청크 기반 MOLAP 큐브를 위한 비트맵 인덱스
(A Bitmap Index for Chunk-Based MOLAP Cubes)

임 윤 선 † 김 명 ††
(Yoonsun Lim) (Myung Kim)

요 약 다차원 온라인 분석처리 (MOLAP, Multidimensional On-line Analytical Processing) 시스템은 데이타를 큐브라고 불리는 다차원 배열에 저장하고 배열 인덱스를 이용하여 데이타를 엑세스한다. 큐브를 디스크에 저장할 때 각 변의 길이가 같은 작은 청크들로 조각내어 저장하게 되면 데이타 클러스터링 효과를 통해 모든 차원에 공평한 질의 처리 성능이 보장되며, 이러한 큐브 저장 방법을 '청크기반 MOLAP 큐브' 저장 방법이라고 부른다. 공간 효율성을 높이기 위해 밀도가 낮은 청크들은 또한 압축되어 저장되는데 이 과정에서 데이타의 상대 위치 정보가 상실되며 원하는 청크들을 신속하게 엑세스하기 위해 인덱스가 필요하게 된다. 본 연구에서는 비트맵을 사용하여 청크기반 MOLAP 큐브를 인덱싱하는 방법을 제시한다. 인덱스는 큐브가 생성될 때 동시에 생성될 수 있으며, 인덱스 수준에서 청크들의 상대 위치 정보를 보존하여 청크들을 상수 시간에 검색할 수 있도록 하였고, 인덱스 블록마다 가능한 많은 청크들의 위치 정보가 포함되도록 하여 범위 질의를 비롯한 OLAP 주요 연산 처리 시에 인덱스 엑세스 회수를 크게 감소시켰다. 인덱스의 시간 공간적 효율성은 다차원 인덱싱 기법인 UB-트리, 그리드 파일과의 비교를 통해 검증하였다.
키워드 : 다차원 온라인 분석처리, 청크기반 MOLAP 큐브, 다차원 인덱스

Abstract MOLAP systems store data in a multidimensional array called a 'cube' and access them using array indexes. When a cube is placed into disk, it can be partitioned into a set of chunks of the same side length. Such a cube storage scheme is called the chunk-based MOLAP cube storage scheme. It gives data clustering effect so that all the dimensions are guaranteed to get a fair chance in terms of the query processing speed. In order to achieve high space utilization, sparse chunks are further compressed. Due to data compression, the relative position of chunks cannot be obtained in constant time without using indexes. In this paper, we propose a bitmap index for chunk-based MOLAP cubes. The index can be constructed along with the corresponding cube generation. The relative position of chunks is retained in the index so that chunk retrieval can be done in constant time. We placed in an index block as many chunks as possible so that the number of index searches is minimized for OLAP operations such as range queries. We showed the proposed index is efficient by comparing it with multidimensional indexes such as UB-tree and grid file in terms of time and space.
Key words : OLAP, Chunk-based MOLAP Cube, Multidimensional Indexes

1. 서 론

온라인 분석처리(OLAP, On-Line Analytical Processing)는 데이타를 다차원적으로 분석하여 그 결과를 사용자에게 실시간에 제공해 줌으로써 기업이 경영 전략이나 마케팅 전략을 세우는데 필요한 지식공학 기술이다. 데이타의 신속한 다차원적 분석을 위해 OLAP 시스템들은 주로 다차원 집계 데이타를 미리 계산하여 저장해 놓게 되는데, 이 때 집계 데이타의 생성, 저장, 검색 방식은 OLAP 시스템의 성능에 큰 영향을 끼치게 된다. 본 연구에서는 집계 데이타의 효율적인 저장을 위한 인덱스 구조의 설계에 초점을 맞추고 있다.

· 본 연구는 한국과학재단 목적기초연구(R04-2001-000-00191-0) 지원에 의하여 수행되었음
† 비 회 원 : 이화여자대학교 과학기술대학원 컴퓨터학과
lys96@ewha.ac.kr
†† 종신회원 : 이화여자대학교 과학기술대학원 컴퓨터학과 교수
mkim@ewha.ac.kr
논문접수 : 2002년 10월 15일
심사완료 : 2003년 3월 17일

데이타의 다차원적 분석을 대형 할인 체인점의 판매 데이타를 예제로 하여 살펴보자. 물품 판매액 데이타가 체인점별, 시기별, 물품별로 저장되어 있다고 하자. 이와 같은 3차원 데이타로부터 체인점에 관계없이 시기별 물품별로 판매액의 집계 데이타를 구하거나, 시기에 관계없이 체인점별 물품별로 판매액의 집계 데이타를 구하는 연산은 흔히 발생하는 OLAP 연산이다. 이 때 사용자가 어떤 차원들의 조합을 기준으로 집계 데이타를 요청할 것인지 미리 예측할 수 없으므로 OLAP 시스템들은 차원들의 모든 조합별로 집계 데이타를 계산하는 큐브 연산(cube operator)을 제공한다. 본 연구에서는 큐브 연산을 통해 계산된 집계 데이타 세트를 'OLAP 큐브'로 부르기로 한다.

큐브의 저장 방식에 따라 OLAP 시스템들은 크게 ROLAP(relational OLAP) 시스템과 MOLAP(multi-dimensional OLAP) 시스템으로 나뉜다. ROLAP 시스템들은 큐브를 관계형 데이타베이스에 테이블 형태로 저장하고 RDBMS 질의 처리 기술을 바탕으로 OLAP 질의를 처리한다. 이러한 방식은 다양하고 복잡한 질의 처리를 허용하지만 처리 속도가 느리고 데이타의 차원 정보가 함께 저장되므로 저장공간이 낭비된다는 단점이 있다. 반면에 MOLAP 시스템들은 큐브를 다차원 배열에 저장한다. 예를 들어, 위의 3차원 데이타는 3차원 배열에 저장하고, 2차원 집계 테이블들은 2차원 배열 형태로 저장한다. 이와 같이 데이타의 위치가 고정적이므로 데이타가 신속하게 엑세스될 수 있다. 반면, 배열에 유효(valid) 데이타가 적은 경우에는 다양한 데이타의 압축 기술[1, 2]이 사용되고 있다.

MOLAP 큐브는 질의 처리 속도를 높이기 위해 다양한 방식으로 저장될 수 있다. 그 중의 하나가 '청크 기반의 MOLAP 큐브 저장 방법'[2, 3]이다. 이 방법은 각 집계 테이블을 독립적인 배열에 저장하며, 각 배열을 한 변의 길이가 동일한 작은 조각(청크, chunk)들로 나누어 압축한 후 [2, 3], 이들을 데이타가 클러스터를 이룰 수 있도록 Hilbert 인덱스 또는 Z 인덱스 순서로 저장한다[4]. 이 때 각 집계 테이블은 청크된 파일(chunked file)[5]에 저장되었다고 한다.

배열의 데이타를 청크 단위로 조각 내어 디스크에 저장하는 이유는 OLAP의 주요 연산인 슬라이스(slice)와 다이스(dice)와 같은 범위 질의처리 속도가 특정 차원에 불리하지 않도록 하려는 것이다. 예를 들어, 그림 1(a)와 같이 배열이 그대로 디스크에 저장되어 있다고 하자. 디스크 블록의 크기를 4K 바이트로 한다면 한 블록에 화살표 방향의 한 행이 저장된다. 이 때 *BC*차원에 수평인 평면 데이타를 읽는 슬라이스 연산을 실행하면 데이타가 저장되어 있는 디스크의 모든 블록에는 연산에 필요한 데이타 셀이 한 개씩 들어 있어서 배열 전체를 읽어야 한다. 그러나 배열을 그림 1(b)와 같이 청크들로 저장하게 된다면 모든 차원에 공평한 연산 속도가 보장되고 디스크 엑세스 시간을 줄일 수 있다.

데이타 밀도가 낮은 청크들은 압축되어 저장된다. 예를 들어, 위의 예제에서와 같이 판매액 데이타를 3차원 배열에 저장하는 경우 모든 체인점이 항상 모든 물품을 판매하는 것이 아니기 때문에 배열은 희박한 경우가 많다. 저장 공간을 최소화하기 위해 밀도가 낮은 청크들은 압축되어 저장된다. 이 때 청크들의 크기가 달라지면서 청크 위치와 크기 정보가 손실되고, 인덱스를 사용하지 않고는 원하는 청크를 신속하게 찾기 힘들게 된다.

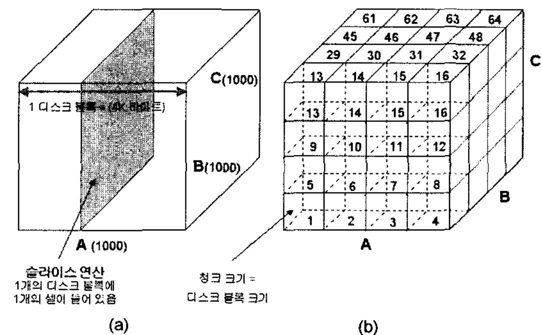


그림 1 디스크에 배열 저장하는 방법 (a) 특정 차원 기준으로 1차원 배열화한 경우, (b) 청크 단위로 저장하는 경우

청크들을 엑세스하기 위해서, 이들을 다차원 공간 상의 한 점으로 간주하고 기존의 다차원 인덱스 구조인 해쉬 기반의 그리드 파일[6], MLGF (Multi-Level Grid File)[7] 또는 트리 기반의 R-트리[8], UB-트리[9] 등을 쓸 수 있다. 그러나 이와 같은 인덱스들은 청크를 인덱싱하기 위해서 각 청크마다 기본 정보로 청크 번호, 밀도 상태, 주소를 인덱스 차원에서 가지고 있어야 하므로 저장공간을 많이 차지한다. 따라서 인덱스 블록에 많은 청크 정보가 포함될 수 없게 되어 범위 질의 속도가 저하된다. 본 연구에서는 시간 공간적 측면에서 기존의 인덱스들보다 좀 더 효율적인 인덱스 구조를 설계하는 것을 목표로 한다.

청크된 파일의 특징을 살펴보면, 청크들의 상대 위치가 고정되어 있어서 데이타 갱신은 청크마다 여유 공간을 두어 해결하며, 여유 공간이 없는 경우에는 파일이

재구성된다[5]. 그러나 기본적으로 청크된 파일은 청크들의 위치를 유지하여 데이타 검색 성능을 높이므로 데이타 갱신이 발생하지 않는 환경에 적합한 구조이다. 따라서 본 연구에서는 이러한 특징을 고려하여 읽기 위주의 분석 시스템에 적합한 인덱스 구조를 제시하고자 한다.

본 연구에서 제안하는 인덱스는 청크가 디스크에 저장되는 순서대로 값이 정해지면서 구축되어 간다. 청크를 인덱싱하기 위해 청크마다 3비트가 사용된다. 큐브가 생성될 때 오버헤드가 거의 없이 인덱스가 생성될 수 있고, 질의 처리 시에 원하는 청크들을 찾기 위해 인덱스 블록은 최대 1개만 읽으면 된다는 장점이 있다. 인덱스의 크기가 매우 작고, 인덱스 블록에 많은 수의 청크 정보가 포함되어 있어서 범위 질의를 할 때 인덱스 엑세스 회수가 크게 감소된다. 제안한 인덱스는 해쉬 기반의 인덱스인 그리드 파일과 트리 기반의 인덱스인 UB-트리와 비교되었다. 그리드 또는 트리 스타일의 인덱스가 차지하는 최소 공간과 본 연구에서 제안한 인덱스의 저장 공간 비교를 통해 인덱스의 저장 공간 효율성을 분석하였고 질의 처리할 때 청크 당 인덱스 조회 회수 등의 비교를 통해 인덱스의 시간 효율성을 분석하였다.

본 논문의 구성은 다음과 같다. 2절에서 기존의 대표적인 인덱스 구조를 소개하고, 3절에서 청크 기반 MOLAP 큐브를 위한 인덱스 구조를 제시한다. 4절에서 기존의 인덱스 구조와의 비교를 통해 제안한 인덱스의 시간 공간적 효율성을 검증하고, 5절에서 결론을 맺기로 한다.

2. 대표적인 다차원 인덱스 구조

청크 기반 MOLAP 큐브에 사용할 수 있는 다차원 인덱스 구조들과 본 연구의 기초가 되는 비트맵 인덱스 구조를 간략하게 살펴보기로 한다. 다차원 인덱스는 크게 해쉬 기반 다차원 인덱스와 트리 기반 인덱스 구조로 분류할 수 있다.

2.1 해쉬 기반 다차원 인덱스

그리드 파일[6], BANG-File(Balanced And Nested Grid)[10], MLGF[7] 등이 있다. 그리드 파일은 다차원 공간을 격자식으로 분할하여 각 격자 셀마다 인덱스를 두고, 그 인덱스가 해당 데이타 버킷을 가리키도록 하는 인덱싱 방식이다. 상수 시간에 데이타를 엑세스할 수 있는 장점이 있으나 격자 간격이 버킷 밀도와 인덱스의 성능에 큰 영향을 끼친다는 단점이 있다. 이를 보완한 방법으로 BANG-File과 MLGF를 들 수 있는데 이들은 데이타가 많은 다차원 공간의 특정 부분을 지역적으로 분할하여 인덱스 크기를 조절한다. 이러한 방식을 쓰면 데이타의 삽입과 삭제가 있는 경우 동적으로 대처할 수 있고 인덱스 크기가 데이타의 크기에 비례하도록 할 수 있으나 인덱스에 계층 구조가 형성되면서 질의 처리시에 인덱스 검색에 부가적인 시간이 들게 된다.

2.2 트리 기반 다차원 인덱스

R-트리[8]와 UB-트리[9]를 들 수 있다. R-트리는 객체 중심의 영역 분할 방식을 취하고 B-트리와 비슷한 높이 균형 트리(height balanced tree)이다. R-트리의 중간노드는 자신의 자식 노드를 포함하는 가장 작은 k-차원 사각형에 해당되고, 리프 노드는 공간 데이타 객체에 대한 인덱스 엔트리를 저장한다. R-트리는 적은 수의 노드들만을 방문하여 원하는 데이타를 검색할 수 있도록 설계되었으나, 사각형간에 겹쳐진(overlap) 곳의 데이타를 검색하기 위해서는 여러 노드를 검색해야 하는 문제가 있다. UB-트리는 다차원 데이타를 Z 인덱스 순서로 1차원적으로 나열한 후 그 위에 B-트리 인덱스를 구축한 것이다. B-트리를 다차원 데이타에 적용하여 다차원 범위 질의의 속도를 향상시킨 인덱스이다. 트리 구조의 인덱스는 저장 공간의 낭비를 줄일 수 있으나 해당 데이타를 찾는데 트리의 높이에 해당하는 인덱스 블록들을 읽어야 한다는 단점이 있다.

2.3 비트맵 인덱스

비트맵 인덱스[11, 12, 13, 14]는 해당 데이타의 값이 특정 값이면 1, 그렇지 않으면 0으로 표현한 후에 데이타의 순서와 같은 순서로 그 비트들을 저장하는 인덱스 구조이다. 따라서 특정 값을 갖는 데이타를 찾으려면 비트맵 전체를 스캔하면서 해당 데이타의 위치를 찾아야 한다. 많은 수의 비트가 디스크 한 블록에 저장될 수 있어서 인덱스가 차지하는 공간이 작으나 데이타의 값마다 하나의 비트 시퀀스가 형성되므로 카디널리티가 작은 경우에 사용이 가능하다. 또한 다차원 데이타의 경우, 각 차원마다 비트 시퀀스가 독립적으로 생성되므로 다차원적 범위 질의를 하는 경우에는 해당 차원의 비트 시퀀스들에 비트 연산을 적용하여 원하는 데이타를 찾아야 한다.

3. 청크 비트맵 인덱스

우선 청크 기반 MOLAP 큐브의 구조를 자세히 소개한 후에 본 연구에서 제안하는 청크 비트맵 인덱스(CBM 인덱스, Chunk BitMap Index) 구조를 설명하기로 한다. 그리고 특정 청크를 검색하는 알고리즘과 각 차원마다 범위가 주어졌을 때 그 범위에 속하는 모든 청크들을 검색하는 범위 검색 알고리즘을 소개한다.

3.1 청크 기반 MOLAP 큐브 저장 구조

청크 기반 MOLAP 큐브 저장 구조는 다음과 같다.

우선 큐브 내의 각 집계 테이블은 그림 2와 같이 독립적인 배열에 저장된다. 그림에는 A B, C 차원으로 구성된 3차원 배열을 저장한 모습이 보인다. 배열은 모든 차원에 공평한 기회를 부여하기 위해 작은 청크들로 나뉘어져서 저장된다. 청크의 크기는 디스크 데이타 블록 크기 정도로 하여, 청크 1개를 메모리로 읽어들일 때, 디스크 블록 1개가 엑세스되도록 한다. 일반적으로 큐브는 그림과 같이 밀집 청크(dense chunk, 진한 색 청크)들과 희박 청크(sparse chunk, 연한색 청크)들로 구성된다. [2]는 청크에 유효 셀이 40% 이상 들어 있는 경우를 밀집 청크로 분류한다. 밀집 청크를 파일에 저장할 때는 청크를 배열 형태 그대로 저장한다. 희박 청크는 유효 셀만을 골라서 그 셀의 청크 안에서의 주소(*offset*)와 셀 값(value) 쌍으로 만들어서 저장한다. 희박 청크의 경우는 청크의 앞부분에 청크 번호와 청크의 크기가 저장된다.

청크들의 저장 순서가 OLAP 주요 연산인 슬라이스, 다이스, 범위 질의 등의 속도에 영향을 미친다는 연구 결과가 [15]에 나와 있다. 청크들을 저장하는 대표적인 순서로는 행 우선 순서(row-major order, linear order), Hilbert 인덱스 순서(Hilbert order)와 Z 인덱스 순서(Z-index order, shuffled-row-major order)를 들 수 있다. 행 우선 순서로 청크들을 저장하면 범위 질의 처리 시에 특정 차원에 유리하다. Z 인덱스 순서 또는 Hilbert 인덱스 순서로 청크들을 저장하면 데이타 클러스터링 효과에 의해 모든 차원에 공평한 질의 처리 속도가 보장된다. 청크들은 또한 밀도에 따라 분리되어 저장될 수 있다. [15]에서는 청크들을 밀집 청크 그룹과 희박 청크 그룹으로 나누어 따로 저장하며, 각 그룹 안에서는 청크들을 Z 인덱스 순서로 저장하는 것이 범위 질의 성능을 높인다는 것을 보였다.

이와 같이 청크들은 다양한 순서로 저장될 수 있다. 그러나 어떤 경우라고 해도 청크의 상대적 위치는 상수 시간 계산을 통해 찾을 수 있도록 하는 것이 바람직하다. 그렇지 않다면 위치 정보를 통해 데이타를 신속하게 엑세스할 수 있다는 MOLAP의 장점이 크게 감소하기 때문이다. 본 연구에서 제안하는 CBM 인덱스에는 청크들이 저장되는 순서와 동일한 순서로 해당 청크를 나타내는 비트들이 저장된다. 본 논문에서는 편의상 그림 3과 같은 저장 구조를 가정하고 인덱스 구조를 설명하기로 한다. 즉, 청크들은 밀집 청크 그룹과 희박 청크 그룹으로 나뉘고 각 청크 그룹 안에서 청크들은 Z 인덱스 순서로 저장된다.

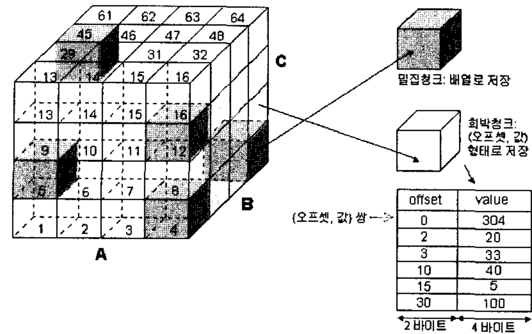
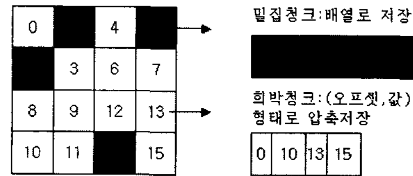


그림 2 3차원 MOLAP 큐브의 청크 기반 저장 방식



(a) 청크로 나눈 데이터 큐브 (b) 청크가 저장된 파일

그림 3 2차원 MOLAP 큐브의 청크 기반 저장 방식 (밀집 청크, 희박 청크 분리 저장)

3.2 CBM 인덱스 구조

CBM 인덱스를 구축하는 목표는 데이타 큐브가 생성될 때 희박 청크들이 압축되고 널(null) 청크들이 제거되면서 손실된 청크의 위치 정보를 인덱스 수준에서 복원하여 질의 처리 시에 청크 당 인덱스 블록 검색 회수를 최대 1번으로 제한하려는 것이다. 또한 하나의 인덱스 블록에 많은 인접한 청크들의 위치 정보를 포함시켜 범위 질의 처리 시에 인덱스 블록 검색 오버헤드를 최소화하려고 한다. CBM 인덱스는 데이타 큐브가 구축되는 과정에서 오버헤드가 거의 없이 구축될 수 있다는 장점이 있다.

CBM 인덱스의 골격을 그림 4의 예제를 통해 설명하기로 한다. 그림 4(a)에 청크로 나누어진 2차원 큐브가 있다. 청크 0, 3, 8과 같은 검은 색 청크들은 밀집 청크이고, 청크 1, 4, 6과 같은 회색 청크들은 희박 청크이며 청크 2, 5, 7과 같은 흰색 청크는 널 청크들이다. 모든 청크들은 그림 4(b)와 같이 가상적으로 Z 인덱스 순서로 번호가 매겨지고 그림 2와 같은 방식으로 밀집 청크 그룹과 희박 청크 그룹으로 나뉘어 저장된다.

이와 같은 큐브를 위한 CBM 인덱스 구조는 그림 4(c)와 4(d)에 나타나 있다. 큐브의 청크들이 디스크에 저장되는 과정에서 그림 4(c)와 같은 비트 시퀀스 B_0,

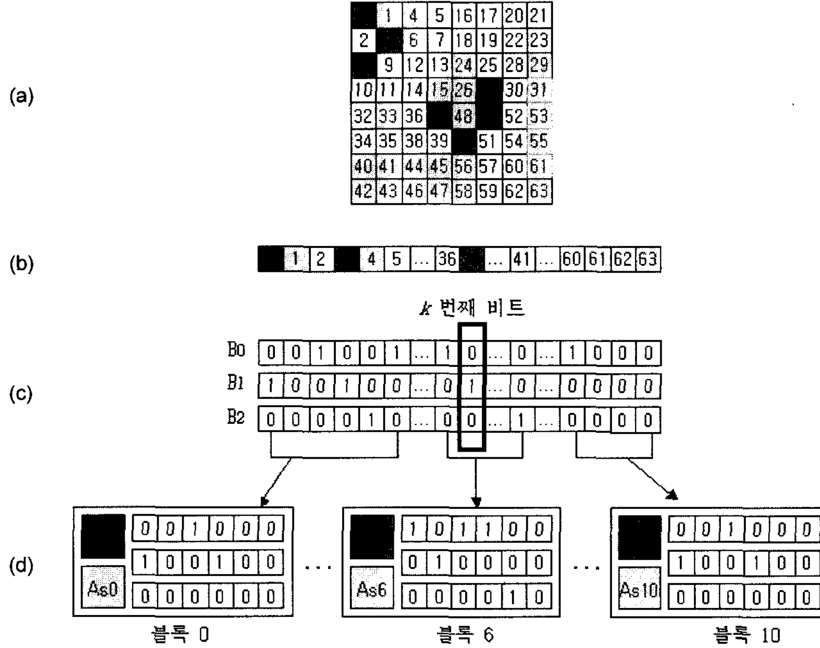


그림 4 CBM 인덱스 구조: (a) 청크로 분할된 2차원 큐브, (b) Z 인덱스 순서로 나열된 청크들, (c) CBM 인덱스의 기본 골격인 비트 시퀀스, (d) CBM 인덱스 블록들

B_1, B_2가 함께 만들어진다. 청크 k가 디스크에 저장될 때 그 청크에 대한 인덱스도 함께 만들어지는 것이다. 청크 k의 인덱스는 3비트로 구성되며, 이는 그림 4(c)의 B_0, B_1, B_2의 k번째 비트들이다. 이 비트들에 관해서는 나중에 설명하기로 한다. B_0, B_1, B_2의 시퀀스는 디스크 한 블록 단위로 나뉘어져 그림 4(d)와 같이 저장된다. 그림 4(d)의 블록들은 인덱스 블록이라 부른다. 그림에서는 인덱스 블록에 6개의 청크 정보가 포함된 것으로 가정되어 있다. 모든 인덱스 블록 i의 앞부분에는 그 블록에 포함된 첫 밀집 청크의 주소(AD_i)와 첫 희박 청크의 주소(AS_i)가 함께 저장된다.

이제 각 인덱스 블록의 내용에 관해 살펴보기로 한다. CBM 인덱스 블록은 그림 5와 같은 구조를 갖는다. 인덱스 블록 i에는 두 가지 정보가 포함되어 있다. 첫째는 인덱스 블록 i에 포함된 밀집 청크들의 시작 주소(AD_i)와 희박 청크들의 시작 주소(AS_i)가 저장된 인덱스 엔트리이다. 둘째는 세 그룹의 비트 시퀀스 B_0, B_1, B_2이다. 각 비트 그룹의 k번째 비트는 이 인덱스 블록에 포함된 청크들 중에서 k번째 청크에 관한 정보이다. 이들을 각각 $B_0(i, k)$, $B_1(i, k)$, $B_2(i, k)$라고 부르기로 한다. 이들은 각각 아래의 정보를 담고 있다.

$B_0(i, k)$: 인덱스 블록 i의 k번째 청크가 널 청크인가를 나타낸다.

$B_1(i, k)$: 인덱스 블록 i의 k번째 청크가 밀집 청크인지 희박 청크인지를 구분한다.

$B_2(i, k)$: 인덱스 블록 i의 k번째 청크가 희박 청크인 경우, k번째 청크가 바로 앞에 있는 희박 청크와 동일한 디스크 블록에 있는가를 나타낸다. $B_2(i, k)$의 값은 1로써, 희박 청크 시작 주소를 나타낸다.

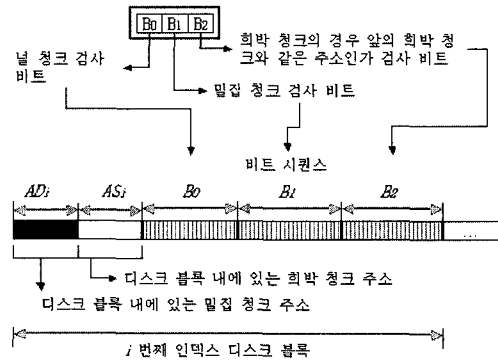


그림 5 CBM 인덱스 블록의 구조

3.3 특정 청크 검색 알고리즘

청크 번호가 주어졌을 때 그 청크를 엑세스하는 방법을 설명하기로 한다. 예를 들어 3차원 큐브의 한 청크를 검색하기로 하자. 그 청크의 3차원 공간 상의 좌표를 d_1, d_2, d_3라고 하고 큐브는 3(2)절에서 가정한대로 저장되어 있다고 하자. 우선 청크 좌표인 d_1, d_2, d_3로부터 해당 Z 인덱스를 구한다. 이는 d_1, d_2, d_3의 비트들을 순서대로 병합하여 쉽게 얻을 수 있다 [15]. 구해진 Z인덱스를 x라고 하고, 인덱스 블록에 포함될 수 있는 청크 개수를 c라고 하자. 검색하고자 하는 청크 정보가 포함된 인덱스 블록은 다음과 같이 찾는다.

인덱스 블록 번호 = i = x div c

인덱스 블록 내의 청크 위치 = j = x mod c

인덱스 블록 i를 메모리로 읽은 후에 다음과 같은 연산을 통해 해당 청크의 주소를 찾는다. (i, j)우선 $B_0(i, j)$를 검사하여 청크 x가 널 청크인가를 확인한다. 널 청크가 아닌 경우는 $B_1(i, j)$를 검사하여 청크 x가 밀집 청크인가를 확인한다. 밀집 청크이면 인덱스 블록 i안에서 청크 x를 선행하는 모든 밀집 청크의 개수를 구한다. 청크 개수(C_j)는 $\sum_{k=1}^{j} B_1(i,k)$으로 계산된다. 밀집 청크들은 크기가 일정하기 때문에 C_j값과 인덱스 블록 i에 속한 첫 밀집 청크의 주소인 AD_i를 사용하면 청크 x의 디스크 주소를 쉽게 구할 수 있다. C_j 값은 매핑 테이블 한 개를 메모리에 상주시켜 신속하게 구할 수 있으며 이는 나중에 설명하기로 한다.

청크 x가 희박 청크인 경우는 인덱스 블록 i에 포함된 희박 청크들의 시작 주소인 AS_i와 희박 청크들에 관한 정보가 담긴 $B_2(i,1) \sim B_2(i,j)$값을 사용하여 청크 x의 주소를 찾는다. $B_2(i,j)$가 0이면 이는 청크 x가 바로 앞의 청크와 동일한 디스크 블록에 저장되어 있다는 것을 나타내므로 $\sum_{k=1}^{j} B_2(i,k)$는 청크 x가 AS_i 디스크 블록으로부터 몇 개의 블록이 떨어진 곳에 저장되어 있는가를 나타낸다. 청크 x는 해당 디스크 블록을 메모리에 읽어 들인 후에 그 블록에 속한 각 청크를 스캔하면서 쉽게 찾을 수 있다.

$\sum_{k=1}^{j} B_1(i,k)$과 $\sum_{k=1}^{j} B_2(i,k)$의 계산은 j개의 비트 중에서 1로 세팅된 것의 개수를 세는 연산이다. 인덱스 한 블록의 크기를 4K 바이트로 가정한다면 이는 최대 32K개가 될 수 있다. 이 계산의 속도를 향상시키기 위한 한 방법은 메모리에 매핑 테이블을 상주시켜 사용하는 것이다. 예를 들어, 16 비트 (2 바이트) 길이의 시퀀스에 들어 있는 1의 개수를 세기 위해서는 16비트 길이의 시퀀스의 모든 조합에 들어 있는 1의 개수를 세어 1차원 배열에 저장해 둔다. 16비트 길이의 시퀀스 값을 정수로 변환하여 해쉬 키이로 사용하면 해당 1의 개수를 배열에서 상수 시간에 찾을 수 있다. 이와 같은 방법을 사용하면 인덱스 블록의 검색 시간을 줄일 수 있다. 이 계산에 걸리는 시간은 디스크를 여러 번 엑세스하면서 블록들을 읽어들이는 시간과 비교할 때 무시할 수 있는 정도이다. 주어진 청크 번호로부터 청크가 저장된 디스크 블록 주소를 찾는 알고리즘은 다음과 같다.

[특정 청크 검색 알고리즘]

가정: 디스크 한 블록을 M 바이트, 인덱스 블록 안의 청크 개수를 c 라 하자.

입력: 검색할 데이타가 속한 청크 좌표 $(d_1, d_2, d_3, \ldots, d_n)$를 입력 받는다.

출력: 청크가 포함된 데이타 블록 주소 C_{addr}.

알고리즘:

단계 1. 청크 좌표 $(d_1, d_2, d_3, \ldots, d_n)$로 Z 인덱스 x를 구한다.

단계 2. 인덱스 블록 번호 i = x div c를 계산한다.

단계 3. 인덱스 블록 내의 청크 위치 j= x mod c를 계산한다.

단계 4. i 번째 인덱스 블록을 읽어온다.

단계 5. $B_0(i,j)$= 1이면 (1), $B_1(i,j)$ = 1이면 (2), 그 외에는 (3)을 실행하고 종료한다.

(1) C_{addr} = NULL.

(2) $C_{addr} = AD_i + M \times (\sum_{k=1}^{j} B_1(i,k) - 1)$

(3) $C_{addr} = AS_i + M \times (\sum_{k=1}^{j} B_2(i,k)' - 1)$

3.4 범위 질의 처리 알고리즘

큐브의 각 차원마다 범위를 두어 데이타의 모든 차원값이 그 안에 속하는 경우, 그러한 데이타를 검색하는 것을 범위 질의 처리라고 한다. 예를 들어 청크 $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$가 범위 질의 구간에 포함된 청크들이라고 하자. MOLAP 큐브는 개념적으로 배열 구조이므로 어떤 청크들이 범위 질의 구간에 속하는가는 간단한 계산을 통해 알 수 있다. 여기서 $i_1 \le i_2 \le \cdots \le i_k$ 이라고 하자.

범위 질의는 다음과 같이 처리된다. 검색할 모든 청크들의 번호를 Z 인덱스 번호로 변환하고나서 변환된 청크 번호들을 오름차순으로 정렬한다. 정렬한 후의 청크들의 Z 인덱스 번호를 $cz_{i_1}, cz_{i_2}, \cdots, cz_{i_k}$라고 하자. 즉

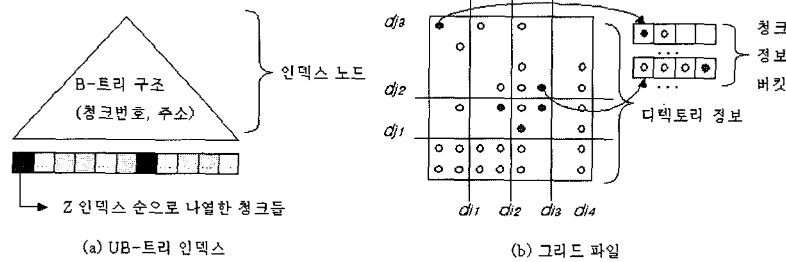


그림 6 청크 기반 MOLAP 큐브에 적용한 다차원 인덱스.

$cz_{i_1} \le cz_{i_2} \le \cdots \le cz_{i_k}$이다. 먼저 cz_{i_1}이 속한 인덱스 블록을 메모리로 읽어 들인다. 이 인덱스 블록에 포함되어 있으면서 범위 질의 구간 안에 있는 모든 청크들을 검색한다. $cz_{i_1}, \cdots, cz_{i_j}$들이 검색되었다고 하자. 다음에는 정렬된 청크번호들 중에서 cz_{i_j}의 바로 다음 청크인 $cz_{i_{j+1}}$의 인덱스 블록을 메모리로 읽어 들인다. 역시 이 인덱스 블록에 포함되어 있으면서 범위 질의 구간 안에 있는 모든 청크들을 검색한다. $cz_{i_1}, cz_{i_2}, \cdots, cz_{i_k}$ 청크들이 모두 검색될 때까지 이와 같은 작업을 반복한다. 범위 질의 처리 알고리즘은 다음과 같다.

[범위 질의 처리 알고리즘]

가정: 디스크 한 블록을 M 바이트, 인덱스 블록 안의 청크 개수를 c라 하자.

입력: 범위 안에 있는 청크 $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$를 입력 받는다.

출력: 청크들의 주소 $Array(C_{addr})$

알고리즘:

단계 1. $c_{i_1}, c_{i_2}, \cdots, c_{i_k}$를 Z 인덱스로 변환한 후 정렬한다. 정렬된 시퀀스를 $cz_{i_1}, cz_{i_2}, \cdots, cz_{i_k}$라고 하자.

단계 2. 첫 번째 청크를 $cz = cz_{i_1}$에 할당한다.

단계 3. 인덱스 블록 번호 $i = cz$ div c을 계산한다.

단계 4. i번째 인덱스 블록을 읽어온다.

(1) cz가 속한 청크 주소를 구하여 $Array(C_{addr})$에 추가한다. (특정 청크 검색 알고리즘 참조)

(2) 다음 청크를 cz에 할당한다.

(3) 인덱스 블록 번호 $j = cz$ div c를 계산한다.

(4) $i = j$이면 단계 4(1)을 실행하고 아니면 i에 j를 할당하고 단계 4를 실행한다.

단계 5. $Array(C_{addr})$를 리턴하고 종료한다.

4. 성능 분석

이제 CBM 인덱스의 저장 공간 효율성과 검색 시간 효율성에 관해 분석해 보기로 한다. 분석은 CBM 인덱스와 대표적인 트리 구조의 다차원 인덱스인 UB-트리와 해쉬 구조의 그리드 파일과의 비교를 통해 수행하였다. UB-트리와 그리드 파일은 데이타의 삽입, 삭제를 허용하는 인덱스이므로 이를 효율적으로 처리하기 위한 여유 공간이 포함된 구조이다. 그러나 본 연구에서는 UB-트리와 그리드 파일의 저장 공간을 최소화하였을 경우 공간 효율성과 검색 시간 효율성을 비교하기 위해 데이타 갱신을 위한 여유 공간은 포함하지 않았다. 데이타 삽입, 삭제와 같은 점진적 갱신이 발생할 경우에 대한 분석은 뒤에 설명하기로 한다. 그리고 다른 다차원 인덱스와의 비교 분석에 관해서도 나중에 설명하기로 한다.

분석에 사용하기 위해 UB-트리와 그리드 파일의 저장 공간은 다음과 같이 가정하였다. UB-트리의 기본 구조는 그림 6(a)와 같이 데이타를 Z 인덱스 순서로 나열한 후에 이들을 디스크 블록 크기로 나누어 저장하고, 각 데이타 블록에 대해 B-트리와 같은 형태로 인덱스를 만든 것이다. UB-트리를 청크 기반 MOLAP 큐브 인덱싱에 사용하기 위해서는 청크들을 Z 인덱스 순서로 나열하고 이들을 디스크 블록 크기로 잘라서 저장하고, 각 디스크 블록에 대해 B-트리를 만든다고 가정하였다. 삽입과 삭제를 허용하는 것이 아니고 청크들이 생성되는 순서로 UB-트리의 리프 노드들도 함께 생성되는 것이므로 UB-트리 노드 안에는 여유 공간을 전혀 두지 않았다. 또한 널 청크들은 저장되지 않는다고 가정하였다. 본 연구에서 제안한 CBM 인덱스는 밀집 청크들과 희박 청크들을 분리하여 저장하는 것을 가정하였으나 CBM 인덱스를 사용하기 위해 반드시 청크들을 밀도에 따라 분리 저장할 필요는 없다. UB-트리의 경우에는 밀집 청크들과 희박 청크들을 분리하는 경우나 그렇지 않은 경우에 저장 공간 상에 전혀 차이를 보이지 않으므로 여기서는 모든 청크들을 Z 인덱스 순서로 저장하는 것으로 분석을 단순화 시켰다.

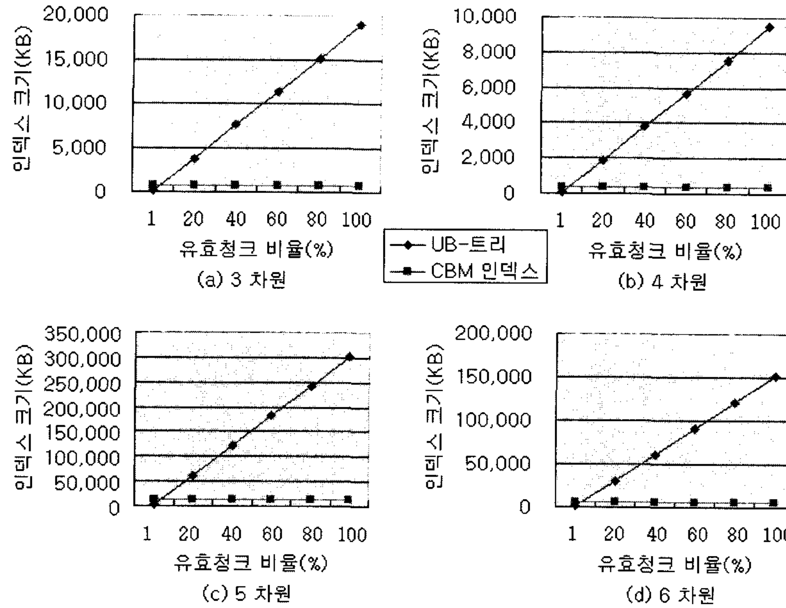


그림 7 CBM 인덱스와 UB-트리의 저장공간 비교

그리드 파일은 그림 6(b)와 같은 구조를 갖는다. 다차원 공간을 격자로 나누고 각 격자 셀은 그 셀이 나타내는 범위에 속하는 데이타가 저장되어 있는 버킷을 가리킨다. 이 구조에서는 격자 셀이 하나의 디스크 블록을 가리키도록 격자 크기를 잘 조정해서 정의할 필요가 있다. 청크 기반 MOALP 큐브의 청크들을 인덱싱하기 위해서 본 연구에서는 데이타가 균일하게 분포되어 모든 격자 셀 안에는 동일한 개수의 청크가 포함되어 있는 것으로 가정하였다. 즉 각 격자 셀은 청크 정보(청크 번호, 밀도 상태 주소)가 들어 있는 디스크 블록 1개를 가리키도록 하였으며, 실제 데이타의 크기에 비해 상당히 크기가 작으므로 디렉토리는 메모리에 상주하는 것으로 가정하였다. 즉 본 연구에서는 그리드 파일을 사용하여 청크들을 인덱스하는데 필요한 최소한의 저장 공간만을 가정하였다.

4.1 저장 공간 효율성에 관한 분석

4.1.1 CBM 인덱스, UB-트리, 그리드 파일의 저장 공간 비교

CBM 인덱스의 경우에는 모든 청크에 대해 3비트의 정보를 저장한다. 또한 인덱스 블록마다 첫 밀집 청크의 주소와 첫 희박 청크의 주소를 저장한다. 주소는 각각 4 바이트씩 차지한다고 가정하였다. UB-트리는 리프 노드와 인덱스 노드로 나뉜다. 리프 노드에는 유효 청크(널 청크들을 제외한 밀집 또는 희박 청크를 뜻함)들이 Z 인데스 순서로 저장되는데, 각 청크마다 청크 번호, 청크 밀도 상태(밀집, 희박, 널 청크를 구분), 청크 주소를 저장하기 위해 각각 4 바이트, 1 바이트, 4 바이트가 필요하다. 그리드 파일의 경우에는 디렉토리 정보가 메모리에 상주한다고 가정할 때 유효 청크 정보만 저장하는 버킷들만이 디스크에 저장되므로 이는 UB-트리의 리프 노드가 차지하는 공간만이 필요할 뿐이다. 삽입, 삭제를 허용하지 않으므로 여유 공간을 없애고 각 격자 셀에 포함되는 청크 개수를 동일하게 하여 인덱스 공간을 최소화했기 때문이다.

분석할 데이타는 표 1과 같이 준비하였다. 일반적으로 MOLAP 데이타는 10차원 이하이므로[16] 본 연구에서는 3차원~6차원 큐브를 사용하였다. 그 이상의 차원에서도 유사한 실험결과를 나타내며 이에 대해서는 나중에 설명하기로 한다. 청크의 크기는 데이타가 100% 밀집한 경우 4K 바이트 크기의 디스크 블록에 가장 근접

표 1 분석 데이타 세트

큐브 차원 수	차원별 멤버 개수	큐브 셀 개수 (100% 밀도)	큐브의 청크 개수
3차원	1280	2 Giga	128^3 (= 2 Mega)
4차원	160	655 Mega	32^4 (= 1 Mega)
5차원	128	34 Giga	32^5 (= 33 Mega)
6차원	48	12 Giga	16^6 (= 17 Mega)

한 크기가 되도록 하였다. 데이타가 100% 밀집한 경우 총 청크의 개수는 표 1과 같다.

CBM 인덱스는 유효 청크 개수에 관계없이 크기가 고정되어 있으나 UB-트리와 그리드 파일의 크기는 유효 청크 개수에 의존적이다. 따라서 본 연구에서는 유효 청크의 비율을 변화시켜 가면서 각 인덱스의 저장 공간을 비교하였다. UB-트리의 경우 리프 노드들을 제외한 상위 레벨의 노드들이 차지하는 공간이 상대적으로 매우 작으므로 분석에서는 UB-트리와 그리드 파일이 차지하는 공간이 거의 비슷하였으며 그림 7과 그림 8에서와 같이 이러한 인덱스의 크기는 유효 청크 개수에 정비례하는 것을 알 수 있다. 반면에 CBM의 경우는 큐브 밀도와 무관하게 크기가 고정적이나 크기가 작아서 유효 청크 비율이 4% 정도일 때 인덱스가 차지하는 공간이 UB-트리나 그리드 파일과 같다는 것을 알 수 있다. 참고로, 청크 밀도는 대체로 낮기 때문에 실제 큐브의 데이타 밀도는 4%보다 훨씬 낮다. 좀 더 자세히 관찰해 보면, 유효 청크 비율이 10%일 때 CBM 인덱스는 UB-트리나 그리드 파일에 비해 약 0.5배의 저장 공간을 사용하고, 유효 청크 비율이 100%가 되면 CBM 인덱스는 다른 인덱스들에 비해 0.05배의 저장 공간만을 사용한다.

지금까지의 실험은 UB-트리와 그리드 파일의 저장 공간을 최소화하기 위해 데이타 갱신이 발생하지 않는다고 가정하였다. 그러나 점진적 갱신을 허용한다면 큐브의 모든 청크에 여유 공간을 두어야 하므로 널청크가 없어지고 모든 청크는 유효하게 된다. 이는 그림 7, 8에서 유효 청크 비율이 100%인 경우에 해당된다. 즉, CBM 인덱스는 유효 청크 비율에 상관없이 크기가 일정하므로 점진적 갱신이 허용되는 경우 공간 효율성이 더욱 높다.

본 연구의 실험에서 인덱스 저장 공간의 크기를 비교하기 위해 3차원~6차원을 실험 차원으로 설정하였으나 그림 7, 8의 그래프에서 살펴봤듯이 인덱스 저장 공간의 크기는 차원에 상관없이 유효 청크의 비율에 따라 그 크기가 결정된다. 따라서 7차원 이상의 고차원에서도 유효 청크의 비율에 따라 동일한 인덱스 저장 공간 양상이 나타나게 된다.

4.1.2 데이타 큐브와 CBM 인덱스의 저장 공간 비교

청크 기반의 데이타 큐브 크기와 이를 인덱싱하는 CBM 인덱스의 저장 공간 크기를 비교함으로써 인덱스가 상대적으로 차지하는 공간의 비율을 알아보기로 한다. 분석에는 표 1의 3차원 데이타를 사용하였으나 4차원, 5차원, 6차원 데이타의 경우에도 유사한 비율을 나타내었다. CBM 인덱스는 큐브 데이타의 밀도와 무관하게 차지하는 공간이 동일하나 큐브 자체는 밀도에 따라

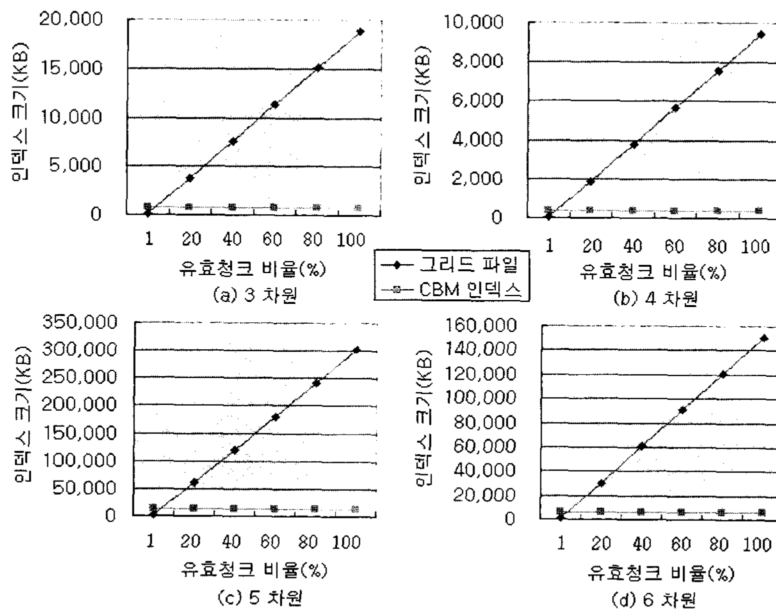


그림 8 CBM 인덱스와 그리드 파일의 저장공간 비교.

저장 공간의 양이 변화하므로, 분석에는 데이타의 밀도를 1%~100%를 10% 간격으로 변화시켰다. 분석 결과는 그림 9에 있다.

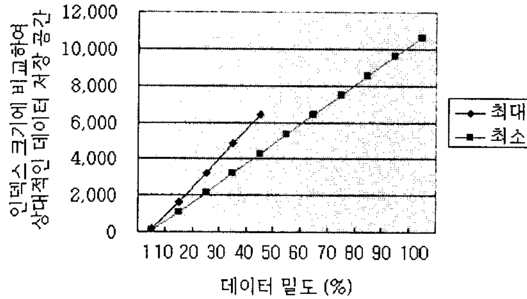


그림 9 3차원 데이타 큐브에서 데이타 밀도에 따른 CBM 인덱스에 비교하여 상대적인 데이타 저장 공간의 비율

큐브의 모든 청크들이 100% 밀집한 경우에 큐브를 저장하는 공간이 최소가 된다. 그림 9에서 데이타가 10% 일 때 '최소'가 1/1,065배가 된다는 뜻은 큐브는 데이타 밀도가 10%이지만 이 10%의 데이타가 모두 100% 밀도의 밀집 청크들에 모여 있다고 가정한 것이다. 즉, 데이타 큐브를 저장하는데 드는 공간이 최소가 된다는 것을 뜻한다. 그림 9에서와 같이 데이타가 1%인 경우 CBM 인덱스는 데이타 저장 공간 크기의 약 1/106의 크기를 보이고, 데이타 밀도가 10배씩 증가함에 따라 CBM인덱스가 차지하는 공간은 1/10씩 감소하여 데이타 밀도가 100%가 되면 CBM 인덱스가 차지하는 공간은 약 데이타 공간에 비하여 1/10,646의 저장 공간이 필요하다.

동일한 데이타 밀도에 대해 큐브가 차지하는 공간이 최대가 되는 것은 데이타가 모두 균일하게 큐브내에 퍼져 있어서 널 청크가 없고 모두 희박한 청크들만 있는 경우이다. 희박 청크의 경우는 하나의 데이타 셀을 저장하는데 그 셀의 청크 내의 오프셋과 데이타의 값이 저장되므로 밀집 청크의 경우와 비교할 때 1.5배의 저장 공간을 필요로 하며, 각 희박 청크는 자신의 청크 주소와 청크의 셀 개수를 나타내는 6바이트가 함께 저장된다. 그림 9의 '최대'를 나타내는 선이 바로 이러한 경우의 CBM 인덱스의 상대적 저장 공간을 나타낸다. 데이타가 균일하게 분포되어 있으면서 밀도가 40%가 넘게 되면 모두 밀집 청크가 되므로, 그래프는 데이타 밀도가 40% 인 경우까지만 계산하여 만들었다. 데이타 밀도가 10%인 경우 CBM 인덱스는 데이타 공간에 비해 1/1,613배 크기이고, 데이타의 밀도가 10%씩 증가함에 따라 1/2배씩 저장 공간이 감소한다. 데이타 밀도가 40%가 되면 대략 1/6,403배의 저장 공간만을 차지한다. 대표적인 MOLAP 제품으로 가장 효율적인 인덱스를 자랑하는 Essbase사의 경우 인덱스 크기가 전체 데이타의 평균 2%[17]라는 것을 볼 때 본 연구에서 제안한 CBM 인덱스는 크기가 상당히 작다는 것을 알 수 있다.

4.2 인덱스 검색 시간 효율성에 관한 분석

4.2.1. 특정 청크 검색 성능 분석

데이타 셀 1개를 검색하기 위해 필요한 인덱스 검색 오버헤드를 살펴보기로 한다. 해당 데이타가 있는 청크 번호가 주어지면 CBM 인덱스의 경우는 청크 번호로부터 그 청크의 위치 정보를 담고 있는 인덱스 블록의 주소를 쉽게 계산할 수 있고 (특정 청크 검색 알고리즘의 단계 1, 2에 해당), 그 인덱스 블록을 검색하면 청크가 있는 디스크 블록의 주소를 찾을 수 있다. 즉 모든 경우에 인덱스 블록은 1번만 읽게 된다.

그리드 파일의 경우는 디렉토리 엑세스 1번, 청크 주소가 들어 있는 버킷 1번을 엑세스해야 해당 청크를 검색할 수 있다. 그러나 디렉토리는 일반적으로 크기가 작아서 메모리에 상주 시키는 경우가 많아 실제 상황에서는 청크 주소 버킷만 1회 엑세스하면 된다.

B-트리 구조의 UB-트리의 경우는 리프 노드를 엑세스하기까지 $O(\log_m n)$개의 트리 노드들을 엑세스해야 한다. 여기서 m은 한 노드에 표현될 인덱스 키의 개수, n은 전체 인덱스 키의 개수를 뜻한다. 그러나 표 1의 데이타의 경우에는 트리의 높이가 리프 레벨을 포함하여 3이고, 리프 노드들을 제외한 다른 노드들이 차지하는 공간이 작아서 역시 이 노드들을 메모리에 상주시켜서 인덱스 노드의 엑세스를 1회만 하도록 할 수 있다. 데이타가 큰 경우는 인덱스 블록을 2회 이상 엑세스하게 된다.

본 연구에서 분석에 사용한 표 1의 데이타의 경우에 CBM 인덱스가 차지하는 공간은 3차원 데이타의 경우 788K 바이트, 4차원 데이타의 경우 384K 바이트, 5차원 데이타의 경우에 약 12M 바이트가 된다. 따라서 데이타가 이와 같이 작은 경우에는 CBM 인덱스 전체를 메모리에 로드시켜 놓고 데이타 큐브를 부가적인 인덱스 블록 엑세스 없이 검색할 수도 있다는 것을 참고로 하자.

4.2.2 범위 질의 처리 성능 분석

CBM 인덱스는 범위 질의 처리에 높은 성능을 보인다. 범위 질의 처리시에 엑세스해야 하는 인덱스 블록의 개수를 UB-트리, 그리드 파일과 비교해 보기로 한다. 청크들로 구성된 다차원 공간 U 상에서 각 차원에 범위를 주어 만든 Q를 범위 질의 구간 (또는 그 구간에

속한 총 청크 개수)이라고 하자. Q_1은 그 구간 안에 속한 유효 청크 개수라고 하자.

UB-트리의 경우 읽기 연산만을 허용하여 인덱스가 압축되어 있을 때, Q에 속한 청크들을 찾기 위해 검색할 인덱스 블록의 수는 평균적으로 $\lceil 2 \times Q_1/M_1 \times O(\log_m n) \rceil$을 넘지 않는다는 연구 결과가 [9]에 나와 있다. 여기서 M_1은 하나의 리프 노드에 포함된 청크 개수이고, m은 UB-트리 내부 노드에 포함된 청크들의 개수이며, n은 UB-트리의 리프 노드 개수이다. $\lceil 2 \times Q_1/M_1 \rceil$ 회의 리프 노드 검색을 해야 한다는 것이다. 본 연구에서 실험한 데이타를 예로 든다면, 리프 노드에 포함되는 청크 개수인 M_1은 500이고, Q안에 널 청크가 없는 경우, 즉 $Q = Q_1 = 3,000$일 때 12회의 리프 노드 검색을 해야 한다. 각 리프 노드 검색에는 $\lceil O(\log_m n) \rceil$ 회의 UB-트리 내부 노드 검색이 포함된다.

그리드 파일을 사용하는 경우에는 $\lceil 2 \times Q_2/M_2 \rceil$ 회의 인덱스 검색이 필요하다. 여기서 Q_2는 질의 구간과 겹치는 격자 셀에 포함되는 청크 개수이고, M_2는 최상의 경우에 UB-트리의 M_1과 같으나 실제 데이타의 경우는 청크의 생성 순서, 밀도, 분포도에 따라 $M_2 \le M_1$이 된다.

CBM 인덱스의 경우는 Q에 속한 청크들을 찾기 위해 평균 $\lceil 2 \times Q/M_3 \rceil$ 회의 인덱스 블록을 검색하게 된다. 여기서 Q는 질의 구간에 속한 총 청크 개수이고, M_3은 CBM 인덱스 블록에 속한 청크의 개수를 뜻한다. 본 연구에서 실험에 사용한 데이타의 경우 M_3은 10K 개가 된다. $Q = 3,000$인 경우 1개의 인덱스 블록만을 검색하면 된다. 예를 들어, UB-트리의 높이를 2, $M_1 = M_3 = 500$, $M_2 = 10,000$, $Q = 3,000$ 이라고 하고, 큐브 내의 유효 청크의 개수를 각각 10%, 50%, 100%라고 했을 때 표 2와 같은 인덱스 블록의 검색이 필요하게 된다.

종합해 보면, 범위 질의 처리시의 인덱스 블록의 검색 회수는 하나의 인덱스 블록에 포함된 유효 청크 개수에 비례한다. CBM 인덱스 블록에는 10,000개의 청크가 포함된 반면에 UB-트리와 그리드 파일의 경우에는 500개 정도의 청크가 포함되기 때문에 위의 표의 예제와 같은 양상을 보이는 것이다. 데이타가 삽입이나 삭제와 같은 점진적 갱신이 발생하는 경우를 위해 여유 공간을 갖게 되는 경우 유효 청크 비율이 100%에 해당되는 것이므로 CBM 인덱스가 범위 질의에 높은 성능을 나타내게 된다.

표 2 범위 질의시 인덱스 블록 검색 회수 비교

유효 청크 비율	인덱스 블록 검색 회수		
	CBM 인덱스	UB-트리	그리드 파일
100%	1	24	12
50%	1	12	6
10%	1	3	2

4.2.3 Z 인덱싱을 통한 청크 클러스터링 효과

CBM 인덱스는 청크의 저장 순서를 상수 시간 안에 계산할 수 있는 구조에 쓸 수 있는 인덱스 구조이다. 그러나, 청크를 본 연구에서 예제로 썼던 것과 같이 밀집 청크들과 희박 청크들을 구별하여 Z 인덱스 순서로 저장하면 범위 질의에 필요한 많은 청크들이 서로 인접해 있게 되고, 인덱스 블록의 정보 역시 동일한 순서를 따르므로 청크의 클러스터링 효과를 크게 볼 수 있다. 이것은 인덱스 디스크 블록을 한 번 엑세스함으로 필요한 청크의 정보들이 대부분 메모리에 존재하게 된다는 것을 의미한다.

앞의 실험 결과를 통해 기존의 다차원 인덱스들은 청크를 인덱싱하기 위해 각 청크마다 기본적인 정보로 청크 번호, 청크 밀도 상태, 청크 주소를 저장해야 한다는 것을 알 수 있다. 이와 같은 기본적인 청크 정보를 저장하기 위한 공간이 인덱스 저장 공간의 대부분을 차지한다. 그러므로 다른 다차원 인덱스들도 UB-트리나 그리드 파일과 비슷한 정도의 저장 공간을 필요로 한다. 또한 하나의 인덱스 디스크 블록에도 비슷한 개수의 청크가 포함되어 실험 결과와 유사한 질의 처리 속도를 보이게 된다.

5. 결 론

청크 기반의 MOLAP 큐브 저장 기법은 다차원 배열을 각 변의 길이가 같은 작은 청크들로 나누어 저장하는 기법으로써, 각 차원에 공평한 질의 처리 성능을 보장하는 장점을 갖는다. 그러나 실생활의 데이타는 대체로 희박하기 때문에, 저장 공간의 효율을 높이기 위해 희박한 청크들을 압축하여 저장하게 되며, 이로 인해 청크들의 크기가 변하고 상대 위치 정보를 잃게 되어 인덱스가 필요하게 된다.

본 연구에서는 청크 기반의 MOLAP 큐브를 위해 비트맵 스타일의 인덱스를 제시하였다. 각 청크마다 3 비트를 할당하여 인덱스를 구축하기 때문에 공간 효율성이 높고, 각 인덱스 블록마다 그 블록에 속한 밀집 청크와 희박 청크의 시작 주소를 두어 특정 청크를 검색할 때 비트맵 전체를 스캔하는 대신 인덱스 블록 1개만을

검색하도록 하였다.

하나의 인덱스 블록에 10K 개의 청크 위치 정보를 포함시켜 범위 질의 처리 시에 인덱스 블록의 검색 회수를 최소화하였다. UB-트리, 그리드 파일과 비교해 본 결과 범위 질의 처리 시에 읽어야 하는 인덱스 블록의 개수는 각 인덱스의 크기에 비례한다는 것을 보였다. 큐브의 유효 청크의 비율이 10% 정도이면 CBM 인덱스가 UB-트리나 그리드 파일에 비해 크기가 1/2이 되고 100%가 되면 약 1/20이 된다.

또한 CBM 인덱스는 큐브의 크기에 비해 상당히 작다. 큐브 데이타의 밀도가 1%, 10%, 100% 인 경우 인덱스의 상대적 크기는 대략 1/176, 1/1613, 1/10,646 이 된다. 본 연구는 큐브가 생성될 때 작은 크기의 인덱스를 함께 생성하여 범위 질의를 포함한 OLAP 주요 연산들의 속도를 크게 향상시키는 방안을 제시하였다.

참 고 문 헌

[1] Arbor Software Corporation, Robert J. Earle, U.S. Patent #5359724, 1994.

[2] Yihong Zhao, Prasad Deshpande, Jeffrey Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," Proc. ACM SIGMOD, pp. 159-170, 1997.

[3] S. Sarawagi, M. Stonebraker, "Efficient Organization of Large Multidimensional Arrays," Proc. 10th ICDE, Feb. 1994.

[4] Hosagrahar V. Jagadish, "Linear Clustering of Objects with Multiple Attributes," Proc. ACM SIGMOD, pp. 332-342, 1990.

[5] Prasad M. Deshpande, Karthikeyan Ramasamy, Amit Shukla, Jeffrey F. Naughton, "Caching Multidimensional Queries Using Chunks," Proc. ACM SIGMOD, pp. 259-270, 1998.

[6] Jurg. Neivergelt, Hans Hinterberger, Kenneth C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," ACM Transactions on Database Systems, Vol. 9, No. 1, pp. 38-71, March 1984.

[7] Kyu-Young Whang, Ravi Krishnamurthy, "The Multilevel Grid File-A Dynamic Hierarchical Multidimensional File Structure", DSFAA, pp. 449-459, 1991.

[8] Antonin Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD, pp. 47-57, Boston, 1984.

[9] Rudolf Bayer, "The Universal B-Tree for Multidimensional Indexing", Technische Universitat Munchen, TUM-I9637, November 1996.

[10] Michael Freeston. "The bang file: a new kind of grid file," Proc. ACM SIGMOD, pp. 260-269, 1987.

[11] Chee-Yong Chan, Yannis E. Ioannidis, "Bitmap index design and evaluation," Proc. ACM SIGMOD, pp. 355-366, June 1998.

[12] Patrick O'Neil , Goetz Graefe , "Multi-Table Joins Through Bitmapped Join Indices," SIGMOD Record 24(3), September 1995.

[13] Ming-Chuan Wu, Alejandro P. Buchmann. "Encoded bitmap indexing for data warehouses," Proc. 14th ICDE, Orlando, Florida, pp. 220-230, 1998.

[14] Patrick O'Neil, Dallan Quass. "Improved Query Performance with Variant Indexes," Proc. ACM SIGMOD, pp. 38-49, Tucson, Arizona, 1997.

[15] Myung Kim, Yoonsun Lim, "A Z index based MOLAP Storage Scheme", Journal of Korea Information Science Society, Vol. 29, No. 4, pp. 262-273, August 2002.

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

[17] Comshare, Inc., "ROLAP: 그 기대치와 현실", White Paper, http://www.olapforum.com/ForumOLAP/OLAP%20General/White%20Paper/ROLAP-그%20기대치와%20현실.asp, 1995.(금용찬 역)

임 윤 선

1987 중앙대학교 수학과 학사. 2003 이화여자대학교 컴퓨터학과 석사. 1992~현재 (주)아리스트 개발이사. 관심분야는 OLAP, 데이타마이닝, 지식관리

김 명

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