효율적인 ROLAP 큐브 생성 방법
(An Efficient ROLAP Cube Generation Scheme)

김 명 † 송 지 숙 ††
(Myung Kim) (Ji-sook Song)

요 약 ROLAP(Relational Online Analytical Processing)은 다차원적 데이타 분석을 위한 제반 기술로써, 전사적 데이타 웨어하우스로부터 고부가가치를 창출하는데 필수적인 기술이다. 질의처리 성능을 높이기 위해서 대부분의 ROLAP 시스템들은 집계 테이블들을 미리 계산해 둔다. 이를 큐브 생성이라고 하며, 이 과정에서 기존의 방법들은 데이타를 여러 차례 정렬해야 하고 이는 큐브 생성의 성능을 저하시키는 큰 요인이다. [1]은 MOLAP 큐브 생성 알고리즘을 통해 간접적으로 ROLAP 큐브를 생성하는 것이 훨씬 빠르다는 것을 보였다. 본 연구에서도 MOLAP 큐브 생성 알고리즘을 사용한 신속하고 확장적인 ROLAP 큐브 생성 알고리즘을 제시하였다. 분석할 입력 사실 테이블을 적절하게 조각내어 메모리 효율을 높였고, 집계 테이블들을 최소 부모 집계 테이블로부터 생성하도록 하여 큐브 생성 시간을 단축하였다. 제안한 방법의 효율성은 실험을 통해 검증하였다.

키워드 : 온라인 데이타 분석, 큐브 생성

Abstract ROLAP(Relational Online Analytical Processing) is a process and methodology for a multidimensional data analysis that is essential to extract desired data and to derive value-added information from an enterprise data warehouse. In order to speed up query processing, most ROLAP systems pre-compute summary tables. This process is called 'cube generation' and it mostly involves intensive table sorting stages. [1] showed that it is much faster to generate ROLAP summary tables indirectly using a MOLAP(multidimensional OLAP) cube generation algorithm. In this paper, we present such an indirect ROLAP cube generation algorithm that is fast and scalable. High memory utilization is achieved by slicing the input fact table along one or more dimensions before generating summary tables. High speed is achieved by producing summary tables from their smallest parents. We showed the efficiency of our algorithm through experiments.

Key words : OLAP, cube generation

1. 서 론

OLAP(Online Analytical Processing)은 데이타 웨어하우스에 저장된 데이타를 다차원적으로 분석하여 그 결과를 온라인으로 사용자에게 제시하는 제반 기술이다 [2, 3, 4]. 이는 데이타 마이닝, XML/HTML 문서처리 기술과 함께 고객단위 개별서비스를 지원하는 전자상거래 시스템 구축의 핵심 기술 중의 하나이다.

대용량 데이타를 분석하여 그 결과를 온라인으로 제공해야 하는 제약으로 인해 대부분의 OLAP 시스템들은 분석결과의 일부를 미리 계산하여 저장한다. 이를 '집계 테이블 사전 연산 (summary table pre-calculation)' 또는 '큐브 생성(cube generation)'이라 하는데, 이 작업은 상당한 시간을 요할 뿐 아니라, 이 때 생성되는 데이타는 그 양이 방대하여 신속한 질의처리를 위해 최적화 되어 저장되어야 한다.

OLAP 시스템들 중에서 큐브를 릴레이션 테이블에 저장하는 시스템을 ROLAP 시스템이라고 한다. 이러한 시스템들로는 MicroStrategy사의 OLAP Provider[5], Informix사의 Metacube[6], Information Advantage [7] 등이 있다. 큐브 생성 과정에서 분석 대상인 사실테이블(fact table)과 도중에 생성되는 집계테이블들이 여러 번 스캔되면서 집계 계산(aggregation)이 이루어져야 하며, 이 과정의 효율성을 높이기 위해서 서로 연관 있는

· 본 연구는 한국과학재단 목적기초연구(R04-2001-00191) 지원으로 수행되었음.
† 종신회원 : 이화여자대학교 컴퓨터학과 교수
mkim@ewha.ac.kr
†† 비 회 원 : 이화여자대학교 컴퓨터학과
sunfish@ewha.ac.kr
논문접수 : 2001년 3월 27일
심사완료 : 2002년 1월 4일

데이타가 모여 있어야 하고, 이 때 정렬이나 해싱이 사용된다[8]. 특히 기업에서 사용하는 비즈니스 데이타들은 그 양이 방대하여 큐브 생성 과정에서 정렬할 테이블 전체가 메모리에 로드될 수 없는 경우가 흔한데, 이 때 외부 정렬이 그러한 테이블마다 여러 회 실행되어야 하며, 이 과정이 큐브 생성의 효율성을 떨어뜨리는 큰 요인이다.

MOLAP(Multidimensional OLAP) 기술을 이용하면 ROLAP 큐브 생성의 효율성을 크게 높일 수 있다는 연구결과가 [1]에 제시되어 있다. MOLAP 시스템은 큐브를 다차원 배열에 저장하는 방식을 쓴다. 이러한 시스템들로는 하이페리온사의 Essbase[9], 마이크로소프트사의 SQL Server 2000[10], 오라클사의 Express Server [11]을 들 수 있다. 큐브내의 데이타는 배열 위치정보를 통해 엑세스되므로, 유효(valid) 셀이 많을 때는 큐브가 상당히 효율적인 저장방식이나 그렇지 않을 때는 적절한 데이타 압축 기술이 필요하다. [1]은 저밀도 큐브의 데이타 압축 방법을 제시하였고, 사용자 질의처리 시에 디스크 블록 읽는 회수를 줄이기 위해 [12]가 제안한 청크 단위 배열 저장방식을 기반으로 한 큐브 생성 알고리즘을 제안하였다. [1]에서는 이 알고리즘을 ROLAP 집계 테이블의 생성에 적용하는 방안을 제시하였고, 그 방안이 테이블 상에서 ROLAP 집계 테이블들을 직접 생성하는 것보다 훨씬 효율적이라는 것을 실험적으로 보였다.

본 연구에서는 MOLAP 큐브 생성 알고리즘을 사용한 효율적인 ROLAP 큐브 생성 방법을 제시한다. 멤버의 수가 가장 큰 차원을 기준으로 사실테이블을 조각내어 놓으면, 사실테이블을 1번만 스캔하면서 최소의 집계 연산만으로 집계 테이블들을 생성할 수 있고, 이 과정에서 사용되는 메모리는 가장 작은 1차 집계 테이블 1개와 이로부터 생성 가능한 2차 이상의 집계 테이블들의 저장공간 뿐이라는 점에 착안하여 시간/공간적 성능을 향상시킨다. 제안한 방법의 효율성은 기존의 ROLAP 큐브 생성 알고리즘들보다 월등하게 속도가 빠른 [1]의 방법과의 비교를 통해 보였다. 논문은 다음과 같이 구성된다: 2절에서 OLAP 큐브 생성과 기존의 연구 결과를 설명하고, 3절에서 새로운 ROLAP 큐브 생성 방법을 제시한다. 4절에서 이 방법들의 효율성을 입증하고, 5절에서 결론을 맺는다.

2. OLAP 큐브 생성과 기존의 연구 결과

여기서 기존의 ROLAP/MOLAP 시스템들의 큐브 생성 방식들을 간략하게 소개한다. 우선 ROLAP 집계 테이블 계산에 관해 살펴보기로 하자. 예를 들어, 다음과 같은 스키마로 정의된 데이타를 다차원적으로 분석하려 한다고 하자. Product-schema = (p-key, p-name, p-category), Store-schema = (s-key, s-name, s-city, s-country), Time-schema = (t-key, t-day, t-month, t-year), Sales-schema = (p-key, s-key, t-key, sales). Sales는 분석 대상 데이타인 사실테이블이고, Product, Store, Time은 이 데이타를 설명하는 차원테이블(dimension table)들이다. 예를 들어, (p1, apple, fruit), (s3, LG-Mart, Seoul, Korea), (t2, 2000-10-25, 10, 2000)이 각각 순서대로 Product, Store, Time 차원 테이블에 저장된 튜플이라면 Sales 테이블의 (p1, s3, t2, 150) 튜플은 "LG-Mart에서 2000년10월25일에 150만원 어치의 사과를 판매했다"는 것을 나타낸다. 즉, Sales 테이블의 판매액(sales)은 상품, 상점, 기간에 의해 정해진 3차원 데이타이다.

Sales-schema 스키마로 정의된 사실테이블을 각 차원의 첫 글자를 따서 *PST*(Product, Store, Time) 테이블이라고 하자. 표 1에 *PST*의 예제가 있다. 이 테이블로부터 판매시기에 관계없이 모든 상품들의 상점별 판매총액을 계산한 것이 *PS* 테이블이다. 또한 각 상품이 상점과 기간에 무관하게 판매한 총액은 *P* 테이블에 계산되어 있다. *PST*는 분석할 입력 데이타이고, 이로부터 한 차원씩 집계 계산을 통해 생성된 *PS*, *P*와 같은 테이블들을 집계 테이블이라고 한다. OLAP의 큐브 생성 단계는 *PS*, *PT*, *ST*, *P*, *S*, *T*, *all* 을 생성하는 것이다. 즉, 분석할 데이타가 n 차원인 경우 2^n-1 개의 집계 테이블이 생성된다. ROLAP 시스템들은 이러한 집계 테이블들을 테이블 형태로 저장하고, MOLAP 시스템들은 이들을 배열 형태로 저장한다.

표 1 사실테이블과 집계 테이블들

PST

p-key	s-key	t-key	sales
1	1	1	2
1	1	2	3
1	1	3	1
1	2	1	3
1	2	2	4
1	2	3	2
1	3	2	5
1	3	4	4
1	4	2	3
2	1	2	2
2	1	3	4
2	2	1	3
...	...	...	...

PS

p-key	s-key	sales
1	1	6
1	2	9
1	3	9
1	4	3
2	1	6
2	2	3
...	...	...

P

p-key	sales
1	27
2	28
3	24
4	32

ROLAP 큐브 생성의 성능을 향상시키기 위한 여러 가지 기법들이 제안되어 있다. 대표적인 기법들로는 PipeSort, PipeHash, Overlap을 들 수 있다. PipeSort 알고리즘[8]은 정렬 기반의 파이프라인 방식 알고리즘이다. 우선 모든 집계 테이블에 대하여 그 테이블로부터

1개의 차원만을 제거하면서 생성할 수 있는 모든 집계 테이블들을 찾아 이들을 노드로 하는 큐브 래티스(cube lattice)를 만든다. 큐브 래티스로부터 정렬 비용을 최소화시키는 큐브 트리를 생성한다. 그 다음에 파이프라인 방식으로 집계 테이블들이 계산될 수 있도록 큐브 트리를 몇 개의 패스(path)로 분해하고, 각 패스를 따라 가면서 집계 테이블을 파이프라인 방식으로 계산한다. PipeSort 알고리즘의 단점은 미리 추정된 정렬 비용을 사용하여 큐브 트리를 결정한다는 것이다. 집계 테이블의 정렬 비용은 아직 그 테이블이 생성되지 않은 상태에서 추정하는 것이고, 정렬할 때 그 테이블이 메모리에 있는지 디스크에 있는지가 고려되지 않기 때문에 실제 이 알고리즘을 적용하는 경우 예측했던 비용보다 훨씬 더 큰 비용을 유발할 가능성이 높다.

PipeHash 알고리즘[8]은 PipeSort 알고리즘을 고안한 연구팀에서 개발한 것으로 PipeSort 알고리즘과 마찬가지로 큐브 트리를 결정한 후에 집계 테이블들을 생성한다. 큐브 트리의 각 노드(집계 테이블에 해당)는 크기가 가장 작은 부모 노드로부터 계산되도록 하는 MST(minimum spanning tree)를 큐브 트리로 결정한다. 하나의 집계 테이블로부터 자식 집계 테이블을 계산하는 과정에서는 자식 집계 테이블을 위해 해쉬 테이블을 사용한다. 부모집계 테이블이 정렬되어 있지 않기 때문이다. 해쉬 테이블이 메모리에 들어올 수 없을 정도로 큰 경우에는 특정 차원으로 부모 집계 테이블을 조각내어 디스크에 저장해 두고, 집계 테이블의 계산은 각 조각에 대해 독립적으로 계산한다. PipeHash 알고리즘은 대체로 PipeSort 알고리즘보다 성능이 좋지 못하다.

Overlap 알고리즘[8]은 PipeSort 알고리즘과 같이 정렬기반의 파이프라인 방식 알고리즘이다. PipeSort 알고리즘과는 달리 Overlap 알고리즘은 정렬할 테이블의 기존 정렬 상태를 최대한 고려하여 정렬의 비용을 줄이려고 한다. Overlap 알고리즘은 모든 차원에 순서를 정해 놓고 이를 알고리즘 전 과정에서 사용한다. 여러 차원으로 구성된 집계 테이블의 이름도 이 순서를 따른다. 큐브 래티스로부터 큐브 트리를 생성할 때는 모든 노드의 부모 노드를 찾아야 하는데, 이 때 부모 노드는 자식 노드와 이름의 접두부(prefix)가 가장 긴 것으로 선택된다. 큐브 트리가 결정된 후, 트리 각 노드를 계산하는데 필요한 메모리 양을 추정한다. 메모리 안에서 동시에 계산될 수 있는 노드들의 집합이 선택되고, 그 집합에 속한 노드들한테 추정한 메모리 양이 할당되어 집계 테이블들은 파이프라인 방식으로 계산된다. 이 집합에 속하지 않은 노드들한테는 디스크의 한 페이지에 해당하는 메모리 버퍼가 할당되고 해당 집계 테이블을 위한 중간 계산 결과가 버퍼를 통해 디스크로 출력된다. 이는 나중에 필요에 따라 합병되어 결과 튜플로 만들어진다. Overlap 알고리즘 역시 분석 대상 사실테이블이 희박한 경우, 부가적 디스크 I/O를 유발하는 집계 테이블 수가 사실 테이블 차원 수의 제곱이 되어 효율이 떨어지는 단점이 있다[13].

MOLAP 큐브 생성 알고리즘을 사용하여 ROLAP 큐브 생성 효율을 높이는 방안이 [1]에 제시되어 있다. 이를 ZDN 알고리즘이라고 부르기로 한다. 이 알고리즘에서는 ROLAP 사실 테이블을 배열로 변환 한 후에 MOLAP 큐브 생성기를 사용하여 큐브를 생성하고 이를 다시 ROLAP 집계 테이블 저장 방식으로 변환한다. 알고리즘의 첫 단계에서 사실테이블을 메모리에 한꺼번에 로드될 수 있는 크기로 조각(partition)내어 디스크에 저장한다. 다음 단계에서 파티션을 1개씩 읽어들이면서 이를 디스크 한 블록 크기의 청크(chunk)들로 나누어 다시 디스크에 쓴다. 청크의 각 변의 길이는 가능한 한 비슷하게 만들어서 모든 차원에 공평한 기회를 주도록 한다. 그 다음, 청크들을 디스크로부터 1개씩 읽어들이면서 파이프라인 방식으로 큐브를 생성한다. 생성된 셀들은 디스크에 저장되기 전에 해당 튜플로 변환되어 저장된다. 메모리가 충분한 경우 각 청크는 1번씩만 메모리로 읽혀 들여진다. [1]은 메모리 제약이 있는 경우의 대책도 제시하고 있다. [1]에서는 ZDN 알고리즘의 성능을 정렬 기반의 ROLAP 알고리즘들 중에서 효율적인 Overlap 알고리즘과 실험적으로 비교하여, ZDN 알고리즘이 훨씬 좋은 성능을 나타낸다는 것을 보였다. ZDN 알고리즘은 배열을 이용하여 집계 테이블을 계산하기 때문에 정렬 기반 ROLAP 큐브 생성 알고리즘들에 비해 성능이 뛰어나다. 따라서 본 논문에서는 새로운 ROALP 큐브 생성 알고리즘을 제안하고 이 알고리즘의 성능을 ZDN 알고리즘과 비교, 분석하였다.

3. 새로운 ROLAP 큐브 생성 기법

이제 본 연구에서 제안하는 ROLAP 큐브 생성 알고리즘을 소개한다. 이는 분석대상인 사실테이블을 멤버의 수가 가장 큰 차원을 기준으로 슬라이스(slice)해 놓고 집계 연산의 순서를 조정함으로써 큐브를 생성할 때 필요한 메모리 양과 디스크 I/O를 감소시키는 방법이다. 큐브 생성 단계에서 사용되는 메모리는 가장 작은 1차 집계 테이블 1개와 이로부터 생성될 수 있는 2차 이상의 집계 테이블들을 저장할 공간으로 줄어든다. 자식 집계 테이블은 항상 최소 부모 집계 테이블로부터 생성된다.

우선 이 방법에 쓰인 기본 아이디어를 설명한 후에 1차 집계 테이블이 한 개조차 메모리에 상주할 수 없는 초대형 사실테이블을 취급하는 경우의 대책도 설명하기로 한다.

그림 1과 같이 3차원 사실테이블 ABC로부터 1차 집계 테이블 AB, AC, BC 를 생성하는 과정을 살펴보자. $|A| \le |B| \le |C|$라고 가정하자. 사실테이블을 그림 1(1)과 같이 차원 C를 기준으로 슬라이스하여 디스크에 저장해 놓는다. 즉, 한 슬라이스에 들어 있는 사실테이블 튜플들은 동일한 C값을 갖는다. 이제 슬라이스를 순서대로 1개씩 읽으면서 1차 집계 테이블 AB, AC, BC를 계산해 간다. 슬라이스를 읽는 동안에 1차 집계테이블인 AB는 항상 메모리에 상주해야 하지만, AC, BC는 그림 1(2)와 같이 1개의 행을 저장할 수 있는 공간인 A와 B만 메모리에 있어도 된다. i번째 슬라이스를 읽고 나면 AC, BC의 i번째 행의 값이 완성되어 디스크로 출력가능하기 때문이다. 또한 AC의 i행이 출력되기 전에 이를 A차원을 기준으로 집계 계산하여 C의 i번째 셀이 계산될 수 있다. 이 작업을 위해 필요한 총 메모리는 AB, A, B, all(상수 개) 이다. 집계 테이블 AB, AC, BC, C는 슬라이스된 사실테이블을 읽는 동안에 계산이 완료된다. 그 후에는 계산이 완료되어 현재 메모리에 있는 AB로부터 디스크 엑세스 없이 AB를 루트로 하는 트리에 속한 모든 집계 테이블을 계산한다. 즉, 그림 1(3)의 점선 타원형 안에 있는 집계 테이블 A, B, all이 파이프라인 방식으로 계산된다. 그리고, 이를 위한 메모리는 이미 확보되어 있다는 것을 알 수 있다.

이 방법을 4차원 데이타로 확장해 보자. 그림 2와 같이 사실테이블 $ABCD$ 로부터 큐브를 생성한다고 하고, $|A| \le |B| \le |C| \le |D|$라고 하자. $ABCD$를 D차원을 기준으로 슬라이스하여 디스크에 저장한다. 슬라이스된 $ABCD$를 순서대로 읽으면서 1차 집계 테이블인 ABC, ABD, ACD, BCD를 계산한다. D차원이 포함되지 않은 ABC는 슬라이스된 사실 테이블 전체가 스캔된 후에 완성되고, D차원이 포함된 ABD, ACD, BCD는 슬라이스를 1개씩 읽을 때마다 1행씩 계산된다. 따라서 가장 작은 집계 테이블인 ABC는 메모리에 있어야 하고, ABD, ACD, BCD의 경우는 1행에 해당하는 2차 집계 테이블인 AB, AC, BC의 공간만 있으면 된다. AD, BD, CD, D는 각각 ABD, ACD가 한 행씩 계산될 때마다 파이프라인 방식에 의해 한 행씩 생성된다. 슬라이스된 사실테이블이 모두 읽히고 나면 그림 2의 1단계 계산이 끝난 것이고 이 때 사용된 메모리는 그림 2의 2단계 계산에 속한 집계 테이블들의 계산에 재활용된다. 이제 ABC로부터 이를 루트로하는 트리에 속한 모든 집계 테이블들이 계산된다. 이 방법은 고차원 사실테이블의 경우에도 확장되어 사용될 수 있다. 알고리즘을 정리하면 다음과 같다. 여기서 최대 크기 차원을 D_M 이라고 부르기로 한다.

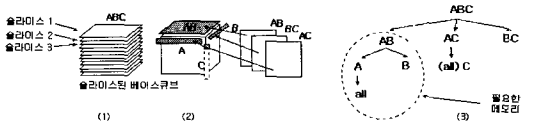


그림 1 집계테이블 생성을 위한 최소한의 메모리 공간

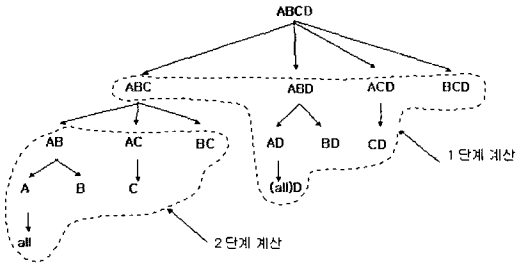


그림 2 4차원 사실 테이블로부터의 큐브 생성

<알고리즘 1>

[단계 1] 사실테이블을 D_M 차원 기준으로 슬라이스 한다. 즉, 사실테이블의 튜플들을 스캔하면서 D_M 차원 값이 같은 것끼리 모아 해당 슬라이스 버퍼에 넣는다. 버퍼가 찰 때마다 버퍼의 튜플들을 디스크에 있는 해당 슬라이스에 추가시킨다.

[단계 2] 디스크에 저장된 슬라이스를 1개씩 순서대로 읽으면서 모든 1차 집계 테이블의 값을 계산해 간다. 슬라이스 1개를 읽을 때마다 D_M 차원이 속한 1차 집계 테이블들의 경우는 1행씩 계산이 완료된다. 계산된 각 행으로부터 자신을 최소 부모로하는 모든 상위 집계 테이블들의 1행을 파이프라인 방식으로 계산한다. 계산이 완료된 행들을 테이블 형태로 변환하여 디스크에 저장한다. 모든 슬라이스에 대해 [단계 2]를 반복한다.

[단계 3] D_M 차원이 포함되지 않은 완성된 1차 집계 테이블로부터 이를 최소 부모로 하는 모든 2차 이상의 집계 테이블들을 파이프라인 방식으로 계산한다. 모든 계산 결과는 테이블 형태로 변환하여 디스크에 저장한다.

제안한 알고리즘을 분석해 보기로 한다. 첫째, 사실테이블을 슬라이스하는 [단계 1]에서는 각 튜플의 D_M 차

원 열만 스캔되면서 그 값을 기준으로 튜플이 해당 슬라이스 버퍼에 들어간다. 따라서 사실테이블 슬라이스 단계는 고속으로 처리될 수 있다. 만약 앞으로의 큐브 생성 단계에서의 디스크 I/O를 줄이고자 한다면 이 때 튜플의 모든 차원정보를 모아서 (슬라이스내의 오프셋, 값)의 2 열짜리 튜플로 변환하여 저장할 수도 있다.

둘째, [단계 2]와 [단계 3]에서는 연산 결과를 디스크에 쓰는 것 이외에 부수적인 디스크 I/O가 전혀 없고, 각 집계 테이블은 집계 연산시간을 최소화하는 부모 집계 테이블(smallest parent)로부터 계산된다. 또한 각 튜플은 집계 연산과정에서 단 한 번만 읽혀진다. 메모리에 전체가 상주해야 하는 1차 집계 테이블은 1차 집계 테이블 중에서 가장 작은 테이블뿐이다.

셋째, 사실테이블이 희박한 경우라고 해도 집계 테이블들의 밀도가 높아지므로 배열형태의 집계 테이블을 취급하는 것이 밀도가 지나치게 낮지 않는 한 테이블 형태의 집계 테이블들을 취급하는 것보다 효율적이다. 또한 제안한 알고리즘은 2차 이상의 상위 집계 테이블들을 계산할 때 메모리를 동적으로 할당하여 사용하고 셀들을 항상 정해진 순서대로 스캔하고 생성하기 때문에 셀의 배열 인덱스를 계산하는데 시간이 거의 들지 않는다.

넷째, 큐브 생성에 필요한 메모리 양은 D_M차원을 제외한 남은 차원들로 구성된 큐브의 크기와 같으므로 $\prod_{i=1}^{n-1}(d_i+1)$로 표현된다. 여기서, n은 사실테이블의 차원 수이고, d_i는 차원 $D_i,(i\neq M)$,의 멤버 수이다. 제안한 알고리즘과 ZDN 알고리즘을 메모리 사용 효율성을 기준으로 비교해 보자. ZDN 알고리즘은 큐브를 생성하기 전에 사실 테이블을 배열 형태로 변환하고, 이를 디스크 한 블록 크기 정도의 작은 청크들로 나누어 디스크에 저장한다. 배열에 저장한 n차원 사실 테이블(또는 베이스 큐브)의 모든 차원 D_i가 d개의 멤버를 갖고, 청크의 한 변의 길이가 c라고 할 때, 사실테이블을 1번 스캔하면서 큐브를 생성하기 위한 최소한의 메모리는 $c^n+(d+1+c)^{n-1}$이다[1]. 차원 멤버의 수가 동일하지 않은 경우는 이 식의 d 값을 $(\prod_{i=1}^{n-1}|D_i|)^{1/(n-1)}$로 놓는다[1]. 표 2에 두 알고리즘의 메모리 사용량을 비교한 결과가 있다. 표에서 D열은 사실테이블의 차원 수를 나타내고, d와 c는 위에서 언급한 값들을 말한다. i열은 차원 i의 멤버의 수이다. new(%)와 zdn(%)은 각각 제안한 알고리즘과 ZDN 알고리즘의 메모리 사용량을 배열 형태의 집계 테이블들 전체가 차지하는 메모리 공간과 비교한 것을 비율로 나타낸 것이다. new(%)와 zdn(%)에는 사실 테이블을 저장하기 위한 공간은 제외하였다. 사실 테이블을 메모리로 읽어들이기 위한 공간은 디스크 블록 1개 크기로 보면 된다. 여기서 c의 값이 차원마다 다르게 설정된 이유는 청크 1개가 디스크 블록 크기인 4K 바이트에 근접하도록 하였기 때문이다. 그림 2(1)과 같이 모든 차원 멤버의 수가 같은 경우 메모리 사용량이 최대가 되고, 그림 2(2)와 같이 차원간의 멤버 수가 차이를 보이면 두 방법 모두 메모리를 효율적으로 사용한다는 것을 알 수 있다. 또한 본 연구에서 제안한 알고리즘이 메모리를 더 효율적으로 사용한다는 것을 알 수 있다.

본 연구에서 제안한 알고리즘이 큐브를 생성하는데 사용하는 메모리의 양은 최소 크기 1차 집계 테이블 1개와 이 테이블을 최소 부모로 하는 모든 상위 집계 테이블을 계산하는데 드는 메모리 공간이다. 만약 이만큼의 메모리 공간이 확보될 수 없다면 사실 테이블을 2개 이상의 차원을 기준으로 슬라이스하여 필요한 메모리 공간을 줄일 수 있다. 예를 들어, 표 2와 같은 4차원 데이타를 보자. D차원만을 기준으로 사실테이블을 슬라이스 했을 때 필요한 메모리 공간은 ABC, AB, AC, BC, A, B, C, all인 반면에 C차원과 D차원을 함께 슬라이스 기준으로 했을 때 필요한 메모리 공간은 그림 3의 점선에 속한 부분인 AB, AB, A, B, all 이다. 메모리 사용량이 크게 줄어든다는 것을 알 수 있다.

표 2 제안한 알고리즘과 ZDN 알고리즘의 메모리 사용량의 비교

1,000	10	33.37%	34.04%
400	8	25.09%	26.24%
100	4	20.40%	29.93%
60	3	17.36%	22.07%
30	3	15.73%	27.38%
20	2	14.74%	27.86%

(1) 차원 멤버의 수가 같은 경우

50	1000	500000					224	10	0.01%	0.01%
60	180	900	3600				148	6	1.10%	1.23%
20	40	100	120	1200			56	4	0.94%	1.21%
12	18	24	45	90	1200		29	3	0.44%	0.68%
6	12	15	30	60	90	120	24	3	2.69%	4.79%
4	8	10	15	20	30	50	100	15	2	2.17%	4.36%

(2) 차원 멤버의 수가 다른 경우

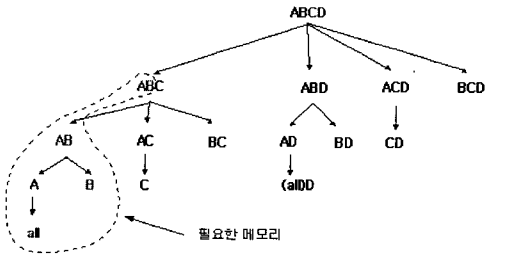


그림 3 C, D차원 기준으로 슬라이스한 경우 필요한 메모리

차원 2개를 기준으로 사실 테이블을 슬라이스하는 경우의 큐브 생성 과정을 살펴보자. 예로써, 그림 3과 같은 4차원 사실테이블을 사용하여 설명해 보기로 한다. $|A| \le |B| \le |C| \le |D|$라고 가정한다. 먼저 사실테이블을 C 차원과 D차원 기준으로 슬라이스하여 저장한다. 즉 한 슬라이스에 속한 튜플들은 동일한 C와 D값을 갖는다. 이제 슬라이스 C_iD_j $(1 \le C_i \le n, 1 \le D_j \le m)$를 순서대로 1개씩 읽으면서 1차 집계 테이블 ABD, ACD, BCD를 계산해 간다. 슬라이스들은 C를 행으로 보고, D를 열로 보았을 때 열우선순서(column major order)로 읽힌다. 예를 들면, C_1D_j, C_2D_j, C_3D_j, ... , C_nD_j, C_1D_{j+1}, C_2D_{j+2} 순서로 읽힌다. 따라서, 동일한 D값을 갖는 모든 튜플들이 그룹화되어 처리된다. 슬라이스 1개의 처리를 위해서 필요한 메모리는 다음과 같다. (D 차원만 포함되어 있는) ABD를 위해서는 메모리 공간 AB가 필요하고, (C차원과 D차원이 함께 포함되어 있는) ACD와 BCD를 위해서는 메모리가 각각 A와 B가 필요하다. 슬라이스 C_iD_j의 튜플들을 모두 스캔하고 나면 ABD의 경우는 중간 결과가 생성된 셈이고, C차원과 D차원이 모두 포함되어 있는 ACD와 BCD의 경우는 C_i와 D_j의 값을 갖는 모든 튜플들이 생성된 것이다. 이렇게 생성된 튜플들은 디스크로 옮겨지게 되는데, ACD의 경우는 디스크에 쓰기 전에 CD의 한 행을 계산한다. CD의 최소 부모가 ACD이기 때문이다. 이와 같은 방식으로 D차원의 값이 동일한 슬라이스들의 처리가 끝나면 그림 4의 1단계 계산이 끝난다. 1단계 계산이 끝나고 나면, 즉 동일한 D값을 갖는 모든 슬라이스의 처리가 끝나면, ABD의 한 행이 완성된 것이다. 이를 디스크로 출력하기 전에 ABD를 최소 부모로하는 2차 이상의 집계 테이블들의 1 행이 계산되어 디스크로 출력된다. 각 D값에 대해 1 단계와 2 단계와 이와 같이 반복적으로 실행된다. 사실 테이블 전체가 스캔되고 나면 1단계와 2단계에 포함된 모든 집계 테이블의 생성이 완료된다.

이제 사실 테이블을 한 번 더 스캔하면서 3단계와 4단계를 실행한다. 이번에는 사실테이블 슬라이스들을 순서대로 읽는데, C값이 같은 슬라이스들을 그룹화하여 읽는다. 동일한 C값을 갖는 튜플들이 모두 스캔되고 나면 3단계에 속한 ABC, AC, BC, C, all의 한 행이 계산되어 디스크로 출력된다. 사실테이블의 스캔이 모두 끝나면 AB가 완성되고 이제 4단계가 시작된다. 4단계에서 남은 A, B, all이 계산된다. 2개의 차원으로 사실테이블을 슬라이스해도 메모리가 부족한 경우에는 3개 이상의 차원을 사용하여 사실 테이블을 슬라이스할 수 있다. 이 알고리즘은 일반화하여 정리하면 다음과 같다.

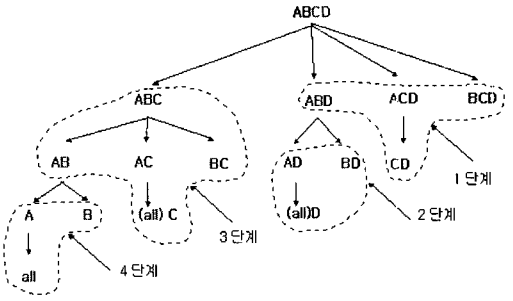


그림 4 두 차원으로 슬라이스한 경우 큐브 생성

알고리즘 2

가정 : 슬라이스 기준이 되는 차원들을 $D_{i_1}, D_{i_2}, \cdots, D_{i_s}$라고 하자. 여기서, $|D_{i_1}| \le |D_{i_2}| \le \cdots \le |D_{i_s}|$이고, 이들은 사실테이블의 차원들 중에서 멤버의 수가 가장 많은 s개의 차원들이다. $A = \{D_{i_2}, D_{i_3}, \cdots, D_{i_s}\}$라고 하자.

[단계 1] 사실테이블의 튜플들을 스캔하면서 $D_{i_2}, D_{i_3}, \cdots, D_{i_s}$ 차원 값이 동일한 튜플들을 동일한 슬라이스를 위한 버퍼에 넣는다. 버퍼가 찰 때마다 버퍼의 튜플들을 디스크에 있는 해당 슬라이스에 추가시킨다.

A의 모든 부분집합에 관해 [단계 2]를 실행한다. [단계 2]가 A의 부분집합인 B에 관해 실행된다고 가정하자.

[단계 2] $\{D_{i_1}\} \cup B$를 기준으로 파티션한 슬라이스를 1개씩 읽으면서 $\{D_{i_1}\} \cup B$이 포함된 집계 테이블들의 각 행을 파이프라인 방식으로 계산하여 디스크에 저장한다. D_{i_1}은 포함하지 않지만 B에 속한 모든 차원을 포함하는 집계 테이블은 $|D_{i_1}|$개의 슬라이스를 처리한 후에 계산이 완료된다. 이 집계 테이블은 디스크에 저장되기 전에 자신을 최소 부모 집계 테이블로 하는 모든 집계 테이블들을 파이프라인 방식으로 계산한다. 모든 슬라이스를 읽을 때까지 위의 과정이 반복되며 디스크에 저장되는 모든 집계 테이블들은 저장되기 전에 테이블 형태로 변환된다.

알고리즘 2의 메모리 사용량을 계산해 보자. 그림 3의 예를 보면 필요한 메모리는 AB를 루트로하는 트리에 속한 모든 집계 테이블을 저장하는 공간 (즉, AB를 사실 테이블로 보았을 때 생성되는 전체 큐브를 위한 공간)과 AB를 위한 공간이다. 이를 일반화시키면 다음과 같다. 사실 테이블의 차원 수를 n이라 하고, 이를 슬라

이스할 때 기준되는 차원의 수를 s라고 하자. $|D_1| \le |D_2| \le \cdots \le |D_n|$이고 d_i는 D_i 차원의 멤버의 수라고 하자. 이 때 알고리즘 2에 필요한 메모리의 양은 $\prod_{i=1}^{s} d_i + \prod_{i=1}^{s}(d_i+1)$이 된다. 표 3과 4에 알고리즘의 메모리 사용량을 가상의 데이타에 적용한 예가 있다. 표에서 'base cube size'란 사실 테이블을 배열에 압축없이 저장하는 경우의 셀의 개수를 나타낸다. 데이타의 크기를 1~10 테라 셀 정도로 하였다. 표에서 'tree'는 $\prod_{i=1}^{s}(d_i+1)$을 나타내고, 'extra(root)'는 그 트리의 루트에 해당하는 추가 공간인 $\prod_{i=1}^{s} d_i$ 을 나타낸다. 알고리즘의 전체 메모리 사용량은 'mem req(%)'로써 이는 사실테이블을 제외한 모든 집계 테이블을 배열로 저장하는 경우에 드는 메모리의 크기와 알고리즘이 큐브 생성에 필요로하는 메모리의 크기를 비율로 표현한 것이다. 표 3은 1개의 차원을 기준으로 사실 테이블을 슬라이스한 경우를 나타내고, 표 4는 2개의 차원을 기준으로 사실 테이블을 슬라이스한 경우를 나타낸다. 알고리즘은 특히 실제 데이타의 모양에 가까운 차원의 멤버의 개수가 다른 경우에 메모리 사용량이 크게 줄어들어 있는 것을 알 수 있다.

알고리즘 2가 실행되는 동안에 슬라이스된 사실 테이블은 단계 2가 실행될 때마다 스캔된다. 즉 이는 A의 모든 부분집합의 개수와 동일하다. 예를 들어 2개 차원 기준으로 슬라이스하는 경우는 $|A|=1$이 되어 사실 테이블은 1회 스캔된다. 3개 또는 4개 차원으로 슬라이스하는 경우는 사실 테이블이 각각 4회와 8회가 스캔된다. 그러나, 표 3, 4에 보이는 것과 같이 일반적인 경우에 사실테이블을 2~3개 이상의 차원으로 슬라이스할 필요는 그다지 많지 않을 것으로 본다.

표 3 알고리즘의 메모리 요구량 (차원 1개를 사용하여 슬라이스한 경우)

(1) 차원 멤버의 수가 같은 경우

A	B	C	D	E	F	G	H	base cube size	tree	extra(root)	mem req(%)
10,000	10,000	10,000						1,000,000,000,000	10,001	10,000	0.007%
1,000	1,000	1,000	1,000					1,000,000,000,000	1,002,001	1,000,000	0.050%
300	300	300	300	300				2,430,000,000,000	27,270,901	27,000,000	0.133%
100	100	100	100	100	100			1,000,000,000,000	104,060,401	100,000,000	0.332%
60	60	60	60	60	60	60		2,799,360,000,000	844,596,301	777,600,000	0.472%
40	40	40	40	40	40	40	40	6,553,600,000,000	4,750,104,241	4,096,000,000	0.618%
								Tera	Giga	Giga	

(2) 차원 멤버의 수가 다른 경우

A	B	C	D	E	F	G	H	base cube size	tree	extra(root)	mem req(%)
1,000	10,000	100,000						1,000,000,000,000	1,001	1,000	0.000%
500	700	1,500	2,000					1,050,000,000,000	351,201	350,000	0.015%
100	200	300	400	500				1,200,000,000,000	6,110,601	6,000,000	0.044%
10	50	100	200	300	400			1,200,000,000,000	11,388,861	10,000,000	0.012%
10	20	50	70	90	100	200		1,260,000,000,000	76,117,041	63,000,000	0.049%
10	20	30	40	50	60	70	80	4,032,000,000,000	913,392,711	720,000,000	0.134%
								Tera	Giga	Giga	

표 4 알고리즘의 메모리 요구량 (차원 2개를 사용하여 슬라이스한 경우)

(1) 차원 멤버의 수가 같은 경우

A	B	C	D	E	F	G	H	base cube size	tree	extra(root)	mem req(%)
1,000	1,000	1,000	1,000					1,000,000,000,000	1,001	1,000	0.0000%
300	300	300	300	300				2,430,000,000,000	90,601	90,000	0.0004%
100	100	100	100	100	100			1,000,000,000,000	1,030,301	1,000,000	0.0033%
60	60	60	60	60	60	60		2,799,360,000,000	13,845,841	12,960,000	0.0078%
40	40	40	40	40	40	40	40	6,553,600,000,000	115,856,201	102,400,000	0.0152%
								Tera	Mega	Mega	

(2) 차원 멤버의 수가 다른 경우

A	B	C	D	E	F	G	H	base cube size	tree	extra(root)	mem req(%)
500	700	1,500	2,000					1,050,000,000,000	501	500	0.0000%
100	200	300	400	500				1,200,000,000,000	20,301	20,000	0.0001%
10	50	100	200	300	400			1,200,000,000,000	56,661	50,000	0.0001%
10	20	50	70	90	100	200		1,260,000,000,000	836,451	700,000	0.0005%
10	20	30	40	50	60	70	80	4,032,000,000,000	14,973,651	12,000,000	0.0022%
								Tera	Mega	Mega	

4. 제안한 방법의 효율성 검증

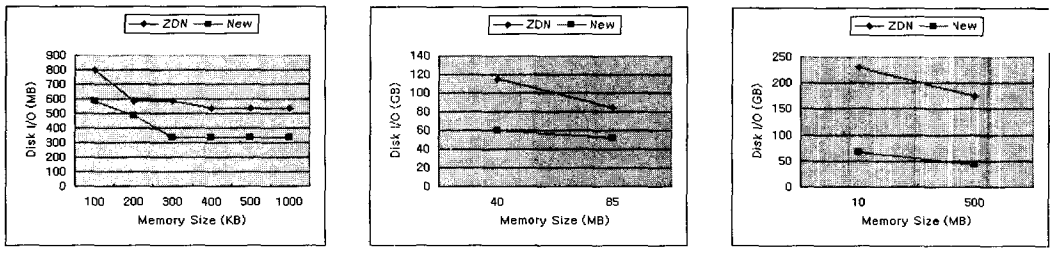
본 연구에서 제안한 알고리즘과 ZDN 알고리즘의 속도를 비교해 보기로 한다. 제안한 알고리즘의 성능을 ZDN 알고리즘과 비교하는 이유는 이미 [1]에서 ZDN 알고리즘이 기존의 ROLAP 큐브 생성 알고리즘들보다 훨씬 효율적이라는 것을 증명해 놓았기 때문이다. 또한 ZDN 알고리즘 역시 메모리의 여유가 있는 경우에는 사실 테이블을 한 번만 스캔하면서 MOLAP 큐브 생성 알고리즘을 통해 큐브 전체를 생성하므로 본 연구에서 제안한 방법과 유사하면서 효율적인 알고리즘이기 때문이다. 편의상 앞으로 본 연구에서 제안한 알고리즘을 언급할 때 New 알고리즘이라고 하기로 한다.

4.1 디스크 I/O 시간 측정을 통한 New 알고리즘의 성능 평가

우선 두 알고리즘의 디스크 I/O 양을 계산해 보기로 한다. 디스크 I/O는 메모리에 제약이 있는 경우, 사실 테이블이 희박한 경우, 사실 테이블의 차원이 높은 경우를 대상으로 계산하였다. 메모리 크기에 제약이 있는 경우를 살펴보자. 메모리가 충분하지 않으면 New 알고리즘은 2개 이상의 차원으로 사실 테이블을 슬라이스한다. ZDN 방법 역시 이를 해결하기 위해 [1]에 제안되어 있는 Multi-Pass 알고리즘을 사용한다. 이 알고리즘은 큐브 트리를 여러 개의 서브 트리로 나누어 서브 트리 한 개씩 처리하는 방법이다. 실험에는 [1]에서 사용한 데이타 세트인 4차원 사실 테이블을 사용하였다. 사실 테이블의 각 차원 크기는 40, 40, 40, 1000으로 하였다. 데이타 밀도는 10%으로 하여 사실 테이블에는 6,400,000개의 튜플이 있고 사실 테이블이 차지하는 공간은 128MB이다. 메모리 크기를 100KB에서 1MB까지 변화시켜 가면서 두 알고리즘의 디스크 I/O 분량을 계산한 것이 그림 5(1)에 있다. ZDN의 경우 메모리 크기가 400KB 이상이면 전체 큐브가 한꺼번에 생성될 수 있고, 200KB일 때는 서브 트리가 2개 생기고, 100KB인 경우는 서브 트리가 3개 생기게 된다. 반면에 New 알고리즘의 경우는 메모리가 300KB 이상이면 1개 차원 기준으로 사실 테이블을 슬라이스하면 되고, 메모리가 그 이하인 경우는 2개 차원을 기준으로 사실 테이블을 슬라이스해야 한다. New 알고리즘이 디스크 I/O를 적게 한다는 것을 알 수 있다.

사실 테이블의 밀도가 낮은 경우를 살펴보자. 밀도가 0.1%인 4차원 사실 테이블을 가정하자. 각 차원의 크기는 100, 200, 1000, 100000이다. 사실 테이블의 튜플 수는 1기가(giga) 개이고 테이블이 차지하는 공간은 20GB이다. 메모리의 크기가 40MB인 경우와 85MB인 경우를 고려하였다. 메모리가 85MB인 경우 ZDN 알고리즘은 큐브 생성에 필요한 메모리를 충분히 가지고 있고, New 알고리즘은 사실 테이블을 슬라이스하는데 1개의 차원만 사용하면 된다. 메모리가 40MB 인 경우는 ZDN 알고리즘의 경우 서브 트리 2개가 발생하고 New 알고리즘은 2개의 차원을 기준으로 사실 테이블을 슬라이스 해야 한다. ZDN 알고리즘의 경우는 큐브 생성 도중에 디스크 I/O를 하는 경우 데이타 희박성을 줄이기 위해 배열을 압축한 채로 디스크 I/O를 한다는 가정을 하였다. 그럼에도 불구하고 그림 5(2)와 같이 New 알고리즘이 디스크 I/O를 작게 한다는 것을 알 수 있다.

사실 테이블이 고차원 데이타인 경우를 고려해 보자. 밀도가 0.1%인 6차원 사실 테이블을 선택하였다. 각 차원의 크기는 10, 10, 100, 100, 200, 10000으로 하였다. 테이블의 크기는 28GB이다. 메모리의 크기는 10MB, 500MB인 경우로 정하였다. ZDN 알고리즘의 경우는 메모리가 10MB, 500MB일 때 각각 서브 트리가 3개, 2개 발생한다. 이 때 발생하는 오버헤드는 디스크 I/O의 60% 이상으로 차원이 높아지면 오버헤드가 급격하게 커진다는 것을 알 수 있다. New 알고리즘의 경우는 메모리가 500MB인 경우는 1개 차원 기준으로 사실 테이



(1) 메모리 제약이 있는 경우 (2) 사실 테이블의 밀도가 낮은 경우 (3) 사실 테이블이 고차원인 경우

그림 5 ZDN과 New의 디스크 I/O 비교

블을 슬라이스를 하고, 메모리가 10MB인 경우는 2개 차원 기준으로 사실 테이블을 슬라이스하여 오버헤드가 발생하지만 이는 전체 디스크 I/O의 11%정도이다. 차원이 큰 데이타일수록 New 알고리즘의 효율성이 커진다는 것을 알 수 있다.

디스크 I/O를 제외한 CPU 시간은 New 알고리즘이 적게 걸린다. 그 이유는 ZDN 알고리즘의 경우 청크 단위로 데이타를 관리하기 때문에 셀의 위치를 계산하기 위한 인덱스 계산에 오버헤드가 든다. 또한 희박한 청크의 경우는 압축되어 저장되는데 이 때 압축된 셀 인덱스는 나중에 차원 정보로 변환되어 사용되어야 하기 때문이다. 본 연구에서는 2차 이상의 집계 테이블들을 항상 정해진 순서로 스캔하기 때문에 인덱스 계산을 위한 CPU 시간이 거의 들지 않는다.

4.2 실험을 통한 New 알고리즘의 효율성 검증

실험에는 512MB 메모리와 20GB 하드 디스크가 장착되고 SunOS 5.7 버전의 운영체제가 장착된 Sun UltraSparc 10 워크스테이션을 사용하였다. 가능한 한 정확한 분석결과를 얻기 위해 ZDN이 사용한 데이타 세트를 사용하였다. 각 실험결과는 10회 실시한 후 평균값으로 나타내었다. 사실테이블의 각 열은 4바이트를 차지하는 것으로 가정하고, ZDN과 마찬가지로 데이타는 큐브 내에 균일하게 분포되었다고 가정하였다. 실험에 사용한 데이타 세트는 다음과 같다.

데이타 세트 1 : 유효 셀의 개수를 고정하고, 차원 멤버의 수만 변경한다. 4차원 데이타로, 3개의 차원은 크기가 40이고, 남은 차원은 크기가 각각 400, 1,000, 10,000이다. 유효 셀 개수는 모두 6,400,000개이며, 데이타의 밀도는 각각 25%, 10%, 1%이다. 따라서 사실테이블의 크기는 128.5MB이다.

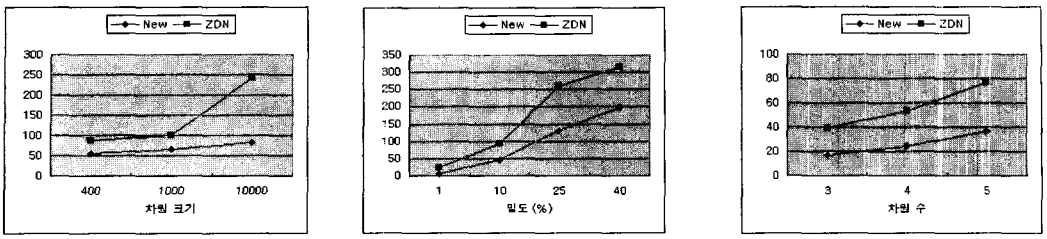
데이타 세트 2 : 차원 크기를 고정하고, 데이타 밀도를 변경한다. 모든 데이타 집합은 4차원이며, 크기는 $40\times40\times40\times1000$이다. 데이타의 밀도는 각각 1%, 10%, 25%, 40%이며, 대응되는 사실테이블의 크기는 각각 12.8MB, 128MB, 321MB, 512MB이다.

데이타 세트 3 : 유효 셀 개수와 데이타 밀도를 고정하고, 차원 수를 변경한다. 3, 4, 5차원 데이타로, 각각의 크기는 $40\times400\times4000$, $40\times40\times40\times1000$, $10\times40\times40\times40\times100$이다. 유효 셀 개수는 3,200,000개이고 밀도는 5%이다. 대응되는 사실테이블의 크기는 51.2MB, 64MB, 76.8MB이다.

데이타 세트 1, 2, 3을 사용하여 ZDN 알고리즘과 New 알고리즘의 효율성을 측정한 결과가 그림 6에 있다. 그림 6(1)의 경우는 사실테이블 튜플 수를 고정하고 D_M차원의 크기를 변화시켜 사실 테이블의 밀도를 25%, 10%, 1%로 변화시키면서 큐브 생성시간을 측정한 것이다. 즉, 입력 데이타의 분량은 동일하나 베이스 큐브내의 무효 셀 개수와 집계 연산 후 생성되는 셀 개수가 다른 경우이다. D_M의 크기가 10,000인 경우는 400인 경우보다 무효 셀의 분량이 25배 많다. 또한 베이스 큐브의 데이타 분포가 균일하다고 가정할 때에 집계 계산 후 생성되는 유효 셀의 개수 역시 D_M의 크기가 400인 경우는 88만 셀 정도인데, 10,000 인 경우는 20메가 셀로써 23배에 달한다. 그러나 알고리즘 실행시간을 보면 ZDN과 New 알고리즘 모두 각각 3배, 1.3배만 증가했을 뿐이다. 사실 테이블이 희박하다고 해도 두 알고리즘의 실행시간은 안정적이며 큰 변화가 없다는 것을 알 수 있다. 특히 New 알고리즘은 사실 테이블에서 1차 집계 테이블들을 생성할 때를 제외하고는 집계 테이블의 셀을 항상 정해진 순서로 스캔하고 생성하기 때문에 셀 인덱스의 계산에 거의 시간이 들지 않는다. 이로 인해 집계 테이블이 희박한 경우도 무효 셀을 스캔하는데 시간이 많이 들지 않는다.

그림 6(2)의 경우는 사실 테이블의 모양을 고정한 채로 큐브의 밀도만을 변경시키면서 실험한 결과이다. ZDN 알고리즘의 경우 데이타 밀도가 10%~40% 사이에서 실행시간이 많이 드는 이유는 청크를 압축하는 일과 증가된 데이타를 디스크로 쓰고 읽는 일에 시간이 많이 소요되기 때문이다. 데이타의 밀도가 40% 정도가 되면 ZDN 알고리즘의 경우 데이타 압축단계가 없어진다. 두 알고리즘 모두 베이스 큐브의 밀도가 높아짐에 따라 실행시간이 안정적으로 증가한다는 것을 알 수 있다. 밀도가 10%에서 25%, 40%로 증가할 때 실행시간이 2.9배, 4.3배 증가한 것을 볼 때, New 알고리즘은 베이스 큐브의 밀도 증가에 매우 안정적인 방법임을 알 수 있다.

그림 6(3)의 경우는 사실 테이블의 튜플 수를 고정하고 그 대신 데이타의 차원 수를 증가시키면서 큐브 생성 시간을 측정한 것이다. 각 경우 유효 셀의 개수를 3.2 메가 셀 (밀도 5%)로 고정하였다. 사실테이블의 크기가 51.2BM, 64MB, 76.8MB인 이유는 차원이 증가하면서 사실 테이블의 튜플 크기가 증가했기 때문이다. 3차원, 4차원, 5차원의 경우 생성해야 하는 집계 테이블의 수는 각각 7, 15, 31개이고, 데이타 균일 분포를 가정했을 때 생성되는 셀의 수는 각각 1.8M, 5.0M, 12.6M개이다. 차원이 증가할수록 New 알고리즘이 ZDN 알고리즘에 비해 우수한 성능을 보인다는 것을 알 수 있다. 종합적으로, New 알고리즘은 데이타의 밀도, 차원의 증가에 안정적이고 효율적인 알고리즘이다.



(1) 데이타 세트 1을 사용한 실행시간 (2) 데이타 세트 2를 사용한 실행시간 (3) 데이타 세트 3을 사용한 실행시간

그림 6 다양한 데이타 세트를 사용한 ZDN과 New의 실행시간 비교 (실행시간 단위: 초)

5. 결 론

본 연구에서는 MOLAP 큐브 생성 알고리즘을 사용하여 ROLAP 집계 테이블을 신속하게 생성하는 알고리즘을 제안하였다. 이 알고리즘은 가장 큰 차원들을 기준으로 사실 테이블을 슬라이스하고, 집계 테이블 생성에 필요한 메모리를 체계적으로 재사용함으로써 큐브 생성에 필요한 메모리 사용량을 크게 줄인다. 메모리 사용량을 감소시켜서 모든 집계 테이블이 메모리 상에서 최소 부모 집계 테이블로부터 생성될 수 있도록 하여 집계 연산의 시간을 감소시켰다. 또한 2차 이상의 집계 테이블들을 생성할 때는 셀을 순서대로 스캔하기 때문에 셀의 인덱스를 계산하는 시간을 줄였다. 또한 알고리즘이 간단하여 구현하여 사용하기 수월하다는 장점을 갖는다. 기존의 ROLAP 큐브 생성 알고리즘들보다 성능이 월등하게 나은 [1]의 큐브 생성 알고리즘과 비교한 결과 본 연구에서 제안한 알고리즘이 사실 테이블의 밀도, 차원의 개수에 안정적이고 효율적이라는 것을 보였다.

참 고 문 헌

[1] Yihong Zhao, Prasad Deshpande, Jeffrey Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," Proc. ACM SIGMOD '97, pp. 159-170.

[2] Business Intelligence Ltd, "The Olap Report : Database Explosion," http://www.olapreport.com/DatabaseExplosion.htm, 2000.

[3] Won Kim and Myung Kim, "Performance and Scalability in Knowledge Engineering: Issues and Solutions," Journal of Object-Oriented Programming, Vol. 12, No. 7, pp. 39-43, Nov/Dec. 1999.

[4] Erik Thomsen, "OLAP Solutions: Building Multidimensional Information Systems," John Wiley & Sons, New York, 1997.

[5] MicroStrategy, Inc., "The Case for Relational-OLAP," White Paper, http://www.microstrategy.com/files/whitepapers/wp_rolap.pdf, 2000.

[6] Informix Corporation, "Informix MetaCube 4.2: Delivering the Most Flexible Business-Critical Decision Support Environments," http://www.informix.com/informix/products/tools/metacube/metacube_ds.pdf, 1999.

[7] Information Advantage, "OLAP-Scaling to the Masses", White Paper, http://www.infoadvan.com/, 2000.

[8] Sameet Agarwal, Rakesh Agrawal, Prasad M. Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramacrishnan, Sunita Sarawagi, "On the Computation of Multidimensional Aggregates," Proc.of the 22nd VLDB Conference, 1996.

[9] Hyperion Corp. "Large-Scale Data Warehousing Using Hyperion Essbase OLAP Technology," http://www.hyperion.com/downloads/teraplex.pdf, Jan. 2000.

[10] Microsoft Co. "Product Overview," http://www.microsoft.com/sql/productinfo/prodover.htm, 2000.

[11] Oracle Corporation, "Oracle Express Server: Delivering OLAP to the Enterprise," http://otn.oracle.com/products/exp_server/pdf/expsrv97.pdf, 1997.

[12] Sunita Sarawagi and Michael Stonebraker, "Efficient Organization of Large Multidimensional Arrays," Proc. of 10th Data Engineering Conference, Feb. 1994.

[13] Kenneth A. Ross and Divesh Srivastava, "Fast Computation of Sparse Datacubes," Proc. of 23th VLDB, pp. 116-185, 1997.

김 명

1981년 이화여자대학교 수학과 학사. 1983년 서울대학교 계산통계학과 석사. 1990년 미네소타대학교 컴퓨터학과 석사. 1993년 캘리포니아 주립대학교 (산타바바라 소재) 박사. 1993년 ~ 1994년 캘리포니아 주립대학교 (산타바바라 소재) Posdoc, 강사. 1995년 ~ 1999년 이화여자대학교 컴퓨터학과 조교수. 2000년 ~ 현재 이화여자대학교 컴퓨터학과 부교수. 관심분야는 지식공학, OLAP, 인터넷 기술, 고성능 컴퓨팅 등

송 지 숙

2000년 이화여자대학교 컴퓨터학과 학사. 2000년 ~ 현재 이화여자대학교 컴퓨터학과 석사과정. 관심분야는 OLAP, 병렬 컴퓨팅, 데이타 마이닝 등