한국지리정보학회지 5권 2호 : 54~68(2002)

대용량 데이터를 위한 효율적인 다차원 색인구조

이병엽[1] · 유재수[2※]

An Efficient Multi-Dimensional Index Structure for Large Data Set

ByoungYup LEE[1] · Jae-Soo YOO[2※]

요 약

최근 지리정보시스템, 움직임 객체관리시스템, 동영상/이미지 내용기반 검색시스템, 시계열 데이터베이스시스템과 같이 다차원 데이터를 이용하는 응용에 대한 관심이 고조되고 있다. 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 저장공간을 줄이기 위해서 VA-파일의 벡터 근사치 개념을 이용하여 데이터량이 증가해도 검색 성능이 저하되지 않도록 하는 트리 형태의 구조를 갖는다. VA-트리는 MBR 기반의 색인구조이지만 MBR 간에 겹침이 발생하지 않는 분할 방법을 사용하여 검색 효율을 높인다. 제안하는 색인구조와 기존의 여러 다차원 색인구조와의 성능 평가를 통해 제안하는 방법의 우수함을 보인다.

주요어: 색인구조, 다차원데이타, 지리정보시스템, GIS, MBR, 벡터근사치

ABSTRACT

In this paper, we propose a multi-dimensional index structure, called a VA(vector approximate)-tree that constructs a tree 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 the efficient index structure for large amount of multi-dimensional data.

KEYWORDS: Index Structure, Multi-Dimensional Data, GIS, MBR, Vector Approximates

2002년 5월 3일 접수 Received on May 3, 2002 / 2002년 6월 24일 심사완료 Accepted on June 24, 2002
1 대우정보시스템(주) Daewoo Information System, Inc.
2 충북대학교 컴퓨터정보통신연구소 Research Institute for Computer and Information Communication, Chungbuk Nat'l Univ.
※ 연락저자 E-mail: yjs@cbucc.chungbuk.ac.kr

서 론

최근 지리정보시스템, 움직임 객체관리시스템, 동영상/이미지 내용기반 검색시스템, 시계열 데이터베이스시스템과 같이 다차원 데이터를 이용하는 응용에 대한 관심이 고조되고 있다. 다차원 데이터에 대한 검색을 위해 다차원 색인구조가 사용된다는 것은 이미 잘 알려진 사실이다. 다차원 색인구조는 지난 20여년간 매우 활발히 진행되어 왔고 그 결과 KDB-트리(Robinson, 1981), hB-트리(Lomet와 Salzberg, 1989), GRID-파일(Nievergelt 등, 1984), BANG-파일(Freeston, 1987), R-트리(Guttman, 1984), R^+-트리(Sellis 등, 1987), R^*-트리(Beckmann 등, 1990), TV-트리(Lin, 등, 1994), SS-트리(White와 Jain, 1996a), VAMkd-트리(White와 Jain, 1996b), VAMSplitR-트리(White와 Jain, 1996b), X-트리(Berchtold 등, 1996), SR-트리(Katayama와 Satoh, 1997), 피라미드 기법(Berchtold 등, 1998), VA-파일(Weber 등, 1998) 등 매우 많은 수의 색인구조들이 제안되어 왔다.

기존에 제안된 색인구조들은 적게는 2~10차원, 많게는 11~200 또는 그 이상의 차원들을 대상으로 하고 있다. 하지만 대부분의 색인구조들은 10차원을 넘는 경우 색인구조로서의 기능을 상실하는 차원의 저주문제를 내포하고 있다. 이 문제를 해결하기 위해서 VA-화일이 제안되었다. 이 방법은 고차원의 데이터에 대해서는 트리형태의 색인구조는 무의미하므로 데이터를 표현하는 양을 줄여서 순차검색의 속도를 향상시키고 있다. 하지만 이 방법은 저차원의 데이터에 대해서는 기존의 트리구조의 색인구조에 비해서 성능이 떨어진다. 또한 순차 검색이므로 데이터의 양이 증가할수록 성능저하가 발생한다.

모든 다차원 색인구조의 응용이 10차원이 넘는 고차원의 데이터를 처리하는 것은 아니다. 특히, 지리정보 시스템이나 최근 부상되고 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 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-파일의 벡터 근사치 개념을 이용하면서 데이터량이 증가해도 검색성능이 저하되지 않도록 트리 형태를 취한다. 또한 VA-트리는 MBR 기반의 색인구조이지만 MBR 간에 겹침이 발생하지 않는 분할 방법을 사용하여 검색 효율을 높인다.

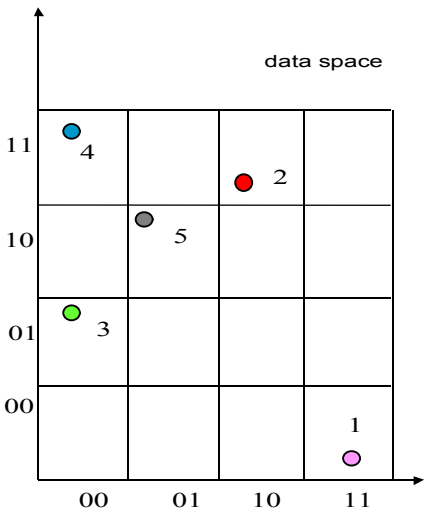
관련 연구

기존에 제안된 다차원 색인 구조들을 분류해보면 TV-트리, X-트리, SS-트리, SR-트리와 같은 데이터 분할을 사용하는 색인 구조와 KDB-트리, hB-트리, LSDh-트리, BANG 파일, GRID 파일과 같이 공간분할을 사용하는 색인 구조들로 크게 나누어 볼 수 있다. 또한, Hybrid-트리와 같은 이들의 혼합 형태의 색인구조가 존재하며 기타 LS(locality sensitive)해싱 기법을 사용하는 색인구조와 VA-파일와 IQ-트리처럼 요약 기법을 사용하는 색인 구조도 존재한다.

공간 분할을 사용하는 색인구조들은 공간을 서로 겹치지 않도록 분할하여 표현한다. 이들의 특징은 비 단말 노드의 엔트리의 크기가 차원과 독립적이다. 하지만 차원이 증가할수록 죽은 공간(dead space)이 증가하고 하향연쇄 분할(downward cascading split)의 빈도

수가 증가하여 저장공간 활용률이 현저히 떨어진다. 데이터 분할을 사용하는 색인 구조들은 MBR(minimum bounding region)형태로 비 단말 노드의 엔트리를 표현하기 때문에 죽은 공간이 없다. 또한 겹침을 허용하기 때문에 하향 연쇄분할 같은 문제는 발생하지 않는다. 하지만 MBR 표현 방식 때문에 차원이 증가할수록 비 단말 노드의 팬-아웃은 떨어지며 겹침 영역이 증가하게 된다. 이런 두 방법의 장점을 혼합한 방식이 Hybrid-트리이다. Hybrid-트리는 공간분할방식의 비 단말 노드 엔트리의 크기가 차원에 독립적이라는 특성과 MBR로 엔트리를 표현하며 겹침을 허용하여 하향 연쇄 분할을 피하고 있다. 하지만 완벽하게 비 단말 노드의 엔트리 크기가 차원과 독립적이라고 말할 수 없다.

이런 문제를 해결하는 또 다른 방법으로 피라미드 기법과 VA-화일을 들 수 있다. 피라미드 기법은 다차원 공간을 1차원으로 변환하여 B-트리를 이용하여 색인하는 방법이다. 이 방법의 최대의 문제점은 K-최근접 질의를 지원하지 못한다는 것이다. VA-파일은 피할 수 없는 순차 스캔을 가능한 빨리 수행하기 위하여 벡터 근사치를 이용하여 유사도 검색을 수행하는 방법이다. VA-파일은 그림 1과 같이 벡터 공간을 셀(cell)로 분할한다. 이 셀들은 각 특징벡터를 비트로 코드화한 벡터 근사치를 만드는데 사용한다. 이 방법은 GRID-파일과 분할 해싱 방법과 유사하다. VA-파일은 모든 벡터 근사치에 대한 간단한 배열구조의 파일이다. 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 계산해야하는 객체 수를 상당히 줄일 수 있으므로 디스크 I/O 회수가 그만큼 적어진다. VA-파일은 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.



	Vector data	VA-File
1	0.9 0.1	11 00
2	0.6 0.8	10 11
3	0.1 0.4	00 01
4	0.1 0.9	00 11
5	0.3 0.7	01 10

FIGURE 1. Structure of VA-file

이 논문에서는 이 문제를 해결하기 위해서 적은 수의 비트로 표현되는 벡터근사치들을 가지고 트리를 구성한다. 또한 트리의 검색성능 향상을 위해서 비단말 노드에 하위 노드에 대한 MBR을 저장하고 MBR이 겹치지 않는 분할 기법을 제안한다.

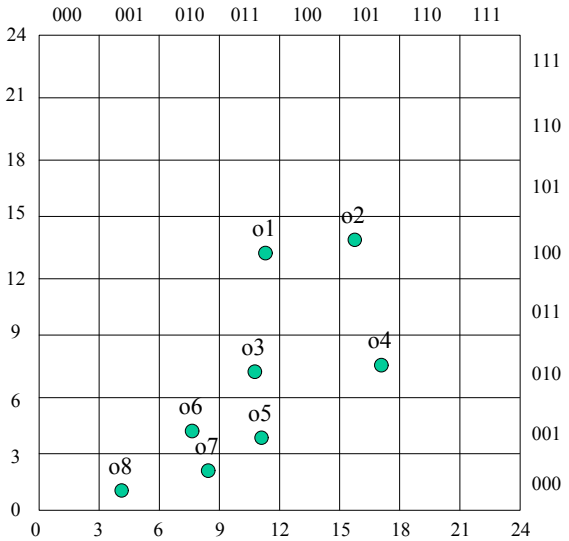
VA-Tree의 특징

1. 개요

VA-트리는 특징벡터를 벡터 근사치로 매핑시킨 후, 이 벡터 근사치를 이용하여 트리를 구성한다. K-D-B트리 기반의 다른 색인 구조처럼 노드간 겹침은 없다. 노드는 중간노드와 단말노드로 구분된다. 중간노드 구조는 [하위노드를 포함하는 영역 비트, 하위노드에 대한 포인터], 단말노드 구조는 [벡터 근사치, 실제 데이터 레코드에 대한 포인터]로 구성된다. 중간노드는 하위 중간노드 또는 단말노드를 포함하는 영역 정보를 비트 형태로 가지고 있다. VA-트리에서는 유사도 탐색시 최소 거리 경계만을 사용한다. 중간노드의 영역 비트 정보만을 보고 질의 포인트에서 중간노드까지의 최소 거리 경계를 계산한다. 또한 단말노드의 벡터 근사치를 이용하여 질의 포인트에서 벡터 근사치까지의 최소 거리 경계를 계산한다. 이러한 최소 거리 경계를 이용하여 효과적인 가지치기를 수행할 수 있으므로 빠른 검색 성능을 보인다.

2. 벡터 근사치 표현

특징벡터를 벡터 근사치로 매핑시키는 방법은 VA-파일과 유사하다. 각 차원당 b비트로 표현할 때, 각 차원의 분할영역 수는 2^b, 분할포인트 수는 2^b+1개가 발생한다. 모든 DIM 차원을 고려했을 때 분할 영역 개수(전체 셀 개수)는 $(2^b)^{DIM}$이다. 그림 2에서 각 차원의 최소영역과 최대영역이 0에서 24라고 할 때, 차원당 3비트로 표현하고 균등하게 분할하면, 각 차원의 분할 포인트 값은 [0, 3, 6, 9, 12, 15, 18, 21, 24]가 되고 전체 셀의 개수는 64개이다. 분할된 데이터 공간에 2차원의 특징벡터가 8개가 놓여 있을 때, 특징벡터를



FITURE 2. Vector space

벡터 근사치로 표현하면 표 1과 같다. 한 예로 객체 1이 속해 있는 셀이 (011 100)이므로 이 값이 객체 1의 벡터 근사치가 된다. VA-파일과 Hybrid-트리에서 벡터 근사치로 표현할 때 각 차원 당 4~5비트로 표현하는 것이 성능이 좋다고 제시하고 있다.

TABLE 1. Vector approximates

	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)

3. 분할 영역 표현

분할 영역은 R-트리 기반 색인 구조에서 MBR에 해당하는 것이다. 하위 노드를 모두 포함할 수 있는 영역을 표현하는 방법이다. 그림 3과 그림 4는 예제 데이터를 이용하여 분할 영역을 표현한 예이다. 그림 4는 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.

중간노드에는 하위노드들이 포함하고 있는 모든 영역을 표시하기 위하여 하위영역을 나타내는 벡터근사치와 상위영역을 나타내는 벡터근사치를 내포하고 있다. 단말노드 (5)가 포함하고 있는 엔트리는 1, 2번 객체이다. 각각의 객체를 나타내는 벡터근사치는 [(011, 100), (101, 100)]이다. 단말노드 (5)가 포함된 상위 노드에는 단말노드(5)의 영역정보를 나타내는 벡터근사치 [(011, 100) (101, 100)]과 포인터가 포함된다. 이것으로 단말노드(5)가 y축을 기준으로 분할되었음을 알 수 있다. 단말노드 (1), (2), (4)를 포함하고 있는 중간노드 (A)의 영역정보를 나타내는 방법도 앞서 설명한 방법과 같다. 단말노드 (1), (2), (4)의 모든 객체를 포함하는 영역의 벡터 근사치는 [(001, 000),

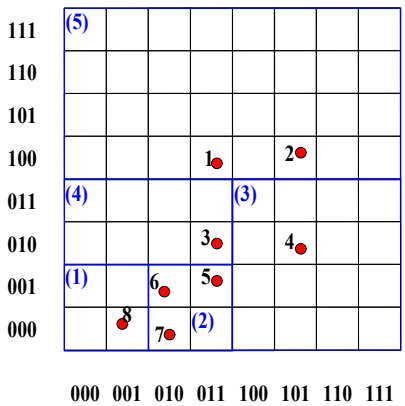


FIGURE 3. Vecotr approximate space

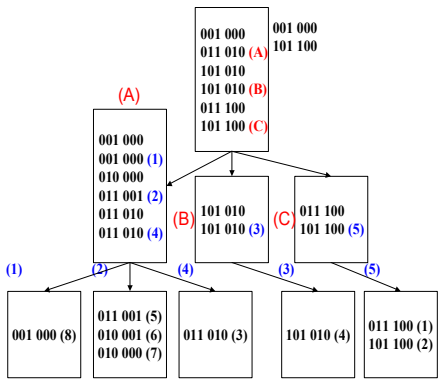


FIGURE 4. Construction of VA-tree

(011, 010)]이다. 그러므로 상위노드에서 중간 노드 (A)를 나타내는 영역의 벡터근사치는 [(001, 000), (011, 010)]이다. 이렇게 영역정보를 나타냄으로써 데드 스페이스를 최소로 줄이고 실제 객체가 포함된 영역인 라이브 스페이스만이 나타나도록 영역정보를 표현할 수 있다.

4. VA-트리 구성

그림 4의 VA-트리는 단말노드와 비단말노드에 최대로 저장될수 있는 엔트리의 수를 3으로 가정한 예이다. 여기에서 볼 수 있듯이 VA-트리는 K-D-B-트리를 기반으로 하고 있다. 벡터 근사치 4개(#1, #2, #3, #4)가 입력되면 단말노드의 분할이 일어난다. 단말노드 분할 기준에 의해 분할 차원과 위치를 선택한다. 2차원에서 분할이 일어나기 때문에 벡터근사치 #1, #2를 포함하는 노드와 #3, #4를 포함하는 노드로 분할된다. 그리고 중간노드에 2차원으로 분할된 단말노드에 대한 영역정보 [(011 100), (101, 100)], [(011, 010), (101 010)]가 상위에 반영된다.

다시 #5, #6이 입력되면 단말 노드 3은 1차원에 대해서 분할이 발생되어 #3, #6, #5을 포함하는 노드와 #4을 포함하는 노드로 분할된다. 그리고 두 노드에 대한 정보를 상위 노드에 반영한다. #7이 입력되면 다시 #3, #6, #5를 포함하는 노드에 넘침이 발생되고 #3을 포함하는 노드와 #5, #6, #7을 포함하는 노드로 분할된다. 이 분할을 상위에 반영할 때 루트노드에 넘침이 발생하게 되고 루트 노드가 분할되게 된다. 분할하는 방법에 대한 자세한 내용은 뒤의 분할 알고리즘에서 설명한다.

5. 최소 거리

질의 포인트와 특징벡터 간의 경계 거리를 유도하기 위하여 벡터 근사치를 이용한다. 질의 벡터 $\overrightarrow{v_q}$에서부터 특징벡터 $\overrightarrow{v_i}$ 까지의 실제 거리 함수로 R_p가 있고, 특징벡터 $\overrightarrow{v_i}$가 포함되어 있는 셀에서 부터 질의벡터 $\overrightarrow{v_q}$까지의 최소 거리 경계 l_i가 있다. 최소 거리 경계 l_i는 질의에서부터 셀까지 가장 짧은 거리이다. l_i과 R_p의 관계는 $l_i \le R_p(\overrightarrow{v_q}, \overrightarrow{v_i})$이다.

벡터간의 실제거리 R_p가 근사 거리인 최소 거리 경계 보다 항상 크기 때문에 적중 착오가 발생하지 않는다. 특징벡터 $\overrightarrow{v_i}$가 포함되어있는 비트 형태의 벡터 근사치는 셀까지의 최소 거리 경계를 계산하기 충분한 정보를 포함하고 있다. 그림 5에서 점선은 질의 벡터와 중간노드까지의 최소 거리 경계이고, 실선은 벡터 근사치까지의 최소 거리 경계이다. 탐색시 최소 거리 경계를 이용하여 중간노드나 상당수의 벡터 근사치를 가지치기할 수 있기 때문에 CPU 연산 비용을 줄일 수 있으므로 검색성능이 향상된다.

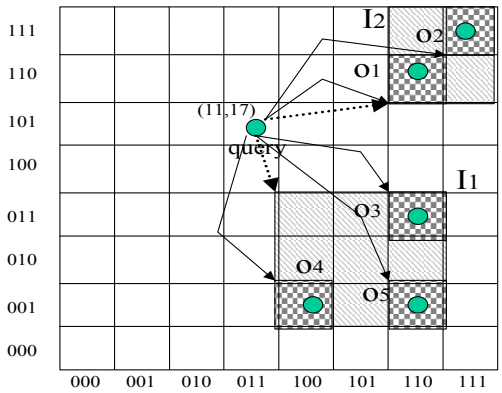


FIGURE 5. Computation of minimum distance

질의 포인트 q(11, 17)에서부터 벡터 근사치 o_1(110, 110)까지의 최소 거리 경계는 다음과 같이 계산된다. 먼저 질의 포인트 q(11, 17)의 벡터 근사치 (011, 101)로 변경된다. 그런 다음 질의의 벡터 근사치 q(011, 101)과 객체의 벡터 근사치 o_1(110, 110)와의 거리가 계산된다 따라서 질의 q와 객체 o_1의 거리는 $(3-6)^2+(5-6)^2 = 10$ 이다.

질의 포인트 q(11, 17)에서부터 중간노드 I_1[(100, 001), (110, 011)]까지의 최소 거리 경계는

다음과 같이 구한다. 질의 포인트의 벡터 근사치 q(011 101)와 중간노드 [(100, 001), (110, 011)]의 첫 번째 비트를 비교한다. 1차원의 첫 번째 비트를 보고 중간노드가 더 크다는 것을 알 수 있고, 2차원의 첫 번째 비트를 보고 중간노드가 더 작다는 것을 알 수 있다. 따라서 최소 거리 경계는 (노드 경계점의 벡터 근사치 - 질의 포인트의 벡터 근사치)로 $(3-4)^2+(5-3)^2 = 5$ 이다.

6. VA-Tree의 삽입, 삭제, 검색 알고리즘

1) 삽입 알고리즘

새로운 특징벡터를 삽입하기 위하여 새 특징벡터를 벡터 근사치로 매핑시킨다. 벡터 근사치를 포함하는 가지들을 선택하면서 탐색하여 단말노드에 도달하면 그 노드에 벡터 근사치를 삽입한다. 삽입시 단말노드에 오버플로우가 발생되면 단말노드 분할 알고리즘에 따라 분할하고, 중간노드의 오버플로우는 중간노드 분할 알고리즘에 의해 처리한다.

```
Procedure Insert

Start procedure
1  입력 특징벡터(F)를 벡터근사치(V)로 변환
2  Leafnode := Newent를 삽입할 단말 노드를
찾는다(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
```

FIGURE 6. Insertion algorithm

그림 6에서 1행은 특징벡터(F)를 벡터 근사치(V)로 변환한다. 2행은 벡터 근사치를 삽입할 단말노드를 LocateNode 함수를 이용하여 검색한다. 3~7행은 새로운 벡터 근사치를 삽입한 단말노드에 오버플로우가 발생하면 분할을 수행하기 위해 SplitNode 함수를 호출하고, 그렇지 않은 경우에는 단말노드에 새로운 벡터 근사치를 삽입한 후 노드의 엔트리 수를 하나 증가시킨다.

2) 단말노드 검색(LocateNode) 알고리즘

새로운 벡터 근사치가 삽입될 단말노드를 루트노드부터 검색하여 찾아내는 함수이다. 중간노드에서 가장 적합한 하위 노드를 찾는 방법은 다음과 같다. 먼저 새로운 엔트리를 포함할 수 있도록 노드의 첫 번째 MBR을 확장한다(ExMBR). 그리고 ExMBR과 그외의 다른 MBR과의 겹침이 있는지를 확인한다. 만일 겹침이 없으면 그것이 가장 적정한 하위 노드가 된다. 만일 겹침이 있으면 두 번째 MBR을 확장해서 ExMBR을 만들고 이와 다른 MBR들과의 겹침을 검사하여 겹침을 검사한다. 이를 반복하여 확장한 MBR과 겹침이 발생하지 않는 때를 찾아낸다. 다시 선택한 하위 노드로 가서 이 노드가 단말노드이면 순회를 멈추고 그렇지 않으면 다시 방금 수행했던 일을 반복한다. 이런 식으로 루트노드부터 적정한 단말노드를 찾아내고 이를 찾을 때까지의 경로를 스택에 저장하여 반환한다.

```
Procedure LocateNode

Start procedure
1  If (Node 가 단말 노드)
2     return Node;
3  End If
4  for ( each MBR of Node )
5     ExMBR = CurMBR을 NewEnt를 포함하도록
확장;
6     ExMBR 과 CurMBR을 제외한 다른 MBR 과
겹침이 있는지 조사;
7     If ( 겹침이 없으면 )
8        break;
9     Else
10       CurMBR = 다음 MBR;
11 End for
12 CurMBR에 해당하는 자식노드에 대해서 1부터 반복
End procedure
```

FIGURE 7. Retrieval algorithm of terminal nodes

3) 분할(SplitNode) 알고리즘

분할 함수가 호출되면 이는 단말노드가 분할

되는 것이다. 분할된 단말노드를 상위 노드에 반영하고 이를 반영할 때 상위노드에 오버플로우가 발생하면 인터널 노드를 분할하여 이를 상위노드에 반영한다. 이러한 과정을 인터널 노드에 오버플로우가 발생하지 않을 때까지 반복한다. 노드를 분할할 때 분할 차원과 분할 위치를 결정해야 한다. 분할 위치를 결정하는 일반적인 방법은 [8 9 10]에서처럼 중간 값이다. 중

```
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

Procedure SplitLeafNode

Start procedure

1  분할을 수행할 차원(dim)과 비트(bit) 결정;
2  bit를 이용해서 분할 위치(splitpos) 결정;

/* firstbit : 1000, secondbit : 0100 */
3  if ( bit == firstbit )
4     splitpos = 7;
5  else if ( bit == secondbit )
6     splitpos = 3  (mbr 의 dim 값이 3  ~ 4 를 포함 );
7     splitpos =11 : 11 ~ 12
8  else if ( bit == thirdbit )
9     splitpos =1 :  1  ~ 2
10    splitpos =5 :  5  ~ 6
11    splitpos =9 :  9  ~ 10
12    splitpos =13 : 13 ~ 14
13 else if ( bit == fourthbit )
14    splitpos =0 :  0  ~ 1
15    splitpos =2 :  2  ~ 3
16    splitpos =4 :  4  ~ 5
17    splitpos =6 :  6  ~ 7
18    splitpos =7 :  7  ~ 8
19    splitpos =8 :  8  ~ 9
20    splitpos =10 : 10 ~ 11
21    splitpos =12 : 12 ~ 13
22    splitpos =14 : 14 ~ 15
23 end if

24 dim 과 splitpos를 이용해서 노드 분할수행
25 분할 된 두 노드의 ent 생성 및 반환 ( oldent, splitent );

End procedure

Procedure SplitInternalNode


Start procedure

/* 겹침 없이 분할 가능한 차원이 있는지 검사한다. */
1  for ( all dimensions )
2     if ( mbr의 해당차원 값이 7~8 포함 )
3        splitpos= 7;
4     else if ( mbr의 해당차원 값이 4~3, 11~12 포함)
5        splitpos = 3 ; (3~4)
6        splitpos = 11; (11~12)
7     else if (mbr의 해당차원 값이
                1~2, 5~6, 9~10, 10~11)
8        splitpos = 1; (1~2)
9        splitpos = 5; (5~6)
10       splitpos = 9; (9~10)
11       splitpos = 13; (13~14)
12    else if (mbr의 해당차원 값이 0~1, 2~3, 4~5,
                6~7, 8~9, 10~11, 12~13, 14~15)
13       splitpos = 0; (0~1)
14       splitpos = 2; (2~3)
15       splitpos = 4; (4~5)
16       splitpos = 6; (6~7)
17       splitpos = 8; (8~9)
18       splitpos = 10; (10~11)
19       splitpos = 12; (12~13)
20       splitpos = 14; (14~15)
21    end if
22    splitable_dim = splitpos을 기준으로
      겹침 없이 분할 가능;
23 end for

24 if ( splitable_dim > 1 )
25    for ( 모든 splitable_dim 에 대해서 )
26       splitable_dim = 균등하게 분할되는 차원 선택;
27    end for
28 end if
29 if ( splitable_dim > 1 )
30    for ( 모든 splitable_dim 에 대해서 )
31       splitable_dim = 가장 범위가 큰 차원 선택
32    end for
33 end if
34 splitable_dim 과 splitpos 에 따라서 노드를 분할;
35 두 노드에 대한 엔트리 생성 및 반환
   (oldent,splitent);

End procedure
```

FIGURE 8. Split algorithm

간 값을 선택하는 것은 분할 후 생성된 두 개 노드에 데이터가 균등하게 분포되어있기 때문이다. 데이터가 균일한 분포의 어플리케이션에서는 한 노드를 동일한 크기의 영역을 가지는 두 개 노드로 분할하는 것이 유리하다.

4) 단말노드 분할

분할을 수행하기 위해서 가장 먼저 분할 차원과 분할 위치를 결정한다. 이후의 설명은 사용하는 비트의 수가 4일 때를 가정한다. 각 비트별로 각 차원에 대해서 분할이 가능한지를 검사한다. 예를 들어서 (1101 0010), (1110 0001), (1010 0100), (1011 0000)과 같이 4개의 MBR이 노드에 존재한다고 하자. 이 예에서 첫 번째 비트에 대해서는 이미 1, 2차원 모두 분할이 한번 수행되었기 때문에(1, 2차원 모두 첫 번째 비트가 모두 같다) 첫 번째 비트는 분할 대상에서 제외된다. 이 예에서 가능한 비트는 두 번째 비트가 되며 이 비트에 대해서 1 차원으로 분할을 수행한다.

실질적인 단말노드의 분할 기준은 다음과 같다. 다음 기준은 일단 그 비트가 분할이 가능한 비트일 경우에 해당한다.

기준 1. 각 차원별로 분할될 위치의 1,0 비트 편차가 가장 적은 것을 선택

기준 2. 1,0의 비트 편차가 적은 차원들 중 [최대치-최소치]가 큰 것을 선택

분할 기준 1은 데이터가 골고루 분포되어 있는 차원을 선택하기 위함이다. 이때 2개 이상의 차원이 분할 기준 1을 만족하면 분할 기준 2를 적용하여 분할 차원을 선택한다. 분할 기준 2는 데이터가 넓게 분포되어 있는 차원을 선택하기 위함이다.

5) 중간노드 분할

중간노드의 분할은 다음과 같이 수행한다. 먼저 분할이 가능한 비트를 선택한다. 분할이 가능하다는 것은 그 비트를 기준으로 분할을 했을 때 하위노드들이 그 분할위치에 걸치지 않음을 말한다. 먼저 분할이 가능한 비트와 차원을 선택한 후 다음의 기준을 이용해서 분할을 수행한다.

기준 1. 데이터가 넓게 분포되어 있는 차원을 선택

기준 2. 1,0의 비트 편차가 적은 차원을 선택

분할 기준 1에서 단말노드와 마찬가지로 데이터가 넓게 분포되어 있는 차원을 선택한다. 이때 2개 이상의 차원이 분할 기준 1을 만족하면 분할 기준 2를 적용한다.

6) 삭제 알고리즘

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

```
Procedure Delete

Start procedure

1  입력 특징벡터(F)를 벡터근사치(V)로 변환
2  Leafnode := Targetent를 삭제할 단말 노드를
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
```

FIGURE 9. Deletion algorithm

7) 검색 알고리즘

VA-트리에서 유사도 검색은 다른 색인 구

조와 마찬가지로 루트부터 시작하여 단말노드까지 각 노드별로 엔트리들을 검사하여 수행하는데, 이 논문에서는(Seidl과 Kriegel, 1998)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 중간노드의 비트 정보만을 이용하여 질의 포인트에서 중간노드까지의 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.

유사도 검색을 위해 그림 10에서와 같이 후보집합, 결과집합 우선 순위 큐 두개를 사용한다. 후보집합에는 질의 포인트에서 벡터 근사치나 중간노드까지의 근사 거리 순으로 정렬되고, 결과집합에는 질의 포인트에서 실제 데이터까지의 거리 순으로 정렬된다. 결과집합의 k번째 거리를 나타내는 K번째 거리 값(result_k_dist)은 초기치 ∞에서 점점 작아진다. 루트부터 트리를 순회하면서 질의 포인트에서 중간노드까지의 거리를 계산하여 후보집합에 넣는다. 3행은 질의 포인트부터 지금까지 방문한 K번째 데이터까지의 실제거리가 후보집합내의 거리보다 작으면 탐색을 끝내고 결과집합에서 k개를 반환한다. 4~6행은 후보집합에서 읽은 데이터가 중간노드이면 그 중간노드내 엔트리까지의 근사 거리를 계산하여 후보집합에 저장한다. 7~9행은 후보집합에서 꺼낸 데이터가 단말노드이면 그 단말노드내의 벡터 근사치까지의 거리를 계산하여 후보집합에 저장한다. 10~12행은 후보집합에서 읽은 데이터가 벡터 근사치이면 질의 포인트와의 거리를 계산하여 결과집합에 저장하고, K번째 거리 값을 갱신한다.

```
Procedure Search

Start procedure
1  결과집합, 후보집합, K번째 거리 = ∞
2  질의포인트 ==> 비트근사치로 변환
3  while ( K번째 거리  > 후보집합 첫번째 거리)
4    if (후보집합 첫번째가 중간노드)
5      중간노드내의 엔트리까지의 거리 계산
6      K번째 거리보다 작은 엔트리를 후보집합에
삽입
7    else if (후보집합 첫번째가 단말노드)
8      단말노드내의 엔트리까지의 거리 계산
9      K번째 거리보다 작은 엔트리를 후보집합에
삽입
10   else if (후보집합 첫번째가 비트근사치)
11     질의에서 비트근사치까지의 거리 계산
12     결과집합에 삽입
13     K번째 거리를 갱신
14   end if
15 end while
14 결과집합에서 K개를 반환
End procedure
```

FIGURE 10. Retrieval algorithm

실험 및 결과 분석

여기서는 제안된 VA-트리를 구현하여 실험한 성능평가 결과를 기술한다. 제안된 VA-트리의 검색 성능을 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. 컴파일러는 gcc 2.7.1.이다.

이 논문에서 비교평가 대상으로 사용된 기존의 색인구조는 논문의 저자가 제공해준 원시 프로그램의 내용을 수정하지 않고 사용하였다. 한 차원의 벡터 데이터를 표현하기 위한 비트수는 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 만들

어 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. (Weber 등, 1998)

TABLE 2. Input parameters

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	4kB~32kB

TABLE 3. Performance evaluation parameters

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

그림 11은 VA-트리와 R*-트리, X-트리의 삽입 시간을 비교한 결과이다. 노드크기를 32kB, 데이터 차원을 10차원, 데이터 개수를 200,000개에서 1,000,000개까지 늘려가면서 색인을 구성할 때 걸리는 시간을 측정하였다. 측정 결과 VA-트리를 구축하는데 가장 적은 시간이 소요됨을 확인할 수 있었다. 특히, 데이터 개수가 늘어남에 따라 색인 구축시간의 차이가 점점 커지는 것을 확인할 수 있었다. 삽입 알고리즘은 가장 복잡한 단계를 거친다. 그러므로 삽입 시간이 빠르다는 것은 삭제 알고리즘과 같은 다른 연산의 속도가 빠르다는 것을 의미한다고 볼 수 있다.

그림 12와 그림 13은 균일 분포 데이터 집합들을 이용하여 VA-트리와 VA-파일, R*-트리, X-트리에서 K-NN 질의를 수행한 경우에 대한 검색 성능을 비교한 것이다. 그림 12는 노드크기를 4kB, 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. 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 중간노드의 영역정보가 오버랩이 없다. 이런 이유로 접근하지 않는 노드의

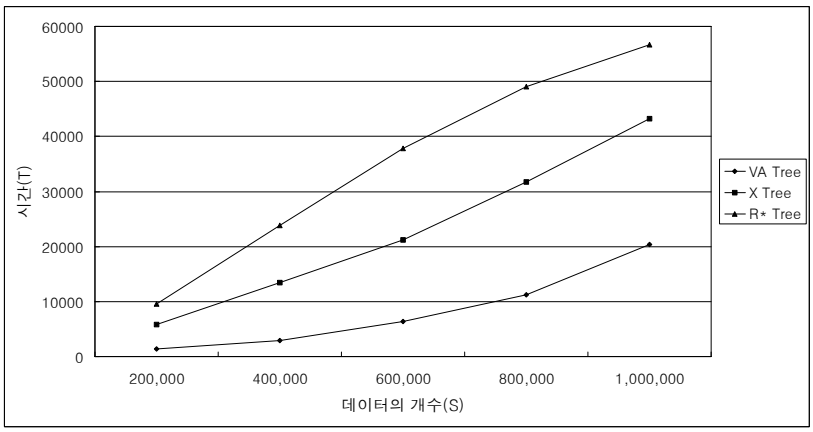
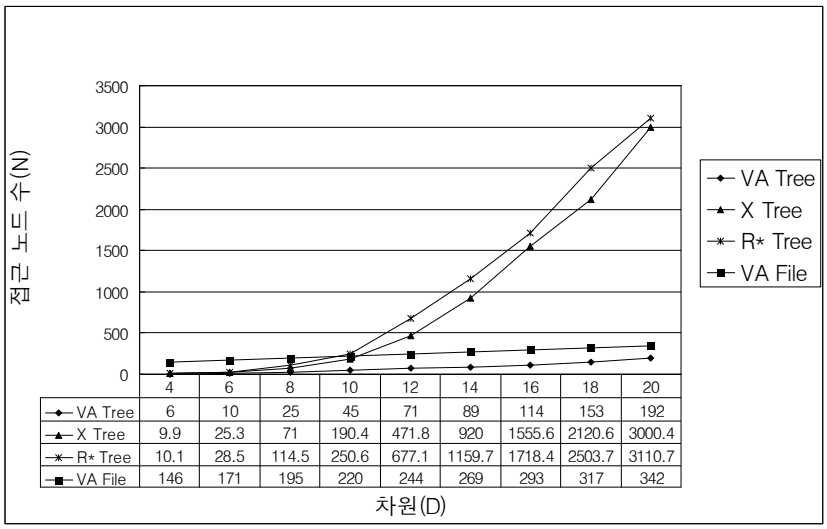


FIGURE 11. Comparison of tree construction time



	4	6	8	10	12	14	16	18	20
VA Tree	6	10	25	45	71	89	114	153	192
X Tree	9.9	25.3	71	190.4	471.8	920	1555.6	2120.6	3000.4
R* Tree	10.1	28.5	114.5	250.6	677.1	1159.7	1718.4	2503.7	3110.7
VA File	146	171	195	220	244	269	293	317	342

FIGURE 12. Comparison of accessed nodes according to the increase of dimension

수가 많아지게 되는 것이다.

그림 13은 노드크기를 4kB, 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-파일은 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. 이 결과에 따르면 VA-트리는 기존의 색인구조에 비해 성능이 우수함을 보인다.

그림 14는 노드크기를 4kB, 차원을 9차원, 데이터 개수를 300,000개의 실제 데이터를 이용해 색인을 구성한 후, 주어진 질의 데이터에

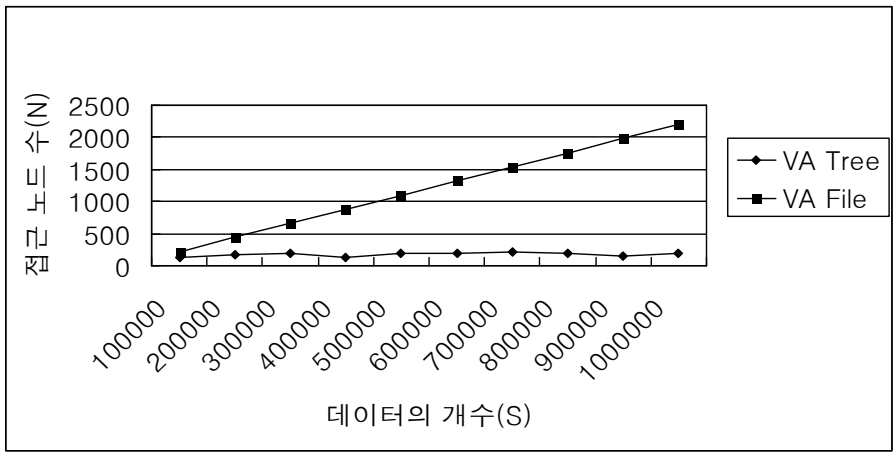


FIGURE 13. Comparison of accessed nodes according to the increase of data

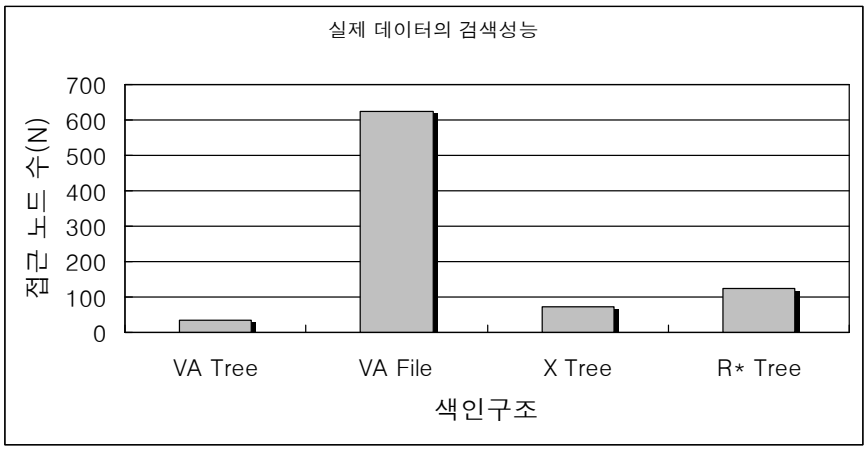


FIGURE 14. Comparison of accessed nodes on real data sets

대해서 10개의 근접 데이터를 100번 검색했을 때 접근되는 평균 접근 노드 수를 측정한 결과이다. 데이터는 동영상의 각각 장면에서 장면을 특징 지우는 색상정보를 9개의 수치 데이터로 추출하여 사용하였다. 실제 데이터에 대한 성능측정 결과에서도 VA-트리는 기존의 색인구조에 비해 성능이 우수함을 보였다.

저장공간 활용율을 평가하기 위해 노드크기를 4kB, 데이터 개수를 100,000개, 차원을 4에서 20까지 늘려가면서 색인을 구성한 후 생성된 노드의 개수(M)를 그림 15에서 보여준다. 그림 15를 보면 R*-트리와 X-트리는 생성되는 노드의 개수가 지수적으로 증가되는 것을 알 수 있다. 또한 제안한 VA-트리가 저장공간활용 측면에서도 뛰어난 성능을 보임을 알 수 있다. VA-트리가 VA-파일에 비해 많은 노드가 생성되는 이유는 VA-파일은 트리 형태로 구축하지 않고 순서대로 단순히 저장만하기 때문이다. 그러나 이러한 특징으로 대용량 데이터에 대해서는 검색성능이 급격히 떨어지는 문제점이 발생한다.

그림 16과 그림 17은 다차원 색인구조 중에 최근에 제안된 피라미드 기법과의 검색성능을 비교한 것이다. 성능측정은 유사도를 5%에서

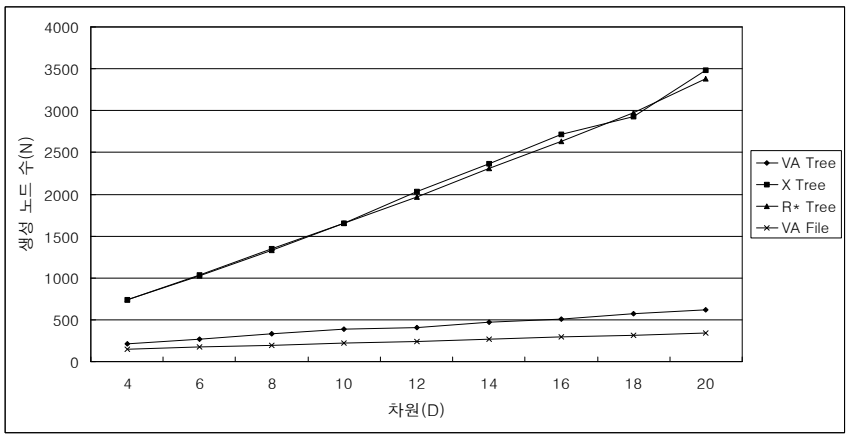


FIGURE 15. Space overhead of each method

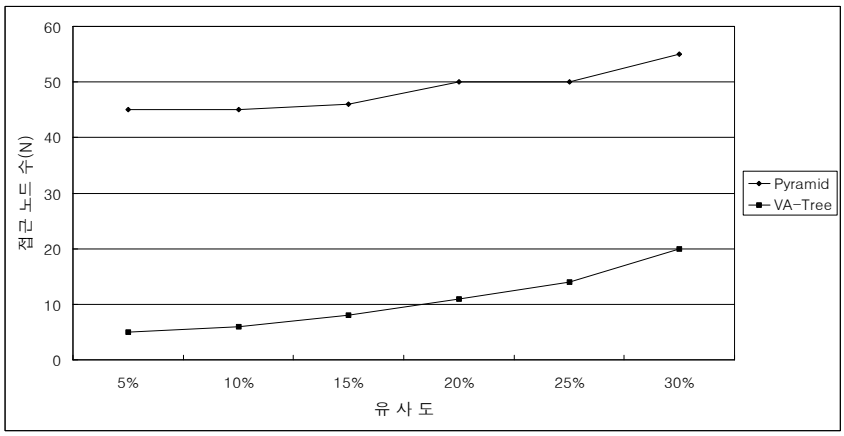


FIGURE 16. Comparison of accessed nodes in the eight dimensions data set

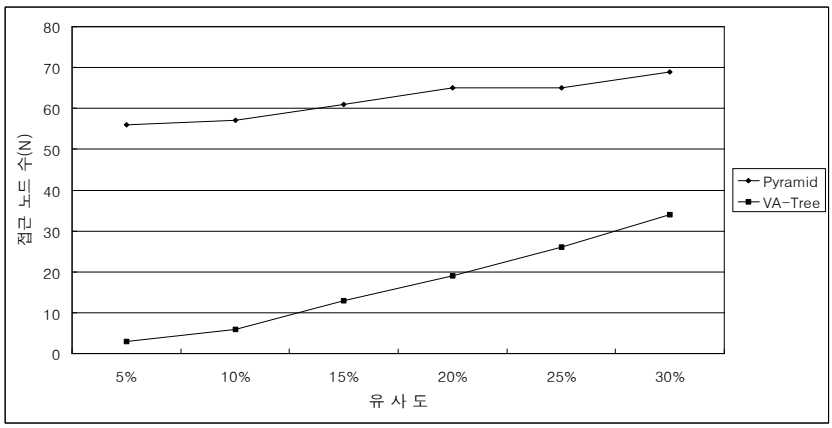


FIGURE 17. Comparison of accessed nodes in the sixteen dimensions data set

30%까지 늘려가면서 접근되는 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 16은 노드크기 16kB, 8차원, 200,000개 데이터에 대한 실험 결과이다. VA-트리가 피라미드 기법에 비해 평균 78%의 성능 향상을 확인하였다. 그림 17은 노드크기 16kB, 16차원, 200,000개 데이터에 대한 실험 결과이다. 이 경우에는 VA-트리가 피라미드 기법에 비해 평균 73%의 성능향상을 확인하였다.

결론 및 향후 연구

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 2~10차원에서 기존의 색인구조에 비해서 매우 뛰어난 검색

성능을 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 동시성 제어 및 회복 기법에 대한 연구를 수행한다. KAGIS

참고문헌

Beckmann, N., H. Kriegel, R. Schneider and B. Seeger. 1990. The R*-tree: An efficient and robust access method for points and rectangles ACM SIGMOD. pp.322-331.

Berchtold, S., D.A. Keim and H. Kriegel. 1996. The X-tree: An index structure for high-dimensional data. Proceedings of the 22th on VLDB. pp.28-39.

Berchtold, S., C. Bohm and H. Kriegel. 1998. The pyramid-technique: Towards breaking the curse of dimensionality. Proceedings of SIGMOD. pp.142-153.

Freeston, M. 1987. The BANG file: a new kind of grid file. Proceedings of VLDB Conference. pp.260-269.

Guttman, A. 1984. R-trees: A dynamic index structure for spatial searching. ACM SIGMOD. pp.47-57.

Katayama, N. and S. Satoh. 1997. The SR-tree: An index structure for high dimensional nearest neighbor queries. Proceedings of SIGMOD.

Lin, K.I., H. Jagadish and C. Faloutsos. 1994. The TV-tree: An index structure for high-dimensional data. The VLDB Journal 3(4):517-549.

Lomet, D. and B. Salzberg. 1989. The hB-tree: A robust multiattribute search structure. Proceedings of ICDE Conference. pp.296-304.

Nievergelt, J., H. Hinterberger and K.C. Sevcik. 1984. The Grid file: An adaptable, symmetric multikey file structure. ACM Transactions on Database Systems. pp.38-71.

Robinson, J.T. 1981. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. ACM SIGMOD. pp.10-18.

Sellis, T., N. Roussopoulos and C. Faloustos. 1987. The R+-tree: A dynamic index for multi-dimensional objects. Proceedings of the 13th International Conference on VLDB. pp.507-518.

Seidl, T. and H. Kriegel. 1998. Optimal multi-step K-nearest neighbor search. ACM SIGMOD. pp.154-165.

Weber, R., H. Schek and S. Blott. 1998. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. Proceedings of VLDB. pp.194-205.

White, D.A. and R. Jain. 1996a. Similarity Indexing with the SS-tree. Proceedings of the 12th International Conference On Data Engineering. pp.515-523.

White, D.A. and R. Jain. 1996b. Similarity indexing: Algorithms and performance. Proceedings of SPIE: Storage and Retrieval for Image and Video Databases IV. pp.62-73. KAGIS