데이타 웨어하우스에서 데이타 큐브를 위한 효율적인 점진적 관리 기법

(An Efficient Incremental Maintenance Method for Data Cubes in Data Warehouses)

이 기 용 † 박 창 섭 †† 김 명 호 †††
(Ki Yong Lee) (Chang-Sup Park) (Myoung Ho Kim)

요 약 데이타 큐브는 차원 애트리뷰트의 모든 가능한 조합에 대해 데이타를 집단화하는 연산자이다. 차원 애트리뷰트의 수가 n일 때, 데이타 큐브는 2^n개의 group-by를 계산한다. 데이타 큐브에 포함된 각각의 group-by를 *큐보이드(cuboid)*라 부른다. 데이타 큐브는 흔히 미리 계산되어 형태 뷰(materialized view)의 형태로 데이타 웨어하우스에 저장된다. 이러한 데이타 큐브는 소스 릴레이션이 변경되면 이를 반영하기 위해 갱신되어야 한다. 데이타 큐브의 점진적 관리는 데이타 큐브의 변경될 내용만을 계산하여 이를 데이타 큐브에 반영하는 방법을 의미한다. 2^n개의 큐보이드로 이루어진 큐브의 변경될 내용을 계산하기 위하여, 기존의 방법들은 데이타 큐브와 동일한 개수의 큐보이드를 가지는 *변경 큐브*를 계산한다. 따라서, 차원 애트리뷰트의 수가 증가할수록 변경 큐브를 계산하는 비용이 매우 커지게 된다. 변경 큐브에 포함된 각 큐보이드들을 변경 *큐보이드(delta cuboid)*라 부른다. 본 논문에서는 2^n개의 변경 큐보이드 대신 ${}_nC_{\lceil n/2 \rceil}$개의 변경 큐보이드만을 사용하여 데이타 큐브를 갱신하는 방법을 제안한다. 이에 따라 제안하는 방법은 변경 큐브를 계산하는 비용을 크게 줄일 수 있다. 성능 평가 결과는 제안하는 방법이 기존의 방법에 비해 더 좋은 성능을 가지고 있음을 보여준다.

키워드 : 데이타 큐브, 점진적 뷰 관리, 큐브 관리

Abstract The data cube is an aggregation operator that computes group-bys for all possible combination of dimension attributes. When the number of the dimension attributes is n, a data cube computes 2^n group-bys. Each group-by in a data cube is called a *cuboid*. Data cubes are often precomputed and stored as materialized views in data warehouses. These data cubes need to be updated when source relation change. The incremental maintenance of a data cube is to compute and propagate only its changes. To compute the change of a data cube of 2^n cuboids, previous works compute a *delta cube* that has the same number of cuboids as the original data cube. Thus, as the number of dimension attributes increases, the cost of computing a *delta cube* increases significantly. Each cuboid in a delta cube is called a delta cuboid. In this paper, we propose an incremental cube maintenance method that can maintain a data cube by using only ${}_nC_{\lceil n/2 \rceil}$ delta cuboids. As a result, the cost of computing a delta cube is substantially reduced. Through various experiments, we show the performance advantages of our method over previous methods.

Key words : Data Cube, Incremental View Maintenance, Cube Maintenance

1. 서 론

데이타 큐브는 데이타 웨어하우스에서 데이타를 다양한 차원으로 집단화(aggregation)하는데 사용되는 연산자이다. n개의 차원 애트리뷰트가 주어졌을 때, 데이타 큐브는 주어진 차원 애트리뷰트들의 모든 조합에 대한 group-by를 계산한다. 예를 들어, 애트리뷰트 a, b, c, m을 가지는 릴레이션 F(a, b, c, m)가 있다고 하자. 다음은 3개의 차원 애트리뷰트 a, b, c에 대한 데이타 큐브 C를 나타낸다.

```
C: SELECT a, b, c, SUM(m)
```

† 학생회원 : 한국과학기술원 전산학
kylee@dbserver.kaist.ac.kr
†† 정 회 원 : 수원대학교 인터넷정보공학과 교수
park@suwon.ac.kr
††† 종신회원 : 한국과학기술원 전산학 전공 교수
mhkim@dbserver.kaist.ac.kr
논문접수 : 2005년 10월 10일
심사완료 : 2005년 12월 22일

```
FROM F
CUBE BY a,b,c
```

CUBE BY 키워드는 데이타 큐브 연산자를 나타낸다. Group-by를 그의 그룹화 애트리뷰트만으로 간단히 표시하면, C는 abc, ab, ac, bc, a, b, c, none의 총 8개의 *group-by*를 계산한다. 여기서 *none*은 *group-by* 애트리뷰트가 존재하지 않는 빈(empty) group-by를 나타낸다. 데이타 큐브에 포함된 각각의 group-by를 *큐보이드(cuboid)*라 부른다. 일반적으로 n개의 차원 애트리뷰트에 대한 데이타 큐브는 2^n개의 큐보이드를 가진다.

데이타 큐브는 그의 계산 비용이 매우 크기 때문에, 많은 경우 미리 계산되어 데이타웨어하우스에 저장된다. 이렇게 미리 계산되어 저장된 데이타 큐브를 큐브 뷰라고 한다[1]. 큐브 뷰는 그의 데이타 소스가 변경되면 이를 반영하기 위해 갱신되어야 한다. 큐브 뷰를 갱신하는 데에는 전체 데이타 큐브를 처음부터 다시 계산하는 재계산 방법과, 큐브 뷰의 변경될 내용만을 계산하고 이를 큐브 뷰에 반영하는 점진적 관리(incremental maintenance) 방법이 있다. 변경될 내용의 양이 적은 경우 큐브 전체를 다시 계산하는 재계산 방법보다 점진적 관리 방법이 더 효율적일 수 있다. 이에 따라 큐브 뷰의 점진적 관리 방법에 대한 연구가 있어왔다[1-4].

큐브 뷰의 점진적 관리는 큐브 뷰의 변경될 내용을 계산하는 *변경 계산 단계(propagate stage)*와 변경될 내용으로 큐브 뷰를 갱신하는 *변경 반영 단계(refresh stage)*로 나뉜다[1]. 위 예에서 F에 삽입된 튜플들이 ΔF로 주어졌다고 하자. 변경 계산 단계에서는 다음과 같이 ΔF로 인한 C의 변경 ΔC를 계산한다.

```
ΔC: SELECT a, b, c, SUM(m)
    FROM ΔF
    CUBE BY a, b, c
```

ΔC를 C에 대한 변경 큐브라고 부르자. ΔC는 FROM 절에 F 대신 ΔF가 나타난다는 것을 제외하고는 C와 동일한 정의를 가진다. ΔC는 C와 마찬가지로 8개의 큐보이드를 가진다. ΔC를 구성하는 각각의 큐보이드를 *변경 큐보이드(delta cuboid)*라고 부른다. ΔC의 변경 큐보이드들을 Δabc, Δab, Δab, Δbc, Δa, Δb, Δc, Δnone으로 표시하자. 변경 반영 단계에서는 그림 1(a)이 나타내는 바와 같이 ΔC의 변경 큐보이드들을 사용하여 C의 각 큐보이드들을 갱신한다. Δc → c는 Δc의 내용을 c에 반영함으로써 c를 갱신함을 나타내는 기호이다. (구체적인 반영 방법은 2장에서 설명한다.) C의 점진적인 관리에 드는 총 비용은 변경 계산 단계의 비용과 변경 반영 단계의 비용의 합이다.

지금까지의 연구는 2^n개의 큐보이드로 이루어진 큐브 뷰를 갱신하기 위해 그림 1(a)와 같이 변경 반영 단계

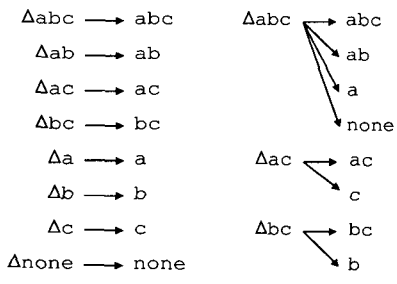


그림 1 데이타 큐브 C의 변경 반영 단계

에서 2^n개의 변경 큐보이드를 사용해 왔다. 따라서, n이 커질수록 변경 큐보이드들을 계산하는 비용이 매우 커지게 된다. 본 논문에서는 2^n개의 변경 큐보이드를 모두 사용하지 않고 일부의 변경 큐보이드만을 사용하여 전체 큐브 뷰를 갱신하는 방법을 제안한다. 그림 1(b)는 본 논문에서 제안하는 방법을 나타낸다. 본 논문에서 제안하는 방법은 하나의 변경 큐보이드로 여러 개의 큐보이드를 갱신한다. 본 논문에서 제안하는 방법은 계산해야 하는 변경 큐보이드의 수를 줄임으로써 변경 계산 단계의 비용을 줄인다. 그러면서도 각 큐보이드를 갱신하는 변경 반영 단계의 비용은 증가시키지 않는다. 본 논문의 공헌은 다음과 같다.

- 본 논문에서는 2^n개 대신 일부의 변경 큐보이드만을 사용하면서도 변경 반영 단계의 비용을 증가시키지 않고 전체 큐브 뷰를 갱신하는 방법을 제안한다.
- 변경 반영 단계의 비용을 증가시키지 않고 전체 큐브 뷰를 갱신할 수 있도록 하는 변경 큐보이드의 집합은 여러 가지가 있을 수 있다. 본 논문은 이러한 집합들 중 변경 계산 단계의 비용을 최소화하는 변경 큐보이드의 집합을 찾는 문제를 정의하고, 해당 문제에 대한 해결책을 제안한다.

본 논문의 구성은 다음과 같다. 2장에서는 큐브 뷰의 점진적인 관리 방법에 대한 기존 연구를 살펴본다. 3장에서는 2^n개의 변경 큐보이드를 다 구하지 않고도 변경 반영 단계의 비용 증가 없이 전체 큐브 뷰를 갱신하는 방법을 설명한다. 4장에서는 변경 반영 단계의 비용을 증가시키지 않고 전체 큐브 뷰를 갱신할 수 있도록 하는 변경 큐보이드의 집합들 중 최적의 집합을 찾는 문제를 정의하고 그의 해결책을 제안한다. 5장에서는 제안하는 방법에 대한 성능 평가 결과를 보여주며, 6장에서는 결론을 맺는다.

2. 관련 연구

2.1 하나의 집단화 뷰(aggregation view)에 대한 점진적 관리 방법

Group-by 연산자를 포함하는 하나의 집단화 뷰에 대한 점진적 관리 방법에 대해서는 많은 연구가 있어왔다[5-7]. 큐브 뷰를 구성하는 큐보이드들도 각각 하나의 집단화 뷰에 해당한다. 1장의 예에서의 큐보이드 abc에 대한 점진적 관리 방법에 대해 알아보자. 다음은 abc와, ΔF에 의한 abc의 변경 Δabc을 구하는 방법을 나타낸다.

```
abc: SELECT a, b, c, SUM(m)
     FROM F
     GROUP BY a, b, c
Δabc: SELECT a, b, c, SUM(m)
      FROM ΔF
      GROUP BY a, b, c
```

위와 같이 Δabc가 구해지면, abc는 그림 2와 같은 방법으로 갱신된다. 그림 2는 Δabc의 각 튜플 Δt에 대해 그룹화 애트리뷰트의 값이 동일한 튜플을 abc에서 검색한다. 만약 해당 튜플이 abc에 존재하면 검색된 튜플의 집단화 값을 갱신하고, 그렇지 않으면 Δt를 abc에 삽입한다.

그림 2의 방법으로 갱신이 가능하려면 집단화 뷰에서 사용된 집단화 함수가 분배적(distributive) 함수[8]여야 한다. 본 논문에서는 설명의 편의를 위해 집단화 함수로 SUM()만을 가정한다. 그러나 본 논문에서 제안하는 방법은 일반적인 분배적 함수로 쉽게 확장이 가능하다.

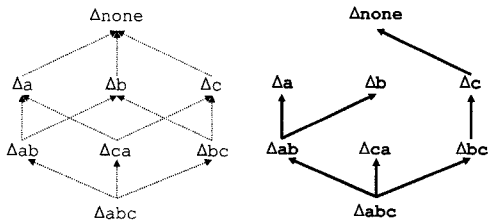
F에서 튜플이 삭제되는 경우도 그림 2와 비슷한 방법으로 처리될 수 있다. 단, 이 경우에는 Δabc의 튜플로 abc를 검색한 뒤, 찾아진 튜플을 삭제하거나 찾아진 튜플의 집단화 값을 감소시킨다. F의 튜플이 갱신되는 경우는 튜플의 삭제 후 삽입으로 처리가 가능하다. 삭제 또는 갱신에 대한 자세한 내용은 [1]을 참조하라. 본 논문에서는 설명의 편의를 위해 F에 튜플이 추가되는 경우만을 가정한다. 그러나 본 논문에서 제안하는 방법은 F의 튜플이 삭제 또는 갱신되는 경우로도 쉽게 확장이 가능하다.

```
For each tuple Δt = (a_i, b_i, c_i, m_i) in Δabc
    Find t = (a_i, b_i, c_i, m_i') in abc;
    If t is not found,
        Insert Δt into abc;
    Else
        Update t by (a_i, b_i, c_i, m_i' + m_i);
```

그림 2 Δabc를 사용하여 abc를 갱신하는 방법

2.2 큐브 뷰에 대한 점진적 관리 방법

본 절에서는 여러 개의 큐보이드를 포함하는 큐브 뷰에 대한 점진적 관리 방법을 알아본다. 1장에서 언급한



(a) ΔC에 대한 격자 그래프 (b) ΔC의 계산계획

그림 3 ΔC에 대한 격자 그래프 및 계산 계획

바와 같이 큐브 뷰에 대한 점진적 관리는 변경 계산 단계와 변경 반영 단계로 이루어진다. 지금까지의 방법은 변경 계산 단계에서 2^n개의 변경 큐보이드를 계산한다. 변경 큐브는 격자 그래프(lattice graph)로 표현될 수 있다[9]. 그림 3(a)는 1장의 변경 큐브 ΔC를 격자 그래프로 나타낸 것이다. 격자 그래프의 각 노드는 변경 큐보이드를 나타내며, 간선 $(\Delta c_i, \Delta c_j)$는 Δc_j가 Δc_i로부터 계산될 수 있음을 의미한다. 그림 3(b)는 2^n개의 변경 큐보이드에 대한 계산 계획을 보여준다. 한 변경 큐보이드는 여러 변경 큐보이드들로부터 계산될 수 있기 때문에 그림 3(b)와 같은 계산 계획에는 여러 가지가 있을 수 있다. 변경 계산 단계의 비용을 줄이기 위해서는 이들 중 최소 비용의 계산 계획을 찾아야 한다. 최소 비용의 계산 계획을 찾는 문제는 4장에서 논의한다.

변경 계산 단계에서 위와 같이 변경 큐보이드들이 계산되면, 변경 반영 단계에서는 그림 1(a)와 같이 ΔC의 각 변경 큐보이드들로 C의 큐보이드들을 갱신한다. 각 큐보이드들에 대한 갱신 방법은 2.1절에서 설명한 바와 같다.

데이타 큐브의 효율적인 계산 방법에 대해서는 많은 연구가 있어왔다[10-12,14]. 이에 비해 큐브 뷰의 점진적 관리 방법에 대한 연구는 비교적 많이 다뤄지지 않은 편이다. [1]은 큐브 뷰의 점진적 관리에 대해 최초로 언급한 논문이다. [2]는 cubetree라 불리는, 큐브 뷰를 효율적으로 관리할 수 있도록 해주는 큐브의 저장 구조를 제안하였다. [3]은 큐브 뷰의 정의에 포함된 차원 애트리뷰트가 변경된 경우에 대한 점진적 관리 기법을 제안하였다. [4]는 IBM DB2 UDB에서 집단화 뷰의 점진적 관리를 위해 실제로 구현된 관리 방법을 설명하였다. 지금까지 제안된 집단화 뷰 및 큐브 뷰에 대한 관리 방법에 대한 정리는 [15]를 참조하라.

그러나 지금까지 제안된 방법들은 2^n개의 큐보이드로 이루어진 큐브 뷰를 갱신하는 데 2^n개의 변경 큐보이드를 사용한다. 따라서 n이 커질수록 변경 큐보이드들을 계산하는 비용이 커진다. 본 논문에서는 2^n개의 변경 큐보이드를 다 사용하지 않고도 전체 큐브 뷰를 갱신하는

방법을 제안한다. 본 논문에서 제안하는 방법은 하나의 변경 큐보이드로 여러 개의 큐보이드를 갱신한다. 그러면서도 제안하는 방법은 변경 반영 단계의 비용을 증가시키지 않는다. 다음 장에서는 제안하는 방법을 설명한다.

3. 접근 방법

본 장에서는 2^n개의 변경 큐보이드를 모두 사용하지 않고도 전체 큐브 뷰를 갱신하는 방법을 설명한다.

3.1 하나의 변경 큐보이드로 여러 큐보이드 갱신하기

1장에서의 변경 큐보이드 Δabc를 고려하자. 지금까지의 방법은 Δabc를 abc를 갱신하는 데만 사용하였다. 그러나 Δabc는 abc 외에 ab를 갱신하는 데에도 사용될 수 있다. 그림 4는 Δabc로 abc와 ab를 동시에 갱신하는 방법을 보여준다.

그림 4는 Δabc의 각 튜플 $\Delta t = (a_i, b_i, c_i, m_i)$에 대해 abc와 ab에서 각각 $(a_i, b_i, c_i, *)$, $(a_i, b_i, *)$에 해당하는 튜플을 찾아 이를 갱신한다. ('*'은 임의의 값을 의미한다.) 그림 4에서 Δab는 사용되지 않는다. 그림 4의 방법은 Δabc만으로 간단하게 abc와 ab를 모두 갱신할 수 있게 해준다.

그림 4의 방법은 abc에 대해서는 그림 2의 방법과 완전히 동일하다. 그러나 ab에 대해서는 Δab를 사용하여 갱신될 때보다 접근 횟수가 증가하게 된다. 이는 ab에서 $(a_i, b_i, *)$에 해당하는 튜플을 찾아 이를 갱신하는 작업이 Δabc의 튜플 $(a_i, b_i, *, *)$에 대해 반복적으로 수행되기 때문이다. 이에 따라 그림 4는 ab에 대한 변경 반영 비용이 증가한다는 단점이 있다.

그러나 Δabc의 튜플들이 abc의 값으로 정렬되어 있다면, ab에 대한 변경 반영 비용을 증가시키지 않고도 abc와 ab를 동시에 갱신할 수 있다. 그림 5는 본 논문에서 제안하는 방법을 나타내는 그림이다. m_{ab}는 Δabc를 순환하는 과정에서 a 애트리뷰트와 b 애트리뷰트의 값이 동일한 튜플들의 집단화 값을 누적하는 변수이다. 그림 5에서 Δabc는 abc의 값에 따라 오름차순으로 정렬되어 있다. 제안하는 방법은 Δabc의 매 튜플마다 ab에 접근하는 대신, Δabc의 현재 튜플이 속한 그룹이 바뀌는 경우에만 ab에 접근한다. 예를 들어, 그림 5에서는 Δabc의 튜플 (1, 1, 1, 3)에 대해서는 ab의 튜플 (1, 1, 9)를 갱신하지 않는다. 대신 Δabc 튜플의 값이 (1, 1, *, *)에서 (1, 2, *, *)로 바뀌는 순간에만 m_{ab}에

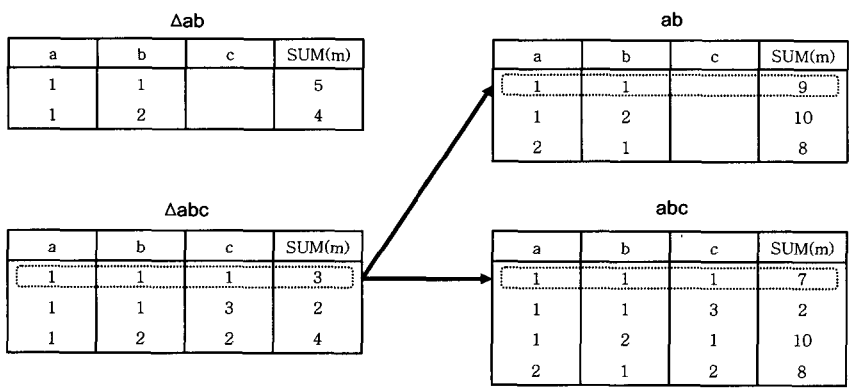


그림 4 Δabc로 abc와 ab를 동시에 갱신하는 방법

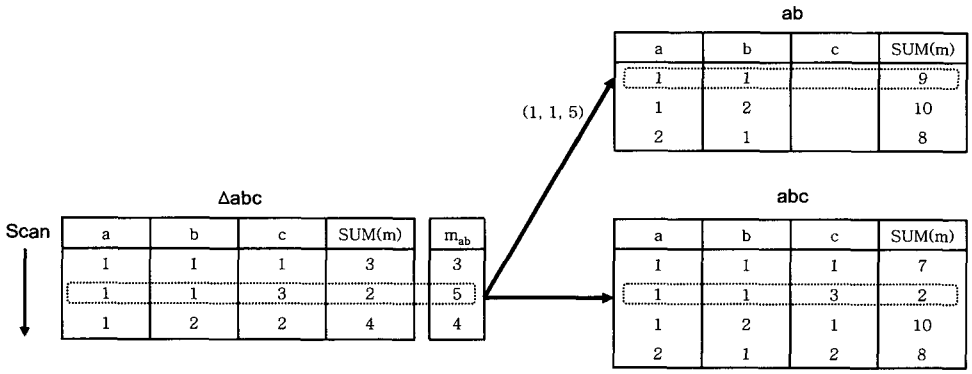


그림 5 ab에 대한 변경 반영 비용의 증가 없이 abc와 ab를 동시에 갱신하는 방법

```
1: Procedure Refresh_abc_and_ab()
2:   m_ab = 0;
3:   For i = 1 to |Δabc| do
4:     Let (a_i, b_i, c_i, m_i) be the ith tuple of Δabc;
5:     Let (a_i+1, b_i+1, c_i+1, m_i+1) be the (i+1)th tuple of Δabc;
6:
7:     m_ab = m_ab + m_i;
8:     Update ((a_i, b_i, c_i, m_i), abc);
9:     If (a_i ≠ a_i+1 or b_i ≠ b_i+1) {Update ((a_i, b_i, m_ab), ab); m_ab = 0;}
10:  End for
11: End procedure
12:
13: Procedure Update(tuple Δt = (d_1, d_2, …, d_n, m), cuboid c)
14:   Find t = (d_1, d_2, …, d_n, m') in c;
15:   If t is not found,
16:     Insert Δt into c;
17:   Else
18:     Update t by (d_1, d_2, …, d_n, m' + m);
19: End procedure
```

그림 6 △abc로 abc와 ab를 동시에 갱신하는 알고리즘

누적된 값 5를 사용하여 (1, 1, 5)로 ab의 튜플 (1, 1, 9)를 갱신한다.

이렇게 Δabc 튜플들의 집단화 값을 누적하다가 Δabc의 현재 튜플이 속한 그룹이 바뀌는 경우에만 ab에 접근하면 Δab를 사용하여 ab를 갱신하는 것과 동일한 효과를 얻을 수 있다. 이 경우 ab에 대한 변경 반영 비용은 Δab를 사용할 때와 동일하다. 따라서, 지금까지 설명한 방법을 사용하면 Δabc를 한번만 스캔하면서도 abc와 ab를 각각에 대한 변경 반영 비용의 증가 없이 동시에 갱신할 수 있다. 그림 6은 지금까지 설명한 방법을 사용하여 Δabc로 abc와 ab를 동시에 갱신하는 알고리즘이다. 그림 6은 Δabc만 가지고도 Δabc와 Δab를 사용할 때와 동일한 비용으로 abc와 ab를 각각 갱신할 수 있도록 해준다. 하지만 그림 6은 Δabc가 정렬되어 있어야 한다는 제약이 있다.

그러나 대부분의 데이타웨어하우스 환경에서는 Δabc를 계산하는 과정에서 큰 부가적인 비용 없이 Δabc의 결과를 정렬시키는 것이 가능하다. 대부분의 관계형 상용 데이타베이스에서는 group-by 연산자가 정렬 알고리즘으로 계산된다[16].[1] Group-by 연산자가 정렬 알고리즘으로 계산되면, 그의 결과는 자동적으로 group-by 애트리뷰트들의 값으로 정렬된다. 특히, SQL 질의에서 order-by 연산자가 group-by 연산자와 같이 사용되는 경우, group-by 연산자는 정렬 알고리즘으로 처리되고 order-by 연산자를 처리하기 위한 부가적인 비용은 거의 들지 않는다[17].

본 논문에서는 모든 group-by 연산자는 정렬 알고리즘으로 계산되며, 변경 큐보이드의 결과는 별도의 추가적인 비용 없이 차원 애트리뷰트의 값에 따라 정렬시킬 수 있다고 가정한다. 또한, 이 때 order-by 연산자 등을 사용하여 차원 애트리뷰트의 정렬 순서를 자유롭게 지정할 수 있다고 가정한다. 예를 들어, Δabc를 계산할 때는 그의 결과를 abc로 정렬할지, cba로 정렬할지를 자유롭게 선택할 수 있으며, 두 경우 모두 Δabc를 계산하는 비용 외에는 추가적인 비용이 들지 않는다. 이와 같은 가정 하에서, 그림 6은 별도의 추가적인 비용 없이 abc와 ab를 갱신하는데 사용될 수 있다.

3.2 전체 큐브 뷰 갱신하기

이제 3.1절에서 설명한 방법을 임의의 변경 큐보이드로 일반화하자. k개의 차원 애트리뷰트 $d_1, d_2, \cdots, d_k$로 이루어진 변경 큐보이드 $\Delta d_1 d_2 \cdots d_k$에 대해 SortOrder ($\Delta d_1 d_2 \cdots d_k$)을 $\Delta d_1 d_2 \cdots d_k$의 차원 애트리뷰트의 정렬

1) 집단화 연산자의 처리에는 정렬 알고리즘 외에 해싱(hashing) 알고리즘이 사용되기도 한다. 자세한 내용은 [16]을 참조하라.

순서라고 하자. 예를 들어, SortOrder(Δabc) = cba이면 Δabc가 애트리뷰트 c, b, a에 따라 정렬되어 있다는 것을 의미한다. 3.1절에서 설명한 방법을 일반화하면, $SortOrder(\Delta d_1 d_2 \cdots d_k) = s_1 s_2 \cdots s_k$ (단, $\{s_1, s_2, \cdots, s_k\} = \{d_1, d_2, \cdots, d_k\}$)인 변경 큐보이드 $\Delta d_1 d_2 \cdots d_k$은 다음의 큐보이드들을 각각에 대한 변경 반영 비용을 증가시키지 않으면서 동시에 갱신할 수 있다.

$\{s_1 s_2 \cdots s_k, s_1 s_2 \cdots s_{k-1}, \cdots, s_1, none\}$

예를 들어, SortOrder(Δabc) = cba이면 Δabc는 각각에 대한 변경 반영 비용을 증가시키지 않으면서 {cba, cb, c, none}를 동시에 갱신할 수 있다. 정렬 순서에 따라 변경 큐보이드가 동시에 갱신할 수 있는 큐보이드의 집합이 달라짐에 유의하라. SortOrder(Δabc) = abc이면 Δabc는 각각에 대한 변경 반영 비용을 증가시키지 않으면서 {abc, ab, a, none}을 동시에 갱신할 수 있다. 여기서 cba와 abc, Δcba와 Δabc는 표기 방법만 다를 뿐, 각각 같은 큐보이드와 같은 변경 큐보이드를 나타냄에 주의하라. 이렇게 SortOrder(Δc) = s인 변경 큐보이드가 각각에 대한 변경 반영 비용의 증가 없이 동시에 갱신할 수 있는 큐보이드의 집합을 UpdatableCuboids(s)라 표시한다.

Δc → $\{c_1, c_2, \cdots, c_i\}$는 변경 큐보이드 Δc로 큐보이드 $c_1, c_2, \cdots, c_i$를 갱신한다는 것을 의미한다. 단, 이 때 SortOrder(Δc) = s라 하면 $\{c_1, c_2, \cdots, c_i\} \subseteq$ UpdatableCuboids(s)이어야 한다. 이것은 모든 변경 큐보이드는 그가 각각에 대한 변경 반영 비용의 증가 없이 동시에 갱신할 수 있는 큐보이드들만 갱신할 수 있다는 것을 의미한다. 예를 들어, SortOrder(Δabc) = cba이면 UpdatableCuboids(cba) = {cba, cb, c, none}이므로 Δabc → {cba, c}는 가능하나 Δabc → {abc, ab}는 불가능하다.

이제 전체 큐브 뷰의 갱신 방법을 알아보자. 1장에서의 큐브 C는 abc, ab, ac, bc, a, b, c, none의 8개의 큐보이드를 가진다. 그림 7(a)는 3개의 변경 큐보이드

```
Δabc → {abc, ab, a, none}   (SortOrder(Δabc) = abc)
Δca → {ca, c}               (SortOrder(Δca) = ca)
Δbc → {bc, b}               (SortOrder(Δbc) = bc)
                    (a)

Δabc → {abc, ac}            (SortOrder(Δabc) = abc)
Δab → {ab, b}               (SortOrder(Δab) = ab)
Δbc → {cb, c}               (SortOrder(Δbc) = cb)
Δa → {a, none}              (SortOrder(Δa) = a)
                    (b)
```

그림 7 C의 갱신 방법 예

Δabc, Δca, Δbc만으로 C의 모든 큐보이드들을 갱신하는 예를 보여준다. 그림 7(a)는 3개의 변경 큐보이드만을 사용하지만, 8개의 변경 큐보이드를 사용할 때와 비교하여 각 큐보이드들에 대한 변경 반영 비용을 증가시키지 않는다. 그림 7(b)는 4개의 변경 큐보이드를 사용하는 예를 보여준다. 이 경우는 4개의 변경 큐보이드만을 사용하지만, 역시 8개의 변경 큐보이드를 사용할 때와 비교하여 각 큐보이드들에 대한 변경 반영 비용을 증가시키지 않는다.

그림 7(a)와 그림 7(b)는 모두 C의 갱신 방법을 나타낸다. 하지만 사용되는 변경 큐보이드나 각 변경 큐보이드가 갱신하는 큐보이드들이 서로 다르다. 이와 같이, 주어진 큐브 뷰에 대한 갱신 방법에는 여러 개가 있을 수 있다. 그러나 제안하는 방법에서는 어떠한 변경 큐보이드들이 사용되든지 간에 각 큐보이드들에 대한 변경 반영 비용은 일정하게 유지된다. 따라서, 큐보이드의 갱신에 사용되는 변경 큐보이드들을 계산하는 비용을 최소화하는 것이 중요하다.

그림 8은 C의 갱신 방법이 주어졌을 때, 해당 갱신 방법에서 사용되는 변경 큐보이드들을 계산하기 위한 계산 계획을 보여준다. 그림 8(a)는 8개의 변경 큐보이드를 모두 사용하는 기존 방법을 위한 변경 큐보이드 계산 계획의 예를 나타낸다. 그림 8(b)와 그림 8(c)는

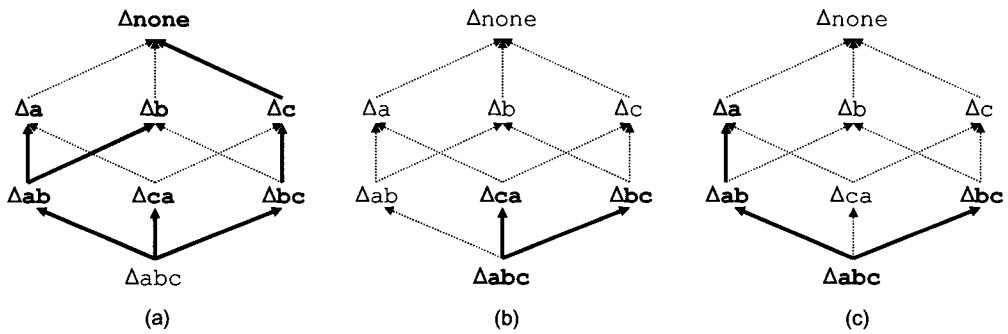


그림 8 변경 큐보이드 계산 계획의 예

각각 그림 7(a)와 그림 7(b)의 갱신 방법을 위한 계산 계획의 예를 나타낸다. 그림 7(a)와 그림 7(b)는 각각 3개와 4개의 변경 큐보이드만을 계산한다. 그림 8(a), 그림 8(b), 그림 8(c)는 서로 다른 비용을 가진다. 큐브 뷰의 갱신 비용을 최소화하기 위해서는 이들 중 최소 비용을 가지는 변경 큐보이드 계산 계획을 찾아야 한다. 다음 장에서는 주어진 큐브 뷰를 위한 최소 비용의 변경 큐보이드 계산 계획을 찾는 문제를 논의한다.

4. 문제 정의 및 해결 방법

4.1 문제 정의

n개의 차원 애트리뷰트의 집합 $D = \{d_1, d_2, \cdots, d_n\}$에 대해 $C = \{c \mid c \text{ is a subset of } D\}$는 D에 대해 정의된 큐브를 나타낸다. 큐브의 각 원소를 큐보이드라 부른다. 편의를 위해 큐보이드 $\{d_i, d_j, \cdots, d_k\}$를 $d_id_j\cdots d_k$로 간단히 표시할 수 있다. 특히, 큐보이드 {}는 none이라고 표시한다. 큐브 C에 대해 $\Delta C = \{\Delta c \mid c \in C\}$는 C의 변경 큐브를 나타낸다. 변경 큐브의 원소 Δc를 변경 큐보이드라 부르며, Δc는 큐보이드 c의 변경을 의미한다. $numDim(\Delta c)$는 변경 큐보이드 Δc의 차원 애트리뷰트의 개수를 나타낸다. $c_i \supset c_j$인 변경 큐보이드 Δc_i와 Δc_j에 대해 $Cost(\Delta c_i, \Delta c_j)$는 Δc_i로부터 Δc_j를 구하는 비용을 나타낸다. 본 논문에서는 $Cost(\Delta c_i, \Delta c_j)$는 주어져 있다고 가정한다. $Cost(\Delta c_i, \Delta c_j)$를 구하는 방법은 [10,18]을 참조하라.

정의 1 (변경 격자 그래프). 주어진 변경 큐브 ΔC에 대해, *변경 격자 그래프(delta lattice graph)* $G = (V, E, W)$는 다음과 같이 정의되는 유향 비순환 그래프(directed acyclic graph)이다.

- $V = \Delta C$
- $E = \{(\Delta c_i, \Delta c_j) \mid \Delta c_i, \Delta c_j \in V, c_i \supset c_j, numDim(\Delta c_i) = numDim(\Delta c_j) + 1\}$
- $W((\Delta c_i, \Delta c_j)) = Cost(\Delta c_i, \Delta c_j)$ for $(\Delta c_i, \Delta c_j) \in E$ □

정의 2 (갱신 체인). 주어진 변경 큐브 **ΔC**에 대해, 다음을 만족하는 시퀀스(sequence) $\{\Delta c_1, \Delta c_2, \cdots, \Delta c_i\}$를 ΔC의 갱신 체인이라 한다.

- $\Delta c_1, \Delta c_2, \cdots, \Delta c_i \in \Delta C$
- $c_1 \supset c_2 \supset \cdots \supset c_i$

갱신 체인 $s = \{\Delta c_1, \Delta c_2, \cdots, \Delta c_i\}$의 첫 원소를 커버 변경 큐보이드(cover delta cuboid)라 부르고 $h(s)$로 표시한다. 즉, $h(s) = \Delta c_1$이다. □

갱신 체인 $\{\Delta c_1, \Delta c_2, \cdots, \Delta c_i\}$는 $\Delta c_1 \rightarrow \{c_1, c_2, \cdots, c_i\}$를 의미한다. 즉, Δc_1으로 $c_1, c_2, \cdots, c_i$를 갱신한다는 것을 의미한다. 그러나 $\Delta c_1 \rightarrow \{c_1, c_2, \cdots, c_i\}$이 가능하기 위해서는 $\{c_1, c_2, \cdots, c_i\} \subseteq UpdatableCuboids(s)$가 되도록 하는 $\Delta c1$의 적절한 정렬 순서 s가 존재해야 한다. 다음은 임의의 갱신 체인 $\{\Delta c_1, \Delta c_2, \cdots, \Delta c_i\}$가 주어졌을 때, $\Delta c_1 \rightarrow \{c_1, c_2, \cdots, c_i\}$를 가능하게 하는 Δc_1의 정렬 순서를 구하는 방법을 설명한다.

보조 정리 1. 임의의 갱신 체인 $\{\Delta c_1, \Delta c_2, \cdots, \Delta c_i\}$에 대해, $\Delta c_1 \rightarrow \{c_1, c_2, \cdots, c_i\}$를 가능하게 하는 $SortOrder(\Delta c_1)$이 항상 존재한다.

증명. 갱신 체인의 정의에 따라 일반성의 손실 없이 $\Delta c_1 = \Delta d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z}$, $\Delta c_2 = \Delta d_i\cdots d_{i+x}\cdots d_{i+y}$, $\cdots$, $\Delta c_i = \Delta d_i\cdots d_{i+x}$라고 나타내자. 본 증명에서는 $SortOrder(\Delta c_1) = d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z}$이면 $\{c_1, c_2, \cdots, c_i\} \subseteq UpdatableCuboid(d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z})$임을 보인다. $c_1 = d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z}$, $c_2 = d_i\cdots d_{i+x}\cdots d_{i+y}$, $\cdots$, $c_i = d_i\cdots d_{i+x}$이고, 정의에 따라 $UpdatableCuboid(d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z}) = \{d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z}, \cdots, d_i\cdots d_{i+x}\cdots d_{i+y}, \cdots, d_i\cdots d_{i+x}, \cdots, d_i\}$이다. 따라서 $\{c_1, c_2, \cdots, c_i\} \subseteq UpdatableCuboid(d_i\cdots d_{i+x}\cdots d_{i+y}\cdots d_{i+z})$이다. □

예를 들어 갱신 체인 {Δabc, Δbc, Δc}에 대해, Δabc → {abc, bc, c}를 가능하게 하는 SortOrder(Δabc)는 cba이다. 보조 정리 1에 따라 임의의 갱신 체인 $\{\Delta c_1, \Delta c_2, \cdots, \Delta c_i\}$에 대해 항상 $\Delta c_1 \rightarrow \{c_1, c_2, \cdots, c_i\}$가 가능함을 알 수 있다.

정의 3 (갱신 파티션). 주어진 변경 큐브 ΔC에 대해, ΔC를 상호 배타적인(disjoint) 갱신 체인들로 나눈 파티션을 ΔC의 갱신 파티션이라 한다. □

예제 1. 변경 큐브 ΔC = {Δabc, Δab, Δac, Δbc, Δa, Δb, Δc, Δnone}에 대해, {{Δabc, Δab, Δa, Δnone}, {Δca, Δc}, {Δbc, Δb}}는 3개의 갱신 체인으로 이루어진 갱신 파티션을 나타낸다. 한편, {{Δabc, Δac}, {Δab, Δb}, {Δcb, Δc}, {Δa, Δnone}}는 4개의 갱신 체인으로 이루어진 다른 갱신 파티션을 나타낸다. 이 두 갱신 파티션은 각각 그림 7(a)과 그림 7(b)의 갱신 계획을 나타낸다. □

갱신 파티션은 큐브 뷰 전체에 대한 갱신 계획을 나타낸다. 갱신 파티션 $\{s_1, s_2, \cdots, s_k\}$에 따라 큐브 뷰를 갱신하기 위해서는 갱신 파티션에 포함된 각 갱신 체인의 커버 변경 큐보이드들, 즉, $h(s_1), h(s_2), \cdots, h(s_k)$를 계산해야 한다. 3.2절에서 설명한 바와 같이, 어떤 커버 변경 큐보이드들을 사용하든지 각 큐보이드들에 대한 변경 반영 비용은 일정하게 유지된다. 따라서 갱신 파티션의 비용은 $h(s_1), h(s_2), \cdots, h(s_k)$를 계산하는 비용으로 나타낼 수 있다.

정의 4 (변경 큐보이드 계산 계획). 갱신 파티션 $\{s_1, s_2, \cdots, s_k\}$를 위한 변경 큐보이드 계산 계획이란

$h(s_1)$, $h(s_2)$, …, $h(s_k)$를 포함하는 변경 격자 그래프의 한 부트리(subtree)를 말한다. □

정의 5 (변경 큐보이드 계산 계획의 비용). 변경 큐보이드 계산 계획 T의 비용은 T에 속한 모든 간선들의 가중치의 합이다. 이 때 T의 비용을 Cost(T)로 표시한다. □

주어진 갱신 파티션에 대한 변경 큐보이드 계산 계획에는 여러 가지가 있을 수 있다. 갱신 파티션의 비용을 최소화하기 위해서는 이들 중 최소 비용을 가지는 변경 큐보이드 계산 계획을 찾아야 한다. 이제 문제를 다음과 같이 정의하자.

문제 정의. 변경 큐브와 그의 변경 격자 그래프가 주어졌을 때, Cost(T)를 최소화하는 갱신 파티션과 그의 변경 큐보이드 계산 계획 T를 찾아라. □

이렇게 정의된 문제를 *Optimal Cube Maintenance Plan(OCMP)* 문제라 부른다. 다음 절에서는 OCMP 문제의 해를 찾는 방법에 대해 논의한다.

4.2 알고리즘

주어진 갱신 파티션에 대해, 최소 비용의 변경 큐보이드 계산 계획을 찾는 문제는 NP-complete이다. 주어진 갱신 파티션에 대한 최소 비용의 변경 큐보이드 계산 계획을 찾기 위해서는 변경 격자 그래프에서 갱신 파티션의 커버 변경 큐보이드들을 포함하는 최소 비용의 부트리를 찾아야 한다. 이 문제는 그래프에서 주어진 정점(vertex)들을 포함하는 최소 비용의 부트리를 찾는 *최소 비용 스타이너 트리 문제(minimum steiner tree problem)*[19]과 같다. 최소 비용 스타이너 트리 문제는 큐브 그래프와 같은 특수한 형태의 그래프에 대해서도 NP-complete임이 알려져 있다[20]. OCMP 문제는 최소한 최소 비용 스타이너 트리 문제보다 어려운 문제다. 따라서 OCMP 문제는 NP-hard에 속한다. 더욱이, 주어진 변경 큐브에 대해 가능한 갱신 파티션의 개수는 변경 큐보이드의 수에 대해 지수적으로 증가한다. 따라서 본 논문에서는 OCMP에 대한 최적 해를 찾는 알고리즘 대신 휴리스틱 알고리즘을 제안한다.

갱신 파티션은 다양한 수의 갱신 체인을 포함할 수 있다. 예제 1은 갱신 체인이 3개와 4개인 갱신 파티션의 예를 보여준다. 갱신 체인의 수는 계산해야 하는 커버 변경 큐보이드의 수를 의미한다. 두 개의 갱신 파티션 $S = \{s_1, s_2, \cdots, s_i\}$와 $P = \{p_1, p_2, \cdots, p_j\}$가 주어졌다고 하자. 이 때, S와 P 사이에 $\{h(s_1), h(s_2), \cdots, h(s_i)\} \supset \{h(p_1), h(p_2), \cdots, h(p_j)\}$의 관계가 성립한다고 하자. 이 경우 S에 포함된 갱신 체인의 수보다 P에 포함된 갱신 체인의 수가 적다. 이 때, S와 P에 대한 최소 비용 변경 큐보이드 계산 계획을 각각 T_s와 T_p라고 하면 변경 큐보이드 계산 계획의 정의에 따라 $Cost(T_s) > Cost(T_p)$가 성립한다. 따라서 변경 큐보이드 계산 계획의 비용을 줄이기 위해서는 갱신 체인의 수를 최소화하는 것이 바람직하다. 본 논문에서 제안하는 휴리스틱은 최소한의 갱신 체인만을 포함하는 갱신 파티션을 찾는다.

보조 정리 2. 2^n개의 변경 큐보이드로 이루어진 변경 큐브의 갱신 파티션에는 최소한 ${}_nC_{\lceil n/2 \rceil}$개 이상의 갱신 체인이 존재한다.

증명. 변경 큐브에는 차원 애트리뷰트의 개수가 $\lceil n/2 \rceil$ 변경 큐보이드가 정확히 ${}_nC_{\lceil n/2 \rceil}$개 존재한다. 갱신 파티션의 정의에 따라 이들 중 어떤 두 변경 큐보이드도 같은 갱신 체인에 존재할 수 없다. 따라서, 갱신 파티션에는 최소한 ${}_nC_{\lceil n/2 \rceil}$개의 갱신 체인이 존재한다. □

본 논문에서 제안하는 휴리스틱은 정확히 ${}_nC_{\lceil n/2 \rceil}$개의 갱신 체인을 가지는 갱신 파티션을 찾는다. 이에 따라 제안하는 방법은 2^n개의 변경 큐보이드 대신 ${}_nC_{\lceil n/2 \rceil}$개의 변경 큐보이드만 계산하면 된다는 것을 보장한다.

제안하는 방법은 변경 큐브의 각 변경 큐보이드들이 각각 하나의 갱신 체인을 형성하는, 2^n개의 갱신 체인으로 이루어진 갱신 파티션으로부터 출발한다. 이 갱신 파티션은 2^n개의 변경 큐보이드를 모두 사용하는 기존의 방법을 나타냄을 상기하라. 이 갱신 파티션에 대한 최소 비용의 변경 큐보이드 계산 계획은 변경 격자 그래프 전체에 대한 최소 비용 신장 트리(minimum spanning tree, MST)가 된다. 제안하는 방법은 이 MST가 나타내는 변경 큐보이드 계산 계획으로부터 시작하여 변경 큐보이드 계산 계획의 단말 노드들을 단계적으로 제거해 나간다. 단말 노드가 제거되었다는 것은 단말 노드가 포함된 갱신 체인이 다른 갱신 체인과 합병되었다는 것을 의미한다. 단말 노드들이 제거될 때마다 변경 큐보이드 계산 계획의 비용이 줄어든다. 이러한 단말 노드의 제거는 변경 큐보이드 계산 계획에 정확히 ${}_nC_{\lceil n/2 \rceil}$개의 노드만 남을 때까지 진행된다.

변경 격자 그래프 G가 주어졌다고 하자. 제안하는 방법은 먼저 G의 MST T를 구한다. 변경 격자 그래프에서 차원 애트리뷰트의 수가 i개인 변경 큐보이드들의 집합을 Level(i)로 표시하자. 제안하는 방법은 Level(0)부터 시작하여 Level(n - 1)으로 진행하면서 T의 단말 노드들을 단계적으로 제거해 나간다. 각 Level(i)에 대해, 제안하는 방법은 Level(i)에 포함된 T의 단말 노드와 Level(i + 1)의 변경 큐보이드들간의 최대 비용 이분할 부합(maximum weight bipartite matching)[2][21]을 찾는다. 찾아진 이분할 부

2) 최대 비용 이분할 부합 문제는 다음과 같이 정의된다: 두 상호 배타적인 정점의 집합 V_1, V_2와 V_1과 V_2를 연결하는 간선의 집합 E를 가지는 그래프가 주어졌다고 하자. 각 간선에는 고정된 가중치가 주어져 있다. 최

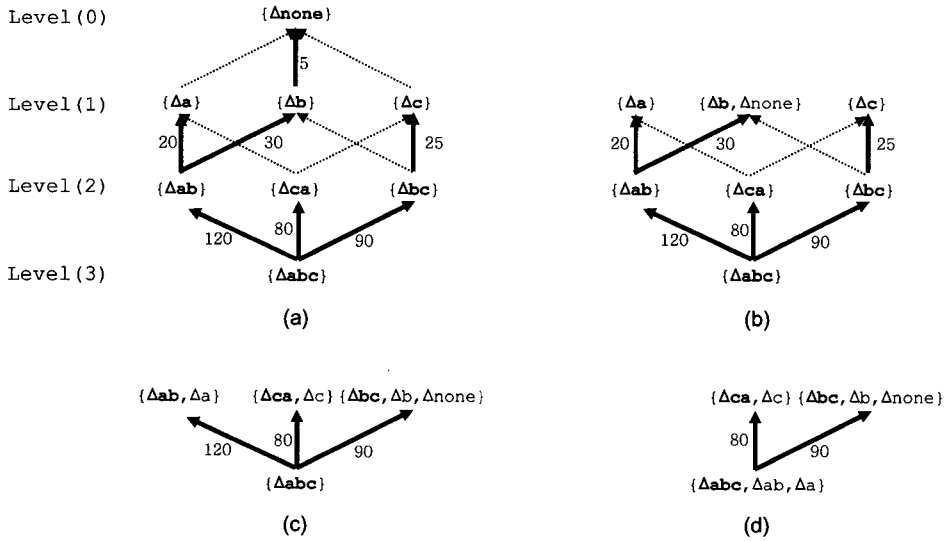


그림 9 제안하는 방법의 수행 예

합에서 Level(i)에 포함된 T의 단말 노드 Δc_i가 Level(i + 1)의 변경 큐보이드 Δc_{i+1}과 부합되었다고 하자. 그러면 T에서 노드 Δc_i와, Δc_i로 향하는 간선 $(\Delta c_{i+1}', \Delta c_i)$를 제거한다. 그리고 Δc_i를 포함하는 갱신 체인과 Δc_{i+1}를 포함하는 갱신 체인을 합병한다. 이 때 부합 $(\Delta c_{i+1}, \Delta c_i)$가 반드시 T의 간선일 필요가 없으므로 T에서 제거되는 간선은 $(\Delta c_{i+1}, \Delta c_i)$가 아닌 $(\Delta c_{i+1}', \Delta c_i)$임에 유의하라. 그러므로 Δc_i와 Δc_{i+1}의 부합으로 얻어지는 이득은 Cost$(\Delta c_{i+1}, \Delta c_i)$가 아닌 Cost$(\Delta c_{i+1}', \Delta c_i)$가 된다. 최대 비용 이분할 부합은 이러한 이득을 최대화한다.

그러나 Δc_{i+1}과 Δc_i의 부합으로 얻는 이득은 Cost$(\Delta c_{i+1}, \Delta c_i)$가 아니라 Cost$(\Delta c_{i+1}', \Delta c_i)$이다. 따라서, 최대 비용 이분할 부합을 찾기 전에 Level(i + 1)과 Level(i)간의 간선들의 비용을 수정해 주어야 한다. 즉, 각 간선 $(\Delta c_{i+1}, \Delta c_i)$에 대해 Cost$(\Delta c_{i+1}, \Delta c_i)$를 Cost$(\Delta c_{i+1}', \Delta c_i)$로 수정한다. 단, $(\Delta c_{i+1}', \Delta c_i)$는 Δc_i로 향하는 T의 간선이다.

그림 9는 지금까지 설명한 방법으로 갱신 파티션과 그에 대한 변경 큐보이드 계산 계획을 구하는 한 예를 보여준다. 그림 9(a)는 각 변경 큐보이드가 하나의 갱신 체인을 형성하는, 즉, 2^n개의 갱신 체인으로 이루어진 갱신 파티션에 대한 변경 큐보이드 계산 계획의 예를 나타낸다. 굵게 표시된 변경 큐보이드는 각 갱신 체인의 커버 변경 큐보이드를 나타내며, 각 숫자는 해당 간선의 가중치를 나타낸다. 여기에서는 변경 큐보이드 계산 계획에 속하는 간선들의 가중치만을 표시하였다. 그림 9(b)는 Level(0)과 Level(1)간의 부합이 이루어진 후의 상태를 보여준다. 그림 9(b)에서는 갱신 체인 {Δnone}과 갱신 체인 {Δb}가 부합되어 갱신 체인 {Δb, Δnone}으로 합병되었으며, 그로 인해 변경 큐보이드 계산 계획의 비용이 5만큼 감소하였다. 간선의 가중치에 따라 이 외에도 여러 가지 방법으로 부합이 이루어질 수 있음에 유의하라. 그림 9(c)는 Level(1)과 Level(2)간의 부합이 이루어진 후의 상태를 보여준다. 그림 9(c)에서는 {Δab}와 {Δa}가 부합되어 {Δab, Δa}로, {Δbc}와 {Δb, Δnone}이 부합되어 {Δbc, Δb, Δnone}로, 그리고 {Δca}와 {Δc}가 부합되어 {Δca, Δc}로 합병되었다. 그에 따라 변경 큐보이드 계산 계획의 비용은 20 + 30 + 25 = 75만큼 감소하였다. 마지막으로, 그림 9(d)에서는 {Δabc}와 {Δab, Δa}이 부합되어 {Δabc, Δab, Δa}로 합병되었으며, 그에 따라 변경 큐보이드 계산 계획의 비용은 120만큼 감소하였다. 이러한 합병의 결과는 정확히 ${}_nC_{\lceil n/2 \rceil}$개의 갱신 체인을 가지는 갱신 파티션과 그에 대한 변경 큐보이드 계산 계획을 나타낸다. 그림 10의 *HeuristicCubeMaintenancePlan* 프로시저는 지금까지 설명한 알고리즘을 나타낸다.

4.3 알고리즘 분석

보조 정리 3. 2^n개의 변경 큐보이드를 포함하는 변경 격자 그래프가 주어졌을 때, *HeuristicCubeMaintenancePlan*은 정확히 ${}_nC_{\lceil n/2 \rceil}$개의 갱신 체인을 포함하

대 비용 이분할 부합이란 가중치의 합이 최대가 되는 E의 부분 집합으로서, 선택된 부그래프(subgraph)에서 V_1의 각 정점은 많아야 V_2의 한 정점으로 연결된다(V_2에서 V_1으로도 마찬가지이다).

```
Procedure HeuristicCubeMaintenancePlan(Input: delta lattice G = (V, E, W))
  Let n be the number of the dimension attributes of G;
  Let T = (V', E') be the minimum spanning tree of G;

  For i = 0 to n - 1 do
    Level(i)' = The leaf nodes of T that are included in Level(i);

    For each edge e = (Δc_{i+1}, Δc_i) between Level(i + 1) and Level(i)' do
      Assign Cost(Δc_{i+1}', Δc_i) to e where (Δc_{i+1}', Δc_i) is an edge in E';

    Find the maximum weight bipartite matching M between Level(i + 1)
    and Level(i)';

    For each match m = (Δc_{i+1}, Δc_i) ∈ M do
      Delete Δci from V';
      Delete (Δc_{i+1}', Δc_i) from E';
End procedure
```

그림 10 휴리스틱 알고리즘

는 갱신 파티션을 찾는다.

증명. 본 증명에서는 *HeuristicCubeMaintenancePlan*으로 찾아진 갱신 파티션에서 임의의 변경 큐보이드 Δc는 반드시 어떤 변경 큐보이드 $\Delta c' \in Level(\lceil n/2 \rceil)$와 같은 갱신 체인에 포함됨을 보인다. $Level(\lceil n/2 \rceil)$에 포함되어 있는 변경 큐보이드의 개수는 ${}_nC_{\lceil n/2 \rceil}$개이고 이들은 같은 갱신 체인에 포함될 수 없으므로, 이들을 포함하고 있는 갱신 체인의 개수는 정확히 ${}_nC_{\lceil n/2 \rceil}$개이다. 따라서 위 가설이 증명되면 이것은 *HeuristicCubeMaintenancePlan*으로 찾아진 갱신 파티션에 포함된 갱신 체인의 수가 ${}_nC_{\lceil n/2 \rceil}$개를 넘지 않는다는 것이 증명된다.

1) $\Delta c \in Level(\lceil n/2 \rceil)$일 경우: Δc가 $\Delta c'$에 해당하므로 명백하다.

2) $\Delta c \in Level(i)$, 단, $i < \lceil n/2 \rceil$일 경우:

$Level(i), Level(i + 1), \cdots, Level(\lceil n/2 \rceil)$를 고려하자. $|Level(i)| \leq |Level(i + 1)| \leq \cdots \leq |Level(\lceil n/2 \rceil)|$이므로 큐브 그래프의 최대 비용 이분할 부합에서 $\Delta c \in Level(i)$는 반드시 어떤 $\Delta c_{i+1} \in Level(i + 1)$과 부합된다. 또한, $\Delta c_{i+1} \in Level(i + 1)$은 반드시 어떤 $\Delta c_{i+2} \in Level(i + 2)$과 부합된다. 이와 같이 $\Delta c_{\lceil n/2 \rceil - 1} \in Level(\lceil n/2 \rceil - 1)$은 반드시 어떤 $\Delta c_{\lceil n/2 \rceil} \in Level(\lceil n/2 \rceil)$과 부합된다. *HeuristicCubeMaintenancePlan*에 따라 이렇게 부합된 변경 큐보이드들은 모두 하나의 갱신 체인에 포함된다. 따라서 $\Delta c \in Level(i)$는 반드시 어떤 $\Delta c_{\lceil n/2 \rceil} \in Level(\lceil n/2 \rceil)$과 같은 갱신 체인에 포함된다.

3) $\Delta c \in Level(i)$, 단, $i > \lceil n/2 \rceil$일 경우:

$Level(i), Level(i - 1), \cdots, Level(\lceil n/2 \rceil)$를 고려하자. 그러면 위와 동일한 방법으로 $\Delta c \in Level(i)$는 반드시 어떤 $\Delta c_{\lceil n/2 \rceil} \in Level(\lceil n/2 \rceil)$과 같은 갱신 체인에 포함된다는 것을 보일 수 있다.

위 증명에 따라, *HeuristicCubeMaintenancePlan*으로 찾아진 갱신 파티션에 포함된 갱신 체인의 수는 ${}_nC_{\lceil n/2 \rceil}$개를 넘지 않는다. 보조 정리 2에 따라 어떤 갱신 파티션도 ${}_nC_{\lceil n/2 \rceil}$개 미만의 갱신 체인을 가질 수 없다. 따라서 *HeuristicCubeMaintenancePlan*으로 찾아진 갱신 파티션에 포함된 갱신 체인의 수는 정확히 ${}_nC_{\lceil n/2 \rceil}$개이다. □

보조 정리 3에 따라, 제안하는 방법은 큐브 뷰를 갱신하는데 2^n개의 변경 큐보이드대신 ${}_nC_{\lceil n/2 \rceil}$개의 변경 큐보이드만을 계산하면 된다는 것을 보장한다. 이에 따라 제안하는 방법은 변경 큐보이드를 계산하는 데 드는 비용을 크게 줄일 수 있다.

보조 정리 4. 2^n개의 변경 큐보이드를 포함하는 변경

격자 그래프 G가 주어졌다고 하자. G에서 Level(0), Level(1), …, Level(⌈n/2⌉)를 제거한 G의 부그래프(subgraph)를 G/2라고 하자. G와 G/2의 MST를 각각 T_G, $T_{G/2}$라고 하고, *HeuristicCubeMaintenancePlan*이 찾은 변경 큐보이드 계산 계획을 T라고 하면 다음이 성립한다. 단, $G \supset G'$는 G'가 G의 부그래프임을 뜻한다.

$$T_G \supset T_{G/2} \supset T$$

이에 따라 다음이 성립한다.

$$Cost(T_G) > Cost(T_{G/2}) > Cost(T)$$

증명. $T_G \supset T_{G/2}$는 명백하므로 $T_{G/2} \supset T$를 보인다. 보조 정리 3에 따라, *HeuristicCubeMaintenancePlan*으로 찾아진 갱신 파티션의 모든 갱신 체인에는 반드시 어떤 $\Delta c \in Level(\lceil n/2 \rceil)$가 포함된다. 따라서 Level(0), Level(1), …, Level(⌈n/2⌉)에 속한 변경 큐보이드는 *HeuristicCubeMaintenancePlan*으로 찾아진 어떤 갱신 체인의 커버 변경 큐보이드도 될 수 없다. *HeuristicCubeMaintenancePlan*에 따라 T에는 커버 변경 큐보이드만 포함된다. 따라서, T에는 Level(0), Level(1), …, Level(⌈n/2⌉)에 속한 변경 큐보이드들은 포함될 수 없다. 한편, T는 T_G에서 단말 노드를 제거해서 만들어지는 그래프이므로 T_G의 부그래프임이 분명하다. 따라서 T는 Level(0), Level(1), …, Level(⌈n/2⌉)의 변경 큐보이드들을 포함하지 않는 T_G의 부그래프이므로 $T_{G/2} \supset T$임이 성립된다. □

보조 정리 4는 제안하는 방법이 2^n개의 모든 변경 큐보이드들을 계산하는 기존의 방법보다 적은 비용을 보장한다는 것을 의미한다. 또한 보조 정리 4에 따라 제안하는 방법은 전체의 절반 이하의 변경 큐보이드들, 즉, Level(⌈n/2⌉), Level(⌈n/2⌉+1), …, Level(n)에 속하는 변경 큐보이드들만을 계산하는 비용보다도 더 적은 비용을 보장한다.

5. 성능 평가

본 장에서는 제안하는 방법에 대한 성능 평과 결과를 보인다. 본 장에서는 제안하는 방법과 2^n개의 변경 큐보이드를 모두 다 사용하여 큐브를 갱신하는 기존의 방법을 비교한다. 각 방법의 성능은 해당 방법을 사용하여 큐브 뷰를 갱신하는데 걸리는 시간으로 측정하였다. 실험 환경으로는 750MHz UltraSPARC III CPU와 512MB의 메모리가 장착된 Sun Blade 1000 워크스테이션에 설치된 Oracle9*i*이 사용되었다. 실험에 사용된 테이블들의 스키마와 데이타는 TPC-H 벤치마크의 스키마와 데이타[22]를 사용하였으며, 큐브의 점진적 관리 방법들은 Oracle9*i*의 PL/SQL을 기반으로 구현되었다.

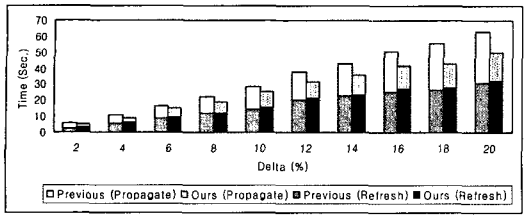
본 실험에서는 TPC-H 스키마의 lineitem 테이블에 대해 큐브 뷰를 정의하였다. lineitem 테이블은 TPC-H 스키마에서 사실 테이블(fact table)의 역할을 하는 테이블이다. 큐브 뷰의 차원 애트리뷰트들로는 lineitem 테이블의 l_orderkey, l_partkey, l_suppkey, l_shipdate, l_receiptdate 애트리뷰트들이 사용되었으며, 측정 애트리뷰트로는 l_quantity 애트리뷰트가 사용되었다. 실험에서는 표 1과 같은 큐브 뷰 C_1, C_2, C_3를 정의하였다.

표 1 실험에 사용된 큐브 뷰

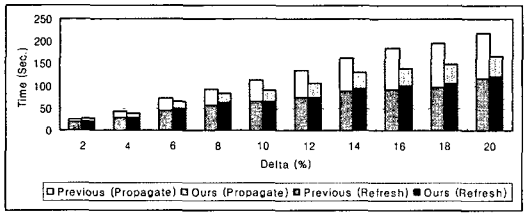
큐브	차원 애트리뷰트	측정 애트리뷰트
C_1	l_orderkey, l_partkey, l_suppkey	l_quantity
C_2	l_orderkey, l_partkey, l_suppkey, l_shipdate	l_quantity
C_3	l_orderkey, l_partkey, l_suppkey, l_shipdate, l_receiptdate	l_quantity

실험에서는 lineitem 테이블에 대한 변경의 크기를 원 테이블의 크기에 대해 2%에서 20%까지 변화시키면서 각 방법의 성능을 측정하였다. lineitem 테이블에 대한 변경은 lineitem 테이블에 새로운 튜플들을 삽입함으로써 이루어졌다. 새로운 튜플들의 삽입으로 인한 차원 테이블의 변경은 데이타 웨어하우스 환경에서 매우 중요한 부분을 차지한다. 대부분의 데이타 웨어하우스 애플리케이션에서 사실 테이블에 대한 변경은 새로운 날짜에 발생한 새로운 데이타의 삽입에 따라 발생하기 때문이다[1].

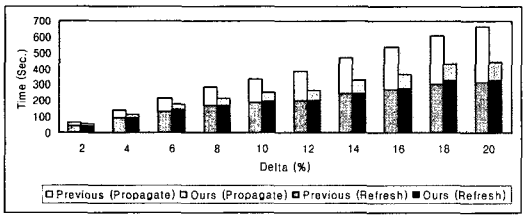
그림 11은 고정된 lineitem 테이블의 크기(약 600,000개 튜플)에 대해, 발생한 변경의 크기를 2%에서 20%로 증가시키며 두 방법의 성능을 평가한 결과를 나타낸다. *Ours*는 제안하는 방법을 나타내며, *Previous*는 2^n개의 변경 큐보이드를 모두 사용하는 기존의 방법을 나타낸다. 동일한 실험이 C_1, C_2, C_3에 대해 수행되었다. 2.2절에서 언급한 바와 같이, 큐브 뷰의 갱신은 변경 계산 단계와 변경 반영 단계로 나뉜다. 변경 계산 단계는 변경 큐보이드들을 계산하는 단계이며, 변경 반영 단계는 계산된 변경 큐보이드들을 사용하여 큐보이드들을 갱신하는 단계이다. 그림 11에서 *Ours(Propagate)*와 *Ours(Refresh)*는 각각 제안하는 방법에서의 변경 계산 단계와 변경 반영 단계의 시간을 나타내며, *Previous (Propagate)*와 *Previous(Refresh)*는 각각 기존 방법에서의 변경 계산 단계와 변경 반영 단계의 시간을 나타낸다. 예상한 바와 같이, 제안하는 방법은 ${}_nC_{\lceil n/2 \rceil}$개의 변경 큐보이드만을 계산하므로 변경 계산 단계의 시간



(a) C_1

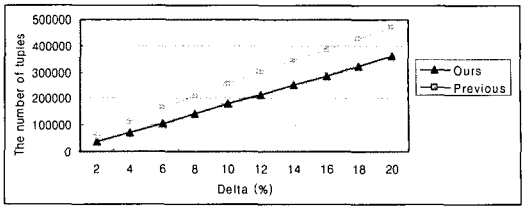


(b) C_2

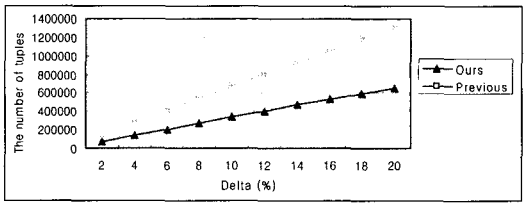


(c) C_3

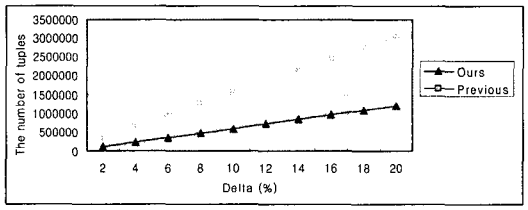
그림 11 변경의 크기를 변화시켰을 때



(a) C_1



(b) C_2



(c) C_3

그림 12 계산하는 튜플 수 비교

이 줄어들었지만 큐보이드들의 변경 반영 단계에 걸리는 시간은 2^n개의 변경 큐보이드들을 모두 사용할 때와 거의 차이가 나지 않음을 볼 수 있다. 다만 제안하는 방법에서 변경 반영 단계의 시간이 조금 더 걸리는 이유는, 변경 큐보이드의 매 튜플마다 집단화 값을 누적하는 비용과 현재 튜플이 속해 있는 그룹이 이전 튜플이 속해 있는 그룹과 다른지를 비교하는 비용이 추가적으로 들기 때문이라 판단된다. 그러나 이러한 추가 비용은 전체적인 성능에 큰 영향을 미치지 않는다. 특히, 제안하는 방법은 큐브 뷰의 차원 애트리뷰트가 증가함에 따라 2^n개의 변경 큐보이드를 모두 계산해야 하는 기존의 방법보다 더 좋은 성능을 보이고 있음을 알 수 있다.

변경 계산 단계에서의 성능 차이를 좀 더 자세히 분석하기 위해, 그림 12에서는 제안하는 방법과 기존의 방법이 변경 계산 단계에서 계산하는 변경 큐보이드의 튜플 수를 비교하였다. 실험 환경은 그림 11과 동일하다. 계산되는 튜플의 수는 각 방법이 요구하는 계산량 및 디스크 I/O 양과 밀접한 관계가 있다. 그림 12에서 볼 수 있듯이, 제안하는 방법은 기존 방법에 비해 적은 수의 튜플들만을 계산한다. 그림 12의 비교를 통해서도, 제안하는 방법이 기존의 방법에 비해 좋은 성능을 보일 수 있음을 알 수 있다.

6. 결 론

본 논문에서는 큐브 뷰의 효율적인 점진적 관리 기법을 제안하였다. 차원 애트리뷰트의 수가 n인, 2^n개의 큐보이드로 이루어진 큐브 뷰를 갱신하기 위해 기존의 방법은 2^n개의 변경 큐보이드를 사용한다. 제안하는 방법은 정렬된 변경 큐보이드로 여러 개의 큐보이드를 동시에 갱신함으로써, 2^n개가 아닌 일부의 변경 큐보이드만을 사용하고도 각 큐보이드에 대한 변경 반영 비용의 증가 없이 모든 큐보이드들을 갱신할 수 있게 해준다. 본 논문은 이러한 변경 큐보이드들 중 큐브 뷰의 갱신 비용을 최소화하는 최적의 변경 큐보이드들을 찾기 위한 Optimal Cube Maintenance Plan(OCMP) 문제를 정의하고, 그에 대한 휴리스틱 알고리즘을 제안하였다. 제안하는 휴리스틱은 2^n개의 변경 큐보이드 대신 ${}_nC_{\lceil n/2 \rceil}$개의 변경 큐보이드만을 사용하여 큐브 뷰를 갱신할 수 있음을 보장한다. 또한, 제안하는 휴리스틱은 2^n개의 변경 큐보이드를 사용하는 기존 방법보다 항상 적은 비용을 가지고 있음을 보장한다. 본 논문에서는 제안하는 방법을 구현하고, 실험을 통해 기존 방법과 성능을 비교하

였다. 실험 결과를 통해 제안하는 방법은 기존의 방법보다 좋은 성능을 보이고 있음을 알 수 있었다. 특히 큐브 뷰의 차원 애트리뷰트의 수가 증가할수록 성능 상의 이득이 커짐을 기대할 수 있었다.

참 고 문 헌

[1] I. S. Mumick, D. Quass, B. S. Mumick, Maintenance of Data Cubes and Summary Tables in a Warehouse, In Proceedings of the ACM SIGMOD Conference, 1997.

[2] Nick Roussopoulos, Yannis Kotidis, Mema Roussopoulos, Cubetree: Organization of and Bulk Incremental Updates on the Data Cube, In Proceddings of the ACM SIGMOD Conference, 1997.

[3] C. A. Hurtado, A. O. Mendelzon, A. A. Vaisman, Maintaining Data Cubes under Dimension Updates, In Proceedings of ICDE, 1999.

[4] W. Lehner, R. Sidle, H. Pirahesh, R. Cochrane, Maintenance of Cube Automatic Summary Tables, In Proceedings of the ACM SIGMOD Conference, 2000.

[5] A. Gupta, I. S. Mumick, V. S. Subrahmanian, Maintaining views incrementally, In Proceedings of the ACM SIGMOD Conference, pp. 157-166, 1993.

[6] D. Quass, Maintenance expressions for views with aggregation, In Workshop on Materialized Views: Techniques and Applications, pp. 110-118, 1996.

[7] H. Gupta, I.S. Mumick, Incremental maintenance of aggregate and outerjoin expressions, Technical Report, Stanford University, 1999.

[8] Jim Gray, Adam Bosworth, Andrew Layman, Hamid Pirahesh, Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals, In Proceedings of the 12th International Conference on Data Engineering, 1996.

[9] V. Harinarayan, A. Rajaraman, J. D. Ullman, Implementing Data Cubes Efficiently, In Proceedings of the ACM SIGMOD Conference, 1996.

[10] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramakrishnan, S. Sarawagi, On the Computation of Multidimensional Aggregates, In Proceedings of VLDB Conference, 1996.

[11] K. A. Ross, D. Srivastava, Fast Computation of Sparse Datacubes, In Proceedings of VLDB Conference, 1997.

[12] Kevin Beyer, Raghu Ramakrishnan, Bottom-Up Computation of Sparse and Iceberg CUBEs, In Proceedings of the ACM SIGMOD Conference, 1999.

[14] Ying Feng, Divyakant Agrawal, Amr El Abbadi, Ahmed Metwally, Range CUBE: Efficient Cube Computation by Exploiting Data Correlation, In Proceedings of the International Conference on Data Engineering, 2004.

[15] Yannis Kotidis, Aggregate View Management in Data Warehouses, Handbook of Massive Data Sets, pp. 711-741, 2002.

[16] Goetz Graefe, Query Evaluation Techniques for Large Databases, ACM Computing Surveys, Vol. 25, Issue 2, pp. 73-169, 1993.

[17] Surajit Chaudhuri, Kyuseok Shim. Including Group-By in Query Optimization. In Proceedings of the Twentieth International Conference on Very Large Databases, pp. 354-366, 1994.

[18] Zhimin Chen, Vivek Narasayya, "Efficient Computation of Multiple Group By Queries," In Proceedings of the ACM SIGMOD Conference, 2005.

[19] M. R. Garey, D. S. Johnson, Computers and Intractability, chapter Appendix, pp. 208-209.

[20] L. R. Foulds, R. L. Graham, The Steiner Problem in Phylogeny is NP-Complete, Advances in Applied Mathematics, vol. 3, pp. 43-49, 1982.

[21] C. H. Papadimitriou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, chapter 11, pp. 247-254, 1982.

[22] TPC Committee, Transaction Processing Council, http://www.tpc.org/

이 기 용

1998년 2월 한국과학기술원 전산학과 학사. 2000년 2월 한국과학기술원 전산학과 석사. 2006년 2월 한국과학기술원 전자전산학과 전산학전공 박사. 관심분야는 데이타 웨어하우스, OLAP, XML

박 창 섭

정보과학회논문지 : 데이타베이스
제 33 권 제 1 호 참조

김 명 호

정보과학회논문지 : 데이타베이스
제 33 권 제 1 호 참조