Journal of Korea Multimedia Society Vol. 9, No. 11, November 2006(pp. 1395-1409)

다차원 개념 계층을 지원하는 공간 데이터 큐브의 점진적 일괄 갱신 기법

옥근형[†], 이동욱[††], 유병섭[†††], 이재동[††††], 배해영[†††††]

요 약

공간 데이터 웨어하우스에서는 OLAP(On-Line Analytical Processing) 연산을 제공하기 위해 다차원 데이터를 공간 데이터 큐브의 형태로 관리한다. 개념 계층을 지원하는 공간 데이터 큐브의 크기는 삽입되는 데이터에 비해 방대하기 때문에 구축된 큐브의 구조를 최대한 유지하면서 새로 삽입되는 데이터를 반영시킬 수 있는 점진적 갱신 기법이 연구되어 왔다. 하지만 접두 및 접미의 중복을 제거하여 데이터를 압축 저장하는 큐브에서는 병합된 경로 간의 충돌로 인해 큐브 갱신 시 갱신 내용과 상관없는 셀까지 동시에 갱신되어 갱신이상 현상이 발생한다. 본 논문에서는 공간 데이터 큐브의 점진적 일괄 갱신 기법을 제안한다. 제안 기법은 갱신에 필요한 노드 복사본을 관리하는 자료 구조 및 재귀 탐색을 이용하여, 경로 간의 충돌이 발생할 경우 해당 노드의 복사본을 생성한 후 이를 갱신함으로써 갱신이상 현상을 방지한다. 이를 통해 다차원 개념 계층이 포함된 공간 데이터 큐브를 효율적으로 갱신할 수 있다. 성능 평가를 통해 기존 갱신 기법에 비해 제안 기법의 갱신 속도가 향상되었음을 보인다.

Incremental Batch Update of Spatial Data Cube with Multi-dimensional Concept Hierarchies

Geun-Hyoung Ok[†], Dong-Wook Lee[††], Byeong-Seob You[†††], Jae-Dong Lee[††††], Hae-Young Bae[†††††]

ABSTRACT

A spatial data warehouse has spatial data cube composed of multi-dimensional data for efficient OLAP(On-Line Analytical Processing) operations. A spatial data cube supporting concept hierarchies holds huge amount of data so that many researches have studied a incremental update method for minimum modification of a spatial data cube. The Cube, however, compressed by eliminating prefix and suffix redundancy has coalescing paths that cause update inconsistencies for some updates can affect the aggregate value of coalesced cell that has no relationship with the update. In this paper, we propose incremental batch update method of a spatial data cube. The proposed method uses duplicated nodes and extended node structure to avoid update inconsistencies. If any collision is detected during update procedure, the shared node is duplicated and the duplicate is updated. As a result, compressed spatial data cube that includes concept hierarchies can be updated incrementally with no inconsistency. In performance evaluation, we show the proposed method is more efficient than other naive update methods.

Key words: Spatial Data Warehouse(공간 데이터 웨어하우스), Spatial Cube Index(공간 큐브 색인), Concept Hierarchy(개념 계층), Incremental Update(점진적 갱신)

※ 교신저자(Corresponding Author) : 배해영, 주소 : 인천광역시 남구 용현동 253(402-751), 전화 : 032)860-8719, FAX : 032)862-9845, E-mail : ghsy0718@dblab.inha.ac.kr
접수일 : 2006년 8월 8일, 완료일 : 2006년 9월 27일
[†] 인하대학교 대학원 컴퓨터정보공학과(석사과정)
(E-mail : ghsy0718@dblab.inha.ac.kr)
[††] 인하대학교 컴퓨터정보공학과 박사과정
(E-mail : dwlee@dblab.inha.ac.kr)
[†††] 인하대학교 대학원 컴퓨터정보공학과(박사과정)
(E-mail : subi@dblab.inha.ac.kr)
[††††] 정회원, 단국대학교 정보컴퓨터학부 컴퓨터과학교수
(E-mail : letsdoit@dku.edu)
[†††††] 정회원, 인하대학교 컴퓨터공학부 교수
※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 육성.지원사업의 연구결과로 수행되었음.

1. 서 론

공간 데이터 웨어하우스는 이질적인 데이터 소스로부터 추출된 공간 및 비 공간 데이터를 하나의 저장소에 통합하여 데이터 분석가에게 OLAP 연산을 제공하는 정보시스템이다[1,2]. 공간 데이터 웨어하우스에 통합 저장되어있는 다차원 데이터는 효율적인 집계 연산을 위해 공간 데이터 큐브의 형태로 관리된다.[3,4] 대부분의 OLAP 연산은 여러 차원의 개념 계층상의 다양한 단계들을 통해 큐브에 저장된 데이터를 자세히 또는 간략히 집계하는 것이기 때문에 개념 계층을 지원하는 데이터 큐브에 대한 연구가 활발히 진행되어왔다[5-8]. 특히, 공간 및 비 공간 차원에 대한 개념 계층을 지원하는 공간 데이터 큐브가 제안되는데, 이는 공간 및 비 공간 차원의 개념 계층 구조를 트리 형태로 구성한 후, 각 차원의 계층 트리를 공간 차원의 지역성을 기준으로 연결하여 개념 계층을 내부에 포함하는 공간 데이터 큐브로서 효율적인 저장비용과 개념 계층 집계연산 성능을 보였다[9]. 하지만 새로운 데이터가 삽입되어 큐브 집계 값에 변화가 일어날 경우에 대한 업데이트 기법을 고려하지 않았다.

큐브를 업데이트하는 가장 기본적인 방법으로 큐브 재구축을 통해 새로운 레코드를 반영하는 기법이 있다. 이는 기존 레코드에 삭제 또는 수정이 있고 새로운 레코드도 삽입되는 등 테이블의 변경 사항이 복잡하거나 알 수 없을 경우에 사용되는 기법이다. 하지만 공간 데이터 큐브의 경우 부피가 크고 개념 계층을 지원하므로 구조가 복잡하기 때문에 재구축에 많은 시간이 필요하며, 일반적으로 기존 데이터의 수정이나 삭제 없이 주로 새로운 레코드가 삽입되는 데이터 웨어하우스 환경에 적합하지 않다.

추출된 소스 데이터에 대한 ETL(Extract Transformation Load) 처리와 임시 저장을 담당하는 ODS(Operational Data Store)에서 공간 데이터 웨어하우스에 새로 삽입될 레코드에 대한 인덱스를 만들어 관리하다가 이를 대량 삽입하는 기법에 관한 연구가 있었다[10]. 본 논문에서 다루고자하는 공간 데이터 큐브는 접두 및 접미의 중복을 제거하는 압축 기법 사용으로 인해 하위 노드에서 나타나는 경로 병합문제 때문에 대량 삽입기법을 그대로 적용하는 것이 불가능하다. 그리고 최근 연구 및 제안되는 데이터 큐브들을 살펴보면, 실시간 처리가 요구되는 데이터 웨어하우스 환경의 추세에 따라 이를 점진적으로 갱신하는 알고리즘을 기본적으로 제공하고 있다[8,11,12].

본 논문은 공간 데이터 웨어하우스에서 개념 계층을 지원하는 공간 데이터 큐브에 대한 점진적 일괄 갱신 기법을 제안한다. 이는 큐브 갱신에 참여하는 노드의 복사본을 관리하는 자료 구조와 큐브에 대한 재귀 탐색기법을 이용하여, 경로 간의 충돌이 발생할 경우 해당 노드의 복사본을 생성한 후 이를 갱신함으로써 갱신이상 현상을 방지하는 알고리즘에 대해 서술한다. 이를 통해, 각 차원에 대한 개념 계층을 트리로 구성한 공간 데이터 큐브의 원본 사실 테이블에 ODS에서 구축 및 관리되던 서브 큐브가 적재되면, 이를 점진적으로 큐브에 반영하여 공간 데이터 큐브의 적시성을 보장할 수 있다.

본 논문의 구성은 다음과 같다. 2장에서는 관련 연구에 대해 살펴보고, 3장에서 개념 계층을 지원하는 공간 데이터 큐브에 대한 점진적 업데이트 시 나타나는 문제점과 이를 해결하기 위한 알고리즘에 대해 서술한다. 4장에서는 제안 기법의 성능평가 결과를 분석하고, 5장에서 결론을 맺고 향후 연구 과제에 대해 논한다.

2. 관련연구

본 장에서는 데이터 웨어하우스에서 데이터 큐브를 효율적으로 저장 및 관리하기 위한 Hierarchical Dwarf와 QC-tree의 점진적 갱신 기법에 대하여 기술한다[7,12]. 이들은 모두 일반 데이터 큐브에 대한 압축 기법과 점진적 갱신을 제공하는 큐브이다. 특히, Dwarf의 경우 개념 계층을 지원하기 위해 확장된 Hierarchical Dwarf가 제안되었으나 갱신 기법은 Dwarf의 것을 그대로 사용하였다[11].

2.1 Hierarchical Dwarf의 점진적 갱신 기법

Hierarchical Dwarf는 개념 계층을 지원하는 데이터 큐브로서 접두와 접미의 중복을 제거하고 개념 계층을 큐브 내부에 포함하여 효율적인 저장 비용을 가지고 있으며 개념 계층을 이용한 질의를 빠르게 처리할 수 있는 기능을 제공한다. 하지만 접미의 중복을 제거하기 위해 병합한 경로 간의 충돌로 인해 데이터 업데이트 시 갱신 이상 현상이 발생하여, 이

를 고려한 점진적 갱신 기법을 제공하였다.

새로 삽입 또는 삭제되는 데이터가 발생할 경우 각 레코드의 차원 키 값을 기 구축된 큐브와 비교하면서 해당되는 셀을 찾아 이에 대한 집계 값을 하나씩 갱신하는 기법이다. 이때, 경로 간의 충돌이 감지되면 충돌이 일어난 지점에 있는 노드의 복사본을 만들어 복사본에 갱신을 수행하여 갱신 이상 현상을 방지하였다. 이는 점질의 처리 시 사용되는 알고리즘을 거의 그대로 사용할 수 있어 구현이 간편하고 새로 갱신될 데이터의 양이 적을 경우 효율적이지만 추출된 데이터가 ODS로부터 주기적으로 다량 적재되는 데이터 웨어하우스 환경에서는 갱신 비용이 커질 수 있다. 또한, 큐브를 압축하여 저장하기 때문에 갱신되는 셀의 개수가 어느 정도 줄어드는 효과가 있지만 일반적으로 차원의 수가 d인 데이터 큐브의 경우 사실 테이블에서 레코드 하나의 변화로 인해 갱신되는 큐브의 셀은 2^d개로 차원의 개수가 증가함에 따라 기하급수적으로 증가한다.

2.2 QC-tree의 점진적 일괄 갱신 기법

데이터 큐브를 압축하기 위해 의미가 같은 셀들의 집합인 여러 클래스로 나누어 이에 대한 정보만을 저장하는 Quotient Cube가 제안되었다[13]. 이는 같은 집계 값을 갖고 큐브 내에서 부모-자식 관계를 갖는 셀들을 분류한 것으로 의미론적 OLAP 연산을 제공하는 밀집된 큐브 구조이다. QC-tree는 이를 실제로 구현하기 위해 각 클래스의 상위경계 간에 공통되는 접두를 사용해 트리형태로 고안된 구조이다.

QC-tree의 갱신기법은 여러 개의 레코드가 삽입되는 경우와 삭제되는 경우에 따라 각각 다르게 처리한다. 삽입의 경우, QC-tree 구축 시 사용되는 클래스 추출 알고리즘을 새로 삽입되는 데이터에 적용하여 전처리를 거친 후 이에 대한 결과를 이용해 새로운 데이터를 일괄적으로 QC-tree에 삽입한다. 여기서 사용되는 클래스 추출 알고리즘은 입력된 테이블에서 나타나는 의미론적 클래스들의 상위경계 리스트를 구하는 기능을 한다. 새로 삽입되는 데이터에 대해 클래스 추출 알고리즘을 통해 얻은 상위경계들을 정렬하여 기 구축된 QC-tree와 비교하면서 하나씩 순서대로 삽입하는 형태를 취한다. 기존에 없던 새로운 클래스일 경우 다른 처리 없이 QC-tree 내에 그대로 삽입이 되고, 새로 삽입되는 클래스에 기존 클래스가 포함되는 경우 해당 클래스 전체를 갱신한다. 또한, 기존 클래스에 포함될 경우 해당 클래스에서 삽입될 클래스에 해당되는 셀들만 갱신하고 갱신되지 않은 셀들과 분리하여 클래스를 분할한다. 사실 테이블에서 데이터 삭제를 QC-tree에 갱신하기 위해서 각 클래스에 대해 레코드 카운터를 만들어 커버되는 레코드의 개수를 유지한다. 또한, 삭제 시에도 삭제되는 레코드들에 대해 클래스 추출 알고리즘을 사용하며 정렬 순서는 삽입 시와 반대이다. 추출된 각 클래스를 순서대로 QC-tree에 적용한다. 이때, 삭제로 인해 카운터가 0이 되는 클래스는 삭제하고, 그렇지 않을 경우 필요하면 클래스를 병합한다.

QC-tree의 갱신 기법은 갱신될 데이터에 대한 전처리 과정을 통해 각 레코드마다 하나씩 갱신하는 방법에 비해 실제 갱신에 사용되는 시간의 복잡도를 현저하게 낮추었다. 하지만 QC-tree는 데이터를 의미론적으로 압축하는 Quotient Cube를 위한 구조로서 차원의 개념 계층에 대한 고려가 미비하였으며 새로운 클래스를 갱신하기 위해 매번 QC-tree에 대해 해당 클래스를 찾는 점 질의를 해야 하는 단점이 있다.

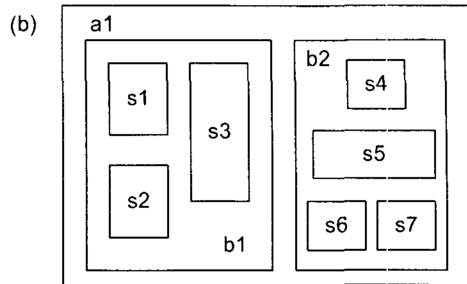
2.3 개념 계층을 지원하는 공간 데이터 큐브

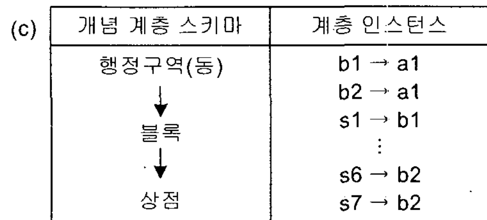
본 논문에서 갱신기법에 대해 다루고자하는 공간 데이터 큐브는 Hierarchical Dwarf와 같이 큐브를 압축하고 개념 계층을 큐브 구조에 포함시켜 효율적인 저장 비용과 개념 계층 질의에 대한 빠른 처리 능력을 가지고 있다. 하지만 Hierarchical Dwarf가 공간 차원에 대한 집계를 지원하지 못하는 반면, 공간 차원과 그에 대한 개념 계층을 포함하며 공간 차원에 대한 개념 계층 트리는 공간 데이터베이스에서 인덱스로 사용되는 R-tree와 유사한 구조를 가지고 있어 빠른 공간 검색이 가능하도록 고안되었다.

공간 데이터 큐브를 구축하기 위해 사실 테이블에서 첫 번째 차원의 필드 값 리스트를 얻어와 개념 계층 스키마를 참조하여 계층 트리를 만든다. 예를 들어, [그림 1]과 같은 사실 테이블과 공간 차원 및 개념 계층 스키마가 존재할 경우, 사실 테이블에서 첫 번째 차원인 '상점'의 필드 값 리스트 {*s1*, *s2*, *s5*, *s7*}을 얻어온다. 주어진 필드 값 리스트와 메타 데이터의 개념 계층 스키마 정보를 참조하여 계층 트리를 만들면 [그림 2]와 같은 모습이 된다. 그림 상에는

(a)

상점	날짜	판매 물건	판매 수량
s1	06-01-01	p1	10
s1	06-01-02	p2	20
s2	06-01-01	p1	5
s2	06-01-03	p3	10
s5	06-01-01	p2	30
s7	06-01-02	p2	25
s7	06-01-03	p1	15





(c)

개념 계층 스키마	계층 인스턴스
행정구역(동) ↓ 블록 ↓ 상점	b1 → a1 b2 → a1 s1 → b1 ⋮ s6 → b2 s7 → b2

그림 1. (a)사실테이블 SALES (b)공간 차원 (c)개념 계층 스키마

편의를 위해 *s1*과 같은 공간 객체에 대한 키 값을 나타내었지만 실제로는 그 자리에 해당하는 공간 객체가 저장된다.

계층 트리에는 여러 개의 노드가 있고 각 노드에는 여러 개의 셀이 저장되어 있다. 특히 모든 노드에는 노드 내의 모든 셀에 대한 집계 값을 구하는 경로를 가리키는 *ALL*이라는 명칭의 셀이 포함되어 있다. 예를 들어, *s1*, *s2*가 포함된 노드의 *ALL* 셀은 *b1* 블록 전체에 대한 집계 값을 구하는 경로를 가리키게 된다. '상점' 차원의 계층 트리에서 모든 단말 노드의 셀들과 *ALL* 셀에 대해 각각 다음 차원인 '날짜' 차원의 계층 트리를 같은 방법으로 구성하여 연결시킨다. 이때, 접미의 중복, 즉, 동일한 계층 트리가 나타날 경우 중복하여 만들지 않고 이를 공유하도록 하여 저장 공간의 효율성을 증대시킨다. 다음 차원인 '판매 물건' 차원에 대한 계층 트리까지 구성하여 각 계층 트리를 DAG(Directed Acyclic Graph) 형태로 연결하여 완성한 모습이 [그림 3]에 나타나있다.

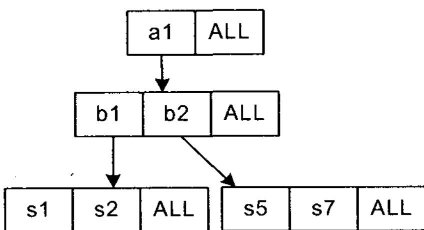


그림 2. 상점 차원의 계층 트리

이러한 공간 데이터 큐브를 통해 사용자는 원하는 차원의 원하는 계층에 대한 집계 값을 구하는 점질의 및 영역질의에 대한 결과를 얻을 수 있다. 단, [그림 3]의 공간 데이터 큐브는 집계함수 'SUM()'을 사용하여 구축한 것이다. 사용자가 질의하고자하는 2차원 공간상의 점, *QPoint*가 주어졌을 때, '*QPoint*에 해당하는 상점에서 06년 1월에 팔린 p2의 개수'를 구하는 점 질의를 예로 들면, 질의 처리를 위해 루트

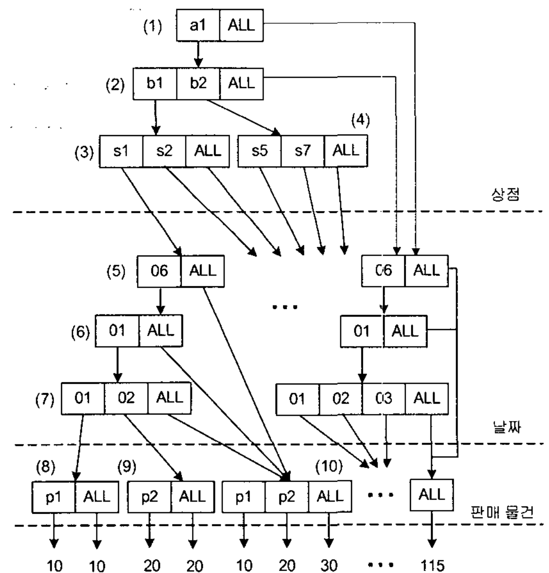


그림 3. 사실 테이블 SALES에 대한 개념 계층 공간 데이터 큐브

노드인 (1)번 노드에서부터 큐브 탐색을 시작한다. 상점 차원은 공간 차원이며 계층 트리가 R-tree와 유사한 구조이므로 공간 검색도 R-tree와 같은 알고리즘을 사용하여 *QPoint*에 해당하는 상점을 찾는다. 검색 결과가 *s1* 상점이라고 가정하면 (3)번 노드의 *s1* 셀이 가리키는 '날짜' 차원의 계층 트리로 이동한다. 06년 01월에 대한 질의이므로 (7)번 노드의 *ALL* 셀이 가리키는 (10)번 노드로 이동하고 *p2*에 대한 질의이므로 결과 값으로 20을 얻을 수 있다. 단, 편의상 '판매 물건' 차원에는 개념 계층이 없다고 가정하였다. 영역 질의는 공간 데이터 웨어하우스에서 여러 개의 점 질의로 컴파일 되어 각각의 결과를 다시 집계하여 최종 결과를 구한다. 하지만 Dwarf나 QC-tree등과 같은 기존의 큐브에 대한 연구들에 비해 이 연구에서는 큐브의 갱신 기법에 대한 연구가 부족하였다.

3. 공간 데이터 큐브의 점진적 일괄 갱신 기법

이 장에서는 제안 기법이 해결하고자 하는 문제 정의와 제안 기법을 표현한 갱신 알고리즘에 대해 기술한다. 2.3장에서 기술한 공간 데이터 큐브는 일종의 실체화 뷰이며, 기반 테이블이 되는 사실 테이블에 추출된 데이터가 ODS로부터 주기적으로 적재되면 뷰의 적시성을 보장하기 위해 변화를 반영시켜야 한다. 기본적으로 실체화 뷰를 처음부터 다시 계산하여 재구축하는 방법과 변화가 일어난 레코드를 하나씩 찾아 점진적으로 갱신하는 방법이 있다.

재구축하는 방법은 따로 알고리즘이 필요 없고 구현하기 용이하다는 장점이 있지만 기반 테이블의 작은 변화에도 재구축을 하게 되어 비효율적인 시간 비용을 요구한다. 또한, 큐브의 기반 테이블인 사실 테이블은 데이터 웨어하우스의 특성상 점점 데이터가 누적되어 그 부피가 늘어나므로 재구축에 걸리는 시간은 점점 길어지며 재구축하는 동안에 사용자에게 큐브의 적시성을 보장할 수 없게 된다.

변화가 일어날 때마다, 혹은 일정 주기마다 로그를 탐색하여 변화가 일어난 레코드를 하나씩 큐브에 점진적으로 갱신하는 기법은 하나의 레코드의 변화가 2^d개의 셀에 영향을 미치는 큐브의 특성에 적합하지 않다. 특히, 큐브가 개념 계층을 지원할 경우 그 복잡도는 더 커진다. 그리고 사실 테이블에 적재될 새로운 데이터는 ODS로부터 주기적으로 한꺼번에 전송되는 데이터 웨어하우스의 환경에도 부합하지 않는다. 이러한 환경에 가장 적합한 방법은 2.2장에 설명된 QC-tree의 갱신 기법과 같이 갱신될 레코드들을 모아서 전처리를 한 후 이를 이용하여 점진적으로 일괄 갱신하는 기법이다.

본 논문에서는 공간 데이터 웨어하우스에 적재하기 위해 임시 저장되어 있는 데이터에 대해 앞서 기술한 개념 계층 공간 데이터 큐브를 미리 구축하였다가 적재 주기가 되어 ODS가 적재 데이터와 함께 미리 구축된 공간 데이터 큐브를 전송하면, 전송된 공간 데이터 큐브를 공간 데이터 웨어하우스 내에 기 구축된 공간 데이터 큐브에 점진적으로 일괄 갱신하는 기법을 제안한다.

3.1 일괄 갱신에 따른 문제 정의

일괄 갱신 시 나타나는 문제의 원인은 사실 테이블의 데이터를 무 손실 압축을 통해 저장하는 공간 데이터 큐브가 접미의 중복을 제거하기 위해 최대한 다음 차원의 계층 트리를 공유하는 구조를 가지고 있다는 점이다. 이 때문에 큐브를 갱신하는 과정에서 공유 계층 트리를 수정할 경우와 노드에 새로운 셀이 삽입되는 경우 그리고 ALL 셀과 계층 트리를 공유하는 세 가지 경우에 갱신이상 문제가 발생한다.

공유 계층 트리를 수정할 경우, 특정 데이터가 해당 트리를 공유하고 있는 모든 경로를 통해 갱신할 경우 문제가 생기지 않지만, 특정 데이터와 상관없는 경로가 하나라도 존재하면 갱신 이후 그 경로를 통하여 처리되는 질의는 잘못된 결과를 얻게 된다는 것이다.

예를 들어, [그림 4]에서 굵은 선으로 표시된 (2)번 노드와 (3)번 노드가 (5)번 노드를 공유하여 생기는 경로 병합은 블록 *b1*에 있는 상점들 중에서 06년 01월 02일에 물건을 판매한 상점이 *s2* 밖에 없기 때문에 발생한 것이다. 이때, *s1*에서 06년 01월 02일에 물건 p1을 10개 판매하였다는 새로운 데이터가 (*s1, 06-01-02, p2, 10*)와 같이 삽입될 경우, 이를 갱신하기 위해 (1)번 노드에 02값을 갖는 셀을 추가하고 '판매 물건' 차원에 새로운 계층 트리를 만들어 연결하여 갱신할 수 있다. *s1* 상점은 *b1* 블록에 속하므로 *b1* 블록의 06년 01월 02일의 집계 값에도 영향을 미치므로 (3)번 노드의 02셀이 가리키는 (5)번 노드를 갱신하게 된다. 하지만 (5)번 노드는 (2)번 노드도 참조하고 있고, (2)번 노드는 (*s1, 06-01-02, p2, 10*)라

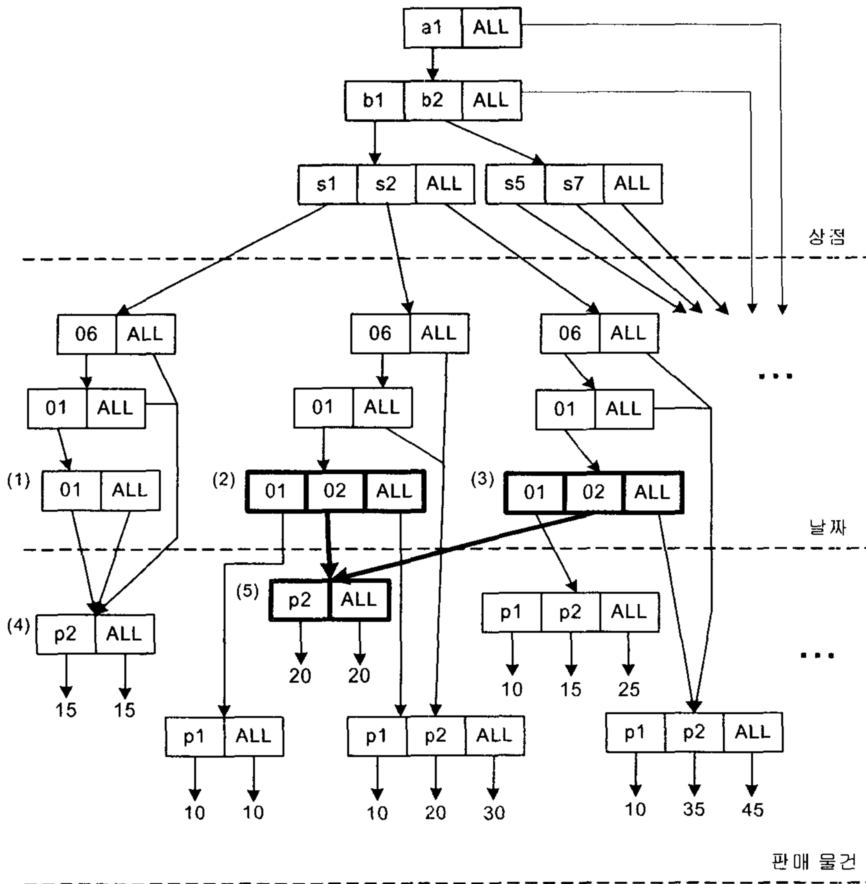


그림 4. 경로 병합에 의한 충돌의 예

는 데이터에 영향을 받지 않으므로 (5)번 노드가 갱신되면 (2)번 노드를 경유하여 (5)번 노드를 구하는 '*s2*에서 06년 01월 02일에 팔린 *p2*의 개수'와 같은 질의는 잘못된 결과를 얻게 된다.

이를 방지하기 위해서는 큐브 갱신 시 경로의 병합을 미리 감지하여 갱신할 계층 트리의 복사본을 만들어 복사본을 갱신하는 기법을 사용한다. 또한, 이후에 같은 노드에 대한 갱신 요청이 들어오면 갱신된 복사본을 반환하여 중복 작업을 최소화하고 접미의 중복을 제거한 압축률을 유지할 수 있다. 이를 위해서는 아래의 [그림 5]와 같이 각 계층 트리마다 자신을 참조하고 있는 셀의 개수를 유지하고 있는 계수기(*refCount*)가 필요하다. 또한 자신의 복사본에 대한 포인터(*copy*)를 유지하고 자신이 갱신되어 복사본이 존재한다고 표시할 수 있는 자료구조(*updated*)가 필요하다. 이러한 자료구조는 각 계층 트리마다 하나씩 필요하므로 각 트리에 헤더로서 추가된다.

또한, 각각의 셀에 있는 *sib*는 노드 내에서 자신의 오른쪽에 있는 셀을 나타내며 *label*이 *ALL*인 셀의 *sib*는 다시 노드내의 맨 왼쪽의 셀을 가리켜 링 모양

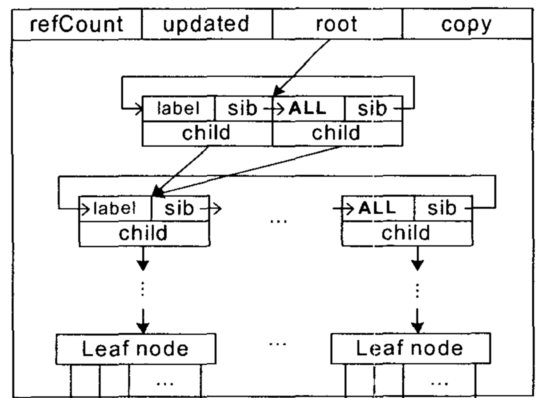


그림 5. 갱신을 위해 확장된 공간 데이터 큐브의 계층 트리 자료구조

의 연결 리스트를 형성한다. 이는 한 노드에 *ALL* 셀을 제외한 보통 셀이 하나밖에 없을 경우 그 셀과 *ALL* 셀은 항상 병합되기 때문에 3.2절에서 기술할 갱신 알고리즘에서 이를 쉽게 감지하기 위한 구조이다. 그리고 *child*는 계층 트리에서 해당 셀의 하위 개념 계층에 대한 노드를 가리키고, 단말 노드의 *child*는 다음 차원의 계층 트리를 가리킨다.

일괄 갱신으로 공간 데이터 큐브에 새로운 셀이 추가되는 경우, 새로운 셀뿐만 아니라, ODS에서 갱신 데이터를 기반으로 구축한 큐브 내에서 추가될 셀의 *child*, 즉, 하위 계층의 노드와 나머지 차원의 계층트리도 함께 삽입된다. 이때, 새로 추가된 *child*가 다른 셀에서 공유하고 있었고, 그 셀도 새로 추가되는 상황이 되면 이미 추가된 *child*를 다시 추가하지 않고 원본 큐브 내에서도 공유되도록 해야 한다. 이를 위해서는 [그림 5]의 *updated*라는 플래그 대신 갱신 적용이 되었는지의 여부를 나타내는 *applied*와 적용이 되었다면 해당 계층 트리가 새로 추가되었는지의 여부를 나타내는 *inserted*라는 두개의 플래그가 필요하다. 즉, ODS에서 갱신 데이터를 기반으로 공간 데이터 큐브를 구축할 때, [그림 5]의 자료구조를 그대로 이용하지 않고 *updated* 대신 *applied*와 *inserted*라는 두개의 플래그를 사용하여야 한다.

ALL 셀과 공유하고 있는 계층 트리를 갱신하는 경우, 또 다른 갱신 이상 현상이 발생한다. 예를 들어, [그림 3]의 (1)번 노드와 같이 *ALL* 셀을 제외한 보통 셀이 하나밖에 없는 노드에 새로운 셀이 추가되는 경우, *ALL* 셀은 더 이상 왼쪽의 셀과 계층 트리를 공유하지 못한다. 그러므로 추가될 새로운 셀과 이에 대한 하위 계층 트리를 모두 삽입한 후, 같은 노드 내의 *ALL* 셀에 대한 하위 계층 트리에 대해 복사본을 만들고 *updated* 플래그는 원본과 복사본 모두 *false*로 유지하여, 알고리즘의 흐름에 따라, 그 *ALL* 셀이 갱신될 차례가 되면 복사본에 갱신을 반영하도록 *ALL* 셀의 *child*가 복사본을 가리키게 한다. 또한, (1)번 노드의 상위 노드들도 *ALL* 셀을 제외한 보통 셀이 하나밖에 없기 때문에 같은 충돌이 발생하므로 (1)번 노드에 대한 처리가 끝나면 상위 노드들에 대한 처리가 수반되어야 한다.

3.2 점진적 일괄 갱신 알고리즘

개념 계층을 지원하는 공간 데이터 큐브는 여러 개의 계층 트리로 이루어져 있으며 전체적으로는 트리에 가까운 DAG 형태로 연결되어 있다. 이러한 특성 때문에 큐브를 갱신하는 과정에서 큐브 탐색 시에는 재귀 함수를 이용한 알고리즘이 적합하다. 다음 [알고리즘 1]은 원본 큐브(*old_cube*)와 ODS에서 구축한 임시 큐브(*delta_cube*), 그리고 차원의 개수를 입력으로 받아 임시 큐브의 내용이 갱신된 큐브를 만드는 알고리즘이다.

최상위 계층 트리의 차원번호인 1과 함께 임시 큐브와 원본 큐브의 각 최상위 계층 트리를 [알고리즘 2]에 기술된 재귀함수 *ApplyNewHTree*에 매개변수로 넘긴다. 차원번호는 사실 테이블에 나타난 순서대로 번호를 붙인 것으로 [그림 1]의 (a)에 나타난 *SALES* 테이블을 예로 들면, 1번 차원이 '상점', 2번 차원이 '날짜', 3번 차원이 '판매 물건'이 되고 3차원

〔알고리즘 1〕 공간 데이터 큐브의 점진적 일괄 갱신 알고리즘

```
Input
old_cube, delta_cube, nDim
Output
updated_Cube

Algorithm UpdateCube(old_cube, delta_cube, nDim)
Begin
01 : currHTree <- delta_cube.next
02 : oldHTree <- old_cube.root
03 : ApplyNewHTree(oldHTree, currHTree, 1, nDim)
04 : return old_cube
End
```

〔알고리즘 2〕 기존 계층 트리에 새로운 계층 트리의 내용을 반영하는 재귀함수

```
Input
oldHTree, newHTree, D: oldHTree 및 newHTree의 차원번호, nDim: 차원의 개수
Output
updatedHTree: newHTree의 내용이 반영된 oldHTree
```

```
Algorithm ApplyNewHTree(oldHTree, newHTree, D, nDim)
Begin
01 : old_cube <- (oldHTree가 속해있는 cube)
02 : currCell <- newHTree.next
03 : if D = nDim then    //마지막 차원
04 :   while currCell is not NULL
05 :     if currCell is in oldHTree then
           oldHTree.rddt.child <-
               AGG_FUNCTION(oldHTree.rddt.child, currCell.child)
06 :     else oldHTree.insert(currCell)
07 :     end if
08 :     currCell <- newHTree.next
09 :   end while
10 : else                               //중간 차원
11 :   while currCell is not NULL
12 :     if currCell is in oldHTree then
           tempHTree <- PrepareHTree(old_cube, currCell)
13 :       if tempHTree is not NULL then
             ApplyNewHTree(tempHTree, currCell.child, D+1, nDim)
15 :       end if
16 :     else InsertNewCell(oldHTree, currCell)
17 :     end if
18 :     currCell <- newHTree.next
19 :   end while
20 : end if
21 : return oldHTree
End
```

큐브이므로 *nDim*이라는 입력 변수는 3이 된다.

재귀함수 *ApplyNewHTree*는 매개변수로 입력된 계층 트리가 마지막 차원일 경우와 아닌 경우 두 가지 상황에 대해 다르게 동작한다.

매개변수로 입력된 계층 트리가 마지막 차원일 경우, 계층 트리의 *child*는 다음 차원에 대한 계층 트리를 가리키는 포인터 대신 집계 값이 존재한다. 이 경우 ODS에서 구축한 임시 큐브의 한 부분인 *newHTree*를 DFS(Depth-First Search) 순서대로 탐색하여 각 셀들을 하나씩 *oldHTree*에 반영시킨다. 반영하고자 하는 셀(*currCell*)이 *oldHTree*에 존재하지 않으면 새로 삽입하고, 이미 존재할 경우 기존의 집계 값(*oldHTree.rddt.child*)과 *currCell*의 집계 값을 집계함수(*AGG_FUNCTION*)에 따라 결합한다. 예를 들어, [그림 4]에 그려진 큐브의 경우 SUM() 집계함수를 사용하였으므로 두 집계 값을 더하고, MIN()이나 MAX()와 같은 집계함수를 사용한 경우 두 값의 크기를 비교하여 집계 값을 갱신하는 것을 말한다.

입력된 계층 트리가 마지막 차원이 아닌 경우 마찬가지로 *newHTree*를 DFS 방식으로 탐색하여 각각의 셀들을 *oldHTree*에 반영하는 기법을 사용한

다. *currCell*이 기존에 없던 셀일 경우 이를 새로 삽입하는 *InsertNewCell* 함수를 호출하며 이는 [알고리즘 3]에 기술되어 있다. 또한, 갱신을 반영하려는 *currCell*이 존재할 경우, 이에 대한 *child*의 복사본을 준비하거나, 이미 복사되어 갱신되었으면 *NULL* 값을 반환하는 *PrepareHTree* 함수, 즉, [알고리즘 4]를 호출하여 *child*의 복사본(*tempHTree*)이 준비되면 이를 다시 갱신하기 위해 *ApplyNewHTree* 함수를 *tempHTree*와 *currCell*의 *child*를 매개변수로 하여 재귀 호출한다. *tempHTree*에 *NULL*값이 반환된 것은 이미 *currCell*이 갱신되었다는 의미이므로 다른 과정을 거치지 않고 DFS 순서에 의해 다음 셀을 갱신하도록 반복문을 실행한다.

마지막 차원을 제외한 차원의 계층 트리에 새로운 셀을 추가할 경우 사용하는 *InsertNewCell* 함수는 위의 [알고리즘 3]과 같이 기술된다. 3.1장에서 제시한 갱신 이상 현상들 중 세 번째로 정의된 문제점을 고려하기 위해 [알고리즘 3]의 02번 라인에서 이를 감지하여 경로의 병합이 발생할 경우 충돌되는 하위 계층 트리를 복사한다. 또한, 상위 노드에도 ALL 셀을 제외한 보통 셀이 하나만 존재할 경우 동일한 충돌이 발생하므로 이를 08번부터 13번 라인의 반복문

[알고리즘 3] 주어진 계층 트리에 새로운 셀을 추가하는 함수

```
Input
HTree, newCell: HTree에 삽입할 셀
Output
updatedHTree: newCell이 삽입된 HTree

Algorithm InsertNewCell(HTree, newCell)
Begin
01 : firstCell <- (HTree 상에서 newCell이 삽입될 노드의 첫번째 셀)
02 : if firstCell.sib.sib = firstCell then   // ALL 셀의 포인터가 병합되어 있음
03 :   srcHTree <- firstCell.sib.child
04 :   copyHTree <- MakeCopy(srcHTree)
05 :   firstCell.sib.child <- copyHTree
06 :   srcHTree.refCount <- srcHTree.refCount - 1
07 :   firstCell <- find the first cell of parent node in Htree
08 :   while firstCell.sib.sib = firstCell  //상위에 ALL 셀의 병합을 찾아 수정
09 :     firstCell.sib.child <- copyHTree
10 :     copyHTree.refCount <- copyHTree.refCount + 1
11 :     srcHTree.refCount <- srcHTree.refCount - 1
12 :     firstCell <- find the first cell of parent node in Htree
13 :   end while
14 :   if srcHTree.refCount = 0 then Delete(srcHTree)
15 :   end if
16 : end if
17 : if newCell.child.inserted = TRUE then
18 :   newCell.child = newCell.child.copy
19 :   HTree.insert(newCell)
20 : else
21 :   newCell.child.inserted <- TRUE
22 :   newCell.child.applied <- TRUE
23 :   HTree.insertRecursively(newCell)
24 : end if
End
```

을 통해 상위의 모든 노드에서 나타나는 문제점을 해결한다. 이후 17번부터 24번 라인은 새로운 셀이 추가될 때 나타나는 또 다른 갱신 이상 현상으로 3.1장의 두 번째 문제를 고려하는 부분이다. 추가하려는 셀이 이미 원본 큐브에 복사되어 삽입되었다면 *HTree*에 *newCell*만 복사하여 추가하고 *newCell*의 *child*는 이미 삽입된 계층 트리를 가리키도록 한다. 그렇지 않을 경우에는 23번 라인의 *insertRecursively*라는 함수를 통해 *newCell*과 함께 *newCell*의 *child*에 해당하는 모든 계층 트리를 삽입한다. *insert Recursively*는 재귀적으로 마지막 차원의 계층 트리까지 탐색하여 모두 삽입하는 간단한 함수로서 [그림

〔알고리즘 4〕 해당 셀의 하위 계층 트리를 복사하여 갱신을 준비하는 함수

```
Input
cube, findCell: 검색하고자 하는 셀
Ouput
copyHTree: 검색된 셀의 child에 해당하는 트리의 복사본

Algorithm PrepareHTree(cube, findCell)
Begin
01 : idCell <- find cell in cube by findCell.child.prefix
02 : HTree <- idCell.child
03 : if HTree.updated = FALSE then   //업데이트 된 적이 없는 트리일 경우
04 :   if HTree.refCount = 1 then return HTree
05 :   else  //여러 개의 셀이 참조하는 경우 복사본을 만들어 충돌 예방
06 :     copyHTree <- MakeCopy(HTree)
07 :     HTree.copy <- copyHTree
08 :     HTree.refCount <- HTree.refCount - 1
09 :     HTree.updated <- TRUE
10 :     findCell.child.applied <- TRUE
11 :     idCell.child <- HTree.copy
12 :     return copyHTree
13 :   end if
14 : else                              //이미 업데이트 된 트리일 경우
15 :   if findCell.child.applied = TRUE then
16 :     HTree.refCount <- HTree.refCount - 1
17 :     idCell.child <- HTree.copy
18 :     if HTree.refCount = 0 then Delete(HTree)
19 :     end if
20 :     return NULL
21 :   else  // delta_cube 내에서는 충돌이 없을 경우
22 :     copyHTree <- MakeCopy(HTree)
23 :     HTree.refCount <- HTree.refCount - 1
24 :     findCell.child.applied <- TRUE
25 :     idCell.child <- copyHTree
26 :     if HTree.refCount = 0 then Delete(HTree)
27 :     end if
28 :     return copyHTree
29 :   end if
30 : end if
End
```

5]에서 제시한 계층 트리 자료형의 멤버함수이다.

마지막으로, 아래 [알고리즘 4]에 기술된 *PrepareHTree*는 [알고리즘 2]의 *ApplyNewHTree* 함수의 12번 라인에서 갱신 하려는 계층 트리에서 경로 병합에 의한 충돌의 방지가 필요하여 호출하는 함수이다. 이는 3.1장에서 첫 번째로 정의한 문제점에 해당하는 현상을 방지한다. [알고리즘 1]에서 입력(*old_cube*)으로 주어진 원본 공간 데이터 큐브(*cube*)와 갱신하려는 셀(*findCell*)을 입력받아 01번 라인에서 *findCell*과 일치하는 접두(*prefix*)를 가진 셀을 *cube*에 대한 점질의 알고리즘을 이용하여 검색한다. 검색된 셀(*idCell*)의 *child*가 가리키는 계층 트리(*HTree*)에서 병합이 일어날 수 있는 경우에 호출되는 함수이므로 *HTree*를 공유하는 셀들 간의 충돌을 감지하여 이를 해결하는 것이 목적이다. 이러한 처리를 마치면 충돌로 인한 갱신 이상 현상이 방지되어 갱신 준비를 마친 계층 트리가 반환된다. 반환된 계층 트리는 *ApplyNewHTree* 함수의 13번 라인에서 재귀호출시 갱신 대상 트리(*oldHTree*)에 해당하는 매개변수로 사용된다. *PrepareHTree* 함수의 처리 과정은 다음과 같이 *HTree*가 이번에 처음으로 갱신되는 경우와 이미 갱신된 경우에 대하여 진행된다.

처음 갱신이 되는 경우 *HTree*를 공유하고 있는 셀의 개수를 나타내는 *refCount*를 검사하여 그 값이 1이면 병합이 일어나지 않아 3.1장에서 거론된 문제들이 발생하지 않으므로 복사본을 만들지 않고 *HTree*를 그대로 반환한다. 만약 *HTree*가 공유되고 있다면 이에 대한 복사본을 만들고, 이후에 *HTree*를 공유하는 나머지 셀들을 처리하기위해 복사본을 가리키는 포인터를 [그림 5]에 나타나 있는 *HTree*의 *copy* 변수에 저장한다. 그리고 01번 라인에서 검색된 *idCell*의 *child*를 복사본으로 변경하고, *HTree*를 공유하는 셀이 하나 줄었으므로 *refCount*를 하나 줄인다. 마지막으로 *HTree*가 이미 복사되었고 갱신되었음을 나타내기 위해서 *updated* 플래그와 *idCell*의 *applied* 플래그를 *TRUE*로 변경하고 복사된 계층 트리를 반환한다.

*HTree*가 이미 갱신된 경우는 [알고리즘 1]에서 입력된 갱신 큐브(*delta_cube*) 내에서도 *findCell*의 *child*가 이미 적용이 되었는지 여부에 따라 다르게 처리한다. *HTree*가 이미 갱신되었어도 이번에 갱신하려는 내용이 이전에 갱신된 것과 다를 경우 별도의 복사본이 필요하기 때문이다. 만약 이번에 갱신하려는 내용의 *applied* 플래그가 *TRUE*이면 같은 내용의 갱신이 이미 이루어진 것이므로 16번과 17번 라인에서 *idCell*의 *child*를 갱신된 복사본으로 바꾸고 *refCount*를 하나 줄인다. 그리고 호출자에게 중복해서 갱신할 필요가 없다는 것을 알리기 위해 *NULL* 값을 반환한다.

한편, 앞에서 언급한 것과 같이 이전에 갱신된 것과 다른 내용의 갱신일 경우에는 06번부터 12번 라인의 절차와 유사하게 22번부터 28번 라인에서 별도의 *HTree* 복사본을 준비하고 이를 반환한다. 또한, 18번과 26번 라인에서는, 여러 번의 갱신을 통해 더 이상 *HTree*를 공유하는 셀이 없어서 *refCount*가 0이 되면, *HTree*를 저장할 필요가 없으므로 해당 계층 트리를 원본 공간 데이터 큐브에서 삭제한다.

이러한 알고리즘들을 사용하여 [그림 4]의 큐브에 (*s1*, *06-01-02*, *p2*, *10*)을 삽입한 결과는 [그림 6]과 같다. 새로 삽입된 노드는 일점쇄선 및 음영으로 표시하였고, 빗금 친 부분은 복사 및 수정된 노드이다.

s1 상점에서 06년 01월 02일에 팔린 물건이 없으므로 (1)노드에 '02' 셀이 추가되고 하위 노드들도 새로 삽입된다. (1)번 노드에 새로운 데이터가 삽입되었으므로 해당하는 *ALL* 셀을 갱신하여야 하는데 [그림 4]에서 보는 바와 같이 '01' 셀과 개념 계층 트리를 공유하고 있으므로 (2)번 노드를 복사하여 (3)번 노드를 만들고 이를 갱신하여 집계 값이 15에서 10을 합한 25가 된다. 마찬가지로 *s1* 상점이 속한 *b1* 블록의 데이터도 갱신하려면 (4)번 노드를 갱신해야하지만 삽입하려는 레코드와 상관없는 노드와 공유하고 있으므로 (4)번을 복사하여 (5)번을 만들어 갱신한다. 마찬가지로 *b1* 블록이 속한 *a1* 지역에 대한 데이터와 모든 지역에 대한 데이터를 갱신하고 레코드 삽입과정을 마친다.

4. 성능 평가

이 장에서는 제안 기법의 효율성을 평가하기위해 ODS로부터 데이터가 적재될 때마다, 큐브를 재계산하여 새로 생성하는 재구축 기법과 적재된 레코드를 하나씩 삽입하는 개별삽입 기법을 제안 기법인 점진적 일괄 갱신 기법과 비교분석하였다. 실험환경은 [표 1]과 같으며 실험을 위해 인자로 2를 사용한 Zipf 분포를 갖는 합성 데이터와 전국의 실제 날씨 데이터

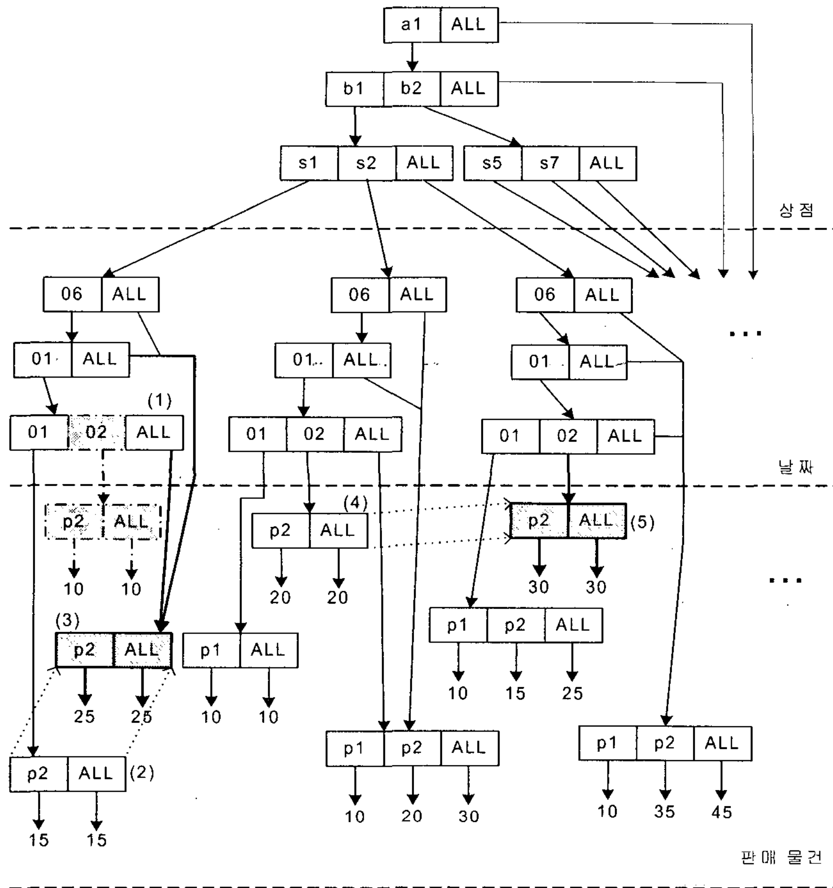


그림 6. 갱신된 공간 데이터 큐브의 예

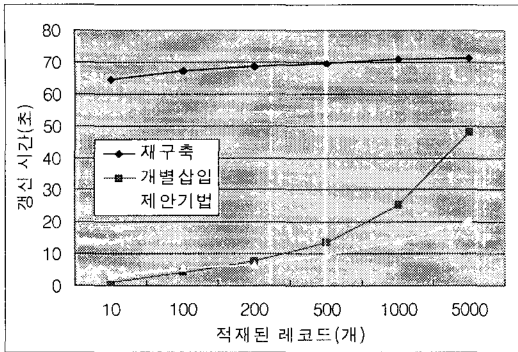
표 1. 실험 환경

중앙 처리 장치	Pentium4 3.0GHz
주 기억 장치	1GB
보조 기억 장치	150GB
운영체제	Windows XP Professional
개발 환경	Visual C++ 6.0
개발 언어	C/C++

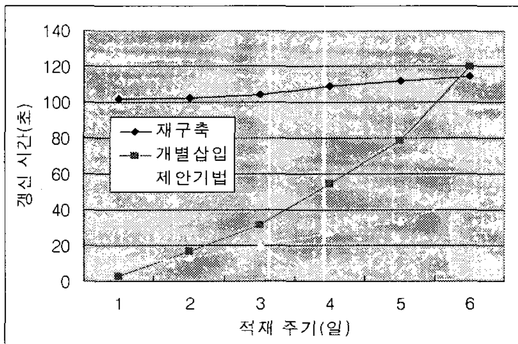
를 담고 있는 두 개의 사실 테이블을 사용하였다.

합성 데이터를 갖는 테이블은 공간 차원으로 주소 개념 계층을 갖는 서울시 강남구에 대한 공간 데이터를 저장하고 있다. 이외에 8개의 비 공간 차원을 포함하고 있는데, 이는 개념 계층에 대해 균일한 기수(cardinality)를 갖고 Zipf(2) 분포를 나타내는 데이터를 담고 있다. 집계 값 또한 같은 방법으로 생성된 수치 데이터이다. 날씨 데이터는 1998년부터 2000년까지 서울시의 행정 구역 별 강우량을 나타내며 5개의 차원을 갖고 있다. 이러한 두 개의 사실 테이블에 대해 각각 집계함수 SUM()과 AVG()를 사용한 큐브를 생성하여 앞에서 언급한 두 가지 기법과 제안 기법에 의한 갱신 속도와 저장 용량을 각각 사실 테이블에 적재된 레코드의 개수 및 차원의 개수를 변화시키면서 측정하는 두 가지 실험을 하였다.

첫 번째 실험으로 100,000개의 레코드를 가지고 있는 합성 사실 테이블에 ODS로부터 적재되는 레코드의 개수를 10개에서 5000개까지 변화시키면서 각각의 기법을 통해 갱신이 완료되는데 걸리는 시간을 측정하였으며 실험 결과는 [그림 7]의 (a)와 같다. 또한 1998년부터 1999년까지의 날씨 데이터를 담고 있는 사실 테이블에 적재되는 2000년 데이터의 적재주기를 변화시키면서 실험한 결과는 [그림 7]의 (b)와 같다. 실험 결과는 두개의 대상에 대해 비슷한 양상을 보였다. [그림 7]의 (a)를 예로 들면, 재구축 기법은 기존 100,000개의 레코드와 적재된 레코드를 함께



(a)

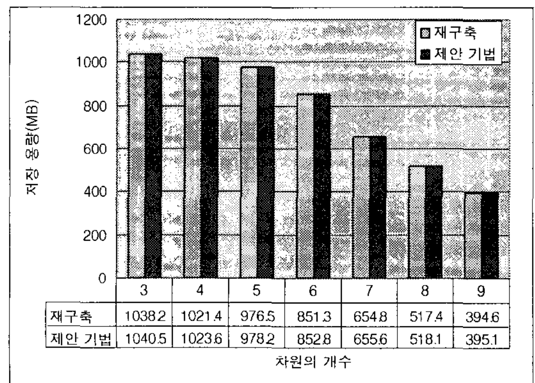


(b)

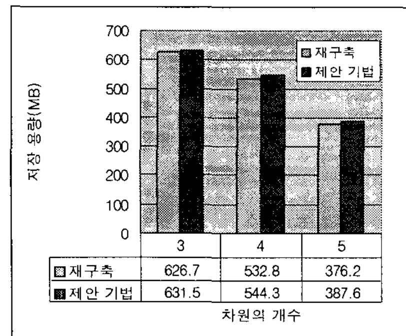
그림 7. 적재되는 데이터의 크기에 따른 갱신 시간 측정 결과: (a) 합성 데이터, (b) 전국 날씨 데이터

한 번에 재계산 하는 방식으로 큐브를 갱신하기 때문에 적재된 레코드의 개수가 기존 레코드의 개수에 비해 적을 경우 적재된 레코드의 개수에 따른 갱신 시간의 증가량이 적어 거의 일정한 갱신 시간을 보인다. 하지만 전체적으로 갱신 시간이 길어 나머지 두 기법보다 현저히 낮은 속도를 나타낸다. 한편, 개별삽입 기법은 적재된 레코드의 개수가 200개 이하인 영역에서는 제안 기법에 비해 더 빠르거나 비슷한 속도를 보인다. 이는 적재된 레코드의 개수가 200개 이하로 적을 경우 적재된 레코드에 대한 전처리과정이 없는 개별삽입 기법이 적재될 데이터에 대한 큐브를 구축하는 전처리가 필요한 제안 기법에 비해 더 유리하기 때문이다. 하지만 적재된 레코드의 개수가 늘어남에 따라 500개 이후의 영역에서는 기하급수적으로 갱신 시간이 증가하며 적재된 레코드의 개수가 20,000개 이상이 되면 재구축 기법보다 느린 속도를 보인다.

두 번째 실험은 각 사실 테이블의 차원의 개수를 변화시키며 합성 사실 테이블에는 5,000개의 레코드, 날씨 사실 테이블에는 5일치 데이터가 적재된 경우 대해 제안 기법과 재구축 기법을 이용하여 갱신을 마친 각각의 큐브에 대한 저장 용량을 비교 측정하였다. 이 실험은 제안 기법을 통한 갱신이 공간 데이터 큐브의 압축을 통한 효율적인 저장 성능을 저하시키지 않고 그대로 유지함을 증명하며, 그 결과는 [그림 8]과 같다. 전체적으로 제안 기법의 저장 용량이 다소 더 큰 이유는 [그림 5]와 같이 점진적 일괄 갱신 기법을 위해 확장된 헤더를 각 계층 트리마다 하나씩 추가 저장해야하기 때문이다. 즉, 점진적 일괄 갱신을 위한 추가정보의 저장비용을 제외하면, 재구축을 통해 생성된 큐브와 제안 기법을 이용해 갱신된 큐브의 압축률이 차원의 개수와 상관없이 거의 같다고 할 수 있다. 또한, 차원의 개수가 늘어남에 따라 저장 용량이 작아지는데, 이는 일정한 개수의 레코드를 갖는 사실 테이블에서 차원의 개수가 늘어날수록 각 차원의 기수가 줄어들어 발생하는 잦은 경로의 병합으로 인해 큐브 압축률이 커지기 때문이다.



(a)



(b)

그림 8. 차원의 개수에 따른 저장 용량: (a) 합성 데이터, (b) 날씨 데이터

5. 결론 및 향후 연구

본 논문에서는 공간 및 비 공간 차원의 개념 계층을 지원하는 공간 데이터 큐브의 점진적 일괄 갱신기법을 제안하였다. 이는 DFS 방식으로 큐브를 재귀탐색하면서 데이터 갱신 시 병합된 경로를 감지하고, 복사본을 관리하기 위해 확장된 자료구조를 사용한다. 제안 기법은 충돌이 일어나는 부분의 사본을 만들어 이를 갱신함으로써 갱신이상 현상을 방지하였고, 기존 공간 데이터 큐브의 압축률을 유지하면서 재구축 기법이나 개별삽입 기법 보다 안정적이고 빠른 속도를 보였다. 이를 통해 공간 데이터 웨어하우스 사용자에게 보다 높은 적시성을 갖는 공간 데이터 큐브를 제공할 수 있다. 성능평가에서 재구축 기법 보다 평균 5배 빠른 속도를 보였으며 갱신 소요시간 면에서 개별삽입 기법보다 더 낮은 증가율을 보였다. 또한 재구축 기법과 동일한 압축률을 보여 점진적 일괄 갱신이 기존 공간 데이터 큐브의 압축 성능을 유지함을 보였다. 향후 연구로는 구축된 큐브를 효율적으로 저장 및 관리하는 파일 구조에 대한 연구가 필요하다.

참 고 문 헌

[1] E. F. Codd, S. B. Codd, and C. T. Salley, "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate," *Technical Report, IBM*, San Jose, CA, 1993.

[2] S. Chaudhuri, and U. Dayal, "An Overview of Data Warehousing and OLAP Technology," *ACM SIGMOD*, Vol. 26, No. 1, pp 65-74, 1997.

[3] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub- Totals," *Proc. of the 12th Int. Conf. on Data Engineering, IEEE ICDE*, pp 152-159, 1996.

[4] S. Bimonte, A. Tchounikine, and M. Miquel, "Towards a Spatial Multidimensional Model," *Proc. of the 8th ACM Int. workshop on Data warehousing and OLAP, ACM DOLAP*, pp 39-46, 2005.

[5] M. Ester, J. Kohlhammer, and H.-P. Kriegel, "DC-tree: A Fully Dynamic Index Structure for Data Warehouses," *In Proc. Int. Conf. on Data Engineering, IEEE ICDE*, pp 379-388, 2000.

[6] F. Rao, L. Zhang, X.L. Yu, Y. Li, and Y. Chen, "Spatial Hierarchy and OLAP-Favored Search in Spatial Data Warehouse," *Proc. of the 6th ACM Int. workshop on Data warehousing and OLAP, ACM DOLAP*, pp 48-55, 2003.

[7] Y. Sismanis, A. Deligiannakis, Y. Kotidis, and N. Roussopoulos, "Hierarchical Dwarfs for the Rollup Cube," *Proc. of the 6th ACM Int. workshop on Data warehousing and OLAP, ACM DOLAP*, pp 17-24, 2003.

[8] K. Hu, L. Chen, S. Jie, Q. Gu, and X. Tang, "A Highly Performance Dimension Hierarchy Aggregate Cube and Its Incremental Update Algorithm," *Proc. of Int. Conf. on Machine Learning and Cybernetics*, Vol. 4, No.1, pp 2195-2199, 2005.

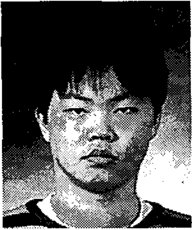
[9] 옥근형, 이동욱, 유병섭, 배해영, "공간 데이터 웨어하우스에서 개념 계층을 지원하는 공간 데이터 큐브," 한국정보처리학회 춘계학술대회, Vol. 13, No. 1, pp 35-38, 2006.

[10] N. Roussopoulos, Y. Kotidis, and M. Roussopoulos, "Cubetree: Organization of and Bulk Incremental Updates on the Data Cube," *Proc. of Int. Conf. on Management of data, ACM SIGMOD*, pages 89-99, 1997

[11] Y. Sismanis, A. Deligiannakis, N. Roussopoulos, and Y. Kotidis, "Dwarf: Shrinking the PetaCube," *Proc. of Int. Conf. on Management of data, ACM SIGMOD*, pp 464-475, 2002.

[12] L. Lakshmanan, J. Pei, and Y. Zhao, "QC-Trees: An Efficient Summary Structure for Semantic OLAP," *Proc. of Int. Conf. on Management of data, ACM SIGMOD*, pp 64-75, 2003.

[13] L. Lakshmanan, J. Pei, and J. Han, "Quotient Cube: How to Summarize the Semantics of a Data Cube," *Proc. 28th Int. Conf. on Very Large Database, IEEE VLDB*, pp 778-789, 2002.

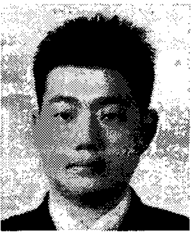


옥 근 형

2005년 인하대학교 컴퓨터공학부(공학사)
2005년~현재 인하대학교 대학원 컴퓨터정보공학과(석사과정)
관심분야 : 공간 데이터베이스, 공간 데이터웨어하우스, 공간 OLAP

이 동 욱

1996년~2003년 상지대학교 전자계산공학과 학사
2003년~2005년 인하대학교 컴퓨터정보공학과 석사
2005년~현재 인하대학교 컴퓨터정보공학과 박사과정
관심분야 : Spatial Database Warehouse, Spatial Information Management, Ubiquitous 환경을 위한 SDBMS



유 병 섭

2002년 인하대학교 컴퓨터공학부(공학사)
2004년 인하대학교 컴퓨터공학부(공학석사)
2004년~현재 인하대학교 대학원 컴퓨터정보공학과(박사과정)
관심분야 : 공간데이터베이스, 공간 데이터 웨어하우스, Data Stream, 유비쿼터스 컴퓨팅

이 재 동

1985년 인하대학교 전자계산학과 학사
1991년 미국 Cleveland State Univ., Dept. of Computer & Information Science (M.S.)
1996년 미국 Kent State Univ., Dept. of Computer Science(Ph.D.)
1996년~1997년 (주)두루넷 기술기획팀 팀장
1997년~현재 단국대학교 정보컴퓨터학부 컴퓨터과학교수
2004년~2006년 단국대학교 정보통신원장 (CIO)
2002년~현재 농협중앙회 전산고문
2004년~2006년 전국대학정보화협의회 이사
2006년~현재 민관확대 콘텐츠 정책협의회 위원
관심분야 : Contents Technology, High Performance Computing(Clustering systems etc.), GIS Technologies and Applications, Many aspects of parallel/distributed processing

배 해 영

1974년 인하대학교 응용물리학과(공학사)
1978년 연세대학교 대학원 전자계산학과(공학석사)
1989년 숭실대학교 대학원 전자계산학과(공학박사)
1985년 Univ. of Houston 객원교수
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
관심분야 : 분산 데이터베이스, 공간 데이터베이스, 지리정보 시스템, 멀티미디어 데이터베이스