시공간 데이타웨어하우스를 위한 힐버트큐브
(Hilbert Cube for Spatio-Temporal Data Warehouses)

최 원 익 † 이 석 호 ††
(Wonik Choi) (Sukho Lee)

요 약 최근 시공간 데이타에 대한 OLAP연산 효율을 증가시키기 위한 여러 가지 연구들이 행하여지고 있다. 이들 연구의 대부분은 다중트리구조에 기반하고 있다. 다중트리구조는 공간차원을 색인하기 위한 하나의 R-tree와 시간차원을 색인하기 위한 다수의 B-tree로 이루어져 있다. 하지만, 이러한 다중트리구조는 높은 유지비용과 불충분한 질의 처리 효율로 인해 현실적으로 시공간 OLAP연산에 적용하기에는 어려운 점이 있다.

본 논문에서는 이러한 문제를 근본적으로 개선하기 위한 접근 방법으로서 힐버트큐브(Hilbert Cube, H-Cube)를 제안하고 있다. H-Cube는 집계질의(aggregation query) 처리 효율을 높이기 위해 힐버트 곡선을 이용하여 셀들에게 완전순서(total-order)를 부여하고 있으며, 아울러 전통적인 누적합(prefix-sum) 기법을 함께 적용하고 있다. H-Cube는 대상공간을 일정한 크기의 셀로 나누고 그 셀들을 힐버트 값 순서로 저장한다. 이러한 셀들이 시간순서로 모여 큐브형태를 이루게 된다. 또한 H-Cube는 시간의 흐름에 따라 변화되는 지역적인 데이타 편중에 대처하기 위해 적응적으로 셀을 정제한다. H-Cube는 정적인 공간 차원에서 움직이는 점 객체에 초점을 두고 있는 적응적이며, 완전순서화되어 있으며, 또한 누적합을 이용한 셀 기반의 색인구조이다. 본 논문에서는 H-Cube의 성능 평가를 위해서 다양한 실험을 하였으며, 그 결과로서 유지비용과 질의 처리 효율성면 모두에서 다중트리구조보다 높은 성능 향상이 있음을 보인다.

키워드 : 시공간 데이타웨어하우스, 시공간 색인구조, 힐버트큐브, 집계질의

Abstract Recently, there have been various research efforts to develop strategies for accelerating OLAP operations on huge amounts of spatio-temporal data. Most of the work is based on multi-tree structures which consist of a single R-tree variant for spatial dimension and numerous B-trees for temporal dimension. The multi-tree based frameworks, however, are hardly applicable to spatio-temporal OLAP in practice, due mainly to high management cost and low query efficiency.

To overcome the limitations of such multi-tree based frameworks, we propose a new approach called Hilbert Cube(H-Cube), which employs *fractals* in order to impose a total-order on cells. In addition, the H-Cube takes advantage of the traditional *prefix-sum* approach to improve query efficiency significantly. The H-Cube partitions an embedding space into a set of cells which are clustered on disk by Hilbert ordering, and then composes a cube by arranging the grid cells in a chronological order. The H-Cube refines cells adaptively to handle regional data skew, which may change its locations over time. The H-Cube is an adaptive, total-ordered and prefix-summed cube for spatio-temporal data warehouses. Our approach focuses on indexing dynamic point objects in static spatial dimensions. Through the extensive performance studies, we observed that The H-Cube consumed at most 20% of the space required by multi-tree based frameworks, and achieved higher query performance compared with multi-tree structures.

Key words : Spatio-Temporal Data Warehouses, Spatio-temporal index structure, Hilbert Cube, aggregation query

· 이 논문은 2003년도 두뇌한국21사업과 정보통신부의 대학 IT연구센터(ITRC)에 의하여 지원되었음
† 학생회원 : 서울대학교 전기컴퓨터공학부
styxii@db.snu.ac.kr
†† 종신회원 : 서울대학교 전기컴퓨터공학부 교수
shlee@comp.snu.ac.kr
논문접수 : 2003년 3월 24일
심사완료 : 2003년 8월 13일

1. 서 론

시공간 데이타베이스(Spatio-temporal databases)는 시간에 따라 공간적인 위치 또는 모양이 변할 수 있는 객체를 표현하고 저장하며 이들 객체에 대해 질의를 처리한다. 객체의 기하학적이고 시변적인 특징으로 인해

시공간 데이타베이스는 막대한 양의 시공간 데이타를 축적해왔다. 전통적인 관계데이타베이스의 OLAP(on-line analytical processing)과 데이타마이닝(data mining)의 성공에 맞추어 시공간 데이타베이스 영역에서도 시공간 데이타웨어하우스를 구축하고 요약화된 시공간 정보를 효율적으로 추출하기 위해 시공간 OLAP연산을 지원하기 위한 연구의 중요성이 대두되고 있다. 가까운 미래에 시공간 데이타웨어하우스는 교통관제시스템, 물류시스템, 디지털전장시스템(digital battle fields) 그리고 모바일 전자상거래(mobile e-commerce)등과 같은 다양한 응용을 위한 의사결정 시스템의 기반기술로서 필수불가결한 역할을 하게 될 것이다.

지난 수년간 GPS 그리고 무선통신기술의 지속적이고 빠른 발전은 움직이는 객체의 개별적인 위치를 추적하거나 기록하는 일을 가능케 하였다. 이와 더불어 이동객체를 색인하고 각종 시공간 질의들을 처리해야할 필요성이 이동객체데이타베이스(Moving Object Databases) 응용의 광범위한 영역에서 대두되고 있다. 예로서 교통관제시스템을 생각해보자. 이 교통관제시스템은 교통정체감지, 최적경로안내 등의 서비스 제공을 목표로 차량의 움직임이나 교통량 등을 관찰하는 시스템이다. 이 시스템에서 각 차량은 GPS를 장착하고 있으며 이 장치를 통해 자신의 위치를 파악하고 무선통신망을 통해 중앙서버에게 일정한 주기로 자동적으로 위치정보를 전송한다. 이렇게 전송된 위치정보들은 서버에 축적되며 막대한 양의 정보들에 대해 분석을 요하는 질의를 효율적으로 처리하기 위해서는 시공간 데이타웨어하우스 구축이 필수적이다. 이 시스템에서의 주된 질의는 그 결과로서 각 차량의 ID를 요구하기보다는 어떤 특정 시간간격동안 특정 지역 내에 있던 차량의 수들과 같은 요약된 정보(aggregated value)들을 요구하게 된다. 또한 이 시스템은 유지비용을 최소화하며 질의에 대한 응답 역시 최소화해야 한다.

시공간 데이타에 대한 OLAP연산의 중요성에도 불구하고 현재 진행된 연구는 한정적이다. 이 연구의 대부분은 다중트리구조(multi-tree structure)[1]를 기반으로 하고 있다. 다중트리구조란 다음과 같이 설명할 수 있다. 공간차원을 색인하기 위해서 R-tree[2]의 변종인 R^*-tree[3](이하 R-tree로 칭함)를 이용한다. 다중트리구조에서 R-tree는 단 1개만이 필요하며 전집계(pre-aggregation)기법[4]이 적용되어 있다. 시간차원에 대해서는 다수의 B^+-tree(이하 B-tree로 칭함)들을 이용하여 시간정보를 유지하고 있다. 보다 구체적으로 기술하자면, R-tree의 각 엔트리는 그 엔트리에 대한 시간정보를 유지하고 있는 B-tree에 대한 포인터를 가지고 있다. 이러한 다중트리구조는 다음과 같은 단점이 있다.

첫째, 다중트리구조는 하나의 R-tree와 다수의 B-tree들의 조합으로 인해 그 유지비용이 너무 크다. 예를 들면 R-tree가 100,000개의 엔트리가 있다고 한다면 B-tree가 100,000개가 유지되어야 한다. 이 경우 R-tree가 갱신될 때 변경되는 엔트리의 수만큼의 B-tree도 역시 갱신되어야 한다. 더욱이 두 시간스탬프사이에 아주 적은 개수의 객체가 이동하여도 전체 R-tree가 갱신될 수 있다는 사실은 매 시간스탬프마다 전체 엔트리의 개수에 해당하는 모든 B-tree들도 함께 갱신되어야 한다는 것을 의미하며 이는 시스템에 막대한 양의 오버헤드를 일으키게 된다.

둘째, 다중트리구조는 질의 수행과정에서 많은 디스크 접근을 필요로 하며 이러한 사실은 다중트리구조를 이용한 온라인(on-line) 질의 처리가 불가능하다는 것을 의미한다. R-tree는 데이타분할(data-partitioning) 방식의 구조적인 특성상 잠재적으로 많은 오버랩(overlap)이 발생할 수 있으며 그에 따라 이러한 오버랩들은 방문해야 할 B-tree의 개수를 급격히 증가시키게 된다. 더욱이 시간이 진행됨에 따라 즉, 더 많은 객체가 삽입되고 갱신되는 과정에서 R-tree의 성능이 저하된다는 사실은 이러한 문제점을 더욱 악화시키게 된다.

셋째, 다중트리구조는 막대한 양의 저장장치를 필요로 한다. 예를 들어 10,000개의 구역(즉, 10,000개의 엔트리)에 대해 1,000시간스탬프 동안 진화된 다중트리구조의 크기는 약 100MB[1]에 달한다. 이 경우 B-tree의 크기는 약 10KB이었다. 만약 R-tree에 200,000개의 엔트리가 존재하고 B-tree의 크기가 50KB라면 필요한 저장장치의 양은 10GB에 이른다. 이러한 사실은 시간이 지남에 따라 다중트리구조의 크기가 급격히 증가된다는 것을 말해주고 있으며 따라서 실제 응용에 적용되기에는 어렵다는 사실을 말해주고 있다.

종합적으로 볼 때 이러한 사실들은 시공간 데이타웨어하우스 영역에서는 다중트리구조와 같은 데이타분할 방식의 구조보다는 공간분할(space-partitioning) 접근 방식이 더 적합하다는 것을 말해주고 있다.

본 논문에서는 이러한 사실을 바탕으로 시공간 데이타웨어하우스를 위한 색인구조로서 H-Cube(Hilbert Cube)를 제안한다. H-Cube는 유지비용과 저장 공간의 최소화와 함께 집합질의(aggregation query), 예를 들어 "시간간격 q_t동안 질의영역 q_s내에 존재한 객체의 총 개수를 검색하라"와 같은 질의를 효율적으로 처리하는 것을 목표로 하고 있다. H-Cube는 적응적이며 완전순서화된 그리고 누적합(prefix-sum)[5] 기법을 적용한 셀 기반의 색인 구조이다. H-Cube는 대상 시공간을 일정크기의 셀로 분할하고 시간 축으로 배열하여 큐브형태를 이루고 있다. H-Cube의 각 셀들은 그 중심좌표에

대한 힐버트 값이 부여되고 그 값 순서대로 디스크에 저장된다. 특히 이 셀들이 담고 있는 집계값(aggregated value)들은 누적합 형태로 저장되어 질의의 시간간격의 길이에 독립적으로 상수시간에 원하는 집계값을 계산할 수 있다. 이러한 H-Cube의 셀들은 기본적으로 형태가 변화하지 않으며 대신, 데이타가 편중되는 셀들에 한해서 적응적으로 정제하는 기법을 제공하고 있다.

H-Cube의 설계의 근간은 공간분할방법 측면에서는 사분트리(quadtree)[6,7]의 장점을, 그리고 접근방법 측면에서는 격자화일(gridfile)[7,8]의 장점만을 결합하고자 하는 것이며 아울러 이를 바탕으로 힐버트 곡선 및 전집계기법의 적용으로 한층 강화된 구조를 제안하고자 하는 것이다. 본 논문에서는 이 H-Cube가 다중트리구조 보다 더 시공간 데이타웨어하우스에 적합하다는 것을 보이고자 한다. H-Cube의 주된 특징을 요약하면 다음과 같다.

- H-Cube는 적응적이며 완전순서화된 셀 기반 색인 구조를 시공간 데이타웨어하우스에 적용하고자하는 최초의 시도이다.
- H-Cube는 다중트리구조가 필요로 하는 저장 공간의 20%미만의 저장 공간을 요구한다.
- H-Cube는 힐버트 곡선과 누적합기법을 적용하고 있어 다중트리구조에 비해 뛰어난 질의처리 성능을 보여준다.

논문의 구성은 다음과 같다. 제2장에서는 관련 연구를 설명하며 단점과 문제점을 분석한다. 제3장에서는 H-Cube의 구조 설명과 함께 갱신과 질의 처리과정을 설명하며 제4장에서는 H-Cube의 성능평가 결과를 제시한다. 마지막으로 본 논문의 결론과 향후 연구 방향을 제5장에서 서술한다.

2. 관련 연구

이 장에서는 전통적인 OLAP기법과 함께 정적인 공간 차원에서 시공간 데이타를 다루기 위한 다중트리구조에 대해 살펴본다.

전통적인 데이타웨어하우스는 개념적으로 데이타큐브[9]로 모델링할 수 있다. 데이타큐브는 다차원데이타베이스의 속성(attribute)중 일부분 또는 전체 속성으로 구성된다. 특정 속성들은 차원(dimensions) 또는 기능적 속성(functional attributes)으로 선택되는데 이는 속성의 도메인을 기술하고 있다. 기능적 속성의 예로 "상품"와 "상점" 등을 들 수 있다. 한편 또 다른 속성들은 측정치 속성(measure attributes)으로 선택되어지는데 이 측정치 속성이 바로 우리가 관심을 두는 값이 된다. 예를 들면 "판매량", "재고량" 등을 들 수 있다. 이러한

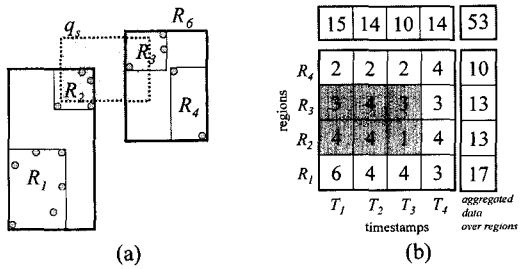

그림 1 간단한 데이타 큐브의 예

전통적인 OLAP모델을 기반으로 시공간 데이타를 데이타큐브로 표현할 수 있다. 기능적 속성은 시간 그리고 공간 차원을 나타내고 측정치 속성은 바로 집계값이 된다. 그림 1은 이러한 간단한 예를 보여주고 있다. 그림 1(a)는 시간스탬프 T_1의 상황을 나타내고 있는데 구역 R_1, R_2, R_3, R_4는 차례로 6, 4, 3, 2개의 객체를 가지고 있다. 시간이 지남에 따라 T_4까지의 상황을 데이타큐브로 나타낸 것이 그림 1(b)이다. 구역 R_1은 시간스탬프 T_1에는 6개의 객체를 가지고 있고 T_2, T_3 그리고 T_4에는 각각 4, 4 그리고 3개의 객체를 갖고 있는 예이다. 마찬가지로 구역 R_4에는 시간간격 [T_1, T_3]동안 계속 2개의 객체가 있었고 T_4에는 4개로 증가되었다는 것을 의미한다.

이 데이타큐브를 디스크에 저장하는 과정을 생각해보자. 이 경우 공간차원에는 완전순서를 생각하지 않았기 때문에 테이블의 열을 하나의 블록으로 모아 시간 순으로 저장할 수밖에 없다. 따라서 테이블 열과 행이 디스크의 여러 페이지에 분산되어 존재하게 되며 이로서 질의 수행과정에서 불필요한 디스크의 탐색시간이 소요된다. 질의 수행과정은 먼저 B-tree를 이용해 질의의 시간간격 q_t를 만족하는 열들을 찾아낸 후, 그 열들을 각각 전체 스캔하면서 그 열의 구역들 중 q_s를 만족하는 구역들을 검색해야 한다. 이 과정은 막대한 비용을 필요로 한다.

또 다른 접근 방법으로서 본 논문에서 제안하고 있는 기법과 같이 정적인 공간 차원에 초점을 두고 있는 기법인 a3DR-tree와 aRB-tree를 살펴본다. a3DR-tree는 aR-tree(Aggregation R-tree)[4]를 시간 축으로 확장함으로써 얻을 수 있다.

aR-tree는 R-tree를 각 노드의 MBR에 속한 객체들에 해당하는 집계값을 노드에 함께 가지고 있는 구조로 강화시킨 것이다. 그림 2는 그림 1(b)의 경우에 해당하는 aR-tree를 보여주고 있다. 그림 2의 aR-tree의 단말 노드 중 첫 번째 엔트리 <R_1, 6>은 시간스탬프 T_1에

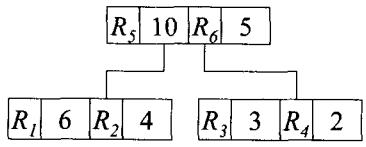

그림 2 aR-tree의 예

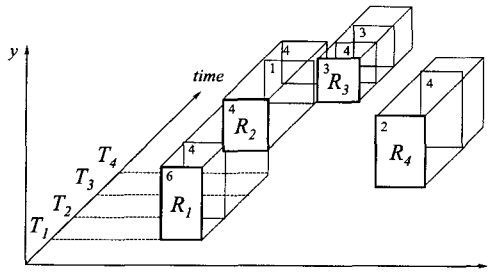

그림 3 a3DR-tree의 예

구역 R_1에 6개의 객체가 있었다는 것을 의미하며 마찬가지로 한 레벨 올라가서 $<R_5, 10>$의 의미는 구역 R_1, R_2에 속한 객체 들이 10개라는 것을 의미한다. 하지만 aR-tree는 시간정보를 유지할 수 있는 기능을 가지고 있지 않으므로 이 기능을 제공하기 위해서는 시간축으로 확장해야 한다. 이 경우 aR-tree의 엔트리 < *MBR, XldPtr, Value*>에 *LifeSpan*이 추가된 < *MBR, XldPtr, Value, LifeSpan*>의 형태가 된다. 여기서 *XldPtr*는 자식 노드를 가리키는 포인터이며 *Value*는 집계값, 그리고 *LifeSpan*는 그 집계값이 유효한 시간간격을 의미한다. 이렇게 aR-tree를 시간 축으로 확장한 구조가 a3DR-tree이다. 그림 3은 그림 1(b)의 데이타큐브의 예를 나타낸 a3DR-tree의 예를 보여주고 있다. 앞서 언급한 바와 같이 a3DR-tree는 공간차원의 정보는 시간에 따라 변하지 않는다고 가정하고 있다. 따라서 a3DR-tree는 *Value*값이 변할 때 마다 새로운 엔트리가 생성되면서 같은 MBR정보들이 매번 반복 저장된다. 이러한 단점으로 인해 색인의 크기가 급격히 늘어나고 질의처리 성능도 함께 저하된다.

Papadias외 3인은 [1]에서 시공간 데이타웨어하우스를 위한 프레임웍을 최초로 제안하였다. 그들은 하나의 R-tree와 그 R-tree의 엔트리 개수만큼의 B-tree들로 이루어진 다중트리구조를 제안하였다.

aRB-tree(aggregate R- B- tree)로 불리는 이 트리는 aR-tree의 각 엔트리에 대한 시간 정보를 B-tree에 저장한다. 즉, aRB-tree의 엔트리는< *MBR, XldPtr, Value, BtreePtr*>의 형태로 구성된다. 여기서 *BtreePtr*가 그 엔트리의 해당 B-tree를 가리키는 포인터이다. 그림 4는 그림 1의 예로 구성한 aRB-tree를 도시하고

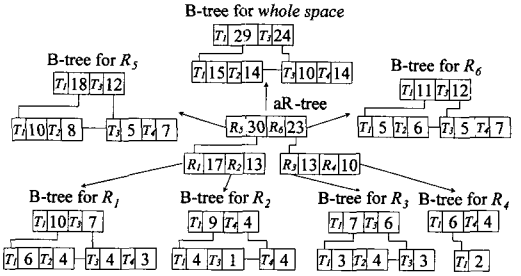

그림 4 aRB-tree의 예

있다.

aR-tree의 구역 R_5에 해당하는 B-tree의 단말 노드의 첫 번째 엔트리 $<T_1, 10>$는 시간스탬프 T_1에 10개의 객체가 존재한 것을 의미한다. 또한 한 레벨 위로 올라가서 루트 노드의 첫 번째 엔트리 $<T_1, 18>$는 시간간격 $[T_1, T_2]$동안 18개의 객체가 존재했다는 것을 의미한다. aR-tree의 루트 노드에 해당하는 B-tree는 전체 공간에 대한 집계값을 유지하고 있다.

aRB-tree의 질의 수행 과정은 다음과 같다: 주어진 질의는 시간간격 $[T_1, T_3]$동안 그림 1(a)에 주어진 q_s 안에 존재한 객체의 개수를 검색하는 것이라 가정하자. 검색은 aR-tree의 루트부터 시작한다. 루트의 엔트리 R_5, R_6 모두 q_s와 오버랩되었기 때문에 그들의 자식 노드를 검색해야 한다. 이들 중에 두개의 엔트리 즉, R_2, R_3이 q_s와 오버랩되었으므로 그 엔트리에 해당하는 B-tree를 방문해야 한다. 먼저 R_2에 해당하는 B-tree를 살펴보자. 이 B-tree의 루트의 첫 번째 엔트리 $<T_1, 9>$는 시간간격 $[T_1, T_3]$동안 9개의 객체가 있었다는 것을 의미한다. 따라서 R_2에 해당하는 결과값은 9가 된다. 마찬가지 방법으로 R_3에 대해 살펴보자. 루트의 첫 번째 엔트리 $<T_1, 7>$은 $[T_1, T_2]$동안 7개의 객체가 존재했다는 것을 의미한다. T_3에 대한 정보를 얻기 위해서는 단말 노드까지 방문해야 한다. 단말 노드의 $<T_3, 3>$에서 T_3에 3개의 객체가 R_3 내에 존재했다는 것을 알 수 있다. 따라서 R_3에 해당하는 결과값은 7+3이 된다. 결국 최종 질의 결과는 이 결과들의 합인 9+7+3=19가 된다. 이 결과는 그림 1(b)에 음영 처리된 셀들의 값을 합한 것과 일치하는 것이다.

위의 예에서 aRB-tree는 질의 수행과정에서 너무 많은 디스크 접근을 필요로 하고 있다는 것을 알 수 있으며 aR-tree의 크기가 커질수록 즉, 엔트리의 개수가 많아질수록 방문해야 될 B-tree의 수는 그 만큼 증가하게 된다. 더욱이 이러한 B-tree들은 디스크 내의 산재된

페이지 내에 저장될 수밖에 없기 때문에 많은 탐색시간을 소요하게 되어 성능의 저하를 가져오게 된다. 또한 aRB-tree는 aR-tree의 엔트리 개수만큼의 B-tree가 필요하기 때문에 시간이 지남에 따라 막대한 양의 저장공간을 필요로 하게 될 것은 자명한 일이다. 이러한 사실은 aRB-tree가 특정 저장 공간 또는 유지비용 제한조건을 위반할 가능성이 높다는 것을 설명한다. 이러한 문제점들은 다중트리구조를 시공간 데이타웨어하우스에 적용하기에는 현실적으로 어렵다는 것을 말해주고 있으며, 새로운 접근 방법이 요구되고 있다는 것을 암시해 주고 있다.

3. 힐버트큐브(H-Cube)

3.1. H-Cube의 구조

이 절에서는 H-Cube를 자세히 소개하고 갱신과 질의 처리 알고리즘을 다룬다. H-Cube는 공간 채움 곡선(space filling curve) 즉, 프랙탈(fractals)을 이용하고 있다. 프랙탈 중 힐버트 곡선(Hilbert curve)을 이용하여 셀들에게 선형적인 순서를 부여하며 이 순서에 맞추어 디스크에 셀들을 저장한다. 힐버트 곡선이 가장 이상적으로 클러스터(cluster)의 개수를 최소화 할 수 있다고 널리 알려져 있다[10,11]. 이러한 특성에 대한 이론적인 분석들이 [11-13]에 나타나 있다. 이 문헌들은 모두 일관성 있게 다른 어떤 공간 채움 곡선보다도 힐버트 곡선이 가장 작은 개수의 클러스터를 만들어낸다고 밝혀내고 있다. 이러한 사실은 디스크의 한 페이지 즉, 셀들에게 힐버트 값(Hilbert value)을 이용하여 순서를 부여하고 그 순서대로 디스크에 저장하게 되면 추가의 탐색시간을 최소화하면서 연속적인 디스크 페이지를 읽음으로써 셀들을 접근 할 수 있다는 것을 말해주고 있다. 이러한 사실을 바탕으로 H-Cube의 셀들은 다음과 같이 정의될 수 있다.

- 셀의 힐버트 값은 그 중심점의 힐버트 값으로 정의된다.
- 셀은 엔트리(entry)들의 집합을 저장하고 있다. 각 엔트리 들은 그 셀에 속한 객체들의 개수를 누적합 형태로 가지고 있다.
- 셀들은 각자의 힐버트 값 순서대로 디스크에 배열된다.

해싱 함수는 수식 (1)과 같이 정의되며 셀 또는 객체의 좌표를 입력으로 하여 힐버트 값을 반환하는 함수이다. 이 함수에서 필요한 힐버트 곡선의 생성 알고리즘은 [11,12,14]에서 찾을 수 있다.

$$f(x, y) = h \qquad (1)$$

여기서 x, y는 2차원 공간상의 좌표이며 h는 힐버트 값을 의미한다. 함수 f를 통해 셀과 같은 힐버트 값을 갖는 객체는 그 셀에 포함되어 있다는 것을 알 수 있다.

H-Cube는 대상 공간을 N개의 고정크기의 셀들로 분할한다. 각 셀들은 앞선 정의와 같이 구성되며 이러한 셀들이 시간 축으로 배열되면서 큐브 형태를 나타내고 있다. 이때 힐버트 값은 2차원 공간상에만 적용되고 있음에 주목하라. 그 이유는 시간 축은 이미 완전순서가 존재하고, H-Cube는 누적합을 이용하여 집계질의를 처리하기 때문에 시간 축으로는 물리적으로 클러스터링되어 있을 필요가 없기 때문이다. 이렇게 공간적인 힐버트 값 순서가 시간 순서와 함께 셀들에게 완전순서(total-order)를 부여함으로써 접근하고자하는 셀들은 상수 시간(constant time)에 접근할 수 있는 메카니즘을 제공하고 있다. 이러한 접근 방법은 집계질의 처리시 많은 성능 이득을 얻게 해주고 있다.

N의 선택은 H-Cube의 공간 이용률과 질의 처리 성능에 중요한 영향을 미친다. Bently외 2인[15]은 셀의 크기와 질의 윈도우의 크기가 같을 때 최적(nearly optimal)임을 보였다. 하지만 대부분의 응용에서 질의윈도우는 크기뿐만이 아니라 그 위치 역시 가변적이므로 이는 셀 크기를 선택할 수 있는 정보로 사용될 수 없다. H-Cube는 다음 두 가지 사항을 N을 정하는 대안으로 제시하고 있다. 첫째는 대상 영역의 객체 밀도이며 둘째는 객체의 최대 속도이다.

대상 영역의 예상 가능한 객체의 개수를 O이라고 하고 하나의 셀에 저장할 수 있는 객체의 개수(예를 들면 객체의 좌표들의 수)를 V이라 한다면 N은 아래와 같은 수식 (2)로 구할 수 있다.

$$N = \left(\left\lceil \sqrt{\frac{O}{V}} \right\rceil \right)^2 \qquad (2)$$

또한 객체의 예상되는 최대 속도가 N을 선택하는데 이용될 수 있다. 즉, 최대 속도를 예상할 수 있다면 두 개의 연속된 시간스탬프사이에 이동할 수 있는 거리를 알 수 있고 이 거리는 셀 크기를 정할 수 있는 중요한 정보로 이용될 수 있다. 왜냐하면 연속된 시간스탬프 사이에 이동할 때 최대 이동할 수 있는 거리와 셀의 크기가 크거나 같으면 그만큼 그 객체는 같은 버켓에 머물러 있을 확률이 높으며 이로서 갱신비용을 줄일 수 있는 방법을 제공할 수 있기 때문이다.

H-Cube의 각 셀들은 물리적으로 디스크의 한 페이지에 해당하며 < *Value* >와 같은 형태의 엔트리들을 저장하게 된다. 여기서 *Value*는 누적합을 나타내고 있다. 일반적으로 누적합 기법은 임의의 위치에서 갱신이 일어날 때 그 위치이후의 모든 값들이 연속적으로 갱신이 되어야 하기 때문에 갱신비용이 크다는 것이 단점으로 지적되어 왔다. 하지만 시공간 데이타웨어하우스와 같이 시간이 지남에 따라 데이타가 추가되는(append only)

형태의 응용에 있어서는 문제가 되지 않는다. H-Cube는 이러한 누적합의 단점은 피하고 상수시간에 집계값을 계산할 수 있는 장점만을 적용하고 있다. 기술의 간편함을 위해 이 논문에서는 집계함수(aggregation function)를 SUM으로 제한하고 있지만 다른 어떠한 분산(*distributed function*) 또는 대수(*algebraic function*)[9] 함수도 쉽게 적용할 수 있다. 이러한 엔트리에 누적합 이외에 공간에 대한 정보 그리고 시간에 대한 정보가 명시적으로 저장되어있지 않음에 주목하라. 셀들에 대한 시공간 정보는 완전순서화된 셀들의 위치와 그 셀 내의 엔트리의 위치로부터 쉽게 얻어낼 수 있으며 아울러 이러한 사실은 주어진 시공간 정보를 통해 해당 셀과 엔트리를 상수시간에 접근할 수 있음을 말해주고 있다.

M을 한 셀이 최대로 저장할 수 있는 엔트리의 개수라 하자. 그러면 하나의 셀은 시간간격 $[T_1, T_M]$에 해당하는 엔트리들을 담고 있게 된다. 이 셀들이 N개가 모인 것을 셀블럭(cellblock)이라 부른다. 이러한 셀블럭들이 시간 축으로 배열되어 H-Cube를 형성하게 된다. 이러한 H-Cube의 구조가 그림 5에 나타나 있다.

특정 엔트리를 나타내고 할 때 $E[h, t]$라고 표시한다. 여기서 h는 힐버트 값이며 t번째 시간스탬프를 의미한다. 이와 비슷하게 특정 셀은 $C[h, s]$로 나타낸다. 이를 이용하여 셀과 셀블럭을 형식적으로 나타내면 아래와 같다.

$$C[h, s] = \bigcup_{i=(s-1)M+1}^{sM} E[h, i]$$

$$B[b] = \bigcup_{i=0}^{N-1} C[i, b]$$

여기서 $1 \le h \le N$, $s \ge 1$ 그리고 $b = 1, M+1, 2M+1, \ldots$을 의미한다. 이를 바탕으로 H-Cube H는 아래와 같이 모델링할 수 있다.

$$H = \bigcup_{i=1}^{n} B[(i-1)M+1]$$

위와 같이 H-Cube는 셀블럭의 집합으로 구성되어 있으며 각 셀블럭들은 특정 시간간격 동안에 해당하는 시공간 데이타웨어하우스 기능을 담당한다. 셀블럭은 물리적으로 연속적인 페이지의 집합에 해당한다. 이때 셀블럭 내의 셀들은 힐버트 순서로 배치된다. 또한 이 셀블럭들은 시간이 지남에 따라 추가되는데 이렇게 새로이 추가되는 셀블럭은 시간 순서에 따라 연속적으로 배치되며 그 결과 셀들은 완전순서화되어 디스크에 저장된다.

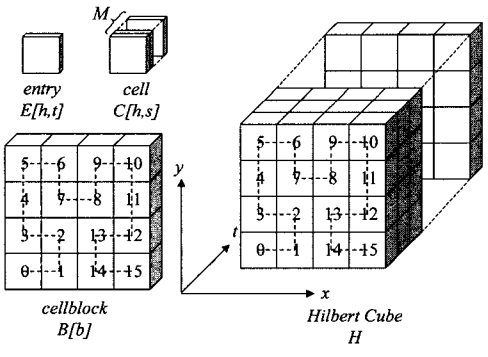

그림 5 H-Cube의 구조

3.2 적응적 셀 분할 기법

이동객체는 동적이다. 또한 그 분포는 예상 불가능하며 편중될 가능성이 크다. 이러한 데이타 편중은 셀의 크기를 증가시키고 따라서 셀 접근시 효율을 떨어뜨리게 된다. 이러한 문제를 다루기 위해 H-Cube는 셀의 객체 수에 따라 적응적으로 셀을 정제한다. 즉 만약 셀 내에 속한 객체의 수가 임계값 V를 초과하면 이 셀들은 모든 서브셀(sub-cell)들이 V개 이하의 객체를 갖게 될 때까지 재귀적으로 분할된다. 통상적으로 임계값 V는 한 페이지에 저장될 수 있는 이동객체의 위치들의 개수와 같게 설정된다. 이 페이지를 버켓(bucket)이라고 부르며 이 버켓에는 질의 요구시 정확한 값이 필수적으로 필요한 경우 실제 결과값을 얻어낼 수 있는 각 객체의 위치정보가 기록되어 있다. 보다 자세히 설명하자면 질의윈도우와 부분적으로 오버랩된 셀들의 경우는 해당 버켓을 추가 접근하여 객체의 위치와 질의윈도우와 비교해서 실제로 그 질의윈도우에 포함된 객체들만을 검색하게 된다. 반면에 질의 결과의 정밀도가 요구되지 않는 상황이라면 질의윈도우에 포함된 셀의 크기 비율에 비례하여 결과값을 추정할 수 있다. 이러한 방법들은 다중트리구조에도 그대로 적용되며 이 경우 기본 테이블(base table)이 버켓에 해당하게 된다. 즉 다중트리구조에서도 정밀한 결과값이 요구될 때는 기본테이블을 추가 접근하여 질의윈도우에 실제로 포함된 객체만을 정제하는 과정이 필요하다. 본 논문에는 공정성을 위해 버켓과 기본테이블 접근은 배제하고 있다.

그림 6은 적응적 셀 분할 기법을 도시하고 있다. 시간스탬프 T_1에는 어떠한 셀도 V값을 넘지 않았기 때문에 분할된 셀을 찾아볼 수 없다. 하지만 T_2에서는 $C[7,1]$이 V값을 넘었고 T_3에서는 $C[13,1]$이 V값을 넘었다. 이에 따라 $E[7,2]$는 4개의 서브셀로 분할된 상황을 그림 6 중간부분에서 보여주고 있다. 마찬가지로 $E[13,3]$은 16개의 서브셀로 분할되었다. 이렇게 셀이 분할된 경우에는 그 엔트리에 누적합이 저장되어있는 것이 아니라 서브큐브(*Subcube*)라 불리는 구조체를 가리키는 포인터를 대신 담고 있다. 서브큐브는 분할된 셀들에 해당하는 각 엔트리들을 역시 힐버트 값 순서로

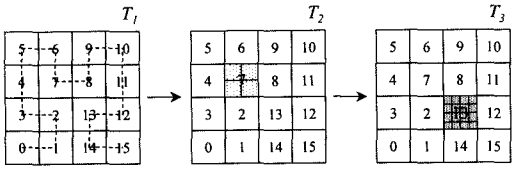

그림 6 적응적 셀 분할 기법

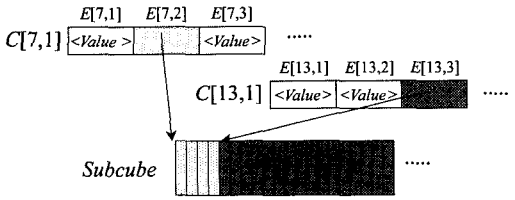

그림 7 서브큐브(Subcube)의 저장 구조

저장하고 있는 연속된 저장 공간을 말한다. 이 서브큐브 구조체는 질의윈도우가 분할된 셀들을 부분적으로 오버랩하고 있는 경우 해당 서브큐브 구조체를 접근하여 결과값을 보다 더 정제할 수 있는 기법을 제공하고 있다. 그림 7은 그림 6의 경우에 해당하는 서브큐브의 저장 구조를 보여 주고 있다.

$C[7,1]$는 시간스탬프 T_1에는 분할되지 않았기 때문에 엔트리 $E[7,1]$는 누적합 < *Value*>를 담고 있다. 반면 $E[7,2]$는 4개의 서브셀로 분할되었기 때문에 이 경우에 $E[7,2]$는 서브큐브를 가리키는 포인터를 담고 있다. 마찬가지 이유로 $C[13,1]$의 $E[13,3]$도 해당 16개의 엔트리는 서브큐브에 저장되어있고 $E[13,3]$은 이 부분을 가리키고 있는 구조를 보여주고 있다. 서브큐브내의 셀의 순서 그리고 셀 내의 엔트리 저장 순서도 힐버트 값 순서를 따르고 있다.

이와 같이 H-Cube의 기본적인 구조는 재귀적인 공간 분할의 원리를 따르고 있다는 점에서 사분트리(quad-tree)와 유사하다. 하지만 사분트리의 기본적인 구조는 주메모리 기반 구조로서 대형 시공간데이타베이스에는 적합하지 않으며 더욱이 사분트리는 불균형 n진 트리를 기반으로 하고 있기 때문에 객체의 밀도가 높은 지역에서는 심한 불균형으로 인해 질의 성능이 저하될 수 있다. 반면 H-Cube는 해싱 방법을 기본 골격으로 한 디스크 기반 구조이며 적응적 셀 분할 기법으로 데이타 편중 문제를 완화시키고 있다.

또 다른 관점으로서 해싱 방법의 관점에서 보면 H-Cube는 격자화일(gridfile)의 구조와 유사하다. H-Cube와 격자화일의 가장 큰 차이점은 격자디렉토리(grid directory)의 유무이다. 격자화일의 가장 큰 단점은 균일한 분포의 데이타에 대해서도 격자디렉토리의 크기가 선형적(super linear growth)[7]로 증가한다는데 있다. H-Cube는 격자디렉토리를 배제하고 힐버트 곡선을 이용한 해싱 함수를 공간적인 접근 경로로 택하고 있으며 이로서 색인 구조의 유지비용을 최소화하고 있다. 서론에서 언급한 바와 같이 H-Cube의 설계 근간은 사분트리와 격자화일의 기본 개념을 단점 없이 결합하는 것이다.

다차원 색인 구조가 가져야할 중요한 특성 중에 하나는 데이타 분포에 독립적인 성능을 유지해야 하는 것이다[14]. 이러한 관점에서 R-tree와 같은 계층 기반 구조가 해싱 방법 보다 편중된 데이타 입력에 더 효율적으로 대처하는 것은 사실이다. 동시에 계층 기반 구조의 노드가 해싱 방법에서의 버켓보다 공간이용률이 더 높다는 것도 지적할 만 하다. H-Cube 역시 이러한 단점을 내포하고 있다. 하지만 그 영향을 앞서 설명한 힐버트 곡선 및 누적합의 적용, 그리고 적응적 셀 분할 기법을 통해 완화하고 있다. 4장에서 다양한 실험결과를 통해 적응적 셀 분할 기법이 심하게 편중된 데이타에 대해서도 충분히 대응할 만한 성능 결과를 보여주고 있으며 공간요구량도 다중트리구조에 비해 훨씬 적음을 보일 것이다.

3.3 H-Cube의 갱신과 질의 알고리즘

시공간 데이타웨어하우스를 이야기할 때 자주 간과되는 관점은 갱신비용이다. 시공간 데이타베이스 내의 원자료(raw data)가 변경되면 이들 변경사항은 시공간 데이타웨어하우스에 전달되어야 한다. 일반적으로 이 전달 과정은 성능과 안전상의 이유로 온라인 상태에서 실행되지 않는다. 변경사항이 웨어하우스에 전달되어 재조직 과정이 수행될 필요가 있을 경우에는 웨어하우스는 일정시간 오프라인 상태로 전환된 후 재조직 과정이 수행되는 것이 일반적이다. 따라서 웨어하우스의 전체적인 효율을 높이기 위해서는 이러한 오프라인 기간을 최소화하고 재조직 비용 역시 최소화할 필요가 있다. 많은 OLAP응용에서 바로 이러한 이유로 인해 질의 성능의 희생 없이 웨어하우스 갱신 비용을 최소화하는 것이 중요한 이슈가 되고 있으며 H-Cube역시 이러한 비용 최소화를 목표로 하고 있다. H-Cube의 구축 및 갱신 과정은 직관적이며 매우 효율적으로 이루어져 있다. 의사코드(pseudo code)가 그림 8에 나타나 있다.

우선, 수식 (1)의 함수 f를 이용하여 힐버트 값을 구한다(그림 8의 7행). 그리고 그 힐버트 값에 해당하는, 즉 그 셀에 속한 객체들의 개수를 누적합 형태로 해당 엔트리에 저장한다(그림 8의 8행). 그리고 필요하다면 버켓에 객체의 좌표를 저장한다(그림 8의 9행). 누적합은 아래 식으로 구할 수 있다.

$$E[h,t] = \sum_{i=1}^{t} A[h,i]$$

```
Algorithm Update_H_Cube(S_t)
1   Input:
2      S_t: 시간스탬프 t의 객체 집합
3   Begin
4      set ∀i E[i,t] ←0, (1 ≤ i ≤ N);
5      set ∀i bucket[i] ← empty, (1 ≤ i ≤ N);
6      For each Object ∈ S_t Do // O_x, O_y는 객체의 좌표
7            h = f(O_x, O_y);
8            E[h,t]++;
9            bucket[h]← Object;
10     End For
11     For each E[i,t](1 ≤ i ≤ N) Do
12        If E[i,t] > V Then
13           PartitionCell(bucket[i]);
14     End For
15  End
```

그림 8 H-Cube의 갱신 알고리즘

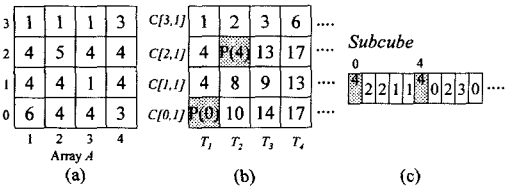

그림 9 H-Cube의 예

즉, 엔트리의 누적합은 지금까지의 개수와 현재 시간 스탬프의 개수를 모두 합한 값이 된다. 위 식에서 행렬 A는 그림 9(a)에 나타나 있으며 이는 그림 1(b)에 해당하는 배열이다. 이에 해당하는 누적합 행렬(prefix-sum array)이 그림 9(b)에 나타나 있다.

3.2절에서 언급한 바와 같이 임계값 V를 넘은 셀들은 적응적으로 정제된다. 이러한 작업은 그림 8의 14행에 있는 *PartitionCell*() 에서 행하여진다. 이 함수는 V를 넘는 셀들에 대해 V를 넘지 않을 때까지 분할하고 그 분할된 서브셀의 엔트리 들을 역시 힐버트 값 순서로 서브큐브에 덧붙인다. 그리고 기본 셀의 엔트리에는 그 서브큐브의 옵셋을 가리키게 한다. 예를 들어 V=4라고 가정하자. 이때 T_1의 $E[0,1]$은 6을 가지고 있기 때문에 분할이 일어나고 서브큐브의 옵셋 0을 가리키게 된다. $E[0,1]$의 P(0)이 이를 나타내고 있다. 마찬가지로 $E[2,2]$도 5이기 때문에 P(4)라는 포인터를 가지고 있다.

완전순서화된 셀 기반의 구조인 H-Cube는 집계 질의 처리과정 또한 효율적으로 이루어진다. 더욱이 누적합 형태로 구성된 H-Cube는 질의로 주어진 시간간격의 양 끝에 해당하는 엔트리만 접근해서 원하는 결과를 얻어낼 수 있다. 즉 시간간격의 끝 엔트리의 누적합에서 시작 엔트리의 누적합을 빼면 그 셀의 SUM값을 얻을 수 있다. 이를 식으로 표현하면 아래 식 1과 같다.

$$CellSum(h,i,j) = E[h,j] - E[h,i-1] \quad (3)$$

예를 들어, 그림 9에서 시간간격 $[T_2, T_4]$동안의 $C[3,1]$의 SUM값을 알고 싶다면 $E[3,4]$에서 $E[3,(2-1)]$를 빼면 된다. 즉, $CellSum(3,2,4) = E[3,4] - E[3,1]$가 되며 결과는 6-1=5가 된다.

이제 H-Cube의 집계질의 처리 알고리즘을 알아보자. 알고리즘은 기본적으로 두 단계로 이루어져 있다. 첫 번째 단계에서는 다음과 같이 집계질의의 중간 결과(intermediate result)값이 계산된다. 먼저 q_s와 오버랩되는 모든 셀들을 찾아내 시간간격 q_t에 해당하는 값들을 수식 (3)을 이용하여 계산한 후 모두 합하면 중간 결과값을 얻을 수 있다. 이러한 중간 결과값들은 아주 정확한 결과가 필요하지 않은 경우와 신속한 응답이 필요한 경우 매우 유용한 결과이다. 이 중간 결과값들은 두 번째 단계에서 점진적으로 정제되면서 보다 정확한 결과를 계산할 수 있다. 이 과정에서 서브큐브가 이용되는데 정리 1에 의해 서브큐브를 이용하여 정제되어야 할 셀들을 선택할 수 있다.

정리 1 임계값 V과 시간간격 q_b 즉, $[T_i, T_j]$가 주어져 있다고 가정한다면 어떠한 $C[h,s]$에 대해 다음과 같은 식이 성립한다.

$$CellAvg(h,i,j) \geq V \rightarrow (\exists k \; E[h,k] \geq V)$$

여기서 $CellAvg(h,i,j) = \frac{CellSum(h,i,j)}{j-i+1}$, $j \geq i$ 그리고 $i \leq k \leq j$ 이다. 역은 성립하지 않는다.

증명. $C[h,s]$의 모든 엔트리의 평균값이 임계값 V보다 작지만, $CellAvg(h,i,j)$는 임계값 V보다 크다고 가정하자. 이 가정으로부터 다음 식을 얻는다.

$$\begin{aligned} \forall k \; E[h,k] &= E[h,k] - E[h,k-1] \\ &= CellSum[h,k,k] < V \end{aligned}$$

여기서 $i \leq k \leq j$ 이다. 이러한 엔트리를 모두 합하면 아래와 같이 표현된다.

$$\sum_{k=i}^{j} CellSum(h,k,k) < V(j-i+1) \quad (4)$$

$CellAvg(h,i,j)$의 분모를 수식 (4)로 치환하면 다음 식을 얻을 수 있다.

$$CellAvg(h,i,j) = \frac{\sum_{k=i}^{j} CellSum(h,k,k)}{j-i+1} < V$$

따라서 이는 가정에 위배된다. 그러므로 만약 $CellAvg(h,i,j)$가 V가 크거나 같다면 그 셀은 적어도 하나 이상의 V보다 큰 엔트리를 가지고 있다는 것을 알 수 있다. □

모든 셀들은 아래 세 가지 경우에 따라 집계값을 계산할 수 있다.

① 질의윈도우에 완전히 포함된 셀은 수식 (3)에 의해 집계값을 얻을 수 있다. 이 경우 서브큐브를 방문

할 필요 없다.

② 질의윈도우와 부분적으로 오버랩된 셀은 다시 아래의 두 가지 경우로 나뉜다.

- 정리 1을 만족하지 않는다면 서브큐브는 방문할 필요가 없이 ①의 경우와 같이 계산한다.
- 정리 1을 만족한다면 해당 서브큐브를 방문해서 결과를 정제한다.

③ 그 외의 경우 그 셀은 방문하지 않는다.

첫 번째 단계에서 질의윈도우 q_s와 부분적으로 오버랩하고 정리 1을 만족하는 셀만이 큐에 삽입된다. 이렇게 큐에 삽입된 셀들은 두 번째 단계에서 차례대로 삭제되면서 결과를 정제하게 된다.

그림 10은 질의 처리과정의 예를 보여주고 있다. 질의윈도우는 그림 1의 q_s를 사용하기로 한다. 임계값 V는 4이며 질의의 시간간격 q_t은 $[T_1, T_3]$이라고 가정한다. 먼저 함수 f를 이용하여 질의윈도우 q_s와 겹쳐있는 셀들을 구한다. 이는 질의윈도우의 네 개의 꼭지점의 좌표와 함수 f를 이용하여 쉽게 구할 수 있다. 이 예의 경우 질의윈도우 q_s의 네 꼭지점에 대해 f가 반환한 값

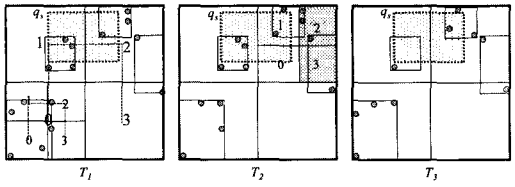

그림 10 H-Cube의 집계질의 처리 예

```
Algorithm AggregationQuery(q_s, q_t)
1   Input:
2     q_s: 질의윈도우
3     q_t: 질의의 시간간격 [T_i, T_j]
4   Output:
5     A : 집계값(aggregated value)
6   Begin
7     set Queue Q ← empty;
8     set A←0;
9   // Stage 1 : Compute intermediate answer.
10    For each C[h,t] intersects q_s Do
11      A += CellSum(h,i,j);
12      If C[h,t] partially overlaps q_s &&
13        CellAvg(h,i,j) ≥ V then
14          Q.insert(C[h,t]);
15      End If
16    End For
17  // Stage 2 : Refine the answer.
18    While Q is not empty Do
19      C[h,t] = Q.pop();
20      For each E[h,k] ∈ C[h,t] Do
21        If E[h,k] points to Subcube Then
22          A -= Refine(E[h,k]);
23        End If
24      End For
25    End While
26  End
```

그림 11 H-Cube의 집계 질의 처리 알고리즘

은 1과 2가 되며 이는 두 개의 셀 즉 $C[1,1]$과 $C[2,1]$과 오버랩하고 있다는 사실을 말해준다. 이 두 셀들의 해당 엔트리들과 수식 (3)을 이용하여 중간 결과값을 계산할 수 있다. $C[1,1]$의 경우 $E[1,3]-E[1,0]$을 계산하면 9-0=9를 얻을 수 있다. 마찬 가지로 $C[2,1]$는 13-0=13을 얻을 수 있기 때문에 중간결과값은 이 둘을 합한 22가 된다.

이 첫 번째 단계에서 $CellAvg(2,1,3)$즉 13/3=4.3으로 임계값 4보다 크기 때문에 $C[2,1]$는 큐에 삽입된다. 이어서 두 번째 단계가 진행되는데 이 과정에서 서브큐브를 방문하면서 중간 결과값을 점진적으로 정제한다. 큐로부터 $C[2,1]$을 삭제하여 엔트리들을 살펴보면 그림 10의 중간부분의 그림에서 나타나 있듯이 $E[2,2]$가 4개의 셀로 분할된 것을 알 수 있다. 이 4개의 셀 중 2번과 3번 셀 즉, 그림에서 음영처리 된 두개의 셀은 질의윈도우 q_s와 오버랩되어 있지 않기 때문에 2번과 3번 셀에 해당하는 값은 중간 결과값에서 감해주어야 한다. 따라서 22-3=19가 최종 결과값이 된다. 이 작업은 그림 11의 22행에 있는 *Refine*()함수에서 이루어지며 아울러 전체적인 집계질의 처리 알고리즘은 그림 11에 기술되어 있다.

4. 성능 평가

4.1.실험데이타 및 실험 환경

이 절에서는 성능 평가에 앞서 실험에 쓰인 데이타셋(data set)과 실험 환경에 대해 먼저 설명한다. 데이타셋은 GSTD[16]를 이용하여 다음 시나리오에 따라 생성하였다: 객체들은 초기에 대상공간의 좌하단 모서리에 집중되어 모여 있다가 점차 우상단 모서리 쪽으로 이동한다. 이동 과정 중에 객체의 분포는 초기에 편중(skew)된 분포에서 균일(uniform) 분포로 점자 변해 가다가 다시 편중된 분포로 변하게 된다. 실세계의 현상을 최대한 반영하기 위해 객체들은 각 시간스탬프에 모두 움직이지 않는다고 가정한다. 즉 각 시간스탬프마다 일정 분 율의 객체만이 자신의 위치를 변경하게 하였다. 이를 객체활동률(object activity) α_{obj}로 정의하며 전체 객체 중 α_{obj}%만이 이동하였다는 의미를 가진다. 데이타분포는 Zipf인자(Zipf parameter) z로 표기된다. z=0은 멱함수(power-low function) $P_i \sim \frac{1}{i^z}$의 지수를 나타낸다. z=0일 경우 균일분포를 나타내며 z가 커질수록 더 큰 편중도(skewness)를 보이게 된다. 데이타셋과 작업부하(workloads)를 조절하는 4가지 주요한 인자를 정리하면 표 1과 같다

표 1 데이타셋과 작업부하의 표기법

인자	설명	값
α_{obj}	객체활동률(object activity)	1, 10, 20, 30, 40, 50
z	편중도(skewness)	0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0
q_s	질의윈도우	전체면적의 0.01%, 0.05%, 0.25%, 0.5%, 1%, 5%, 10%
q_t	시간간격	1, 25, 50, 75, 100 시간스탬프

객체의 개수는 100,000개이며 1,000시간스탬프동안 이동한 결과를 이용하였다. 데이타셋을 구별하기 위해 $D[\alpha_{obj}, z]$로 나타내며 작업부하는 $Q[q_s, q_t]$로 표기한다. 각 작업부하는 1,000개의 질의로 구성되었으며 평균 페이지접근수로 측정한다. aRB-tree의 경우 단말노드는 10,000개로 하였으며 100,000개의 객체가 중복 없이 10,000개의 MBR에 속하도록 개수를 센 후 각 노드의 집계값으로 설정하였다.

실험은 Solaris 2.7이 운영되는 Sun UltraSPARC-II 296MHz에서 이루어졌으며 이 워크스테이션은 512MB의 메모리와 SCSI2방식의 18GB의 디스크가 장착되었다. 이 디스크의 탐색시간은 8.9ms이었다. 운영체제의 버퍼 효과를 배제하기 위해 Solaris운영체제의 직접 I/O(direct I/O)기능을 이용하였다. 실험에 쓰인 페이지의 크기는 1KB로 하였다. 따라서 B-tree, aR-tree의 팬아웃(fanout)은 각각 82, 36이였으며 H-Cube의 임계값 V는 128이었다. H-Cube의 셀 개수 N은 수식 (1)에 의해 32×32로 설정하였다.

4.2. H-Cube의 성능

이 절에서는 H-Cube의 성능을 a3DR-tree와 aRB-tree의 성능과 비교한 결과에 대해 기술한다. 우선 저장공간과 구축시간에 대한 실험결과를 살펴보자. 그림 12는 데이타셋 $D[\alpha_{obj}, 1]$일 때 요구되는 저장 공간을 α_{obj}의 함수로 나타내고 있다. a3DR-tree의 크기는 α_{obj}가 커짐에 따라 급속히 증가하고 있다. 이는 2장에서 언급한 바와 같이 집계값이 변할 때 마다 MBR정보가 중복되어 저장되기 때문이며 이러한 현상은 α_{obj}가 증가함에 따라 더욱더 현저하게 나타나게 된다. aRB-tree의 크기는 a3DR-tree의 크기보다는 작지만 H-Cube 크기의 수배에서 수십 배에 달한다. aRB-tree 크기의 주된 요소는 B-tree의 크기이며 aR-tree의 크기는 300여KB에 불과하였다. 이는 시간이 지남에 따라 B-tree에 유지될 정보가 많아지고 아울러 크기도 비례하여 증가한다는 것을 의미한다. 그림 12에서 α_{obj}=10까지 aRB-tree의 크기가 일정하게 유지된 이유는 α_{obj}=10이후에 B-tree의 크기가 증가되기 시작했기 때문이다. 한편 H-Cube는 저장 공간을 가장 적게 요구하고 있으며

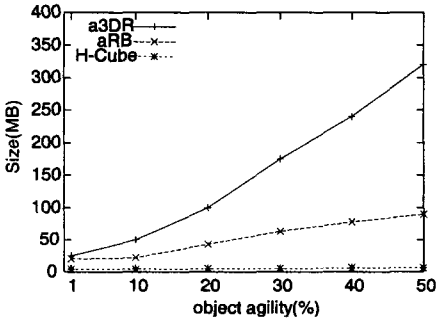

그림 12 크기 비교 : $D[\alpha_{obj}, 1]$

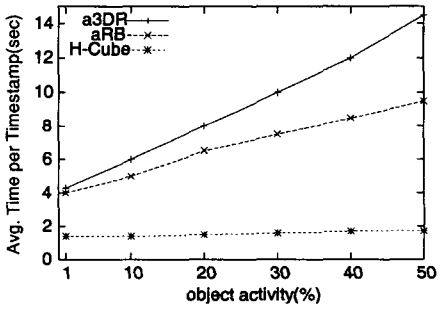

그림 13 구축 시간 비교 : $D[\alpha_{obj}, 1]$

α_{obj}의 영향을 가장 적게 받고 있음을 알 수 있다.

그림 13은 각 구조를 구축하는데 소요된 실제시간(wall-clock time)을 각 시간스탬프당 소요된 평균시간으로 보여주고 있다. H-Cube가 가장 적은 시간이 소요되었음을 알 수 있다. aRB-tree의 구축 시간은 그 크기에 비해서 많은 시간이 소요되었는데 그 이유는 B-tree가 디스크에 산재되어있고 그에 따라 많은 탐색시간이 소요되었기 때문으로 분석할 수 있다.

그림 14는 데이타셋 $D[\alpha_{obj}, 1]$에 대해 작업부하 $Q[0.25, 50]$을 적용했을 때의 평균페이지 접근수를 보여주고 있다. H-Cube는 α_{obj}에 독립적이면서 월등한 성능을 보여주고 있다. 반면 a3DR-tree와 aRB-tree는 α_{obj}에 영향을 받는 대신 그림 15와 같이 편중도 에는 영향을 받지 않고 있다. 이는 a3DR-tree와 aRB-tree에는 집계값이 저장되기 때문에 나타나는 현상이다.

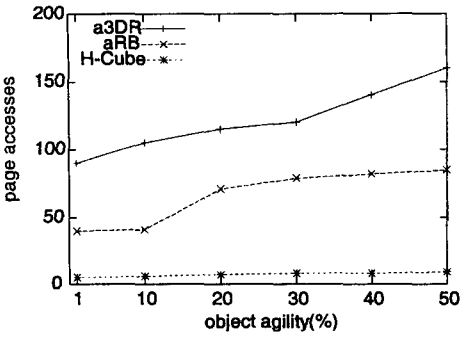

그림 14 $D[\alpha_{obj}, 1]$, $Q[0.25, 50]$

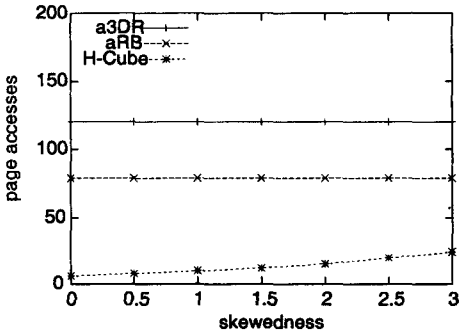

그림 15 $D[30, z]$, $Q[0.25, 50]$

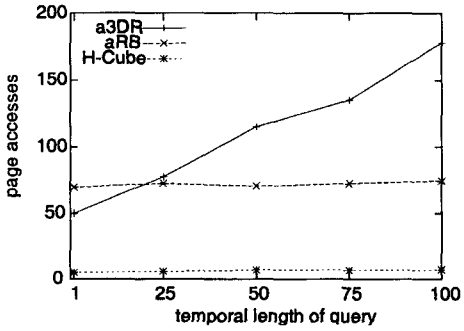

그림 16 $D[30, 1]$, $Q[0.25, q_t]$

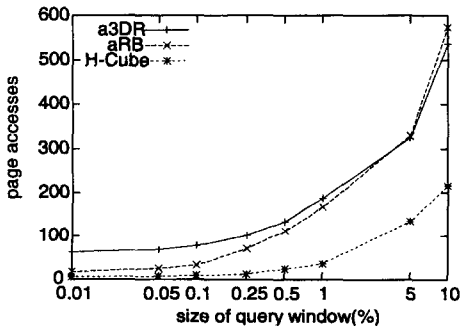

그림 17 $D[30, 1]$, $Q[q_s, 50]$

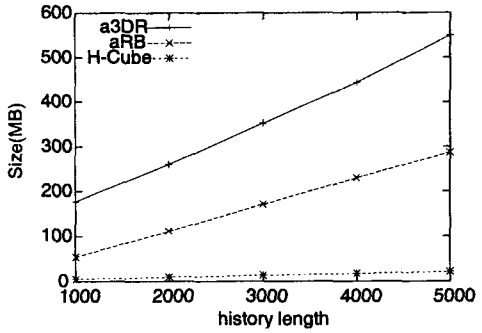

그림 18 시간에 따른 크기의 변화

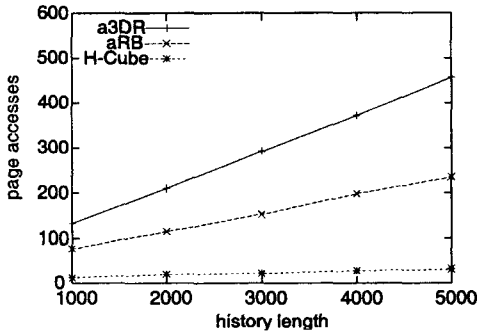

그림 19 시간에 따른 성능의 변화

또한 그림 16을 보면 aRB-tree는 q_t에 영향을 받지 않고 있는데 이는 방문해야 될 B-tree의 개수가 거의 일정하고 더욱이 노드가 질의윈도에 완전히 포함된 경우 해당 B-tree의 루트만 방문해도 되기 때문이다. 반면에 a3DR-tree는 점점 증가되는 크기와 높이로 인해 q_t가 증가함에 따라 성능이 급격히 나빠지는 것을 알 수 있다. 하지만 짧은 시간간격에는 a3DR-tree가 aRB-tree보다 좋은 성능을 보이는 경우도 보이고 있다. 이는 시간간격이 짧아질수록 a3DR-tree와 aRB-tree의 성능은 같아지며(즉, 만약 시간간격이 0이라면 a3DR-tree와 aRB-tree의 성능은 동일하다), 더욱이 a3DR-tree의 경우 방문해야할 B-tree를 갖고 있지 않기 때문이다. 한편, H-Cube 역시 q_t에 영향을 받지 않고 있는데 이는 누적합 형태를 유지하고 있기 때문에 나타나는 의미 있는 결과이다.

그림 17은 질의윈도우의 크기를 조절했을 때의 성능 평가 결과를 보여주고 있다. 이 경우도 역시 H-Cube가 월등한 성능 우위를 보여 주고 있다. aRB-tree의 경우 q_s에는 많은 영향을 받고 있는데 이는 q_s가 커질수록 방문해야 할 B-tree의 개수도 함께 증가하기 때문이다. 그 결과 5%보다 큰 질의윈도우에 대해서는 aRB-tree가 a3DR-tree보다 더 나쁜 성능을 보여주고 있다.

마지막으로 각 구조의 시간에 따른 확장성에 대해 알아보았다. 이를 위해서 $D[30,1]$과 $Q[0.25,5\%]$의 데이타셋에 대해 시간스탬프를 1,000개에서 5,000까지 늘려보았다. 그에 따른 크기와 성능의 변화가 그림 18과 19에 각각 나타나 있다. a3DR-tree와 aRB-tree의 크기는 시간스탬프의 길이에 선형적으로 비례하여 증가하고 있는 것을 확인할 수 있다. 반면 H-Cube의 크기는 시간스탬프가 5,000인 경우 a3DR-tree와 aRB-tree의 크기의 10%정도밖에 되지 않았다. 성능에 있어서도 비슷한 양상을 보여주고 있다. 이러한 사실은 H-Cube가 크기와 질의 효율면 모두에서 가장 확장성이 뛰어나고 이와 더불어 시공간 데이타웨어하우스에 보다 더 적합하다는 것을 말해주고 있다.

5. 결 론

본 논문에서는 시공간 데이타웨어하우스를 위한 유지비용을 최소화하고 집계질의를 효율적으로 수행하는 새로운 접근 방법을 제시하고 있다. 이 접근 방법은 계층구조 방식(hierarchical access method)을 기반으로 한 구조가 아닌 셀 기반 구조로서의 최초의 시도이다. 본 논문에서 제안하고 있는 힐버트큐브(Hilbert Cube, H-Cube)는 적응적이며, 완전순서화되었으며 또한 누적합 기법을 적용한 셀 기반 색인 구조이다.

H-Cube는 힐버트 곡선을 이용하여 셀들에게 완전순서를 부여한 후 이 순서를 이용하여 디스크에 클러스터링(clustering)하고 있으며, 아울러 누적합(prefix-sum) 기법을 적용함으로써 집계질의(aggregation query) 효율성을 높이고 있다. 또한 시공간 데이타의 변화하는 분포에 대처하기 위해 적응적 셀 분할 기법(adaptive cell partitioning scheme)을 제공하고 있다.

본 논문에서는 H-Cube의 성능을 다각도로 분석하기 위해 다양한 데이타셋과 작업부하를 통해 구축비용과 질의처리 성능 비교 실험을 실행하였다. 이 광범위한 실험을 통해 H-Cube는 저장 공간과 유지비용 그리고 질의처리의 효율성 모든 면에서 기존 접근 방법보다 월등히 우세하다는 결과를 보였다. 이 사실은 H-Cube가 시공간 데이타웨어하우스에 적합한 구조라는 것을 말해주고 있다.

추후 연구과제로서 H-Cube에 대해 선택도 분석(selectivity estimation) 및 근사오차(approximation error)등을 예측할 수 있는 이론적인 근거를 제시하고자 한다. 또한 H-Cube를 통해 객체의 이동방향과 패턴에 대한 질의와 같이 보다 복잡한 질의를 처리할 수 있는 기능을 제공하고자 한다.

참 고 문 헌

[1] Dimitris Papadias, Yufei Tao, Panos Kalnis, and Jun Zhang, "Indexing Spatio-Temporal Data Warehouses," In Proceedings of the Eighteenth International Conference on Data Engineering, pp. 166-175, 2002.

[2] Antonin Güttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," In Proceedings of the 1984 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 47-57, 1984.

[3] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger, "The R*-Tree: An efficient and Robust Access Method for Points and Rectangles," In Proceedings of the 1990 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 322-331, 1990.

[4] Dimitris Papadias, Panos Kalnis, Jun Zhang, and Yufei Tao, "Efficient OLAP Operations in Spatial Data Warehouses," In Advances in Spatial and Temporal Databases, 7th International Symposium, SSTD 2001, 2001.

[5] Ching-Tien Ho, Rakesh Agrawal, Nimrod Megiddo, and Ramakrishnan Srikant, "Range Queries in OLAP Data Cubes," In Proceedings of the 1997 ACM SIGMOD Int'l. Conf. on Management of Data, pp. 73-88, 1997.

[6] Hanan Samet, "The quadtree and related hierarchical data structures," ACM Computing Surveys, Vol.16, No.2, pp. 187-260, 1984.

[7] Volker Gaede and Oliver Günther, "Multidimensional Access Methods," ACM Computing Surveys, Vol.30, No.2, pp. 170-231, 1998.

[8] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. "The Grid File: An Adaptable, Symmetric Multi-Key File Structure," ACM Transactions on Database Systems, Vol.9, No.1, pp. 38-71, 1984.

[9] Jim Gray, Adam Bosworth, Andrew Layman, and Hamid Pirahesh, "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total," In Proceedings of the Twelfth International Conference on Data Engineering, pp. 152-159, 1996.

[10] David J. Abel and David M. Mark, "A Comparative Analysis of some 2-Dimensional Orderings," International Journal of Geographical Information Systems, Vol.4, No.1, pp. 21-31, 1990.

[11] H. V. Jagadish, "Linear Clustering of Objects with Multiple Attributes," In Proceedings of the 1990 ACM SIGMOD Int'l Conf. on Management of Data, pp. 332-342, 1990.

[12] Christos Faloutsos and Shari Roseman, "Fractals for secondary key retrieval," In Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pp. 247-252, 1989.

[13] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," TKDE, Vol.13, No.1, pp. 124-141, 2001.
[14] J. G. Griffiths, "An Algorithm for Displaying a Class of Space-filling Curves," Software-Practice and Experience, Vol.16, No.5, pp. 403-411, 1986.
[15] Jon Louis Bentley, Donald F. Stanat, and E. Hollings Williams Jr., "The Complexity of Finding Fixed-Radius Near Neighbors," In Information Processing Letters, Vol.6, No.6, pp. 209-212, 1977.
[16] Yannis Theodoridis, JeRerson R. O. Silva, and Mario A. Nascimento, "On the Generation of Spatiotemporal Datasets," In Proceedings of the 6th Int'l. Symp. on Spatial Databases, pp. 147-164, 1999.

최 원 익

1996년 서울대학교 컴퓨터공학과 졸업 (공학사). 1998년 서울대학교 대학원 컴퓨터공학과 졸업(공학석사) 1998년~현재 서울대학교 전기 컴퓨터공학과 박사과정 관심분야는 다차원공간 색인구조, 시공간 색인구조, XML 등

이 석 호

정보과학회논문지 : 데이타베이스
제 30 권 제 2 호 참조