AVL 트리를 사용한 효율적인 스트림 큐브 계산

김 지 현[†] · 김 명[††]

요 약

스트림 데이터는 끊임없이 고속으로 생성되는 데이터로써 최근 이러한 데이터를 분석하여 부가가치를 얻고자 하는 노력이 활발히 진행 중이다. 본 연구에서는 스트림 데이터의 다차원적 분석을 위해 큐브를 고속으로 계산하는 방법을 제안한다. 스트림 데이터는 비즈니스 데이터와는 달리 정렬되지 않은 채로 도착하며, 데이터의 끝에 도달하지 않은 상태에서는 집계 결과를 낼 수 없어서, 고속으로 집계하는 과정에서 저장 공간의 낭비를 심하게 초래한다. 또한 큐브에 속한 집계 테이블들을 모두 생성하는 것은 시간/공간 측면에서 비효율적이라는 점이 지적되고 있다. 이러한 문제를 해결하기 위해 본 연구에서는 기존 연구들과 마찬가지로 큐브에 포함시킬 집계 테이블들을 사용자가 미리 정하도록 하였고, 정렬되지 않은 스트림 데이터를 고속으로 집계하는 과정에서 배열과 AVL 트리들로 구성된 자료구조를 집계 테이블의 임시 저장소로 사용하였다. 제안한 알고리즘은 생성하려는 큐브가 메모리에 상주할 수 없을 정도로 큰 경우에도 집계 연산을 수행할 수 있다. 이론적 분석과 성능 평가를 통해 제안한 큐브 계산 알고리즘이 실용적임을 입증하였다.

키워드 : 스트림 데이터, 큐브 계산, 블로킹 연산, 데이터 집계

Efficient Computation of Stream Cubes Using AVL Trees

Jihyun Kim[†] · Myung Kim[††]

ABSTRACT

Stream data is a continuous flow of information that mostly arrives as the form of an infinite rapid stream. Recently researchers show a great deal of interests in analyzing such data to obtain value added information. Here, we propose an efficient cube computation algorithm for multidimensional analysis of stream data. The fact that stream data arrives in an unsorted fashion and aggregation results can only be obtained after the last data item has been read, cube computation requires a tremendous amount of memory. In order to resolve such difficulties, we compute user selected aggregation tables only, and use a combination of an array and AVL trees as a temporary storage for aggregation tables. The proposed cube computation algorithm works even when main memory is not large enough to store all the aggregation tables during the computation. We showed that the proposed algorithm is practically fast enough by theoretical analysis and performance evaluation.

Key Words : Stream Data, Cube Computation, Blocking Operator, Data Aggregation

1. 서 론

스트림 데이터[1]는 고속으로 끊임없이 생성되어 수집되는 데이터를 말한다. 거리 곳곳에 설치되어 있는 공기 오염도 측정 센서들로부터 수집되는 데이터가 좋은 예가 될 수 있다. 이외에도 증권 거래 데이터, 네트워크 트래픽 모니터링 데이터, 웹 사이트 클릭 스트림 데이터, 건강관리를 위한 생체 신호 데이터, 고객들의 무선 전화 통화 내역, 등이 흔하게 찾아볼 수 있는 스트림 데이터이다.

센서 네트워크 기술의 발달과 유비쿼터스 환경 구축이 실용화되면서 스트림 데이터를 분석하여 유용한 부가가치 정보를 얻기 위한 노력이 최근 활발히 진행되고 있다. 대표적인 스트림 데이터 처리 시스템으로는 STREAM[1, 2, 3], 네트워크 트래픽 모니터링을 위한 Aurora[4]와 Gigascope[5] 등을 들 수 있으며, 이외에도 다양한 스트림 처리 시스템들이 현재 개발되고 있다. 이러한 시스템들은 기존의 비즈니스 데이터 처리를 위한 DBMS들과는 다른 데이터 처리 방식을 택하고 있는데, 그 주요 이유는 스트림 데이터는 빠른 속도로 끊임없이 발생하여 수집되며, 그 발생 속도를 조절하기가 어렵고, 분량이 방대하여 저장해 두기 곤란하다는 특징 때문이다. 따라서 대부분의 스트림 데이터 처리 시스템들은 질의를 미리 등록해 놓고, 흘러가는 데이터 상에서 등록된 질의들을 처리하는 방식을 택한다. 특정 기준을 만족하는 데이터를 선별하는 필터링 연산이 스트림 데이터 상에서 수행되는 연산의 좋은 예이다.

스트림 데이터의 다차원적 분석에 유용하게 쓰이는 연산들 중에 집계 연산(aggregation) [6, 12]이 있다. 이 연산은 다

※ 본 연구는 학술진흥재단의 지원(과제번호: R04-2004-000-10149-0)으로 수행되었음.
† 준 회 원 : 이화여자대학교 컴퓨터학과 박사과정
†† 종신회원 : 이화여자대학교 컴퓨터학과 교수
논문접수 : 2007년 5월 21일, 심사완료 : 2007년 8월 23일

차원 데이터(multidimensional data)를 특정 차원 조합을 기준으로 데이터를 집계하여 요약 정보를 생성하는 연산이다. 집계 연산에는 덧셈(summation), 개수(count), 평균(average) 등이 속한다. 예를 들면, 고객 무선 전화 통화 내역이 (발신인, 발신지역, 수신인, 수신지역, 통화 시작시간)의 5개 차원으로 구성되어 있고, 집계할 측정 데이터가 통화 시간이라고 하자. 집계 연산에는 발신 지역별 통화량, 수신 지역별 통화량, 시간대별 통화량 등의 계산을 들 수 있다.

집계 연산 결과를 내기 위해서는 대상 데이터 전체를 스캔해야 한다. 따라서 방대한 양의 비즈니스 데이터를 다차원적으로 분석하고자 할 때는 모든 집계 테이블들을 사전에 생성해 두는 것이 일반적이며, 이 때 생성된 집계 테이블 집합을 큐브(cube)라고 한다 [6]. 또한 시간/공간 효율적인 큐브 생성을 위해 분석 대상 데이터는 전처리를 통해 특정 기준으로 정렬해 두는 것이 일반적이다[13].

스트림 데이터로부터 큐브를 생성하기는 쉽지 않다. 정렬되지 않은 채로 끊임없이 흘러 들어오는 스트림 데이터 전체를 시간/공간 효율적으로 스캔하며 집계하기 어렵기 때문이다. 따라서 스트림 데이터를 집계할 때는 데이터를 특정 단위로(윈도우 단위) 분할한 후 윈도우 [11]마다 집계 연산을 하며, 큐브의 일부인 사용자가 선택한 집계 테이블들만 생성하는 방법을 사용한다. 정렬되지 않은 스트림 데이터의 효율적인 집계를 위해 다양한 집계 방법과 저장 구조가 제안되었다. 대표적인 연구로 StreamCube [7]와 Gigascope [5]의 집계 연산 처리 방식을 들 수 있다.

StreamCube[7]는 집계 연산을 용이하게 하고 연산 결과의 분량을 줄이기 위해 데이터 도메인 전문가가 사전에 정해 놓은 집계 테이블들만 생성하도록 한다. 생성될 테이블들은 단 하나의 시퀀스 S_1, S_2, ... , S_k를 이룬다. 여기서 S_{i+1}은 S_i로부터 집계 연산을 통해 생성이 가능하며, 사용자는 드릴다운(drill-down)이나 롤업(roll-up) 연산을 통해 이 시퀀스에 속한 집계 테이블들을 검색할 수 있다. 이 시퀀스에 속하지 않은 집계 테이블들은 필요시에 따로 생성하도록 한다. 시퀀스에 속한 집계 테이블들은 하나의 거대한 트리로 표현되며 주기억장치에 상주한다.

Gigascope[5]는 네트워크를 통과하는 데이터 패킷들을 모니터링하면서 그 패킷들의 소스, 목적지, 패킷 개수 등을 다차원적으로 신속하게 분석하기 위해 개발된 시스템이다. 스트림 데이터를 속도 빠른 캐쉬 메모리에서 부분적으로 집계하여 압축된 형태로 주기억 장치로 올려 보내는 방법을 쓴다. 이 방법은 네트워크 트래픽 데이터와 같이 클러스터링이 잘 되어 있는 데이터의 집계 연산에 효율적이다. 그러나 데이터가 다차원 공간에 고르게 분포되어 있는 경우는 캐쉬를 자주 비워야 하고, 주기억장치로 올려 보내지는 데이터가 정렬되어 있는 것이 아니므로 생성하고자 하는 집계 테이블들이 모두 주기억장치에 상주 가능한 정도로 작아야 한다.

이러한 알고리즘들은 제한적인 환경에서 작동한다. 생성되는 집계 테이블들이 모두 생성과정에서나 생성 후에도 주기억장치에 충분히 상주가능하다는 것이다. StreamCube에서는 또한 생성하고자 하는 집계 테이블의 선택에 제한을 두며, Gigascope는 집계 테이블 생성 알고리즘이 캐쉬메모리를 관리해야 하며, 클러스터된 데이터의 집계에 초점을 맞추고 있다.

본 연구에서는 정렬되지 않은 스트림 데이터를 보다 일반적인 환경에서 고속으로 집계하는 방법을 제안한다. 생성할 집계 테이블 선택은 StreamCube에서와 같이 제약을 두는 것이 아니라 사용자가 임의로 정할 수 있도록 한다. 정렬되지 않은 스트림 데이터로부터 직접 생성되는 집계 테이블들을 고속으로 생성하기 위해서 배열과 AVL 트리를 혼합한 자료구조를 집계 테이블의 임시 저장소로 사용한다. 생성하려는 집계 테이블들이 주기억장치에 상주할 수 없을 정도로 커도 큐브 생성이 가능하다. 큐브 생성에 AVL 트리를 사용하는 것이 실용적이라는 것을 분석과 실험을 통해 검증하였다.

본 논문은 다음과 같이 구성된다. 2절에서 데이터의 다차원적 분석에 대해 간략히 소개하고 기존 관련 연구 결과를 분석한다. 3절에서는 본 연구가 제시하는 큐브 생성 방법을 소개하고, 4절에서 큐브 생성 과정을 고속화하고 메모리 사용을 최적화하는 추가적인 방법들을 설명한다. 5절에서 본 연구가 제안한 큐브 생성 방법을 분석하고 실험을 통해 효율성을 보이며, 6절에서 결론을 맺기로 한다.

2. 문제 정의 및 기존 연구 분석

먼저 본 연구에서 해결하고자 하는 문제를 정의하고, 관련 연구 결과를 간단히 소개하며 보완되어야 할 점들을 분석해 보기로 한다. 예를 들어, 통신회사에서 고객의 무선 전화 통화 내역을 바탕으로 다차원적 분석을 한다고 가정해 보자. 고객 통화 내역이 발신인(A), 수신인(B), 발신 지역(C), 수신 지역(D), 통화 시작 시각(E)의 5개 차원으로 구성되어 있으며, 집계할 측정 데이터(measure data)는 통화 시간이라고 하자. 이 데이터 스트림(또는 테이블)을 $ABCDE$로 부르기로 한다. 각 차원은 다음과 같은 계층 구조를 갖는다고 하자. 발신인 차원 A는 개인(A_1)과 연령별 그룹(A_2) 계층으로 나뉘고, 수신인 차원 B도 유사하게 B_1과 B_2로 나뉜다. 발신 지역 차원 C는 지역 범위에 따라 동(C_1), 구(C_2), 시(C_3) 계층을 가지며, 수신 지역 차원 D도 유사하게 D_1, D_2, D_3으로 나뉜다. 그리고 통화 시작 시각 E차원은 분(E_1), 시(E_2), 일(E_3), 주(E_4) 계층을 갖는다. 이러한 통화 내역 레코드가 스트림을 형성하며 입력 될 때, 매 시간 단위로 특정 연령대의 사람들이 전화를 걸었는가를 도시의 구 단위 범위로 분석해 보려면 집계 테이블 $A_2C_2E_2$를 생성해야 하고, 도시마다 시간대별로 전화 수신량을 분석해 보려면 집계 테이블 D_3E_2를 생성해야 한다.

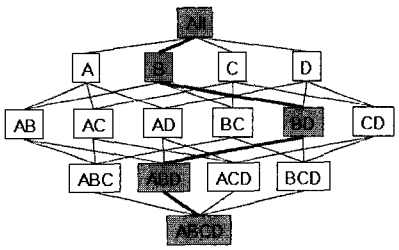
분석할 데이터가 n차원 데이터이고, 차원에 계층이 별도로 없는 경우는 생성 가능한 집계 테이블들의 개수는 스트림 데이터(원본)을 제외하면 2^n-1개가 된다. 예를 들면, 5차원 데이터의 경우 생성 가능한 집계 테이블의 수는 $31(=2^5-1)$개이다. 이를 D_1, D_2, D_3,$\cdots$, D_k 차원을 갖는 데이터로 확장해 보자. 차원 D_i, $1 \le i \le k$, 가 h_i개의 계층을 갖는다고 하면, 큐브에 속한 집계 테이블의 개수는 $(h_1+1)\times(h_2+1)\times\cdots\times(h_k+1)$ 이 되고, 이는 달리 표현하면, $\prod_{i=1}^{k}(h_i+1)$로 표시할 수 있다. 이 계산에 의하면, 각각 3 계층을 갖는 5차원 데이터 테이블 (스트림)으로부터 생성될 수 있는 집계 테이블의 수는 $4^5=1{,}024$ 개가 된다. 또한 집계 테이블의 밀도가 높은 경우 방대한 양의 집계 결과가 생성된다는 것을 알 수 있다.

본 연구에서는 대용량 스트림 데이터가 매우 빠른 속도로 생성되어 도착한다는 가정 하에 큐브에 속한 집계 테이블들 중에서 사용자(또는 분석가)가 미리 정해 놓은 집계 테이블들을 고속으로 생성하는 방법을 제안하고자 한다. 스트림 데이터는 발생 시간 기준으로 윈도우로 구분되며, 큐브 계산은 윈도우 1개가 끝나는 곳에서 완료된다.

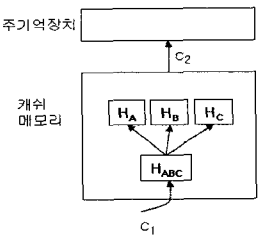
스트림 데이터를 이와 같이 요약하여 다차원적 분석을 해 보려는 노력 중에 대표적인 결과로 StreamCube[7]와 Gigascope [8]의 연구를 들 수 있다. 여기서 이들을 간략하게 소개하고 분석해 보기로 한다. 참고로 비즈니스 데이터의 경우는 [9, 10, 13]을 비롯한 효율적인 큐브 생성 방법이 있으나 이는 데이터가 정렬되어 있다는 가정 하에 알고리즘이 작동하므로 이러한 큐브 생성 방법은 (정렬되어 있지 않은) 스트림 데이터의 다차원적 분석에는 직접 활용이 어렵다.

StreamCube[7]에서는 사용 빈도가 높은 집계 테이블들만 선택하여 생성하는 방안을 제시하였다. 생성되는 집계 테이블들은 하나의 시퀀스를 이루며, 이를 S_1, S_2, ... , S_k라고 하자. 이 시퀀스는 데이터 도메인 전문가가 정한다. 집계 테이블 S_i와 S_{i+1}의 관계는 S_i로부터 S_{i+1}이 집계 연산을 통해 생성 가능하다는 것이다. (그림 1)에 예제가 있다. $ABCD \Rightarrow ABD \Rightarrow BD \Rightarrow B \Rightarrow All$ 이 도메인 전문가가 정한 집계 테이블 시퀀스이고, 다차원적 분석은 주로 이 시퀀스에 속한 집계 테이블들을 이동해 가면서 이루어진다고 본다. 다른 집계 테이블들은 필요시에 생성한다.

StreamCube의 생성과 저장 구조를 살펴보자. StreamCube에 속한 집계 테이블들은 모두 하나의 트리 구조에 묶여 있다. 집계 테이블 S_i의 셀들은 트리의 레벨 i 노드가 된다. 데이터 스트림에서 하나의 셀(레코드)이 입력되면 그 셀은 트리의 맨 하위 레벨의 해당 셀에 집계된 후, 상위 레벨로 올라가면서 계속 집계된다. 예를 들어 (그림 1)에서는 입력 데이터가 트리의 레벨 4의 노드에 입력된 후, 레벨 3(ABD), 레벨 2(BD), 레벨 1(B)과 레벨 0(All)의 해당 노드에 차례로 집계된다. StreamCube의 단점은 생성할 집계 테이블의 선택에 제약이 크며, 생성할 모든 집계 테이블들이 트리로 연결되어 있어서, 트리 전체가 메모리에 상주해야만 집계 연산이 가능하며, 차후에 행해질 다차원적 분석 연산들은 이 트리에 링크로 연결되어 있는 노드들 상에서 이루어진다는 점이다.



(그림 1) StreamCube 생성 예제



(그림 2) GigaScope의 집계 연산 과정

Gigascope[8]는 네트워크 트래픽 모니터링 시스템이다. 이 시스템은 네트워크를 통과하는 데이터 패킷들의 소스, 목적지, 패킷 개수 등을 효율적으로 분석하는 기능을 포함하고 있다. 이 연구에서는 집계 속도 향상을 위해 캐쉬 메모리를 직접 관리한다. 데이터 스트림이 입력되면 속도 빠른 캐쉬 메모리에 해쉬 테이블 구조로 저장되어 있는 각 집계 테이블에 데이터가 집계된다. 집계과정에서 충돌(collision)이 발생하거나 정해진 시간(예, 1~2분)이 되면 해쉬 테이블에 저장되어 있는 집계 결과가 주기억장치로 옮겨져서 집계되는 구조를 갖는다. (그림 2)의 예를 살펴보자. 이는 데이터 스트림 $ABCD$를 읽으면서 집계 테이블 ABC, A, B, C를 생성하는 경우이다. 캐쉬 메모리에는 이 들 각 집계 테이블에 대해 해쉬 테이블 H_{ABC}, H_A, H_B, H_C가 있다. 데이터 스트림으로부터 데이터(c_1)가 하나씩 입력될 때마다 그 값은 H_{ABC}, H_A, H_B, H_C에 모두 집계된다. 입력 값이 집계될 셀이 캐쉬 메모리에 존재하는 한 집계는 캐쉬 메모리에서 수행되며, 속도 향상 효과가 커진다. 그러나 캐쉬 메모리에 있는 해쉬 테이블들이 실제 집계 테이블보다 작기 때문에, 집계 테이블의 여러 셀이 해쉬 테이블의 셀 한 개를 공유하게 된다. 예를 들어, 집계 테이블 A의 셀 a_i와 셀 a_j가 H_A의 셀을 공유한다고 하자. 입력되는 데이터가 계속하여 a_i에 매핑되어 집계될 때는 H_A에서 집계된다. 그러다가 a_j에 매핑되는 셀이 입력되면 그동안 H_A에서 집계되고 있던 셀 a_i의 값은 주기억장치의 집계 테이블 A의 해당 셀에 옮겨지며 집계되며, 이제 H_A의 해당 셀은 a_j를 대표하게 되는 것이다. 이와 같이 a_i와 a_j가 공유하는 해쉬 테이블의 셀은 a_i와 a_j를 번갈아 가면서 부분적으로 집계하는데 도움을 준다.

Gigascope는 네트워크 트래픽 데이터와 같이 데이터가 클러스터되어 있는 경우에 상당한 데이터 압축 효과를 줄 수 있다는 장점을 갖는다. 그러나 데이터가 고르게 분포되어 있는 경우에는 캐쉬에서 데이터가 빈번히 주기억장치로 보내지게 되는 단점이 있다. 또한 캐쉬에서 주기억장치로 보내지는 데이터가 정렬된 것이 아니므로 주기억장치에는 해당 집계 테이블들이 모두 상주해야 하며, 그들이 배열과 유사한 형태(즉, 데이터의 위치를 통해 임의 접근이 가능한 형태)가 아니면 캐쉬에서 보내진 데이터의 삽입이 비효율적일 수밖에 없다. 만약 이를 해결하기 위해 집계테이블들이 주기억장치에서 배열 형태를 취한다면 집계 테이블의 데이터 희박성 문제에 부딪칠 수도 있다.

본 연구에서는 이러한 제약점들을 다음과 같은 측면에서 개선한 큐브 생성 방법을 제안한다. ① 생성할 집계 테이블들은 사용자가 임의로 지정한다. ② 메모리가 작아서 생성하려는 집계 테이블들이 모두 메모리에 상주할 수 없는 경우에도 실행 가능한 큐브 생성 알고리즘을 제시한다. ③ 큐브 생성 알고리즘은 시간, 공간 측면에서 효율적이며 실용적이다.

3. 큐브 생성 알고리즘

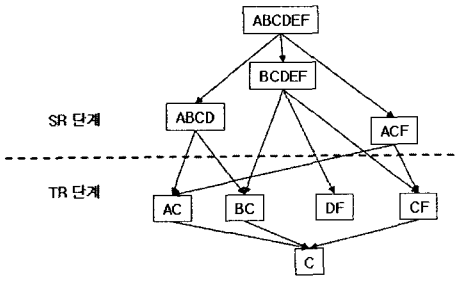
이 절에서는 본 연구에서 제안하는 큐브 생성 알고리즘을

소개한다. 알고리즘의 개략적 전략 및 구조를 설명한 후에 알고리즘 각 부분에서 효율을 높이는 방안을 설명한다.

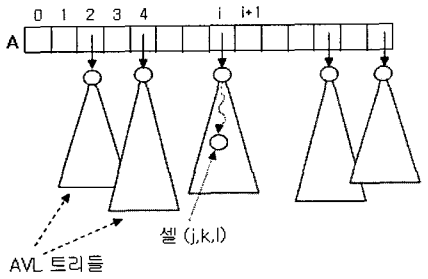
큐브 계산 알고리즘을 설명하기 위해 (그림 3)의 예제를 사용하기로 한다. 사용자는 6차원 스트림 데이터 *ABCDEF* 로부터 8개의 집계 테이블 *BCDEF*, *ABCD*, *ACF*, *AC*, *BC*, *DF*, *CF*, *C*를 생성하려 한다. 집계 테이블간의 화살표는 집계 테이블 생성에 사용될 수 있는 모든 부모 테이블들을 나타낸다. 입력 스트림으로부터 큐브 전체를 생성하고자 하는 경우에는 (그림 1)과 같이 화살표가 래티스(lattice) 구조를 이루겠지만 여기서는 사용자가 임의로 지정한 집계 테이블들만 생성하는 것이므로, (그림 3)과 같은 구조를 갖게 된다.

큐브 계산 알고리즘은 두 단계에 걸쳐 진행된다. 첫째 단계에서는 (정렬되지 않은 채로 도달하는) 데이터 스트림을 읽어 들이면서, 이로부터 직접 생성할 수밖에 없는 집계 테이블들을 생성한다. 예제의 *BCDEF*, *ABCD*, *ACF*가 이에 해당된다. 이 단계를 SR 단계(stream reading phase)로 부르기로 한다. 둘째 단계에서는 SR 단계에서 생성된 집계 테이블들을 정렬된 상태로 읽어서 남아 있는 집계 테이블들인 *AC*, *BC*, *DF*, *CF*, *C*를 생성한다. SR 단계에서 생성된 집계 테이블들은 트리 형태로 저장되므로, 이들을 읽으면서 집계하는 둘째 단계를 TR 단계(tree reading phase)로 부르기로 한다.

이제 각 단계에서의 집계 방법과 집계 테이블 저장구조에 대해 자세히 설명하기로 한다. SR 단계에서는 입력 데이터 스트림으로부터 직접 생성할 수밖에 없는 집계 테이블들이 생성된다. 예로써, 입력 스트림으로부터 집계 테이블 *ABCD* 가 생성되는 과정을 살펴보자. 입력 스트림 데이터가 특정 형태로 정렬되어 있는 것이 아니므로, 입력 데이터 $a_i b_j c_k d_l e_m f_n$ 를 읽게 되면 이 데이터가 집계될 *ABCD*의 $a_i b_j c_k d_l$ 을 신속하게 찾을 수 있는 방법이 필요하다. 만약 집계 테이블 *ABCD*를 저장하기 위해 배열 구조를 사용한다면 집계 연산은 신속하게 처리될 수 있으나 저장 공간의 낭비가 클 수도 있고, 저장 공간 부족으로 인해 남아 있는 집계 테이블의 생성이 곤란해질 가능성이 높다.



(그림 3) 큐브내의 집계 테이블간의 관계



(그림 4) SR 단계에서 생성되는 집계 테이블 *ABCD*의 임시 저장소 구조

본 연구에서는 이를 해결하기 위해서 SR 단계에서 생성되는 집계 테이블을 (그림 4)와 같이 1차원 배열과 이에 연결된 AVL 트리 집합 구조로 표현한다. 집계 테이블을 구성하는 차원 중에서 가장 큰 차원을 1차원 배열로 표현하고, 이 배열의 각 원소는 그림에서와 같이 AVL 트리 한 개를 가리키도록 한다. 예를 들어, 집계 테이블 *ABCD*의 가장 큰 차원이 *A*인 경우, $a_i b_j c_k d_l$ 셀은 1차원 배열 원소 $A[i]$가 가리키는 AVL 트리의 $b_j c_k d_l$를 나타내는 노드에 저장된다. AVL 트리 노드들은 (j,k,l)로부터 계산된 1차원적 인덱스를 가지며, 이 인덱스를 기준으로 데이터가 정렬되어 집계/저장되는 것이다. AVL 트리의 노드는 그 노드에 처음으로 데이터가 집계될 때 생성되고, 차후에 집계될 때는 이진 탐색 기법을 통해 해당 노드를 찾은 후 그 노드에 데이터가 집계된다. 따라서 AVL 트리는 집계 연산이 진행되는 가운데 계속 커지게 된다.

SR 단계에서 사용되는 집계 테이블의 저장구조는 다음과 같은 장점을 갖는다. 우선, 정렬되어 있지 않은 (희박한) 입력 스트림 데이터를 적은 양의 메모리를 사용하여 신속하게 집계할 수 있다. 1차원 테이블(헤드 테이블로 부르기로 함)을 사용하여 해쉬 효과를 볼 수 있을 뿐 아니라, AVL 트리의 깊이를 낮출 수 있다. 이진 탐색을 위해 AVL 트리를 선택한 이유는 AVL 트리가 탐색 트리의 높이를 최소로 유지하도록 하며, 높이를 최소화하는데 드는 비용이 매우 작기 때문이다. 높이를 최소화하기 위해 노드 회전이 필요한데 최악의 경우라고 해도 링크 4개 정도 변경하는 정도의 비용만이 들기 때문이다.

정해진 윈도우 내의 입력 스트림이 모두 읽히고 나면 SR 단계 집계 테이블 생성이 완료된 것이다. 이제 AVL 트리를 읽으면서 남아있는 집계 테이블을 생성하는 TR(tree reading phase) 단계가 시작된다. 이 단계에서는 SR 단계에서 생성된 집계 테이블의 헤드 테이블의 원소를 순서대로 스캔하면서 각 셀이 가리키는 AVL 트리를 inorder tree traversal 방식으로 읽어 나간다. 즉, (그림 3)의 예제에서는 집계 테이블들 *BCDEF*, *ABCD*, *ACF*를 읽으면서 *AC*, *BC*, *DF*, *CF*, *C*를 생성하는 것이다. (완성된 집계 테이블들을 모두 디스크에 저장하고자 한다면 트리 형태로 저장된 *BCDEF*, *ABCD*, *ACF*를 정렬된 순서로 읽어 나가는 것과 동시에 이들을 테이블 구조로 저장하면서 디스크로 내보내면 된다.)

TR 단계는 SR 단계와는 달리 입력 데이터가 정렬되어 있다. 따라서 이 단계에서는 기존의 비즈니스 데이터의 OLAP 큐브 생성에 사용되었던 효율적인 집계 테이블 생성 알고리즘[10]을 활용할 수 있다. TR 단계에서의 집계 테이블 생성은 계층적으로 진행된다. 집계 테이블들 중에는 SR 단계에서 완성된 집계 테이블들로부터 직접 생성되는 것도 있지만, *C*와 같이 부모 테이블이 TR 단계에서 생성되는 것들도 있다. TR 단계에서의 집계 테이블 생성 시간은 부모 테이블을 한 번 스캔하는데 걸리는 시간이므로 부모 테이블의 크기에 비례한다. 따라서 *AC*, *BC*, *CF*와 같이 부모 집계 테이블이 여럿인 경우는 테이블의 차원 길이의 곱이 가장 작은 것을 부모로 택한다.

TR 단계에서 생성되는 집계 테이블들은 배열 형태로 저장된다. 이 단계에서 생성되는 집계 테이블들은 스트림 데이터로부터 이미 두 단계에 걸쳐 집계되는 테이블들이므로 압축률이 높아서 밀도가 상당히 높기 때문이다. TR 단계에서 생성되는 집계 테이블들의 생성 시간은 SR 단계에서 완성된 집계 테이블들을 한 번만 스캔하면서 배열 형태의 여러 집계 테이블에 집계 연산을 하는 정도이다. 개략적인 큐브 계산 알고리즘은 아래의 Cube_Computation_Algorithm과 같고, 구체적인 설명과 분석은 이어서 하기로 한다.

Cube_Computation_Algorithm

SR: Stream Reading Phase

[SR-1] SR 단계에서 생성할 집계 테이블 리스트 SR-List1을 결정한다.

[SR-2] 입력 데이터 스트림을 스캔하면서 SR-List1에 속한 집계 테이블들을 생성한다.
(집계 테이블들은 배열과 AVL 트리를 혼용한 자료구조에 저장한다.)

TR: Tree Reading Phase

[TR-1] TR 단계에서 생성될 집계 테이블들의 부모 집계 테이블들을 정한다. 집계 테이블들을 다음과 같이 두 개의 독립적인 집합(리스트)로 분류한다.
TR-List1: SR-List1에 속한 집계 테이블을 부모로 하는 집계 테이블들
TR-List2: TR-List1에 속한 집계 테이블을 부모로 하는 집계 테이블들

[TR-2] AVL 트리에 저장되어 있는 SR-List1 집계 테이블들을 inorder traverse 방식으로 읽으면서 순서대로 디스크에 출력하고, TR-List1에 속한 (배열 형태의) 집계 테이블들을 생성한다.

[TR-3] TR-List1에 속한 집계 테이블들을 읽으면서 순서대로 디스크에 출력하고, 집계 테이블이 TR-List2에 속한 집계 테이블의 부모인 경우는 해당 자식 집계 테이블들을 생성한다. TR-List2에 속한 집계 테이블들도 배열 형태로 저장된다.

[TR-4] TR-List2에 속한 집계 테이블들을 순서대로 읽으면서 디스크에 출력한다.

이제 큐브 계산 알고리즘에 대한 구체적인 설명과 효율성을 높이기 위해 고려해야 할 사항들을 살펴보기로 한다.

3.1 SR 단계에서 생성할 집계 테이블 선택

SR 단계에서 생성할 집계 테이블들은 알고리즘의 [SR-1]에서 결정된다. 집계 테이블 T_c가 집계테이블 T_p로부터 생성가능하려면 집계 테이블 T_c를 구성하는 모든 차원들이 집계 테이블 T_p를 구성하는 차원들에 속해 있어야 한다. 예를 들어, (그림 3)에서 *ACF*는 *ABCDEF*로부터 생성될 수는 있지만 *ABCD*나 *BCDEF*로부터 생성될 수는 없다. 문자열 '*ACF*'가 문자열 '*ABCD*'나 '*BCDEF*'의 서브 스트링이 아니기 때문이다. [SR-1]에서는 입력 스트림 이외의 다른 집계 테이블로부터는 생성이 불가능한 집계 테이블들만 생성한다. 이러한 집계 테이블들만 SR 단계에서 생성하는 이유는 집계 테이블의 생성 시간은 결국 부모 테이블을 읽는 시간이기 때문이다. 따라서 입력 스트림을 읽는 것보다는 일단 집계된 다른 테이블을 읽어서 추가로 집계하는 것이 더 빠르다.

3.2 SR 단계에서 사용되는 AVL 트리의 크기 예측과 트리 유지 오버헤드

SR 단계의 수행 시간 효율성은 각 집계 테이블 저장에 사용되는 AVL 트리들의 깊이에 크게 좌우된다. 예를 통해 AVL 트리의 크기에 대해 살펴보기로 한다. 주기억장치 크기를 1 GB(Giga bytes)라고 하자. AVL 트리의 각 노드에는 그 노드가 나타내는 셀 번호(4 바이트), 집계되는 데이터(4 바이트), 두 자식 노드를 위한 링크(8 바이트)가 소요된다고 하자. SR 단계에서 5~6개 정도의 집계 테이블이 생성된다고 가정하면 각 집계 테이블에는 평균적으로 10M(Mega)개의 셀이 포함된다고 볼 수 있다. 집계 테이블의 헤드 테이블(1차원 배열)은 멤버의 수가 큰 차원이 되며, 헤드 테이블은 대략 1,000~10,000개 정도의 원소를 갖는다고 해보자. 그렇다면 헤드 테이블의 각 원소는 평균 1,000~10,000개의 노드를 갖는 AVL 트리를 가리키게 된다. 이러한 AVL 트리의 깊이는 대략 10~15 정도가 되어, AVL 트리를 탐색하거나 갱신하기 위해서는 대략 10~15개 정도의 노드 탐색이 필요하게 된다.

AVL 트리의 깊이를 균등하게 유지하기 위해서 깊이 불균형이 발생하는 경우 서브 트리를 회전하게 된다. 단회전과 복회전이 있는데 단회전의 경우는 링크 2개, 복회전인 경우는 링크 4개를 변경해야 하며, 이는 깊이 균형을 유지하는데 드는 오버헤드로는 매우 작다고 볼 수 있다. AVL 트리의 유지 비용은 4절에서 실험을 통해 점검하기로 한다.

3.3 SR 단계에서 생성되는 집계 테이블 저장에 다차원 헤드 테이블 사용

집계 테이블 데이터가 다차원 공간에 고르게 분포되어 있지 않다면, 데이터를 저장하는데 쓰이는 AVL 트리들이 지나치게 커질 수 있다. 이런 경우에는 헤드 테이블에 2개 이상의 차원을 사용할 수 있다. 예를 들어, 집계 테이블 *ABCD*를 나타내기 위해 차원 *A*와 *B*를 헤드 테이블에 사용한다고 하자. 즉, 헤드 테이블은 2차원 배열인 *AB*이다. 각 AVL 트리에는 *CD*의 셀들만 포함되며, AVL 트리의 크기는 최대 $1/B$만큼 줄어들게 된다. 특히 헤드 테이블로 사용될 평면이 밀도가 높게 되도록 헤드 테이블에 사용할 차원들을 선택한다면 *CD*를 저장하는 AVL 트리의 높이가 훨씬 낮아진다. 또한 *ABCD*를 이와 같은 자료구조로 저장한다면, 이 집계 테이블은 *ABCD*로 정렬된 것으로 볼 수도 있고, *BACD*로 볼 수도 있다. 여러 순서로 데이터를 정렬시킬 수 있다는 것은 TR 단계에서 생성될 집계 테이블들이 차지할 메모리 공간을 줄일 수 있다[10]는 것을 의미한다.

3.4 TR 단계에서 생성될 집계 테이블들의 부모 테이블 결정

TR 단계에서 생성될 집계 테이블들은 부모 테이블이 SR 단계에서 이미 생성된 것인가에 따라 두 부류 TR-List1, TR-List2로 나뉜다. TR-List1에 속한 집계 테이블들은 SR 단계에서 생성된 AVL 트리 형태의 집계 테이블을 부모로 하는 경우이며 [TR-2]에서 생성된다. 이 경우는 부모가 될

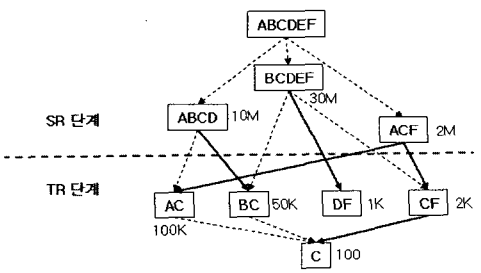
수 있는 집계 테이블들의 크기를 미리 알 수 있으므로 그들 중에서도 가장 작은 것을 택하여 부모로 삼는다. [TR-2]에서조차 생성되지 않은 모든 집계 테이블들은 TR-List2에 속한다. TR-List2에 속한 집계 테이블들은 부모 집계 테이블이 완성된 상태에서 생성 계획이 수립되는 것이 아니므로 부모 테이블의 크기를 미리 알 수 없다. 따라서 부모가 될 수 있는 집계 테이블들 중에서 데이터가 100% 밀집해 있다고 볼 때 가장 작은 집계 테이블, 즉 각 차원 길이의 곱이 가장 작은 집계 테이블을 부모로 정한다. 부모 테이블들이 배열로 저장되기 때문에 그것이 바로 부모 테이블의 실제 크기가 된다. 예를 들어 (그림 5)를 보자. *ABCD*, *BCDEF*, *ACF*는 SR 단계에서 이미 생성되어 있고, 크기는 각각 10M, 30M, 2M이다. *AC*, *BC*, *CF*, *DF*는 TR-List1에 속하며 SR 단계에서 생성되어 AVL 트리 형태로 저장되어 있는 집계테이블들 중에서 가장 작은 테이블을 부모노드로 선택한다 (그림에서 굵은 선 표시). 집계 테이블 *C*는 TR-List2에 속하며, [TR-2]에서 생성된 집계 테이블을 부모로 하게 된다. D_i가 차원 i의 크기라고 하고, D_A=1K, D_B =500, D_C =100, D_D=50, D_E=30, D_F= 20이라고 가정하자. 그러면 집계 테이블 *C*의 부모는 크기가 2K 이면서 가장 작은 집계 테이블인 *CF*가 된다.

3.5 TR 단계의 집계 테이블 계산에 필요한 메모리 공간 예측

알고리즘의 [TR-1] 단계에서 집계 테이블 생성 계획이 수립되면 TR 단계에서 필요한 메모리의 양을 알 수 있다. 예를 들어, 집계 테이블 *AC*는 *ACF*에서 생성된다. 집계 테이블 *ACF*는 $A \Rightarrow C \Rightarrow F$ 순서로 정렬되어 있기 때문에 *ACF*를 순서대로 읽으면 *AC*의 셀도 순서대로 생성되기 때문에 이 때 필요한 공간은 1셀 뿐이다. 즉 자식 집계 테이블에 속한 차원들이 부모 집계 테이블의 차원들을 나타내는 스트링의 prefix이면 자식 집계 테이블 생성에는 상수 개의 셀만이 필요하다. 이를 일반화해 보면, 자식 집계 테이블이 $D_1 D_2 \cdots D_i D_{j_1} \cdots D_{j_p}$ 이며, 첫 i차원이 부모 집계 테이블과 동일한 경우, 자식 집계 테이블을 생성하기 위해서는 남은 차원들의 조합만큼 (즉, $D_{j_1} \cdots D_{j_p}$)의 메모리 공간이 필요하다.

3.6 SR 단계 수행에 메모리가 부족한 경우의 대책 방안

SR 단계는 TR 단계에 비해 메모리를 많이 차지한다. 정렬되지 않은 입력 스트림 데이터를 한 번만 스캔하면서 상대적으로 고차원인 집계 테이블들을 생성해야 하기 때문이다. 만약 SR 단계 수행 중에 AVL 트리들이 크게 자라나서 메모리가 부족하게 되는 경우는 다음과 같은 대책을 세울 수 있다. 우선 TR 단계에서 생성될 집계 테이블들을 배열에 저장시킨다고 가정하고 이들의 저장 공간을 확보한다. SR 단계는 이들이 차지하는 공간을 남겨 놓고 남은 공간만을 활용한다. SR 단계 수행 중에 메모리가 거의 차게 되면 SR 단계를 중단하고 현재 AVL 트리에 저장되어 있는 값만으로 TR 단계에서 생성해야 할 집계 테이블에 집계 연산을 한다. AVL 트리를 읽어 가면서 동시에 그 내용을 디스크 파일에도 출력한다. AVL 트리를 비우고 TR 단계의 집계 테이블에 현재 값이 집계되면 다시 입력 스트림을 읽으면서 SR 단계를 재개한다. SR 단계 실행 중에 다시 AVL 트리가 커지면서 메모리가 차게 되면 SR 단계를 중단하고 다시 앞서서 수행했던 것과 같이 TR 단계에서 생성되는 집계 테이블에 AVL 트리 내용을 집계하면서 트리를 비우고 동시에 AVL 트리의 내용을 파일에 비운다. 디스크 파일들에 쓰여진 값들은 순서가 같으므로 차후 해당 집계 테이블이 필요한 경우 파일의 값을 합병(merge)하여 사용한다.



(그림 5) TR 단계에서 생성되는 집계 테이블들의 부모 테이블 결정

3.7 알고리즘의 정확성

Cube_Computation_Algorithm의 정확성을 설명하기로 한다. 우선 사용자가 생성하고자 하는 집계테이블들을 정하고 나면, 각 집계 테이블의 이름으로부터 (그림 3)과 같은 부모 자식 관계를 찾을 수 있다. 부모가 여럿인 경우는 위의 설명 (3.1)의 방법을 사용하여 SR 단계에서 생성할 집계 테이블 리스트인 SR-List1을 정할 수 있다. SR-List1의 집계 테이블들은 입력 스트림을 읽으면서 AVL 트리의 해당 셀에 읽은 값을 더해 가면 된다. TR 단계에서 생성할 집계 테이블들은 TR-List1과 TR-List2에 속한다. 생성하고자 하는 집계 테이블이 이 두 리스트 중에서 어떤 리스트에 속하는가는 설명 (3.4)에 자세히 기술되어 있다. TR-List1에 속한 집계 테이블들을 생성할 때는 SR 단계에서 생성한 집계 테이블들을 inorder 방식으로 순행하면서 정렬된 상태로 집계할 셀들을 TR 단계로 입력한다. TR-List2에 속한 집계 테이블들을 생성할 때는 TR-List1 단계에서 생성된 집계 테이블들을 역시 정렬된 채로 읽기 때문에 기존의 효율적인 비즈니스 데이터를 집계하는 알고리즘을 사용한다. 따라서 Cube_Computation_Algorthm은 정확하게 사용자가 지정한 집계 테이블들을 생성할 수 있다.

4. 집계 연산 알고리즘의 성능 평가

본 연구에서 제안한 큐브 계산 알고리즘의 성능 평가를 실시하였다. 실험 목적은 SR 단계에서 사용한 AVL 트리들의 오버헤드 정도를 파악하고 제안한 알고리즘이 대용량 스트림 데이터에 대해서도 실용적이라는 점을 보이기 위한 것이다. 참고로, 기존의 집계 연산 방법들과는 환경적인 가정에 차이가 있기 때문에 이들과는 성능을 비교하지 않았다. StreamCube는 생성할 집계 테이블들의 셀들이 메모리에 상주하는 거대한 트리의 노드에 매핑되도록 하여 집계 연산이 고속으로 실행될 수 있도록 하며, 집계 연산 결과도 주기억장치에 그대로 남아 있도록 한다. Gigascope는 캐쉬메모리를 관리해야 하고, 주기억장치 용량이 충분하여 집계 결과가 역시 주기억장치에 상주하도록 한다. 그러나 본 연구에서는 생성할 집계 테이블의 선택에 제한을 두지 않고 주기

억장치의 크기가 집계 연산 도중에 부족할 수도 있다는 일반적인 가정을 하며 집계 연산 결과가 디스크로 출력되도록 한다. 따라서 본 실험에서는 기존 집계 연산 방법들에 비해 실행속도가 우수하다는 것을 보이는 것이 아니라 일반적으로 사용될 수 있는 비교적 실용적인 집계 알고리즘이라는 것을 보이는데 초점을 둔다.

실험에는 디스크 파일에 저장되어 있는 6차원 스트림 데이터를 사용하였으며, 데이터는 실험 목적에 맞는 구조로 합성하여 생성하였다. 실험은 1 GB 메모리가 장착된 3.0 GHz 펜티엄 PC에서 실시하였다. 큐브 계산 알고리즘은 C언어로 구현하여 마이크로소프트사의 .NET 2003 환경에서 실행하였으며 정확도를 높이고자 각 실험은 10회 실시한 후 그 평균값으로 산정하였다.

4.1 SR 단계에서 사용된 AVL 트리 유지 오버헤드

큐브 계산 알고리즘의 SR 단계에서는 집계 테이블 저장을 위해 배열에 연결된 AVL 트리들을 사용한다. 이 단계의 실행시간은 생성할 집계 테이블들의 차원의 개수나 집계 테이블의 개수에 달려 있는 것이 아니라, SR 단계에서 생성되는 셀(레코드)의 개수와 어떤 방식으로 셀들을 저장하는가에 크게 달려 있다. 이 실험에서 입력 데이터 스트림은 10 M 셀로 구성되며, 154 Mbytes 크기의 파일에 저장되어 있다. 이 입력 스트림으로부터 1 M 셀을 갖는 4차원 집계 테이블을 생성하여 디스크 파일(크기: 10 Mbytes)에 저장하는 것을 실험하였다. 집계 연산 과정에서 이 집계 테이블은 배열에 붙어 있는 AVL 트리들에 저장되도록 하였다.

실험에서는 동일한 양의 입력 데이터로부터 동일한 양의 셀을 생성하는 과정에서 AVL트리의 개수를 달리 함으로써 AVL 트리들의 평균 깊이가 변화될 때의 집계 테이블 생성 시간을 계산해 보았다. 실험 결과는 <표 1>에 있다. 1 M 셀을 생성할 때의 실행 시간을 살펴보자. AVL 트리의 깊이가 대략 6~7인 경우 실행시간은 48.28초이고, AVL 트리의 깊이가 대략 16~17인 경우 실행시간은 51.82초로써 3.54초의 차이만을 보인다. 100 K 셀을 생성하는 경우에도 AVL 트리의 깊이가 3~4인 경우와 13~14인 경우의 실행시간 차이는 단지 1.79초에 불과하다. 10 K 셀을 생성하는 경우에도 집계 테이블 생성 시간이 트리의 깊이에 크게 좌우되지 않는다는 것을 볼 수 있다. 또한 깊이가 16~17인 AVL 트리의 노드 수가 대략 100,000개 인 것을 감안하면 이러한 크기의 AVL 트리도 집계 연산의 고속처리를 위한 테이블 임시 저장소로 쓰이는데 실용적이라는 것을 알 수 있다.

4.2 AVL 트리와 이진 탐색 트리의 유지 비용 비교

AVL 트리는 깊이 균형을 유지하기 위해 균형이 깨지는 경우 노드들을 회전시킨다. 노드 삽입시에 이진 탐색 트리와 마찬가지로 트리의 단말까지 내려가고, 균형이 깨지는 경우 단회전이 필요하면 노드 링크 2개 갱신, 복회전이 필요한 경우는 노드 링크 4개 갱신이 요구된다. 여기서는 이러한 균형 유지 비용에 대해 평가해 보려고 한다. 실험에는 입력 스트림 셀들이 임의 순서인 파일(rand)과 정렬되어 있는 파일(skew)을 사용하였다. 두 파일 모두 10M개의 셀을 갖는다. 각 파일에 대해 AVL 트리를 사용한 집계 시간과 이진 탐색 트리를 사용한 집계 시간을 계산해 보았다.

실험 결과는 <표 2>에 나타나 있다. 모든 경우 집계 결과는 대략 10 K 셀 (1.1 Mbytes) 정도이다. 결과를 보면 AVL 트리의 경우는 동일한 입력에 대해서는 트리의 깊이에 영향을 거의 받지 않고 실행 시간이 비슷하다. 또한 데이터가 정렬되어 있어서 이진탐색 트리의 경우에 매우 불리한 경우인 skew 데이터에 대해 오히려 임의 순서 데이터를 처리할 때보다 더 속도가 빠르다는 것을 알 수 있다. 입력 데이터가 임의 순서인 경우는 AVL 트리를 사용하는 경우와 이진탐색 트리를 사용하는 경우 실행 시간 차이를 거의 볼 수 없다. 그러나 데이터가 매우 skew 되어 있는 경우 이진 탐색 트리는 실용적으로 사용할 수 없다.

4.3 샘플 데이터의 집계 시간

위에서는 AVL 트리의 성능을 위주로 실험하였으나 여기서는 샘플 데이터를 정하여 집계 연산을 해 보았다. 샘플 데이터와 생성할 집계 테이블들은 (그림 3)과 같다. 즉, 입력 스트림은 *ABCDEF*의 6차원 데이터이고, 집계 테이블 생성 계획은 (그림 5)와 같다. 실험은 입력 데이터 스트림 크기를 10 K~10 M로 변경해 가고, 출력 셀 개수를 1,200~1.11 M 로 변경하면서 실시하였다. 입력되는 데이터와 생성될 셀의 개수가 대략 10배 차이가 날 때 집계 알고리즘의 실행시간도 대략 10배 차이가 나는 것을 볼 때 (행간 비교), 이 알고리즘은 scalable하다고 볼 수 있다. 데이터가 임의 순서일 때 AVL 트리를 사용하는 경우와 이진탐색 트리를 사용하는 경우의 실행시간은 비슷하며, 위에서와 마찬가지로 AVL 트리를 사용하는 경우는 데이터가 정렬되어 있어

〈표 1〉 AVL 트리 유지에 드는 오버헤드

입력 스트림 셀 개수	AVL 트리 깊이	실행 시간 (단위: 초)		
		1 M 셀 생성	100 K 셀 생성	10 K 셀 생성
10 M	16~17	51.82	-	-
10 M	13~14	50.69	42.03	-
10 M	9~10	49.42	41.37	39.71
10 M	6~7	48.28	41.81	38.73
10 M	3~4	-	40.24	38.61

〈표 2〉 AVL 트리와 이진 탐색 트리의 성능 비교

입력 셀 개수	출력 셀 개수 (평균)	AVL 트리 깊이	실행시간 (단위: 초)			
			AVL rand	AVL skew	BS rand	BS skew
10 M	1.11 M	12~13	63.69	43.53	66.87	500.51
1 M	111 K	11~12	5.10	4.24	5.24	50.38
100 K	11.1 K	9~10	0.50	0.46	0.53	0.96
10 K	1200	6~7	0.05	0.05	0.05	0.05

〈표 3〉 예제 집계 연산 전 과정을 위한 집계 시간

AVL 트리 크기	AVL 트리 깊이	실행 시간 (단위: 초)			
		AVL skew	AVL rand	BS skew	BS rand
10,000	13~14	38.58	42.03	315.12	42.27
1,000	9~10	38.38	41.37	84.09	41.95
100	6~7	38.50	41.81	44.64	41.72
10	3~4	38.95	40.24	37.90	40.74

skew 되어 있는 경우 더욱 나은 성능을 보인다는 것을 알 수 있다.

5. 결 론

본 연구에서는 스트림 데이터의 다차원 분석에 필요한 큐브를 메모리 효율적이면서 고속으로 계산하는 방법을 제시하였다. 스트림 데이터 큐브 계산이 기존의 비즈니스 데이터 큐브 계산과 크게 다른 점은 스트림 데이터가 정렬되어 입력되지 않으며 단 한 번의 스캔만이 허용된다는 점이다. 이러한 데이터를 고속으로 집계하기 위해서는 배열과 같이 데이터 값으로 집계될 곳의 위치를 찾을 수 있어야 한다. 그러나 희박할 가능성이 높은 여러 개의 집계 테이블을 배열 형태로 유지하면서 집계해 간다면 메모리의 큰 낭비를 초래할 뿐 아니라 알고리즘 자체가 수행되지 못할 수도 있다. 본 연구에서는 집계 테이블의 저장 공간을 최소로 유지하면서 집계할 장소를 신속하게 찾을 수 있도록 하기 위해 배열과 AVL 트리를 혼용한 자료구조에 집계 테이블을 저장하도록 하였다.

실험을 통해 집계 테이블의 생성 시간이 AVL 트리의 깊이에 크게 좌우되지 않는다는 것을 보였다. 예를 들어, 10 M개의 입력 스트림 데이터를 집계하는 과정에서 AVL 트리마다 100,000개의 노드를 유지하는 경우의 실행시간은 수 백개의 노드를 유지하는 경우에 비해 2초 정도 더 걸릴 뿐이다. 또한 입력 스트림 데이터가 임의 순서인 경우 AVL 트리를 사용하는 경우와 이진 탐색 트리를 사용하는 경우 실행시간에 차이를 거의 볼 수가 없다는 점을 볼 때 AVL 트리의 유지 비용이 큐브 계산 전체에 드는 비용 중에 거의 무시할 수 있다는 것을 알 수 있다. 제안한 알고리즘은 메모리가 부족한 경우에도 적응적으로 변형하여 사용할 수 있다.

참 고 문 헌

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. "Models and Issues in Data Streams," *In Proc. ACM Symp. on Principles of Database Systems*, pp.1-16, June 2002.

[2] Stanford Stream Data Management (STREAM) Project. http://www-db.stanford.edu/stream.

[3] S. Babu and J. Widom. "Continuous Queries Over Data Streams," *In Proc. ACM SIGMOD Record*, Vol.30, pp.109-120, 2001.

[4] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebracker, N. Tatbul, and S. Zdonik. "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, Vol.12, pp.120-139, 2003.

[5] C. Cranor, T. Johnson, O. Spatscheck, and V. Shkapenyuk, "Gigascope: A stream database for network applications," *In Proc. ACM SIGMOD*, pp.647-651, 2003.

[6] S. Agarwal, R. Agrawal, P. M. Deshpande, A. Gupta, J. F. Naughton, R. Ramacrishnan, S. Sarawagi, "On the Computation of Multidimensional Aggregates," *In Proc. of the 22nd VLDB Conference*, pp.506-521, 1996.

[7] B. J. Han, Y. Chen, G. Dong, J. Pei, B. W. Wah, J. Wang, Y. D. Cai, "Stream Cube: An Architecture for Multi-Dimensional Analysis of Data Streams," *Distributed and Parallel Databases*, Vol.18, pp.173-197, 2005.

[8] R. Zhang, N. Koudas, B. C.Ook, D. Srivastava, "Multiple Aggregations Over Data Streams," *In Proc. ACM SIGMOD*, pp.299-310, 2005.

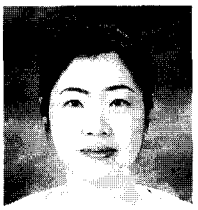
[9] Y. Sismanis, A. Deligiannakis, N. Roussopoulous, Y. Kotidis, "Dwarf: Shrinking the PetaCube," *In Proc. ACM SIGMOD*, pp.464-475, 2002.

[10] 김명, 송지숙, "효율적인 큐브 생성 방법," 한국정보과학회 논문지(데이터베이스), 제29권 2호, pp.99-109, 2002.

[11] M. Datar, A. Gionis, P. Indyk, and R. Motwani. "Maintaining stream statistics over sliding windows," *In Proc. of the 2002 Annual ACM-SIAM Symposium on Discrete Algorithms*, pp.635-644, 2002.

[12] S. Guha and N. Koudas, and K. Shim. "Data-streams and histograms," *In Proc. of the 2001 Annual ACM Symposium on Theory of Computing*, pp.471-475, 2001.

[13] Y. Zhao, P. Deshpande, and J. Naughton, "An Array-Based Algorithm for Simultaneous Multidimensional Aggregates," *In Proc. ACM SIGMOD*, pp.159-170, 1997.



김 지 현

e-mail : jhrosa@ewhain.net

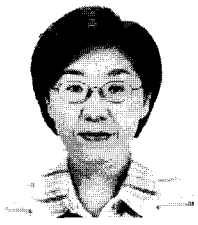
1995년 상명대학교 정보과학과(학사)

2000년~2004년 (주)한진정보통신주식회사 근무

2007년 이화여자대학교 컴퓨터학과(공학석사)

현 재 이화여자대학교 컴퓨터학과 박사과정

관심분야 : 지식공학, 스트림 데이터처리, 온톨로지 등



김 명

e-mail : mkim@ewha.ac.kr

1981년 이화여자대학교 수학과(학사)

1983년 서울대학교 계산통계학과 (이학석사)

1990년 미네소타대학교 컴퓨터학과 (공학석사)

1993년 캘리포니아 주립대학교(산타바바라) 컴퓨터학과 (공학박사)

1993년~1995년 캘리포니아 주립대학교(산타바바라) Postdoc, 강사

1995년~현 재 이화여자대학교 컴퓨터학과 교수

관심분야 : 지식공학, 스트림 데이터처리, 온톨로지, 고성능컴퓨팅, 등