다이나믹 데이터 웨어하우스 환경에서 OLAP 영역-합 질의의 효율적인 처리 방법

전 석 주† · 이 주 홍††

요 약

데이터 웨어하우스에서 사용자는 전형적으로 상호작용적으로 질의를 부여함으로서 추세와 패턴 또는 예외적인 데이터의 행위를 검색한다. OLAP 영역-합 질의는 데이터 웨어하우스에서 추세를 발견하거나 또는 애트리뷰트들간의 관계를 발견하는데 폭 넓게 사용되고 있다. 최근의 기업환경은 데이터 큐브의 데이터 요소들이 자주 바뀌게 된다. 문제는 프리픽스 섬 큐브를 업데이트 하는 비용이 매우 크다는 것이다. 이 논문에서는 Δ-트리로 불리는 인덱싱 구조를 사용하여 업데이트 비용을 상당히 줄이는 참신한 알고리즘을 제안한다. 또한, 근사 또는 정확한 해를 제공하므로 질의의 전체비용을 줄일 수 있는 하이브리드 방법을 제안한다. 이는 의사 결정 지원 시스템과 같이 시간을 많이 소비하는 정확한 해보다는 빠른 근사 해를 필요로 하는 다양한 응용들에 큰 장점이 있다. 폭 넓은 실험은 우리의 방법이 다른 방법들과 비교하여 다양한 차원에서 매우 효율적으로 수행됨을 보여준다.

Efficient Processing method of OLAP Range-Sum Queries in a dynamic warehouse environment

Seok-Ju Chun† · Ju-Hong Lee††

ABSTRACT

In a data warehouse, users typically search for trends, patterns, or unusual data behaviors by issuing queries interactively. The OLAP range-sum query is widely used in finding trends and in discovering relationships among attributes in the data warehouse. In a recent environment of enterprises, data elements in a data cube are frequently changed. The problem is that the cost of updating a prefix sum cube is very high. In this paper, we propose a novel algorithm which reduces the update cost significantly by an index structure called the Δ-tree. Also, we propose a hybrid method to provide either approximate or precise results to reduce the overall cost of queries. It is highly beneficial for various applications that need quick approximate answers rather than time consuming accurate ones, such as decision support systems. An extensive experiment shows that our method performs very efficiently on diverse dimensionalities, compared to other methods.

키워드 : 데이터 웨어하우스(Data warehouses), 프리픽스-섬 큐브(Prefix-sum Cube), 의사결정 지원 시스템(Decision Support System)

1. 서 론

데이터 웨어하우스는 다양한 형태의 다차원 데이터의 상호작용적인 분석을 위해 효율적인 데이터 마이닝을 쉽게 해주는 OLAP(On-Line Analytic Processing) 도구를 제공한다[10]. OLAP은 분석가에게 정보의 다양한 뷰(view)를 접근할 수 있도록 함으로서 데이터의 집계(aggregation)에 대한 안목을 가지게 하는 데이터베이스 기술의 범주에 속한다. 이것은 종종 애트리뷰트의 다양한 상세수준에서의 요약된 데이터와 애트리뷰트의 다양한 조합을 필요로 한다. 전형적인 OLAP 응용으로는 생산효율, 이익, 판매 프로그램, 영업캠페인의 효과 및 판매 예측과 생산 능력 등을 들 수 있다.

다양한 OLAP 응용분야 중에서 흔히 데이터 큐브로 알려진 다차원 데이터베이스(MMDB)를 위한 데이터 모델이 점점 중요해 지고 있다. 데이터 큐브는 이러한 데이터베이스에서 애트리뷰트의 부분집합으로부터 만들어 진다. 어떤 애트리뷰트는 메이저 애트리뷰트 즉, 그것의 값이 관심의 대상이 되는 메이저 애트리뷰트로 선택되고 나머지 애트리뷰트는 차원(dimensions) 또는 함수애트리뷰트(functional attributes)로 선택된다[7]. 메이저 애트리뷰트가 이러한 차원에 따라서 집계되어 진다.

다음의 자동차 판매회사에서 사용되는 데이터 큐브를 고

† 정 회 원 : 안산1대학 인터넷정보과 교수
†† 종신회원 : 인하대학교 컴퓨터공학부 교수
논문접수 : 2002년 11월 5일, 심사완료 : 2003년 3월 19일

려해보자. 데이터 큐브의 차원이 MODEL_NO, YEAR, REGION, COLOR 즉, 4차원이고 메이저 애트리뷰트가 AMOUNT_OF_SALES 이라고 가정해보자. 이때, MODEL_NO는 30 models을 가지고 YEAR는 1990부터 2001년, REGION은 40개 그리고 COLOR는 {white, red, yellow, blue, gray, black}를 가진다고 하자. 그러면 데이터 큐브는 3012406의 셀을 가지며 각각의 셀은 4개의 함수 애트리뷰트 즉, MODEL_NO, YEAR, REGION과 COLOR로 만들어진 조합에 대한 AMOUNT_OF_SALES를 메이저 에트리뷰터로 가진다.

데이터 큐브는 유용한 영역-합 질의(range-sum query)라고 불리는 데이터 상에서의 분석 툴을 제공한다[6]. 영역-합 질의는 질의 영역내의 메이저 애트리뷰트에 집계 연산을 적용하여 행해진다.

대표적인 예로 "지난 1995년부터 2000년 사이에 빨간색을 가진 모든 모델에 대해 서울에서 판매된 자동차의 총대수는 얼마인가?"를 들 수 있다. 이 같은 형태의 질의는 매우 흔하며 OLAP에서 중요하다. 사용자와 상호작용을 요구하는 OLAP 응용에 대해 응답시간 또한 매우 중요하다. 영역-합 질의를 처리하는 직접적인 방법은 데이터 큐브 자체를 접근 하는 것이다. 그러나 그것은 영역-합을 얻기 위해 너무 많은 셀을 접근해야 한다는 사실에 어려움을 겪고 있다. 접근해야 할 셀의 수는 질의에 의해 정의된 서브 큐브의 크기에 비례한다.

검색 효율을 증가시키기 위해서 PC(Prefix-sum Cube)라 불리는 또 하나의 큐브를 사용하는 프리픽스 섬 방법(prefix sum method)이 제안되었다[9]. 현재의 기업 환경은 큐브내의 데이터 요소가 다이나믹하게 변경되도록 요구하고 있다. 그와 같은 환경에서 응답시간은 큐브의 검색 시간 뿐만 아니라 업데이트 시간에 의해서도 영향을 받는다. 그러나, 프리픽스 섬 방법은 검색 시간을 줄이는데에만 초점을 두고 있으므로 업데이트가 자주 발생하는 환경에서는 사용할 수가 없다.

최근에 다양한 좋은 연구 결과들[4, 6, 7, 11]이 업데이트 비용을 줄이기 위해 행해졌다. 이러한 방법들은 RPC(Relative Prefix-sum Cube)와 같은 부가적인 데이터 구조를 사용하여 PC상에서의 업데이트 증식(update propagation)을 최소화 한다. 그러나, 이러한 방법들은 어느 정도까지는 업데이트 증식을 줄였다고 하나 여전히 어떤 제한을 가진다. 왜냐하면 RPC 또한 PC를 조금 변형한 것이기 때문이다. 더군다나 이러한 업데이트의 속도 증가는 검색효율을 희생하므로 얻어지게 되므로 많은 제한을 가진다. 많은 OLAP 응용에서 검색 효율의 희생을 최소화하면서 업데이트 성능을 개선하는 것이 중요한 쟁점이 되고 있다.

본 논문에서는 Δ-트리로 불리는 인덱싱 구조을 사용하여 업데이트 증식을 크게 줄일 수 있는 효율적인 알고리즘을 제안한다. R-트리 이후로 많은 다차원 인덱싱 구조들[1, 2, 12]이 제안되었다. Δ-트리는 데이터 큐브의 변경된 값들을 저장하고 효율적인 질의 처리를 지원하기 위해 R*-트리[2]를 변형한 것이다. 제안된 알고리즘은 업데이트 비용을 많이 감소시키면서 전통적인 프리픽스 섬 방법의 장점을 그대로 이용하여 상당한 검색 효율을 얻을 수 있다. 본 논문은 인덱싱 구조인 Δ-트리를 이용하는 다이나믹 업데이트 큐브라고 불리는 새로운 기술을 제시한다. 우리의 공헌은 다음과 같이 요약된다.

- Δ-트리를 이용하여 프리픽스 섬 방식의 이점을 가지면서 업데이트 비용을 상당히 줄이는 효율적인 알고리즘을 제안한다. 다양한 방법들의 업데이트 성능에 대한 비교를 제시한다. 제안된 알고리즘의 업데이트 시간 복잡도는 $O(\log N_u)$인데, N_u는 데이터 큐브에서 변경된 셀들의 수이다. 다양한 크기의 데이터 큐브에 대한 분석을 제공하며 실험적인 결과로서 다른 방법들과 비교하여 다양한 차원에서 매우 효율적으로 수행됨을 보여준다.
- OLAP 영역-합 질의에 대하여 근사 결과 또는 정확한 결과를 제공하는 하이브리드 방법을 제안하고 또한 근사오차를 상당히 줄이는 방법을 제안한다. 저자의 지식으로는 사용자의 요구에 맞추어서 정확한 해와 근사해를 같이 제공할 수 있는 하이브리드 방법을 제안한 것은 우리의 방법이 처음이다. 폭 넓은 실험을 통하여 근사 방법이 상당한 정확도를 유지하면서 질의 처리에 있어서 뚜렷한 속도 향상을 보여준다.

이 논문의 나머지는 다음과 같이 구성되어진다.

2장에서 관련연구가 주어진다. 3장에서는 우리가 제안한 방법과 직접적인 관련이 있는 프리픽스 섬 방법에 대해 간단히 설명한다. 4장에서는 우리가 제안한 작업에 대한 자세한 설명이 주어진다. 5장에서는 근사 결과 또는 정확한 결과를 제공하는 하이브리드방법을 나타낸다. 6장에서는 우리가 제안된 알고리즘의 성능평가에 대한 실험적인 결과들이 주어지고 6장에서 결론을 맺는다.

2. 관련 연구

이전의 단락에서 간단하게 소개한 바와 같이 OLAP 데이터 큐브 상에서의 질의를 언급하는 다양한 접근 방법이 제안되었다. 호 등[9]은 프리픽스 섬 방법 이라 불리는 데이터 큐브에서 영역-합 질의를 계산하는 세련된 알고리즘을 제시하였다. 프리픽스 섬 방법의 본질적인 아이디어는 데이

터 큐브의 많은 프리픽스 섬들을 미리 계산하여 실시간에 임의의 질의를 처리하는데에 사용하지는 것이다. 이러한 방법은 매우 강력한 것으로 판명되었다. 그러나 큐브 내의 데이터 값이 자주 바뀔 때 PC를 관리하는 것은 많은 비용이 드는 문제가 있다.

PC에서 업데이트 증식을 감소시키기 위해 거프너 등[7]은 RPC(relative prefix-sum cube)를 사용한 방법을 제안하였다. 이것은 프리픽스 방법과 직접적인 방법 사이에서 질의와 업데이트간의 비용 균형을 맞추기 위해 노력한 것이다. 그러나 이러한 방법은 고차원과 대용량의 데이터 큐브에서는 비현실적이다. 왜냐하면 업데이트 비용이 지수적으로 증가하기 때문이다.

찬 등[4]은 두 개의 직각차원에 기초한 HC(Hierarchical Cubes)로 불리는 새로운 형태의 큐브를 제안하였다. 그들은 HBC(Hierarchical Band Cube)가 거프너에 의해 제안된 알고리즘 보다 상당히 더 좋은 질의-업데이트 트레이드오프를 가짐을 보였다. 그러나 고층 레벨의 추상적인 큐브를 저층 레벨의 구체적인 큐브로의 인덱싱 메핑이 실제로 구현하기엔 너무나 복잡하다. 또한 그들이 제안한 방법의 해석적인 결과를 실험적으로 검증하지 못했다.

더 최근에 거프너 등[6]은 PC를 재귀적으로 분해(decomposite)해서 얻어지는 다이나믹 데이터 큐브(Dynamic Data Cube)를 제안하였다. 그들은 또한 가정하기를 데이터 큐브의 각 차원이 같은 크기를 가진다고 가정하여 이러한 분해 기술에 의해 트리 구조 만들었다. 그러나 서론에서 제시한 카-세일과 같이 실제적인 환경에서 데이터 큐브는 각 차원의 크기가 각각 다르다(앞의 예에서 각 차원의 크기는 각각 30, 12, 40 그리고 6이다). PC를 분해할 때 각 차원의 크기가 각각 다르면 트리의 균형을 유지하기가 무척 어렵게 된다. 더군다나 데이터 큐브가 고차원 대용량이면 이러한 방법은 큰 실행 오버헤드를 초래하게 된다.

3. 프리픽스 섬 방법

이 장은 우리가 제안한 방법에 밀접한 관계가 있는 프리픽스 섬 큐브에 관한 배경 정보를 소개한다. 프리픽스 섬 방법에서는 데이터 큐브와 같은 크기의 프리픽스 섬 큐브가 사용되며 프리픽스 섬 큐브에는 데이터 큐브로 부터 미리 계산된 다양한 프리픽스 섬(prefix-sum)을 저장한다. 프리픽스 섬 큐브의 각 셀은 데이터 큐브의 각 셀을 포함한 그 셀까지의 모든 합을 저장한다. (그림 1)은 6×8 데이터 큐브 A와 프리픽스 섬 큐브 PC를 보여준다.

PC[4, 6]는 A[0, 0]에서 A[4, 6] 범위내의 모든 셀을 합한 값을 가진다. 따라서, 데이터 큐브 A의 전체의 합은 마지막 셀 PC[5, 7]에서 찾을 수 있다. D = {1, 2, ⋯, d}를 차원의 집합을 나타내고 n_i가 각차원에 있는 셀의 수를 나타낸다고 가정하자. 호 등[9]은 미리 계산된 프리픽스 섬을 저장하기 위해 $N=\prod_{i=1}^{d} n_i$개의 부가적인 셀을 필요로 하는 간단한 방법을 제안하였는데 이 방법은 임의의 영역-합을 구하기 위해서 2^d의 적절한 프리픽스 섬을 사용하여 2^d-1의 스텝만으로 계산을 가능하게 한 것이다.

명시적으로, $0 \le x_i < n_i$이고 $i \in D$인 모든 x_i와 i에 대하여

$$PC[x_1, x_2, \cdots, x_d] = Sum(0:x_1,\ 0:x_2, \cdots,\ 0:x_d) = \sum_{i_1=0}^{x_1}\sum_{i_2=0}^{x_2}\cdots\sum_{i_3=0}^{x_3} A[i_1, i_2, \cdots, i_d]\text{이다.}$$

예를 들면, d가 2일때, $0 \le x < n_1$이고 $0 \le y < n_2$에 대하여

$$PC[x, y] = Sum(0:x,\ 0:y) = \sum_{i=0}^{x}\sum_{i=0}^{y} A[i, j]\text{이다.}$$

(그림 1)은 2차원에 대한 데이터 큐브 $A[x_1, x_2]$와 그에 대응하는 $PC[x_1, x_2]$를 보여주는 예이다. 프리픽스 섬 방법은 매우 강력하여 데이터 큐브의 크기에 상관없이 상수시간에 영역-합 질의를 처리 할 수 있다. 아래의 보조정리 3.1은 어떻게 데이터 큐브 A의 임의의 영역-합을 PC중에서 기껏해야 2^d개의 적절한 요소를 이용하여 계산 할 수 있는가를 보여준다

Index	0	1	2	3	4	5	6	7
0	4	5	2	8	3	7	5	6
1	2	1	5	3	7	2	4	2
2	5	3	9	3	4	7	1	3
3	3	5	6	1	8	5	1	6
4	3	2	1	4	7	8	6	4
5	6	2	2	6	1	9	5	2

(a) 데이터 큐브 *A*

Index	0	1	2	3	4	5	6	7
0	4	9	11	19	22	29	34	40
1	6	12	19	30	40	49	58	66
2	11	20	36	50	64	80	90	101
3	14	28	50	65	87	108	119	136
4	17	33	56	75	104	133	150	171
5	23	41	66	91	121	159	181	204

(b) *PC*

(그림 1) 6×8 데이터 큐브 *A*와 그것의 프리픽스 섬 큐브 PC

아래 식의 왼편은 A의 영역질의를 명시하고 오른편은 2^d개의 더하기 항으로 구성되어 있으며 각각의 항은 모든 s(j)의 곱에 의해 결정되는 "+" 또는 "-" 부호를 가지는 PC

값으로 구성된다. 표기의 편리성을 위해 어떤 $j (j \in D)$에 대하여 $x_j = 1$이면 $PC[x_1, x_2, \cdots, x_d] = 0$으로 한다.

보조정리 3.1 [9]. 모든 $j (j \in D)$에 대하여 다음의

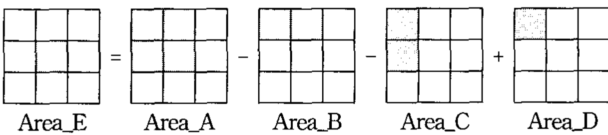
$$s(j) = \begin{cases} 1 & if\ x_j = h_j \\ -1 & if\ x_j = l_j - 1 \end{cases}$$ 이라 가정하자.

그러면 모든 $j(j \in D)$에 대하여 다음의 식이 성립한다.

$$Sum(l_1 : h_1, l_2 : h_2, \cdots, l_d : h_d) = \sum_{\forall x_j \in \{l_j - 1, h_j\}} \left\{ \left(\prod_{i=1}^{d} s(i) \right) * PC[x_1, x_2, \cdots, x_d] \right\}$$ ■

예제 3.2 d가 2일 때, 영역-합 $Sum(l_1 : h_1, l_2 : h_2)$은 다음의 계산으로 구할 수 있다. 즉, $PC[h_1, h_2] - PC[h_1, l_2 - 1]$ $PC[l_1 - 1, h_2]$ + $PC[l_1 - 1, l_2 - 1]$. (그림 1)에서 설명한 바와 같이 영역-합 Sum(1 : 4, 2 : 6)은 다음의 식으로부터 유도 될 수 있다. PC[4, 6] - PC[0, 6] - PC[4, 1] + PC[0, 1] = 150 - 34 - 33 + 9 = 102. ■

(그림 2)는 2차원 경우를 계산하기 위한 기하학적인 설명을 준다.



(그림 2) 2차원 경우에 대한 기하학적인 설명

4. 다이나믹 업데이트 큐브

우리는 WWW과 같은 인터넷 기술이 도래와 더불어 장소와 시간에 구애 받지 않고 OLAP 서버와 같은 다양한 응용들을 사용할 수 있게 되었다. 사용자들은 지리적으로 넓게 퍼져 있으며 또한 수 많은 사용자들이 동시에 대규모의 OLAP 서버를 사용하길 원할 것이다. 이러한 환경에서 동시에 질의 처리와 다이나믹한 데이터를 업데이트를 하려고 할 때 성능이 이슈가 된다. 더군다나 PC에 기초한 이전의 방법들의 경우에 있어서 만일 사용자들의 질의가 업데이트 처리 때문에 변경되어야 될 셀을 하나라도 포함한다면 사용자들의 많은 질의가 블록(block)되어 진다. 그렇지 않다면 질의 결과들은 잘못된 값을 만들어 낼 것이다. 즉, 하나의 업데이트 오퍼레이션이 많은 질의들을 블록 되도록 만들 수가 있다. 그러므로 업데이트 비용이 높을 경우 이러한 블럭킹이 OLAP 서버의 전체 성능을 감소 시킬 것은 명백하다.

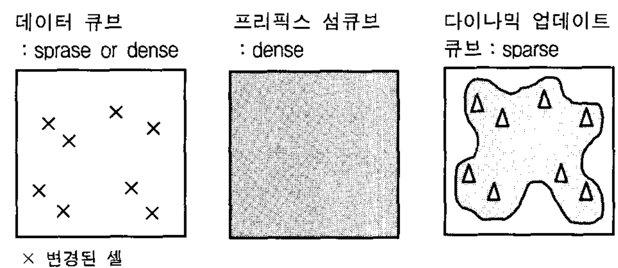
OLAP 서버는 의사결정 프로세스을 지원하는 시스템으로 널리 사용되어진다. 많은 사용자들은 그들이 어떤 결정에 도달할 때까지 그들의 관심과 관련된 수많은 질의를 실행하게 된다. 만일 모든 질의가 정확한 값을 요구한다면 고비용의 실행시간을 초래하며 OLAP 서버에 큰 부담을 주게 된다. 반대로 모든 질의에 근사 해가 응답되어 진다면 이 또한 많은 사용자들이 부정확한 결과로 인해 혼동을 받게 될 것이다. 따라서, 저 비용의 근사 해를 사용하여 관심있는 부분을 찾아 질의의 범위를 줄인 후에 고비용의 정확한 해를 구할 수 있는 하이브리드 방법이 필요하다. 그러나, PC에 기초한 이전의 방법들은 이러한 형태의 하이브리드 방법을 제공하지 못한다.

4.1 아이디어

다이나믹한 OLAP 환경에서 데이터 큐브 셀은 자주 바뀌게 된다. 문제는 PC를 업데이트 하는 비용이 매우 크다는 것이다. 기본적인 아이디어는 변경된 셀들을 PC에 직접 저장하지 않고 '다이나믹 업데이트 큐브'라고 불리는 가상 큐브에 저장하여 관리하는 것이다. 영역-합 질의가 처리될 때 PC와 다이나믹 업데이트 큐브가 동시에 사용되어 진다.

데이터 큐브가 밀집(dense) 또는 희박(sparse) 하든지 간에 프리픽스 섬 큐브는 항상 밀집하다. 반면에 다이나믹 업데이트 큐브는 희박하다. 왜냐하면 다이나믹 업데이트 큐브는 데이터 큐브의 변경된 셀들 만을 포함하기 때문이다.

예제 4.1 (그림 3)에서 보는 바와 같이 비록 데이터 큐브가 희박하더라도 PC는 데이터 큐브 셀 들의 누적된 합들을 저장하기 때문에 항상 밀집하다. 그림에서 '×'는 변경된 셀을 나타내고 'Δ'는 변경된 셀의 차이를 나타낸다. 즉, $\Delta = \times_{new} - \times_{old}$ 이다. 다이나믹 업데이트 큐브는 Δ값 만을 저장하므로 항상 희박하다. ■

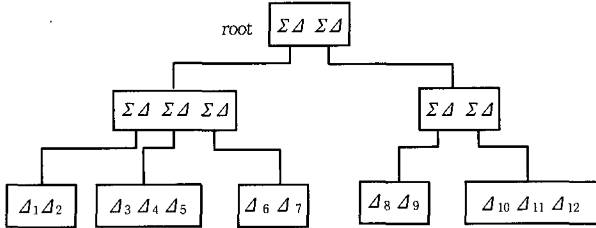


(그림 3) 다이나믹 업데이트 큐브의 기본 개념

다이나믹 업데이트 큐브 셀의 위치는 다차원 공간의 점으로 표현 될 수 있다. 따라서, 이러한 셀들은 'Δ-트리'로 불리는 다차원 인덱스 구조에 저장된다. 공간적으로 서로 이웃에 위치한 셀들은 이에 대응하는 MBR(minimum bounding rectangle)에 클러스터링 된다. Δ-트리를 검색할 때 Δ-트리의 겹치지(overlap) 않는 MBR들은 효율적으로 잘려나간다. 자세한 것은 4.4절에서 설명되어진다. 다이나믹 업데이트 큐브 셀의 위치와 Δ값은 Δ-트리에 저장된다.

본 논문에서 제안된 아이디어는 다음과 같이 요약 할 수 있다.

- 프리픽스 섬 큐브는 밀집하고 다이나믹 업데이트 큐브는 희박하기 때문에 데이터 큐브가 바뀔 때마다 우리는 프리픽스 섬 큐브를 직접 업데이트 하지 않고 대신에 데이터 큐브의 변경된 정보를 Δ-트리에 저장하여 Δ-트리를 관리한다. 이것은 업데이트 비용을 줄여주고 프리픽스 섬 방식에서 발생되는 업데이트 증식(update propagation) 문제를 해결할 수있다.
- 영역-합 질의를 처리할 때 Δ-트리를 부분적으로 검색함으로 근사 해를 얻을 수 있다. 즉, 루트부터 단말 노드까지 검색하는 대신 레벨 i의 내부 노드까지 수행 한다. 근사 해를 계산하는 자세한 내용은 5장에서 설명된다.
- 데이터 큐브의 셀이 바뀔 때 마다 Δ-트리의 크기는 증가하게 된다. Δ-트리가 너무 커지면 검색과 업데이트의 비용이 많아지게 된다. 그러면 Δ-트리에 저장된 모든 정보는 응용들에 따라서 주별, 월별 또는 어떤 임계치에 맞추어서 주기적으로 프리픽스 섬 큐브에 반영되어 질 필요가 있다(즉, 'bulk updates').



(그림 4) Δ-tree의 구조

4.2 Δ-트리

이 장에서 우리는 (그림 4)에서 보는 바와 같이 Δ-트리의 구조를 소개한다. Δ-트리의 생성과정은 R*-트리의 생성과정과 동일하다. 처음에는 Δ-트리는 단지 하나의 디렉토리 노드 즉, 루트 노드를 가진다. 데이터 큐브 셀이 업데이트 될 때마다 데이터 큐브 셀의 현재와 과거의 값의 차(Δ)와 그 셀의 위치 정보가 Δ-트리로 저장된다. 다음과 같이 Δ-트리를 정형화 할 수 있다.

정의 4.2 (Δ-트리)

① 디렉토리 노드는 (L_1, L_2, …, L_n)을 포함한다. L_i는 i번째 차일드 노드 C_i에 관한 튜블(tuple)이고 각 튜블은(ΣΔ, M, cp_i, MBR_i)의 형태를 가진다. ΣΔ는 C_i가 디렉토리 노드(데이터 노드)일 때 C_i의 모든 ΣΔ(Δ)값의 합이다. cp_i는 C_i의 주소이고 MBR_i는 C_i에 있는 모든 엔트리들을 감싸는 MBR이다. M은(μ_1, μ_2, …, μ_d)를 가지는 데 여기서 d는 차원이고 μ_j는 MBR_i의 j번째 평균 위치로서 다음과 같이 정의 할 수 있다. $\mu_j = \frac{\sum_{m=1}^{n_j} mF_j(m)}{\sum_{m=1}^{n_j} F_j(m)}$

여기서 $F_j(m) = \sum_{\substack{k_h=1,n_h \\ k_j=m \\ h\neq j}} f(k_1, \cdots, k_j, \cdots, k_d)$이고 $f(k_1, k_2, \cdots, k_d)$는 MBR_i에 있는 업데이트 위치 $(k_1, k_2, \cdots, k_d)$의 값이다. 여기서 k_j 값의 범위는 $1 \le k_j \le n_j$이고 n_j는 MBR_i의 j번째 차원의 파티션 수이다.

② 데이터 큐브는 레벨이 0이고 $(D_1, D_2, \cdots, D_n)$를 포함한다. 여기서 D_i는 i번째 데이터 엔트리에 관한 튜플이며 (P_i, Δ_i)의 형태를 가진다. P_i는 포지션 인덱스이며 Δ_i는 변경된 셀 값의 차이이다.

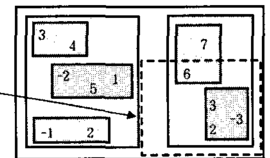
ΣΔ를 사용하는 목적은 영역-합 질의 빠른 근사 해를 구하기 위한 것이고 M를 사용하는 목적은 근사 테크닉을 개선 시키는 것이다(5.2절에서 예제 5.2에서 설명).

4.3 영역-합 질의

우리는 영역-합 질의를 처리하기 위해서 PC와 Δ-트리를 모두 사용한다. 이전에 언급한 것과 같이 PC에는 가장 최근에 벌크 업데이트(bulk updated)된 정보들을 포함하는 반면에 Δ-트리에는 그 후에 업데이트된 정보들을 포함한다. 공간적으로 서로 인접한 업데이트된 셀들은 그에 대응하는 MBR로 클러스터링 된다.

Index	0	1	2	3	4	5	6	7
0	7*	5	2	8	3	7	12*	6
1	2	5*	5	3	7	2	4	2
2	5	1*	9	4*	4	1	3*	13
3	3	5	11*	1	8	5	6	6
4	3	2	1	4	7	8	6	1*
5	5*	2	4*	6	1	9	7	2

(a) Data cube A



(b) Dynamic update cube U

(그림 5) 데이터 큐브와 영다이나믹 업데이트 큐브에서의 영역-합 질의

예제 4.3 (그림 5)에서 '*'로 표시된 셀은 (그림 1)(a)에 있는 데이터 큐브로 부터 최근에 업데이트 되었음을 나타낸다. (그림 5)(b)에서 다이나믹 업데이트 큐브의 가장 낮은 레벨에 있는 각 MBR들은 PC에 아직 반영되지 않은 이러한 셀들은 포함한다. ■

영역-합 질의 Q가 주어질 때, 여기서 Q는($l_1 : h_1$, $l_2 : h_2$, …, $l_d : h_d$), 우리는 Q의 해를 구하기 위해 PC와 Δ-트리를 같이 사용한다. Sum(Q)를 질의 Q의 결과를 돌려주는 함수이고, PC_sum(Q)는 PC로부터 계산된 결과 값을 돌려주는 함수, 그리고 Δ_sum(Q)은 Δ-트리로 부터 구한 결과값을 돌려주는 함수라고 하자. 그러면 해는 다음의 식과 같다.

$$Sum(Q) = PC_sum(Q) + \Delta_sum(Q)$$

예제 4.4 (그림 5)에 대해서 영역-합 질의 Q가 아래와 같이 주어질 때 우리는 PC와 Δ-트리를 이용하여 Q의 해를 구할 수 있다.

```
Range-sum query(Q) : Select   Sum(A.sales)
                     From     A
                     Where    2 ≤ A.x ≤ 5 and 4 ≤ A.y ≤ 7
```

우리는 위의 식으로부터 해를 구할 수 있다. 즉, Sum(2 : 5, 4 : 7) = PC_sum(2 : 5, 4 : 7) + Δ_sum(2 : 5, 4 : 7)이다. 함수 PC_sum(2 : 5, 4 : 7)은 보조정리 2.1로 부터 즉시 계산될 수 있다. ■

정의 4.5 (격리, 포함, 교차)

MBR_Q와 MBR_T를 각각 질의 Q의 MBR, 노드 T의 MBR이라고 하자. 그러면 MBR_Q와 MBR_T사이의 관계(Relationship) 다음과 같이 정형화 되어진다.

① 격리 iff $MBR_Q \cap MBR_T = \varphi$
② 포함 iff $MBR_Q \supseteq MBR_T$
② 교차 iff $MBR_Q \cap MBR_T \neq \varphi$ and not 포함
여기서 $MBR_Q \cap MBR_T$는 교차로 정의된다.

Δ_sum(Q)의 결과 값을 얻기 위해 Δ-트리를 검색할 때 루트노드를 맨 처음 방문하며 정의 4.5에 설명된 것 처럼 루트노드의 각 엔트리에 대해 MBR_Q와 MBR_T의 공간적인 관계(spatial relationship)가 조사되어진다. 함수 Δ_sum(Q)의 간단한 알고리즘은 아래에 보여진다.

Algorithm Δ_sum()
input : 영역-합 질의 Q, Δ-트리
output : 질의에 대한 Δ-트리에서의 계산된 결과 값
procedure :
① 깊이 우선으로 루트부터 시작해서 Δ-트리에 있는 노드들을 방문한다. 만일 더 이상 방문할 노드가 없을시 해를 돌려준다.
② 노드의 각 엔트리가 평가되어 질 때 각 엔트리의 MBR_T와 MBR_Q 사이에 관계는 3가지 경우가 있다. 이러한 경우들과 이에 대응하는 pruning 전략은 다음과 같다.

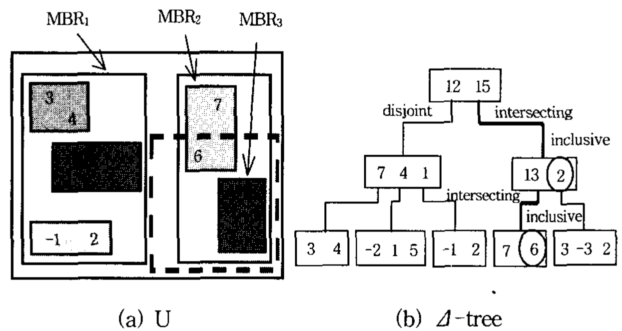
경우 1 (격리)
MBR_T에 연관된 엔트리는 질의 Q와 무관하다. 따라서 그 아래의 서브트리는 잘려나간다.

경우 2 (포함)
MBR_T에 연관된 엔트리의 ΣΔ가 해에 더해진다. ΣΔ는 이 엔트리에 대해서 정확한 값을 갖고 있기 때문에 더 이상 서브트리를 탐색할 필요가 없다.

경우 3 (교차)
이 경우에 영역-합을 위해 두 가지 선택을 가진다 : 정확한 해 또는 근사해. 정확한 해를 얻기 위해서 MBR_T에 속하는 모든 자식 MBR을 평가할 필요가 있다. 근사 해를 얻기 위하여 우리는 MBR_T에 연관된 ΣΔ의 근사 값을 계산하여 해에 더한다. 어떻게 근사 해를 얻는 가에 대한 자세한 설명이 5장에서 논의된다. 이 엔트리 아래의 서브트리에 대해서 알고리즘 Δ_sum()이 재귀적으로 수행된다.

정의 4.6 (그림 6)에 보여지는 것과 같이, 영역-합 질의 Q(2 : 5, 4 : 7)가 주어질 때(점선 박스), MBR_1은 격리, MBR_2는 교차, 그리고 MBR_3는 포함 관계이다. 그러므로, 함수 Δ_sum(2 : 5, 4 : 7)의 해가 8인 것을 알 수 있다. 따라서, 함수 Sum(2 : 5, 4 : 7)를 다음과 같이 완성할 수 있다. 즉,

$$\begin{aligned} Sum(2:5, 4:7) &= PC[5, 7]PC[1, 7]PC[5, 3] \\ &\quad + PC[1, 3] + \Delta_sum(2:5, 4:7) \\ &= (204 - 66 - 91 + 30) + (6 + 2) = 85. \blacksquare \end{aligned}$$



(그림 6) 다이나믹 업데이트 큐브 U와 U에 대응하는 Δ-tree.

4.4 업데이트

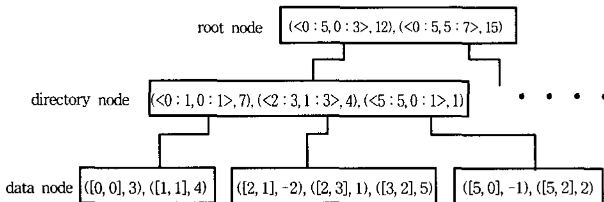
데이터 큐브 안의 셀의 값이 변경될 때, PC는 전혀 영향을 받지 않는다. 대신 Δ-트리 안의 적절한 위치에 있는 값을 변경하기만 하면 된다. 셀의 값이 변경되면 Δ-트리에 어떻게 영향을 주는지 알아보자. 업데이트 요청은 (P, Δ)의 형태로 주어진다. 여기서 P는 위치 인덱스이고 Δ는 변경된 셀의 이전 Δ값과의 차이이다. 업데이트의 첫 번째 단계는 업데이트될 서브트리를 찾는 것이다. 업데이트될 서브트리를 선택하는 작업은 R*-트리에 에서의 업데이트와 동일하다. 즉 업데이트될 서브트리를 반복적으로 찾음으로써 업데이트 요청이 반영될 데이터 노드가 최종적으로 선택된다. 한번 데이터 노드가 지정되면 그 노드 안에 위치 P의 데이터 엔트리가 있는지 검사한다. 이때 두 가지 경우가 있다.

경우 1(위치 P가 노드에 존재할 때)

이 경우에는 데이터 엔트리 (P, Δ_{OLD})에 업데이트 하는 경우이다. 여기서 Δ_{OLD}는 P위치의 현재의 Δ값이다. Δ값은 Δ_{OLD}에 더해진다. 그리고 트리에 있는 모든 데이터 노드의 조상 노드들에 대해서 $(\Sigma\Delta)_{ancestor} = (\Sigma\Delta)_{ancestor} + \Delta$을 수행한다. 여기서 $(\Sigma\Delta)_{ancestor}$는 데이터 노드의 조상 노드의 $\Sigma\Delta$이다. 이러한 과정이 루트 노드까지 반복된다.

경우 2 (위치 P가 노드에 존재하지 않을 때)

만약에 위치 P가 존재하지 않을 경우에는 데이터 엔트리 (P, Δ)가 노드의 끝에 추가된다. 또한, 데이터 노드의 모든 조상 노드에 대해서 경우 1에서와 같이 $(\Sigma\Delta)_{ancestor} = (\Sigma\Delta)_{ancestor} + \Delta$를 수행한다. 때때로 삽입 과정 중에 데이터 노드 안의 데이터 엔트리의 갯수가 한도를 초과하는 경우에는 오버플로우가 발생할 수 있다. 이 경우에는 노드가 두 개의 노드로 분리되는데 R*-트리의 삽입/분리 알고리즘이 그대로 적용될 수 있다. R*-트리의 삽입/분리 알고리즘에 대한 자세한 사항은 [2]을 참조하면 되므로 생략한다. 노드 S가 S_1과 S_2로 분리된다고 가정하자. 그러면 S_1과 S_2의 부모 노드의 $\Sigma\Delta$를 다시 계산하여야 할 필요가 있다. S_1 과 S_2의 부모 노드의 변경된 $\Sigma\Delta$는 모든 조상 노드들에 대해서 루트 노드까지 반영된다.



(그림 7) (그림 6)(b)의 Δ-트리의 상세한 구조

예제 4.7 (그림 5)(a)에서 셀 A[2, 3]의 값이 4에서 6으로 변경되었을 때의 갱신 과정을 생각하자. 이 것은 갱신 요청 ([2, 3], 2)을 Δ-트리에 반영하는 과정이다. (그림 7)에서 보는 바와 같이 엔트리 (<0 : 5, 0 : 3>, 12)가 먼저 루트 노드에서 선택된다. 트리를 아래로 탐색해 내려가면서 중간 노드에 있는 엔트리(<2 : 3, 1 : 3>, 4)가 선택된다. 위치 [2, 3]에 있는 데이터 엔트리는 엔트리 (<2 : 3, 1 : 3>, 4)가 가리키는 노드 안에 있으므로 Δ값 2는 이전의 값인 1에 더해져서 3이 된다. 데이터 노드의 $\Sigma\Delta$값을 변경한 후에 조상 노드들의 값들이 각각 (<2 : 3, 1 : 3>, 4)에서 (<2 : 3, 1 : 3>, 6)로, (<0 : 5, 0 : 3>, 12)에서 (<0 : 5, 0, 3>, 14)로 바뀐다. ■

4.5 다이나믹 업데이트 큐브의 복잡도

다이나믹 업데이트 큐브는 이전의 방법들[6, 7, 9, 11]에 비해 상당한 효율성을 제공하여 준다. Δ-트리 안에 한 개의 셀을 갱신하는 시간 복잡도는 $O(\log N_u)$이다. 여기서 N_u는 변경된 셀들의 개수이다. 일반적으로 데이터 큐브의 전체 셀의 개수에 비해서 변경된 셀의 개수는 매우 적다. 복잡도 $O(\log N_u)$는 트리에서 하나의 패스(path)로 내려가며 탐색하는 길이에 해당한다. <표 1>은 여러 가지 방법들의 복잡도를 비교한 표이다. 여기서, $N = n^d$이라고 가정하였고 n은 각 차원에 있는 셀들의 개수이다. <표 1>에서 보는 바와 같이 다이나믹 업데이트 큐브의 크기는 원래의 데이터 큐브의 크기에 비해서 매우 작기 때문에, 즉 $N_u << N$이다. 우리가 제안한 방법이 다른 방법들에 비해서 월등히 업데이트 성능이 좋다는 것은 자명하다.

〈표 1〉 시간 복잡도의 비교

방 법	갱신 시간 복잡도
Prefix Sum[9]	$O(n^d)$
Relative Prefix Sum[7]	$O(n^{d/2})$
Dynamic Data Cube[6]	$O(\log^d n)$
Dynamic Update Cube	$O(\log N_u)$

<표 1>의 시간 복잡도의 비교와 같이, <표 2>는 차원(표에서 d)이 2, 4, 8일 때와 각 차원의 크기가 10^1과 10^2일 때 여러 가지 방법들의 업데이트 비용에 대한 비교를 보여준다. 예를 들면 d가 4이고 n이 10^2일 때, 데이터 큐브의 전체 크기 $N(= n^d)$은 10^8이다. 이때 Δ-트리의 팬-아웃(fan-out)이 10이라고 가정하였다. 즉, 우리의 방법의 복잡도에서 로그(log)의 밑 수가 10이다. 또한 동적 데이터 큐브에 대해서 로그의 밑 수로서 10을 사용했다. 대개의 경우에 N_u은 N의 1% 정도 라고 생각된다. 따라서 N_u가 N의 0.1%, 1%, 10%인 경우에 대해서 우리의 방법을 평가했다. <표 2>에서 보여준 결과에서 알 수 있듯이, 우리의 방법이 다른 방법들에 비해 월등히 업데이트 성능이 좋다.

〈표 2〉 다양한 방법들의 업데이트 비용과의 비교

n	d	$N = n^d$	Prefix -Sum	Relative PS	Dynamic Data Cube	Dynamic Update Cube		
						N_u = 0.001N	N_u = 0.01N	N_u = 0.1N
10	2	10^2	10^2	10^1	11	-	1	1
	4	10^4	10^4	10^2	118	1	2	3
	8	10^8	10^8	10^4	14064	5	6	7
100	2	10^4	10^4	10^2	43	1	2	3
	4	10^8	10^8	10^4	1897	5	6	7
	8	10^{16}	10^{16}	10^8	3600406	13	14	15

RPC와 같이 PC의 변형된 형태에 기반한 이전 방법들은 업데이트 비용을 개선시키기 위해서 질의 비용이 증가하는 것을 감수한다. 우리 방법은 Δ-트리를 사용하여 매우 효율적인 업데이트 성능을 보여주지만 질의 처리에 약간의 추가 비용이 있다. 따라서 이전 방법이나 우리의 방법은 모두

프리픽스 섬 방법과 비교하여 추가의 비용이 있다. 6장에서의 실험은 우리의 방법에 의한 질의 처리가 매우 효율적임을 보여준다.

5. 하이브리드 방법

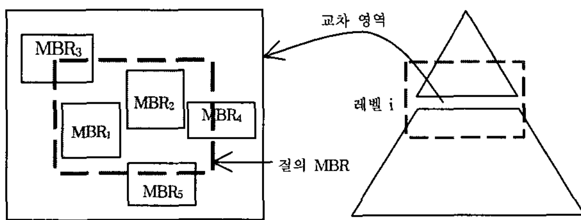
실제 OLAP 환경에서 사용자들은 대개 대화식 질의를 반복 실행하여 경향이나 패턴 또는 특이한 데이터 변화 들을 찾고자 한다. 이 때에는 최대한 빠르게 응답을 받기 위해서 근사값의 결과를 원할 수도 있다. 이 절에서는 질의의 전체 실행 비용을 줄이기 위해서 근사 결과와 정확한 결과를 혼합한 하이브리드 방법을 제안한다. 이것은 의사 결정 지원 시스템과 같이 시간이 많이 걸리는 정확한 결과 보다는 근사 결과라도 빨리 얻기를 원하는 응용 들에서 큰 장점이 된다. 우리는 근사 기법을 제안하고 이 방법이 어떻게 적은 추가 실행 비용으로도 근사값의 에러를 줄일 수 있는지를 설명한다.

구간 합 질의를 수행할 때, Δ-트리를 부분적으로 검색하면 근사 결과를 얻을 수 있다. 즉, 루트에서 하위 노드를 검색 할 때, 단말 노드까지 가지 않고 내부 노드까지만 검색한다. Δ-트리를 검색할 때 영역-합 질의의 결과에 연관된 MBR들이 여러개 있다. 이들 MBR들을 다음과 같이 두 가지로 분류한다.

① 포함된 MBR : MBR_i(i = 1, …, m), m은 포함된 MBR의 개수

② 교차된 MBR : MBR_j(j = m+1, …, n), n-m은 교차된 MBR의 개수

예제 5.1 (그림 8)에서와 같이, Δ-트리의 레벨 i에서의 교차 영역(cross-section)을 볼 수 있다. 즉 MBR_1과 MBR_2는 포함된 MBR들이며 MBR_3, MBR_4와 MBR_5는 교차된 MBR 들이다. ■



(a) Δ-tree의 레벨 i에서의 MBR 들 (b) Δ-트리

(그림 8) 질의 MBR과 Δ-트리의 레벨 i의 MBR들의 모양

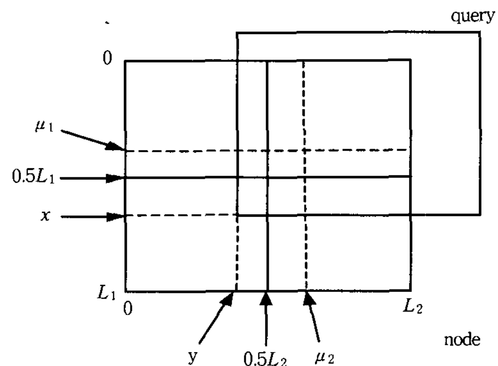
$(\Sigma\Delta)_i$(i = 1, …, m)를 i번째 포함된 MBR의 값이라고 하자. 그리고 $(\Sigma\Delta)_j$(j = m+1, …, n)는 각 교차된 MBR의 값이라고 하자. 그러면 Δ-트리의 레벨 i에서의 구간 합 질의의 결과는 다음 식으로 근사 될 수 있다.

$$\text{Approx_sum}(Q) = \sum_{i=1}^{m}(\Sigma\Delta)_i + \sum_{j=m+1}^{n}\left(\frac{Vol(MBR_j \cap MBR_Q)}{Vol(MBR_j)} \times (\Sigma\Delta)_j\right) + PC_sum(Q)$$

이 절에서는 근사 기법을 개선시키기 위해서 평균 위치들의 목록을 사용하는 방법을 제안한다. 좀 더 정확한 근사 결과를 얻기 위해서 질의의 영역을 재조정한다. 그리고 재조정하지 않고 구한 근사 값과 재 조정하고 구한 근사값의 차이값을 사용하여 더 검색하여야 하는 노드를 결정하는데 이용한다. 예제 5.2에서 이 방법에 관해 자세히 설명한다.

예제 5.2 (그림 9)에서, 교차되는 영역은 (0 : x, y : L_2)이다. μ_1, μ_2을 계산하고 좀더 정확한 근사값을 얻기 위해 영역을 재조정한다. 노드의 좌우 변을 생각하자. [0, μ_1]은 노드에 포함된 데이터들의 반을 포함하고 있다. 데이터들이 균일하게 분포되어 있고 μ_1이 평균의 위치일 때, [0, x]안의 데이터들을 포함하는 [0, α]의 α를 구한다. 그러면 $\frac{x-\mu_1}{L_1-\mu_1} = \frac{\alpha-0.5L_1}{0.5L_1}$로부터 $\alpha = \frac{L_1}{2}\left(1+\frac{x-\mu_1}{L_1-\mu_1}\right)$의 식을 구할 수 있다. 따라서 0 : x는 0 : $\frac{L_1}{2}\left(1+\frac{x-\mu_1}{L_1-\mu_1}\right)$로 재조정된다. y : L_2의 재조정은 비슷한 방법을 사용하여 구하면 $\frac{L_2}{2} \times \frac{y}{\mu_2}$: L_2로 된다. 그리고 재조정하여 구한 근사값과 재조정하지 않고 구한 근사값과의 차이를 사용하여 다음 하위 레벨에서 검색해야 되는 노드를 찾는다. 즉, 다른 것들 보다 값의 차이가 큰 노드들을 선택한다. ■

레벨 i안의 교차되는 노드들을 검색할 때, 큰 차이를 갖는 레벨 i의 노드들을 레벨 i-1까지 검색하면 추가 비용을 적게 들이고도 에러를 상당히 줄일 수 있다.

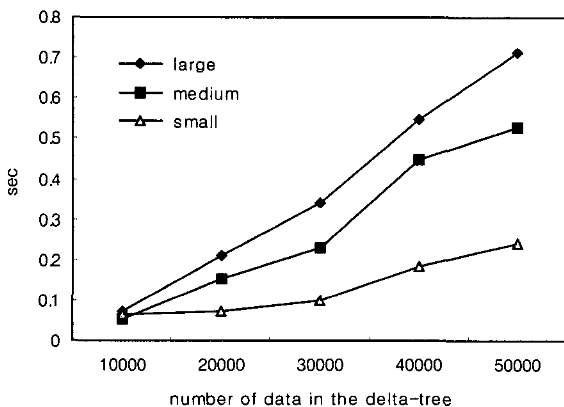


(그림 9) 질의 재조정

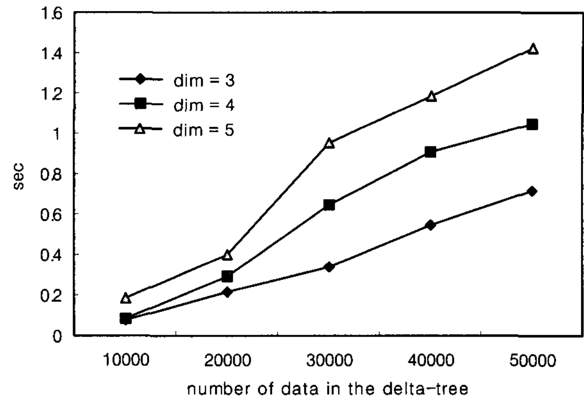
6. 실험 및 평가

본 절에서는 성능을 실험한 결과를 보여 준다. 본 논문에서 제안된 방법은 PC와 Δ-트리를 함께 사용하는 방법이다. 즉, 업데이트는 Δ-트리 만을 사용하고 주기적으로 PC에 반영해 주므로 Δ-트리로의 업데이트 성능을 측정하였다. 질의 성능에 대해서는 PC와 Δ-트리를 함께 고려하였고, 질의 정확도에 대해서는 PC는 정확하므로 Δ-트리 만으로 실험하였다. 질의 성능을 측정할 때에는 페이지 접근 횟수와 시간으로서 측정하였다. Δ-트리는 R*-트리를 수정하여 구현하였는데, 트리의 깊이를 깊게 하여 근사 값을 구할 수 있도록 하기 위하여 노드의 크기를 600바이트로 제한하였다. 사용한 데이터는 균일분포(Uniform distribution) 데이터와 Zipf 데이터를 사용하였다. Zipf 데이터의 z 파라미터는 차원에 관계없이 일정한 값을 주었다 (z = 0.9). 데이터의 차원은 2, 3, 4, 5차원의 데이터를 사용하였다. 그리고 각 차원의 크기(cardinality)는 2차원 일 때 1024, 3차원 일 때 512, 4차원 일 때 128, 5차원 일 때 64로 하였다. Δ-트리에 들어가는 데이터의 갯수는 1만개에서 5만개로 하였다. 질의 선택률(selectivity)의 크기에 따라서 3가지의 질의를 사용하였다 : large(= 0.1), medium(= 0.05), small(= 0.01). 모든 실험은 메인 메모리 256M, 10G 하드디스크를 갖춘 Sun Untra 2 웍스테이션에서 실험하였다. 실험결과의 오차는 백분율 오차를 사용하였고 30개의 질의를 실행시킨 결과의 평균을 구하였다.

(그림 10)은 3차원에서 large, medium, small 크기의 질의 수행 성능을 보여 준다. (그림 11)은 medium 크기의 질의를 각각 3차원, 4차원, 5차원 데이터에 대해 실행시킨 결과를 보여준다. 질의 수행시에 Δ-트리의 데이터 노드인 레벨 0까지 검색한 경우로서 질의의 결과 값이 정확한 경우이다. 성능은 수행 시간으로 측정하였고 3차원인 경우에는 1초미만의 빠른 수행 시간을 보여주고 있다.

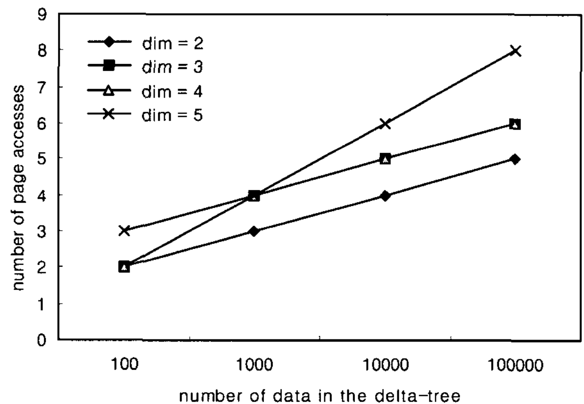


(그림 10) 정확한 해를 구할 때 질의 성능(레벨이 = 0), 차원 = 3, 질의 크기 = large, medium, small



(그림 11) Uniform 분포에서 정확한 해를 구할 때 질의 성능(레벨이 = 0), 차원 = 3, 4, 5, 질의 크기 = medium

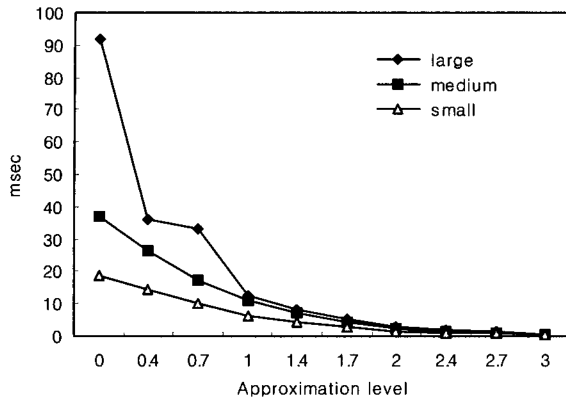
(그림 12)는 변경된 셀의 값을 Δ-트리에 삽입하는 성능을 보여준다. X축은 기존에 Δ-트리에 들어 있는 데이터의 개수이고 Y축은 1개의 값을 삽입할 때 방문되는 페이지의 개수를 보여주는데 Δ-트리의 깊이와 일치한다. 그림에서 보여주는 바와 같이 성능이 O(logNu)인 것을 확인할 수 있다(Nu는 변경된 셀의 개수).

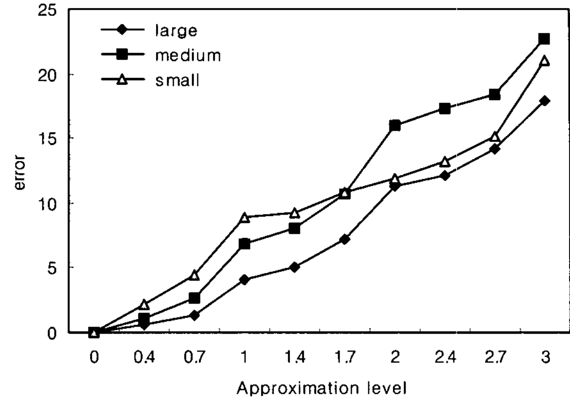


(그림 12) 삽입 성능

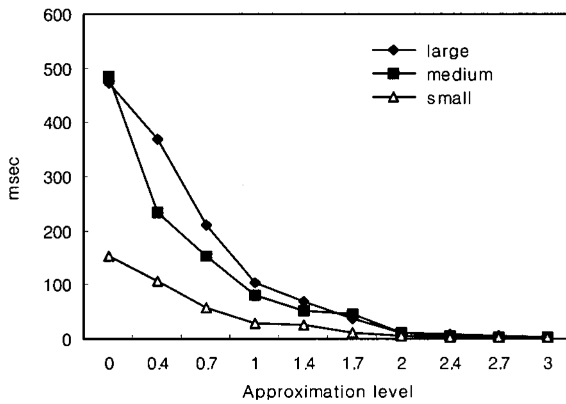
(그림 13)은 4차원의 Uniform한 데이터 10000개가 Δ-트리 안에 저장되어 있는 상태에서 질의 수행을 근사 기법을 적용하였을 때의 성능과 에러율을 보여주는 그림이다. X축의 값은 근사방법을 수행한 트리의 레벨을 나타내고 있다. 여기서 단말 노드인 데이터 노드의 레벨이 0이고 하나 상위의 노드는 아래 노드보다 1 만큼 큰 값을 가진다. 근사 레벨이 0이라는 것은 데이터 노드까지 검색한다는 의미이며, 1, 2는 각각 레벨 1, 2의 노드까지 검색한다는 것을 의미한다. 근사 레벨이 0.4(= 0.60 + 0.41)라는 것은 레벨 1의 노드 중에서 40%의 노드는 레벨 1까지 검색 하고 60%의 노드는 레벨 0까지 검색 한다는 것을 의미 한다.

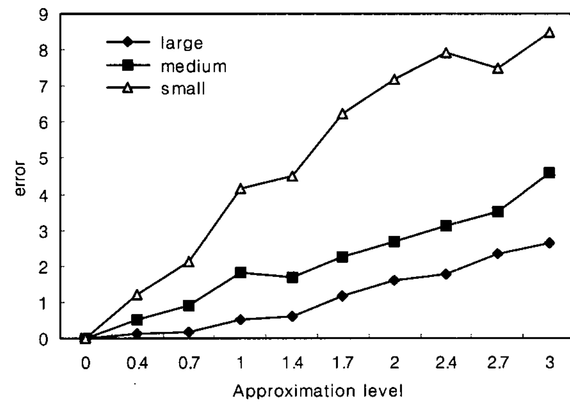
유사하게 근사 레벨이 1.7(= 0.31 + 0.72)이라는 것은 레벨 2의 노드 중에서 70%의 노드는 레벨 2까지 검색하고 30%의



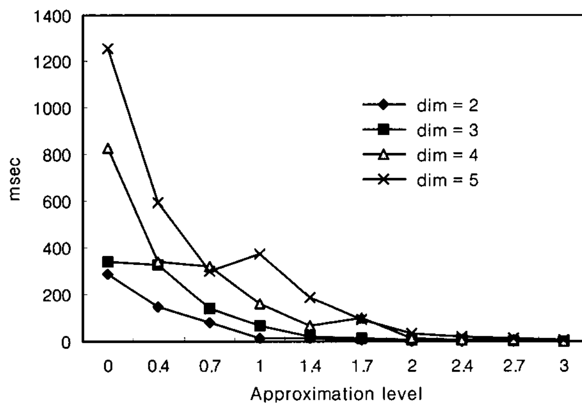


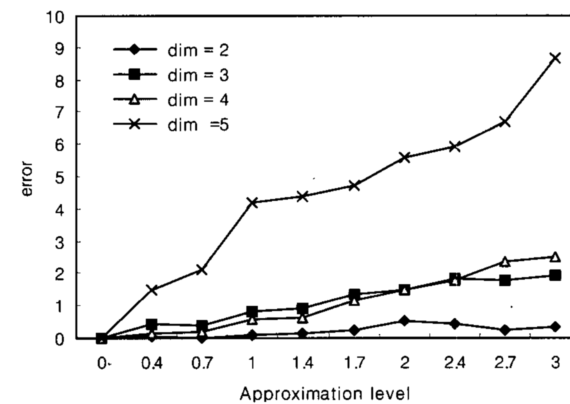
(그림 13) Uniform 분포 데이터에서 차원 = 4, 질의 크기 = large, medium, small, 데이터 수 = 10000 일 때 근사 해를 구하는 성능 및 에러율





(그림 14) Uniform 분포 데이터에서 차원 = 4, 질의 크기 = large, medium, small, Δ-트리의 데이터 수 = 50000 일 때 성능 및 에러율



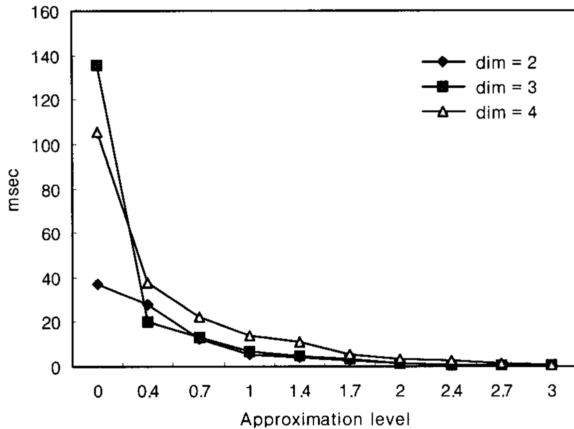


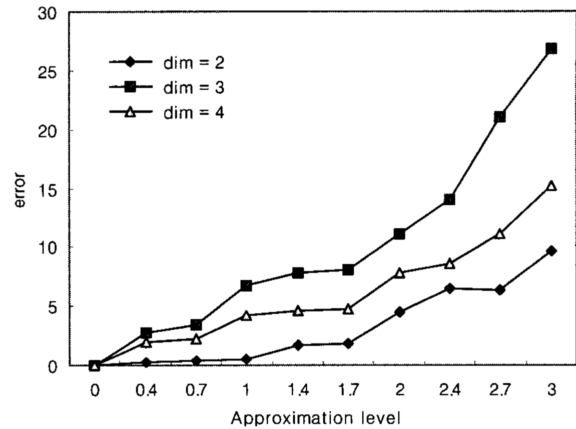
(그림 15) Uniform 분포 데이터에서 차원 = 2,3,4,5, 질의 크기 = large, Δ-트리의 데이터 수 = 50000 일 때 성능 및 에러율

노드는 레벨 1의 노드까지 검색한다는 의미이다. 이때 근사값의 차가 큰 노드에 대해서 검색이 한 레벨 더 행해진다. (그림 13)에서 보여 주는 바와 같이 근사레벨을 점점 높임에 따라서 검색되는 페이지와 시간은 급격히 감소함을 볼 수 있다. 그런데 상대적으로 에러율은 조금 증가함을 볼 수 있다. 따라서 근사레벨을 적절히 높이면 허용 가능한 정도의 에러율 안에서 매우 큰 성능의 향상을 얻을 수 있음을 볼 수 있다.

(그림 14)는 (그림 13)과 같은 조건에서 데이터의 개수를 50000로 늘린 경우를 보여준다. 여기서는 Δ-트리에 데이터의 수가 늘어날수록 에러율은 나빠지는 반면에 질의 성능은 좋아짐을 보여 준다.

(그림 15)는 Uniform 데이터 분포에서 데이터의 수가 50000, 질의 크기는 large 그리고 차원은 2에서 5차원까지 변화시





(그림 16) Zipf분포 데이터에서 차원 = 2,3,4, 질의 크기 = large, Δ-트리의 데이터 수 = 50000 일 때 성능 및 에러율

키면서 성능과 에러율을 보여주는 그림이다. (그림 16)은 Zipf 분포 데이터에서 데이터의 수가 50000일 때 차원을 2에서 4차원까지 변화시켰을 때 성능과 에러율을 보여준다.

6. 결 론

이 논문에서 우리는 최근 기업의 동적인 데이터 환경에 적합한 새로운 데이터 큐브를 제안 했다. 즉, 기존의 PC를 이용한 프리픽스 섬 방법의 단점인 업데이트시의 고비용을 해결하기위해 Δ-트리라고 부르는 계층 자료 구조를 탐구하였다. Δ-트리는 데이터 큐브의 업데이트된 정보 만을 별도로 관리함으로서 프리픽스 섬 방식에서 발생되는 업데이트 증식문제를 해결하였다. 또한, Δ-트리의 계층 구조의 장점을 이용하여, 근사 결과와 정확한 결과를 혼합한 하이브리드 방법을 제공함으로 질의의 전체 수행 비용을 줄일 수 있도록 했다. 이것은 의사 결정 지원 시스템과 같이 근사 결과라도 빨리 보기를 원하는 다양한 많은 응용에서 매우 유용하다. 우리는 여러가지 차원과 여러 크기의 질의들에 대해서 다양한 실험을 행하여 질의와 업데이트 효율성, 근사 결과의 정확성과 효율성을 평가하였다. 실험 결과는 우리의 방법이 업데이트와 질의 수행에서 매우 효율적이라는 사실을 보여 주었으며 제안된 하이브리드 방법을 사용하면 근사값의 에러를 합리적인 수준에서 유지하면서 질의 속도를 획기적으로 줄일 수 있음도 보여 주었다. 우리가 제안한 방법 역시 질의의 성능을 높이기 위해서 PC를 이용하는 데 PC를 사용하는 방법의 문제점은 PC를 저장하는 저장공간이 많이 든다는 데있다. 이와 같이 PC사용 함으로서 발생되는 부가적인 저장공간의 오버헤드를 개선시키는 방법을 개발하는 것이 앞으로 더 연구해야 할 과제이다.

참 고 문 헌

[1] S. Berchtold, D. Keim and H. Kriegel, The X-tree : an index structure for high dimensional data, Proceedings of Int'l Conference on Very Large Data Bases, India, pp.28-39, 1996.

[2] N. Beckmann, H. Kriegel, R. Schneider and B. Seeger, The R*-tree : an efficient and robust access method for points and rectangles, Proceedings of ACM SIGMOD Int'l Conference on Management of Data, New Jersey, pp.322-331, 1990.

[3] Alex Berson, Stephen J. Smith, Data WareHousing, Data Mining, & OLAP, McGrawHill, 1997.

[4] C.-Y. Chan, Y. E. Ioannidis, Hierarchical cubes for range-sum queries, Proceedings of Int'l Conference on Very Large Data Bases, Scotland, pp.675-686, 1999.

[5] E. F. Codd, Providing OLAP(on-line analytical processing) to user-analysts : An IT mandate, Technical report, E. F. Codd and Associates, 1993.

[6] S. Geffner, D. Agrawal, A. El Abbadi, The Dynamic Data Cube, Proceedings of Int'l Conference on Extending Database Technology, Germany, pp.237-253, 2000.

[7] S. Geffner, D. Agrawal, A. El Abbadi, T. Smith, Relative prefix sums : an efficient approach for quering dynamic OLAP Data Cubes, Proceedings of Int'l Conference on Data Engineering, Australia, pp.328-335, 1999.

[8] A. Guttman, R-trees : a dynamic index structure for spatial searching, Proceedings of ACM SIGMOD Int'l Conference on Management of Data, Massachusetts, pp.47-57, 1984.

[9] C. Ho, R. Agrawal, N. Megido, R. Srikant, Range queries in OLAP Data Cubes, Proceedings of ACM SIGMOD Int'l Conference on Management of Data, pp.73-88, 1997.

[10] J. Han, M. Kamber, Data Mining Concepts and Techniques, Morgan Kaufmann Publishers, 2001.

[11] W. Liang, H. Wang, M. E. Orlowska, Range Queries in dynamic OLAP data cubes, Data & Knowledge Engineering 34, pp.21-38, 2000.

[12] T. Sellis, N. Roussopoulos and C. Faloutsos, The R+-tree : a dynamic index for multi-dimensional objects, Proceedings of Int'l Conference on Very Large Data Bases, England, pp.507-518, 1987.

전 석 주

e-mail : chunsj@ansan.ac.kr

1987년 경북대학교 전자공학과 컴퓨터공학 전공(학사)

1989년 경북대학교 대학원 전자공학과 컴퓨터공학전공(석사)

2002년 한국과학기술원 정보및통신공학과 (박사)

1989년~1995년 현대중공업 중앙연구소 주임연구원

1997년~현재 안산1대학 인터넷정보과 조교수

관심분야 : 데이터마이닝, 데이터 웨어하우스와 OLAP, 멀티미디어 데이터베이스 등

이 주 홍

e-mail : juhong@inha.ac.kr

1983년 서울대학교 컴퓨터공학과(학사).

1985년 서울대학교 컴퓨터공학과(석사).

2001년 한국과학기술원 정보및통신공학과 (박사)

1985년~1989년한국통신 사업지원단 전임 연구원

1989년~1993년 한국아이비엠 소프트웨어 연구소 선임프로그래머

2001년~현재 인하대학교 컴퓨터공학부 조교수

관심분야 : 소프트컴퓨팅과 데이터마이닝, 데이터 웨어하우스와 OLAP, 데이터베이스, 웹 서비스, 정보검색