SPJ 실체화 뷰의 효율적인 점진적 관리 기법

이 기 용[†] · 손 진 현[††] · 김 명 호[†††]

요 약

데이터 웨어하우스에서는 질의를 빠르게 처리하기 위해 실체화 뷰(materialized view)가 흔히 사용된다. 실체화 뷰는 그의 정의에 포함된 데이터 소스들이 변경되면 이를 반영하기 위해 갱신되어야 한다. 실체화 뷰의 갱신은 많은 부하를 야기하므로, 실체화 뷰를 효율적으로 갱신하는 것은 매우 중요한 문제이다. 실체화 뷰의 효율적인 갱신 방법에 대해서는 이미 많은 연구가 있어왔지만, SPJ(Select-Project-Join) 형태로 정의된 실체화 뷰를 효율적으로 갱신하는 방법은 충분히 연구되지 않았다. 본 논문에서는 데이터 소스들에 대한 접근 비용을 최소화함으로써 SPJ 실체화 뷰를 효율적으로 점진적으로 갱신하는 방법을 제안한다. 제안하는 방법은 동적 계획법 알고리즘을 사용하여 최적의 갱신 방법을 찾는다. 마지막으로, 다양한 성능 평가 실험을 통해 제안하는 방법이 우수한 성능을 가지고 있음을 보인다.

키워드 : 실체화 뷰, 점진적 뷰 관리, 데이터 웨어하우스

An Efficient Incremental Maintenance of SPJ Materialized Views

Ki Yong Lee[†] · Jin Hyun Son[††] · Myoung Ho Kim[†††]

ABSTRACT

In the data warehouse environment, materialized views are typically used to support efficient query processing. Materialized views need to be updated when source data change. Since the update of the views may impose a significant overhead, it is essential to update the views efficiently. Though various view maintenance strategies have been discussed in the past, the efficient maintenance of SPJ materialized views has not been sufficiently investigated. In this paper, we propose an efficient incremental view maintenance method for SPJ materialized views that minimizes the total accesses to data sources. The proposed method finds an optimal view maintenance strategy using a dynamic programming algorithm. We also present various experimental results that shows the efficiency of our proposed method.

Key Words : Materialized View, Incremental View Maintenance, Data Warehouse

1. 서 론

데이터 웨어하우스(data warehouse)는 효과적인 의사 결정을 지원하기 위해 구축된 시스템이다[1]. 데이터 웨어하우스에서는 특정한 데이터를 요구하는 질의보다는 전체 데이터의 패턴을 분석하는 복잡한 형태의 질의가 주로 사용된다. 데이터 웨어하우스에서 발생되는 이러한 복잡한 질의를 효과적으로 처리하기 위한 기법을 OLAP(On-Line Analytical Processing)이라고 부른다. OLAP 질의는 흔히 여러 릴레이션(relation)들의 조인(join)을 포함하고 있는데, 이러한 질의 처리는 매우 큰 비용을 요구한다. 데이터 웨어하우스는 이러한 질의들을 빠르게 처리하기 위해 처리 비용이 큰 질의들을 미리 계산하고, 그 결과를 실체화 뷰(materialized view)의 형태로 저장해 놓는다.

이러한 실체화 뷰는 그의 정의에 사용된 데이터 소스가 변화하면, 이를 반영하기 위해 갱신되어야 한다. 실체화 뷰의 갱신은 많은 부하를 야기하므로, 실체화 뷰를 효율적으로 갱신하는 것은 매우 중요한 문제이다. 데이터 소스가 변화되었을 때, 이를 실체화 뷰에 반영하는 방법은 크게 재계산(recomputation) 방법과 점진적 관리(incremental maintenance) 방법으로 나뉜다. 점진적 관리 방법이란 뷰의 변경될 부분만을 계산하여 반영하는 방법을 말한다. 일반적으로 데이터 웨어하우스에서는 기존 데이터의 크기에 비해 변경 데이터의 크기가 매우 작으므로, 대부분의 경우 점진적 관리 방법은 재계산 방법에 비해 매우 효과적이다. 이러한 이유로 효율적인 점진적 관리 방법에 관한 많은 연구가 있어 왔다[2-10].

※ 본 연구는 정보통신부 및 정보통신연구진흥원의 대학 IT연구센터 육성 · 지원 사업(IITA-2006-C1090-0603-0031)의 연구결과로 수행되었음.
† 정 회 원 : 삼성전자(주) 책임연구원
†† 종신회원 : 한양대학교 컴퓨터공학과 조교수
††† 정 회 원 : 한국과학기술원 전산학전공 교수
논문접수 : 2006년 5월 29일, 심사완료 : 2006년 9월 8일

1.1 관련 연구

데이터 웨어하우스에는 다양한 형태의 실체화 뷰들이 저장된다. [2, 3]은 Select-Project-Join(SPJ) 형태로 정의된 SPJ 뷰를 점진적으로 관리할 수 있도록 해주는 식(expression)을 제안하였다. [4, 11, 12]는 집단화(aggregate) 뷰를 점진적으로 관리할 수 있도록 해주는 식을 제안하였다. 최근에는 비분배적(non-distributive) 집단화 뷰에 대한 점진적 관리 방법도 제안되었다[13]. 이 외에도, 상위 k개를 선택하는 질의(top-k query)가 포함된 실체화 뷰에 대한 점진적 관리 방법[14]과 피봇(pivot) 연산자가 포함된 실체화 뷰에 대한 점진적 관리 방법[15]도 제안되었다. 한편으로는 위에서 제안된 방법들을 기반으로 하여, 분산 환경에서 정의된 실체화 뷰를 점진적으로 관리하는 방법들도 제안되었다[16-19].

본 논문에서는 데이터 웨어하우스에서 널리 쓰이고 있는 뷰의 한 종류인 SPJ 뷰에 대한 점진적 관리 방법을 논의한다. n개의 릴레이션 $R_1, R_2, ..., R_n$의 조인으로 정의된 SPJ 뷰 $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$에 대해, V의 변경 ΔV는 다음의 식으로 구할 수 있다[2].

$$\Delta V = (\Delta R_1 \bowtie R_2 \bowtie ... \bowtie R_n) \cup (R_1 \bowtie \Delta R_2 \bowtie ... \bowtie R_n) \cup ... (\Delta R_1 \bowtie \Delta R_2 \bowtie ... \bowtie R_n) \cup ... \cup (\Delta R_1 \bowtie \Delta R_2 \bowtie ... \bowtie \Delta R_n) \quad (1)$$

ΔR_i는 R_i의 변경될 튜플들로 이루어진 변경 릴레이션을 의미한다. V를 갱신하기 위해서는 위의 식에 따라 ΔV를 계산한 뒤 이를 V에 반영하면 된다. 식 (1)은 $(2^n - 1)$개의 항으로 이루어져 있다. $(\Delta R_1 \bowtie R_2 \bowtie ... \bowtie R_n, R_1 \bowtie \Delta R_2 \bowtie ... \bowtie R_n, ..., \Delta R_1 \bowtie \Delta R_2 \bowtie ... \bowtie \Delta R_n)$. 식 (1)은 릴레이션의 수가 증가함에 따라 항의 수가 지수적으로 증가하므로, 다음과 같이 n개의 항으로 이루어진 식이 제안되었다[3].

$$\Delta V = (\Delta R_1 \bowtie R_2 \bowtie ... \bowtie R_n) \cup (R_1' \bowtie \Delta R_2 \bowtie ... \bowtie R_n) \cup ... \cup (R_1' \bowtie R_2' \bowtie ... \bowtie \Delta R_n) \quad (2)$$

여기서 R'은 $R' = R \cup \Delta R$을 의미하며, ΔR이 반영된 릴레이션 R을 의미한다. 식 (2)는 식 (1)에 비해 적은 수의 항을 갖지만, 식 (1)과 같은 결과를 낸다는 점에서 동등(equivalent)하다.

최근 [20]은 뷰의 변경을 비동기적으로 단계적으로 반영하는 방법을 제안하였다. [10]은 하나 이상의 뷰를 갱신할 때, 그들 간에 공통적인 중간 결과들을 재사용하여 전체 성능을 높이는 방법을 제안하였다. [21]은 릴레이션들의 변경을 각기 다른 정책으로 뷰에 반영함으로써, 뷰의 갱신 비용을 좀 더 줄일 수 있는 방법을 제안하였다. 이 방법들은 모두 식 (2)를 사용하고 있다. [8]은 SPJ 뷰의 점진적 관리 식 중에서 식 (2)가 가장 효율적인 식이라고 하였다.

1.2 연구 동기

앞서 언급한 바와 같이, 주어진 뷰에 대해 적용 가능한 점진적 관리 식에는 여러 가지가 있을 수 있다. 이들 중 어떠한 식을 사용하느냐에 따라 뷰의 갱신 비용이 달라진다. 예를 들어, 식 (1)을 사용하는 경우 식 (1)에서 각 릴레이션들은 $(2^{n-1} - 1)$번씩 나타나며, 따라서 식 (1)을 계산하기 위해 각 릴레이션들은 $(2^{n-1} - 1)$번씩 접근되어야 한다. 반면에 식 (2)을 사용하면 각 릴레이션들은 $(n - 1)$번씩만 접근되면 된다. 릴레이션들에 대한 접근 횟수가 감소할수록 릴레이션들에 대한 접근 비용이 감소한다. [8]은 이러한 접근 비용의 측면에서 식 (2)가 SPJ 뷰에 대한 점진적 관리 식들 중에서 가장 효율적인 식이라고 하였다.

하지만 우리는 식 (2)보다 각 릴레이션에 대한 접근 횟수를 더 줄일 수 있다. 식 (2)를 사용할 경우, 뷰 $V = R_1 \bowtie R_2 \bowtie R_3$의 변경 ΔV는 다음과 같은 식으로 구할 수 있다.

$$\Delta V = (\Delta R_1 \bowtie R_2 \bowtie R_3) \cup (R_1' \bowtie \Delta R_2 \bowtie R_3) \cup (R_1' \bowtie R_2' \bowtie \Delta R_3) \quad (3)$$

식 (3)에서 R_3는 두 번 접근됨에 주목하라. 식 (3)의 첫 번째 항과 두 번째 항을 묶으면 다음과 같이 R_3에 대한 접근 횟수를 두 번에서 한 번으로 줄일 수 있다.

$$\Delta V = ((((\Delta R_1 \bowtie R_2) \cup (R_1' \bowtie \Delta R_2)) \bowtie R_3) \cup (R_1' \bowtie R_2' \bowtie \Delta R_3) \quad (4)$$

R_3가 R_1과 R_2에 비해 매우 크다면, 식 (4)는 R_3에 대한 접근 횟수를 줄임으로써 식 (3)보다 더 적은 비용으로 ΔV를 구할 수 있다. 일반적으로, 주어진 뷰의 점진적 관리 비용을 최소화하기 위해서는 주어진 뷰에 대해 가능한 점진적 관리 식들 중 최소 비용의 점진적 관리 식을 찾아야 한다. 그러나 식 (3)과 식 (4)의 경우 외에, 임의의 SPJ 뷰에 대해 최적의 점진적 관리 식을 구하는 것은 쉬운 일이 아니다. 본 논문에서는 주어진 임의의 SPJ 뷰에 대해 최적의 점진적 관리 식을 구하는 방법을 제안한다.

본 저자의 기존 연구[9]에서는 위와 같은 동기에서, 주어진 SPJ 뷰에 대한 최적의 관리 식을 찾는 방법을 논의했었다. 그러나 해당 연구에서는 주어진 뷰에 대해 적용 가능한 모든 점진적 관리 식 중 일부 제한된 형태의 식들만을 고려했었다. 본 연구에서는 좀 더 일반적인 식들을 고려하여 보다 일반적인 최적의 관리 식을 찾는 방법을 제안한다. 또한 이전 연구[15]에서 향후 연구 과제로 다루었던, 둘 이상의 뷰가 존재할 때 이들 간에 중간 결과를 공유하는 방법에 대해서도 논의한다.

본 논문의 구성은 다음과 같다. 2장에서는 본 논문에서 고려하는 실체화 뷰의 형태를 정의하고, 점진적 관리 식의 예상 비용을 구하기 위한 비용 모델을 설명한다. 3장에서는 본 논문에서 제안하는 방법을 설명한다. 본 논문에서 제안하는 방법을 둘 이상의 뷰에 대해 확장한 내용은 4장에서 다룬다. 5장에서는 성능 평가 결과를 제시하며, 6장에서는 결론을 맺는다.

2. 환경 모델

본 장에서는 본 논문에서 고려하는 뷰의 형태를 설명하고, 점진적 관리 식의 계산 비용을 예상하기 위한 비용 모델을 설명한다.

2.1 뷰 정의 모델

본 논문에서는 Select-Project-Join 질의의 형태로 정의된 SPJ 실체화 뷰를 고려한다. SPJ 뷰는 데이터 웨어하우스에서 많이 쓰이는 실체화 뷰의 한 종류로서, 일반화된 프로젝션(generalized projection)[6] 연산자를 사용하면 집단화(aggregate) 연산자가 포함된 뷰까지도 표현할 수 있다. n개의 릴레이션 $R_1, R_2, \ldots, R_n$에 대해 SPJ 뷰 V는 다음과 같이 정의된다.

$$V = \Pi_L \sigma_C (R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)$$

여기서 L은 애트리뷰트들의 리스트를 나타내며, C는 선택 조건을 나타낸다.

릴레이션 R에 대해 R의 변경은 ΔR로 표시한다. R'은 ΔR이 반영된 R을 나타내며 $R' = R \cup \Delta R$로 표시할 수 있다. 여기서 $\cup$ 연산자는 일반적인 합집합 연산자가 아닌 릴레이션 간의 병합(merge) 연산자를 나타낸다. 이에 대한 정확한 정의는 [4-6]을 참조한다. 주어진 뷰 V에 대해 ΔV가 $(\Delta R_1 \bowtie R_2) \cup (R_1' \bowtie \Delta R_2)$라는 식을 통해 계산된다면, V에 대한 점진적 관리는 다음과 같이 나타낼 수 있다.

$$V' = \Delta V \cup (\Delta R_1 \bowtie R_2) \cup (R_1' \bowtie \Delta R_2)$$

2.2 비용 모델

본 논문에서는 주어진 관리 식의 계산 비용을 예측하기 위해 [8]에서 제안된 선형 작업 계량법(linear work metric)을 사용한다. 선형 작업 계량법은 비교적 단순하면서도 효과적으로 점진적 관리 식의 비용을 예측할 수 있다는 장점이 있다[8]. 이 모델에서는 점진적 관리 식에 포함된 항들이 독립적으로 계산된다고 가정한다. 주어진 점진적 관리 식의 예상 비용은 식에 포함된 항들에 대한 예상 비용의 총합이며, 각 항의 예상 비용은 해당 항에 포함된 릴레이션들에 대한 접근 비용의 합이다. 한 릴레이션에 대한 접근 비용은 해당 릴레이션의 크기에 비례한다. Cost(E)를 식 E의 예상 비용이라고 하자. 뷰 $V = R_1 \bowtie R_2 \bowtie R_3$에 대해 식 (3)을 사용해 ΔV를 구하는 경우, 예상 비용은 다음과 같이 구할 수 있다.

$$\begin{aligned} Cost(\Delta V) &= Cost(\Delta R_1 \bowtie R_2 \bowtie R_3) \\ &\quad + Cost(R_1' \bowtie \Delta R_2 \bowtie R_3) \\ &\quad + Cost(R_1' \bowtie R_2' \bowtie \Delta R_3) \\ &= c \cdot (|\Delta R_1| + |R_2| + |R_3|) + c \cdot (|R_1'| \\ &\quad + |\Delta R_2| + |R_3|) \\ &\quad + c \cdot (|R_1'| + |R_2'| + |\Delta R_3|) \end{aligned} \tag{5}$$

여기서, c는 비례상수이고 $|R|$은 릴레이션 R의 크기를 나타낸다.

3. Optimal Delta Evaluation 방법

본 장에서는 주어진 뷰에 대해 가장 최소의 계산 비용을 가진 점진적 관리 식을 찾아내는 방법을 제안한다. 본 논문에서는 제안하는 방법을 *optimal delta evaluation 방법*이라 부르기로 한다. 본 논문에서는 SPJ 뷰의 점진적 관리에 지금까지 고려되었던 식들보다 더 일반적인 식들을 고려함으로써, 이전에 제안되었던 식보다 더 효율적인 식을 찾는다. 다음 절은 제안하는 방법의 핵심 아이디어를 설명한다.

3.1 재귀 분할 (Recursive Partitioning)

뷰 $V = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4$에 대해, 식 (2)를 사용하면 ΔV는 다음과 같이 계산된다.

$$\begin{aligned} \Delta V = &(\Delta R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4) \cup (R_1' \bowtie \Delta R_2 \bowtie R_3 \bowtie R_4) \\ &\cup (R_1' \bowtie R_2' \bowtie \Delta R_3 \bowtie R_4) \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie \Delta R_4) \end{aligned} \tag{7}$$

여기서 첫 번째 항과 두 번째 항을 $R_3 \bowtie R_4$로 묶고, 세 번째 항과 네 번째 항을 $R_1' \bowtie R_2'$로 묶으면 식 (7)은 다음과 같이 된다.

$$\begin{aligned} \Delta V = &(\Delta R_1 \bowtie R_2 \cup R_1' \bowtie \Delta R_2) \bowtie R_3 \bowtie R_4) \\ &\cup (R_1' \bowtie R_2' \bowtie (\Delta R_3 \bowtie R_4 \cup R_3' \bowtie \Delta R_4)) \end{aligned} \tag{8}$$

식 (8)은 식 (7)과 달리 R_1, R_2, R_3, R_4가 각각 두 번씩만 접근됨에 주목하라. $R_i \bowtie R_j \bowtie \ldots \bowtie R_k$의 변경을 $\Delta(R_i \bowtie R_j \bowtie \ldots \bowtie R_k)$로 나타내자. 즉, $R_i' \bowtie R_j' \bowtie \ldots \bowtie R_k' = (R_i \bowtie R_j \bowtie \ldots \bowtie R_k) \cup \Delta(R_i \bowtie R_j \bowtie \ldots \bowtie R_k)$이다. 그러면 식 (8)은 다음과 같이 간단히 쓸 수 있다.

$$\Delta V = (\Delta(R_1 \bowtie R_2) \bowtie R_3 \bowtie R_4) \cup (R_1' \bowtie R_2' \bowtie \Delta(R_3 \bowtie R_4)) \tag{9}$$

여기서 $\Delta(R_1 \bowtie R_2) = \Delta R_1 \bowtie R_2 \cup R_1' \bowtie \Delta R_2$이고, $\Delta(R_3 \bowtie R_4) = \Delta R_3 \bowtie R_4 \cup R_3' \bowtie \Delta R_4$이다. 식 (9)는 ΔV를 다음과 같이 계산한다.

(i) R_1, R_2, R_3, R_4를 $\{R_1, R_2\}$와 $\{R_3, R_4\}$로 나눈다.
(ii) 각 그룹의 조인에 대한 변경, 즉, $\Delta(R_1 \bowtie R_2)$과 $\Delta(R_3 \bowtie R_4)$을 계산한다.
(iii) (ii)의 결과를 사용해서 뷰의 전체 변경 ΔV을 계산한다.

식 (7)은 식 (9) 외에 다음과 같이 변환될 수도 있다.

$$\Delta V = (\Delta(R_1 \bowtie R_2 \bowtie R_3) \bowtie R_4) \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie \Delta(R_4)) \tag{10}$$

여기서 $\Delta(R_1 \bowtie R_2 \bowtie R_3) = (\Delta R_1 \bowtie R_2 \bowtie R_3) \cup (R_1' \bowtie \Delta R_2 \bowtie R_3) \cup (R_1' \bowtie R_2' \bowtie \Delta R_3)$이다. 위 식에서는 R_1, R_2, R_3, R_4를 $\{R_1, R_2, R_3\}$와 $\{R_4\}$로 나누었으며 $\Delta(R_1 \bowtie R_2 \bowtie R_3)$를 먼저 계산한다. 이 경우, R_4에 대한 접근 횟수는 한 번으로 줄어든다. 식 (9)와 식 (10)은 각각 다른 파티션, 즉, $\{\{R_1, R_2\}, \{R_3, R_4\}\}$와 $\{\{R_1, R_2, R_3\}, \{R_4\}\}$를 의미하며, 계산 비용도 서로 다르다.

이제 이러한 개념을 일반화해보도록 하자. 뷰 $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$에 대해, $\{P_1, P_2, ..., P_m\}$을 $\{R_1, R_2, ..., R_n\}$의 파티션이라고 하자. 즉, $\bigcup_{i=1}^{m} P_i = \{R_1, R_2, ..., R_n\}$, $P_i \cap P_j = \emptyset$ $(i \neq j, 1 \leq i, j \leq m)$, $P_i \neq \emptyset$ $(1 \leq i \leq m)$이다. 그러면, ΔV는 다음과 같이 표현될 수 있다.

$$\begin{aligned}\Delta V = &(\Delta(\bowtie P_1) \bowtie (\bowtie P_2) \bowtie ... \bowtie (\bowtie P_m))\\ &\cup ((\bowtie P_1') \bowtie \Delta(\bowtie P_2) \bowtie ... \bowtie (\bowtie P_m))\\ &\cup ... \\ &\cup ((\bowtie P_1') \bowtie (\bowtie P_2') \bowtie ... \bowtie \Delta(\bowtie P_m))\end{aligned} \quad (11)$$

여기서 $P_i = \{R_s, R_t, ..., R_u\}$에 대해, $\bowtie P_i = R_s \bowtie R_t \bowtie ... \bowtie R_u$이고, $\bowtie P_i' = R_s' \bowtie R_t' \bowtie ... \bowtie R_u'$이다. 식 (11)은 릴레이션의 집합 $\{R_1, R_2, ..., R_n\}$을 m개의 그룹 $P_1, P_2, ..., P_m$으로 분할하고, 각 그룹의 조인에 대한 변경을 먼저 계산한다. 즉, 식 (11)은 $\Delta(\bowtie P_1), \Delta(\bowtie P_2), ..., \Delta(\bowtie P_m)$을 먼저 계산하고, 이를 사용하여 뷰의 전체 변경 ΔV를 구한다.

이제 $\Delta(\bowtie P_i)$ $(1 \leq i \leq m)$의 계산을 고려해보자. $\Delta(\bowtie P_i)$도 식 (11)과 같은 식을 사용하여 재귀적으로 계산될 수 있다. 다시 말해, $\Delta(\bowtie P_i)$도 P_i를 여러 개의 그룹으로 분할하여 계산될 수 있다. 이러한 분할은 각 그룹에 속해 있는 릴레이션의 개수가 하나가 될 때까지 재귀적으로 계속된다.

식 (11)을 *delta evaluation expression*이라고 부른다. Delta evaluation expression은 각 $\Delta(\bowtie P_i)$이 단일 릴레이션 ΔR_j로 전개될 때까지 재귀적으로 전개된다. Delta evaluation expression이 ΔV를 올바르게 구함은 쉽게 알 수 있다.

식 (11)에서 $\{R_1, R_2, ..., R_n\}$을 어떻게 분할하느냐에 따라 다양한 delta evaluation expression이 나타난다. 예를 들어, 뷰 $V = R_1 \bowtie R_2 \bowtie R_3$에 대해 $\{R_1, R_2, R_3\}$을 $P_1 = \{R_1, R_2\}$, $P_2 = \{R_3\}$로 분할한 경우, delta evaluation expression은 $\Delta V = (\Delta(R_1 \bowtie R_2) \bowtie R_3) \cup (R_1' \bowtie R_2' \bowtie \Delta(R_3))$가 된다. 한편 $\{R_1, R_2, R_3\}$을 $P_1 = \{R_1\}$, $P_2 = \{R_2\}$, $P_3 = \{R_3\}$로 분할한 경우, delta evaluation expression은 $\Delta V = (\Delta(R_1) \bowtie R_2 \bowtie R_3) \cup (R_1' \bowtie \Delta(R_2) \bowtie R_3) \cup (R_1' \bowtie R_2' \bowtie \Delta(R_3))$가 된다. 이들 중 최적의 delta evaluation expression을 찾아 뷰의 변경을 계산하는 방법을 *optimal delta evaluation 방법*이라고 부른다. 3.3절에서는 최적의 delta evaluation expression을 찾는 방법에 대해 논의한다.

3.2 Delta Evaluation Tree

본 논문에서는 delta evaluation expression을 delta evaluation tree라고 부르는 트리(tree)로 표현한다. 뷰 $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$에 대해 어떤 delta evaluation expression이 주어졌다고 하자. 이 때, delta evaluation tree의 루트(root) 노드는 ΔV, 즉, $\Delta(R_1 \bowtie R_2 \bowtie ... \bowtie R_n)$이 된다. 또한, $\Delta R_1, \Delta R_2, ..., \Delta R_n$은 delta evaluation tree의 단말(leaf) 노드가 된다. Delta evaluation tree의 각 내부(internal) 노드는 $\{R_1, R_2, ..., R_n\}$의 어떤 부분 집합의 조인에 대한 변경을 나타낸다. 즉, 각 내부 노드는 $\Delta(R_s \bowtie R_t \bowtie ... \bowtie R_u)$(단, $\{R_s, R_t, ..., R_u\} \subseteq \{R_1, R_2, ..., R_n\}$)의 형태를 가진다. 주어진 delta evaluation expression에서 $\{R_s, R_t, ..., R_u\}$가 $P_1, P_2, ..., P_m$으로 분할되었다고 하자. 그러면 노드 $\Delta(R_s \bowtie R_t \bowtie ... \bowtie R_u)$의 자식 노드들은 $\Delta(\bowtie P_1), \Delta(\bowtie P_2), ..., \Delta(\bowtie P_m)$이 된다. 이 때, $\Delta(\bowtie P_i)$는 좌측으로부터 i번째 자식이 된다. (그림 1)은 $V = R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie R_6$에 대해 가능한 두 개의 delta evaluation tree를 보여준다. 두 트리는 각각 다음의 delta evaluation expression을 나타낸다.

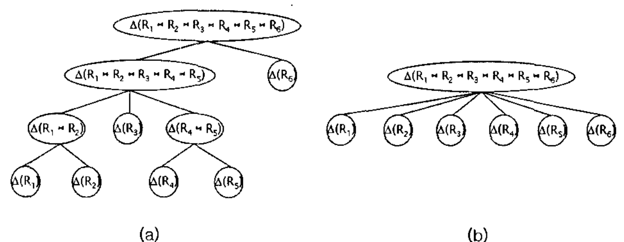
(a)

$$\begin{aligned}&\Delta(R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie R_6)\\ &\quad = (\Delta(R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5) \bowtie R_6)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie R_4' \bowtie R_5' \bowtie \Delta(R_6))\\ &\quad\quad \Delta(R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5)\\ &\quad = (\Delta(R_1 \bowtie R_2) \bowtie R_3 \bowtie R_4 \bowtie R_5)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie \Delta(R_3) \bowtie R_4 \bowtie R_5)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie \Delta(R_4 \bowtie R_5))\\ &\quad \Delta(R_1 \bowtie R_2) = (\Delta(R_1) \bowtie R_2) \cup (R_1' \bowtie \Delta(R_2))\\ &\quad \Delta(R_4 \bowtie R_5) = (\Delta(R_4) \bowtie R_5) \cup (R_4' \bowtie \Delta(R_5))\end{aligned}$$

(b)

$$\begin{aligned}&\Delta(R_1 \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie R_6)\\ &\quad = (\Delta(R_1) \bowtie R_2 \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie R_6)\\ &\quad\quad \cup (R_1' \bowtie \Delta(R_2) \bowtie R_3 \bowtie R_4 \bowtie R_5 \bowtie R_6)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie \Delta(R_3) \bowtie R_4 \bowtie R_5 \bowtie R_6)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie \Delta(R_4) \bowtie R_5 \bowtie R_6)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie R_4' \bowtie \Delta(R_5) \bowtie R_6)\\ &\quad\quad \cup (R_1' \bowtie R_2' \bowtie R_3' \bowtie R_4' \bowtie R_5' \bowtie \Delta(R_6))\end{aligned}$$

위에서 (b)는 식 (2)와 동일한 형태를 가지고 있음을 알 수 있다. 실제로, 식(11)은 $P_1 = \{R_1\}$, $P_2 = \{R_2\}$, ..., $P_n = \{R_n\}$인 경우 식 (2)가 된다. 따라서 식 (2)는 delta evaluation expression의 특수한 경우라고 할 수 있다. 다음 절에서는



(그림 1) Delta evaluation tree의 예

이들 중 최적의 delta evaluation tree를 찾는 방법에 대해 논의한다.

3.3 최적의 Delta Evaluation Tree

본 논문에서 최적의 delta evaluation tree란 주어진 뷰에 대해, 2.2절에서 설명한 선형 작업 계량법에 따른 예상 비용을 가장 최소화 하는 delta evaluation tree를 뜻한다. 본 절에서는 동적 계획법(dynamic programming)을 사용하여 최적의 delta evaluation tree를 찾는 방법을 설명한다.

한 delta evaluation tree가 주어졌을 때, 그의 예상 비용은 다음의 식으로 계산할 수 있다. 식 (11)에 선형 작업 계량법을 적용하면 다음의 재귀식 (12)가 도출된다. 여기서 계산을 간단히 하기 위해 비례상수 c는 1로 한다.

$$
\begin{aligned}
&Cost(\Delta R_i) = 0 \\
&Cost(\Delta(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n)) = \sum_{i=1}^{m} Cost(\Delta(\bowtie P_i)) \\
&\quad + (|\Delta(\bowtie P_1)| + |P_2| + \ldots + |P_m|) \\
&\quad + (|P_1'| + |\Delta(\bowtie P_2)| + \ldots + |P_m|) \\
&\quad + \ldots \\
&\quad + (|P_1'| + |P_2'| + \ldots + |\Delta(\bowtie P_m)|) \qquad (12)
\end{aligned}
$$

여기서 $P_i = \{R_s, R_t, \ldots, R_u\}$라 할 때, $|P_i| = |R_s| + |R_t| + \ldots + |R_u|$, $|P_i'| = |R_s'| + |R_t'| + \ldots + |R_u'|$, $|\Delta(\bowtie P_i)| = |\Delta(R_s \bowtie R_t \bowtie \ldots \bowtie R_u)|$이다. ΔR_i는 주어진 것이고 더 이상 계산할 필요가 없으므로 $Cost(\Delta R_i) = 0$이 된다. 최적의 delta evaluation tree는 바로 식 (12)를 최소화 하는 트리가 된다.

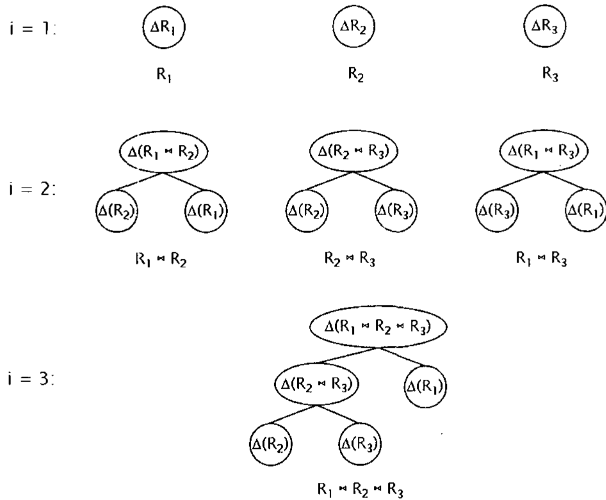
식 (12)를 최소화 하는 트리를 찾기 위해 본 논문에서는 동적 계획법을 사용한다. 동적 계획법은 최적화 문제에서 흔히 사용되는 기법이다. 여기에서는 상향식(bottom-up) 동적 계획법을 사용한다. 상향식 동적 계획법은 먼저 각 릴레이션 $R_i(1 \le i \le n)$에 대한 최적의 delta evaluation tree를 찾는다. 그 다음, 이들을 사용하여 $R_i \bowtie R_j(1 \le i, j \le n, i \ne j)$에 대한 최적의 delta evaluation tree를 찾는다. 그 다음, 이전 결과들을 사용하여 $R_i \bowtie R_j \bowtie R_k(1 \le i, j, k \le n, i \ne j \ne k)$에 대한 최적의 delta evaluation tree를 찾는다. 이러한 방법으로 결국 $R_1 \bowtie R_2 \bowtie \ldots \bowtie R_n$에 대한 최적의 delta evaluation tree를 찾게 된다.

임의의 $R_s \bowtie R_t \bowtie \ldots \bowtie R_u$에 대한 최적의 delta evaluation tree를 찾는 방법을 알아보자. 먼저 식 (12)를 최소화하는 $\{R_s, R_t, \ldots, R_u\}$의 분할 $P_1, P_2, \ldots, P_m$을 찾는다. 그러면 $R_s \bowtie R_t \bowtie \ldots \bowtie R_u$에 대한 최적의 delta evaluation tree는 다음과 같이 생성될 수 있다. 먼저 새 루트 노드 $\Delta(R_s \bowtie R_t \bowtie \ldots \bowtie R_u)$를 생성한다. 그리고 $(\bowtie P_i)$에 대해 구해진 최적의 delta evaluation tree를 새 루트 노드에 대해 좌측에서 i번째 자식 노드로 만든다. *최적성의 원리(Principle of Optimality)*에 따라 최적의 delta evaluation tree의 모든 하부트리(subtree) 역시 최적의 delta evaluation tree가 되어야 한다.

(그림 2)는 지금까지 설명한 알고리즘인 *FindOptimalDeltaEvaluationTree*를 나타낸다. *FindOptimalDeltaEvaluationTree*는 상향식 방법으로 해를 찾는다. 해당 알고리즘은 먼저 각 릴레이션 R_i에 대한 최적의 delta evaluation tree를 구한다. 각 릴레이션 R_i에 대한 최적의 delta evaluation tree는 하나의 노드 ΔR_i만으로 이루어진 트리임이 명백하다. 그 다음, $\{R_1, R_2, \ldots, R_n\}$의 각 부분집합의 조인에 대해 부분집합의 크기를 2로부터 n까지 단계적으로 늘려가며 각각의 부분집합의 조인에 대한 최적의 delta evaluation tree를 구한다. 크기가 $i(1 < i \le n)$인 부분집합의 조인에 대한 최적의 delta evaluation tree를 구할 때는, 크기가 1부터 $(i - 1)$까지의 부분집합들의 조인에 대해 구해진 최적의 delta evaluation tree들을 사용한다. 이와 같은 방법을 사용하면, 최종적으로 $\{R_1, R_2, \ldots, R_n\}$의 조인에 대한 최적의 delta evaluation tree를 구할 수 있게 된다. (그림 3)은 $V = R_1 \bowtie R_2 \bowtie R_3$에 대해 *FindOptimalDeltaEvaluationTree*을 사용하

```
Procedure FindOptimalDeltaEvaluationTree(in: V = R₁ ⋈ R₂ ⋈ ... ⋈ Rₙ)

  /* 초기화 */
  For (i = 1 to n) do
    - Construct the optimal delta evaluation tree for Rᵢ, i.e., ΔRᵢ.
  End for

  /* 알고리즘 본체 */
  For (i = 2 to n) do
    For (each P ⊆ {R₁, R₂, ..., Rₙ} such that cardinality(P) = i) do
      - Find {P₁, P₂, ..., Pₘ} of P that minimizes expression (12).
      - Construct the optimal delta evaluation tree for (⋈P) using
        optimal delta evaluation trees for (⋈P₁), (⋈P₂), ..., (⋈Pₘ).
    End for
  End for

  /* 위의 결과로부터 V에 대한 최적의 delta evaluation tree를 얻는다. */

End procedure
```

(그림 2) 동적 계획법 기반 알고리즘



(그림 3) *FindOptimalDeltaEvaluationTree*의 수행 예

여 최적의 delta evaluation tree를 찾는 과정을 보여준다. (그림 3)의 각 트리는 $\{R_1, R_2\ R_3\}$의 한 부분집합의 조인에 대한 최적의 delta evaluation tree를 나타낸다.

3.3 제안 방법의 분석

먼저 *FindOptimalDeltaEvaluationTree*의 시간 복잡도에 대해 알아보자. 이 알고리즘은 $\{R_1, R_2, ..., R_n\}$의 모든 부분집합의 조인에 대한 최적의 delta evaluation tree를 구한다. $\{R_1, R_2, ..., R_n\}$에 대한 모든 부분집합의 수는 $\Sigma_{i=1}^{n} {}_nC_i$이다. 여기서 ${}_nC_i$는 n개의 서로 다른 개체에서 i개를 뽑는 경우의 수이다. 또한, 각 부분집합에 대해 이 알고리즘은 해당 부분집합의 모든 분할 방법을 고려한다. i개의 원소를 가진 집합에 대한 가능한 모든 분할 방법은 B(i)이다[22]. 따라서 이 알고리즘의 시간 복잡도는 $O(\Sigma_{i=1}^{n}({}_nC_i \cdot B(i)))$가 된다. n이 커짐에 따라 B(n)은 n^n에 접근함이 알려져 있다[23]. 그러나 실제로 데이터 웨어하우스에 존재하는 뷰들은 10개 미만의 릴레이션만을 포함한다. 예를 들어, 의사 결정 시스템에 대한 성능 측정 표준 명세인 TPC-R에 포함된 질의들은 릴레이션의 개수가 많아야 7개를 넘지 않는다[24]. 특히 $n < 10$인 경우, B(n)은 22,000을 넘지 않는다. 따라서 제안하는 알고리즘은 실제 데이터 웨어하우스 환경에서 사용되는데 큰 문제가 없다.

이제 본 논문에서 제안하는 방법에서 각 릴레이션들에 대한 접근 횟수를 알아보자. 본 논문에서 제안하는 방법에서 각 릴레이션들에 대한 접근 횟수는 다음과 같다.

[정리 1] *뷰* $V = R_1 \bowtie R_2 \bowtie ... \bowtie R_n$*에 대한 모든 delta evaluation expression에 대해서 릴레이션* R_i*의 접근 횟수* A_i*는* $1 \leq A_i \leq (n - 1)$*을 만족한다.*

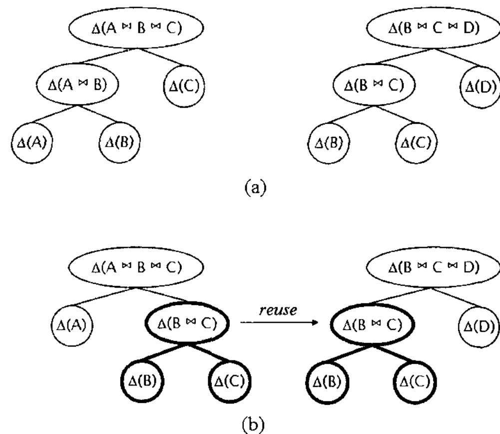
증명) 각 릴레이션에 대한 접근 횟수는 점진적 관리 식에서 해당 릴레이션이 나타나는 횟수와 같다고 가정하자. 식 (11)에서 릴레이션 R_i (또는 R_i')가 나타나는 항은 언제나 $R_i \notin \{R_s, R_t, ..., R_u\}$인 $\Delta(R_s \bowtie R_t \bowtie ... \bowtie R_u)$을 포함하고 있다. 그러므로, R_i이 나타나는 횟수는 $R_i \notin \{R_s, R_t, ..., R_u\}$이고 $\{R_s, R_t, ..., R_u\} \subseteq \{R_1, R_2, ..., R_n\}$인 집합 $\{R_s, R_t, ..., R_u\}$의 수를 넘지 못한다. 또한, 이러한 집합 $\{R_s, R_t, ..., R_u\}$들 간에는 어떠한 릴레이션도 중복되어 나타날 수 없다. 따라서 이러한 집합 $\{R_s, R_t, ..., R_u\}$들의 모든 가능한 수는 $|\{R_1, R_2, ..., R_n\} - \{R_i\}| = (n - 1)$를 넘을 수 없으므로 증명이 완료된다. □

(그림 1 (a))에서 $R_1, R_2, R_3, R_4, R_5, R_6$에 대한 접근 횟수는 각각 4번, 4번, 3번, 4번, 4번, 1번이 된다. 반면, (그림 1 (b))에서는 모든 릴레이션에 대한 접근 횟수가 5번으로 동일하다. 이와 같이 제안하는 방법에서는 릴레이션에 대한 접근 횟수가 각기 다양할 수 있다. R_6이 다른 릴레이션들에 비해 크기가 매우 크다고 하자. 이 경우, 제안하는 방법은 R_6에 대한 접근 횟수를 줄이기 위해 (그림 1 (a))와 같은 delta evaluation tree를 선택할 수 있다. 이 때 R_6에 대한 접근 횟수는 5번에서 1번으로 줄어든다.

4. 둘 이상의 뷰가 존재하는 경우로의 확장

지금까지는 단일 뷰의 갱신에 대해 논의하였다. 그러나 일반적으로 데이터 웨어하우스에는 여러 개의 뷰가 존재할 수 있다. 이러한 뷰들 간에는 공통적으로 사용되는 식들이 흔히 나타난다. 만약 한 뷰를 갱신하는 중에 발생된 중간 결과를 다른 뷰를 갱신하는데 사용할 수 있다면, 전체적인 관리 비용은 더욱 줄어들 것이다.

본 연구에서 사용하는 식 (11)은 ΔV를 계산하기 전에 $\Delta(\bowtie P_1), \Delta(\bowtie P_2), ..., \Delta(\bowtie P_m)$을 먼저 계산한다. 만약 어떤 뷰를 위해 계산한 $\Delta(\bowtie P_i)$를 다른 뷰에서 사용할 수 있다면 전체적인 계산 비용은 더욱 줄어들 수 있을 것이다. 본 장에서는 둘 이상의 뷰가 존재할 때, 뷰들 간에 중간 결과를 재사용하도록 하는 방법을 제안한다. (그림 4)는 뷰들 간에 발생하는 중간 결과를 재사용하는 예를 보여준다.



(그림 4) 뷰들 간에 중간 결과를 재사용하는 예

두 개의 뷰 $V_1 = A \bowtie B \bowtie C$와 $V_2 = B \bowtie C \bowtie D$가 있다고 하자. (그림 4 (a))는 ΔV_1과 ΔV_2를 독립적으로 계산했을 때 각각에 대한 최적의 delta evaluation tree를 나타낸다. 그러나 만약 $\Delta(B \bowtie C)$를 재사용한다면 전체적인 계산 비용은 더 줄어들 수도 있을 것이다. (그림 4 (b))는 $\Delta(B \bowtie C)$를 재사용하는 경우 ΔV_1과 ΔV_2에 대한 최적의 delta evaluation tree를 나타낸다. 그러나 재사용이 항상 더 좋은 결과를 가져오는 것은 아니다. 예를 들어 앞의 예에서 $\Delta(B \bowtie C)$의 크기가 매우 큰 경우, 이것을 재사용하는 것은 오히려 전체적인 비용을 증가시킬 수 수 있다. 따라서 어떤 중간 결과를 재사용하고 어떤 중간 결과를 재사용하지 않을 것인가를 판단해서 전체적인 계산 계획을 세워야 한다. 본 논문에서는 이 문제를 multiple view maintenance problem 이라고 부르도록 한다.

4.1 문제 정의

$V = \{V_1, V_2, ..., V_n\}$을 갱신해야 할 뷰들의 집합이라고 하자. 또한, $MinCost(\Delta V_i)$를 V_i를 위한 최적의 delta evaluation tree의 계산 비용이라고 하자. 만약 우리가 뷰들 간에 중간 결과를 재사용하지 않는다면, $\Delta V_1, \Delta V_2, ..., \Delta V_n$을 계산하기 위한 총 비용 TotalCost(V)는 다음과 같이 계산된다.

$$TotalCost(V) = \Sigma_{V_i \in V} MinCost(\Delta V_i) \quad (15)$$

한편, 재사용하기로 결정한 중간 결과들의 집합을 S라고 하자. 예를 들어, (그림 4 (a))에서는 S = {}가 되고, (그림 4 (b))에서는 $S = \{\Delta(B \bowtie C)\}$가 된다. $MinCost(\Delta V_i|S)$를 S가 이미 계산되어 저장되어 있을 때, 이를 사용하여 ΔV_i를 구하는 최적의 delta evaluation tree의 계산 비용이라고 하자. 명백히 $MinCost(\Delta V_i|S) \leq MinCost(\Delta V_i)$의 관계가 성립한다. S가 주어졌을 때, S를 사용하여 $\Delta V_1, \Delta V_2, ..., \Delta V_n$을 구하는 총 비용 TotalCost(V|S)는 다음과 같이 계산된다.

```
Procedure GreedyMultipleViews(in: V = {V1, V2, ..., Vn})
  /* 초기화 */
  C = SharedExpressions(V)
  S = Ø

  /* Main loop */
  While (C ≠ Ø) do
    Choose c ∈ C which minimizes TotalCost(V|S∪{c})
    If (TotalCost(V|S∪{c}) < TotalCost(V|S)) do
      S = S∪{c}; C = C - {c};
    Else
      C = Ø
    End if
  End while

  /* 결과 */
  Construct the optimal delta evaluation trees for
      V1, V2, ..., Vn
  using S;
End procedure
```

(그림 5) 둘 이상의 뷰를 위한 탐욕 정책 알고리즘

$$TotalCost(V|S) = \Sigma_{S_i \in S} MinCost(\Delta S_i|S - \{\Delta S_i\}) + \Sigma_{V_i \in V} MinCost(\Delta V_i|S) \quad (16)$$

위 식의 첫 번째 항은 S 자체를 구하는데 드는 비용이다. 첫 번째 항에서 $MinCost(\Delta S_i|S)$ 대신 $MinCost(\Delta S_i|S - \{\Delta S_i\})$를 사용했음에 유의하라. 이는 S_i를 계산할 때는 S에 S_i가 존재하지 않기 때문이다. 두 번째 항은 S를 사용하여 V를 계산하는 데 드는 비용을 나타낸다. $MinCost(\Delta V_i)$를 구하는 위해서는 3.3절에서 제안한 *FindOptimalDelta Evaluation Tree* 를 사용하면 된다. 그러나, $MinCost(\Delta V_i|S)$를 구하기 위해서 식 (12)는 다음과 같이 확장되어야 한다.

$$Cost(\Delta R_i|S) = 0$$

$$Cost(\Delta(R_1 \bowtie R_2 \bowtie ... \bowtie R_n)|S) = 0, \text{ if } \Delta(R_1 \bowtie R_2 \bowtie ... \bowtie R_n) \in S$$

$$\begin{aligned} Cost(\Delta(R_1 \bowtie R_2 \bowtie ... \bowtie R_n)|S) = & \sum_{i=1}^{m} Cost(\Delta(\bowtie P_i)|S) \\ & + (|\Delta(\bowtie P_1)| + |P_2| + ... + |P_m|) \\ & + (|P_1'| + |\Delta(\bowtie P_2)| + ... + |P_m|) \\ & + ... \\ & + (|P_1'| + |P_2'| + ... + |\Delta(\bowtie P_m)|), \text{ otherwise.} \end{aligned} \quad (17)$$

$Cost(\Delta V_i|S)$는 S가 계산되어 저장되어 있을 때, 이를 사용하여 ΔV_i를 계산하는 비용을 나타낸다. 식 (17)을 이용하면 $MinCost(\Delta V_i|S)$ 및 S를 사용하는 경우의 V_i에 대한 최적의 delta evaluation tree를 구할 수 있다. 이제, multiple view maintenance problem을 다음과 같이 정의하자: *주어진 뷰들의 집합 $V = \{V_1, V_2, ..., V_n\}$에 대해, TotalCost(V|S)가 최소가 되는 재사용할 중간 결과의 집합 S와, S를 사용하는 $V_1, V_2, ..., V_n$에 대한 최적의 delta evaluation tree를 구하여라.*

4.2 탐욕(Greedy) 알고리즘

앞서 정의한 multiple view maintenance problem은 전통적인 복수 질의 최적화 문제(multiple query optimization problem)[25-28]과 동일한 형태를 가진다. 복수 질의 최적화의 목적은 질의들 간에 공통적으로 사용되는 식들을 재사용하여 전체적인 계산 비용을 줄이는 것이다. 본 논문에서의 delta evaluation tree는 복수 질의 최적화 문제에서의 질의 수행 계획(query plan)에 대응된다. 이와 유사하게, delta evaluation tree간의 공통 노드들은 복수 질의 최적화 문제에서의 질의 간 공통 식들에 대응된다. [25]은 복수 질의 최적화 문제가 NP-hard임을 보였다. 이러한 이유로, 복수 질의 최적화 문제에 관한 대부분의 논문은 휴리스틱(heuristic) 알고리즘을 제안하고 있다. 본 논문에서는 재사용할 S를 탐욕 정책으로 선택하는 휴리스틱 알고리즘을 제안한다. 본 논문에서 제안하는 알고리즘은 [28]에서 제안된 알고리즘과 유사하다. 본 논문에서 제안하는 알고리즘은 재사용할 S를 먼저 선택한 후, 각 뷰에 대해 S를 사용하는 최적의 delta evaluation tree를 구한다.

알고리즘은 먼저 뷰들 간에 공유되는 공통 식들을 뽑아낸다. 이러한 식들의 집합을 C라고 하자. 이러한 C에 대해, S는 C의 한 부분집합임은 명백하다. 즉, S ⊆ C이다. TotalCost(V|S)를 최소화하는 최적의 S를 찾기 위한 가장 간단한 방법은 C의 모든 부분집합에 대해 TotalCost(V|S)를 계산해보고, 그들 중 TotalCost(V|S)를 가장 최소화하는 부분집합을 선택하는 것이다. 그러나 C의 원소의 개수가 n이라고 할 때, C의 모든 부분집합의 개수는 2^n개 이므로, 모든 부분집합을 찾아보는 것은 현실적으로 불가능하다. (그림 5)는 탐욕 정책으로 S를 선택하는 알고리즘을 보여준다. 여기서 *SharedExpressions* 함수는 둘 이상의 뷰에서 공통적으로 사용되는 식들을 추출하여 이를 반환하는 함수이다. 해당 알고리즘은 한번에 하나씩, 재사용할 식을 선택하여 S에 추가한다. 재사용할 식을 하나씩 선택할 때, 해당 알고리즘은 S에 추가했을 때 가장 이득을 주는 식을 선택한다. 해당 알고리즘에 대한 성능 평가 결과는 5장에 나타내었다.

5. 성능 평가

본 장에서는 본 논문에서 제안하는 방법의 성능 평가 결과를 보인다. 실험에 사용된 뷰 관리 방법의 성능은 해당 방법으로 뷰를 갱신했을 때 걸리는 시간으로 측정하였다. 본 실험에서는 TPC-R의 스키마와 데이터[24]를 사용하였으며, 데이터 웨어하우스 시스템으로는 256MB 메모리를 가진 Sun Ultra-60에서 운영되는 Oracle8*i* 데이터베이스 시스템을 사용하였다. 실험에서는 다음과 같은 총 6개의 뷰를 사용하였다. 각각의 뷰는 TPC-R 질의를 기반으로 정의되었다.

V_1 = *CUSTOMER* ⋈ *ORDERS* ⋈ *LINEITEM*
⋈ *SUPPLIER* ⋈ *NATION* ⋈ *REGION*
V_2 = *PART* ⋈ *SUPPLIER* ⋈ *LINEITEM*
⋈ *ARTSUPP* ⋈ *ORDERS* ⋈ *NATION*
V_3 = *PART* ⋈ *SUPPLIER* ⋈ *PARTSUPP*
⋈ *NATION* ⋈ *REGION*
V_4 = *CUSTOMER* ⋈ *ORDERS* ⋈ *LINEITEM*
V_5 = *NATION* ⋈ *LINEITEM* ⋈ *ORDERS* ⋈ *SUPPLIER*
V_6 = *NATION* ⋈ *CUSTOMER*
⋈ *ORDERS* ⋈ *LINEITEM*

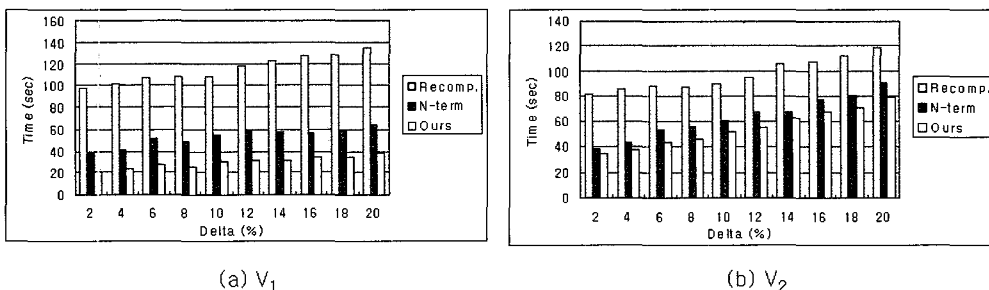
본 실험에서는 본 논문에서 제안하는 방법을 재계산 방법 및 [8]에서 제안된 점진적 관리 방법과 비교하였다. [8]에서 제안된 방법은 뷰를 갱신하기 위해 식 (2)를 사용한다. 식 (2)를 *n-term expression*이라 하고, [8]에서 제안된 방법을 *n-term method*라고 하자. n-term method는 뷰의 변경을 계산하기 위해 *최적의* n-term expression을 사용한다. 최적의 n-term expression이란 주어진 뷰에 대해 적용 가능한 식 (2)의 형태를 가지는 식들 중 예상 비용이 가장 최소가 되는 식을 말한다.

(그림 6)은 각 방법으로 V_1과 V_2을 갱신하는데 걸린 시간을 보여준다. 실험에서 각 릴레이션들에 대한 변경은 새로운 튜플들을 삽입함으로써 이루어졌다. 실험에서 각 릴레이션들은 원 릴레이션의 크기의 2%에서 20%까지 증가되었다. 이 범위는 데이터 웨어하우스 환경에서 주로 일어나는 범위이다[7, 8]. (그림 6)에서 볼 수 있듯이 본 논문에서 제안하는 방법은 실험의 범위에서 다른 두 방법보다 항상 좋은 성능을 보이고 있다.

(그림 6)의 시간은 최적화에 걸리는 시간까지 포함한 것이다. n-term method의 경우에 최적화에 걸리는 시간이란 최적의 n-term expression을 찾는데 걸리는 시간이며, 제안하는 방법의 경우에 최적화에 걸리는 시간은 최적의 delta evaluation tree를 찾는데 걸리는 시간이다. 실험에서는 양쪽 모두 채 1초도 안되는 시간이 걸렸다. 이는 전체 성능에 큰 영향을 미치지 않는다.

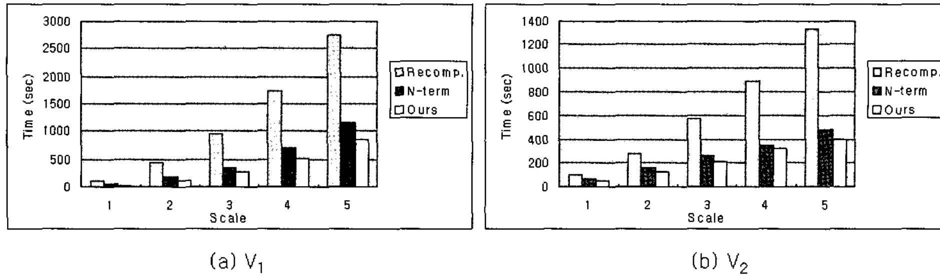
본 논문에서 사용한 비용 모델에 따르면, 릴레이션들의 크기는 뷰의 갱신 비용에 매우 큰 영향을 미친다. (그림 7)은 릴레이션들의 크기를 100%에서 500%까지 증가시켰을 때 세 가지 방법의 성능 평가 결과를 보여준다. 이 경우에도 제안하는 방법이 V_1과 V_2에 대해서 모두 가장 좋은 성능을 보여주고 있다. 따라서 제안하는 방법은 기존 방법들에 비해 좋은 성능을 가지고 있다고 결론내릴 수 있다.

지금까지는 단일 뷰에 대한 성능 평가 결과에 대해 알아보았다. 둘 이상의 뷰에 대한 성능 평가 결과는 (그림 8)에 나타내었다. 여기서는 뷰들 간에 중간 결과를 재사용하는 optimal delta evaluation 방법(*ours(sharing)*이라 표시)을, 뷰들 간에 중간 결과를 재사용하지 않는 optimal delta evaluation 방법(*ours(no sharing)*이라 표시) 및 n-term method와 비교하였다. 이 실험에서는 앞서 정의한 6개 뷰를 모두 사용하였다. (그림 8)에서의 시간은 최적화에 걸리는 시간 및 6개의 뷰 전부를 갱신하는데 걸리는 시간을 모두



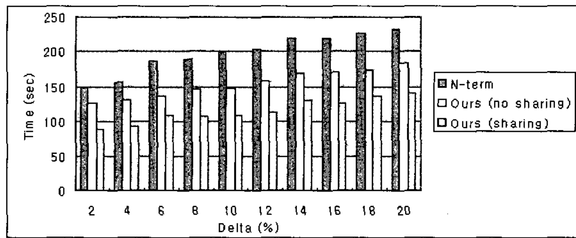
(a) V_1 (b) V_2

(그림 6) 각 릴레이션에 대한 변경의 크기를 변화시켰을 때의 성능 평가 결과



(a) V_1 (b) V_2

(그림 7) 릴레이션의 크기를 변화시켰을 때의 성능 평가 결과



(그림 8) 둘 이상의 뷰에 대한 성능 평가 결과

합한 것이다. 이 경우에도 optimal delta evaluation 방법은 n-term method 보다 좋은 성능을 나타내고 있다. 또한 예상했던 대로, 중간 결과를 재사용하는 optimal delta evaluation 방법은 중간 결과를 재사용하지 않는 optimal delta evaluation 방법보다 좋은 성능을 보이고 있다.

6. 결 론

데이터 웨어하우스는 의사 결정을 지원하기 위해 많은 양의 정보를 요약하여 저장하는 시스템이다. 이러한 요약 데이터는 여러 데이터 소스들에 대해 정의된 실체화 뷰로 볼 수 있다. 데이터 소스들이 변화하면, 실체화 뷰들은 이러한 변화를 반영하기 위하여 갱신되어야 한다. 뷰를 갱신하는 데는 큰 비용이 들기 때문에, 이러한 비용을 최소화하는 것은 매우 중요한 일이다. 본 논문에서는 데이터 웨어하우스 환경에서 SPJ 실체화 뷰의 효율적인 갱신을 위한 optimal delta evaluation 방법을 제안하였다. 각 릴레이션들의 크기를 고려하여 찾아진 최적의 delta evaluation tree는 뷰의 변경을 계산하는데 드는 비용을 최소화할 수 있도록 해준다. 성능 평가 결과, 본 논문에서 제안하는 방법은 기존의 방법들에 비해 더 좋은 성능을 가지고 있음을 보였다. 한편, 둘 이상의 뷰가 존재할 경우 delta evaluation tree의 중간 결과들은 서로 간에 재사용될 수 있다. 본 논문에서는 optimal delta evaluation 방법을 둘 이상의 뷰들 간에 중간 결과를 재사용할 수 있도록 확장하였다. 또한 이에 대한 실험을 통해 보다 좋은 성능을 얻을 수 있음을 보였다.

참 고 문 헌

[1] W. H. Immon, Building the Data Warehouse, Wiley Computer Publishing, 1996.

[2] J. A. Blakeley, P. Larson, F. W. Tompa, Efficiently Updating Materialized Views, In Proceedings of ACM SIGMOD Conference, pp.61-71, 1986.

[3] S. Ceri, J. Widom, Deriving production rules for incremental view maintenance, In Proceedings of the 7th International Conference on Very Large Databases, pp.108-119, 1991.

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

[5] T. Griffin, L. Libkin, Incremental maintenance of views with duplicates, In Proceedings of ACM SIGMOD Conference, pp.328-339, 1995.

[6] L. S. Colby, T. Griffin, L. Libkin, I. S. Mumick, H. Trickey, Algorithms for deferred view maintenance, In Proceedings of ACM SIGMOD Conference, pp.469-492, 1996.

[7] I. S. Mumick, D. Quass, B. S. Mumick, Maintenance of Data Cubes and Summary Tables in a Warehouse, In Proceedings of ACM SIGMOD Conference, pp.100-111, 1997.

[8] W. J. Labio, R. Yerneni, H. Garcia-Molina, Shrinking the Warehouse Update Window, In Proceedings of ACM SIGMOD Conference, pp.383-394, 1999.

[9] Ki Yong Lee, Myoung Ho Kim, "Efficient Incremental View Maintenance in Data Warehouses," Journal of KISS (B), 2000.

[10] H. Mistry, R. Roy, S. Sudarshan, K. Ramamritham, Materialized View Selection and Maintenance Using Multi-Query Optimization, In Proceedings of ACM SIGMOD Conference, 2001.

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

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

[13] Themistoklis Palpanas, Richard Sidle, Roberta Cochrane, Hamid Pirahesh, Incremental Maintenance for Non-Distributive Aggregate Functions, In Proceedings of VLDB, 2002.

[14] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, Yuguo Chen, Efficient Maintenance of Materialized Top-k Views, In

Proceedings of International Conference on Data Engineering, 2003.
[15] Songting Chen, Elke A. Rundensteiner, GPIVOT: Efficient Incremental Maintenance of Complex ROLAP Views, In Proceedings of International Conference on Data Engineering, 2005.
[16] Y. Zhuge, H. Garcia-Molina, J. L. Wiener, Consistency Algorithms for Multi-Source Warehouse View Maintenance, Distributed and Parallel Databases, 6(1):7-40, 1998.
[17] D. Agrawal, A. E. Abbadi, A. Singh, T. Yurek, Efficient View Maintenance at Data Warehouses, In Proceedings of ACM SIGMOD Conference, pp.417-427, 1997.
[18] Gang Luo, Jeffrey F. Naughton, Curt Ellmann, Michael Watzke, A Comparison of Three Methods for Join View Maintenance in Parallel RDBMS, In Proceedings of International Conference on Data Engineering, 2003.
[19] Bin Liu, Elke A. Rundensteiner: Cost-Driven General Join View Maintenance over Distributed Data Sources, In Proceedings of International Conference on Data Engineering (poster), 2005.
[20] K. Salem, K. Beyer, B. Lindsay, How To Roll a Join: Asynchronous Incremental View Maintenance, In Proceedings of ACM SIGMOD Conference, pp.129-140, 2000.
[21] Hao He, Junyi Xie, Jun Yang, Hai Yu: Asymmetric Batch Incremental View Maintenance, In Proceedings of International Conference on Data Engineering, 2005.
[22] B. Niamir, Attribute Partitioning in a Self-Adaptive Relational Database System, Technical Report, Cambridge, Mass.: Laboratory for Computer Science, Massachusetts Institute of Technology, 1978.
[23] M. Hammer, B. Niamir, A Heuristic Approach to Attribute Partitioning, In Proceedings of ACM SIGMOD Conference, pp.93-101, 1979.
[24] TPC Committee, Transaction Processing Council, http://www.tpc.org/
[25] T. K. Sellis, Multiple Query Optimization, ACM Transactions on Database Systems, 13(1):23-52, March, 1988.
[26] J. Park, A. Segev, Using Common Sub-expressions to Optimize Multiple Queries, In Proceedings of International Conference on Data Engineering, 1998.
[27] K. Shim, T. Sellis, D. Nau, Improvements on a Heuristic Algorithm for Multi-query Optimization, Data and Knowledge Engineering, 12:197-222, 1994.
[28] P. Roy, S. Seshadri, S. Sudarshan, S. Bhobe, Efficient and Extensible Algorithms for Multi Query Optimization, In Proceedings of ACM SIGMOD Conference, pp.249-260, 2000.

이 기 용

e-mail : kiyong.lee@gmail.com
1998년 한국과학기술원 전산학과 학사
2000년 한국과학기술원 전산학과 석사
2006년 한국과학기술원 전자전산학과 전산학전공 박사
2006년 3월~현재 삼성전자(주) 책임연구원
관심분야 : 데이터 웨어하우스, OLAP, Flash memory, Embedded database

손 진 현

e-mail : jhson@cse.hanyang.ac.kr
1996년 서강대학교 전산학과 학사
1998년 한국과학기술원 전산학과 석사
2001년 한국과학기술원 전자전산학과 박사
2001년 9월~2002년 8월 한국과학기술원 전자전산학과 박사후 연구원
2002년 9월~현재 한양대학교 컴퓨터공학과 조교수
관심분야 : 데이터베이스, 온디맨드 서비스, 유비쿼터스 컴퓨팅, 임베디드 소프트웨어, 분산 데이터 처리

김 명 호

e-mail : mhkim@dbserver.kaist.ac.kr
1982년 서울대학교 컴퓨터공학과 학사
1984년 서울대학교 컴퓨터공학과 석사
1989년 미국 Michigan State University 전산학과 박사
1989년~현재 한국과학기술원 전산학전공 교수
관심분야 : 데이터베이스 시스템, 분산 시스템, XML, 멀티미디어, 센서 네트워크 등