VA-Tree : 대용량 데이터를 위한 효율적인 다차원 색인구조

송석일[†] · 이석희[††] · 조기형[†††] · 유재수[††††]

요 약

이 논문은 다차원의 특징벡터를 벡터 근사치로 표현한 후 색인 트리를 구성하여 검색의 효율을 높이는 VA (Vector Approximate)-트리를 제안한다. 이 논문에서 제안하는 VA-트리는 전체적인 색인구조의 저장 공간을 줄이기 위해서 VA-화일의 벡터 근사치 개념을 이용하여 데이터양이 증가해도 검색 성능이 저하되지 않도록 하는 트리 형태의 구조를 갖는다. VA-트리는 MBR 기반의 색인구조이지만 MBR간에 겹침이 발생하지 않는 분할 방법을 사용하여 검색 효율을 높인다. 제안하는 색인구조와 기존의 여러 다차원 색인구조와의 성능 평가를 통해 제안하는 방법의 우수함을 보인다.

VA-Tree : An Efficient Multi-Dimensional Index Structure for Large Data Set

Seok Il Song[†], Seok Hee Lee[††], Ki Hyung Cho[†††] and Jae Soo Yoo[††††]

ABSTRACT

In this paper, we propose a multi-dimensional index structure, called a VA(Vector Approximate)-tree that is constructed with vector approximates of multi-dimensional feature vectors. To save storage space for index structures, the VA-tree employs vector approximation concepts of VA-file that presents feature vectors with much smaller number of bits than original value. Since the VA-tree is a tree structure, it does not suffer from performance degradation owing to the increase of data. Also, even though the VA-tree is MBR(Minimum Bounding Region) based tree structure like a R-tree, its split algorithm never allows overlap between MBRs. We show through various experiments that our proposed VA-tree is a suitable index structure for large amount of multi-dimensional data.

Key words: 멀티디미어 데이터베이스, 다차원 색인구조, 유사성 검색

1. 서 론

최근 지리정보 시스템, 움직임 객체 관리 시스템, 동영상/이미지 내용기반 검색 시스템, 시계열 데이터베이스 시스템과 같이 다차원 데이터를 이용하는 응용에 대한 관심이 고조되고 있다. 다차원 데이터에 대한 효율적인 검색을 위해 다차원 색인구조가 사용된다는 것은 이미 잘 알려진 사실이다. 다차원 색인구조에 대한 연구는 지난 20여 년간 매우 활발히 진행되어 왔고 그 결과 KDB-트리[1], hB-트리[2], GRID-화일[3], BANG-화일[4], R-트리[5], R+-트리[6], R*-트리[7], TV-트리[8], SS-트리[9], VAMkd-트리[10], VAMSplitR-트리[11], X-트리[12], SR-트리[13], 피라미드 기법[14], VA-화일[15] 등 매우 많은 수의 색인구조들이 제안되어 왔다.

기존에 제안된 색인구조들은 적게는 2~10 많게는 11~200, 또는 그 이상의 차원들을 대상으로 하고 있다. 하지만 대부분의 색인구조들은 10차원을 넘는

이 연구는 2001년도 학술진흥재단(KRF-2001-041-E00233)의 지원으로 수행되었음.
접수일 : 2002년 3월 22일, 완료일 : 2003년 2월 3일
[†] 충주대학교 전기전자정보공학부 컴퓨터공학과
[††] 동아방송대학 인터넷방송과
[†††] 충북대학교 전기전자 및 컴퓨터 공학부
[††††] 정회원, 충북대학교 전기전자 및 컴퓨터 공학부

경우 색인구조로서의 기능을 상실하는 차원의 저주 문제를 가지고 있다. 이 문제를 해결하기 위해서 VA-화일이 제안되었다. 이 방법은 고차원의 데이터에 대해서는 트리형태의 색인구조는 무의미 하므로 데이터를 표현하는 양을 줄여서 순차 검색의 속도를 향상시키고 있다. 하지만 이 방법은 저차원의 데이터에 대해서는 기존의 트리구조의 색인구조에 비해서 성능이 떨어진다. 또한 순차 검색이므로 데이터의 양이 증가할수록 성능이 감소된다.

모든 다차원 색인구조의 응용이 10차원이 넘는 고차원의 데이터를 처리하는 것은 아니다. 특히, 지리정보 시스템이나 최근 연구되고 있는 움직임 객체 관리 시스템과 같은 응용에서는 10차원 이내의 데이터가 사용된다. 10차원 이내의 데이터는 기존의 R-트리나 R*-트리로도 처리가 가능하지만 증가하는 데이터의 용량과 사용자의 요구를 충족시키기 위해서는 보다 빠른 검색 및 변경이 가능해야 한다.

이 논문에서는 위에서 제시한 최근의 응용 경향에 부응하기 위한 다차원 색인구조를 제안한다. 제안하는 방법에서는 데이터를 벡터 근사치로 표현한 후 이를 트리 형태로 구성하여 검색의 효율을 높이는 색인구조 VA-트리를 제안한다. 제안하는 VA-트리는 색인에 사용되는 데이터의 크기를 줄이기 위해서 VA-화일의 벡터 근사치 개념을 이용하면서 데이터량이 증가해도 검색성능이 저하되지 않도록 트리 형태를 취한다. 또한, VA-트리는 MBR 기반의 색인구조이지만 MBR간에 겹침이 발생하지 않는 분할 방법을 사용하여 검색 효율을 높인다.

이 논문의 구성은 다음과 같다. 2장에서 기존의 다차원 색인구조에 대해서 보다 자세히 설명하고 이 논문의 접근 방향을 기술한다. 3장에서는 제안하는 VA-트리에 대한 소개 및 구조에 대해서 설명하고, 4장에서는 VA-트리의 각 연산 수행 방법에 대해서 자세히 설명한다. 5장에서는 성능평가와 그 결과를 기술하여 제안하는 방법의 우수성을 보이고 6장에서는 결론을 맺는다.

2. 관련 연구

기존에 제안된 다차원 색인 구조들을 분류해보면 TV-트리, X-트리, SS-트리, SR-트리와 같은 데이터 분할을 사용하는 색인 구조와 KDB-트리, hB-트리, BANG 화일, GRID 화일과 같이 공간분할을 사용하는 색인 구조들로 크게 나누어 볼 수 있다. 또한, Hybrid-트리[16]와 같은 이들의 혼합 형태의 색인 구조가 존재하며 기타 LS(Locality Sensitive)해슁 기법[17]을 사용하는 색인구조와 VA-화일과 IQ-트리[18]처럼 요약 기법을 사용하는 색인 구조도 존재한다.

공간 분할을 사용하는 색인구조들은 공간을 서로 겹치지 않도록 분할하여 표현한다. 대부분의 공간 분할 색인구조들은 비단말노드의 엔트리의 크기가 차원과 독립적이다. 하지만 차원이 증가할수록 죽은 공간(Dead Space)이 증가하고 하향 연쇄 분할(Downward Cascading Split)의 빈도수가 증가하여 저장공간 활용률이 현저히 떨어진다. 데이터 분할을 사용하는 색인 구조들은 MBR(Minimum Bounding Region)형태로 비단말노드의 엔트리를 표현하기 때문에 죽은 공간이 최소가 된다. 또한 겹침을 허용하기 때문에 하향 연쇄분할 같은 문제는 발생하지 않는다. 하지만 MBR 표현 방식 때문에 차원이 증가할수록 비단말노드의 팬-아웃은 떨어지며 겹침 영역이 증가하게 된다. 공간분할 방법과 데이터분할 방법의 장점을 혼합한 방식이 Hybrid-트리이다. Hybrid-트리는 공간분할방식의 비단말노드 엔트리의 크기가 차원에 독립적이라는 특성과 MBR로 엔트리를 표현하며 겹침을 허용하여 하향 연쇄 분할을 피하고 있다. 하지만 완벽하게 비단말노드의 엔트리 크기가 차원과 독립적이라고 말할 수 없다.

차원의 저주문제를 해결하는 또 다른 방법으로 피라미드 기법과 VA-화일을 들 수 있다. 피라미드 기법은 다차원 공간을 1차원으로 변환하여 B-트리를 이용하여 색인 하는 방법이다. 이 방법의 최대의 문제점은 K-최근접 질의를 지원하지 못한다는 것이다. VA-화일은 피할 수 없는 순차 스캔을 가능한 빨리 수행하기 위하여 벡터 근사치를 이용하여 유사도 검색을 수행하는 방법이다. VA-화일은 그림 1과 같이 벡터 공간을 셀(Cell)로 분할한다. 이 셀들은 각 특징 벡터를 비트로 코드화한 벡터 근사치를 만드는데 사용한다. VA-화일은 모든 벡터 근사치에 대한 간단한 배열구조의 화일이다. 질의 처리는 모든 벡터 근사치를 스캔하여 각 벡터 근사치에서 질의 포인트까지의 최소 거리 경계와 최대 거리 경계를 계산한다. 이 거리 경계를 이용하여 실제 데이터와의 거리를

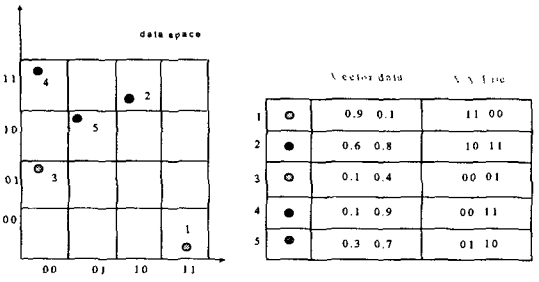


그림 1. VA-화일의 구조

계산해야하는 객체 수를 상당히 줄일 수 있으므로 디스크 I/O 횟수가 그만큼 적어진다. VA-화일은 다차원 데이터에 대한 검색 성능은 상당히 우수한 것으로 알려져 있다. 하지만 모든 벡터 근사치와의 최소 거리 경계와 최대 거리 경계를 계산해야 하므로 CPU 연산 비용이 많이 소요되는 단점이 있다. 또한, 모든 벡터 근사치에 대해 순차 검색을 수행하기 때문에 대용량 데이터에 대해서는 검색 성능의 한계가 있다.

VA-트리에서는 이상 설명한 기존 색인구조들이 가지고 있는 문제들을 해결하기 위한 접근 방법으로 다음과 같은 혼합 구조를 취한다. 먼저, 대용량의 데이터에 대해서 효과적으로 색인이 가능하도록 특징 벡터를 VA-화일의 벡터근사치 표현기법을 이용해서 표현한다. 그리고 각 노드간의 겹침으로 인한 성능저하 문제를 해결하기 위해서 공간 분할 기법(Grid 화일)을 도입하였다. 동시에 공간 분할 기법의 죽은 공간으로 인한 문제를 해결하기 위해 MBR 표현기법을 이용한다. 다음 장부터 제안하는 VA-트리의 특징 및 구조, 그리고 각 연산에 대하여 자세하게 설명한다.

3. VA-트리

3.1 개요 및 특징

2장 후반부에서 언급한 것처럼 VA-트리는 기존 색인구조들에서 제시하고 있는 여러 기법을 효과적으로 혼합한 구조이다. 먼저, 특징벡터를 벡터 근사치로 매핑시킨 후 이 벡터 근사치를 이용하여 트리를 구성한다. 또한, 공간 분할 기법을 도입하여 K-D-B 트리 기반의 색인 구조처럼 노드간 겹침은 없다. 마지막으로 공간 분할기법에서 문제가 되는 죽은 공간을 없애기 위해서 MBR 표현기법을 도입하였다. 전체적인 구조를 살펴보면 노드는 단말노드와 비단말노드로 구분된다. 비단말노드의 한 엔트리는 [하위노드를 포함하는 MBR, 하위노드에 대한 포인터]로 구성되며, 단말노드의 한 엔트리는 [벡터 근사치, 실제 데이터 레코드에 대한 포인터]로 구성된다.

표면적인 형태로는 MBR을 이용하는 R-트리 기반의 색인구조와 유사하지만 내부적인 연산은 전혀 다른 면을 가지고 있다. 이미 설명한 것처럼 노드간의 겹침을 발생시키지 않기 위해서 공간분할 기법을 이용한다. 공간 분할 기법을 이용하기 위해서는 매 분할시 해당 분할이 어떤 차원의 어떤 위치에 대해서 분할이 이루어졌는지에 대한 정보를 가지고 있다가 다음 분할에 이를 이용해야 한다. 하지만 엔트리들을 MBR로 표현하는 경우에는 이 전에 정확히 어느 차원의 어느 위치에 대해서 분할되었는지 알 수 없다. 어떤 차원의 어느 위치가 분할시 사용되었던 차원인지 모르는 경우에는 새로운 엔트리를 삽입할때 문제가 발생할 수 있다.

그림 2에서 이에 대한 예를 보여준다. 2차원의 데이터 영역에 각 차원의 범위가 $0 \sim 24$ 라고 할 때, 차원 당 3비트로 표현하고 균등하게 분할하면, 각 차원의 분할 지점 값은 [0, 3, 6, 9, 12, 15, 18, 21, 24]가 되고 전체 셀의 개수는 64개이다. 노드 A와 B는 1차원의 12를 기준으로 분할되어 생성된 것이다. 이 상태에서 객체 o_1 을 삽입하려고 한다. 노드는 단 두개이므로 객체 o_1이 적당히 삽입될 노드를 둘 중에서 골라야 한다. 기존의 R-트리의 경우 새로운 객체를 삽입할 때 MBR의 확장이 최소가 되는 노드를 선택한다. 그에 따르면 B 보다는 A에 삽입되는 것이 효과적이다. 객체 o_1 이 A에 삽입되어 A가 A′으로 확장된다. 이어서, o_2가 삽입되려 한다. 이때에는 A′ 보다는 B에 삽입되는 것이 더 효과적이므로 B에 삽입되고 B는 B′으로 확장된다. 이렇게 되면 A′과 B′은 서로 겹치게 된다.

이미 언급한데로 제안하는 VA-트리는 공간 분할 기법을 사용하므로 첫번째 차원의 12로 분할되어 A와 B가 생성되었으면 A와 B는 분할된 위치를 걸쳐서 MBR이 확장되어서는 안 된다. 즉, 위의 예에서 o_1이 A에 삽입되면 A의 MBR은 분할위치 12에 걸쳐지므로 안되고 B에 삽입되어야 한다. 그렇게 되면 위와 같은 겹침 문제가 발생하지 않는다. VA-트리는 공간 분할 기법을 이용하되 MBR로 노드를 표현하므로 해당 MBR이 어떤 위치에서 분할되어 생성되

었는지 알 수 없다. 이를 보완하기 위해서 VA-트리에서는 분할위치 그룹이라는 개념을 도입한다. 각 차원의 분할이 가능한 모든 위치들을 우선순위에 따라 그룹지어 놓고 노드를 분할할 때 분할이 가능한 분할 위치들 중 가능 높은 우선순위에 해당하는 분할 위치에 따라서 분할을 한다.

그림 2에서 1차원에 대한 분할위치는 총 7개(3, 6, 9, 12, 15, 18, 21) 이다. 이들중 가장 우선순위가 높은 분할 위치는 가장 중심에 있는 12 이다. 다음 우선순위의 분할 위치는 6 과 18 이다. 그 다음 우선순위의 분할 위치는 3, 9, 15 그리고 21이다. 그림에서 만일 노드 A를 1차원에 대해서 분할하려 한다면 분할 가능한 위치 6, 9 중에서 가장 우선 순위가 높은 6에서 분할한다. 엔트리를 삽입할 노드를 찾을 때는 먼저 각 노드를 새로운 엔트리를 포함하도록 확장한다. 그리고 나서, 각 노드의 확장영역에 포함된 분할 위치들 중 가장 우선순위가 높은 분할 위치가 가장 우선순위가 낮은 노드를 선택해서 삽입한다. 이에 대한 자세한 설명은 삽입 알고리즘을 설명하는 부분에서 다시 한다.

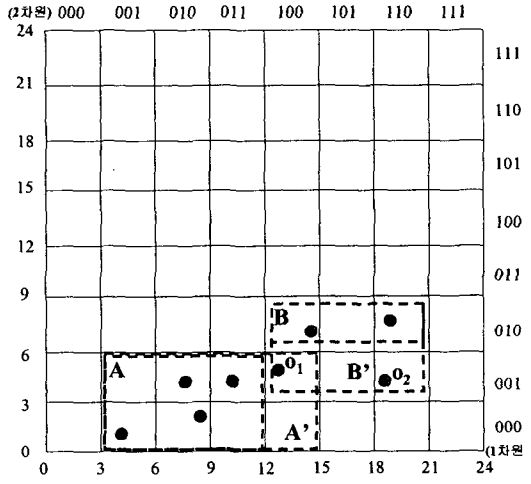


그림 2. 벡터 공간

3.2 벡터 근사치 표현

특징벡터를 벡터 근사치로 매핑시키는 방법은 VA-화일과 유사하다. 각 차원을 b개의 비트로 표현한다고 가정할 때 각 차원으로 2^b개만큼의 셀이 생기며 전체 차원의 수가 d 라면 이 데이터 영역에 대해서는 총 $(2^b)^d$개의 셀이 생긴다. VA-트리에 삽입하려 하는 특징벡터들이 표현 가능한 최대, 최소 값을 이용하여 비트로 표현되는 각 셀에 매핑 시킨다. 수식 1은 특징벡터를 벡터 근사치로 매핑 시킬 때 사용하는 수식이다. 이 수식에서 max_i와 min_i는 차원 i의 데이터영역이 표현할 수 있는 최대 값과 최소 값을 의미한다. 또한 fv_i는 변환하려하는 특징벡터의 차원 i의 값이며 v_i 는 변환된 차원 i의 벡터근사치 값이다.

$$v_i = \left| \frac{(fv_i - \min_i)}{2^b (max_i - min_i)} \right| \qquad \text{(수식 1)}$$

그림 3에서 2차원의 데이터 영역에 각 차원의 범위가 0~24 라고 할 때, 차원당 3비트로 표현하고 균등하게 분할하면, 각 차원의 분할 지점 값은 [0, 3, 6, 9, 12, 15, 18, 21, 24]가 되고 전체 셀의 개수는 64개이다. VA-트리에서는 이 분할 지점 값을 분할된 데이터 공간에 2차원의 특징벡터가 8개가 놓여 있을 때, 특징벡터를 벡터 근사치로 표현하면 표 1과 같다.

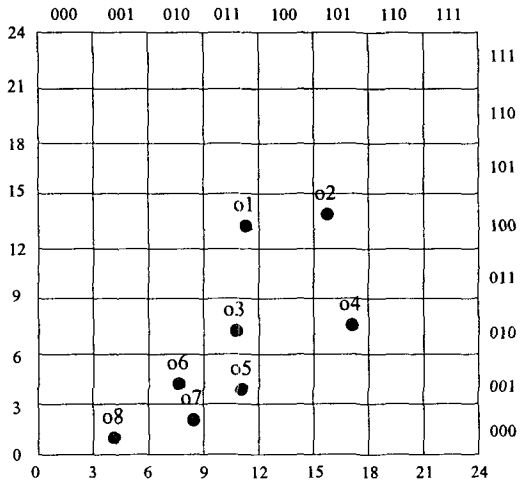


그림 3. 벡터 공간

표 1. 벡터 근사치

	실제 데이터 (1차원, 2차원)		벡터 근사치 (1차원, 2차원)
객체 1	(11, 14)		(011 100)
객체 2	(16, 13)		(101 100)
객체 3	(10, 8)		(011 010)
객체 4	(17, 7)	⇒	(101 010)
객체 5	(11, 4)		(011 001)
객체 6	(8, 5)		(010 001)
객체 7	(8, 2)		(010 000)
객체 8	(4, 1)		(001 000)

한 예로 객체 1이 속해 있는 셀이 (011 100)이므로 이 값이 객체 1의 벡터 근사치가 된다.

3.3 MBR(Minimum Bounding Region)

VA-트리의 MBR은 R-트리에서와 유사하다. 단지 VA-트리에서는 MBR을 셀을 나타내는 벡터근사치로 표현한다는 것이 다르다. 또한, VA-트리는 MBR간에 겹침을 허용하지 않는다. 이를 위해서 VA-트리의 MBR은 다음의 규칙을 따라야 한다. 규칙을 정의하기 전에 먼저 분할위치 그룹에 대해서 먼저 정의한다. 분할위치 그룹이란 각 차원에 대해서 분할을 할 수 있는 위치들의 집합을 말한다. 각 차원에 대해서 주어진 비트가 b라 할 때 2^b-1 만큼이 분할위치가 될 수 있다. 앞서 설명한 것처럼 모든 분할위치들에 각각 우선순위를 주고 우선순위가 같은 분할위치들을 하나의 그룹으로 하였다. 앞으로 $SPGn$을 우선순위가 n인 분할위치 그룹이라고 표시한다. n은 $1 \sim b$ 사이의 정수이며 우선순위는 1이 가장 높으며 b가 가장 낮다.

SPG_n에의 분할위치의 수는 2^{n-1}개이다. 한 차원에 대해 존재하는 SPG_n은 다음의 식 2를 이용해서 얻을 수 있다. n은 $1 \sim b$ 사이의 정수이다. 수식 3으로 계산되는 SPG_n의 분할위치값중 가장 작은 값을 기본 분할 위치값(BSP_n)이라 한다.

$$2^{b-n} \qquad \text{(수식 2)}$$

각 SPG_n의 분할 위치는 다음과 같이 구할 수 있다.

SPG_1: 2^{b-1}

SPG_2: 2^{b-2}, $2^{b-2}+2^{b-1}\times 2$

SPG_3: 2^{b-3}, $2^{b-3}+2^{b-2}\times 1$, $2^{b-3}+2^{b-2}\times 2$, $2^{b-3}+2^{b-2}\times 3$

...

SPG_n: 2^{b-n}, $2^{b-n}+2^{b-(n-1)}\times 1$, $2^{b-n}+2^{b-(n-1)}\times 2$, ..., $2^{b-n}+2^{b-(n-1)}\times n$

이상과 같이 분할 그룹과 그들 간의 우선순위를 정의할 때 VA-트리의 MBR은 다음의 규칙을 만족해야 한다.

규칙 1 : *어떤 MBR의 각 차원 i의 범위 R_i가 걸쳐 있는 분할 위치 중 가장 우선순위가 높은 분할 그룹 SPG_n에 속한 분할위치를 k_i라 하고 그 그룹의 기본 분할 위치를 BSP_n이라 할 때, 각 Ri를 (k_i-$BSP_n \sim k_i$+BSP_n)로 확장했을 때 절대로 이 확장된 MBR과 겹치는 MBR이 트리의 같은 레벨에 있어서는 안 된다.*

그림 4의 예를 통해 규칙 1을 설명한다. 그림 4에서 MBR1과 MBR2는 동시에 존재할 수 없다. MBR1의 각 차원의 최우선 분할 위치는 1차원의 경우 2가 되고 2차원의 경우 4가 된다. 규칙 1에서 언급한 것처럼 MBR1을 확장하면 그림의 확장된 MBR1 과 같이 된다. 이때 MBR2가 확장된 MBR1과 겹치게 되므로 둘은 같이 존재할 수가 없다. 하지만 MBR3와 MBR4는 이런 규칙에 만족한다.

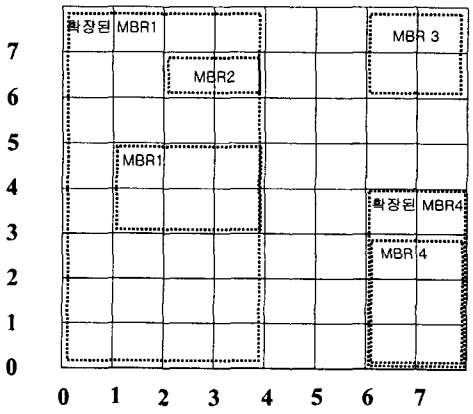


그림 4. MBR 규칙의 예

규칙 1이 어떻게 MBR간에 겹침을 허용하지 않는지에 대한 것은 삽입 알고리즘을 자세히 설명하는 다음에 설명한다. 그림 5와 그림 6은 예제 데이터를 이용하여 MBR과 VA-트리가 구축되는 것을 보여준다. 그림 5는 예제 데이터 1에서 8까지 입력된 후의 트리 모습을 보여준다. 데이터 공간의 셀 정보 중 첫 번째 비트를 보면 1/2 분할 위치 이하의 영역은 0이며, 1/2 분할 위치 이상의 영역은 1이다. 또한 두 번째 비트는 데이터 공간의 1/4 분할 위치 이하이면 0, 이상이면 1이다. 그리고 1/8 분할 위치를 기준으로 세 번째 비트가 0 또는 1이다. 그러므로 이 비트를 보면 어떤 영역을 포함하고 있는지 알 수 있다.

3.4 VA-트리의 구조

VA-트리의 기본적인 구조는 R-트리와 매우 유사하다. 단말노드의 엔트리들을 포함하는 최소의 영역

을 상위노드(비단말)노드에 저장하고 다시 이 노드의 *MBR*을 상위에 저장하는 형식을 취한다. 물론 겹침을 허용하지 않기 위해서 VA-트리만의 고유한 엔트리 삽입방법과 분할 방법가지고 있다.

그림 5와 그림 6에서 예를 통해 VA-트리의 구조를 보여준다. 이 예의 VA-트리는 단말노드와 비단말노드에 최대로 저장될 수 있는 엔트리의 수를 3으로 가정한다. 비단말노드에는 하위노드들이 포함하고 있는 모든 영역을 표시하기 위하여 하위영역을 나타내는 벡터근사치와 상위영역을 나타내는 벡터근사치를 내포하고 있다. 단말노드 N_5가 포함하고 있는 엔트리는 o_1, o_2번 객체이다. 각각의 객체를 나타내는 벡터근사치는 [(011, 100), (101, 100)]이다. 단말노드 N_5가 포함된 상위노드에는 단말노드 N_5의 영역정보를 나타내는 벡터근사치 [(011, 100) (101, 100)]과 포인터가 포함된다. 이것으로 단말노드 N_5가 y축을 기준으로 분할되었음을 알 수 있다. 단말노드 N_1, N_2, N_4를 포함하고 있는 비단말노드 A의 영역정보를 나타내는 방법도 앞서 설명한 방법과 같다. 단말노드 N_1, N_2, N_4의 모든 객체를 포함하는 영역의 벡터 근사치는 [(001, 000), (011, 010)]이다. 그러므로 상위노드에서 비단말노드 A를 나타내는 영역의 벡터근사치는 [(001, 000), (011, 010)]이다. 이렇게 영역정보를 나타냄으로써 죽은 영역을 최소로 줄이고 실제 객체가 포함된 영역인 살아있는 영역이 나타나도록 영역정보를 표현할 수 있다.

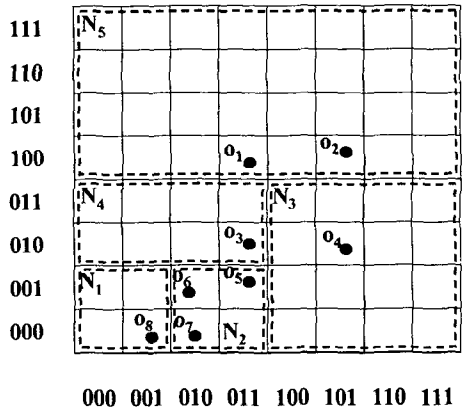


그림 5. 벡터 근사치 공간

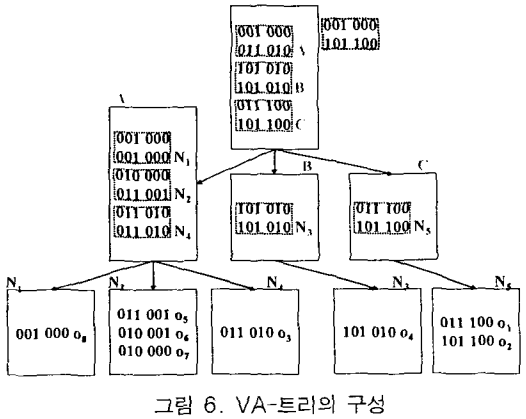


그림 6. VA-트리의 구성

벡터 근사치 4개(o_1, o_2, o_3, o_4)가 입력되면 단말노드의 분할이 일어난다. 다음에 설명할 단말노드 분할 기준에 의해 분할 차원과 위치를 선택한다. 2차원에서 분할이 일어나기 때문에 벡터근사치 o_1, o_2를 포함하는 노드와 o_3, o_4를 포함하는 노드로 분할된다. 그리고 비단말노드에 2차원으로 분할된 단말노드에 대한 영역정보 [(011 100), (101, 100)], [(011, 010), (101 010)]가 상위에 반영된다.

다시 o_5, o_6이 입력되면 단말노드 3은 1차원에 대해서 분할이 발생되어 o_3, o_6, o_5을 포함하는 노드와 o_1을 포함하는 노드로 분할된다. 그리고 두 노드에 대한 정보를 상위 노드에 반영한다. o_7이 입력되면 다시 o_3, o_6, o_5를 포함하는 노드에 넘침이 발생되고 o_3을 포함하는 노드와 o_5, o_6, o_7을 포함하는 노드로 분할된다. 이 분할을 상위에 반영할 때 루트노드에 넘침이 발생하게 되고 루트 노드가 분할되게 된다. 분할하는 방법에 대한 자세한 내용은 각 연산의 알고리즘을 설명하는 부분에서 언급한다.

3.5 VA-트리의 연산

3.5.1 삽입 연산

삽입 연산은 기본적으로 두 단계에 걸쳐서 수행된다. 첫 번째 단계에서는 새로운 특징벡터를 삽입할 단말노드를 찾는다. 이를 위해서 먼저 특징벡터를 벡터 근사치로 매핑한다. 단말노드를 찾는 단계에서는 VA-트리에 겹침이 발행하지 않도록 하는 특별한 하위노드 선택 방법이 있다. 이 방법에 대해서는 다음에 자세히 설명한다. 첫 번째 단계에서는 단말노드를 찾는 일 외에도 특징벡터가 단말노드에 삽입되었을 때 변경될 MBR을 미리 변경하는 일을 수행한다. 즉, 적절한 하위노드가 결정이 되면 상위 노드에 저장되어 있는 하위 노드에 대한 MBR을 삽입할 특징벡터

를 포함하도록 확장한다. 첫 번째 단계가 정상적으로 끝나면 루트부터 찾아낸 단말노드까지의 경로를 스택에 저장해서 발생할 수 있는 분할에 사용될 수 있도록 한다.

두 번째 단계에서는 찾아낸 단말노드에 엔트리를 삽입한다. 이때 단말노드에 새로운 특징벡터를 삽입할 공간이 없으면 넘침이 발생하게 되고 분할을 수행하여 이를 처리한다. 먼저 넘침이 발생한 단말노드를 분할하고 스택으로부터 단말노드에 대한 부모노드를 알아내서 부모노드에 단말노드 분할로 생성된 새로운 엔트리와 변경된 MBR을 반영한다. 부모노드에 더 이상 여유 공간이 없어서 다시 넘침이 발생하면 스택으로부터 다시 그 노드의 부모노드를 꺼내고 단말노드에서와 동일하게 분할을 수행한다. 이것은 새로운 엔트리를 수용할 수 있는 노드가 있을 때까지 반복되며 없을 경우에는 루트노드가 분할되고 트리의 높이가 하나 증가한다. 이런 전반적인 알고리즘에 대한 의사코드를 그림 X에서 보여준다. 다음부터는 각 단계의 주요알고리즘들을 자세히 설명한다.

그림 7에서 1행은 특징벡터(F)를 벡터 근사치(V)로 변환한다. 2행은 벡터 근사치를 삽입할 단말노드를 LocateNode함수를 이용하여 검색한다. 3~7행은 새로운 벡터 근사치를 삽입한 단말노드에 겹침이 발생하면 분할을 수행하기 위해 SplitNode 함수를 호출하고, 그렇지 않은 경우에는 단말노드에 새로운 벡터 근사치를 삽입한 후 노드의 엔트리 수를 하나 증가시킨다.

```
Procedure Insert
Start procedure
1  입력 특징벡터(F)를 벡터근사치(V)로 변환
2  Leafnode := Newent를 삽입할 단말노드를 찾는
   다(LocateNode 호출);
3  If (LeafNode.entnum == OVERFLOW )
4     분할을 수행한다 ( SplitNode 호출 );
5  else
6     LeafNode에 NewEnt를 삽입;
7     LeafNode.EntNum++;
8  End if
End procedure
```

그림 7. Insert 알고리즘

단말노드 검색(LocateNode)

새로운 벡터 근사치가 삽입될 단말노드를 찾는 함수이다. VA-트리의 단말노드 검색알고리즘은 겹침없는 VA-트리를 만드는데 아주 중요한 역할 수행하는 부분이다. 알고리즘은 먼저 루트노드를 접근해서 시작한다. 루트노드에 저장되어 있는 모든 MBR에 대해서 다음과 같은 하위노드 선택 알고리즘을 수행한다.

(1) 새로운 엔트리를 삽입할 수 있도록 MBR을 확장한다. 이때 확장한 MBR을 exMBR이라 한다. 만일 MBR과 exMBR이 동일하면 이 MBR에 해당하는 하위노드를 선택한다.

(2) exMBR을 이미 설명한 MBR 규칙에 따라 exMBR의 각 차원에 해당하느 값이 걸쳐있는 분할 위치 중에서 가장 최우선순위를 갖는 분할 위치를 선택한다. 다음 이 분할위치가 속하는 분할 위치 그룹의 기본값을 분할위치에 더하고(최대값) 빼서(최소값) 최종적으로 orgMBR을 만들어 낸다.

(3) orgMBR과 다른 MBR들을 비교해서 겹침이 발생하지 않으면 이 MBR에 대한 하위노드를 선택한다.

위의 과정을 루트노드의 모든 MBR에 대해서 수행하며 하위노드를 선택하게 되면 과정을 바로 종료하고 다시 하위 노드를 방문해서 동일한 과정을 반복한다. 만일 선택한 하위노드가 단말노드이면 단말노드 찾기를 종료한다. 이 과정에서 선택한 하위노드의 MBR이 변경되는 경우에는 MBR 변경을 수행하고 하위노드를 방문한다. 그리고 이 알고리즘에서는 방문하는 모든 노드를 경로 스택이란 곳에 저장하여 이 함수를 호출한 Insert에 반환한다.

분할(SplitNode)

분할 함수가 호출되면 제일 먼저 넘침이 발생한 단말노드를 분할한다. 단말노드 분할로 생성된 새로운 노드에 대한 MBR을 경로스택에 저장된 부모노드에 반영하고 또한 분할로 변경된 단말노드의 MBR을 상위 노드에 반영한다. 분할된 단말노드를 상위 노드에 반영하고 이를 반영할 때 상위노드에 넘침이 발생하면 비단말노드를 분할하여 이를 상위노드에 반영한다. 이러한 과정을 비단말노드에 겹침이 발생하지 않을때까지 반복한다.

```
Procedure LocateNode
Start procedure
1  If (Node 가 단말노드)
2    return Node;
3  End If
4  for ( each MBR of Node )
5    ExMBR = CurMBR을 NewEnt를 포함하도록 확장;
6    If ( ExMBR 과 CurMBR이 같으면)
7      break;
8    OrgMBR = 분할 우선순위를 이용해서 ExMBR에 대한 orgMBR 생성;
9    OrgMBR과 CurMBR을 제외한 다른 MBR 과 겹침이 있는지 조사;
10   If ( 겹침이 없으면 )
11     break;
12   Else
13     CurMBR = 다음 MBR;
14 End for
15 Node = CurMBR에 해당하는 자식노드, 1부터 반복
End procedure
```

그림 8. LocateNode 알고리즘

분할 알고리즘에서 가장 중요한 부분은 바로 분할 차원과 분할 위치를 결정하는 방법이다. 기존의 여러 다차원 색인구조에서 이를 결정하는 많은 방법을 제시했다. VA-트리에서는 이를 결정할 때 가장 우선으로 생각하는 것이 겹침이 업도록 하는데 있다. 겹침이 없는 색인구조를 위해서 어떻게 분할을 수행하는 가는 다음 단말노드 분할과 비단말노드 분할에서 자세히 설명한다.

단말노드 분할(SplitLeafNode)

분할을 수행하기 위해서 먼저 분할 차원과 분할 위치를 결정한다. 단말노드를 분할할 때는 가장 먼저 분할 위치를 결정하고 이에 따른 분할 차원을 결정한다. 단말노드의 모든 엔트리에 대해서 다음을 수행한다.

(1) 단말노드 MBR의 각 차원의 값이 걸쳐있는 분할 위치중에서 가장 최우선 분할 위치를 선택하고 이때의 차원을 분할 차원으로 결정한다.

(2) 만일 가장 최우선의 분할위치에 걸쳐있는 차원이 2이상이면 데이터가 되도록 골고루 분포되어 있는 차원을 선택한다.

예를 들어 이를 설명해 본다. 벡터근사치를 구하는 비트의 수를 4라고 하자. 어떤 단말노드에 (1101 0010), (1110 0001), (1010 0100), (1011 0000) 과 같이 4개의 벡터근사치가 저장되어 있다고 하자. 첫 번째 비트에 대해서는 1, 2차원 모두 MBR 값이 걸쳐져 있지 않아 분할을 수행할 수 없다. 다음으로 높은 우선순위(두번째 비트)에 1차원의 MBR 값이 걸쳐져 있다. 따라서 분할이 가능한 위치는 두 번째 비트에 해당하는 값이 되며 이 위치에 대해서 1차원으로 분할을 수행한다.

비단말노드 분할(SplitInternalNode)

비단말노드 분할은 단말노드의 분할과 다르다. 비단말노드에는 점이 아닌 사각형의 MBR이 저장되므로 분할위치와 분할 차원을 선택하는데 다음과 같은 절차를 따른다.

(1) 분할을 수행할 비단말노드에 저장된 MBR을 모두 orgMBR로 변환한다.

(2) 각 차원에 대해서 orgMBR들의 값들을 비교하면서 그 차원에 최우선 분할 위치에 대해 분할이 가능한지를 검사한다.

(3) (2)단계에서 최우선 분할위치에 대해서 분할이 가능한 차원이 2 이상이면 분할을 했을 때 되도록 골고루 MBR이 분포할수 있는 차원을 선택한다.

모든 노드들은 항상 최우선 분할 위치부터 분할이 수행되므로 비단말노드에는 항상 최우선 순위 분할 위치로 겹침없이 분할할 수 있는 차원이 하나이상 존재한다.

```
Procedure SplitNode
Start procedure
1  Oldent, Splitent = currentnode를 분할 ( SplitLeafNode 호출 );
2  While (currentnode != ROOT )
3  Parentnode = pop(stack);
4    parentnode에서 currentnode에 대한 엔트리를 oldent로 변경;
5    If (parentnode.entnum == OVERFLOW)
6      Parentnode를 분할한다.
       (SplitInternalNode 호출 )
7    Else
8      splitent를 parantnode에 삽입;
9    end if
10 end while
End procedure
```

그림 9. SplitNode 알고리즘

```
Procedure SplitLeafNode
Start procedure
1  MBR = 단말노드의 MBR;
2  for ( 모든 우선 분할 위치그룹에 대해서 ) // 높은 우선순위의 분할위치그룹 부터
3    for ( 모든 차원에 대해서 )
4      if ( 해당 차원의 MBR 값이 분할 위치 그룹내의 값들중 하나에 걸쳐 있으면 )
5        candidatedim[] = 현재 차원;
6        candidatepos[] = MBR이 걸쳐져 있는 분할 위치 그룹내의 값;
7      end if
8    end for
9  end for
10 if ( candidatedim[] 의 개수가 2 이상이면 )
11   candidatedim[] = 엔트리들이 가장 균등하게 분할되는 차원;
12   if ( candidatedim[] 의 개수가 2 이상이면 )
13     candidatedim[] = candidatedim 에 대해서 범위가 큰 차원;
14     if ( candidatedim[] 의 개수가 2 이상이면 )
15       splitdim = candidatedim[]중 임의의 차원;
16       splitpos = candidatepos[]중 임의의 차원;
17     else
18       spltidim = candidatedim[]; splitpos = candidatepos[];
19     end if
20   else
21     spltidim = candidatedim[];splitpos = candidatepos[];
22   end if
23 else
24     spltidim = candidatedim[];splitpos = candidatepos[];
25 end if
26 splitdim과 splitpos 에 따라서 노드를 분할;
27 두 노드에 대한 엔트리 생성 및 반환 (oldent,splitent);
End procedure
```

그림 10. SplitLeafNode 알고리즘

VA-화일에서와 마찬가지로 VA-트리에서도 하나의 셀에 한 노드에서 수용할 수 있는 객체보다 많은 객체가 저장될 수 있다. 이런 경우에는 분할이 불가능하게 된다. VA-트리에서는 이를 [19]의 수퍼노드 개념을 도입해서 해결한다. 하나의 셀에 들어가는 객체들은 서로 유사한 객체들이며 이들은 어떤 질의에 의해서 같이 접근될 확률이 높다. 따라서 이 객체들을 수퍼노드에 같이 저장해 놓으면 디스크 접근 회수를 줄이는 효과도 볼 수 있다.

3.5.2 삭제 연산

VA-트리에서 특징벡터를 삭제하는 방법은 먼저 특징벡터를 벡터 근사치로 매핑시킨다. 탐색 연산을 이용하여 벡터 근사치가 있는 노드를 찾아서 지우고, 노드 내에 데이터가 없으면 노드를 삭제하고 상위 노드에 반영하는 방법을 사용한다.

3.5.3 검색 연산

VA-트리에서 유사도 검색은 다른 색인 구조와 마찬가지로 루트부터 시작하여 단말노드까지 각 노드별로 엔트리들을 검사하여 수행하는데, 이 논문에서는 [16]에서 사용한 다단계 최근접점 탐색 방법을 이용한다. 질의 포인터에서 가장 가까운 K개의 유사 객체를 찾는 K-근접 질의 검색을 생각해보면, 우선 질의 포인트를 벡터 근사치로 매핑시킨다. 트리를 순회하면서 비단말노드의 비트 정보만을 이용하여 질의 포인트에서 비단말노드까지의 최소 거리 경계와 질의 포인트에서 단말노드내의 벡터 근사치까지의 최소 거리 경계를 계산하여 효과적으로 가지치기할 수 있다. 거리가 가까운 벡터 근사치를 가진 실제 데이터들을 읽어와 실제 거리를 계산한다. K번째 실제 거리보다 가까운 근사 거리가 없을 때 탐색을 끝내고 K개를 반환한다.

```
Procedure SplitInternalNode
Start procedure
1  NodeMBR = 노드의 MBR;
2  for ( 모든 차원에 대해서 )
3    for ( 노드내의 모든 MBR에 대해서 )
4      orgMBR[] = 노드내의 MBR을 orgMBR로 변환;
5    end for
6    splitpos = 해당 차원에대해 NodeMBR 값이 걸쳐져 있는 최우선 분할위치;
7    if( orgMBR[]의 MBR들을 splitpos에 대해서 겹침없이 분할가능)
8      candidatedim[] = 현재 차원;
9    end if
10 end for
11
12 if ( candidatedim[] 의 개수가 2 이상이면 )
13   candidatedim[] = 엔트리들이 가장 균등하게 분할되는 차원;
14   if ( candidatedim[] 의 개수가 2 이상이면 )
15       splitdim = candidatedim[]중 임의의 차원;
16       splitpos = candidatepos[]중 임의의 차원;
17   else
18     spltidim = candidatedim[];splitpos = candidatepos[];
19   end if
20 else
21     spltidim = candidatedim[];splitpos = candidatepos[];
22 end if
23 splitdim과 splitpos 에 따라서 노드를 분할;
24
25 두 노드에 대한 엔트리 생성 및 반환 (oldent,splitent);
End procedure
```

그림 11. SplitInternalNode 알고리즘

```
Procedure Delete
Start procedure
1  입력 특징벡터(F)를 벡터근사치(V)로 변환
2  Leafnode := Targetent를 삭제할 단말노드를 찾는다(LocateNode 호출);
3  If (LeafNode.entnum == Empty)
6    do while ( 1 )
7      상위노드에서 해당 엔트리를 삭제
8      if (상위노드 == Empty)
9        continue;
10     else
11       exit;
12     end if
13   end while
14 else
15   exit;
16 end if
End Procedure
```

그림 12. Delete 알고리즘

유사도 검색을 위해 그림 13에서와 같이 후보 집합, 결과 집합 우선순위 큐 두개를 사용한다. 후보 집합에는 질의 포인트에서 벡터 근사치나 비단말노드까지의 근사 거리 순으로 정렬되고, 결과 집합에는 질의 포인트에서 실제 데이터까지의 거리 순으로 정렬된다. 결과 집합의 k번째 거리를 나타내는 K번째 거리 값(result_k_dist)은 초기치 ∞에서 점점 작아진다. 루트부터 트리를 순회하면서 질의 포인트에서 비단말노드까지의 거리를 계산하여 후보 집합에 넣는다. 3행은 질의 포인트부터 지금까지 방문한 K번째 데이터까지의 실제거리가 후보 집합내의 거리보다 작으면 탐색을 끝내고 결과 집합에서 k개를 반환한다. 4~6행은 후보 집합에서 읽은 데이터가 비단말노드이면 그 비단말노드내 엔트리까지의 근사 거리를 계산하여 후보 집합에 저장한다. 7~9행은 후보 집합에서 꺼낸 데이터가 단말노드이면 그 단말노드내의 벡터 근사치까지의 거리를 계산하여 후보 집합에 저장한다. 10~12행은 후보 집합에서 읽은 데이터가 벡터 근사치이면 질의 포인트와의 거리를 계산하여 결과 집합에 저장하고, K번째 거리 값을 갱신한다.

```
Procedure Search
Start procedure
1  결과 집합, 후보 집합, K번째 거리 = ∞;
2  질의 포인트 ==> 비트근사치로 변환;
3  while ( K번째 거리  > 후보 집합 첫 번째 거리)
4    if (후보 집합 첫 번째가 비단말노드)
5      비단말노드내의 엔트리까지의 거리 계산;
6      K번째 거리보다 작은 엔트리를 후보 집합에 삽입;
7    else if (후보 집합 첫 번째가 단말노드)
8      단말노드내의 엔트리까지의 거리 계산;
9      K번째 거리보다 작은 엔트리를 후보 집합에 삽입;
10   else if (후보 집합 첫 번째가 비트근사치)
11     질의에서 비트근사치까지의 거리 계산;
12     결과 집합 삽입;
13     K번째 거리를 갱신;
14   end if
15 end while
14 결과 집합 K개를 반환;
End procedure
```

그림 13. Search 알고리즘

4. 성능평가

4.1 성능 분석

실험을 통한 성능평가를 수행하기 전에 트리구조에 대한 공간 측면을 해석학적으로 분석해 본다. 제안하는 VA-트리가 차지하는 공간은 다음과 같은 수식에 의해서 계산이 가능하다. 아래 수식에서 N은 전체 특징벡터의 개수를 의미하며 *nodenum(i)*란 트리의 i 레벨에서의 노드 개수를 의미한다. *leaffanout*과 *internalfanout*은 각각 단말노드와 비단말노드의 팬아웃을 의미하며 *total nodenum*이란 트리 전체가

차지하는 노드의 개수를 의미한다.

$$\leq affanout = \left| \frac{nodesize}{sizeof(\leq afentry)} \right| \quad \text{(수식 3)}$$
$$\bigcap nalfanout = \left| \frac{nodesize}{sizeof(\bigcap nalentry)} \right|$$

$$h = \left| \log_{\bigcap nalfanout} \left(\frac{N}{\leq affanout} \right) \right| + 1 \quad \text{(수식 4)}$$

$$total\ nodenum = \sum_{i=0}^{h-1} nodenum(i)$$
$$where, \quad \text{(수식 5)}$$
$$nodenum(i) = \begin{cases} i=0 & \frac{N}{\leq affanout} \\ i>0 & \frac{nodenum(i-1)}{\bigcap nalfanout} \end{cases}$$

R-트리 계열의 색인구조와 VA-트리가 차지하는 공간을 비교해 보기 위해 다음과 같은 가정을 한다. R-트리 계열의 색인구조에서 특징벡터의 한 차원 값을 표현하는데 필요한 공간은 4Byte이고 VA-트리에서는 4Bit이다. 색인구성에 사용할 데이터는 2차원이며 총 1,000,000개 이다. 색인노드의 크기는 단말/비단말 모두 4kbyte이다. 이때 VA-트리와 R-트리 계열의 색인구조가 차지하는 공간을 계산해보면 R-트리 계열의 색인구조는 약 2947개의 노드를 필요로 하고 VA-트리의 경우에는 1223개의 노드를 필요로 한다. R-트리계열의 색인구조가 약 2.4 배 더 많은 공간을 필요로 한다. 이럴 경우 겹침의 허용과 같은 문제는 접어 두고 단순히 색인구조에서 질의를 처리할 때 접근할 노드의 비율을 가지고 탐색성능의 차이를 비교한다고 해도 약 2.4 배의 성능차이가 발생하게 된다.

4.2 실험 및 결과

이 절에서는 제안된 VA-트리를 구현하여 실험한 성능평가 결과를 기술한다. 제안된 VA-트리의 검색성능을 평가하기 위한 방법으로 다음 표 2의 입력인자에 따라 여러 가지 환경에서 기존의 색인구조와 검색성능을 비교함으로써 검색의 효율성을 평가하였다. 성능평가 인자로는 표 3을 사용하였다. 이 논문에 제시된 모든 실험결과는 Solaris 2.7 운영 체제에 Sun Enterprise 3000, 메인메모리 1GB를 장착하고 있는 시스템을 이용하여 C언어로 구현하였다. 컴화일러는 gcc 2.7.1.이다.

이 논문에서 비교 대상으로 한 기존의 색인구조는 논문의 저자가 제공해준 원시 프로그램의 내용을 수정하지 않고 사용하였다. 한 차원의 벡터 데이터를 표현하기 위한 비트 수는 VA-화일에서 4~5비트일 때 가장 좋은 성능을 보이므로 성능 비교를 위하여 4비트로 하였다. 실험데이터는 동영상에서 각각의 프레임을 하나의 이미지로 간주하고 여기에서 추출된 칼라 정보 값을 9개의 수치로 표현한 실제 데이터와 난수발생기를 이용하여 균일 분포를 갖는 실수형 데이터 집합을 만들어 사용하였다. 모든 실험 데이터와 생성되는 인덱스는 같은 디스크 내에 존재하고, 색인을 위해 사용되는 페이지의 크기는 다양한 환경에서의 성능측정을 위해 4kbytes, 8kbytes, 16kbytes, 32kbytes로 변경하면서 실험하였다. 실험 전반에서 VA-트리의 높이는 보통 2였으며 최대 3을 넘지 않았다.

질의는 이미 삽입된 특징벡터중 하나를 난수발생기를 통해 선택하여 기준 특징벡터로 삼고 그 특징벡터와 가장 가까운 10개의 특징벡터를 찾는 K-NN 질의와 기준 특징벡터와 X% 유사한 특징벡터를 찾는 범위질의를 수행하였다. 이미지 검색 시스템을 가정하고 실 예를 들어 본다면 K-NN 의 경우 "기준 이미지와 가장 유사한 10개의 이미지를 검색해 주시오"와 같이 표현할 수 있으며 범위질의의 경우에는 "기준이미지와 약 X% 유사한 이미지들을 찾아주시오"와 같이 표시할 수 있다.

그림 14는 VA-트리와 R*-트리, X-트리의 삽입시간을 비교한 결과이다. 그림 14는 노드크기를 32kbytes, 데이터 차원을 10차원, 데이터 개수를 200,000개에서 1,000,000개까지 늘려가면서 색인을 구성할 때 걸리는 시간을 측정하였다. 측정 결과

표 2. 입력 인자

인 자	설 명	값
D	데이터의 차원 수	4~20
S	데이터의 개수	100,000~1,000,000
Q	질의의 수	100
K	knn 질의시 최근접 개체의 수	10
Node_s	노드의 크기	4kbytes~32kbytes

표 3. 성능평가 인자

인 자	설 명	단위
N	질의시 접근한 노드의 수	개
T	시간	초
M	생성된 노드의 수	개

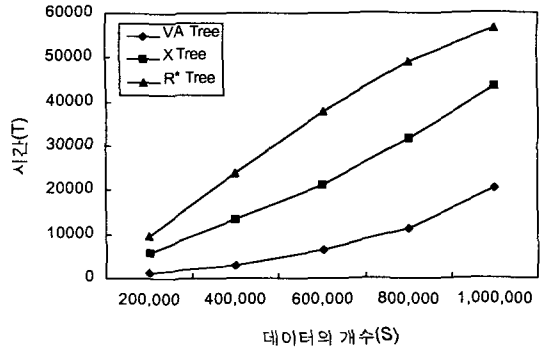


그림 14. VA-트리, X-트리, R*-트리 의 트리 구성 시간 비교

VA-트리를 구축하는데 가장 적은 시간이 소요됨을 확인할 수 있었다. 이런 차이는 근본적으로 보다 적은 디스크 I/O 회수로부터 생긴다. 제안하는 VA-트리가 기존의 X-트리나 R-트리에 비해서 특징벡터 하나를 표현하는데 드는 공간이 약 1/8 정도이므로 VA-트리를 구성하는 노드의 수가 훨씬 적고 이로 인해 디스크 I/O 회수도 줄어들게 된다. 이 외에도 단말노드 찾기 알고리즘이나 분할 알고리즘이 X-트리나 R-트리에 비해서 훨씬 단순한 점도 원인이 될 수 있다.

그림 15와 그림 16은 균일 분포 데이터 집합들을 이용하여 VA-트리와 VA-화일, R*-트리, X-트리에서 K-NN 질의를 수행한 경우에 대한 검색 성능을 비교한 것이다. 그림 11은 노드크기를 4kbytes, 데이터 개수를 100,000개, 차원을 4에서 20까지 늘려가면서 색인을 구성한 후, 주어진 질의 데이터에 대해서 10개의 근접 데이터를 100번 검색했을 때 접근되는 평균 접근 노드 수를 측정한 결과이다. 질의는 모든 경우에 있어서 기존의 색인구조보다 우수한 검색성능을 나타냄을 알 수 있다. VA-트리는 벡터 근사치를 이용하여 색인을 구성하기 때문에 생성되는 노드의 수가 기존의 색인구조에 비해 현저하게 적고 비단말노드가 서로 겹치지 않는 것이 성능 향상의 원인이라 할 수 있다.

그림 15는 노드크기를 4kbytes, 차원을 10차원으로 고정시키고, 데이터 개수를 100,000개에서 1,000,000개까지 늘려가면서 색인을 구성한 후, 주어진 질의 데이터에 대해서 10개의 근접 데이터를 100번 검색했을 때 접근되는 평균 접근 노드 수를 측정한 결과이다. VA-화일은 모든 비트 데이터를 순차 검색하기 때문에 데이터가 많아질수록 검색시간이 매우 증가한다. 반면 VA-트리에서는 데이터 집합이 대용량이라도 접근하는 노드수가 거의 일정한 것을 알 수 있다. 이유는 VA-트리가 트리구조이므로 단말노드의 개수가 많아지더라도 트리의 높이가 증가하지 않으면 거의 일정한 검색 시간을 보장할 수 있기 때문이다.

그림 17은 노드크기를 4kbytes, 차원을 9차원, 데이터 개수를 300,000개의 실제 데이터를 이용해 색인

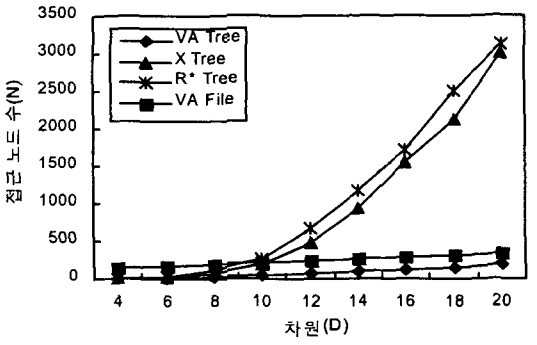


그림 15. 차원증가에 따른 각 색인구조의 접근 노드 수 비교

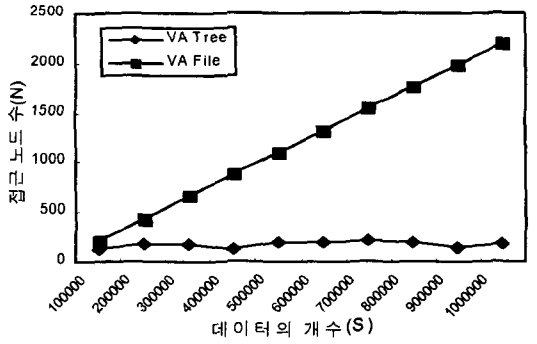


그림 16. 데이터의 증가에 따른 VA-트리와 VA-화일의 노드 접근 수 비교

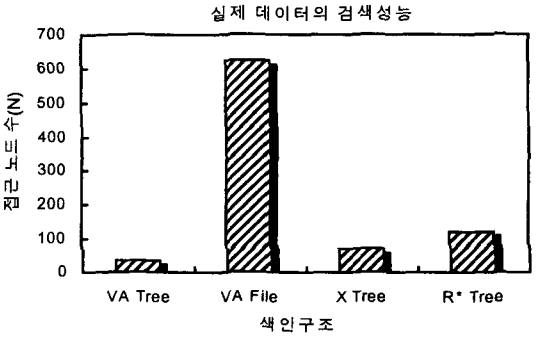


그림 17. 실 데이터로 실험했을 때 각 방법의 노드 접근수 비교

을 구성한 후, 주어진 질의 데이터에 대해서 10개의 근접 데이터를 100번 검색했을 때 접근되는 평균 접근 노드 수를 측정한 결과이다. 데이터는 동영상의 각각 장면에서 장면을 특징지우는 색상정보를 9개의 수치 데이터로 추출하여 사용하였다. 실제 데이터에 대한 성능측정 결과에서도 VA-트리는 기존의 색인 구조에 비해 성능이 우수함을 보였다. VA-화일이 가장 성능이 나쁘게 나왔는데 VA-화일의 경우에는 검색을 위해서는 항상 전체 데이터를 접근해야 하기 때문이다. 고차원으로 가게 되면 다른 트리구조의 색인들이 색인의 의미를 잃어버리면서 VA-화일보다 낮은 성능을 보이겠지만 저차원의 경우 VA-트리를 비롯한 기존의 색인구조들의 트리가 충분히 여과기능을 수행하기 때문에 VA-화일의 성능이 가장 낮을 수밖에 없다.

저장공간 활용 율을 평가하기 위해 노드크기를 4kbytes, 데이터 개수를 100,000개, 차원을 4에서 20까지 늘려가면서 색인을 구성한 후 생성된 노드의 개수(M)를 (그림 14)에서 보여준다. 그림 18를 보면 R*-트리와 X-트리는 생성되는 노드의 개수가 선형적으로 증가되는 것을 알 수 있다. 또한 제안한 VA-트리가 저장공간활용 측면에서도 뛰어난 성능을 보임을 알 수 있다. VA-트리가 VA-화일에 비해 많은 노드가 생성되는 이유는 VA-화일은 트리형태로 구축하지 않고 순서대로 단순히 저장만 하기 때문이다. 그러나 이러한 특징으로 대용량 데이터에 대해서는 검색성능이 급격히 떨어지는 문제점이 발생한다.

그림 19와 그림 20은 다차원 색인구조 중에 최근에 제안된 피라미드 기법과의 검색성능을 비교한 것이다. 피라미드 기법의 경우 K-NN 질의를 처리하지 못하기 때문에 이 실험에서는 범위 질의를 이용하였다. 범위에 해당하는 유사도를 5%에서 30%까지 늘려가면서 접근하는 노드 수를 측정하였다. 유사도는 데이터가 가질 수 있는 가장 적은 값과 큰 값의 거리를 유사도 100%로 정하고 이를 기준으로 5%, 10% 등과 같은 각각의 유사도에 해당하는 거리로 환산하여 질의에 사용하였다.

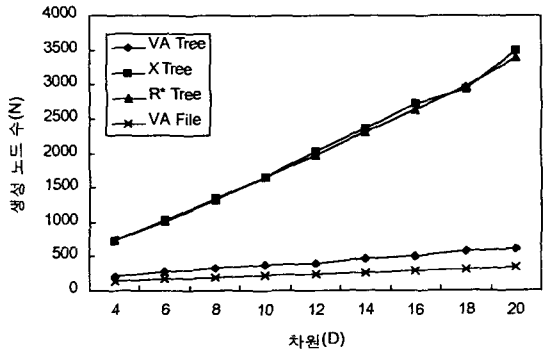


그림 18. 각 방법의 저장공간 사용율

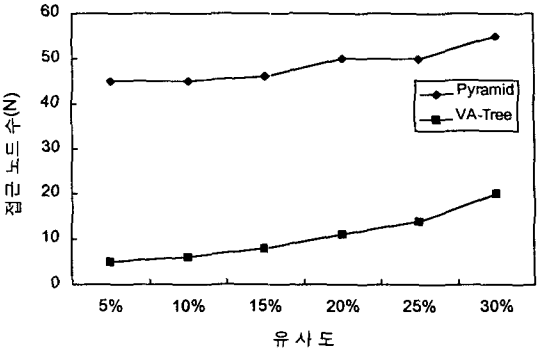


그림 19. 8차원일 때 피라미드 기법과 VA-트리의 노드 접근 수 비교

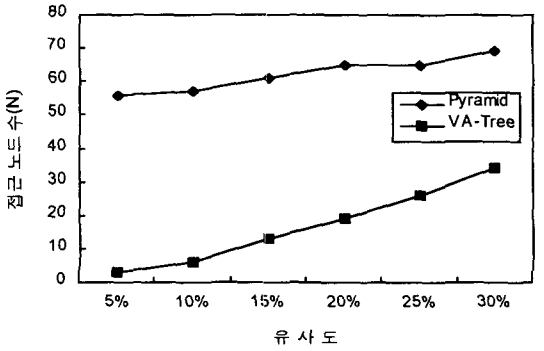


그림 20. 16차원일 때 피라미드 기법과 VA-트리의 노드 접근 수 비교

그림 19는 노드크기 16kbytes, 8차원, 200,000개 데이터에 대한 실험 결과이다. VA-트리가 피라미드 기법에 비해 평균 78%의 성능향상을 확인하였다. 그림 20은 노드크기 16kbytes, 16차원, 200,000개 데이터에 대한 실험 결과이다. 이 경우에는 VA-트리가 피라미드 기법에 비해 평균 73%의 성능향상을 확인하였다. 성능 개선의 요인은 무엇보다도 제안하는 VA-트리가 피라미드 기법에 비해서 생성되는 노드의 수가 적기 때문이다.

그림 21은 VA-화일과 VA-트리의 성능을 시간 측면에서 비교한 것이다. 이때 VA-화일은 raw 디바이스 상에 연속적으로 저장하였고 이때의 페이지 크기는 16K로 하였다. 4~12차원 사이에서는 제안하는

VA-트리가 VA-화일에 비해서 검색 시간이 매우 우수하다. 하지만 12차원을 경계로 차원이 증가할 수록 점점 성능차이가 벌어지는 것을 볼수 있었다. VA-트리는 급격히 증가하는 반면 VA-화일의 검색시간은 아주 소폭으로 증가하였다. 원인을 살펴 보면 VA-화일의 경우 물리적 디스크상에 연속으로 저장되어 있으므로 접근하는 페이지수가 4차원일때 보다 증가하더라도 디스크 접근으로 인한 시간 증가가 아주 적은 반면 VA-트리는 디스크 접근 회수가 증가할 수록 시간 증가가 매우 크기 때문이다.

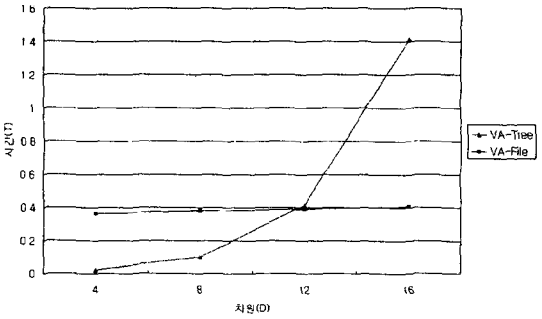


그림 21. VA-화일과 VA-트리의 성능비교(시간)

5. 결 론

이 연구에서는 저차원 데이터를 처리하는 응용의 검색 속도를 향상시키기 위한 VA-트리를 제안하였다. 제안하는 방법은 VA-화일처럼 특징 벡터를 벡터 근사치로 표현하므로 노드의 팬-아웃을 증가시킬 뿐 아니라 전체적인 저장공간을 줄인다. 또한 이를 트리 형태로 표현하기 때문에 검색 속도가 데이터 양의 증가에 크게 영향을 받지 않는다. 또한 다양한 실험을 통해 제안하는 방법이 2~10에서 기존의 색인구조에 비해서 매우 뛰어난 검색성능을 보임을 증명하였다. 또한, 색인구조가 차지하는 저장공간의 크기측면에서도 다른 색인구조에 비해서 매우 적음을 알 수 있었다. 향후연구에서는 이 논문에서 제안한 VA-트리를 상용 DBMS의 한 접근 방법으로 사용할 수 있도록 하기 위한 동시성제어 및 회복 기법에 대한 연구를 수행한다.

참 고 문 헌

[1] J. T. Robinson, "The K-D-B-tree: A Search Structure for Large Multidimensional Dynamic Indexes," ACM SIGMOD, pp. 10-18, Apr. 1981.

[2] D. Lomet and B. Salzberg "The hB-Tree: A Robust Multiattribute Search Structure," In Proc. ICDE Conf., pages 296-304, 1989.

[3] J. Nievergelt, H. Hinterberger and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," ACM Transactions on Database Systems, pp. 38-71, 1984.

[4] M. Freeston. "The BANG file: a new kind of grid file." In Proc. VLDB conf., pages 260-269, 1987.

[5] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," ACM SIGMOD, pp. 47-57, June 1984.

[6] T. Sellis, N. Roussopoulos and C. Faloustos, "The R+-tree: A Dynamic Index for Multi-Dimensional Objects," In Proc. 13th Intl. Conf. On VLDB, pp. 507-518, 1987.

[7] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," ACM SIGMOD, pp 322-331, 1990.

[8] K. I. Lin, H. Jagadish and C. Faloutsos, "The TV-tree: An Index Structure for High-Dimensional Data," The VLDB Journal, Vol 3, No. 4, pp. 517-549, Oct. 1994.

[9] D. A. White and R. Jain, "Similarity Indexing with the SS-tree," In Proc. 12th Intl. Conf. On Data Engineering, pp. 515-523, 1996.

[10] D. A. White and R. Jain, "Similarity Indexing: Algorithms and Performance," In Proc. SPIE: Storage and Retrieval for Image and Video Databases IV, Vol. 2670, pp. 62-73, 1996.

[12] S. Berchtold, D. A. Keim and H. P. Kriegel, "The X-tree: An Index Structure for High-Dimensional Data," Very Large Data Bases (VLDB'96), Proc. 22nd., pp. 28-39, 1996.

[13] N. Katayama and S. Satoh. "The SR-Tree: An index structure for high dimensional nearest neighbor queries," Proc. of SIGMOD, 1997.

[14] S. Berchtold, C. Bohm and H. Kriegel, "The

Pyramid-Technique: Towards Breaking the Curse of Dimensionality," Proc. of SIGMOD, 1998. pp 142-153.

[15] R. Weber, H. Schek and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," In Proc. of VLDB, 1998, pp 194-205.

[16] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree : An Index Structure for High-dimensional Feature Spaces," Proceedings of ICDE, pp. 440-447, 1999.

[17] P. Indyk and R. Motwani "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality," Proceedings of STOC, pp. 604-613, 1998.

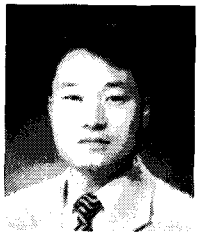
[18] Stefan Berchtold, Christian Bohm, H.V. Jargadish, Hans-Peter Kriegel and Jorg Sander, "Independent Quantization: An Index Compression Technique for High-Dimensional Data Spaces," ICDE 2000, Sandiego, CA.

[19] T. Seidl and H. P. Kriegel, "Optimal Multi-Step K-Nearest Neighbor Search," ACM SIGMOD, pp. 154-165, June 1998.

[20] N. Roussopoulos, S. Kelley and F. Vincent, "Nearest Neighbor Queries," Proc. of SIGMOD, 1995.

송 석 일

1998년 충북대학교 정보통신공학과(공학사)
2000년 충북대학교 정보통신공학과(공학석사)
2003년 충북대학교 정보통신공학과(공학박사)
2003년 3월 ~ 2003년 7월 KAIST 전산학과 박사후과정
2003년 8월~현재 충주대학교 전기전자정보공학부 컴퓨터공학과 전임강사
관심분야 : 데이터베이스 시스템, 트랜잭션, 저장 시스템, 멀티미디어 정보검색, XML, SAN 등
E-mail: sisong@chungju.ac.kr



이 석 희

1994년 충북대학교 정보통신공학과(공학사)
1998년 충북대학교 정보통신공학과(공학석사)
1998년~2001년 충북대학교 정보통신공학과(공학박사)
2000년~현재 동아 방송대학교 인터넷 방송학과 조교수
관심분야 : 데이터베이스 시스템, 정보검색, 영상처리, 분산 객체 컴퓨팅
E-mail: seoklee@dabc.ac.kr

조 기 형

1965년 인하대학교 전기공학과(공학사)
1984년 청주대학교 산업공학과(공학석사)
1992년 경희대학교 전자공학과(공학박사)
1981년~1988년 충주공업전문대학 조교수
1988년~현재 충북대학교 전기전자컴퓨터공학부 교수
관심분야 : 데이터베이스, 소프트웨어 시스템 설계 및 구현, 컴퓨터 네트워크 설계 등
E-mail: khjoe@cbucc.chungbuk.ac.kr

유 재 수

1989년 전북대학교 컴퓨터공학과(학사)
1991년 한국과학기술원 전산학과(공학석사)
1995년 한국과학기술원 전산학과(공학박사)
1995년~1996년 8월 목포대학교 전산통계학과 전임강사
1996년 8월~현재 충북대학교 전기전자컴퓨터공학부 부교수
관심분야 : 데이터베이스 시스템, XML, 멀티미디어 데이터베이스, SAN 소프트웨어 등
E-mail: yjs@cbucc.chungbuk.ac.kr

교신저자

유 재 수 361-763 충북 흥덕구 청주시 개신동 산 48번지 충북대학교 전기전자 및 컴퓨터공학부