T, U) to ret
    ELSE      // N is a leaf node
      calculate intersected rate R of p.MBR and E.MBR
      IF T is current THEN
        calculate multiple value M of the aggre-
            gated value and R
        add time and M to ret
      ELSE
        add HashSearch(H, T, U, R) to ret
END
RETURN ret
```

알고리즘 1에서 EaR-tree의 루트노드 N과 검색을 위해 검색 영역을 갖는 p, 검색시간 범위 T, 검색을 원하는 최소 시간단위 U를 가지고 Search 함수가 호출된다. Search 함수는 해당 노드의 각 엔트리들에 대하여 포함관계를 판단한다. 만약 엔트리의 MBR이 p의 MBR에 완전히 포함된다면 현재 집계정보를 요구하는지 판별한다. 만약 현재 집계정보를 요구한다면 현재 엔트리의 집계정보를 결과리스트에 저장한다. 그렇지 않고 과거 집계정보를 요구한다면 HashSearch 함수를 호출한다. 엔트리의 포함관계에서 만약 엔트리의 MBR이 p의 MBR과 부분적으로 겹친다면 현재 노드가 단말노드인지 판단한다. 만약 단말노드가 아니라면 해당 엔트리의 자식노드를 갖고 Search 함수를 다시 호출한다. 만약 단말노드라면 엔트리의 MBR과 p의 MBR의 겹치는 비율 R을 구한다. 다음으로 사용자가 검색을 원하는 시간이 현재라면 해당 집계정보에 R을 곱한 값을 결과리스트에 저장하고 과거라면 HashSearch 함수를 호출한다.

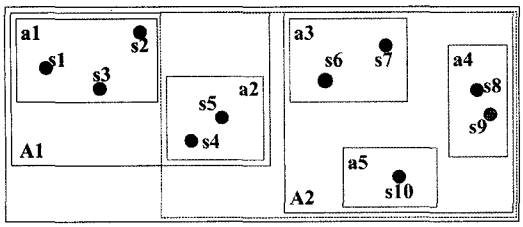
알고리즘 2 sHT의 검색연산 알고리즘

```
HashSearch(H, T, U, R)
Input:  sorted hash table H, temporal range T
        time unit that wants get U, intersected rate R
Output: all slots of both time and the aggregated value
        that is contained T
SlotList ret = null
get bucket B whose year value equals it of T's start time
WHILE B.year <= year value of T's end time
BEGIN
  FOR value V of each time unit on T
  BEGIN
    IF (V >= value of time unit of T's start time) and
       (V <= value of time unit of T's end time) THEN
      calculate multiple value M of the aggre-
          gated value and R
      add time and M to ret
  END
  B ← bucket whose year value is next year of B
END
RETURN ret
```

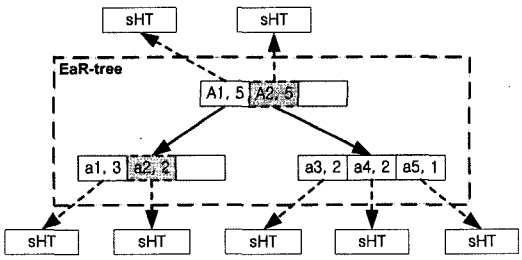
알고리즘 2에서 HashSearch 함수는 sHT의 H와 검색 시간 범위 T, 검색을 원하는 최소 시간 단위 U, 겹치는 영역의 비율 R을 가지고 호출된다. HashSearch 함수는 먼저 H에서 T의 시작시간 중 '연' 단위에 해당하는 값을 갖는 버켓을 찾아 버켓 B에 넣는다. 버켓 B를 찾으면 T의 끝시간 중 '연' 단위에 해당하는 값을 갖는 버켓까지 루프를 돌며 연산을 수행한다. 이때 각 버켓은 '연' 단위의 값에 대해 바로 다음 값에 해당하는 버켓을 가르키는 포인터를 가지고 있어 바로 엑세스가 가능하다. 루프를 도는 동안 각 시간 단위에 대해서 검색 시간 범위 T 안에 존재하는 것에 대해 집계값을 R과 곱하고 결과리스트에 저장한다. '연' 단위 이하에 대하여 위의 과정이 끝나면 다시 다음 '연'에 해당하는 버켓을 가져와 위의 과정을 수행한다. 예를 들어 검색 시간 범위가 2005년 1월 20일~2005년 2월 10일까지라고 가정하자. 먼저 2005년 1월 20일에 해당하는 집계정보를 찾고 이후 2005년 2월 10일까지 '일' 단위의 값을 하나씩 증가하면서 찾아 결과 리스트에 저장한다.

그림 6은 하이브리드 인덱스에서 검색영역이 주어지고 현재 집계정보를 요청할 경우 검색연산을 수행하는 모습을 보여준다.

그림 6의 (a)에서 A1 지역은 검색영역과 부분적으로 겹치며, A2 지역은 검색영역에 완전히 포함된다. 또한, a2와 a3, a4, a5 지역은 모두 검색영역에 포함된다. 이러한 검색영역에 대하여 검색연산이 들어오면 하이브리드 인덱스에서는 검색영역에 대하여 먼저 루트노드에 연산을 수행한다. A1 엔트리의 경우 부분만 겹치므로 하위노드를 검색하며, A1의 하위노드에서는 a1 엔트리는 검색영역에 포함되지 않으므로 검색에서 배제되고 a2 엔트리는 검색영역에 완전히 포함되므로 집계정보인 2를 반환한다. 루트노드의 A2 엔트리의 경우 관리영역이 검색영역에 완전히 포함되므로 집계정보인 5를 반환하고, 더 이상 하위노드를 탐색하지 않는다. 따라서 전체 검색연산의 결과는 7을 반환하게 된다.



(a) 하이브리드 인덱스 관리 영역에서 검색



(b) 현재 집계정보 검색연산에 대한 검색 노드

그림 6 하이브리드 인덱스의 검색연산

4.2 삽입연산

삽입연산은 하이브리드 인덱스의 관리영역에 새로운 객체가 들어오는 연산을 말한다. 객체가 삽입되면 하이브리드 인덱스는 먼저 루트노드를 읽어 해당 객체가 포함되는 지역의 엔트리를 찾는다. 해당 엔트리를 찾으면 해당 엔트리의 집계정보를 변경하고, 해당 엔트리의 sHT에 대하여 삽입을 수행한다. 이때 sHT에 삽입하기 위하여 삽입 시간과 객체 정보를 함께 넘겨준다. sHT는 '연'에 해당하는 값을 해쉬함수에 넣어 해쉬값을 계산하고, 계산된 해쉬값을 이용하여 해당 버켓을 찾는다. 만약 버켓이 존재하지 않는다면 새로운 버켓을 하나 만들어 초기화 한 후 '연'에 해당하는 집계정보를 변경하고, 마찬가지로 '월', '일' 등 하위 시간에 대한 집계정보를 변경한다. 만약 버켓이 존재한다면, 해당 버켓에 대하여 위와 같은 과정을 수행한다. 하이브리드 인덱스의 루트 노드에 대하여 연산이 끝나면 해당 엔트리의 자식 노드에 대하여 위와 같은 과정을 반복한다. 만약 해당 엔트리의 자식 노드가 없다면 삽입연산의 수행이 종료된다.

알고리즘 3과 알고리즘 4는 하이브리드 인덱스의 삽입연산에 대한 알고리즘이다.

알고리즘 3에서 객체 O의 삽입연산을 위해 Insert 함수를 호출한다. Insert 함수는 루트노드부터 단말노드까지 각 노드마다 호출되며, 각 노드의 엔트리들 관리영역에 대해 객체를 포함하는지를 판별한다. 객체를 포함하는 엔트리를 찾으면 해당 엔트리의 집계정보를 변경하고 해당 엔트리의 sHT에 삽입연산을 수행하기 위하여

알고리즘 3 하이브리드 인덱스 삽입연산 알고리즘

```
Insert(N, O, T)
Input:  node N, object O, time T
Output: void
FOR each entry E on N
BEGIN
    IF O.MBR is contained E.MBR THEN
        update aggregation of E
        // invoke InsertHash on the hash table of E
        call InsertHash(E.hash, O, T)
        IF N is not a leaf THEN
            // invoke Insert on the subtree
            // whose root node is referenced by E.ptr
            call Insert(E.ptr, O, T)
END
RETURN
```

InsertHash를 호출한다. sHT에 삽입연산을 마치면 현재 노드가 단말노드인지 판별하고, 만약 단말노드라면 수행을 마치고, 그렇지 않으면 해당 엔트리의 자식 노드를 가지고 Insert 함수를 호출한다.

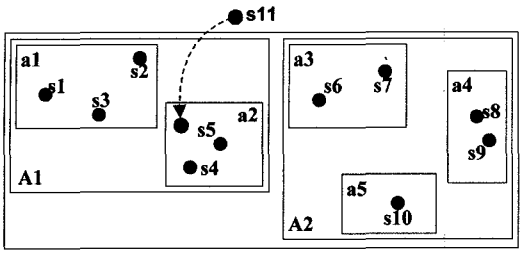
알고리즘 4 sHT의 삽입연산 알고리즘

```
InsertHash(H, O, T)
Input:  sorted hash table H, object O, time T
Output: void
calculate hash value V of year unit of T
FOR each bucket B whose hash value is V on H
BEGIN
    IF B.year equals T.year THEN
        update aggregation of B
        FOR value A of each time unit on T
        BEGIN
            get A'th slot S of current time unit
            update aggregation of S
        END
END
IF not find bucket THEN
create new bucket N
initialize N
InsertHash(H, O, T)
RETURN
```

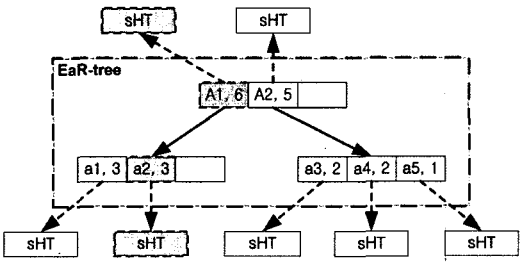
알고리즘 4에서 sHT에 객체 O를 삽입하기 위하여 InsertHash 함수를 호출한다. InsertHash 함수는 '연' 단위의 시간 값을 이용하여 해쉬 값을 계산하고, 이를 이용하여 해당 버켓 리스트를 가져온다. 버켓들 중에서 '연' 단위의 시간 값과 일치하는 버켓을 찾고, 해당 버켓의 '연' 단위 집계정보를 변경한 뒤 하위 시간 단위에 대한 집계정보를 변경한다. 만약 버켓을 찾지 못하였을 경우 아직 해당 '연'의 버켓이 존재하지 않는 경우로 새

로운 버켓을 생성하고 초기화를 한 후 InsertHash함수를 다시 호출한다.

그림 7은 하이브리드 인덱스에 새로운 객체가 삽입되는 삽입연산에 대해 나타내고 있다.



(a) 하이브리드 인덱스 관리 영역에 새로운 객체 삽입



(b) 삽입연산 수행 후의 하이브리드 인덱스

그림 7 하이브리드 인덱스의 삽입연산

그림 7의 (a)는 하이브리드 인덱스의 관리 영역에 새로운 객체 s11이 삽입되는 것을 보여준다. s11이 삽입되는 위치는 a2영역이다. s11이 하이브리드 인덱스에 실제 삽입되는 과정은 다음과 같다. 먼저 A1 지역과 A2 지역을 엔트리로 갖는 루트노드를 읽어 해당 객체가 포함되는 지역을 찾는다. A1 지역이 해당 객체를 포함하므로 A1 지역에 대한 엔트리의 집계값을 1 증가 시킨다. 따라서 집계값은 6으로 변경된다. 이후 삽입된 시간과 객체의 수를 가지고 A1 지역의 sHT에 새로운 데이타를 삽입한다. 만약 삽입된 시간이 "2005년 12월 1일"이라고 한다면 '연'에 해당하는 '2005'를 해쉬 함수에 넣어 해쉬값 5를 계산한다. 해쉬값 5에 해당하는 버켓들 중에서 2005에 해당하는 버켓을 찾고, 만약 없다면 새로운 버켓을 하나 만들고 초기화 한다. 초기화 이후에 '연'에 해당하는 집계정보에 객체 수인 1을 증가시킨다. 만약 '2005'에 해당하는 버켓이 있었다면 해당 버켓의 '연'에 해당하는 집계정보에 객체 수인 1을 증가시킨다. 이후 '월', '일' 등에 대하여 해당 집계정보에 각각 1을 증가시킨다. A1 지역에 대하여 처리가 끝나면 아직 단말노드가 아니므로 A1 지역에 대한 엔트리의 자식 노드에 대하여 연산을 수행한다. A1 지역의 자식 노드에 a1 지역과 a2 지역이 존재하고 그 중 a2 지역에 s11 객체가 삽입되므로 a2 지역의 엔트리에 대해 위와 같은 연산을 수행한다. 따라서 객체 s11의 삽입연산이 수행된 후 제안기법의 인덱스 구조는 그림 6의 (b)와 같이 변경된다. 점선으로 표시된 사각형은 객체 s11의 삽입에 의해 변경된 부분을 나타낸다.

4.3 삭제연산

데이타 웨어하우스의 특성상 과거에 삽입된 객체에 대한 삭제연산은 존재하지 않는다. 즉, 현재 시점에서 객체가 삭제되더라도 과거 시점에서의 객체는 과거 시점에 존재하는 것이다. 하이브리드 인덱스는 현재 뿐만 아니라 과거에 대한 정보를 모두 관리하기 때문에 현재 관리부분에서만 삭제 연산이 수행된다. 따라서 현재 집계정보를 관리하는 EaR-tree의 경우 삭제연산을 수행하고 현재 시간에 대하여 sHT에 변경된 집계정보의 삽입연산을 수행한다.

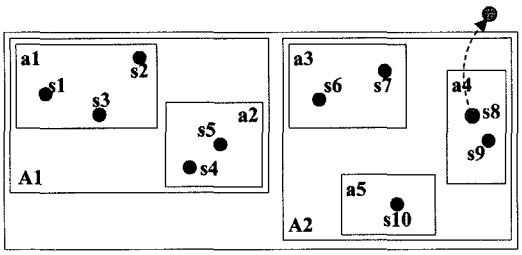
EaR-tree의 삭제연산은 삽입연산의 과정과 거의 동일하다. 먼저 루트 노드에 대하여 삭제하려는 객체가 포함되는 엔트리를 찾고 해당 엔트리의 집계정보를 변경한다. 그리고 sHT에 변경된 집계정보를 현재시간으로 삽입한다. 이후 해당 엔트리의 자식 노드에 대해서 위와 같은 과정을 반복한다. 현재의 노드가 단말노드라면 삭제연산은 종료된다.

알고리즘 5는 하이브리드 인덱스의 삭제연산에 대한 알고리즘이다.

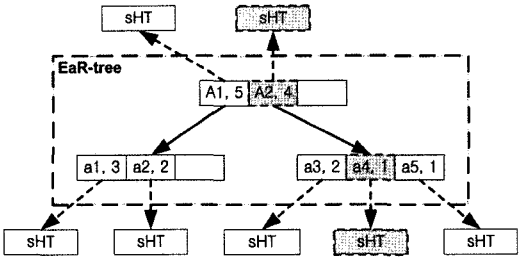
알고리즘 5 하이브리드 인덱스의 삭제연산 알고리즘

```
Delete(N, O)
Input:  node N, object O
Output: void
FOR each entry E on N
BEGIN
    IF O.MBR is contained E.MBR THEN
        update aggregation of E
call InsertHash(E.hash, O, current time T)
        IF N is not a leaf THEN
            // invoke Insert on the subtree
            // whose root node is referenced by E.ptr
            call Delete(E.ptr, O)
END
RETURN
```

알고리즘 5에서 객체 O의 삭제연산을 위해 Delete함수가 루트노드를 갖고 호출된다. 해당 노드의 각 엔트리에 대해서 객체 O가 포함되는 엔트리를 찾고 해당 엔트리의 집계정보를 변경한다. 그다음 변경된 집계정보를 현재 엔트리의 sHT에 현재시간과 함께 삽입한다. 만약 현재 노드가 단말노드라면 삭제연산을 종료하고, 그렇지

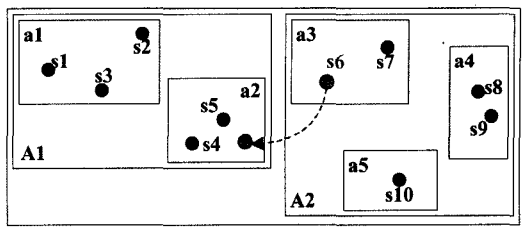


(a) 하이브리드 인덱스 관리 영역에서 객체 삭제

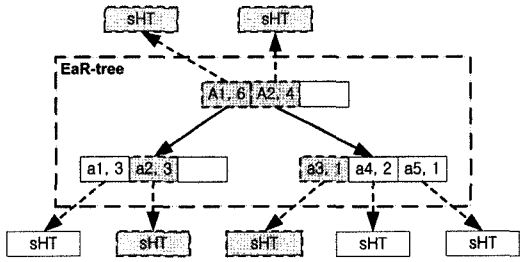


(b) 삭제연산 수행 후의 인덱스의 삭제연산

그림 8 하이브리드 인덱스의 삭제연산



(a) 하이브리드 인덱스 관리 영역에서 객체 갱신



(b) 갱신연산 수행 후의 하이브리드 인덱스

그림 9 하이브리드 인덱스의 갱신연산

않다면 자식 노드를 갖고 Delete함수를 호출한다.

그림 8은 하이브리드 인덱스에서 객체가 삭제되는 연산에 대하여 나타내고 있다.

그림 8의 (a)에서 객체 s8이 하이브리드 인덱스 관리 영역 밖으로 이동하여 삭제연산이 일어나는 것을 보여준다. 이때 루트노드에서 s8이 존재하는 A2 지역의 엔트리를 찾고 해당 엔트리의 집계정보를 1 감소시키고, 현재 집계정보를 sHT에 현재시간과 함께 삽입한다. 해당 엔트리의 자식 노드는 a3, a4, a5 지역의 엔트리들을 가지고 있으며, 객체 s8은 a4 지역의 엔트리에 해당하기 때문에 a4 지역 엔트리의 집계정보를 1 감소시키고, 현재 집계정보를 sHT에 현재시간과 함께 삽입한다. 이때 현재 노드가 단말노드이므로 삭제연산은 종료하게 된다. 따라서 삭제연산 이후의 제안기법의 인덱스 구조는 그림 7의 (b)와 같으며, 점선으로 된 사각형은 삭제연산에 의해 변경된 부분을 나타낸다.

4.4 갱신연산

갱신연산은 하이브리드 인덱스의 관리영역에 존재하는 객체가 관리영역 내의 다른 영역으로 이동하는 것을 의미하며, 갱신하려는 객체에 대하여 갱신 이전 위치에 대한 삭제연산을 수행한 후 갱신 이후 위치에 대한 삽입연산을 수행하면 된다. 따라서 갱신연산은 앞의 두 절의 삭제연산과 삽입연산을 이용한다.

그림 9는 하이브리드 인덱스의 갱신연산에 대하여 나타내고 있다.

그림 9의 (a)에서 보면 객체 s6이 a3 지역에서 a2 지역으로 이동하는 것을 알 수 있다. 이때 먼저 a3 지역에서 객체 s6를 삭제하는 연산을 먼저 수행한다. 따라서 A2 지역 엔트리의 집계정보와 a3 지역 엔트리의 집계정보가 1씩 감소하며, 각각의 sHT에는 현재 집계정보를 현재시간과 함께 삽입한다. 삭제연산이 수행된 다음 객체 s6을 a2 지역에 삽입하는 연산이 수행된다. 이는 A1 지역의 엔트리와 a2 지역 엔트리의 집계정보를 1씩 증가 시키며, 두 엔트리가 갖는 sHT의 삽입연산을 수행하게 된다. 따라서 갱신연산 이후의 제안기법의 인덱스 구조는 그림 8의 (b)와 같으며, 점선으로 된 사각형은 갱신연산에 의해 변경된 부분을 나타낸다.

5. 성능평가

본 장에서는 aR-tree와 OLAP Favored Search에 대해 제안기법과 성능평가를 한다. 테스트 환경은 다음과 같다. 먼저 시스템은 펜티엄 4 CPU 3.0GHz, 1GB의 메모리, 80GB의 하드를 사용하였다. 테스트 데이타는 서울 지역을 공간 데이타로 하였으며, 이 지역에 50,000대의 자동차가 계속적으로 존재하며 50,000대의 자동차는 테스트 영역으로 삽입되는 데이타이며, 다시 50,000대의 자동차가 테스트 영역에서 삭제되는 데이타이다. 따라서 테스트 영역에는 평균 50,000여대의 자동차가 존재한다. 테스트 데이타의 시간 영역은 2003년부터 현재까지의 데이타이며 과거 데이타는 데이타 생성기를 통하여 임의로 데이타를 삽입하여 놓았다.

실험을 위하여 기존 기법과 제안 기법의 프로토타입을 만들었으며, 프로토타입을 이용한 시뮬레이션으로 각 테스트의 응답시간을 비교한다.

5.1 삽입연산의 성능평가

삽입연산의 성능평가를 위하여 50,000개의 데이타를 10~100개씩 동시에 삽입하고, 이때 1개의 데이타가 삽입되는데 걸리는 평균 응답시간을 측정한다. 그림 10은 삽입연산의 성능평가 결과를 보여준다.

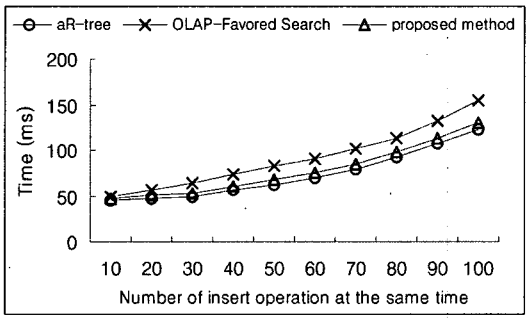


그림 10 삽입연산의 성능평가

그림 10에서 보면 aR-tree의 성능이 가장 좋은 것을 알 수 있다. 이는 확장된 R-tree에 대해서만 연산을 하면 되기 때문이다. 제안기법의 경우 각 엔트리에 대해 sHT의 연산을 수행해야 하므로 aR-tree보다는 성능이 약간 감소한다. 그러나 OLAP Favored Search의 경우 R-tree의 삽입위치를 찾으면 해당 엔트리에 대한 요약 테이블의 레코드를 찾아야 하므로 성능이 가장 나쁘게 나타났다.

5.2 삭제연산의 성능평가

삭제 연산의 경우 100,000 개의 테스트 데이타가 존재하는 가운데 50,000 개의 데이타가 10~100개씩 동시에 삭제될 때 1개의 데이타가 삭제될 때 걸리는 평균 응답시간을 측정한다. 그림 11은 삭제연산의 성능평가 결과를 보여준다.

그림 11에서 보면 제안기법과 aR-tree의 경우 성능이 거의 비슷하다. 이는 두 기법의 삭제 연산이 거의 동일한 방법으로 이루어지기 때문이다. 제안기법의 경우 sHT에 삭제 후 현재시간의 집계정보를 삽입해야 하므로 aR-tree보다는 약간 성능이 저하된다. 그러나 OLAP Favored Search의 경우 삽입연산과 마찬가지로 엔트리를 찾은 이후 요약테이블에서 해당 엔트리의 레코드를 찾아 반영해야 하므로 다른 두 기법에 비해 연산 시간이 오래 걸리는 것을 볼 수 있다.

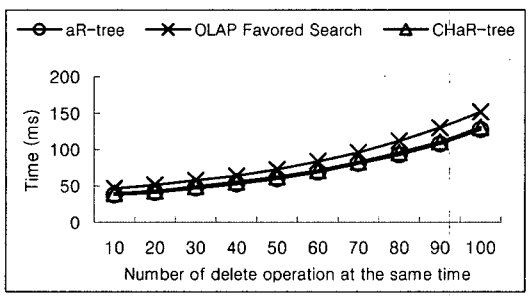


그림 11 삭제연산의 성능평가

5.3 갱신연산의 성능평가

갱신 연산의 경우 100,000 개의 테스트 데이타가 존재하는 가운데 50,000 개의 데이타가 10~100개씩 동시에 갱신될 때 1개의 데이타가 갱신될 때 걸리는 평균 응답시간을 측정한다. 그림 12는 갱신연산의 성능평가 결과를 보여준다.

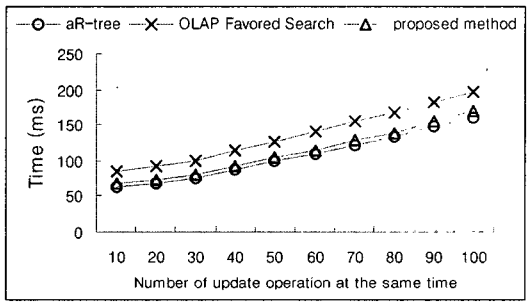


그림 12 갱신연산의 성능평가

그림 12를 보면 삽입연산과 비슷한 결과가 나타난 것을 볼 수 있다. aR-tree의 경우 다른 두 기법에 비해 가장 좋은 성능을 보였으며, OLAP Favored Search의 경우 다른 두 기법에 비해 가장 나쁜 성능을 보였다. 이는 aR-tree는 R-tree에 대해서만 연산하면 되지만 OLAP Favored Search는 해당 엔트리에 대하여 요약 테이블의 정보를 찾아 갱신해야 하기 때문이다. 제안기법의 경우 sHT의 삽입연산이 필요하기 때문에 aR-tree보다는 약간 느린 성능을 보였다.

5.4 검색연산의 성능평가

검색 연산에서는 현재 집계 질의에 대한 성능비교를 하였다. 과거 집계 질의의 경우 제안기법 이외의 두 기법은 수행이 불가능하므로 성능평가를 할 수 없다. 성능비교 방법은 임의 영역에 대한 집계 검색 질의를 보내 연산 결과를 가져오는데 걸리는 응답시간을 비교한다. 이때 100,000 대의 차량이 모든 영역에 분포되어 있고, 영역의 범위를 전체 영역의 크기에 0.001%에서부터 20%까지 측정한다. 같은 크기의 질의영역에 대하여 다른 위치를 검색하는 질의 100개를 수행하기 위한 평균 응답시간을 측정한다. 그림 13은 현재 집계 질의에 대한

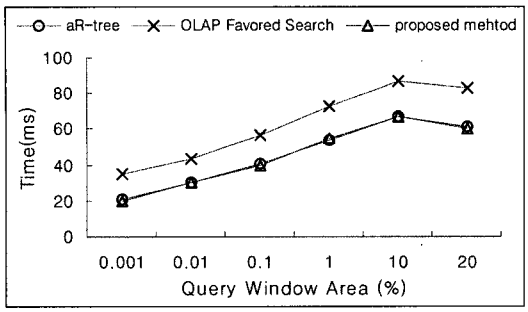


그림 13 현재 집계 질의에 대한 검색연산의 성능평가

검색연산의 성능평가 결과를 보여준다.

그림 13에서 보면 aR-tree와 제안기법의 성능이 거의 같은 것을 볼 수 있다. 이는 현재 집계 질의의 경우 R-tree 내에서 모두 해결할 수 있기 때문이다. 그러나 OLAP Favored Search의 경우 해당 엔트리에 대하여 요약 테이블을 검색해야 하므로 다른 두 기법 보다 응답시간이 길어진다. 모든 기법에 대하여 질의영역의 크기가 약 10%를 넘어가면 응답시간이 줄어드는데, 이는 영역이 넓어질수록 상위노드가 질의영역에 포함되는 경우가 많아져 트리의 검색 횟수가 줄어들기 때문이다. 만약 질의영역이 100%가 되면 루트노드만을 검색하면 결과를 제공할 수 있다.

검색 연산의 경우 현재 집계 질의 이외의 과거 시간별 집계정보를 얻는 질의에 대해서는 5.5절의 저장공간 성능평가와 함께 다룬다.

5.5 저장공간의 성능평가

본 절에서는 aR-tree 및 OLAP Favored Search, 제안기법에 대하여 인덱스를 유지하기 위해 필요한 저장공간의 크기에 대하여 평가한다. 과거의 시간별 데이타를 관리하기 위한 저장공간의 크기를 측정하기 위하여 각 인덱스의 관리지역은 서울지역으로 같은 크기의 R-tree를 갖도록 하였고 각 인덱스마다 저장된 데이타의 수는 고정하였으며, 분석을 위한 데이타의 축적 시간을 1년부터 10년까지 증가함에 따라 인덱스의 크기를 측정하였다. 이때 제안기법에서 과거 시간 계층에 대한 상세화 정도를 '시' 단위까지 하였다.

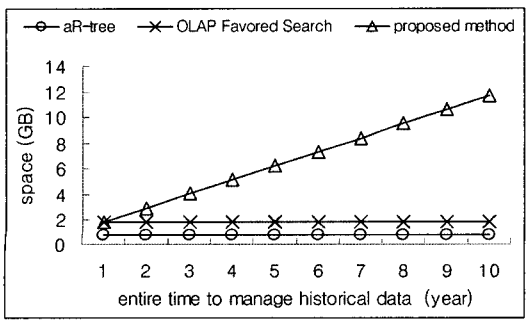


그림 14 축적 시간에 따른 저장공간에 대한 성능평가

그림 14에서 보면 aR-tree와 OLAP Favored Search는 저장공간의 크기가 데이타의 축적 시간과 관계없이 일정하다. 이는 두 인덱스 모두 과거의 시간에 대한 집계정보를 전혀 지원하지 않기 때문에 축적 시간과는 관계없이 일정한 크기를 갖게 된다. 제안기법의 경우 데이타의 축적 시간이 증가함에 따라 저장공간의 크기가 증가하는 것을 볼 수 있다. 이는 과거의 시점별 분석을 지원하기 위한 축적 데이타를 관리하기 위한 크기의 증가이다. 그러나 1년 단위로 데이타의 증가 크기를 보면 약 1GB씩 증가되는 것을 볼 수 있는데, 이는 현재의 저장장치의 크기를 고려하였을 때 저장공간의 오버헤드를 가져오지 않는다는 것을 알 수 있다.

그림 14의 결과는 단지 저장공간만의 크기를 비교하였다. 그러나 시공간 질의에서는 의사결정 지원을 위하여 과거부터 현재까지의 시간별 집계정보를 필요로 한다. 따라서 인덱스를 이용한 시간별 집계정보 검색의 질의응답시간의 평가가 필요하다. 따라서 실제 인덱스의 효용성을 체크하기 위하여 축적 시간에 따른 과거 시간별 질의 처리 시간을 평가하였다. 평가를 하기 위하여 검색 영역의 크기는 고정하고 축적 시간을 1년부터 10년까지 증가함에 따라 각 축적 시간의 10%에 해당하는 '시'단위의 연속된 시간의 집계정보를 가져오는 질의응답시간을 측정하였다. 이때 aR-tree와 OLAP Favored Search는 과거 시간별 집계정보를 지원하지 못한다. aR-tree의 경우 최하위 노드에 실제 객체의 정보를 갖지 않기 때문에 저장데이타의 순차스캔을 이용한 방법을 사용하였고, OLAP Favored Search는 최하위 노드에 실제 객체 정보를 갖기 때문에 트리를 이용한 영역평가 후 선택된 후보 데이타들에 대해서 주어진 조건을 만족하는 데이타의 집계정보를 계산하도록 하였다.

그림 15에서 보면 aR-tree의 응답시간이 가장 느리다. 이는 aR-tree가 최하위 노드로 실제 객체의 정보를

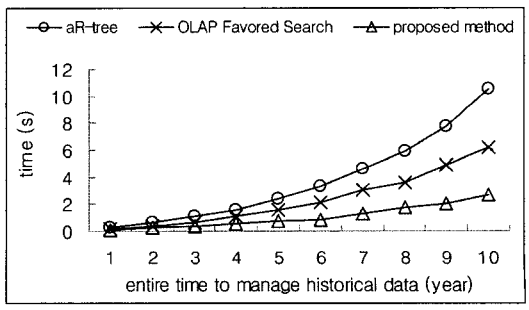


그림 15 축적 시간에 따른 과거 시간별 검색질의에 대한 성능평가

갖지 않기 때문에 저장된 모든 데이타에 대하여 순차검색을 통한 집계정보를 계산하여야 하기 때문이다. OLAP Favored Search의 경우 aR-tree보다 성능이 좋지만 제안기법보다는 성능이 저하된다. 이는 OLAP Favored Search가 최하위 노드에 실제 객체의 정보를 갖기 때문에 영역에 대한 후보 객체들을 쉽게 찾아낼 수 있으나 과거의 시간별 집계정보를 구하기 위해서는 후보 객체들에 대한 저장 데이타를 읽고 이를 집계하여야 하기 때문이다. 제안기법의 경우 해당 영역을 EaR-tree에서 찾고 선택된 엔트리들에 대하여 sHT 링크를 따라 저장된 과거 집계정보를 가져오면 되므로 기존의 두 기법보다 성능이 향상된 것을 볼 수 있다.

6. 결론 및 향후연구

본 논문에서는 aR-tree 기반의 하이브리드 인덱스에 대하여 제안하였다. 제안기법은 공간에 대한 계층과 현재 집계정보를 제공하기 위하여 aR-tree를 확장한 EaR-tree를 이용하였고, 시간에 대한 계층과 과거 집계정보를 제공하기 위하여 해쉬 테이블을 변형한 sHT를 이용하였다. EaR-tree는 aR-tree의 레벨에 따른 공간 계층 개념을 지원하였고, 각 엔트리는 각자가 관리하는 영역에 대한 현재 집계정보를 갖는다. 또한 각 엔트리는 해당 영역에 대한 시간적 집계정보를 관리하기 위해 독립적인 sHT를 갖고 있다. EaR-tree에서 관리되는 영역은 초기 생성시 고정되어 있어 삽입 및 삭제에 대한 트리의 재구성 비용이 감소하였으며, 트리의 레벨을 이용한 공간 계층은 공간 데이타에 대한 일반화 개념을 쉽게 지원할 수 있다. 또한 인덱스에 집계정보를 가져 집계정보를 이용하는 분석에 대하여 처리시간을 감소시켰다. sHT는 '연' 단위의 시간에 대하여 연도를 키 값으로 해쉬값을 생성하고 시간에 따른 집계정보를 관리하는 것으로 해쉬값이 순서대로 정렬되도록 구성된 해쉬 테이블이다. 각 '연에 해당하는 버켓들은 시간에 대한 계층을 지원하는 구조체를 가지고 있으며, 구조체의 크기는 고정되어 있다. 시간의 최소 관리단위는 초기에 결정이 가능하며, 각 시간에 따른 집계정보를 관리한다. 정렬된 버켓들에 의해 시간의 흐름에 따른 경향 분석 비용을 감소시키며, 고정된 크기의 구조체 사용으로 크기 변형에 따른 연산비용의 증가를 막았다. 또한, 시간에 대한 계층 개념을 지원하여 일반화 개념을 쉽게 지원할 수 있다.

제안기법은 공간과 비공간의 계층 개념을 동시에 제공하며, 시간의 흐름에 따른 경향 분석을 제공할 수 있다. 성능평가를 통하여 삽입과 삭제, 수정의 갱신연산에 대하여 우수함을 보였으며, 검색 연산에 대하여 우수한 성능과 기존 기법들이 제공할 수 없는 경향 분석 등을 제공함을 보였다.

향후연구로는 최하위 노드에 대한 검색영역이 부분적으로 일치할 경우에 대해 좀더 정확한 집계정보를 추출하는 방법의 연구가 필요하다.

참 고 문 헌

[1] Eric Sperley, The Enterprise Data Warehouse: Planning, Building, and Implementation, Prentice Hall PTR, pp. 8-15, 1999.

[2] ESRI, "Spatial Data Warehousing for Hospital Organizations," An ESRI White Paper, 1998. http://www.esri.com/library/whitepapers/pdfs/sdwho.pdf.

[3] N. Stefanovic, J. Han, and K. Koperski, "Object-Based Selective Materialization for Efficient Implementation of Spatial Data Cubes," IEEE Transactions on Knowledge and Data Engineering, vol. 12(6), pp. 1-21, 2000.

[4] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient OLAP Operations in Spatial Data Warehouses," Technical Report: HKUST-CS01-01, University of Science & Technology, Hon Kong, 2001.

[5] I.S. Mumick, D. Quass, and B.S. Mumick. "Maintenance of data cubes and summary tables in a warehouse," In Proc. of ACM SIGMOD, Vol.26, No.2, pp. 100-111, 1997.

[6] Y. Tao and D. Papadias, "Range Aggregate Processing in Spatial Databases," IEEE Transactions on Knowledge and Data Engineering, vol. 16(12), pp. 1555-1570, 2004.

[7] A. Guttman, "R-trees: a dynamic index structure for spatial searching," ACM SIGMOD, Vol.14, No.2, pp. 47-57, 1984.

[8] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," ACM SIGMOD, vol. 19, No.2, pp. 322-331 1990.

[9] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-Tree: A dynamic index for multi-dimensional objects," VLDB, 1987.

[10] R. A. Finkel, and J. L. Bentley, "Quad trees: A data structure for retrieval on composite keys," Acta Informatica, Vol.4, pp. 1-9 1974.

[11] L. Zhang, Y. Li, F. Rao, X. Yu, and Y. Chen, "An approach to enabling spatial OLAP by aggregating on spatial hierarchy," In Proc. Data Warehousing and Knowledge Discovery, pp. 35-44, 2003.

[12] F. Rao, L. Zhang, X. L. Yu, Y. Li, and Y. Chen, "Spatial Hierarchy and OLAP-Favored Search in Spatial Data Warehouse," DOLAP, pp. 48-55, 2003.

[13] J. Han, N. Stefanovic, and K. Koperski, "Selective Materialization: An Efficient Method for Spatial Data Cube Construction," In Research and Deve-

lopment in Knowledge Discovery and Data Mining, pp. 144-158, 1998.
[14] W. H. Inmon, Building the Data Warehouse, 3rd Ed., John Wiley & Sons. Inc, 1996.
[15] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: a Relational Aggregation Operator Generalizing Group-by," In Proc. 12th ICDE, pp. 152-159, 1996.
[16] V. Harinarayan, A. Rajaraman, and J. Ullman, "Implementing Data Cubes Efficiently," ACM SIGMOD, Vol.25, No.2, pp. 2005-216, 1996.

유 병 섭

2002년 인하대학교 컴퓨터공학부(공학사). 2004년 인하대학교 컴퓨터공학부(공학석사). 2004년~현재 인하대학교 대학원 컴퓨터정보공학과(박사과정). 관심분야는 공간데이타베이스, 공간 데이타 웨어하우스, Data Stream, 유비쿼터스 컴퓨팅

배 해 영

1974년 인하대학교 공학사(응용물리학) 1978년 연세대학교 공학석사(전자계산학) 1990년 숭실대학교 공학박사(전자계산학) 1992년~1994년 인하대학교 전자계산소 소장. 1982년~현재 인하대학교 컴퓨터공학부 교수. 1999년~현재 지능형GIS연구센터 센터장. 2000년~현재 중국 중경우전대학교 대학원 명예교수. 2004년~2006년 인하대학교 정보통신대학원 원장. 2006년~현재 인하대학교 대학원 원장. 관심분야는 분산 데이타베이스, 공간 데이타베이스, 지리정보 시스템, 멀티미디어 데이타베이스 등