큐브 계산에서 I/O 비용을 줄이는 구간 기반 큐브 분할

(Range-based Cube Partitioning for Reducing I/O Cost in Cube Computation)

박 웅 제 † 정 연 돈 †† 김 진 녕 ††† 이 윤 준 †††† 김 명 호 ††††

(Woong Je Park) (Yon Dohn Chung) (Jin Nyoung Kim) (Yoon Joon Lee) (Myoung Ho Kim)

요 약 본 논문은 OLAP에서의 I/O 비용을 줄이는 큐브 계산 방법으로, 구간 기반 큐브 분할 기법을 제안한다. 제안하는 방법은 큐브 분할 단계들 사이에 존재하는 계산의 일부를 중복시켜 처리하는 방법을 통해 큐브 분할 작업의 I/O 성능을 향상시킨다. 계산의 중복을 위하여 제안하는 방법은 애트리뷰트의 단일 값이 아닌 애트리뷰트 값의 일정 구간을 기준으로 큐브를 분할한다. 분석과 실험을 통하여 제안하는 방법의 성능을 기존 큐브 분할 방법과 비교하여 보인다.

Abstract In this paper we propose a method, called the range-based cube partitioning (RCP) method, for reducing I/O cost of cube computation in OLAP. The method improves I/O performance of cube partitioning process by overlapping some computation between partitioning stages. For overlapping the computation, the method partitions the cube based on the ranges of attribute values, not the points of attribute value. Through analysis and experiments, we show the performance of the proposed method with comparison of the previous cube partitioning method.

1. 서 론

OLAP 시스템은 사용자가 다차원 정보에 직접 접근하여 방대한 양의 데이타를 빠르게 분석할 수 있도록 지원하는 시스템이다[5, 4]. OLAP 시스템은 여러 데이타베이스로부터 긴 시간에 걸쳐 형성된 데이타를 통합한 데이타웨어하우스를 기반으로 한다. 따라서 수 백 기가 바이트(GB)에서 테라 바이트(TB)에 이르는 매우 방대한 데이타에 접근하는 특징을 갖는다. 그리고 의사 결정을 위해 요청되는 질의들은 대부분 데이타 집합의 전체적인 경향이나 정보들을 분석하는 형태로서 복잡한 집계 연산(aggregation)을 포함하는 경우가 많다. 이러한 집계 연산 질의들은 처리 시 긴 수행 시간이 소요되기 때문에, 이에 대한 효율적인 처리가 OLAP 시스템의 성능을 좌우하는 요소가 된다[4].

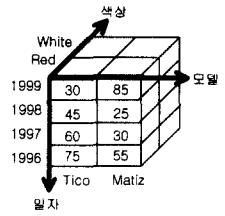
OLAP에서는 사용자가 데이타를 다양한 차원 관점에서 분석할 수 있도록 다차원 데이타 모델을 사용한다. OLAP에서 다루는 다차원 데이타를 N 차원 데이타 큐브라고 부르며, 데이타 큐브는 여러 집계 함수의 대상이 되는 측정(measure) 애트리뷰트(attribute)들과, 하나의 측정 애트리뷰트의 값을 유일하게 결정하는 N 개의 차원(dimension) 애트리뷰트들로 구성된다. OLAP 분야의 대부분의 연구들은 이와 같은 다차원 데이타 모델에서 분석 작업이 효율적으로 수행되기 위한 방법에 대해 다루고 있다. 이러한 연구들 중의 하나로 [8]에서는 다차원 데이타 분석을 지원하기 위해 기존의 SQL에서 제공되는 group-by 연산자를 확장한 큐브(cube) 연산자를 제안하였다.

큐브 연산자는 차원 애트리뷰트 간의 가능한 모든 조합에 대하여 group-by를 계산한다. 즉, N개의 차원 애

본 연구는 BK21 산학협력자금의 지원으로 수행되었음.

† 비 회 원 : 한국과학기술원 전자전산학과 전산학전공
ujpark@dbserver.kaist.ac.kr
†† 비 회 원 : 한국과학기술원 전자전산학과 전산학전공 연구 교수
ydchung@dbserver.kaist.ac.kr
††† 비 회 원 : (주)데이콤 연구원
jnkim@dbserver.kaist.ac.kr
†††† 종신회원 : 한국과학기술원 전자전산학과 전산학전공 교수
yjlee@dbserver.kaist.ac.kr
mhkim@dbserver.kaist.ac.kr

논문접수 : 2000년 8월 1일
심사완료 : 2001년 7월 14일



(a) 3차원 데이타 큐브

```
SELECT 모델, 색상, 일자, SUM(판매량)
FROM   R
CUBE BY 모델, 색상, 일자
```

(b) 3차원 큐브 질의

그림 1 색상, 일자, 모델의 3차원 데이타 큐브와 관련 큐브 질의

트리뷰트로부터 계산되는 큐브는 2^N개의 조합에 대한 group-by를 포함한다. 그림 1 (a)와 같이 모델, 색상, 일자 3개의 차원 애트리뷰트와 판매량 1개의 측정 애트리뷰트로 구성되는 릴레이션 R로부터 모델별 판매량, 모델별 색상별 판매량, 모델별 색상별 일자별 판매량 등과 같은 다양한 차원 관점에서의 집계 함수를 적용하는 큐브 질의는 그림 1 (b)와 같이 표현된다.

큐브 연산자는 N개의 차원 애트리뷰트들의 모든 가능한 조합에 대한 group-by와 집계 연산을 포함하기 때문에 질의 처리에 많은 비용이 들게 된다. 따라서 큐브 연산자를 효율적으로 처리하여 빠른 On-Line 분석 작업을 지원하기 위한 연구들이 진행되어 왔다[1, 11, 12, 2, 3]. 기존의 연구들에서는 큐브 연산자 문제를 해결하기 위해 큐브가 포함하는 각 group by 연산들을 큐보이드(Cuboid)로 정의하고, 큐브를 이 큐보이드들을 노드로 갖는 하나의 래티스(Lattice) 구조로 보았다. 큐브 래티스는 각 노드(큐보이드)간의 계산 가능한 관계에 의해 연결관계를 갖는다. 그림 1 (b)의 큐브 질의는 그림 2의 래티스 구조로 표현된다. (편의상, 모델, 색상, 일자 애트리뷰트를 A, B, C로 표기하였다.) 큐브 연산자를 처리하는 방법은 이 래티스 구조에 표현된 노드간의 관계를 이용해 래티스를 탐색해가는 과정이며, 어떻게 효율적으로 모든 노드들을 탐색할 것인가 하는 것이 바로 효율적인 큐브 연산자의 처리 방법에 대한 연구이다.

기존의 연구들은 큐브 연산자를 효율적으로 처리하기 위해, 메모리에 로딩되는 큐보이드들 사이의 연관성을 유지하여 되도록 많은 group by들이 중첩 계산되도록 하는 방법을 제안하였다. 그리고 I/O 비용을 줄일 수 있

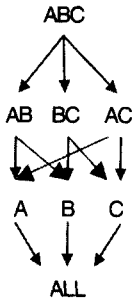


그림 2 큐브 래티스

는 방향으로 메모리에 같이 로드되는 큐보이드들의 연관성을 정의하였다. OLAP에서 사용되는 기본 테이블의 크기가 일반적으로 매우 크기 때문에 큐브를 계산할 때 I/O의 비용은 전체 질의 처리에서 중요한 요소가 된다. 특히 차원의 수가 큰 데이타 큐브에서는 전체 공간에 비해 데이타들의 밀도가 낮기 때문에 각 group by의 결과가 기본 테이블의 크기와 크게 달라지지 않아서 I/O 비용의 비율이 더욱 커진다. 따라서 I/O 횟수를 최대한 줄일 수 있는 질의 처리 방법이 필요하다.

본 논문에서는 다차원 데이타로부터 큐브를 계산할 때, I/O 효율성을 고려한 구간 기반 큐브 분할 (Range-based Cube Partitioning: RCP) 방법을 제안한다. 본 논문에서 제안하는 방법은 기존의 연구들이 사용하는 단계별 처리 방법에서 현 단계의 정보를 이용하여 다음 단계의 I/O를 최대한 적게 발생시키는 방법이다. 이것은 현 단계의 정보를 재사용하여 다음 단계의 작업 일부를 미리 수행하는 원리를 사용한다. 기존 연구들이 메모리 내에서의 효율적인 큐브 계산에 중점을 둔 것과 달리, 제안하는 방법은 메모리를 큐브의 계산을 위한 부분과 다음 단계의 I/O 비용을 개선하기 위한 부분으로 나누어 사용함으로써 메모리 활용도를 높이도록 한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 큐브 계산을 위해 기존에 제안된 큐브 분할 방법에 대해 소개하고, 3장에서는 I/O의 효율성을 높이는 구간 기반 큐브 분할 방법을 제안한다. 4장에서는 제안하는 방법의 성능을 실험을 통하여 보이고, 기존 연구와 비교 분석한다. 끝으로, 5장에서 결론을 맺도록 한다.

2. 배경: 큐브 분할(Cube Partitioning)

큐브의 계산에 있어 가장 기본적인 방법은 2^N개의 조합에 대한 group-by를 모두 개별적으로 구하는 것이다. 하지만 이 방법은 각 큐보이드를 매번 독립적으로 읽어

들이기 때문에 매우 비효율적인 방법이다. 이를 해결하기 위해 그림 2에서 보인 큐브 래티스 구조에서 나타나는 큐보이드들 사이의 연관성을 이용하여, 큐보이드들의 계산을 중첩시키는 '중첩 방법' (overlap method)이 제안되었다 [6]. 하지만, 이 방법은 다차원 데이타 큐브에서 데이타의 밀도가 낮아지는 경우, 그 효율이 낮아지는 문제점을 가지고 있다. 이를 위해 데이타 큐브를 분할하는 방법들이 제안되었다 [2, 3].

Partitioned CUBE 방법[2]은 모든 큐보이드를 한번에 메모리에 로드하여 계산할 수 없는 경우, 주어진 메모리 크기에 맞추어 데이타를 특정 애트리뷰트를 기반으로 분할하는 방법이다. 즉, 주어진 메모리 내에서 부분 일치 순서의 분할 크기를 고려해 동시에 계산하는 큐보이드 집합을 결정하는 것이 아니라, 기준 애트리뷰트를 정하여 그 애트리뷰트에 따라 데이타를 분할함으로써 주어진 메모리 크기에 맞도록 한 방법이다. 이로써 메모리 크기에 맞게 잘린 분할 단위별로 계산 가능한 모든 큐보이드들을 동시에 계산할 수 있게 된다. 아울러 메모리에 읽혀진 데이타로부터 group by들을 효율적으로 계산하기 위한 Memory CUBE 방법을 제안하였다. 이는 각 큐보이드를 계산할 정렬 경로를 구하면, 각 정렬 경로내의 큐보이드들이 계산의 상당 부분이 공유된다는 점을 이용한 방법이다.

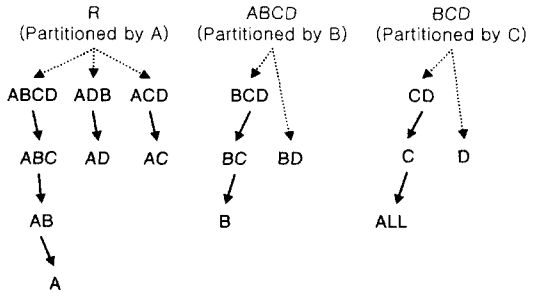


그림 3 Partitioned Cube를 이용한 큐브 계산

그림 3은 차원 애트리뷰트가 4개인 큐브에 대한 Partitioned CUBE 방법을 보이고 있다. 입력 릴레이션 (base relation) R이 메모리에 읽을 수 없을 정도로 크기 때문에 처음 단계에서는 A에 대해 분할을 하고 메모리에 읽은 후 정렬 기반의 Memory CUBE 방법으로 관련 큐보이드들을 계산한다. 그림에서 점선 화살표는 메모리에서의 정렬 연산이 필요함을 나타내고 실선 화살표는 정렬 경로 상에서 공통 접두(prefix) 애트리뷰트를 이용하여 집계 연산을 계산함을 나타낸다. 예를 들어 메모리에 있는 하나의 분할에 대해 ABCD, ADB, ACD로 정렬하여 각 큐보이드를 계산하면서 동시에 ABC, AD, AC를 별도의 정렬 작업 없이 계산할 수 있다. 다음 단계에서는 ABCD의 큐보이드를 B에 대해 분할한 뒤 큐보이드들을 계산한다. 이 과정을 모든 큐보이드의 계산을 완료할 때까지 반복한다.

[2]의 Partitioned CUBE/Memory CUBE 방법이 큐보이드 래티스에서 하향식 접근 방식을 사용한 반면, [3]에서는 상향 계산 (Bottom Up Computation: BUC) 방법을 제안하였다. Partitioned CUBE와 같이 분할 방법을 기반으로 하며, 메모리 내의 연산인 Memory CUBE 방법을 개선한 방법으로, ABCD->ABC->AB->A 의 순서가 아닌 A->AB->ABC->ABCD의 상향식 순서를 채택하고 있다. I/O 비용은 동일하면서, 메모리 내에서의 연산 효율성이 높은 방법으로 알려져 있다. 본 논문에서 제안하는 방법은 상향식 혹은 하향식 방법에 모두 적용할 수 있는 방법으로, 편의상 하향식 방법을 기준으로 설명하기로 한다.

3. 구간 기반 분할 방법

기존의 큐브 분할 방법에서는 각 애트리뷰트를 기준으로 하는 각 분할 단계들 사이에서는 독립적인 연산을 수행하고 있다. (그림 3에서는 '애트리뷰트 A에 의한 분할 단계', '애트리뷰트 B에 의한 분할 단계', 그리고 '애트리뷰트 C에 의한 분할 단계'의 3단계 분할 절차를 사용하고 있다.) 만약 한 단계에서 읽은 데이타를 다음 단계의 계산에 재사용할 수 있다면 분할 방법의 각 단계에게 발생하는 I/O 비용을 최소화 시킬 수 있을 것이다. 본 논문에서는 이를 위해서 주어진 메모리를 큐브의 계산을 위한 부분과 다음 단계의 분할 작업을 위한 부분으로 나눔으로써 I/O 비용을 고려한 개선된 분할 방법을 제안하고자 한다.

본 논문에서 제안하는 '구간 기반 큐브 분할 (Range-based Cube Partitioning: RCP)' 방법은 분할 단계들 사이에서 연산의 중첩 가능한 부분을 찾아내어 이를 활용하는 방법이다. 기존 큐브 분할 방법이 애트리뷰트의 값을 하나씩 분리하여 분할을 생성하는 반면, 제안하는 방법은 애트리뷰트 값의 일정 구간(range)을 기준으로 분할을 생성하게 된다. 제안하는 방법과 구별하기 위하여 기존 방법을 '단일 값 기반 큐브 분할(Point-based Cube Partitioning: PCP)' 방법이라고 부르기로 한다.

3.1 제안하는 방법의 기본 개념

기본적으로 큐브 분할 방법(cube partitioning)에 의해 나누어진 하나의 분할에는 큐브 연산 작업과 다음 단계의 분할 작업에 필요한 모든 데이타가 포함되어 있다. 예를 들어 분할 애트리뷰트 순서가 ABCD라고 하고 현재의 분할이 A에 대해서 이루어진다면, 계산되는 큐보이드들은 ABCD, ABC, ABD, ..., A이다. 그리고 다음 단계인 B에 대한 분할에서 계산되는 큐보이드들은 BCD, BC, BD, B이다. 다음 단계의 분할 작업과 큐브 연산을 위해서는 애트리뷰트 B, C, D의 정보가 필요한데, 이미 현재 읽은 분할내에 애트리뷰트 A의 값 이외에 애트리뷰트 B, C, D의 값이 포함되어 있다. 따라서 다음 단계의 분할에 도움이 될 수 있는 작업을 미리 수행하는 것이 가능해진다. (이 작업의 결과를 중간 결과라고 부르기로 한다.) 이를 위해 본 논문에서 제안하는 '구간 기반 분할 방법'에서는 주어진 메모리를 일반적인 분할 방법들에서 큐보이드를 계산하기 위해 사용하는 메모리 영역(M_C)과, 다음 분할 단계의 작업을 미리 수행하기 위한 메모리 영역(M_{RCP})으로 구분한다. 큐보이드를 계산하기 위한 영역에서 이루어지는 작업은 기존의 분할 방법들과 동일하며, 본 논문에서 제안하는 방법은 다음 분할 단계를 효과적으로 먼저 수행하는 부분에만 영향을 미치게 된다. 기존의 분할 방법과 비교하여 큐보이드 계산에 있어 차이가 없기 때문에, 메모리의 영역 설정에 있어 큐보이드 계산을 위한 영역의 설정에 우선권을 부여하게 된다. 따라서, 큐보이드 계산을 위한 메모리 영역의 설정에 따라 '다음 단계 계산을 위한 메모리 영역'의 설정이 불가능한 경우가 발생할 수 있으며, 이때에는 기존의 분할 방법과 동일하게 분할 작업이 수행된다.

3.2 알고리즘

기존 분할 방법의 큐브 계산은 분할 애트리뷰트에 대해 여러 단계로 이루어진다. 각 단계는 다시 '분할을 구성하는 과정'과 분할 애트리뷰트와 관계된 '큐보이드들을 메모리 내의 연산으로 구하는 과정'으로 구분할 수 있다. 이 두 과정을 비용 요소로 구분지어 보면, 메모리 내의 큐브 계산은 얼마나 빠르게 여러 큐보이드들을 동시에 처리할 수 있는가 하는 시간의 비용이 우선시 된다. 이와는 달리 분할 연산 과정은 데이타의 크기와 분할 과정에서 발생하는 디스크 I/O의 횟수에 의한 비용 요소가 우선시 된다. 즉, 적은 데이타 I/O만으로 원하는 분할을 나누는 것이 목표가 된다.

본 논문에서는 기존의 분할 방법에서 분할 연산을 중심으로 한, 즉 디스크 I/O 비용 입장에서 성능을 향상시키는 방법을 다루고있다. 이를 위해 애트리뷰트 값의 특정 구간에 따라 큐브를 분할하고, 이 분할에 의한 중간 결과 Q를 이용하며, 다음 단계 분할에서의 연산 작업을 효율적으로 처리하고자 한다. 큐브 계산의 나머지 한 과정인 메모리 연산 과정에 대해서는 기존의 분할 방법인 상향식 혹은 하향식 큐보이드 계산 방법을 사용하도록 한다.

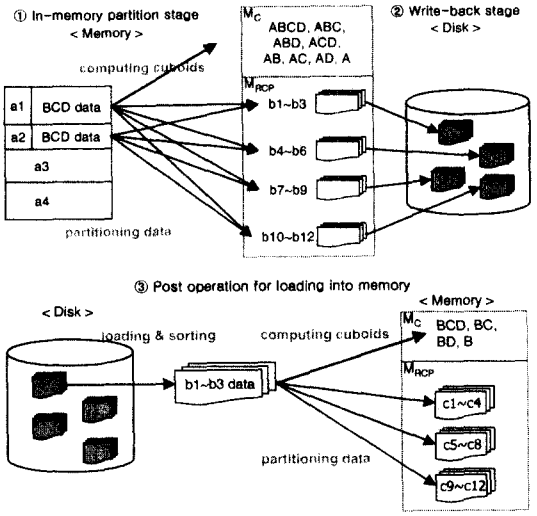


그림 4 구간 기반 분할 방법

본 논문에서 제안하는 방법은 큐보이드 계산에 필요한 메모리가 시스템에서 지원하는 메모리 영역의 크기에 비해 작은 경우, 다음 단계 분할 작업을 위해 별도의 메모리 영역을 설정하여 다음 분할 단계에서 필요한 작업의 결과인 '중간 결과' Q_M을 생성한다. 이 중간 결과는 현재 분할 애트리뷰트인 A의 애트리뷰트 값들에 대하여 분할한 내용들이 M_C 상에 로딩되어 있을 때 계산되어 M_{RCP}에 기록되는 것으로, 추가의 디스크 입출력 비용이 필요하지 않다. (즉, 현 단계 분할 애트리뷰트인 A에 의한 분할들 ABCD, ABC, ..., A는 큐보이드 계산 영역인 M_C에 저장되고, 여기에서 다음 단계 계산을 위한 애트리뷰트 B에 대한 구간 분할들인 b1~b3, b4~b6, .. 등은 메모리 영역 M_{RCP}에 기록된다.) 이것이 첫 단계로서, 읽혀진 데이타로부터 메모리 내에 중간 결과를 형성하는 과정이 된다(그림 4의 1). 각 분할에 대해 '메모리 내에서 구한 중간 결과' Q_M으로부터 '디스크에 저장되는 중간 결과' Q_D를 생성하고, 모든 애트리뷰트 값에 따른 분할에 대해 작업이 수행되면, 이들이 결합되어 전체 중간 결과 Q를 형성하게 된다. 이 중간 결과 Q가

다음 단계의 분할 연산의 결과가 된다. 이것이 두 번째 단계로서 중간 결과를 메모리에서 형성하면서 설정된 메모리 크기를 초과하는 경우 디스크 저장 형태로 전환하는 과정이다(그림 4의 2). 마지막으로, 이전 단계에서 저장되었던 중간 결과 Q의 각 구간 분할들을 메모리에 올려 관련 큐보이드들을 계산하면 다음 단계의 큐보이드 계산을 수행하는 과정이 된다(그림 4의 3).

제안하는 방법에서 첫번째 단계(애트리뷰트 A에 의한 분할 단계)에서는 구간을 기준으로 분할하지 않고, 애트리뷰트 값을 기준으로 분할한다. 차원 애트리뷰트가 A, B, C, D이며 먼저 애트리뷰트 A에 대해 데이타를 분할하여 관련된 큐보이드들을 계산한다고 하자. 그림 4에서처럼 a1을 포함하는 하나의 분할을 읽어 각 큐보이드들 ABCD, ABC, ABD,..., A 등을 계산하면서, 다음 분할 단계를 위해 설정한 메모리 영역 M_{RCP}를 사용하여 다음 단계에서 사용될 수 있도록 데이타를 분할한다. 다음 단계에서 쓰일 데이타가 애트리뷰트 B에 대한 것이므로 B의 값들을 기준으로 다음 계산에서 한번에 메모리에 올릴 수 있는 크기로 구간별 분할을 하며, 이를 위해 할당받은 메모리를 구간별 분할의 개수로 나누어 각 분할들을 유지한다. 할당된 메모리가 꽉 차게 되면 디스크에 저장하여 메모리를 비운 뒤 계속하여 데이타를 구간별 분할한다. 이것이 두 번째 과정으로서 디스크에 저장하는 (write back) 단계이다. RCP 방법의 마지막 과정은 이전 단계에서 저장한 중간 결과들을 메모리로 읽어들여 다음 단계의 큐보이드를 계산하기 위한 준비 작업이다. 구간에 따른 분할 내의 데이타들은 관련된 데이타들의 정보만을 가지고 있을 뿐, 정렬되어 있지 않기 때문에 메모리로 읽으면서 메모리 내에서 정렬해야 한다. 즉, 그림 4에서처럼 애트리뷰트 B에 관한 큐보이드들을 계산하기 위해서는 데이타를 다시 읽는 것이 아니라 이전 단계에서 디스크에 저장해 놓았던 구간별 분할 하나를 읽어 정렬을 하게 되며 마찬가지로 BCD, BC, BD, B의 큐보이드를 계산하면서 동시에 다음 계산을 위해 애트리뷰트 C에 대해 구간별 분할을 수행한다.

이와 같은 세 과정은 매 단계에 반복된다. 즉, 이전 단계에서 읽혀진 데이타를 통해 미리 분할 작업을 수행하여 중간 결과를 생성하고, 이 결과를 통해 다음 단계의 큐보이드 계산을 수행한다. 기존의 단일 값 기반 분할(PCP) 방법에서 매 단계에 이루어진 독립적인 분할 연산은 제안하는 방법에서 중간 결과 생성 과정으로 대체된다.

RCP 방법의 알고리즘은 그림 5와 같다. 알고리즘은 애트리뷰트의 분할 순서에 따라, 첫 번째 애트리뷰트에 대해서 데이타를 분할한다 (PartitionRawData). 분할하는 매 단계에서는 3.4절에서 설명하는 비용 요소에 따라

```
Procedure MakeRangePartition
Input : 하나의 구간 분할
Output : 다음 단계에 사용될 구간별 분할 집합 Q
(1)  for each tuple {
(2)      InsertTupleS_RCP_MEM();   /* 해당 S_RCP_MEM에 튜플을 넣어 준다 */
(3)      if (S_RCP_MEM == FULL) then
(4)          Write_Back();
(5)  }

Procedure RCP
Input : 테이블 R = {(a1, a2, a3, ..., an, M) | ai ∈Card(Ai), 1≤i ≤n}
Output : 큐브 C
StageCount : 현재 수행 단계를 나타내는 변수
N_RCP : 구간 분할의 개수를 나타내는 변수
(1) PartitionRawData();
(2) for (StageCount = 1 to n) {
(3)     SetParameter();
(4)     for ( i = 1 to N_RCP) {
(5)         ReadRangePartition();
(6)         Compute_Cuboids();
(7)         if (StageCount ≠n) then  /* 마지막 단계는 분할하지 않음 */
(8)             MakeRangePartition();
(9)     }
(10) }
(11) Write_Force();
```

그림 5 구간 기반 큐브 분할(RCP) 방법 알고리즘

필요한 파라미터를 설정한 후 미리 가공해 놓은 구간별 분할 데이타를 읽는다. ReadRangePartition에서 이전 단계의 구간별 분할을 읽고 Compute_Cuboids에서 큐보이드를 계산하며 MakeRangePartition에서 다음 단계에서 사용될 새로운 구간별 분할을 생성한다. 모든 분할을 읽어 중간 결과를 생성하고 나면, 메모리에 남아 있는 정보들을 디스크에 저장하기 위해 Write_Force 함수를 호출한다.

3.4 인수 설정 및 I/O 비용 분석

인수들의 관계와 I/O 비용

RCP 방법은 크게 메모리(M_T)와 결과를 저장하는 디스크로 구성된다. 메모리 부분은 다시 일반 큐보이드 계산을 위한 영역(M_C)과 다음 분할 단계 작업을 미리 계산하기 위하여 구간 분할을 생성하는 메모리 영역(M_{RCP})으로 나누어진다. 이 두 인자에 의해 구간 분할(RCP) 또한 두 부분으로 나누어지는데, 메모리 내에 존재하는 구간 분할 영역(RCP_MEM)과 디스크에 저장되는 구간 분할 영역(RCP_DISK)이 이에 해당한다. 이 두 구간 분할 영역은 분할의 크기(S_{RCP})와 구간 분할의 수(N_{RCP}), 그리고 각 구간의 폭(W_{RCP})으로 그 특성이 결정되며, 이에 따라 RCP 방법의 I/O 성능에 영향을 미치게 된다.

메모리와 디스크, 그리고 구간 분할의 종류에 의해 결정되는 각 인수들은 다음과 같은 관계를 갖는다. 한 애트리뷰트에 대해서 도메인의 모든 값이 같은 확률로 나타나는 균일 분포(uniform distribution)를 가정한다.

$$M_T = M_C + M_{RCP} \tag{1}$$

$$N_{RCP} = Card(\neq xtPartitionA\frac{ttribute)}{W_{RCP}} \tag{2}$$

$$S_{RCP\_MEM} = \frac{M_{RCP}}{N_{RCP}} \tag{3}$$

$$S_{RCP\_DISK} = \frac{size(Q_{\neq xt})}{N_{RCP}} \tag{4}$$

여기서 주목할 점은 S_{RCP_MEM}의 값이다. S_{RCP_MEM}는 메모리에서의 하나의 구간 분할의 크기이다. 이 구간 분할은 할당된 영역이 해당 데이타들로 채워졌을 때, 디스크에 write-back되는 양에 해당한다. 즉, RCP 방법의 각 단계에서 I/O 횟수는 S_{RCP_MEM}에 반비례한다. 만약 남은 잉여 메모리 M_{RCP}의 양이 작아서 S_{RCP_MEM}이 I/O의 단위인 1 페이지 크기보다 작게 되면, 데이타의 양은 이미 정해져 있기 때문에 디스크로 write-back하는 연산 수행시 I/O 작업의 효율성이 떨어진다. 반면에 S_{RCP_MEM}의 크기가 디스크 I/O 단위인 1 페이지가 된다면, 해당 구간 분할을 생성하기 위해 필요한 I/O의 횟수는 해당 구간 분할의 크기와 같을 것이다. 이 경우 앞서 설명한 정성적인 I/O 비용 값과 같다. 한 단계에서 발생하는 I/O 비용(I/O 횟수)은 S_{RCP_MEM}에 의해서 다음과 같이 결정된다.

표 1 기호 설명

기호	설명
M_T	전체 시스템 메모리의 양
M_C	큐보이드 계산을 위해 사용되는 일반적인 (기존의) 분할 방법에 의해 사용되는 메모리의 양
M_{RCP}	RCP방법에 의해 사용되는 메모리의 양; 즉, $M_T - M_C$
RCP_MEM	RCP 방법에서 사용하는 메모리 영역
RCP_DISK	RCP 방법에 의해 사용되는 디스크 영역
N_{RCP}	구간의 갯수
S_{RCP_MEM}	M_{RCP}에 저장되는 하나의 구간 분할이 차지하는 크기; 가령, M_{RCP}가 1024K 이고 N_{RCP}가 10 이면, 하나의 구간 분할이 메모리에서 차지하는 크기는 100K 이다
S_{RCP_DISK}	RCP_DISK에 저장되는 하나의 구간 분할의 크기
W_{RCP}	하나의 구간이 갖는 폭; 즉 domain 값의 개수. 본 본문에서는 uniform 분포를 가정하므로 하나의 대표값으로 표현할 수 있다
Card()	애트리뷰트의 cardinality
Size()	데이타의 크기
$S_{RCP\_MEM\_in\_page}$	S_{RCP_MEM}/PageSize
IO_{RCP}	RCP 방법을 사용할 때의 IO 비용
IO_{PCP}	PCP 방법을 사용할 때의 IO 비용
Q_{next}	다음 분할 데이타
$Q_{current}$	현재 분할 데이타

$$IO_{RCP} = \frac{1}{S_{RCP\_MEM\_in\_page}} \cdot size(Q_{current}) \tag{5}$$

$$S_{RCP\_MEM\_in\_page} = \frac{S_{RCP\_MEM}}{size(PAGE)})$$

여기서 $S_{RCP_MEM_in_page}$는 I/O 단위인 1 페이지에 대한 비율을 나타낸다. 이와 같은 특성으로 인해 S_{RCP_MEM}의 값은 크게 유지하는 것이 I/O 비용을 줄이는 하나의 방안이 됨을 알 수 있다(식 (5)). 위의 식 (3)으로부터 주어진 잉여 메모리 내에서 S_{RCP_MEM}을 늘리기 위해서는 구간 분할의 수 N_{RCP} 값을 줄여야 한다. 그리고 N_{RCP} 값을 줄이기 위해서는 주어진 애트리뷰트의 cardinality에 대해서 구간의 폭(W_{RCP})을 늘려야 한다(식 (2)).

위 원리에 의해서 본 논문에서는 분할을 구간으로 구분 짓는다. 주어진 잉여 메모리를 효율적으로 사용하고 디스크로 write-back되는 I/O의 수를 줄이기 위해서 한 번에 쓰여지는 데이타의 양을 늘릴 필요가 있다. 만약

중간 결과를 위한 분할을 애트리뷰트 하나의 값 단위로 구성한다면, 유지해야 하는 분할의 수는 증가하여 한정된 잉여 메모리에 대해서 각 분할이 사용할 수 있는 양은 줄어든다. 앞서 언급한 바와 같이 할당된 메모리의 양이 I/O의 단위인 1 페이지 이하라면, 잦은 write-back으로 I/O 비용을 증가 시키는 결과를 가져 온다. 따라서 각 분할에 할당되는 메모리의 양이 I/O 단위인 1 페이지에 가까운 크기가 되도록 분할을 구간 단위로 구성하는 것이 유리하다.

PCP 방법과 RCP 방법

기존의 단일 값 기반 분할(PCP) 방법은 매 분할 단계마다 독립적으로 분할 연산을 수행하므로 여기에서 발생하는 I/O 비용은 각 단계에 읽어 들이는 데이타 크기와 채택된 분할 알고리즘에 의해 정적으로 결정된다. 사용된 분할 알고리즘이 주어진 메모리에서 α pass로 주어진 데이타를 분할 형태로 만든다고 가정하면, 매 단계별 I/O 비용은 다음과 같이 구해진다.

$$IO_{PCP} = \alpha \cdot size(Q_{current}) \quad (6)$$

여기서 만약 $Q_{current}$가 해당 단계에 RCP 방법에 의해 생산된 중간 결과 Q와 같은 애트리뷰트 집합으로 구성된다고 가정하고, 그 크기도 같다고 보면,

$$IO_{PCP} = \alpha \cdot size(Q_{current}) = \alpha \cdot (S_{RCP-MEM-in-page} \cdot IO_{RCP})$$

가 된다. 즉,

$$\frac{IO_{RCP}}{IO_{PCP}} = \frac{1}{\alpha \bullet S_{RCP\_MEM\_in\_page}}$$

가 되며,

$$S_{RCP\_MEM\_in\_page} > 1/\alpha$$

인 경우, 본 논문에서 제안한 RCP 방법이 기존의 독립적 분할 방법에 비해 좋은 성능을 가지게 된다. 따라서, 위의 식에 따라 S_{RCP_MEM}을 정하게 되면 RCP 방법이 좋은 성능을 가지며, 본 논문에서는 이 값을 기존의 독립적 분할 방법과의 trade-off 점으로 사용한다. 예를 들어 독립적 분할 방법으로 외부 정렬 방법을 사용하고, 주어진 메모리가 외부 정렬 과정을 4 pass만에 처리할 수 있는 충분한 크기라고 가정한다. 이 경우 식 (6)에서 α는 4가 되며,다음 조건에서 더 나은 성능을 갖게 된다.

$$S_{RCP\_MEM\_in\_page} > 1/4 = 0.25 = \sigma \quad (7)$$

이와 같은 값은 RCP 방법이 각 단계별로 적용되는 과정에서 메모리의 제한에 의해 발생할 수 있는 I/O 비용의 과다를 막기 위해 사용될 수 있다. 각 단계에서 중간 결과를 생성하여 다음 분할 연산을 대체하는 과정에서 $S_{RCP_MEM_in_page}$를 예측하여, 이 값이 (σ=0.25) 이하인 경우에는 기존의 독립적 분할 방법을 사용하도록 조정한다. 이러한 기존 방법과의 trade-off 점을 이용하여 RCP 방법은 기존 방법보다 나은 성능을 갖도록 큐브 계산 알고리즘을 구성할 수 있다.

4. 실험 및 분석

이 장에서는 본 논문에서 제안한 RCP 방법이 큐브 계산에 적용될 때에 I/O의 성능을 높일 수 있음을 실험을 통해 보이도록 한다. RCP 방법은 기존의 PCP 방법의 I/O 효율성을 높이기 위한 방법으로 메모리 내에서의 큐브 계산을 위한 방법은 기존의 연구를 활용하는 것을 가정하였다. 따라서 본 실험에서는 큐브 계산 시 RCP를 사용한 경우와 기존의 PCP를 이용한 경우의 I/O 성능을 비교 분석한다.

4.1 실험 환경

본 실험은 Intel Platform에서 운영되는 Linux OS에서 수행하였다. 사용된 시스템의 사양은 Intel Pentium-II 300MHz CPU, 128MB의 메인 메모리, 그리고 2GB의 사용 가능한 디스크로 이루어졌다.

본 논문에서 제안하는 RCP 알고리즘은 PCP 방법의 I/O 성능을 개선하기 위한 방법으로 큐브 연산자를 계산하기 위한 I/O 비용, 특히 계산 시 필요한 데이타에 대해 발생하는 I/O양에 초점을 두었다. 이와 같은 관점의 성능 분석은 시스템의 사양에 독립적이다. 큐브 계산을 위한 CPU 비용에 대한 분석은 사용되는 시스템 환경에 의존적일 뿐만 아니라 I/O 비용의 개선이 실제 CPU 비용에서도 개선 효과를 갖기 때문에, 본 실험에서는 I/O 비용의 개선 정도로 CPU 비용에서의 개선 정도를 대체하는 것으로 한다. 그리고 I/O 비용은 1024B 크기의 페이지 단위로 I/O에 대한 횟수를 계산하였다.

실험에서의 비교 대상은 기존의 PCP 방법에서 채택한 매 단계별 독립적 분할 방법이다. 독립적 분할 방법에서 발생하는 I/O 횟수는 주어진 메모리의 모든 영역이 해쉬 테이블로 사용되는 것을 가정해 결정하도록 한다. 즉, 매 단계에 입력 데이타의 cardinality와 데이타 크기에 대해서 하나의 해쉬 버켓을 위해 한 페이지 크기를 할당할 수 있는 경우, 발생하는 I/O 횟수는 2 회로 가정하며 이 외의 경우에는 4 회로 가정한다.

실험 데이타

본 실험에서 사용된 데이타의 집합은 표 2와 같다. 차원 애트리뷰트의 수는 5차원부터 8차원까지이며 각각의 cardinality는 높은 cardinality(2^9)부터 낮은 cardinality (2^2)에 이르는 다양한 데이타 환경에 따른 영향을 반영할 수 있도록 2^9, 2^8, 2^7, 2^6, ..., 2^2와 같이 2의 지수 승 형태로 설정하였다. 아울러 cardinality간의 차이로부터

표 2 실험 데이타의 값

차원	Cardinality	데이타 밀도	크기	최소 분할 크기
5	2^6: 2^8: 2^7: 2^6: 2^5	10^{-6} 10^{-5} 10^{-4}	768KB 7.9MB 76.8MB	1.5KB 15KB 150KB
6	2^6: 2^8: 2^7: 2^6: 2^5: 2^1 2^6: 2^8: 2^7: 2^6: 2^5: 2^1 100:100:100:100:100:100	10^{-6} 10^{-5} 10^{-6}	14.3MB 143.MB 28M	28KB 280KB 280KB
7	2^6: 2^8: 2^7: 2^6: 2^5: 2^4: 2^3	10^{-6}	131.1MB	256KB
8	2^6: 2^8: 2^7: 2^6: 2^5: 2^4: 2^3: 2^2	10^{-6}	576MB	1.125MB

발생할 수 있는 개선 효과를 최소화하고 RCP 방법만의 개선 정도를 비교할 수 있도록 균일 cardinality 100을 갖는 6차원 데이타를 구성하였다. 그리고 데이타 밀도에 따른 영향을 살펴 보기 위하여 각각의 데이타 집합에 대해서 10^{-6}~10^{-4}의 데이타 밀도를 갖도록 전체 데이타 환경을 구성하였다. 마지막으로 메모리 크기에 따른 RCP 방법의 성능을 평가하기 위해 50KB~10MB 범위에서 메모리를 변화하며 실험하였다.

위의 표 2에서 최소 분할 크기 항목은 최대의 cardinality를 갖는 애트리뷰트에 의해 분할된 한 분할의 크기를 나타낸다. 즉, 큐브 분할 방법에서 사용될 분할의 최소 크기로서 주어지는 메모리의 크기는 이 값보다 커야 의미가 있다.

4.2 결과 및 분석

메모리 증가에 따른 성능 평가

먼저, 첫 번째 실험으로 메모리 증가에 따른 성능을 평가해 보았다. 그림 6(a),(b),(c) 그리고 (d) 5차원부터 8차원까지의 데이타에 대해 데이타 밀도를 10^{-6}으로 고정시키고 메모리 변화에 대한 성능의 변화를 실험한 결과이다. 그림에서 가로축은 메모리의 변화량을 나타내고 세로축은 큐브 전체 계산 중에 소요되는 I/O의 양이 입력 데이타 R의 몇 배인지를 나타낸다.

5차원 데이타의 경우 50KB이상의 메모리가 주어졌을 때 RCP 방법이 적용되어 전체 성능에서 향상을 꾀할 수 있다. 1M의 메모리가 주어지면 PCP 방법이나 RCP 방법 모두 입력 데이타 전체를 메모리에 올릴 수 있기 때문에 I/O의 비용이 발생하지 않거나 급격히 줄어들게 된다. 6차원 데이타의 경우 한 분할의 크기가 더 크기 때문에 100KB이상의 메모리에서 성능이 개선됨을 볼 수 있다. 8차원 데이타는 표 2에 표시한 바와 같이 2MB이상에서만 실험을 수행하였고, I/O 비용이 줄어드는 것을 볼 수 있다.

각 데이타는 작은 메모리(5차원의 50KB나 6차원의 100KB 등)에 대한 성능 개선 효과가 큰 메모리의 경우에 비해 상대적으로 적게 나타나는 것을 알 수 있다. 이것은 성능의 향상을 얻을 수 있을 단계까지만 RCP 방법을 적용하기 때문에 발생하는 현상으로 전체 비용은 기존의 독립 분할 방법보다 작음을 알 수 있다. 그러나 전체적으로 큐브 계산의 초기 단계에 발생하는 많은 양의 I/O를 RCP 방법으로 대체함으로써 기존의 분할-기반 방법에 비해 좋은 성능을 가질 수 있다.

애트리뷰트 수의 증가에 따른 성능 평가

두 번째 실험 결과는 차원의 증가에 따른 성능 평가이다. 이 실험에서는 각 차원 애트리뷰트의 개수에 따른 데이타의 크기와 메모리의 크기에 대한 비율을 대략 50:1정도로 하고 데이타 밀도를 10^{-6}으로 고정하였다.

그림 6(e)에서와 같이 RCP방법은 주어진 데이타의 크기에 대해서 일정한 크기의 메모리가 지원되는 경우 차원의 증가에 큰 영향을 받지 않음을 알 수 있다. 실험 결과를 좀 더 자세히 살펴 보면, 대략 입력 데이타의 4.3 ~ 4.8배에 해당하는 I/O가 발생하면서 전체 큐브를 계산할 수 있으며 이것은 기존의 PCP 방법에 비해 매우 좋은 성능임을 알 수 있다.

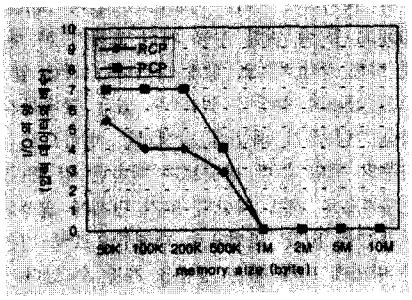
데이타 밀도에 따른 성능 평가

세 번째 실험은 5차원 데이타 집합에 대해 데이타 밀도를 변화시키면서 그 성능을 평가하였다. 그림 6(a)는 밀도가 매우 낮은 데이타에 대한 결과로서 대략 입력 데이타의 4.2배에 해당하는 양만큼 I/O가 발생하는 것을 볼 수 있다. 500KB이상의 경우에는 앞에서 설명한 바와 같이 I/O 비용이 줄어든다. 그림 6(f)는 그림 6(a)에 비해 상대적으로 매우 응집된 데이타에 대한 결과로서 500KB이상의 메모리에서 마찬가지로 고른 성능 효과를 볼 수 있다. 이 결과로부터 본 논문이 제안하는 RCP 방법이 다양한 데이타 밀도에 유연하게 사용될 수 있음을 알 수 있다.

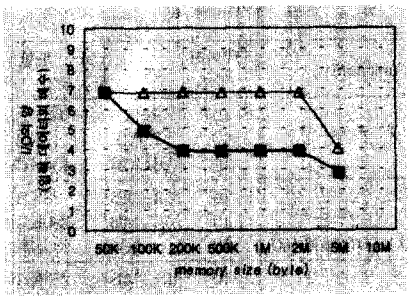
이와 같은 실험으로부터 RCP 방법의 성능에 영향을 주는 요인은 입력 데이타의 크기와 사용 가능한 메모리의 양임을 알 수 있다. 즉, 희박성의 정도에 따라 입력 데이타의 크기가 변화하게 되고 그 크기에 맞는 일정량의 메모리가 주어지는 경우 RCP 방법은 일정 수준의 성능 개선 효과를 얻을 수 있음을 알 수 있다.

단일 cardinality 분포를 갖는 데이타에 따른 성능 평가

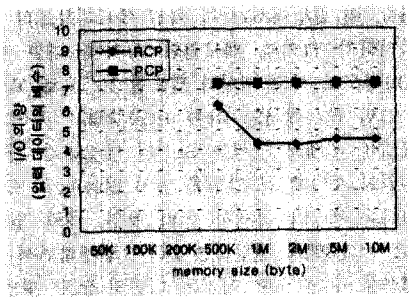
마지막으로 단일 cardinality 분포를 갖는 6차원 데이타를 가지고 그 특성이 어떠한 영향을 끼치는지 실험하였다. 각 차원의 cardinality는 100이며 데이타 밀도는 10^{-6}이다. 그림 6(h)의 결과에서 보듯이 균일 cardinality



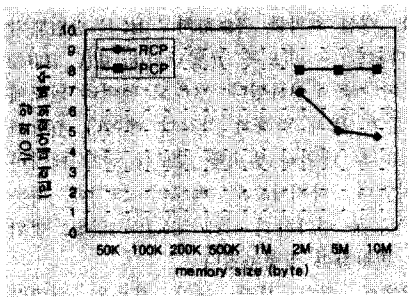
(a) 데이타 밀도 10^{-6}의 5차원 데이타



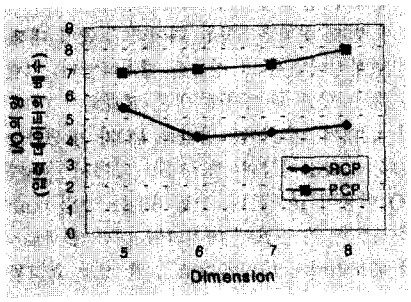
(b) 데이타 밀도 10^{-6}의 6차원 데이타



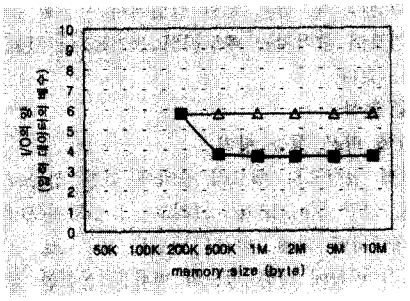
(c) 데이타 밀도 10^{-6}의 7차원 데이타



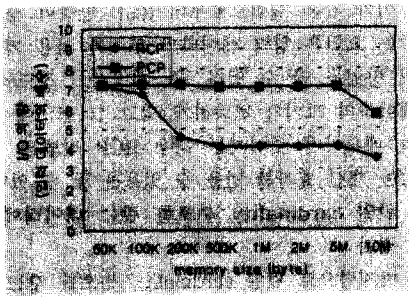
(d) 데이타 밀도 10^{-6}의 8차원 데이타



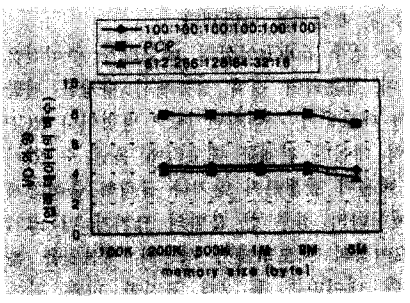
(e) 차원 증가에 다른 성능 변화



(f) 데이타 밀도 10^{-4}의 5차원 데이타



(g) 데이타 밀도 10^{-5}의 5차원 데이타



(h) 균일 *cardinality* 데이타의 성능 비교

그림 6 다양한 차원과 데이타 밀도에 대한 RCP의 성능 평가 실험

데이타는 다양한 cardinality 데이타와 성능에 있어 큰 차이가 없다. 그림 6(e)에서와 같이 RCP 방법이 보장하는 성능 값인 4.3~4.8 사이의 값을 가짐을 알 수 있다. 이로써 주어진 RCP 방법은 균일 cardinality 분포에 대해서도 좋은 성능을 보임을 알 수 있다.

5. 결 론

본 논문에서는 기존의 큐브 분할 방법이 애트리뷰트의 단일 값을 기준으로 분할을 시행하는 과정에서 연속하는 단계들 사이의 계산 중복성을 고려하지 못한 문제점을 지적하고, 이를 해결하여 I/O 효율성을 향상시킬 수 있는 방법으로, '구간 기반 큐브 분할 방법'을 제안하였다. 제안하는 방법은 이전 단계에서 읽은 데이타를 사용하여 큐보이드를 계산하는 과정에서, 다음 단계의 작업을 효과적으로 줄이기 위하여, 애트리뷰트 값의 구간에 기준하여 미리 분할을 시행하는 방법이다.

본 논문에서 제안한 구간 기준 분할 기법은 기존의 단일 값 기준 분할 방법에서 이루어지던 독립적 분할 방법을 대체하여 전체적으로 I/O의 양을 줄일 수 있음을 실험을 통해 비교 분석하였다. 결과적으로 하나의 분할만 올라올 수 있는 크기의 메모리가 주어진다면 I/O 비용에서 기존의 방법에 비해 30%정도의 이득이 있음을 알 수 있었다. 또한, 차원의 크기가 증가하거나 데이타의 밀도가 낮아지더라도 큰 영향 없이 좋은 성능을 나타냄을 보였다.

참 고 문 헌

[1] S.Agarwal, R.Agrawal, et al. "On the Computation of Multidimensional Aggregates," in Proceedings of the 22nd VLDB, pp.506-521, 1996.

[2] K.A.Ross and D.Srivastava. "Fast Computation of Sparse Datacubes". In Proceedings of 23rd VLDB, pp.116-125. 1997

[3] K.Beyer and R.Ramakrishnan. "Bottom-Up Computation of Sparse and Iceberg CUBEs," In Proceedings of the ACM SIGMOD, pp.359-369, 1999.

[4] S.Chaudhuri and U.Dayal. "An Overview of Data Warehousing and OLAP technology," In ACM SIGMOD Record, 1996

[5] E.F.Codd, S.B.Codd, and C.T.Salley. "Providing OLAP(On-Line Analytical Processing) to User-Analysts : An It Mandate," 1993.

[6] P.M.Deshpande, SmAgarwal, et al. "Computation of Multidimensional Aggregates," Technical Report-1314, Univ. of Wisconsinn-Madison, 1996.

[7] V.Harinarayan, A.Rajaraman, and J.D.Ullman. "Implementing data cubes efficiently," In Proceedings of the ACM SIGMOD Conference, pp.205-227, 1996.

[8] J.Gray, A.Bosworth, A.Laymon, and H.Pirahesh. "Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-tabs and Sub- Totals," In Proceedings of the 12th ICDE, pp.12 -159, 1996.

[9] S.Sarawagi, R.Agarwal, and A.Gupta. "Modeling multidimensional databases," In Proceedings of the 13th ICDE, pp.232-243, 1997

[10] S.Sarawagi, R.Agrawal, and A.Gupta, "On Computing The Data Cube," Research Report RJ 10026, IBM Almaden Research Center, San Jose, California, 1996. Available from http://www.almaden.ibm.com/cs/quest

[11] A.Shukla, P.M.Deshpande, J.F.Naughton, and K.Ramasamy. "Storage estimation for multidimensional aggregates in the presence of hierarchies," In Proceedings of the 22nd VLDB Conference, pp.522-531, 1996.

[12] Y.Zhao, P.M.Deshpande, and J.F.Naughton. "An ArrayBased Algorithm for Simultaneous Multidimensional Aggregates," In Proceedings of the ACM SIGMOD Conference, pp.159-170, 1997.

박 웅 제

1989년 ~ 1993년 서강대학교 전자계산학과 학사. 1993년 ~ 1995년 한국과학기술원 전산학과 석사. 1995년 ~ 현재 한국과학기술원 전산학전공 박사과정. 1995년 ~ 현재 (주) 다우데이타시스템 연구소 책임 연구원. 관심분야는 OLAP, Data Warehouse, Distributed Computing, KMS

정 연 돈

정보과학회논문지 : 데이타베이스
제 28 권 제 1 호 참조

김 진 녕

1994년 ~ 1998년 서강대학교 전자계산학과 학사. 1998년 ~ 2000년 한국과학기술원 전산학과 석사. 2000년 ~ 현재 (주)데이콤 연구원. 관심분야는 OLAP, Data Warehouse

이 윤 준

정보과학회논문지 : 데이타베이스
제 28 권 제 2 호 참조

김 명 호

정보과학회논문지 : 데이타베이스
제 28 권 제 1 호 참조