고차원 벡터 데이터 색인을 위한 시그니쳐-기반 Hybrid Spill-Tree의 설계 및 성능평가☆

Design and Performance Analysis of Signature-Based Hybrid Spill-Tree for Indexing High Dimensional Vector Data

이 현 조* 홍 승 태** 나 소 라*** 장 유 진**** 장 재 우***** 심 춘 보*
Hyun-jo Lee Seung-Tae Hong So-Ra Na You-Jin Jang Jae-Woo Chang Choon-Bo Shim

요 약

최근 UCC를 중심으로 동영상 데이터에 대해 사람들의 관심이 증가하고 있다. 따라서 동영상 데이터의 내용-기반 검색을 지원하는 효율적인 색인 기법이 요구된다. 그러나 Hybrid Spill-Tree를 제외한 대부분의 색인 기법들은 대용량의 고차원 데이터를 다루는데 비효율적이다. 본 논문에서는 동영상 데이터의 내용-기반 검색을 지원하기 위한 효율적인 고차원 색인 기법을 제안한다. 제안하는 고차원 색인 기법은 기존 Hybrid Spill-Tree을 기반으로 새롭게 제안하는 클러스터링 방법과 시그니쳐를 이용한 데이터 저장 방법을 결합하여 확장된 색인 기법이다. 또한 제안하는 시그니쳐-기반 고차원 색인 기법이 기존 M-Tree및 Hybrid Spill-Tree에 비해 성능이 우수함을 보인다.

ABSTRACT

Recently, video data has attracted many interest. That is the reason why efficient indexing schemes are required to support the content-based retrieval of video data. But most indexing schemes are not suitable for indexing a high-dimensional data except Hybrid Spill-Tree. In this paper, we propose an efficient high-dimensional indexing scheme to support the content-based retrieval of video data. For this, we extend Hybrid Spill-Tree by using a newly designed clustering technique and by adopting a signature method. Finally, we show that proposed signature-based high dimensional indexing scheme achieves better retrieval performance than existing M-Tree and Hybrid Spill-Tree.

☞ KeyWords : Video Data, High-dimensional indexing scheme, Hybrid Spill Tree, 동영상 데이터, 고차원 색인 기법

1. 서 론

최근 인터넷 포탈 시스템에서 UCC(User Created Contents)등의 동영상 콘텐츠(contents)의 중요성이 점점 증가하고 있다. 대표적인 동영상 사이트인 유튜브(YouTube)의 경우 하루 평균 약 7만건의 동영상이 새로 등록되고 있으며, 하루 평균 600만명이 방문하고 1억여 건에 이르는 동영상을 재생하고 있다. 또한 국내의 다음(DAUM) UCC의 경우, 하루 약 15,000건의 동영상이 새로

* 정 회 원 : 전북대학교 컴퓨터공학과 박사과정
hjlee@dblab.chonbuk.ac.kr
** 준 회 원 : 전북대학교 컴퓨터공학과 석사과정
sthong@dblab.chonbuk.ac.kr
*** 준 회 원 : 전북대학교 컴퓨터공학과 석사과정
srna@dblab.chonbuk.ac.kr
**** 준 회 원 : 전북대학교 컴퓨터공학과 석사과정
yjjang@dblab.chonbuk.ac.kr
***** 정 회 원 : 전북대학교 전기전자컴퓨터공학부 교수
jwchang@chonbuk.ac.kr(교신저자)
****** 정 회 원 : 순천대학교 정보통신공학부 조교수
cbsim@sunchon.ac.kr
[2008/11/18 투고 - 2008/12/09 심사(2009/02/06 2차 - 2009/04/10 3차) - 2009/05/12 심사완료]
☆ 본 연구는 교육과학기술부와 한국산업기술재단의 지역혁신 인력 양성 사업으로 수행된 연구 결과임
☆ 본 연구는 지식경제부의 대학 IT연구센터 지원사업의 연구 결과로 수행되었음 (IITA-2009-(C1090-0902-0047))

등록되고 있으며, 하루 평균 약 310만 여명이 방문하고 있다.

동영상 데이터는 기존의 텍스트 기반 데이터와 달리 이미지, 사운드, 애니메이션, 비디오 등 여러 형태의 데이터들이 결합되어 이루어져 있다. 그러나 기존의 단순 키워드 기반 검색은 정보 검색의 정확도를 저하시켜, 사용자 만족도를 감소시킨다. 검색의 정확도 및 사용자 만족도를 높일 수 있는, 효과적인 동영상 검색을 지원하기 위해서는 이미지, 색, 모양, 사운드 음량 및 음질, 애니메이션, 또는 비디오 내의 주요 장면과 해당 장면에서의 객체의 움직임 등 동영상 데이터를 대표할 수 있는 내용들을 기반으로 검색할 수 있어야 한다. 이러한 동영상 데이터의 다수의 특징들은 고차원의 벡터 데이터로 표현될 수 있으며, 동영상 데이터의 내용-기반 검색은 해당 동영상으로부터 추출한 고차원의 특징 벡터 데이터를 이용하여 유사한 동영상을 탐색하여 수행한다.

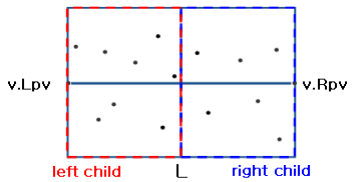
기존에 제안된 고차원 벡터 데이터 색인 기법으로는 X-Tree[1], TV-Tree[2] M-Tree[3], Spill-Tree[4], Hybrid Spill-Tree[4] 등이 있다. 그런데 Hybrid Spill-Tree를 제외한 기존의 색인기법들은 고차원 데이터를 다루는 데 비효율적이다. 따라서, 본 논문에서는 효율적인 고차원 색인 기법을 제안한다. 이를 위해 기존 Hybrid Spill-Tree를 개선한, 시그니쳐-기반 고차원 색인 기법을 제안한다. 제안하는 기법은 첫째, 고차원 데이터의 클러스터링 방법을 새로이 설계하여 성능을 개선하고, 둘째, 데이터의 축약 형태인 시그니쳐를 사용하여 검색 성능을 개선한다.

본 논문의 구성은 다음과 같다. 먼저 2장에서는 고차원 벡터 데이터 색인 기법에 대한 기존 연구를 살펴보고, 3장에서는 본 연구에서 제안한 시그니쳐-기반 고차원 색인 기법에 대하여 논한다. 4장에서는 제안하는 시그니쳐-기반 고차원 색인 기법의 성능 평가를 수행한다. 마지막으로 5장에서는 결론 및 향후 연구 방향에 대하여 서술한다.

2. 관련 연구

2.1 기존 고차원 색인 기법

첫째, M-Tree[3]는 고차원 벡터 데이터를 기반으로 효율적인 최근접점 탐색을 지원하기 위해 제안된 트리 구조이다. M-Tree의 노드 분할 알고리즘은 다음과 같다. 첫째, 노드 내의 모든 데이터들 중, 가장 멀리 떨어진 두 점 Lpv와 Rpv를 찾는다. 둘째, 두 점 Lpv와 Rpv를 잇는 선분에 노드 내의 모든 데이터를 투영시킨다. 투영된 위치가 선분의 이등분점에 가장 가까운 점 A (median point)를 선택하여, 노드 분할 기준선 L을 생성한다. 마지막으로, 기준선 L을 중심으로 왼쪽 자식 (left child)과 오른쪽 자식(right child)으로 노드를 분할한다. 그림 1은 M-Tree의 노드 분할을 나타낸다.

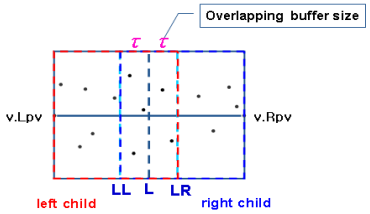


(그림 1) M-Tree의 노드 분할

한편 M-Tree는 MT-DFS(M-Tree Depth First Search)라고 불리는 깊이 우선탐색 알고리즘을 사용하여 정확검색(exact match)을 지원한다. k-최근접점 탐색(k-Nearest Neighbor Search)은 다음과 같다. 첫째, 질의점(q)이 노드(V)에 주어지면, 기준선(L)의 왼쪽에 있는지, 오른쪽에 있는지를 확인한다. 만약 질의가 기준선의 왼쪽에 있다면, 노드의 왼쪽 자식을 먼저 탐색하고, 오른쪽에 있다면 오른쪽 자식을 먼저 탐색한다. 둘째, 방문한 자식 노드에서 후보 셋의 k번째 점 x를 찾고, 이 때 질의와 x와의 거리를 최대 탐색 거리 r 로 설정한다. 마지막으로, 탐색하지 않은 나머지 자식 노드 C'에서, 질의로부터 최대 탐색 거리 이내에 C'의 맴버가 있는지 확인한다, 만약 존재한다면 C'를 탐

색한다. 만약 최대 탐색 거리 이내에 존재하는 C'의 맴버가 없다면, 노드 C'와 C'의 모든 자식 노드는 탐색하지 않는다.

둘째, Spill-Tree[4]는 M-Tree의 단점인 역추적(back-tracking)으로 인한 탐색 시간을 줄이기 위해 제안되었다. Spill-Tree는 데이터 공유 영역("overlapping buffer")을 이용하여 M-Tree를 확장한 색인 기법이다. 그림 2는 Spill-Tree의 노드 분할을 나타낸다.



(그림 2) Spill-Tree에서의 노드 분할

Spill-Tree의 노드 분할 알고리즘은 M-Tree와 유사하지만, 두 가지 차이점이 존재한다. 첫째, Spill-Tree는 Lpv와 Rpv를 잇는 선분의 수직 이등분선을 계산하여, 노드 분할 기준선 L을 생성한다. 둘째, Spill-Tree는 데이터 공유 영역(overlapping buffer)을 지닌다. 데이터 공유 영역은 기준선 L로부터 거리 t 이내의 영역(LL~LR)을 의미하며, 이 영역에 있는 모든 점들은 양쪽 자식 노드(LC, RC)에 공유된다. 식 1 과 2는 LC와 RC가 저장하고 있는 객체들의 집합을 표현한다.

$$N(LC) = \{N(LC) \cup \text{overlapping buffer}\} \quad (1)$$
$$N(RC) = \{N(RC) \cup \text{overlapping buffer}\} \quad (2)$$

한편, Spill-Tree는 Defeatist Search라 불리는 근사 최근접점 탐색 (Approximate NN Search)를 사용한다. Defeatist Search 는 질의점(q)이 놓인 자식 노드만을 검색하여, 가장 처음 만난 단말 노드의 최근접점을 결과 값으로 반환한다. Spill-Tree는 Defeatist Search를 통해 시간 측면에서 검색의 성능을 향상시킨 반면, 검색 정확도는 최악의 경우 50% 밖에 되지 않는 단점을 지니고 있다.

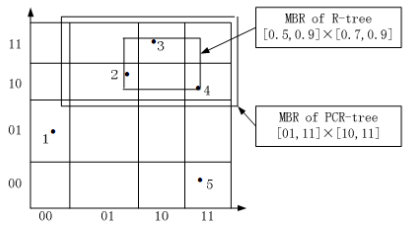
셋째, Hybrid Spill-Tree[4]는 Spill-Tree에서 근사 k-최근접점 탐색 시검색 정확도를 향상시키기 위해 제안되었다. Spill-Tree에서 데이터 공유 영역의 크기를 결정하는 변수 t는 트리의 깊이, 검색 속도 및 정확성에 큰 영향을 준다. 식 (3)과 (4)는 t의 크기에 의한 트리의 깊이 변화를 나타낸다.

$$t = 0 \qquad \log N \quad (3)$$
$$t > |Lpv - Rpv| / 2 \qquad \infty \quad (4)$$

t가 증가할수록 트리의 깊이가 깊어지기 때문에 검색 속도는 저하된다. 그러나 데이터 공유 영역의 크기가 증가하기 때문에 검색의 정확도는 향상된다. 이러한 trade-off 문제를 해결하기 위해, Hybrid Spill-Tree는 t를 상수값으로 설정한다. 기본적으로 Hybrid Spill-Tree는 Spill-Tree의 노드 분할 알고리즘과 Defeatist Search 알고리즘을 사용한다. 단, 노드 분할시 자식 노드 데이터 량/부모 노드 데이터 량이 일정 비율(threshold) 이상일 경우, 비공유(non-overlapping) 노드라 불리는 M-Tree의 방식으로 노드를 분할하고, 해당 노드 검색할 때 MT-DFS 방식으로 검색한다. 즉, 공유(overlapping) 노드에서는 역추적을 수행하지 않는 Defeatist Search 방식을, 비공유 노드에서는 역추적을 수행하는 MT-DFS 방식을 사용하여 k-최근접점 탐색을 수행한다. 한편, 최근 유명 인터넷 검색 사이트인 구글에서 Hybrid Spill-Tree를 기반으로 이미지 데이터 등 대용량의 고차원 벡터 데이터를 위한 분산 저장 및 검색 시스템을 구축하였다[5]. 또한 구축된 고차원 데이터 분산 저장 및 검색 시스템을 통해 구글 이미지 검색과 같은 인터넷 서비스를 지원하고 있다. 이는 Hybrid Spill-Tree가 고차원 데이터 색인에 효율적인 방법임을 나타낸다.

넷째, PCR-Tree는 고차원 벡터 데이터 색인시 차원의 증가에 따른 성능저하를 완화하기 위해 제안되었다[6]. PCR-Tree에서는 고차원 벡터 데이

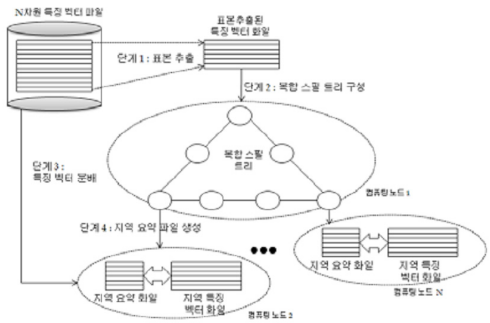
터를 KL 변환 기법(Karhunen-Loeve transformation)을 사용하여 저차원 벡터 데이터로 변환하고, 변환된 저차원 벡터 데이터로부터 근사값을 생성한다. 생성된 저차원 벡터 데이터의 근사값을 R-Tree의 노드 분할 알고리즘을 사용하여 색인한다. PCR-Tree에서는 벡터 데이터의 근사값을 사용하여 노드의 MBR(minimum boundary rectangle)을 생성하기 때문에, PCR-Tree의 노드 MBR은 R-Tree의 노드 MBR에 비해 크기가 증가한다. (그림 3)은 PCR-Tree와 R-Tree의 MBR 의 차이를 나타낸다.



(그림 3) R-Tree와 PCR-Tree에서의 MBR

MBR의 크기가 증가하기 때문에, 다른 노드와 MBR의 겹치는 영역이 증가한다. 이는 k-최근접점 탐색 시 방문해야 하는 노드의 수를 증가시키고, 따라서 전체적인 탐색 성능을 저하시킨다. 이러한 문제를 해결하기 위한 k-최근접점 탐색 알고리즘은 다음과 같다. 첫째, 사용자 질의를 KL 변환 기법 및 근사화를 통해 저차원 벡터 데이터의 근사값으로 변환한다. 둘째, 질의의 근사값을 기준으로 모든 MBR까지의 최소 경계 거리(lower bound distance: l)를 측정한다. 측정된 최소 경계 거리 l을 기준으로 MBR을 오름차순으로 정렬하고, l이 작은 노드로부터 순서대로 방문하여 k개의 후보셋을 생성한다. 만약 질의와 k번째 후보 데이터 사이의 거리가 다음 방문할 노드와 질의 사이의 최소 경계 거리보다 작을 경우, 더 이상의 노드를 방문하지 않고 탐색된 k개의 후보셋을 반환하여 검색을 종료한다. PCR-Tree는 고차원 벡터 데이터를 저차원 벡터 데이터의 근사값으로 변환하는 특성을 지닌다. 이러한 특성 때문에, k-최근접점 탐색시 데이터 사이의 거리 계산 오버헤드를 감소시키는 장점이 존재하는 반면, 고차원 벡터 데이터를 저차원 벡터 데이터로 변환하기 위한 추가적인 처리 시간이 요구되는 단점이 존재한다. 고차원 벡터 데이터의 증가함에 따라 이러한 벡터 변환을 위한 전체 처리 시간 오버헤드가 증가하기 때문에, PCR-Tree는 대용량의 고차원 벡터 데이터를 색인하기에는 부적합하다.

마지막으로, 한국전자통신연구원에서는 저비용 대규모 글로벌 인터넷 서비스 솔루션 GLORY-DB[7]를 개발 중이며, GLORY-DB에서 대용량 동영상 데이터의 내용-기반 검색을 지원하기 위해 확장성을 지원하는 고차원 색인 기법인 Hybrid Spill-Tree with Local Signature Files(이하 HSP-LS)를 제안하였다[8]. 제안한 HSP-LS는 Hybrid Spill-Tree를 VA-file[9]과 결합하여 확장한 기법으로써, 대규모의 클러스터 시스템을 기반으로 설계하였다. 전체 데이터를 분산 저장하기 위한 과정은 다음과 같다. 첫째, 마스터 서버에 Hybrid Spill-Tree를 구성한다. 구성된 Hybrid Spill-Tree의 단말 노드는 클러스터 내의 다른 컴퓨팅 서버(단말 컴퓨팅 서버)와 연결된다. 둘째, 마스터 서버에 구성된 Hybrid Spill-Tree를 이용하여, 전체 데이터를 분산 저장한다. 셋째, 각 단말 컴퓨팅 서버에서는 저장된 데이터로부터 벡터의 근사값인 시그니쳐를 생성하여 메모리에 저장한다. (그림 4)는 HSP-LS의 전체적인 구조를 나타낸다.

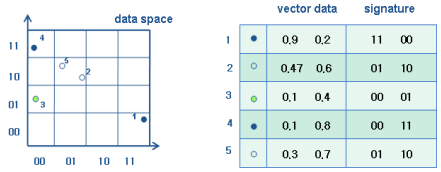


(그림 4) HSP-LS의 전체 구조

HSP-LS의 k-최근접점 탐색 알고리즘은 다음과 같다. 첫째, 마스터 서버에서는 Hybrid Spill-Tree의 노드 탐색 알고리즘을 사용하여, 질의 처리를 위해 방문할 단말 컴퓨팅 서버를 선택한다. 둘째, 선택된 단말 컴퓨팅 서버는 메모리 내의 시그니쳐를 사용하여, k개의 후보 집합을 생성하여 반환한다. 마지막으로, 모든 후보 집합을 병합하여 k개의 결과를 사용자에게 반환한다. HSP-LS 기법은 대규모 클러스터 환경을 기반으로 설계되었으며, 다수의 단말 노드에의 병렬 접근 방법, 다수의 컴퓨팅 노드로부터 전송된 후보 셋 병합 방법, 단말 컴퓨팅 노드에서의 시그니쳐 및 벡터 데이터의 메모리 관리 방법 등을 포함한 분산 환경에 적합한 색인 구조이다. 그러나 분산 환경이 아닌 일반적인 응용 환경에서는, 첫째, 고차원 벡터 데이터가 대용량인 경우 메모리에 상주시키는 것은 어려우며, 둘째, 다수의 단말컴퓨팅 서버를 순차적으로 방문할 경우, 전체 질의 처리 시간이 크게 증가하는 단점이 존재한다.

2.2 시그니쳐 기법

시그니쳐(Signature)란 벡터 데이터의 구간을 여러 구역으로 나누고, 나누어진 각 영역에 비트(bit)를 할당하여 생성된 근사값이다[9, 10, 11]. 그림 5는 2차원 공간에서 시그니쳐가 생성된 모습을 보여준다.



(그림 5) Vector approximation

저차원 데이터의 경우, 그림 3의 점 2와 5 같이 실제 값은 (0.47, 0.6), (0.3, 0.7)으로 서로 다르지만, 같은 시그니쳐 값 '0110'을 지니게 되어 데이터가 서로 동일하게 취급된다. 그러나 차원이 증가할수록 데이터의 근사값으로 쓰이는 셀의 수가 급격하게 증가하여, 실제 하나의 셀에 두 개 이상의 데이터가 같이 존재하는 경우는 매우 드물게 된다. 표 1은 차원수가 D인 두 개의 벡터 데이터를 n 비트 시그니쳐로 생성할 경우 두 벡터 데이터가 같은 시그니쳐 값을 가질 확률을 나타낸다. 따라서 고차원 데이터는 시그니쳐를 사용한 근사값이 하나의 벡터 데이터를 나타내기 때문에 시그니쳐에 의한 근사 탐색이 가능하다.

(표 1) 두 벡터 데이터의 시그니쳐 값이 같을 확률 (D=차원)

데이터 차원수	시그니쳐 비트수	차원 당, 시그니쳐 값이 동일할 확률	두 벡터 데이터의 시그니쳐 값이 동일할 확률
D	n	$1/2^n$	$(1/2^n)D$

시그니쳐 기법을 이용할 경우 두 가지의 장점이 존재한다. 첫째, 저장되는 데이터의 크기가 줄어든다. double형 벡터의 집합으로 이루어진 벡터 데이터를 벡터당 4bit의 시그니쳐를 사용하여 표현할 경우, 데이터량이 1/16로 감소한다. 둘째, 데이터 간의 거리 계산 시 비트 연산(bit poeration)을 수행하기 때문에, 계산 속도가 향상된다. 예를 들어 거리 계산 수행시 벡터 데이터는 double형을 사용하지만, 4비트 시그니쳐는 비트 연산을 이용하여 계산한다. 따라서 데이터 사이의 거리 계산에 의한 오버헤드가 감소한다.

하지만, 시그니쳐는 벡터 데이터의 근사값이므로, 시그니쳐를 통해 얻은 검색 결과의 정확도가 벡터 데이터를 이용할 때보다 감소한다.

3. 시그니쳐-기반 고차원 색인 기법

3.1 설계시 고려사항

고차원 데이터 색인 기법을 설계하기 위해서 세 가지 사항을 고려한다. 첫째, 색인 구조의 저장

공간오버헤드(Overhead)를 감소시켜야 한다. 이를 위해 시그니쳐를 사용하여 데이터를 압축 및 근사화를 수행하는 것이 타당하다. 둘째, 빠른 검색 속도를 보장해야 한다. 이를 위해 기존 Google 의 Hybrid Spill-Tree를 시그니쳐 기반 방법으로 확장한다. 아울러 검색 속도 및 정확도에 가장 큰 영향을 미치는 효과적인 클러스터링 방법이 새로이 설계한다. 셋째, 고차원 데이터에 대한 k-최근접 점 질의 시 벡터 데이터간의 거리 연산 오버헤드를 감소시켜야 한다. 이를 위해, 시그니쳐를 이용한 거리 계산으로 그 부담을 감소시킨다.

따라서 본 논문에서는 고차원 벡터 데이터의 근사값인 시그니쳐를 이용한 고차원 색인 기법인, 시그니쳐-기반 Hybrid Spill-Tree(Signature-based Hybrid Spill-Tree)를 제안한다. 이는 고차원 데이터에 따른 저장 및 검색의 비용을 줄이고, 대용량의 동영상 데이터를 위해, 최소한의 정확도를 만족하면서 빠른 검색을 지원한다. 즉, 시그니쳐를 이용할 경우 첫째, 노드 내에 저장되는 데이터의 크기가 줄어들어 전체적인 트리의 높이가 감소되고, 따라서 트리의 검색 속도가 향상된다. 둘째, 데이터 간의 거리 계산이 빨라져, 전체적인 거리 계산 속도가 향상된다. 따라서 시그니쳐를 사용할 경우, 색인을 위한 트리 높이가 감소하고 거리 계산 속도가 향상되어, 검색 성능이 향상된다.

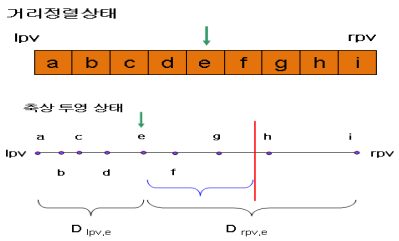
한편, 본 논문에서 제안하는 시그니쳐-기반 Hybrid Spill-Tree는 크게 두 단계로 나누어진다. 첫 번째 단계는 노드 분할로 대표되는 클러스터링 단계이며, 두 번째 단계는 클러스터링이 수행된 데이터의 저장 단계이다. 제안한 시그니쳐-기반 Hybrid Spill-Tree (이하 SigHSP-Tree라 명명)를 구성하기 위해 데이터의 저장은 시그니쳐를 사용한다. 한편, 데이터의 클러스터링을 위해 시그니쳐를 사용할지, 아니면 벡터 데이터를 사용할지에 따라 두 가지 방법을 제안한다. 첫째, 클러스터링 및 데이터 저장을 모두 시그니쳐로 수행하는, 시그니쳐 클러스터링을 이용한 SigHSP-Tree 방법을 제안한다. 둘째, 클러스터링은 원 벡터 데이터로 수행하고, 데이터 저장은 시그니쳐로 하는 벡터 데이터 클러스터링을 이용한 SigHSP-Tree 방법을 제안한다.

3.2 시그니쳐-기반 고차원 색인 기법에서의 클러스터링 방법

3.2.1 분할 기준점 선정 및 보정

SigHSP-Tree에서의 클러스터링을 위해 가장 멀리 떨어진 두 개의 데이터를 분할 기준점으로 선정한다. 분할 기준점 선택을 위해서는 모든 점 사이의 거리 계산이 필요하지만, 데이터 량의 증가에 따라 계산 오버헤드가 급격히 증가한다. 따라서 거리 계산 오버헤드 감소를 위한, 분할 기준점 선정 방법을 제시한다. 먼저 노드 내에 있는 모든 점들에 대해 원점 O로부터의 거리를 계산하여 가장 먼 점을 선택한다. 선택되어진 원점 O로부터 가장 먼 점을 Right Pivot이라 한다. 다음 선택된 점 Right Pivot에서 노드 내의 모든 점까지의 거리를 계산하여 가장 멀리 떨어진 점을 선택하고, 이 점을 Left Pivot이라 하여, 두 분할 기준점을 선정한다.

아울러 데이터의 분포를 고려한 노드 분할을 위해, 분할 기준점 보정 방법을 제안한다. 분할 기준점 보정은 다음과 같은 과정으로 진행된다. 먼저 분할 기준점 선정 방법으로 두 후보 분할 기준점, rvp와 lpv를 선정한다. rpv에서의 거리를 오름차순으로 정렬하고, N/2(N : 노드 내 데이터 수)의 지점에 있는 점 e를 찾아, e가 투영 축 상에서의 중점이 되도록 한쪽의 Pivot을 이동시킨다. 이를 위해 Distance(lpv, Data[N/2]), Distance(rpv, Data[N/2])를 계산하여, 만약 Distance(lpv, Data[N/2]) < Distance(rpv, Data[N/2]) 이면, rpv를 lpv쪽으로 움직이고, 반대의 경우에는 lpv를 rpv로 이동시킨다. 그림 6은 분할 기준점 보정의 예시를 나타낸다.

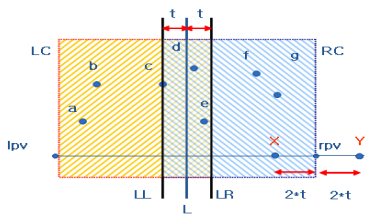


(그림 6) 분할 기준점 보정

3.2.2 노드 분할

분할 기준점 선정 후 노드 분할 기준선, 즉, 두 분할 기준점 Right Pivot(이하 rpv)과 Left Pivot(이하 lpv)를 잇는 선분의 수직 이등분선을 구해야 한다. 그러나 데이터의 차원이 증가할수록, 수직 이등분선을 구하기 위한 오버헤드가 증가한다. 이러한 오버헤드를 감소시키기 위해 다음과 같은 방법을 사용한다.

유클리디언 공간에서는 두 점 B, C와 직선 BC 상에 위치하지 않은 임의의 점 A의 세 점을 통해 한 평면상의 삼각형 ABC가 생성된다. 이 때 A가 직선 BC의 수직 이등분선 상에 존재할 경우, A와 B 사이의 거리와 A와 C 사이의 거리가 같다. 또한 A가 직선 BC의 중점에 있으면, A는 직선 BC의 수직 이등분선 상에 존재한다. 따라서 점 B와 점 C로부터 거리가 같다면, 점 A는 직선 BC의 수직 이등분선 상에 존재한다. 이러한 수직 이등분선의 성질을 이용하여 노드를 분할한다. 그림 7은 데이터 공유 영역을 가지는 Spill-Tree 방식의 노드 분할의 예시를 나타내며, 여기서 점 X와 Y는 각각 LL과 LR을 구하기 위한 기준점이다.



(그림 7) Spill-Tree 노드 분할

3.2.3 표본(Sampling)에 의한 분할 기준점 선정

제안한 분할 기준점 보정 방법은 투영축 상에서의 데이터 분포만을 고려하기 때문에, 실제 분할되는 데이터에는 변화가 없을 수 있는 문제점이 있다. 이를 해결하기 위해, 표본(Sampling)에 의한 분할 기준점 선정 알고리즘을 제안한다. 표본에 의한 분할 기준점 선정 알고리즘의 기본 원리는 다음과 같다.

> "모집단으로부터 추출한 표본 집단의 성질은 모집단성질과 비슷하다."

즉, 노드를 두 개의 자식 노드로 분할하는 것은, 부모 노드 내의 데이터를 임의 추출했을 때 한쪽 자식에 속할 확률을 결정하는 것과 동일하다. 이상적인 노드 분할 기준점은 노드 내의 데이터를 임의 추출 시 한쪽 자식에 속할 확률을 50%로 결정한다. 한편, 분할 기준점 선정을 위해 모집단 내의 모든 데이터를 이용하는 것은 많은 계산 비용이 요구된다. 따라서 모집단의 데이터가 아닌, 표본집단을 추출하여 분할 기준점을 선정한다. 이는 일정 수 이상 추출된 표본집단을 통하여 결정된 분할 기준점이 본래 데이터 셋을 표본 셋과 유사하게 분할하기 때문이다. 그림 8은 표본에 의한 분할 기준점의 신뢰도, 오차율, 표본추출 수 등의 관계를 나타낸다.

$$n \geq p*(1-p)*(Z/d)^2$$

(그림 8) 표본추출 수(n), 발생확률(p), 분할 기준점 신뢰도(Z) 및 오차율(d)의 관계

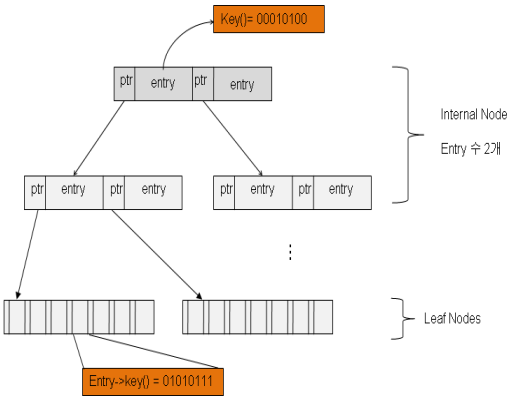
발생확률 p는 한쪽 자식 노드 내의 데이터 비율을 의미하며, 이상적인 경우는 0.5 이다. 또한 신뢰도 Z는 상수로 주어지며, 95% 구간의 신뢰도를 얻기 위한 값은 1.96이 된다. 만약 95% 신뢰도, 오차율(d) 10%, 발생확률 0.5를 원할 경우 최소 표본수(n)는 약 100개이다. 즉, 표본 100개에서 얻어진 분할 기준점은, 95% 신뢰구간에서, 약 10% 오차율을 가지고, 모집단을 그와 유사한 비율로 분

할할 수 있는 분할 기준점이 된다. 예를 들어 표본 분할 기준점을 통해 자식 노드 내에 부모 노드 데이터의 45%가 포함될 경우, 95% 확률로 모집단을 40.5% ~ 49.5% 의 사이 값으로 분할 가능하다.

3.3 시그니쳐 클러스터링을 이용한 SigHSP-Tree

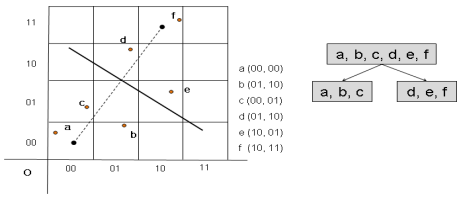
3.3.1 전체 구조

SigHSP-Tree의 구성은 다음와 같다. 중간 노드(Internal node)는 두 개의 엔트리로 구성되며, 각 엔트리에는 시그니쳐가 저장된다. 질의가 주어져 트리를 탐색할 때 중간 노드의 엔트리에 저장된 키이(key)를 살펴봄으로서, 해당 질의가 어느 노드를 탐색해야 하는지 알 수 있다. 또한, 단말 노드(Leaf node)에는 페이지 크기만큼의 엔트리들이 저장되며, 각 엔트리에 저장된 데이터는 시그니쳐이다. 그림 9는 이러한 SisHSP-Tree의 전체 구조를 나타낸다.



(그림 9) SigHSP-Tree의 구조

한편, 시그니쳐 클러스터링을 이용할 경우, 모든 데이터 사이의 거리 계산이 시그니쳐를 통하여 이루어진다. 그림 10은 시그니쳐 클러스터링을 통한 노드 분할을 나타낸다.



(그림 10) 시그니쳐 클러스터링

3.3.2 삽입 알고리즘

제안한 시그니쳐 클러스터링을 이용한 SigHSP-Tree의 삽입 알고리즘은 대량의 데이터를 다룰 수 있도록 설계되었으며, 알고리즘은 다음과 같다. 첫째, 입력된 시그니쳐 데이터가 단말 노드에 저장 가능한지 확인한다. 만약 저장 가능하면, 단말 노드에 시그니쳐 데이터를 저장하고 종료한다. 반면, 시그니쳐 데이터의 크기가 클 경우 중간 노드에 삽입한다. 둘째, 중간 노드에 시그니쳐 데이터 삽입시, 거리 계산 및 표본 추출을 시그니쳐를 이용하여 수행한다. 추출된 표본을 통해 중간 노드 분할 기준점을 선정하고, 각 자식 노드에 저장할 시그니쳐 데이터 셋(data set)을 생성한다. 마지막으로, 삽입 함수를 재귀호출 하여 자식 노드 트리를 생성하고, 이를 부모 노드에 통합하여 완료된다. 그림 11은 시그니쳐 클러스터링을 이용한 SigHSP-Tree의 삽입 알고리즘을 나타낸다.

```
void Insert(data, realdata, n,  tao,  Isparent, *name){
//sigEntry **data : 시그니쳐 데이터 셋
//int n : 데이터 수,          double tao : 공유 영역
//int Isparent : 자식 노드인지 부모 노드인지 결정
//char name : 데이터를 삽입할 트리의 이름
  if(Sizeof(data) > Sizeof(LeafNode)) {
    sigPivot sRpv, sLpv;  //노드 분할 기준점 - 시그니쳐
    Pivot Rpv, Lpv;  //노드 분할 기준점 - 시그니쳐
    sigEntry **sigRc, **sigLc;//자식 노드 시그니쳐 셋

    sigEntry **Parent;  //자식 노드 트리 연결자
    //표본을 이용한 pivot 선정, 시그니쳐를 이용한 계산
    Sampling(data, Rpv, Lpv, sRpv, sLpv);
    //데이터 분할
    SplitDataSigHSPII(data, Rc, Lc, sRc, sLc);
    //재귀호출로 두 개의 Child 노드에 대한 트리 구축
```

```
    Insert(sLc, Lc, Numberofdata(Lc), tao, Isparent, *Lname);
    Insert(sRc, Rc, Numberofdata(Rc), tao, Isparent, *Rname);
    //자식 노드 트리 연결자 설정
    SetParents(Parent, sRpv, sLpv);
    //Child 노드 트리들을 통합하기 위한 Super-Tree 생성
    Insert(Parent, 2, tao, TRUE, *SuperTree);
    Append(*SuperTree, *Lname, *Rname); //트리 통합
  }
  else {
    Findnode(data, path); //data를 삽입 노드 선택
    InsertNode(data, path); //path에 있는 노드에 data 저장
  }
}
```

(그림 11) 시그니쳐 클러스터링을 이용한 SigHSP-Tree 삽입 알고리즘

3.3.3 검색 알고리즘

시그니쳐 클러스터링을 이용한 SigHSP-Tree의 검색 알고리즘은 다음과 같다. 먼저 질의 입력시, 트리의 루트로부터 탐색을 시작한다. 탐색 노드의 자식 노드가 단말 노드일 경우에는 질의 결과에 자식 노드 데이터를 저장한다. 이 때, 검색된 데이터의 거리가 후보 결과셋의 최대 거리보다 더 짧은 경우에만 결과에 저장한다. 만약 자식 노드가 단말이 아니라면, 다음 탐색할 노드를 탐색 노드 큐에 저장한다. 탐색 노드 큐는 질의와 노드와의 거리를 기준으로 노드가 들어올 때 오름차순으로 정렬한다. 이 후 큐에서 다음 탐색 노드를 얻고, 해당 노드와 질의와의 거리를 결과의 최대거리와 비교하여 탐색해야할지 결정한다. 만약 탐색하지 않아도 된다면 탐색 노드 큐를 전부 비우고, 검색을 종료한다. 한편, 노드에 원 벡터 데이터 대신 데이터의 시그니쳐가 저장되기 때문에, 시그니쳐 사이의 거리를 계산한다. 그림 12는 시그니쳐 클러스터링을 이용한 SigHSP-Tree의 검색 알고리즘을 나타낸다.

```
Entry** SigSearch(query, result_num){
//const Query& query : 주어진 질의
//int *result_num    : 검색된 결과 수
 Entry** result;              //검색 결과를 저장
 orderedQueue *SearchNext;//탐색할 노드 저장, 자동 정렬
 node = ReadNode(Root);     //트리의 루트부터 탐색 시작
 while(node!=NULL) {
   for(int i=0; i<node->NumEntries(); i++) {
   //현재 노드의 자식의 영역 내에 질의가 있는지 확인
     bool check = SigCheckEntry(node->entry[i]);
     if(check && node->entry[i] = IsLeaf) { //단말노드
       //자식 노드 데이터를 확인 후 결과 셋으로 저장
        GetSigResult(node->entry[i], result);
     }
     else if(check && node->entry[i] != IsLeaf) {
       //중간노드이므로, 탐색 예정 노드로 저장
       SetNextNode(node->entry[i]);
     }
   }
   node = SearchNext.GetNext();
   //탐색 예정 노드가 질의 영역 내에 있는지 확인
   if(result[max] < node->grade()) { //질의 영역 밖
     SearchNext.Clean; //탐색 예정 노드 셋 삭제
     node = NULL;       //탐색 중지
   }
 }
 Set(result_num, result); //검색되어진 결과 수 저장
 return result;
}
```

(그림 12) 시그니쳐 클러스터링을 이용한 SigHSP-Tree 검색 알고리즘

3.3.4 알고리즘 복잡도

시그니쳐 클러스터링을 이용한 SigHSP-Tree의 삽입 및 검색 알고리즘의 복잡도는 다음과 같다. 첫째, 삽입 알고리즘의 복잡도는 입력 데이터량(N), 단말노드 내에 저장 가능한 시그니쳐 데이터량(S), 공유 데이터 비율(t)에 의해 결정된다. 이는 데이터 삽입의 재귀 호출 횟수가 단말 노드 수에 비례하기 때문이며, 식 (5)와 같이 나타난다.

$$O(N*(1+t)/S) \quad (5)$$

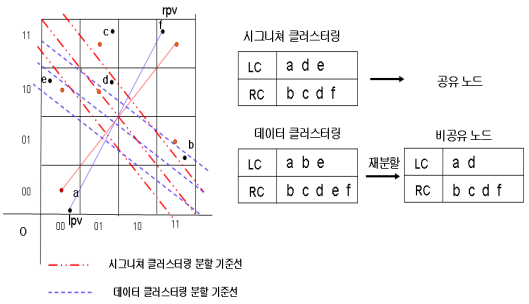
둘째, 검색 알고리즘의 복잡도는 트리의 깊이에 비례한다. 이때 트리의 깊이는 단말 노드 수에 지수적으로 비례하기 때문에 식 (6)과 같이 나타낼 수 있다.

$$O(\log(N*(1+t)/S)) \quad (6)$$

3.4 벡터 데이터 클러스터링을 이용한 SigHSP-Tree

3.4.1 문제점 분석

제안한 시그니쳐 클러스터링을 이용한 SigHSP-Tree는 검색 정확도가 저하된다. 이는 같은 노드에 삽입되어야 할 데이터들이 시그니쳐 클러스터링을 수행한 결과 다른 노드에 삽입되었기 때문이다. 이를 개선하기 위해 벡터 데이터 클러스터링 방법을 제안한다. 즉, 데이터 삽입 시 벡터 데이터를 이용하여 거리를 계산하며, 따라서 실제 데이터 분할과 동일하게 트리를 구성한다. 그림 13은 벡터 데이터 클러스터링과 시그니쳐 클러스터링의 차이점을 나타낸다.



(그림 13) 시그니쳐 클러스터링 및 벡터 데이터 클러스터링에서의 노드 분할의 차이점

한편, 검색의 경우 시그니쳐 클러스터링을 이용한 SigHSP-Tree의 검색 알고리즘이 동일하다. 이는 노드에 저장되는 값과 거리 계산 수행시 사용되는 값이 벡터의 시그니쳐로 동일하기 때문이다. 또한 벡터 데이터 클러스터링을 이용한 SigHSP-Tree의 삽입 및 검색 알고리즘은, 벡터 데이터를 이용한 거리 계산을 제외하면 시그니쳐 클러스터링을 이용한 SigHSP-Tree와 동일하며, 따라서 알고리즘의 복잡도는 동일하다.

3.4.2 삽입 알고리즘

벡터 데이터 클러스터링을 위해 첫째, 시그니쳐 데이터 및 벡터 데이터를 입력받는다. 벡터 데이터는 노드에 저장되지 않고, 단지 노드 분할을 위해 사용된다. 입력된 시그니쳐 데이터가 단말 노드에 저장 가능한지 확인한다. 만약 저장 가능하면, 단말 노드에 시그니쳐 데이터를 저장하고 종료한다. 반면, 시그니쳐 데이터의 크기가 클 경우, 중간 노드에 삽입한다. 중간 노드 삽입 시 거리 계산 및 표본 추출은 벡터 데이터를 이용하여 수행한다. 둘째, 추출된 표본을 이용하여 중간 노드 분할 기준점 계산하고, 벡터 데이터 셋을 분할한다. 분할된 벡터 데이터 셋과 매칭되는 시그니쳐 데이터 셋을 생성한 후, 자식 노드에 삽입한다. 마지막으로 삽입 함수를 재귀호출하여 자식 노드 트리를 생성하고, 이를 부모 노드에 통합하여 삽입을 완료한다. 그림 14는 벡터 데이터 클러스터링을 이용한 SigHSP-Tree의 삽입 알고리즘을 나타낸다.

```
void Insert(data, realdata, n,  tao,  Isparent, *name) {
//sigEntry **data : 시그니쳐 데이터 셋
//realEntry **realdata : 벡터 데이터 셋
//int n : 데이터 수,          double tao : 공유 영역
//int Isparent : 자식노드 트리인지 부모 노드 트리인지 결정
//char name : 데이터를 삽입할 트리의 이름
   if(Sizeof(data) > Sizeof(LeafNode))  {
      sigPivot sRpv, sLpv;  //노드 분할 기준점 - 시그니쳐
      Pivot Rpv, Lpv;  //노드 분할 기준점 - 벡터 데이터
      sigEntry **sigRc, **sigLc;//자식 노드 시그니쳐 셋
      Entry **Rc, **Lc;   //자식 노드 데이터 셋
      sigEntry **Parent;  //자식 노드 트리 연결자
      //표본을 이용한 분할 기준점 선정, realdata를 이용한 계산
      Sampling(data, realdata, Rpv, Lpv, sRpv, sLpv);
      //데이터 분할
      SplitDataSigHSPII(data,  realdata,  Rc,  Lc,  sRc, sLc);
      //재귀호출로 두 개의 Child 노드에 대한 트리 구축
      Insert(sLc,  Lc,  Numberofdata(Lc),  tao,  Isparent, *Lname);
      Insert(sRc,  Rc,  Numberofdata(Rc),  tao,  Isparent, *Rname);
      SetParents(Parent, sRpv, sLpv);//자식 노드 트리 연결자 설정
```

```
        //Child 노드 트리들을 통합하기 위한 Super-Tree 생성
        Insert(Parent, 2, tao, TRUE, *SuperTree);
        Append(*SuperTree, *Lname, *Rname);//트리 통합
    }
    else {
        Findnode(data, path);     //data를 삽입하기 위한 노드선정
        InsertNode(data, path);  //path에 있는 노드에 data 저장
    }
}
```

(그림 14) 벡터 데이터 클러스터링을 이용한 SigHSP-Tree 삽입 알고리즘

4. 성능 평가

4.1 실험 환경

본 장에서는 제안하는 시그니쳐-기반 Hybrid Spill-Tree 기법(SigHSP-Tree)을 구현하여 성능 평가를 수행한다. 구현된 SigHSP-Tree는 클러스터링의 방법에 따라, 시그니쳐 클러스터링를 이용한 SigHSP-Tree와 벡터 데이터 클러스터링를 이용한 SigHSP-Tree로 나눌 수 있다(이하 각각의 기법을 성능평가 상에서 SigHSP-I, SigHSP-II라 표기한다). 고차원 데이터 인덱스 기법의 성능 평가의 실험 환경은 표 2과 같다.

(표 2) 실험 환경

항목	성능
CPU	Intel Xeon 3.0Ghz
Memory	2GB
OS	Windows Server 2003
Compiler	VC++ 6.0

성능평가 시나리오는 다음과 같다. 첫째, 성능 평가 항목은 k-최근접점 탐색 시간, k-최근접점 탐색 정확도, 저장 공간 오버헤드, 캐시 사용시 k-최근접점 탐색 시간을 선정하였다. 먼저 고차원 벡터 데이터 및 사용자의 수가 급증하기 때문에, 가장 중요한 요소는 질의 응답 시간이다. 이는 사용자가 빠른 시간 이내에 질의 결과를 얻고자 하기 때문이며, 따라서 질의 응답 시간에 큰 영향을 미치는 k-최근접점 탐색 시간에 대한 성능 평가를 수행한다. 성능 평가 방법은 삽입된 데이터에서 임의로 100개씩 데이터를 추출하여 생성된 질의 데이터 집합을 사용하였으며, 이러한 과정을 100회 반복하여 평균을 계산한다. 한편, 질의 결과의 정확도 또한 사용자의 만족도에 큰 영향을 미친다. 즉, 사용자의 최소 요구치를 만족하지 못할 경우, 사용자 만족도는 저하된다. 예를 들어 (질의 응답 시간, 질의 결과 정확도)가 (0.1초, 30%)인 경우보다, (0.3초, 90%)인 경우가 사용자 만족도가 높다. 따라서 본 논문에서는 정확도에 대한 사용자 최소 요구치를 90%로 설정하였으며, 제안하는 색인 기법이 이를 만족하는지 여부를 확인하기 위해 k-최근접점 탐색 정확도에 대한 성능 평가를 수행한다. 아울러 고차원 벡터 데이터 증가에 따른 색인 구조 크기 증가를 측정하기 위해, 저장 공간 오버헤드의 성능평가를 수행한다. 저장 공간 오버헤드가 작을 수록 검색 시스템 구축시 소요되는 비용이 낮아지기 때문에, 저장 공간 오버헤드가 작을 수록 성능이 향상된다. 마지막으로 대용량 데이터 검색 서비스를 지원하기 위해서는, 검색 성능 향상이 요구될 수 있다. 따라서 디스크 I/O 시간을 감소시키는 기법이 필요하며, 이를 위해 캐시(cache) 기반 기법을 사용한 k-최근접점 탐색 시간에 대해서 성능 평가를 수행한다. 둘째, 성능 평가를 위한 비교 대상은 Hybrid Spill-Tree[4]와 M-Tree[3]를 선택하였다. Hybrid Spill_Tree는 최근에 개발된 효율적인 색인 구조로써, 구글의 고차원 벡터 데이터 검색에서 사용 중이므로 비교 대상으로 선택한다. 아울러 M-Tree는 기존 고차원 데이터 클러스터링 시스템에서 널리 사용되는 기법이므로 비교 대상으로 선택한다. 한편, 기존의 연구에서 Spill-Tree를 개선한 Hybrid Spill-Tree의 성능이 Spill-Tree에 비해 우수함을 증명하였다[4]. 따라서 비교 우위에 있는 Hybrid Spill-Tree가 성능 평가 대상이기 때문에,

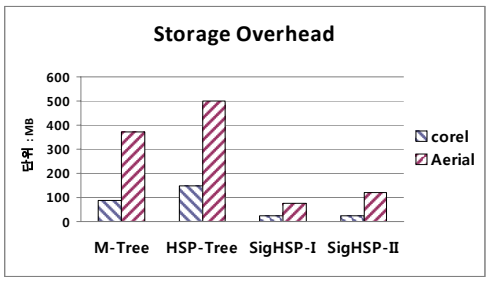
Spill-Tree는 성능 평가 대상에서 제외한다. PCR-Tree의 경우, 고차원 벡터 데이터를 저차원 벡터 데이터로 변환하기 위한 추가적인 처리 시간이 요구되는 단점이 존재한다. 이는 고차원 벡터 데이터가 급격히 증가하는 추세를 고려하면, 대용량의 고차원 벡터 데이터를 색인하기에는 부적합하다. 아울러 한국전자통신연구원에서 제안한 분산 고차원 색인 기법의 경우, 분산 환경을 기반으로 제안된 기법이므로 일반적인 동영상 검색 환경에는 적합하지 않기 때문에, 성능 평가 대상에서 제외한다. 셋째, 실험 데이터는 기존 고차원 색인 구조의 성능 평가에서 널리 사용된 Corel_UCI 및 Aerial40 리얼(real) 데이터를 선택하였다[12]. 그 이유는 많은 실험에서 사용되어 검증된 데이터이며, 아울러 리얼 데이터를 사용함으로써 성능 평가 신뢰도가 보장되기 때문이다. 또한 데이터 편중도가 낮은 Corel_UCI 및 데이터 편중도가 높은 Aerial40 데이터를 사용함으로써, 다양한 응용 환경에서의 실험 결과를 얻을 수 있다. 실험에 사용된 Corel_UCI 및 Aerial40 리얼 데이터의 특징은 표 3과 같다.

(표 3) 실험 데이터

	차원 수	데이터 수	데이터 크기
Corel_UCI	65	66,619	58.4MB
Aerial40	61	270,000	150MB

4.2 저장공간 오버헤드 성능평가

그림 15는 저장공간의 오버헤드를 나타낸다. Corel_UCI 데이터 셋의 경우, M-Tree는 89.4MB의 저장공간 오버헤드를 나타내며, 이는 고차원 데이터를 색인하기 위한 비단말 노드가 데이터의 양에 따라 증가하기 때문이다. 또한 HSP-Tree는 저장공간 오버헤드가 147MB로 가장 크다. HSP-Tree가 데이터 공유 영역을 지니고 있어, 같은 데이터가 여러 노드에 복사되어 있기 때문에 M-Tree보다 큰 저장공간 오버헤드를 나타낸다. 반면 제안하는 SigHSP-I과 SigHSP-II의 저장공간 오버헤드는 23.4MB로 가장 적었으며, 이는 트리에 저장하는 시그니쳐 데이터가 데이터 파일에 비해 크기가 매우 작고, 아울러 단말 노드 내에 저장되는 시그니쳐 데이터 수가 증가하여 트리의 깊이를 감소시키기 때문이다. Aerial40 데이터 셋의 경우, M-Tree의 저장공간 오버헤드는 371MB이며, HSP-Tree는 501MB로 가장 큰 저장공간 오버헤드를 보였다. 한편 SigHSP-I과 SigHPS-II는 각각 75MB, 122MB로 약간의 차이를 보였는데, 이는 Aerial40 데이터 셋의 데이터 밀집도가 높기 때문이다. 즉, 벡터 데이터 클러스터링의 경우, 노드 분할 시 데이터가 편중된 노드가 증가하여, 공유되는 데이터의 크기가 증가한다. 저장공간 오버헤드 성능 평가 결과, 시그니쳐를 이용한 SigHSP-I와 SigHSP-II가 기존의 M-Tree나 HSP-Tree에 비해 매우 좋은 성능을 나타낸다.

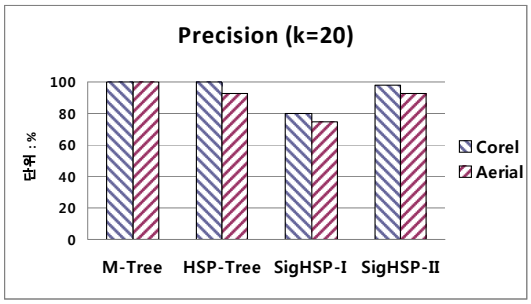


(그림 15) 저장공간 오버헤드

4.3 k-최근접점 탐색 정확도 성능평가

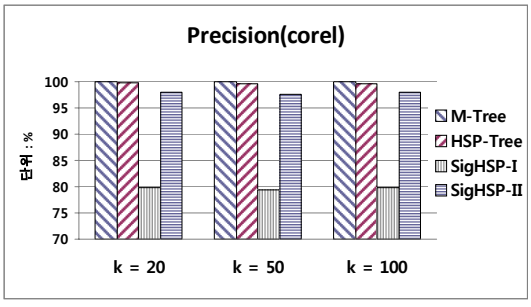
그림 16은 k = 20일 때, k-최근접점 탐색 정확도를 나타낸다. M-Tree의 경우 역추적(back-tracking)을 통해 정확한 결과를 찾기 때문에, 데이터 셋에 상관없이 100%의 정확도를 보인다. Corel_UCI 데이터 셋의 경우, HSP-Tree는 99.75%의 높은 정확도를 보였다. 하지만 SigHSP-I의 경우 79.75%의 낮은 정확도를 보였는데, 이는 시그니쳐-기반의 데이터 분할로 인해 클러스터링 성능이 감소하기 때문이다. 한편, SigHSP-II는 HSP-Tree에 비해 약간 낮은 98%의 정확도를 보이

는데, 이는 검색시 시그니쳐-기반의 거리 계산을 수행하기 때문에 오차가 발생하였기 때문이다. Aerial40 데이터 셋의 경우, HSP-Tree가 92.5%, SigHSP-II가 92.5%의 높은 정확도를 보였으며, SigHSP-I은 74.5%의 낮은 정확도를 보였다.



(그림 16) k-최근접점 탐색 정확도

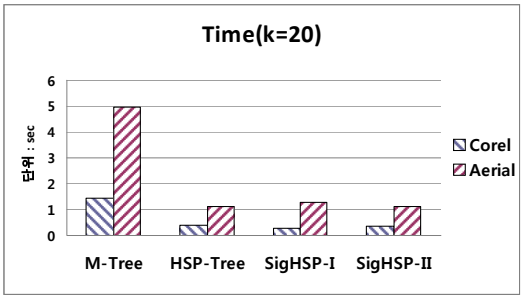
그림 17은 Corel_UCI 데이터 셋에서, k = 20, 50, 100으로 증가시킬 때, k-최근접점 탐색 정확도를 나타낸다. M-Tree의 경우 역추적을 통해 정확한 결과를 찾기 때문에, k의 증가에 상관없이 100%의 정확도를 보인다. HSP-Tree는 k의 증가에 상관없이 모두 99.5% 이상의 높은 정확도를 보였다. 한편, SigHSP-I의 경우 79% 정도의 낮은 정확도를 보였으며, SigHSP-II는 98%이상의 높은 정확도를 보였다. k-최근접점 탐색 정확도 성능 측정 결과, 제안된 두 가지 기법 중 SigHSP-II가 높은 정확도를 보이며 좋은 성능을 나타냈다.



(그림 17) k 증가시 k-최근접점 탐색 정확도 (Corel 데이터 셋)

4.4 k-최근접점 탐색 시간 성능평가

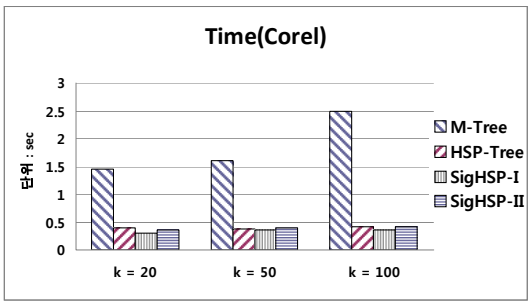
그림 18은 k=20일 때의 k-최근접점 탐색 시간을 나타낸다. Corel_UCI 데이터 셋의 경우, M-Tree가 질의당 1.5초로 가장 저하된 성능를 보여주며, 이는 역추적에 따른 노드 탐색 비용이 증가했기 때문이다. HSP-Tree와 SigHSP-I, SigHSP-II의 경우 각각 0.4초, 0.3초 0.35초로 유사한 시간을 보였는데, 이는 HSP-Tree의 데이터 공유 영역을 통해 역추적 횟수를 줄일 수 있기 때문이다. 제안한 SigHSP-Tree의 경우 시그니쳐를 이용하여 데이터에 비해 계산 비용을 감소시켜, HSP-Tree에 비해 10% 정도의 성능 향상을 나타낸다. Aerial40 데이터 셋의 경우, M-Tree의 탐색 시간은 4.95초로 가장 저하된 성능을 보여주며, HSP-Tree와 SigHSP-I, SigHSP-II의 경우 각각 1.13, 1.3, 1.13 초로 데이터 셋의 크기가 증가했음에도 불구하고 M-Tree에 비해 빠른 탐색 시간을 보인다.



(그림 18) k-최근접점 탐색 시간

그림 19는 Corel_UCI 데이터 셋에서 k = 20, 50, 100으로 증가시킬 때, k-최근접점 탐색 시간을 나타낸다. M-Tree의 경우 k=20일 때 1.4초, k=50일 때 1.65초, k=100일 때 2.5초로 k가 증가함에 따라 탐색 시간이 증가한다. 반면 HSP-Tree와 SigHSP-I, SigHSP-II의 경우 k가 증가해도 평균 0.4초 정도의 탐색 시간을 보인다. 탐색 시간에 약간 오차가 발생하는 이유는, 질의에 따라 탐색하는 공유 노드와 비공유 노드의 수가 조금씩 달라지기 때문이다. k-최근접점 탐색 시간의 성능 측정 결과, 제안

된 두 가지 기법은 HSP-Tree에 비해 최대 10% 정도의 성능 향상을 보였으며, k가 증가해도 탐색 시간이 일정하다.

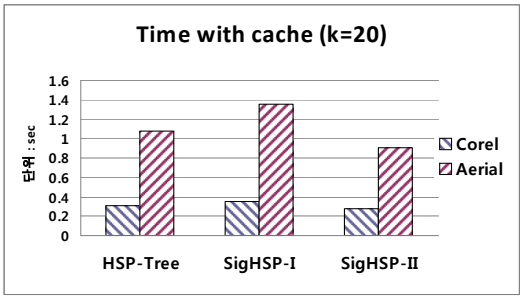


(그림 19) k 증가 시 k-최근접점 탐색 시간 (Corel 데이터 셋)

4.5 캐시를 사용한 k-최근접점 탐색 시간 성능 평가

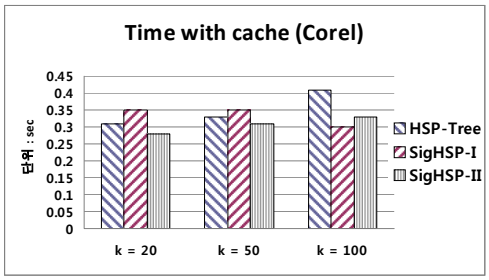
대용량 데이터 검색시 검색 속도는 중요한 요소이며, 이를 향상시킬 수 있는 방안으로는 크게 두 가지 방안이 있다. 첫째, HSP-Tree와 SigHSP-Tree의 경우, 검색 속도의 향상을 위해 공유 노드 비율을 증가시킬 수 있다. 이는 검색 시 역 추적의 횟수를 감소시켜 검색 속도가 향상된다. 그러나 공유 노드의 비율을 증가시키면, 그에 반비례하여 검색 정확도가 저하된다. 즉, 모든 노드가 공유 노드일 경우 검색 속도는 0.02초 정도로 약 20배 정도 향상되지만, 정확도는 평균 30% 정도로 크게 저하된다. 따라서 최소 정확도(90%)를 유지하기 위해서는 공유 노드 비율을 증가시킬 수 없다. 둘째, 캐시를 이용하는 방안이 있다. 즉, 캐시를 이용하여, 트리를 메모리에 상주시킴으로서 검색 시 필요한 디스크 I/O 횟수를 감소시킨다. 캐시 사용을 위해 HSP-Tree와 SigHSP-Tree의 저장공간 오버헤드를 기준으로 요구되는 캐시 크기를 설정한다. 따라서 HSP-Tree, SigHSP-Tree를 전부 캐시에 상주시켰을 때의 성능 평가를 수행하며, 각각의 캐시 크기는 147MB, 23.4MB 이다.

그림 20은 k = 20, 캐시를 사용하였을 때, k-최근접점 탐색 시간을 나타낸다. Corel_UCI 데이터 셋의 경우, 각각 HSP-Tree가 0.31초, SigHSP-I가 0.35초, SigHSP-II가 0.28초의 탐색 시간을 보인다. HSP-Tree와 SigHSP-II의 경우 캐시를 사용하지 않았을 때의 0.4초, 0.35초에 비해 캐시를 사용하였을 때 각각 약 23%, 20% 정도의 성능 향상을 나타낸다. Aerial40 데이터 셋의 경우, 각각 HSP-Tree가 1.08초, SigHSP-I가 1.35초, SigHSP-II가 0.91초의 탐색 시간을 보인다. HSP-Tree의 경우 1.13초에서 1.08초 정도로 캐시 사용시 성능이 크게 향상되지 않지만, 제안한 SigHSP-II 의 경우, 캐시 사용 전 1.13초에서 캐시 사용 후 0.91초로 약 20% 정도 성능이 향상된다.



(그림 20) 캐싱을 이용한 k-최근접점 탐색 시간

그림 21은 Corel_UCI 데이터 셋에서 캐시 사용시 k의 증가에 따른 k-최근접점 탐색 시간을 나타낸다. HSP-Tree는 k가 20, 50, 100으로 증가할 때, 0.31초 , 0.33초, 0.41초로 검색 시간이 증가함을 알 수 있다. SigHSP-I의 경우 k=20, 50일 때 0.35초로 일정하지만, k=100일 때 검색 시간이 0.3초로 향상된다. SigHSP-II의 경우 k가 20, 50, 100으로 증가함에 따라 0.28초, 0.31초, 0.33초로 검색 시간이 증가한다. 그러나 검색 시간 증가폭은 HSP-Tree에 비해서 완만하다.



(그림 21) k 증가시 캐싱을 이용한 k-최근접점 탐색 시간

캐시를 사용한 k-최근접점 탐색 시간의 성능 측정 결과, 제안된 SigHSP-II는 HSP-Tree에 비해 데이터의 용량이 커질수록, 우수한 성능을 나타내며, 캐시 사용 전에 비해 약 20% 정도의 성능 향상을 보인다. 또한 k의 증가에 따른 성능 측정 시, 검색 시간 성능의 저하가 HSP-Tree에 비해 완만하게 저하된다.

4.6 성능평가 고찰

제안한 벡터 데이터 클러스터링을 이용한 SigHSP-Tree 기법은 M-Treee 및 구글 HSP-Tree 기법보다 정확도가 저하되지만, k-최근접점 질의 처리 시간 측면에서는 10~20% 향상된 성능을 지닌다. 이러한 이유는 시그니쳐를 이용하여, 벡터 데이터 사이의 거리 계산시 비트 연산(bit operation)을 수행하여 계산 비용을 감소시켰기 때문이다. 또한 저장 공간 오버헤드 측면에서, M-Treee 및 구글 HSP-Tree 기법에 비해 3~5배 정도 성능이 향상되었다. 이는 시그니쳐를 사용하여 저장되는 벡터 데이터의 크기를 감소시켰기 때문이다. 한편, 인터넷 서비스를 제공하는 시스템은 대량의 사용자들에게 빠른 검색 결과를 반환해야 하기 때문에 높은 정확도보다 빠른 탐색 시간이 요구된다. 아울러 대량의 데이터를 다루어야 하기 때문에, 작은 저장 공간 오버헤드가 요구된다. 따라서 저장 공간 오버헤드가 작고, 가장 빠른 탐색 시간을 지니는, 벡터 데이터 클러스터링을 이용한 SigHSP-Tree기법은 동영상 내용-기반 검색을 지원하는 인터넷 서비스 시스템에 적합하다.

5. 결론 및 향후 연구

본 논문에서는 대용량의 동영상 내용-기반 검색을 효율적으로 수행하기 위한 새로운 색인 기법으로, Hybrid Spill-Tree를 개선한 SigHSP-Tree 기법을 제안하였다. 제안하는 기법은 클러스터링의 방법에 따라 시그니쳐 클러스터링 방법과 벡터 데이터 클러스터링 방법으로 나누어지며, 두 가지 클러스터링 방법을 이용한 SigHSP-Tree를 Windows 환경에서 구현하고 타 기법과 성능 평가를 수행하였다. 리얼 데이터 셋 Corel_UCI와 Aerial40을 통한 성능비교 결과, 저장공간 오버헤드는 기존의 M-Tree나 HSP-Tree에 비해 크게 감소하였다. 또한 벡터 데이터 클러스터링을 이용한 SigHSP-Tree의 경우, k-최근접점 탐색 시 정확도는 HSP-Tree와 유사한 성능을 보였으며, 시간은 10% 정도 향상된 성능을 보였다. 아울러 캐시를 사용하였을 경우, 캐시를 사용하지 않았을 때보다 약 20%정도의 성능이 향상되었으며, 전체적으로 HSP-Tree에 비해 제안한 벡터 데이터 클러스터링을 이용한 SigHSP-Tree 방법의 성능이 향상되었음을 보였다.

향후 연구로는 본 연구를 바탕으로 대용량의 고차원 벡터 데이터 검색을 효율적으로 지원할 수 있는 고차원 벡터 데이터 저장 및 검색 시스템을 설계 및 구현하는 것이다.

참 고 문헌

[1] S. Berchtold, D. A. Keim and H-P. Kriegel, "The X-tree : An Index Structure for High-Dimensional Data", Proceedings of the 22nd VLDB Conference, pp.28-39, 1996.

[2] H.I. Lin, H. Jagadish, and C. Faloutsos, "The TV-tree : An Index Structure for High Dimensional Data", VLDB Journal, Vol. 3, pp.517-542, 1995.

[3] P. Ciaccia, M. Patella, and P. Zezula, "M-tree: An efficient access method for similarity search in

metric spaces", In Proc. of the Int. Conference on Very Large Databases, 1997.

[4] Ting Liu, Andrew W. Moore, Alexander Gray and Ke Yang, "An Investigation of Practical Approximate Nearest Neighbor Algorithms", In proceedings of Neural Information Processing Systems(NIPS 2004), Vancouver, 2004

[5] Ting Liu, Charles Rosenberg, Henry A. Rowley, "Clustering Billions of Images with Large Scale Nearest Neighbor Search", IEEE Workshop on Applications of Computer Vision, 2007

[6] Jiangtao Cui, Shuisheng Zhou, and Shan Zhao1, "PCR-tree: a Compression-based Index Structure for Similarity Searching in High-dimensional Image Databases", Fuzzy Systems and Knowledge Discovery(FSKD), 2007.

[7] Hyun Hwa Choi, Hun Soon Lee, Kyeong Hyeon Park, and Mi Young Lee, "GLORY-DB: A Distributed Data Management System for Large Scale High-Dimensional Data", The 23rd International Technical Conference on Circuits/Systems, 2008.

[8] 이규웅, 이훈순, 이미영, 김명준, "클러스터 파일 시스템의 고확장성 지원을 위한 고차원 인덱스 기법", 한국정보기술학회논문지 제6권 제6호, pp. 209-217, 2008.

[9] Roger Weber, Hans-J. Schek, Stephen Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces", submitted publication, 1998.

[10] 송광택, 장재우, "고차원 멀티미디어 데이터에 대한 근사 k-최근접 데이터 탐색 알고리즘의 설계", 한국 데이터베이스 학술대회 논문집, Vol. 15, No. 1, pp.261-265, 1999.

[11] 장재우, 한성근, 김현진, "셀 기반 필터링 방법을 이용한 고차원 색인 기법", 정보과학회 논문지 제 28권 2호, pp.204-216, 2001.

[12] http://www.autonlab.org/autonweb/15960.html

◐ 저 자 소 개 ◑

이 현 조 (Hyun-Jo Lee)
2006년 전북대학교 컴퓨터공학과 졸업 (공학사)
2008년 전북대학교 대학원 컴퓨터공학과 졸업(공학석사)
2008년~현재 전북대학교 대학원 컴퓨터공학과 박사과정
관심분야 : 데이터 마이닝, 공간 데이터베이스, 고차원 색인 구조
E-mail : hjlee@dblab.chonbuk.ac.kr

홍 승 태 (Seung-Tae Hong)
2008년 전북대학교 컴퓨터공학과 졸업 (공학사)
2008년~현재 전북대학교 대학원 컴퓨터공학과 석사과정
관심분야 : 공간 데이터베이스, 센서 네트워크
E-mail : sthong@dblab.chonbuk.ac.kr

나 소 라(So-Ra Na)
2008년 전북대학교 컴퓨터공학과 졸업 (공학사)
2008년~현재 전북대학교 대학원 컴퓨터공학과 석사과정
관심분야 : 데이터 마이닝, 고차원 색인 구조, 센서 네트워크
E-mail : srna@dblab.chonbuk.ac.kr

장 유 진(You-Jin Jang)
2008년 전북대학교 컴퓨터공학과 졸업 (공학사)
2008년~현재 전북대학교 대학원 컴퓨터공학과 석사과정
관심분야 : 센서 미들웨어, 고차원 색인 구조, 데이터 마이닝
E-mail : yjjang@dblab.chonbuk.ac.kr

장 재 우 (Jae-Woo Chang)
1984년 서울대학교 전자계산기공학과 졸업 (공학사)
1986년 한국과학기술원 전산학과 졸업 (공학석사)
1991년 한국과학기술원 전산학과 졸업 (공학박사)
1996년~1997년 Univ. of Minnesota, Visiting Scholar
2003년~2004년 Penn State Univ., Visiting Scholar
1991년~현재 전북대학교 전기전자컴퓨터공학부 교수
관심분야 : 공간 네트워크 데이터베이스, 상황인식, 하부저장구조
E-mail : jwchang@chonbuk.ac.kr

심 춘 보 (Chun-Bo Sim)
1996년 전북대학교 컴퓨터공학과 졸업 (공학사)
1998년 전북대학교 대학원 컴퓨터공학과 졸업 (공학석사)
2003년 전북대학교 대학원 컴퓨터공학과 졸업 (공학박사)
2004년~2005년 부산가톨릭대학교 컴퓨터정보공학부 전임강사
2005년~현재 순천대학교 정보통신공학부 조교수
관심분야 : 멀티미디어 DB, LBS, 유비쿼터스 컴퓨팅
E-mail : cbsim@sunchon.ac.kr