다차원 데이타를 위한 공간 분할 및 적응적 비트 할당 기반 색인 구조
(An Index Structure based on Space Partitions and Adaptive Bit Allocations for Multi-Dimensional Data)

복 경 수 † 김 은 재 †† 유 재 수 †††
(Kyoung Soo Bok) (Eun Jae Kim) (Jae Soo Yoo)

요 약 본 논문에서는 다차원 데이타의 유사도 검색을 효율적으로 지원하기 위한 벡터 근사 기반의 색인 구조를 제안한다. 제안하는 색인 구조는 공간 분할 방식으로 영역을 분할하고 실제 데이타들이 존재하는 영역에 대해 동적 비트를 할당하여 영역을 표현한다. 따라서, 분할된 영역들 사이에 겹침이 발생하지 않으며 하나의 중간 노드에 많은 영역 정보를 저장할 수 있어 트리의 깊이를 감소시킨다. 또한, 특정 영역에 군집화되어 있는 데이타에 대해서 효과적인 표현 기법을 제공하며 자식 노드의 영역 정보는 부모 노드의 영역 정보를 이용하여 상대적으로 표현함으로써 영역 표현에 대한 정확성을 보장한다. 이를 통해 검색 성능 향상을 제공한다. 제안하는 색인 구조의 우수성을 보이기 위해 기존에 제안된 다차원 색인 구조와의 다양한 실험을 통하여 성능의 우수성을 입증한다. 성능 평가 결과를 통해 제안하는 색인 구조가 기존 색인 구조보다 40% 정도 검색 성능이 향상됨을 증명한다.
키워드 : 다차원 데이타, 색인 구조, 차원의 저주, 벡터 근사

Abstract In this paper, we propose the index structure based on a vector approximation for efficiently supporting the similarity search of multi-dimensional data. The proposed index structure splits a region with the space partition method and allocates to the split region dynamic bits according to the distribution of data. Therefore, the index structure splits a region to the unoverlapped regions and can reduce the depth of the tree by storing the much region information of child nodes in a internal node. Our index structure represents the child node more exactly and provide the efficient search by representing the region information of the child node relatively using the region information of the parent node. We show that our proposed index structure is better than the existing index structure in various experiments. Experimental results show that our proposed index structure achieves about 40% performance improvements on search performance over the existing method.
Key words : Multi-Dimensional Data, Index Structure, Dimensionality Curse, Vector Approximation

1. 서 론

컴퓨터 처리 기술의 발전과 더불어 90년대 초반부터 다양한 응용 분야에 다차원 데이타에 대한 사용이 증가되고 있다. 이와 함께 이러한 다차원 데이타를 효과적으로 검색하기 위한 많은 연구들이 진행되고 있다. 다차원 데이타를 활용하는 대표적인 응용 분야인 내용 기반 이미지 검색에서는 이미지에서 추출한 다차원 특징 값을 이용한 검색 조건과 유사한 이미지를 검색하는 유사도 검색(similarity search)을 수행한다. 다차원의 특징 값은 하나의 이미지를 대표하는 값으로 보통 자동화된 특징 추출 기법에 의해 이미지를 대표할 수 있는 값으로 표현한다.

다차원 데이타에 대한 활용이 증가되면서 대용량의 다차원 데이타들을 효과적으로 처리하기 위한 다수의 색인 구조들이 제안되었다[1,2]. 이러한 색인 구조들은 이미지로부터 추출된 다차원의 특징 값을 다차원 공간에서 하나의 점으로 간주하고 색인을 구성한다. 기존에 제안된 다차원 색인 구조는 다차원의 데이타 영역을 분

· 본 연구는 한국과학재단 목적기초연구(특정기초연구 과제번호 : R01-2003-000-10627-0) 지원 및 산업자원부의 지역혁신 인력양성사업의 연구결과로 수행되었음
† 비 회 원 : 한국과학기술원 전산학과
ksbok@dbserver.kaist.ac.kr
†† 비 회 원 : 충북대학교 정보통신공학과
defeat@netdb.cbnu.ac.kr
††† 종신회원 : 충북대학교 전기전자컴퓨터공학부 교수
yjs@cbucc.cbungbuk.ac.kr
논문접수 : 2004년 9월 10일
심사완료 : 2005년 6월 9일

할하는 방법에 따라 공간 분할(space partition)방식과 데이타 분할(data partition)방식으로 구분한다[3]. 공간 분할 방식은 전체 데이타 공간에 대해 데이타의 분포를 고려하지 않고 겹침이 없는 영역으로 분할하는 기법으로 KDB-트리[4], LSD-트리[5], LSD^h-트리[6], Bkd-트리[7], KDB_{KD}-트리[8] 등이 있다. 그러나 공간 분할 방식은 실제 데이타가 존재하는 영역만을 표현하는 것이 아니라 전체 공간을 표현하기 때문에 사장 공간(Dead Space)이 발생하는 문제점이 있다. 이에 반해, 데이타 분할 방식은 데이타의 분포 특성을 고려하여 분할된 영역에 대해 객체가 존재하는 최소 영역을 표현하는 방법으로 R-트리[9], R^*-트리[10], TV-트리[11], X-트리[1] 등이 있다. 데이타 분할 방식은 전체 데이타 공간에서 실제 데이타가 존재하는 영역만을 표현하기 때문에 기존의 공간 분할 방식의 단점인 사장 공간을 효과적으로 감소시킨다. 그러나 이러한 색인 구조들은 차원이 증가할수록 노드의 팬아웃(fanout)이 감소되어 색인 구조의 높이를 증가시킬 뿐만 아니라 분할된 영역들 사이에 겹침이 증가되어 검색 성능이 저하되는 문제점이 있다. 특히, K-최근접 검색(K-nearest neighbor search)에 대해서는 차원이 증가할수록 검색 성능이 급속히 저하되는 문제점이 있다. 이러한 문제를 차원의 저주(dimensionality curse) 현상이라고 한다.

최근 이러한 차원의 저주 현상을 해결하기 위한 많은 연구들이 진행되었다[12-15]. 차원의 저주 현상을 해결하기 위한 기법으로 전체 데이타 공간을 특정 크기로 분할하고 분할된 영역을 비트 형태로 표현하는 벡터 근사 기법에 대한 연구들이 진행되고 있다[14,16-19]. 벡터 근사 기법의 하나인 VA-파일(Vector Approximation-File)은 각 차원에 특정 비트를 할당하여 전체 데이타 영역을 2^b개의 영역으로 분할한다[12]. 각 분할된 영역에는 길이가 b인 유일한 비트 값을 할당하여 데이타 값들을 근사화한다. 근사화된 데이타 값들을 배열에 저장하고 전체 배열을 검사하면서 검색을 수행한다. CS-트리(Cell based Signature-Tree)는 전체 데이타 공간을 VA-파일과 유사한 방법으로 분할하고 근사화된 값을 R-트리 기반의 색인 구조에 표현한다[17]. A-트리(Approximation-Tree)는 CS-트리와 같이 전체 데이타 공간을 분할하는 것이 아니라 R-트리 기반의 계층 구조의 영역을 기준으로 상대적인 영역을 표현하기 위해 비트 형태로 표현된 VBR(Virtual Bounding Rectangle)을 사용한다[18,19]. 벡터 근사화 기법을 사용하는 색인 구조들은 데이타들이 존재하는 실제 영역보다 큰 영역을 표현한다. 그러나 이러한 색인 구조들은 팬아웃을 증가시켜 색인 구조의 높이를 감소시키는 장점이 있다. 이로 인해 팬아웃을 증가시킬 수 있을 뿐만 아니라 검색 과정에서 탐색해야 할 노드의 수를 감소시킬 수 있다. 그러나 대부분 R-트리 기반의 색인 구조를 이용하기 때문에 차원의 증가에 따라 분할된 영역들 사이의 겹침 영역이 증가된다는 문제점이 있다. 또한, 분할된 영역을 표현하기 위해 고정된 비트 값을 할당하기 때문에 근사화된 영역들에 대한 정확도가 감소된다. 특히, 데이타들이 일부 영역에 군집화되어 있을 경우 검색 성능이 저하된다. 또한 영역의 분포 특성을 고려하지 않고 정해진 비트 값을 할당하기 때문에 계속적인 삽입과 삭제가 발생할 경우 분할된 영역들 사이에 선별력이 저하되어 검색 성능에 많은 영향을 미치게 된다.

따라서 본 논문에서는 기존에 제안된 벡터 근사 기법의 문제점을 해결하기 위한 새로운 다차원 색인 구조를 제안한다. 제안하는 색인 구조는 KDB-트리와 유사한 공간 분할 방식에 의해 분할을 수행하고 실제 데이타들이 존재하는 영역에 대해서만 벡터 근사화를 수행하는 높이 균형트리이다. 제안하는 색인 구조는 분할된 영역에 대해 벡터 근사화를 수행한 영역을 표현하기 때문에 노드의 팬아웃을 증가시킬 수 있으며 이로 인해 트리의 높이를 감소시킬 수 있다. 분할된 영역에 대한 명확한 근사화를 수행하기 위해 계층 구조 내에 존재하는 부모 노드를 기준으로 상대적인 영역을 표현하고 데이타의 분포 특성에 따라 분할된 영역에 동적 비트를 할당하여 BMBR(Bit Minimum Bounding Region)을 표현한다. 이를 위해 각 중간 노드에는 상대적 영역을 표현하기 위한 헤더 정보를 저장한다. 단말 노드는 기존의 공간 분할 방식에 데이타의 분포 특성을 고려한 분할을 수행하고 중간 노드는 영역들 사이에 겹침이 발생하지 않으면서도 연속적인 분할(cascading split)을 발생시키지 않는 분할을 수행한다.

본 논문의 구성은 다음과 같다. 2장에서는 기존에 제안된 다차원 색인 구조의 특징 및 문제점에 대해 기술한다. 3장에서는 기존에 제안된 색인 구조의 문제점을 해결하기 위해 본 논문에서 제안하는 색인 구조의 특징을 기술하고 4장에서는 제안하는 색인 구조의 삽입 및 분할 기법에 대해 자세히 기술한다. 5장에서는 제안하는 색인 구조의 우수성을 입증하기 위해 대용량의 다차원 데이타를 사용하여 기존에 제안된 색인 구조와의 성능 평가를 수행한다. 마지막으로 6장에서는 결론 및 향후 연구에 대해 기술한다.

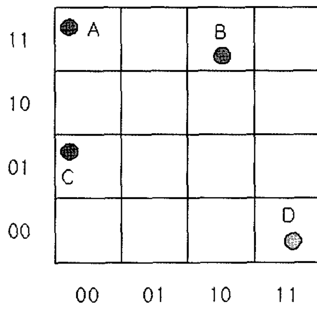
2. 관련 연구

기존에 제안된 대부분의 다차원 색인 구조는 중간 노드에 분할된 영역을 표현하기 위해 많은 저장 공간을 필요로 한다. 예를 들어, 기존의 R-트리 계열의 색인 구조는 d차원 데이타에 대해 중간 노드에서는 $2d$의 정보

를 필요로 하기 때문에 하나의 노드 내에 많은 영역 정보를 저장할 수 없는 단점이 있다. 최근 다차원 색인 구조에 대한 연구는 "차원의 저주" 현상을 극복하기 위해 중간 노드에 존재하는 영역 정보를 감소시키기 위한 방향으로 진행되고 있다[14-16,20]. 이러한 방법의 하나로 전체 데이타 공간을 작은 단위의 영역으로 분할하고 분할된 영역에 특정 비트를 할당하여 분할 영역을 비트 단위로 표현하기 위한 벡터 근사화 기법에 대한 연구들이 진행되고 있다[12,17,18].

VA-파일은 전체 데이타 공간상에 존재하는 데이타들에 대해 근사화를 수행하여 비트 형태로 표현하기 위한 객체 근사화를 지원하는 색인 구조이다[12]. 이러한 VA-파일은 사용자가 정의한 b개의 비트를 이용하여 데이타 공간을 2^b 개의 영역으로 분할한다. 분할된 각 영역에 b개의 비트를 할당하여 데이타들을 근사화한다. 그림 1은 데이타 공간의 각 차원이 0~1 범위에 존재한다고 할 때 VA-파일에서 데이타 포인터를 근사화하는 예이다. 그림 1의 (a)에서 보는 것과 같이 분할된 각 영역에 2비트를 할당할 때 데이타 영역은 16개로 분할되고 분할된 영역 내에 포함된 데이타들은 그림 1의 (b)와 같이 근사화된다. 그러나 VA-파일은 근사화된 데이타 값들을 배열 구조에 저장하기 때문에 K-최근접 검색 과정에서 근사화된 값들을 순차적으로 순회하며 검색을 수행한다. 따라서, 데이타 수의 증가에 따라 검색 성능이 저하된다. 또한 데이타의 분포가 특정 영역에 군집화되어 있을 경우 근사화된 값들 사이에 선별력이 저하되기 때문에 검색 성능이 급격히 저하되는 문제점이 있다.

CS-트리는 기존 R-트리 기반의 다차원 색인 구조에서 차원이 증가할수록 팬아웃이 감소하여 트리의 높이가 증가한다는 문제점을 해결하기 위해 데이타 공간을 일정한 크기로 분할하고 분할된 영역에 대한 근사화를 수행한다[17]. CS-트리는 근사화된 영역을 표현하기 위해 실제 MBR을 특정 비트가 할당된 영역들이 포함할 수 있도록 CMBR(Cell based MBR)를 사용한다. 그림 2는 각 차원이 0~1 범위에 존재할 때 실제 MBR를 CMBR로 표현하는 과정을 나타낸 것이다. 그림 2의 (a)와 같이 분할된 영역에 대한 실제 MBR이 R_1과 R_2와 같다고 할 때, CS-트리는 R_1과 R_2 영역을 비트 형태로 표현하기 위해 CR_1과 CR_2 영역을 생성한다. 이렇게 생

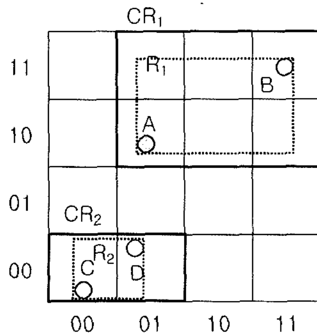


(a) 데이타 공간

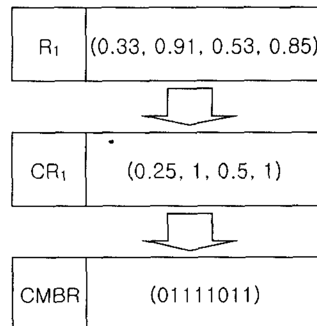
	벡터 데이터	근사화
A	(0.1, 0.9)	0011
B	(0.6, 0.8)	1011
C	(0.1, 0.4)	0001
D	(0.9, 0.1)	1100

(b) 근사화

그림 1 VA-파일



(a) 분할 영역



(b) CMBR 생성 과정

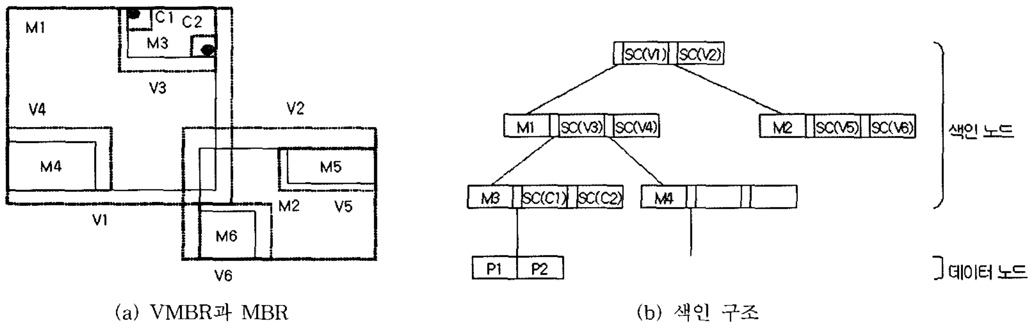
그림 2 CS-트리

성된 영역은 그림 2의 (b)와 같은 과정을 통해 비트 형태로 표현된 *CMBR*를 생성한다. CS-트리는 전체 데이타 공간에 대해 분할된 영역을 비트 형태로 표현하기 때문에 데이타가 균등 분포 형태로 존재할 때는 효과적인 색인 구조이다. 그러나 전체 영역을 고정된 크기의 영역으로 분할하기 때문에 객체의 동적인 삽입이나 삭제에 의해 정확도가 감소될 수 있으며 VA-파일과 같이 특정 영역에 군집화되어 있을 경우 검색 성능이 저하되는 문제가 발생할 수 있다.

A-트리는 CS-트리와 같이 전체 데이타 공간을 고정된 크기의 영역으로 분할하는 것이 아니라 부모 노드에 존재하는 영역을 기준으로 자식 노드에 대한 영역을 상대적으로 표현하기 위한 색인 구조이다[18,19]. 이러한 A-트리는 부모 노드의 영역을 기준으로 상대적인 자식 노드에 대한 영역을 VMBR(Virtual MBR)로 표현한다. 그림 3의 (a)는 A-트리에서 사용하는 VMBR을 표현한 예이다. VMBR은 부모 영역을 기준으로 특정 비트를 이용하여 그림 3의 (b)와 같이 색인 구조를 구성한다. 중간 노드의 처음에 존재하는 M_i는 자식 영역을 표현하기 위한 기준 영역을 나타내고 $SC(V_i)$는 실제 데이타들이 존재하는 영역에 대한 VMBR을 표현한 것이다. 그러나 자식 노드에 대한 영역 정보는 부모 노드를 기준으로 표현하기 때문에 부모 노드에 대한 영역이 변경될 경우 자식 노드를 모두 재구성해야 한다는 문제점이 있다. 또한 계속적으로 데이타가 삽입될 경우 분할된 영역들 사이의 겹침 영역이 증가되어 검색 성능이 저하된다는 문제점이 있다.

기존 벡터 근사 기반의 색인 구조는 데이타 객체가 존재하는 영역을 비트 형태로 표현하기 때문에 노드의 팬아웃을 증가시켜 다차원의 데이타를 효과적으로 색인할 수 있다. 그러나 계속적인 데이타의 삽입으로 분할된 영역들 사이에 노드의 겹침이 증가한다는 문제점이 있다. 또한 데이타의 분포 특성이 일정하게 분포되어 있지 않고 특정 영역에 군집화될 경우 분할된 영역들 사이의 선별력이 감소한다. 따라서 검색 성능이 저하될 수 있다는 문제점이 있다. 기존에 제안된 벡터 근사 기반의 색인 구조의 특징은 표 1과 같다.

[21,22]에서는 R-트리 기반의 다차원 색인 구조에 대한 K-최근접 검색 기법을 제안하였다. [21]에서는 R-트리 기반 색인 구조에서 최근접한 객체를 검색하기 위해 MINDIST와 MINMAXDIST를 정의하고 분기 한계(branch and bound) 기법에 대해 색인 구조를 탐색하는 기법을 제안하였다. [21]에서는 최근접 검색 과정에서 탐색해야할 자식 노드들을 제거하기 위해 질의와 특정 노드의 MINMAXDIST보다 큰 MINDIST을 갖는 노드는 제거한다. 또한, 단말 노드에 존재하는 객체와의 실제 거리가 특정 노드의 MINMAXDIST보다 클 경우 제거한다. 그러나 다차원 색인 구조에서 MINMAXDIST는 많은 계산 비용을 소요할 뿐만 아니라 실제 객체와 큰 차이를 보이기 때문에 탐색해야할 자식 노드를 거의 제거하지 못하는 문제점이 있다. 따라서, [22]에서는 MINMAXDIST를 계산하지 않고 MINDIST만을 사용하는 K-최근접 검색 기법을 제안하였다.



(a) VMBR과 MBR (b) 색인 구조

그림 3 A-트리 색인 구조

표 1 기존 색인 구조의 특징

색인구조 \ 특징	영역 표현	저장 구조
VA-파일	전체 영역에 대해 고정된 비트를 할당	배열
CS-트리	전체 영역에 대해 고정된 비트를 할당	R-트리 기반 색인 구조
A-트리	부노 노드를 기준으로 상대적인 영역을 표현	R-트리 기반 색인 구조

공간 분할 색인 구조에 대표적인 KDB-트리에서는 중간 노드의 분할로 인해 자식 노드를 계속적으로 분할해야하는 연속 분할(cascading split)이 발생한다. 이러한 문제점을 해결하기 위해 [23]에서는 연속 분할을 발생시키지 않으면서 KDB-트리의 검색 성능을 향상시키기 위한 새로운 분할 전략을 제안하였다.

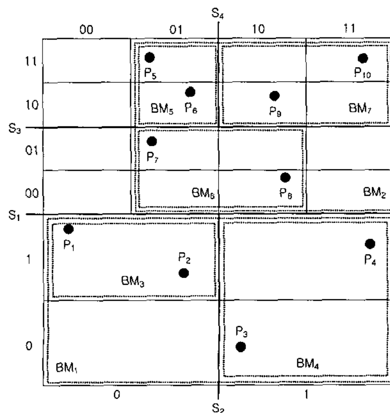
3. 제안하는 색인 구조

3.1 색인 구조의 특징

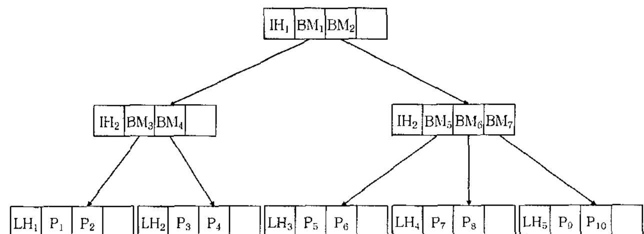
제안하는 색인 구조는 기존의 다차원 색인 구조의 문제점을 해결하기 위해 다차원 데이타 공간을 공간 분할 기법을 이용하여 분할하고 분할된 영역은 실제 데이타가 존재하는 구간에 대해 부모 노드를 기준으로 상대적인 영역을 표현하는 벡터 근사화 트리이다. 제안하는 색인 구조는 기존에 제안된 색인 구조의 문제점을 해결하기 위해 다음과 같은 특징을 고려한다. 첫째, 기존 R-트리 기반의 벡터 근사화 기법은 차원이 증가할수록 계속적인 삽입이나 삭제로 인해 분할된 영역들 사이에 겹침 영역이 증가되어 검색 성능이 저하된다. 이러한 문제점을 해결하기 위해 삽입 및 분할은 공간 분할 방식을 사용하고 실제 데이타들이 존재하는 영역에 대해 벡터 근사화를 수행한다. 공간 분할 방식은 분할된 영역들 사이에 겹침을 제거시킬 수 있으며 실제 데이타들이 존재하는 영역에 대한 근사화를 수행할 경우 영역의 크기를 최소화시켜 검색 성능을 향상시킬 수 있다. 둘째, 기존에 제안된 색인 구조는 벡터 근사화 기법을 수행하기 위해 전체 영역 또는 분할된 일부 영역에 고정된 비트를 할당하여 노드의 팬아웃을 증가시켰다. 그러나 이러한 기법들은 특정 영역에 데이타들이 군집화되어 있을 경우 분할된 영역들 사이에 선별력이 감소될 수 있다. 따라서 제안하는 색인 구조에서는 분할된 영역에 고정된 비트를 할당하는 것이 아니라 데이타의 분포 상태에 따라 동적으로 비트를 할당하여 BMBR를 나타낸다. 셋째, 색인을 구성하기 위해 전체 데이타 공간에 대해 고정된 비트를 미리 할당할 경우 계속적인 데이타의 삽입이나 삭제로 인해 분할된 영역에 대한 선별력이 저하되어 검색 성능이 저하될 수 있다. 이러한 문제점을 해결하기 위해 제안하는 색인 구조는 색인 구조의 계층에 따라 분할된 자식 노드는 부모 노드를 기준으로 상대적인 영역을 표현한다.

제안하는 색인 구조는 높이 균형 트리로 노드에 오버플로우가 발생할 경우 공간 분할을 수행하고 실제 데이타들이 존재하는 영역에 대해서만 근사화된 영역을 표현한다. 그림 4는 각 차원의 0~10 사이의 범위를 갖는 2차원 데이타 공간에서 제안하는 색인 구조를 구성한 예이다. 그림 4에서 P_i는 2차원 데이타 공간에 존재하는 객체를 나타내며 BM_i는 분할된 노드에 대해 벡터 근사화를 수행한 BMBR을 나타낸다. 그림 4의 (a)와 같이 다차원의 데이타 공간이 분할되어 있다고 할 때 이를 통해 색인을 구성하면 그림 4의 (b)와 같다.

그림 4는 S_1을 기준으로 분할된 두 개의 영역을 생성하고 분할된 각 영역은 BM_1과 BM_2로 근사화하여 루트 노드에 저장한다. 또한, BM_1는 S_2을 기준으로 분할되어 BM_3과 BM_4을 영역을 생성하고 BM_2는 S_3과 S_4을 기준으로 세 개의 분할된 영역 BM_5, BM_6, BM_7을 생성한다. 제안하는 색인 구조는 공간 분할에 의해 분할된 영역에 동적인 비트를 할당하여 하위 노드에 대한 상대적인 비트를 표현한다. 따라서, 그림 4의 (b)에서 보는 것과 같이 중간 노드의 처음에는 *BMBR*을 나타내기 위해 필요한 기준 정보를 제공하기 위한 헤더를 저장한다. 단말 노드에는 오버플로우가 발생할 경우 공간 분할을 수행하기



(a) 2차원 데이타 공간



(b) 색인 구조

그림 4 제안하는 색인 구조

위해 기존에 분할된 영역을 헤더로 포함하고 있다. 그림 4의 (b)에서 노드의 회색으로 표현된 부분은 헤더 정보로 자식 노드에 대한 영역 정보 *BMBR*을 표현하기 위한 기준 정보를 나타낸다. 그림 4의 (b)에서 나타난 LH_i와 IH_i는 단말 노드와 중간 노드의 헤더를 나타낸다. 헤더의 첫 번째 값은 *BMBR*을 표현하기 위해 기준 영역 *SP*, 두 번째 값은 *BMBR*을 표현하기 위해 필요한 비트 수 *BitNum* 그리고 세 번째 값은 노드에 존재하는 엔트리들이 공간 분할을 수행한 분할 정보 *PI*을 나타낸다. 그림 4와 같이 BM_1의 자식 노드에 존재하는 BM_3과 BM_4을 표현하기 위해 헤더에 *BitNum*이 2이라면 BM_3는 (0101)이고 BM_4는 (1011)이다. 이와 유사하게, BM_2의 자식 노드에 존재하는 BM_5, BM_6, BM_7을 표현하기 위해 헤더에 *BitNum*이 3이라면 BM_5는 (01100111), BM_6는 (01001001), BM_7는 (10101111)이다.

3.2 노드 구조

중간 노드는 공간 분할 방식에 의해 노드를 분할하고 단말 노드에 존재하는 다차원 데이타를 포함하는 영역을 표현하기 위해 동적 비트를 할당하여 근사화된 영역을 표현한다. 이러한 중간 노드는 자식 노드의 영역을 표현하는 엔트리와 헤더로 구성된다. 그림 5는 중간 노드의 구조를 나타낸 것이다. 중간 노드에 존재하는 헤더는 동적인 비트 할당을 통해 자식 노드의 영역을 표현하기 위한 기준 정보로 $<SP, BitNum, PI>$와 같다. *SP*는 자식 노드를 포함하는 영역 정보로 자식 노드의 영역 정보 *BMBR*을 표현하기 위한 기준 영역이다. 이러한 *SP*는 공간 분할에 의해 생성된 영역으로 동일한 레벨에 존재하는 *SP*들 사이에는 겹침 영역이 발생하지 않는다. *BitNum*는 자식 노드에 대한 *BMBR*을 표현하기 위해 할당된 비트 수로 분할 과정에서 데이타의 분포

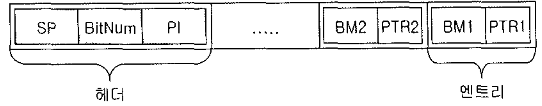
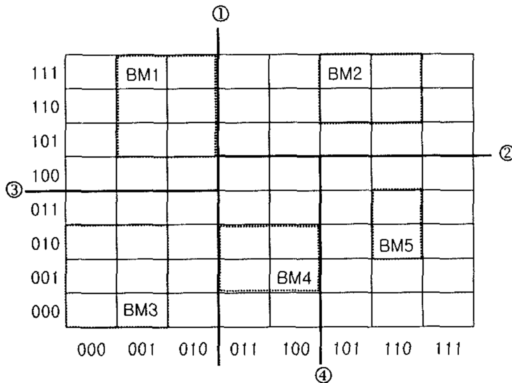


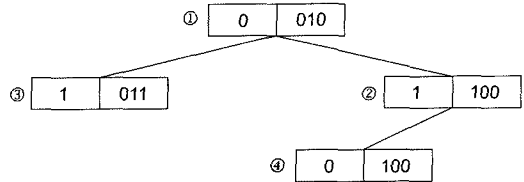
그림 5 중간 노드 구조

특성에 따라 동적으로 결정된다. *PI*는 자식 노드에 대한 분할 정보로 헤더에 존재하는 *SP*을 기준으로 자식 노드에 대한 공간 분할을 수행한 정보를 나타낸다. 중간 노드에 존재하는 엔트리는 하위 노드를 포함하는 영역을 표현하기 위한 영역 정보를 표현하는 것으로 $<BM_i, PTR_i>$와 같다. BM_i는 헤더에 포함된 *SP*을 기준으로 *BitNum*의 수의 비트로 표현된 하위 노드에 대한 영역 정보 *BMBR*을 나타내며 PTR_i는 자식 노드에 대한 포인터를 나타낸다.

자식 노드에 대한 영역 정보를 나타내는 *BMBR*는 실제 데이타들이 존재하는 영역에 대해서 근사화를 수행하기 때문에 중간 노드에 오버플로우가 발생할 경우 *BMBR*을 통해 공간 분할을 수행할 수 없다. 노드에 오버플로우가 발생하여 공간 분할 방식을 수행하기 위해서는 자식 노드에 존재하는 영역들에 대한 분할 정보 즉, 공간 분할 방식을 수행한 정보를 포함해야 한다. 헤더에 존재하는 *PI*는 *SP*을 기준으로 자식 노드가 공간 분할을 수행한 정보를 나타내어 공간 분할 방식에 의해 분할을 수행할 위치를 선택하거나 새로운 데이타가 삽입될 위치를 선택하기 위해 사용된다. 그림 6은 중간 노드에 존재하는 분할 정보 *PI*을 나타낸 것이다. 그림 6의 (a)와 같이 번호 순서에 의해 5개의 분할된 영역이 존재한다고 하면, BM_i는 영역 분할에 의해 생성된 자식 노드에 대한 *BMBR*을 나타낸 것이다. 분할된 5개의 영역에 대한 분할 정보 *PI*는 그림 6의 (b)와 같이 분할 차원과 분할 위치의 쌍$<SplitAxis, SplitPosition>$에 의



(a) 분할 영역



(b) 분할 정보

그림 6 중간 노드의 분할 영역 및 분할 정보

해 표현된다. 분할 차원과 분할 위치를 나타내는 *SplitAxis*과 *SplitPosition*는 비트 형태로 표현된다. 이러한 *PI*는 이진 트리 형태로 구성되지만 실제 저장은 이진 트리 형태를 저장하는 것이 아니라 이진 트리를 순회한 순서에 따라 배열 형태로 저장한다.

단말 노드는 실제적인 다차원의 데이타를 저장한다. 단말 노드는 그림 7에서 보는 것과 같이 중간 노드와 유사하게 헤더와 엔트리로 구성된다. 단말 노드의 엔트리 $< FV_i, OID_i >$는 실제적인 다차원의 데이타 FV_i와 객체 식별자 OID_i로 구성되어 있다. 다차원의 데이타 FV_i는 다차원의 포인트 데이타 $(V_1, V_2, \ldots, V_n)$을 나타낸다. 단말 노드의 헤더는 *SP*로 구성되어 있으며 *SP*는 기존의 공간 분할에 의해 분할된 영역 정보를 나타낸다. 이러한 *SP*는 기존의 단말 노드에 공간 분할을 수행하여 생성된 영역으로 단말 노드에 공간 기반 분할을 수행하기 위한 기준 영역을 나타낸다. 즉, 단말 노드에는 다차원의 데이타를 포함하고 있기 때문에 실제적인 데이타들이 존재하는 영역만을 나타낸다. 따라서, 공간 분할을 수행하기 위해 기존에 공간 분할에 의해 생성된 영역을 표현한다.

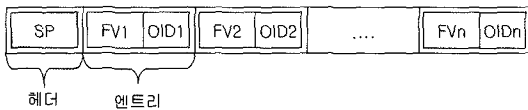


그림 7 단말 노드의 구조

3.3 BMBR의 표현

중간 노드에 존재하는 엔트리들은 자식 노드에 대한 영역 정보를 표현하기 위해 부모 노드를 기준으로 분할된 영역에 동적 비트를 할당하여 근사화된 영역 *BMBR*을 나타낸다. 중간 노드에 존재하는 엔트리들은 부모 노드를 기준으로 공간 분할을 수행하기 때문에 분할된 영역들 사이에는 겹침 영역이 발생하지 않는다. 또한, 공간 분할에 의해 생성된 영역은 분할된 영역 내에서 실제 데이타들이 존재하는 영역에 대해 동적 비트를 할당하기 때문에 실제 영역에 근접한 영역을 표현할 수 있다. 이로 인해 자식 노드에 대한 선별력을 증가시킨다. 그림 8은 2차원 데이타 공간에서 자식 노드에 대한 *BMBR*을 표현한 예이다. *BMBR*을 표현하기 위해 기준이 되는 부모 노드의 영역 즉, 중간 노드의 헤더에 존재하는 기준 영역 *SP*가 (0,0,10,10)이고 *BMBR*을 표현하기 위해 3개의 비트가 할당되었다고 가정하자. S_1, S_2, S_3을 기준으로 *SP*에 대한 공간 분할을 수행하여 생성된 4개의 자식 노드를 *BMBR*로 표현하면 표 2와 같다.

영역의 특정 위치에 데이타들이 군집화되어 있을 때

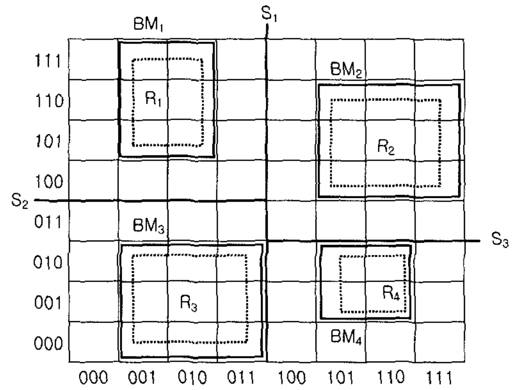


그림 8 분할된 자식 노드

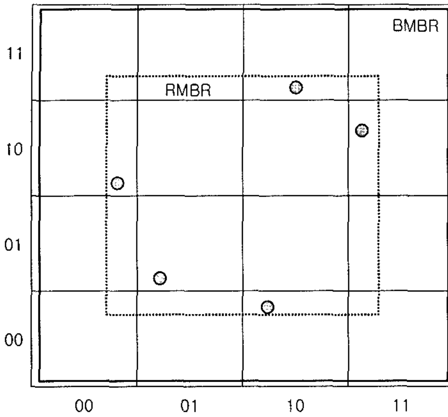
표 2 BMBR 표현

	RMBR	*BMBR*
R_1	(2, 7, 3.5, 9.5)	(001, 101, 010, 111)
R_2	(6.5, 5.5, 9.5, 8)	(101, 100, 111, 110)
R_3	(2, 1, 4.5, 3.5)	(001, 000, 011, 010)
R_4	(7, 1.5, 8.5, 3.5)	(101, 001, 110, 010)

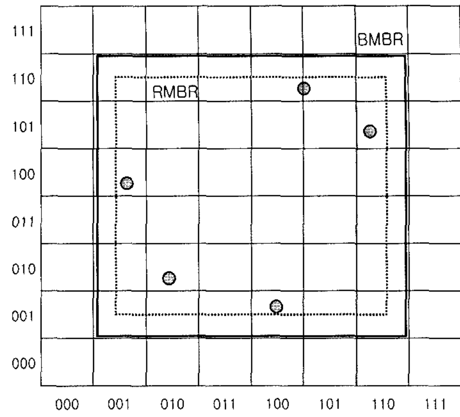
자식 영역을 표현하기 위해 고정된 비트를 할당할 경우 분할된 영역들 사이에 선별력이 저하된다. 제안하는 색인 구조에서는 분할된 영역에 대한 선별력을 증가시키기 위해 데이타의 분포 특성에 따라 동적 비트를 할당한다. *BMBR*을 표현하기 위해 많은 비트를 할당할 경우 실제 데이타들이 존재하는 영역에 근접한 *BMBR*을 생성할 수 있다. 그러나 많은 비트를 할당할 경우 많은 저장 공간을 필요로 하기 때문에 노드의 팬아웃이 저하되어 색인 구조의 높이를 증가시킬 수 있다. 이로 인해 검색 과정에서 탐색해야 할 노드의 수를 증가시켜 검색 성능을 저하시킬 수 있다. 따라서, 효율적인 *BMBR*을 표현하기 위해서는 부모 노드를 기준으로 자식 영역을 표현하기 위해 필요한 비트 수를 결정하는 것이 필요하다.

자식 노드에 대한 *BMBR*을 표현하기 위해 필요한 비트 수는 실제 데이타들이 존재하는 *RMBR*과 근사화에 의해 생성된 영역 *BMBR*의 유사도에 의해 결정한다. 근사화에 의해 생성된 *BMBR*은 항상 *RMBR*보다 크거나 같다. 따라서, *BMBR*과 *RMBR*의 차이가 작을수록 실제 영역에 근접한 *BMBR*을 생성할 수 있다. 영역의 넓이를 $Ex()$라 할 때, *BMBR*을 위해 할당되는 비트 수는 식 (1)과 같다. *BMBR*과 *RMBR*이 포함하는 영역의 크기를 비교하여 특정 임계치 D 이하일 때까지 *BMBR*을 위한 비트를 할당한다.

$$\frac{Ex(BMBR) - Ex(RMBR)}{Ex(BMBR)} \times 100 \leq D \qquad (1)$$



(a) 2개의 비트 할당



(b) 3개의 비트 할당

그림 9 동적 비트 할당

그림 9는 0~10 사이의 범위를 갖는 2차원 데이타 공간에서 동적 비트 할당을 수행하는 과정을 나타낸 예이다. 그림 9의 (a)는 2개의 비트를 할당하여 *BMBR*을 나타낸 것이고 그림 9의 (b)는 3개의 비트를 할당하여 *BMBR*을 나타낸 것이다. 동적 비트 할당을 위한 임계치 *D*를 40이라 하고 실제 데이타들이 존재하는 *RMBR*이 (2,2,8,8)이라 할 때, 그림 9의 (a)와 같이 2개의 비트를 할당할 경우 *BMBR*은 (00,00,11,11)이 된다. 이러한 상황에서 $Ex(RMBR)=36$이고 $Ex(BMBR)=100$이 되어 식 (1)의 조건을 위배하게 된다. 그러나 그림 9의 (b)와 같이 3개의 비트를 할당할 경우 *BMBR*은 (001,001, 110,110)이 되어 $Ex(BMBR)=56.25$가 된다. 따라서, 식 (1)의 조건을 만족하므로 *BMBR*을 표현하기 위해 3개의 비트를 할당하는 것이 적합하다.

3.4 검색

제안하는 색인 구조는 다차원 데이타의 대표적인 범위 검색과 K-최근접 검색을 지원한다. 범위 검색 (*Q,R*)은 검색의 기준이 되는 다차원 포인터 *Q*와 검색 범위인 길이 *R*에 의해 수행한다. 이와 유사하게 K-최근접 검색 (*Q,K*)에 의해 수행한다. 이때, K는 검색해야 할 데이타의 수를 나타낸다. [21]에서는 최근접 검색을 위해 *MINDIST*()와 *MINMAXDIST*()을 이용하여 최근접 검색을 수행하는 알고리즘을 제시하였다. 그러나 *MINMAXDIST*()는 다차원의 색인 구조에서 탐색할 노드에 대한 선별력이 거의 없으며 많은 계산 시간을 요구한다. [22]에서는 이러한 문제점을 해결하기 위해 *MINDIST*()을 이용한 K-최근접 검색 기법을 제시하였다. 제안하는 색인 구조에서 사용되는 K-최근접 기법은 [22]에서 제시한 알고리즘과 동일하다. 그러나 제안하는 색인 구조는 분할된 영역에 대한 실제 영역을 표현하는 것이 아니라 벡터 근사화를 수행한 영역을 표현하기 때문에 기존에 제안된 R-트리 기반 색인 구조에서 사용되는 *MINDIST*()을 직접 사용할 수 없다. 따라서, 근사화된 영역을 통해 *MINDIST*()을 계산하기 위한 새로운 기법이 필요하다. 그림 10은 제안하는 색인 구조에서 사용하는 *MINDIST*()을 나타낸 것이다. 기존의 R-트리 기반 색인 구조에서는 *MINDIST*()을 계산하기 위해 질의 *Q*에 대해 실제 데이타들이 존재하는 영역 *RMBR*의 거리 *MINDIST*(*RMBR,Q*)을 계산하였다. 그러나 제안하는 색인 구조에서는 질의 *Q*에 대해 *BMBR*의 거리 *MINDIST*(*BMBR,Q*)을 계산한다. 이때, *BMBR*은 벡터 근사화를 수행한 값이기 때문에 검색 조건으로 주어진 *Q*와 직접적인 거리 계산을 할 수 없다. 따라서, 실제 *MINDIST*(*BMBR,Q*)을 수행하기 위해서는 *BMBR*을 실제 값으로 변환하여 *Q*와의 거리를 계산한다.

기존의 범위 검색은 질의와 중간 노드에 존재하는 *RMBR*과 겹침이 발생하는 노드를 검색하지만 제안하는

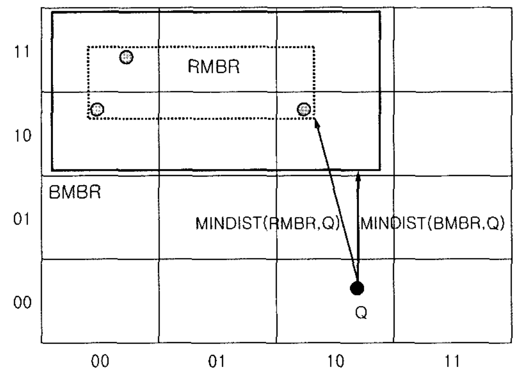


그림 10 MINDIST

색인 구조에서는 *BMBR*과 겹침이 발생하는 노드를 검색한다. 중간 노드에서 질의 *Q*와 *BMBR*에 대한 겹침의 판별은 식 (2)와 같다. 중간 노드에 존재하는 *BMBR*과 질의 *Q*의 거리의 최소 거리 *MINDIST*(*BMBR*, *Q*)가 *R*보다 작거나 같을 경우 겹침이 발생하는 것으로 판별하여 자식 노드를 탐색한다.

$$MINDIST(BMBR, Q) \leq R \qquad (2)$$

범위 검색과 유사하게 K-최근접 검색에서도 질의 *Q*와 *BMBR*의 *MINDIST*()과의 거리를 계산하여 탐색할 자식 노드를 판별한다. K-최근접 검색 과정에서 탐색해야할 노드의 판별은 식 (3)과 같다. 노드에 존재하는 *BMBR*과 질의 *Q*에 대한 *MINDIST*(*BMBR*, *Q*)가 현재까지 검색된 K-최근접 거리 *KNNDIST*(*Q*)보다 작거나 같은 노드만을 탐색한다. 만약 *MINDIST*(*BMBR*, *Q*)가 *KNNDIST*(*Q*)보다 클 경우에는 최근접한 데이타를 포함하지 않기 때문에 자식 노드를 탐색하지 않는다.

$$MINDIST(BMBR, Q) \leq KNNDIST(Q) \qquad (3)$$

4. 삽입 및 분할

4.1 삽입

새로운 데이타의 삽입은 중간 노드에 존재하는 헤더와 실제 자식 노드에 대한 영역을 나타내는 *BMBR*을 통해 새로운 데이타를 삽입하기 적절한 노드를 선택하고 단말 노드에 실제 데이타들을 삽입한다. 중간 노드에 존재하는 *BMBR*은 노드에 존재하는 헤더를 기준으로 자식 노드에 대한 영역을 표현한다. 새로운 데이타를 삽입하기 위한 적절한 노드를 선택하기 위해서는 헤더에 존재하는 분할 정보를 이용하여 분할된 영역을 생성하고 중간 노드의 엔트리들을 이용하여 데이타를 삽입할 자식 노드를 판별한다. 중간 노드에서 데이타를 삽입할 위치의 판별은 공간 분할 방식의 색인 구조와 유사하게 수행된다. 실제 데이타는 단말 노드에 삽입되며 데이타의 삽입으로 영역 정보가 변경될 경우 부모 노드에 *BMBR*을 변경한다. 만약 데이타의 삽입으로 단말 노드에 오버플로우가 발생할 경우에는 *BMBR*을 최소로 생성할 수 있는 위치를 판별하여 공간 분할을 수행한다.

그림 11은 제안하는 색인 구조의 삽입 알고리즘을 나타낸 것이다. 새로운 데이타 *NewE*을 삽입하기 위해서는 먼저 루트 노드에서부터 *NewE*을 삽입할 단말 노드를 검색한다. 제안하는 색인 구조는 공간 분할 방식에 의해 색인을 구성하고 실제 데이타들이 존재하는 영역에 대해 *BMBR*을 표현한다. 따라서, *NewE*는 기존에 공간 분할에 의해 생성된 영역 내에 포함되어야 한다. 따라서, 중간 노드의 헤더에 존재하는 *SP*와 분할 정보 *PI*을 이용하여 기존 공간 분할에 의해 생성된 영역을 판별하고 *NewE*을 삽입할 자식 노드에 접근해야 한다. *FindNode()*는 중간 노드의 헤더 정보를 읽고 헤더에 존재하는 *SP*와 분할 정보 *PI*을 이용하여 *NewE*을 삽입하기 위한 공간 분할 영역 *Region*을 생성한다. *BMBR*은 *Region*보다 항상 작거나 같은 영역을 포함하고 있기 때문에 *Region* 내에 포함되는 *BMBR*을 검사하여 자식 노드에 접근한다. 이러한 과정을 반복하여 단

```
Algorithm Insert(NewE, Root)
{
  PathStack=InitPathStack(); /* 탐색 경로를 기록하기 위한 PathStack을 초기화 */
  Node=ReadNode(Root); /* Root 노드를 읽어 Node에 저장 */
  if(Node==LEAF){ /* Node가 단말 노드일 경우 */
     if(CheckOverFlow(Node, NewE)) /* 오버플로우가 발생할 경우 */
        SplitNode(Node);
     else
        RMBR=InsertLeafEntry(Node, NewE); /* Node에 NewE를 삽입 */
  }
  else{
     Leaf=FindNode(PathStack, Node); /* NewE를 삽입할 단말 노드를 검색 */
     if(CheckOverFlow(Leaf, NewE)) /* 오버플로우가 발생할 경우 */
        SplitNode(Node);
     else{
        RMBR=InsertLeafEntry(Node, NewE); /* Node에 NewE를 삽입 */
        AdjustNode(RMBR, PathStack); /* 노드의 정보를 부모 노드에 반영 */
     }
  }
}
```

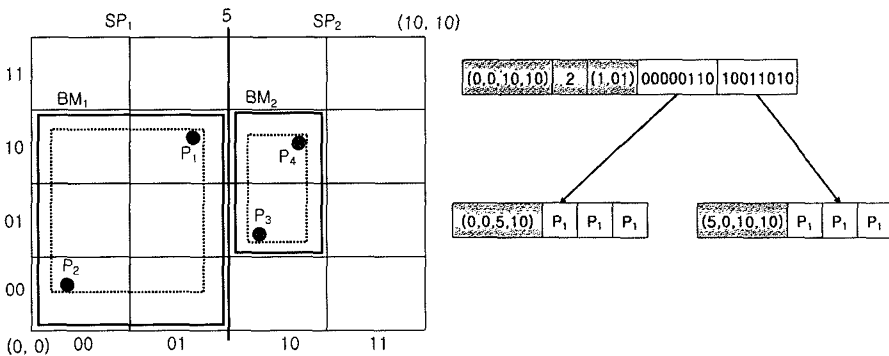
그림 11 삽입 알고리즘

말 노드에 접근하면 *NewE*의 삽입으로 오버플로우가 발생하는지를 검사하여 오버플로우가 발생할 경우 *SplitNode()*을 수행하여 분할을 수행한다. 그러나 *NewE*의 삽입으로 오버플로우가 발생하지 않을 경우에는 *InsertLeafEntry()*를 수행하여 단말 노드에 *NewE*을 삽입하고 *NewE*의 삽입으로 변경된 영역 정보 *RMBR*을 부모 노드에 반영시키기 위해 *AdjustNode()*을 수행한다.

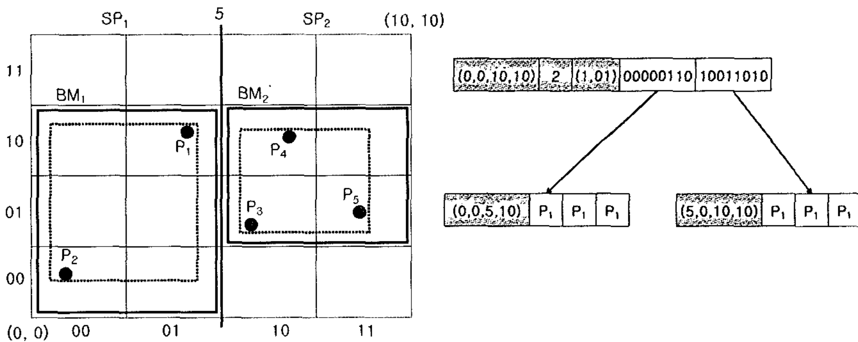
그림 12는 0~10 사이의 범위를 갖는 2차원 데이타 공간에서 노드의 팬아웃이 3일 때 새로운 데이타를 삽입하는 과정을 나타낸 예이다. 그림 12의 (a)와 같이 *SplitPos*에 의해 분할된 두 개의 단말 노드가 존재하고 분할된 노드에 대한 *BMBR*을 표현하기 위해 2개의 비트가 할당되어 있다고 가정하자. 그림 12의 (a)의 헤더에 존재하는 기준 영역 *SP*는 (0,0,10,10)이고 분할 정보 *PI*는 (0,01)이 기록되어 있다. 이러한 상황에서 새로운 데이타 $P_5=(8,3)$이 삽입된다면 먼저 루트 노드에 존재하는 헤더 정보를 읽어 분할 영역을 생성한다. 현재 루트 노드의 헤더에는 *SP*을 기준으로 2개의 비트로 표현된 *PI*를 포함하고 있다. *PI*가 1차원을 기준으로 0.5인 위치에서 분할되었기 때문에 공간 분할에 의해 생성된 두 개의 영역 SP_1과 SP_2는 (0,0,5,10)과 (5,0,10,10)이다. 새로 삽입할 P_5는 SP_2 영역 내에 포함되어 있기 때문에 SP_2 영역 내에 포함되는 *BMBR*을 갖는 엔트리를 검색한다. 루트 노드에서 두 번째 엔트리는 SP_2에 포함되는 *BMBR*을 갖고 있다. 따라서, 두 번째 엔트리를 읽어 자식 노드의 포인터를 획득하고 단말 노드에 데이타를 삽입한다. 그러나 단말 노드의 *BMBR* 내에 P_5가 포함되어 있지 않기 때문에 데이타의 삽입으로 변경된 영역을 부모 노드에 반영한다. 그림 12의 (b)는 P_5을 삽입한 색인 구조를 나타낸 것이다.

4.2 분할

새로운 데이타의 삽입으로 단말 노드에 오버플로우가 발생하면 실제 데이타들이 존재하는 영역 *RMBR*을 최소로 생성할 수 있는 공간 분할 방식을 수행하고 실제 데이타들이 존재하는 영역 *RMBR*과 분할 위치를 부모 노드에 반영한다. 부모 노드는 분할된 영역들에 대한



(a) P_5 삽입 전



(b) P_5 삽입 후

그림 12 2차원 데이타에 대한 삽입 과정

*RMBR*을 *BMBR*로 변환하여 저장하고 분할 위치 정보는 헤더에 존재하는 *PI*에 반영하여 새로운 분할 정보를 생성한다. 기존의 KDB-트리 계열의 공간 분할 방식은 분할된 영역 자체를 부모 노드에 반영하기 때문에 분할된 영역의 합은 항상 동일하다. 그러나 제안하는 색인 구조는 분할된 영역들 사이에 겹침이 없도록 공간 분할을 수행하지만 자식 노드에 대한 영역 정보는 실제 데이타들이 존재하는 영역에 대해서만 근사화를 수행한다. 따라서, 단말 노드에 공간 분할을 수행하기 위해서는 데이타 분할 방식의 특징을 이용하여 실제 데이타들이 존재하는 영역을 *BMBR*로 표현할 때 최소의 비트로 최소의 영역을 생성할 수 있는 위치에 의해 분할을 수행한다. 이에 반해 중간 노드에서는 분할된 영역들에 대한 *BMBR* 자체를 이용하는 것이 아니라 기존에 분할된 정보 *PI*를 기준으로 공간 분할을 수행한다. 그러나 실제 중간 노드의 분할은 공간 분할에 의해 생성된 영역을 *BMBR*로 표현할 때 최소의 영역으로 생성할 수 있는 위치에 의해 수행한다. 중간 노드에서 공간 분할을 수행하는 과정에서 발생할 수 있는 연속 분할이 발생하지 않도록 한다. 그림 13은 *NewE*의 삽입으로 단말 노드에 오버플로우가 발생하여 분할을 수행하는 *LeafSplit*() 알고리즘을 나타낸 것이다. 단말 노드의 분할 위치를 판별하기 위해서는 먼저 *CalculateRange()*와 *CalculateDis()*를 수행하여 각 차원에 대한 영역의 길이와 밀집도를 계산한다. 이를 통해 단말 노드의 분할 위치를 선택한다.

제안하는 색인 구조는 분할된 영역들 사이에 겹침이 없으면서도 실제 데이타들이 존재하는 영역에 대해서만 근사화를 수행하여 *BMBR*을 표현한다. 단말 노드에서 공간 분할을 수행하지만 부모 노드에는 분할된 영역에 대해 실제 데이타들이 존재하는 영역에 대해서만 근사화된 *BMBR*을 표현하기 때문에 실제 데이타들이 존재하는 영역을 최소로 생성할 수 있는 위치를 선택해야 한다. 분할 위치를 선택하기 위해 먼저 실제 데이타들의 분포 특성을 파악하여 분할을 수행할 차원을 판별하고 분할 차원에서 최소의 비트로 최소의 *BMBR*을 표현할 수 있는 위치를 선택한다. 단말 노드에서 분할 차원은 실제 데이타들이 존재하는 영역에 대한 데이타 분포가 가장 넓은 축을 선택한다. 데이타 분포가 넓은 축을 선택할 경우 적은 수의 비트 할당으로 정사각형에 가까운 *BMBR*을 표현할 수 있으며 넓은 영역에 포함되어 있는 사장 공간을 제거시킬 수 있는 효과가 있다. 만약 동일한 분포를 갖는 다수의 차원의 경우 실제 데이타가 존재하는 영역을 기준으로 데이타의 밀집도가 높은 차원을 분할 차원으로 선택한다. 밀집도 높은 차원은 적은 비트 수로도 실제 영역에 근접한 *BMBR*을 표현할 수 있으며 실제 데이타들이 존재하는 영역에 대해서만 *BMBR*을 표현할 수 있어 영역의 크기를 감소시킬 수 있다. 이로 인해 검색 성능을 향상시킬 수 있다. 만약 위의 조건을 모두 만족하는 다수의 차원이 존재할 경우에는 현재까지 분할을 수행하지 않은 차원 중에서 영역의 길이가 가장 큰 차원을 분할 축으로 선택한다. 분할 차원이 선택되면 선택된 차원에서 분할을 수행할 위치를 선택한다. 분할 위치는 오버플로우가 발생한 노드에 존재하는 데이타와 *NewE*을 분할 차원에 대해 정렬하고 노드에 포함되어야 할 최소 데이타 수 *Minfill*을 만족하면서도 실제 데이타들이 존재하는 영역에 대한 *RMBR*을 최소로 할 수 있는 위치를 선택한다. *RMBR*을 최소

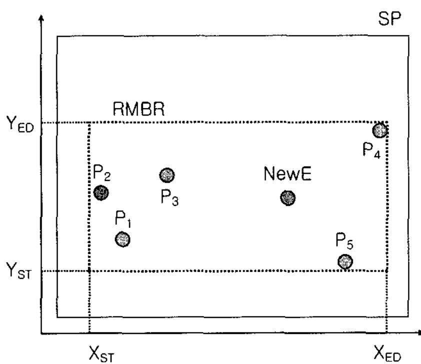
```
Algorithm LeafSplit(Node, NewE, PathStack)
/* 오버플로우가 발생한 단말 노드 Node에 분할을 수행 */
{
   Header=ReadHeader(Node); /* 단말 노드의 헤더를 읽음 */
   LeafE=ReadLeafEntry(Node, NewE); /* 노드에 존재하는 엔트리를 저장 */
   for(i=1;i<=n;i++){ /* 각 차원 i에 대해 */
      Range[i]=CalculateRange(LeafE, i); /* 각 차원 i에 대한 분포 길이를 계산 */
      Dis[i]=CalculateDis(LeafE, i); /* 각 차원 i에 대한 밀집도를 계산 */
   }
   SplitDim=DecideLeafSplitDim(Range, Dis); /* 분할 차원을 선택 */
   SplitPos=DecideLeafSplitPos(Header, SplitDim); /* 헤더를 기준으로 분할 위치를 선택 */
   NewNode=CreateLeafNode(); /* 단말 노드를 생성 */
   NewHeader=WriteLeafHeader(NewNode, Header, SplitPos); /* 새로운 헤더를 생성 */
   UpdateLeafHeader(Node, Header, SplitPos); /* 기존 노드에 헤더를 기록 */
   NewMBR=WriteLeafEntry(LeafE, SplitPos); /* 엔트리를 저장하고 부모 노드에 반영 */
}
```

그림 13 단말 노드 분할 알고리즘

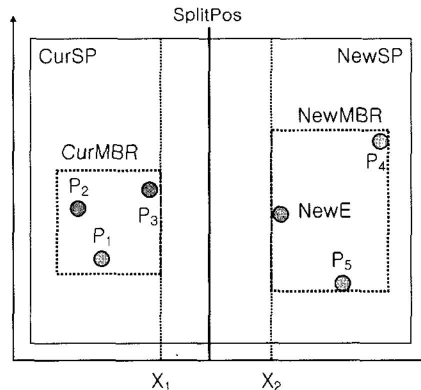
로 생성할 수 있는 분할 위치는 부모 노드에서 *BMBR*을 표현할 때 최소의 영역을 생성할 수 있다.

그림 14는 2차원 데이타 공간에서 단말 노드에 대한 분할을 수행하는 예이다. 그림 14의 (a)와 같이 단말 노드의 헤더에 기존 분할에 의해 생성된 *SP*가 존재하며 실제 데이타들은 *RMBR* 영역 내에 존재한다고 가정하자. 새로운 데이타 *NewE*의 삽입으로 단말 노드에 오버플로우가 발생하면 분할 위치를 선택하기 위해서 실제 데이타가 존재하는 영역에 대한 데이타 분포 특성을 파악하기 위해 X축과 Y축에 대해 사영(Projection)을 수행한다. X축과 Y축에 대해 데이타들이 존재하는 구간은 $[X_{ST}, X_{ED}]$과 $[Y_{ST}, Y_{ED}]$이다. X축은 Y축에 비해 넓은 구간에 데이타들이 존재한다. 따라서, X축을 분할 차원으로 선택한다. X축을 기준으로 분할 위치를 선택하기 위해 먼저 데이타들을 정렬하여 축의 양쪽 끝 부분부터 *Minfill*을 만족할 수 있도록 데이타들을 분할한다. 그림 14의 (b)는 X축에 대해 분할 위치를 선택하는 과정을 나타낸 것이다. 만약 *Minfill*이 2라면 *Minfill*을 만족하도록 분할된 두 개의 그룹에는 $\{P_1, P_2\}\{P_4, P_5\}$가 저장된다. 두 개의 분할 그룹에 *Minfill*을 만족하도록 분할되면 현재 분할에 참여하지 않은 P_3와 *NewE*를 어떤 그룹에 포함시킬 것인지 판별한다. 단말 노드에 최소 영역을 생성할 수 있도록 분할된 영역은 $[X_1, X_2]$ 구간 내에서 분할을 수행하는 것이 적합하다. $[X_1, X_2]$ 구간에서 실제 공간 분할을 수행할 위치 *SplitPos*는 부모 노드의 헤더에 기록된 분할 정보를 이용하여 기존에 할당된 비트로 표현이 가능한 위치를 *SplitPos*로 선택한다. 만약 부모 노드에 할당된 비트로 *SplitPos*을 표현할 수 없다면 가장 최소의 비트로 표현이 가능한 위치를 분할 위치로 선택한다. 그림 14의 (b)는 단말 노드에 대한 분할한 후를 나타낸 것이다. *SplitPos*에 의해 두 개의 영역으로 분할되면 분할을 수행하기 이전 *SF*에 대해 *SplitPos*를 기준으로 공간 분할 영역에 대한 *CurSP*와 *NewSP*를 계산하여 헤더에 저장한다. 노드에 헤더를 기록하면 분할된 영역들에 대한 *BMBR*을 부모 노드에 반영하기 위해 분할된 각각의 영역에서 실제 데이타들이 존재하는 *RMBR*을 계산한다. 기존에 존재하는 노드에 대한 *RMBR*은 *CurMBR*에 저장하고 새로 생성된 노드에 대한 *RMBR*은 *NewMBR*에 저장하여 부모 노드에 반영한다. 부모 노드에서는 부모 노드에 존재하는 헤더 정보를 이용하여 *CurMBR*과 *NewMBR*을 *BMBR*로 변환하여 저장한다.

오버플로우가 발생한 단말 노드의 분할 과정이 완료되면 부모 노드에 분할된 정보를 반영하게 된다. 단말 노드의 분할 정보를 반영하는 과정에 부모 노드에 여유 공간이 없을 경우에는 오버플로우가 발생하여 중간 노드를 분할하게 된다. 공간 분할 방식의 색인 구조에서 중간 노드의 분할은 자식 노드들에 대한 계속적인 분할을 수행시키는 연속 분할을 발생시킬 수 있다. [23]에서는 KDB-트리 계열의 색인 구조에서 발생할 수 있는 연속 분할의 문제점을 해결하기 위한 방법을 제시하였다. 그러나 기존의 공간 분할 방식들은 실제 데이타들이 존재하는 영역의 특성을 고려하지 않고 분할을 수행하기 때문에 분할된 영역 내에 많은 사장 공간을 포함하고 있다. 제안하는 색인 구조는 공간 분할을 수행하지만 실제 데이타들이 존재하는 영역에 대해서만 *BMBR*을 표현하기 때문에 기존의 공간 분할 방식을 직접 사용하는 것은 효과적이지 못하다. 제안하는 색인 구조의 중간 노드의 분할은 자식 노드들에 대한 연속 분할을 발생하지 않으면서도 분할된 영역을 최소의 *BMBR*로 표현할



(a) 분할 차원 선택



(b) 분할 위치 선택

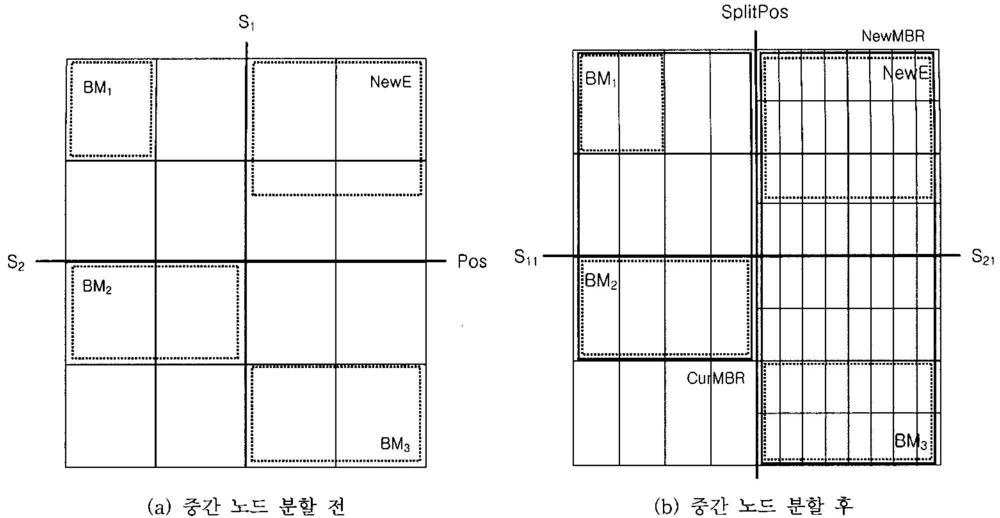
그림 14 단말 노드의 분할 위치 선택

수 있는 분할 기법을 사용한다. 그림 15는 중간 노드에 대한 분할을 수행하는 *InternalSplit*() 알고리즘을 나타낸 것이다. 제안하는 색인 구조에서는 연속 분할이 발생하지 않은 위치를 선택하기 위해 헤더에 존재하는 기존 분할 위치와 자식 노드에 대한 새로운 분할 위치 *Pos*을 이용하여 연속 분할이 발생하지 않는 위치를 선택한다. 연속 분할이 발생하지 않는 차원은 *NonOverlapos()*를 수행하여 다른 차원의 분할 위치와 겹침이 발생하지 않는 차원으로 선택한다. [23]에서 언급한 것과 같이 노드에 대해 분할을 수행한 첫 번째 분할 위치는 항상 연속 분할을 수행하지 않는 위치이다. 따라서, 제안하는 색인 구조의 분할 기법은 항상 연속 분할이 발생하지 않는 위치가 하나 이상 존재한다.

그림 16은 2차원 데이타 공간에서 중간 노드에 대한 분할을 수행하는 예이다. 그림 16의 (a)와 같이 중간 노드에 BM_1, BM_2, BM_3가 존재하고 있을 때 BM_3가 *Pos*에 의해 분할된 새로운 BM_3와 *NewE*가 생성되었다고 가정하자. 자식 노드의 분할로 인해 새로 생성된 *NewE*을 중간 노드에 삽입을 하려는 과정에서 오버플로우가 발생하면 헤더에 존재하는 분할 정보 S_1, S_2 그리고 새로운 분할 위치 *Pos*을 이용하여 하위 노드에 대한 연속 분할을 발생하지 않으면서도 분할된 영역들에 대해 최소의 영역을 생성할 수 있는 분할 위치 *SplitPos*을 계산한다. 그림 16의 (a)에 존재하는 분할 정보를 이용하면 X축에 대해서는 S_1 그리고 Y축에 대해서는 S_2에 의해 분할을 수행할 경우 자식 노드에 대한 연속 분할을 발생시키지 않는다. 그러나 Y축으로 분할하는 영역보다는 X축으로 분할을 수행하는 것이 분할된 영역에 대한 *RMBR*을 계산할 경우 최소의 영역을 생성할 수 있다. 따라서, 중간 노드에 대한 분할 위치는 S_1을 기준으로 분할하는 것이 효과적이다. 그림 16의 (b)는 S_1을 중간 노드의 분할 위치 *SplitPos*로 선택하여 분할한 것이다. *SplitPos*에 분할을 수행하면 각 분할된 영역에 존재하는 엔트리들을 표현하기 위해 필요한 비트 수를 계산하여 각 노드에 존재하는 엔트리들을 *BMBR*로 계산하여 노드에 저장한다. 그림 16의 (b)에서 분할된 영역의 왼쪽 노드는 2개의 비트를 할당하여 *BMBR*을 계산하고 오른쪽 노드는 3개의 비트를 할당하여 *BMBR*을 표현한 것이다. 분할 과정이 완료되면 분할된 영역에 대한 정보를 부모 노드에 반영하기 위해 각 분할된 영역에 대해 실제적으로 데이타들이 존재하는 *CurMBR*과 *NewMBR*을 계산한다. 현재 노드의 부모 노드에 분할 정보 *SplitPos*과 분할된 각 노드에 대한 *RMBR*와 분할 위치 *SplitPos*을 부모 노드에 반영한다.

```
Algorithm IntenalSplit(Node, NewE, Pos, PathStack)
/* 자식 노드의 분할로 인해 중간 노드 Node에서 오버플로우가 발생 */
{
  Header=ReadHeader(Node); /* 단말 노드의 헤더를 읽음 */
  TempPI=ReadPI(Header, Pos); /* 헤더의 PI와 새로운 분할 위치를 읽음 */
  for(i=1;i<=m;i++){ /* TempP에 저장된 분할 위치에 대해 */
    if(NonoverlapPos(TempPI[i])){ /* 다른 분할 위치와 겹침이 발생하지 않음 */
      TempPos[j]=TempPI[i]; /* 연속 분할이 발생하지 않는 위치 */
      j++;
    }
  TempE=ReadInterEntry(Node, NewE); /* 중간 노드에 존재하는 엔트리를 읽음 */
  InternalE=CulateRealMBR(Header, TempE); /* 헤더를 기준으로 실제 영역을 계산 */
  for(i=1;i<=j;i++){ /* 연속 분할이 발생하지 않은 차원에 대해 */
    RMBR[i]=CalculateMBR(InternalE, i); /* 각 차원으로 분할할 때 MBR을 계산 */
    Num[i]=CalculateBit(MBR[i]); /* BMBR을 표현하기 위한 비트를 수를 계산 */
  }
  SplitPos=DecideInterSplitPos(RMBR[i], Num[i]); /* 분할 차원을 선택 */
  NewNode=CreateInterNode(); /* 중간 노드를 생성 */
  NewHeader=WriteInterHeader(NewNode, Header, SplitPos); /* 새로운 헤더를 생성 */
  UpdateInterHeader(Node, Header, SplitPos); /* 기존 노드에 헤더를 기록 */
  BMBR=TransformBMBR(RMBR, SplitPos, Num); /* BMBR을 변경 */
  NewMBR=WriteInterEntry(BMBR, SplitPos); /* BMBR을 저장하고 부모 노드에 반영 */
}
```

그림 15 중간 노드 분할 알고리즘



(a) 중간 노드 분할 전 (b) 중간 노드 분할 후

그림 16 중간 노드 분할

5. 실험 및 성능 평가

5.1 실험 환경

실험을 위해 펜티엄-Ⅳ 1.8GHz CPU와 256MB 메모리를 갖는 윈도우즈 2000 서버 운영체제에서 C언어를 사용하여 구현한다. 실험을 위해 임의로 생성한 10만개의 랜덤 데이타와 코넬 이미지 집합에서 추출한 실제 데이타를 사용한다. 랜덤 데이타는 실험을 위해 16에서 32 차원의 데이타를 임의로 생성한 것으로 각 차원의 값은 0에서 1 사이의 값을 갖도록 생성한다. 실제 데이타는 JPG 형태로 저장되어 있는 코넬 이미지 집합에서 추출한 16에서 32 차원의 특징 값으로 각 차원은 0에서 1 사이의 값을 갖는다. 실제 데이타에 대한 특징 값은 코넬 이미지에서 추출한 칼라 히스토그램 값을 사용한다. 랜덤 데이타와 실제 데이타는 그림 17에서 보는 것과 같이 각 라인은 이미지 하나에 해당하는 값으로 이미지의 식별자와 0에서 1 사이의 값을 갖는 특징 값으로 이루어져 있다.

성능 평가는 삽입 성능과 검색 성능 관점에서 수행하였으며 삽입 성능은 16에서 32 차원의 랜덤 데이타 집합과 실제 데이타 집합을 삽입한 시간을 측정한다. 삽입 성능과 검색 성능을 비교하기 위해 식 (1)을 이용하여 동적 비트를 할당하기 위한 임계치 D는 30%로 고정하여 실험을 수행한다. 검색 성능은 데이타를 삽입하여 구성한 색인 구조에 범위 검색과 K-최근접 검색을 수행한 시간과 페이지 접근 횟수를 비교한다. 제안하는 색인 구조는 동적 비트 할당을 위해 식 (1)을 이용하여 동적 비트 할당을 위한 임계치 D을 선택한다. 그러나 이러한 임계치 D을 선택하기 위한 최적의 방법이 존재하지 않기 때문에 동적 비트 할당을 위한 임계치를 20%에서 40%까지 변경하면서 삽입 시간과 검색 시간을 비교한다. 성능 평가를 위한 파라미터는 표 3과 같다. 성능 평가를 위해 제안하는 색인 구조를 ABA(Adaptive Bit Allocation)-트리라 한다.

5.2 성능 평가

제안하는 색인 구조에 대한 삽입 성능을 평가하기 위해 10만개의 랜덤 데이타와 실제 데이타에 대해 색인을 구성하는 시간을 CS-트리와 비교 분석한다. 그림 18은 색인을 구성하는 시간을 나타낸 것이다. 제안하는 색인

```
...
...
22 0.004897 0.000000 0.001979 0.006355 ................ 0.130010 0.013658 0.121783 0.016677
23 0.220521 0.136146 0.185625 0.149271 ................ 0.000014 0.013126 0.000013 0.009585
24 0.000023 0.071785 0.000019 0.049807 ................ 0.008333 0.015626 0.003333 0.046980
...
...
```

그림 17 실험 데이타 집합

표 3 성능 평가를 위한 파라미터

특성 / 파라미터	특성
랜덤 데이타 집합	100,000개
실제 데이타 집합	68,040개
페이지 크기	4 kbytes
동적 비트 할당 임계치	30%
차원 수	16, 24, 32 차원
특징 벡터의 범위	0 ~ 1 사이의 실수 값

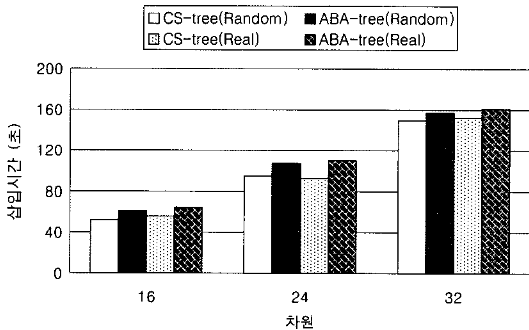


그림 18 삽입 시간

구조는 기존의 CS-트리 보다 데이타를 삽입하는 시간이 약간 더 소요되는 것을 알 수 있다. CS-트리는 전체 데이타 공간을 고정된 크기의 셀로 분할하여 근사화를 수행하지만 제안하는 색인 구조는 데이타의 분포 특성에 따라 동적 비트를 할당하기 때문에 이를 처리하기 위한 시간이 더 소요된다. 또한 색인 구조의 계층에 따라 상대적인 영역을 생성하고 이에 대한 근사화를 수행하기 때문에 삽입 성능이 저하된다.

검색 성능을 평가하기 위해 다차원 데이타의 대표적인 검색 방법인 범위 검색과 K-최근접 검색을 50번 수행한 평균값을 사용한다. 각각의 검색 방법은 페이지 접근 횟수와 검색 시간 관점에서 수행한다. 그림 19와 20은 범위 검색에 따른 페이지 접근 횟수와 검색 시간을 나타낸 것이다. 또한 그림 21과 22는 K-최근접 검색을 수행한 내용이다. 제안하는 색인 구조의 검색 결과 기존에 제안된 CS-트리보다 우수한 성능을 나타낸다. 제안하는 색인 구조는 분할된 영역들 사이에 겹침 영역이 없으며 실제 데이타들이 존재하는 영역에 대해서 근사화된 영역 *BMBR*을 표현하기 때문에 기존의 CS-트리보다 검색 성능이 향상된다. 또한, 제안하는 색인 구조는 자식 노드에 대한 영역을 표현하기 위해 부모 노드를 기준으로 상대적인 영역을 표현할 뿐만 아니라 데이타들의 분포 특성에 따라 동적인 비트를 할당하기 때문에 분할된 영역들 사이에 선별력을 증가시켜 검색 성능을 향상시킨다.

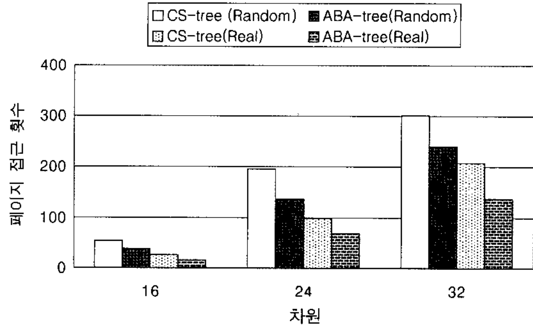


그림 19 범위 검색에 따른 페이지 접근 횟수

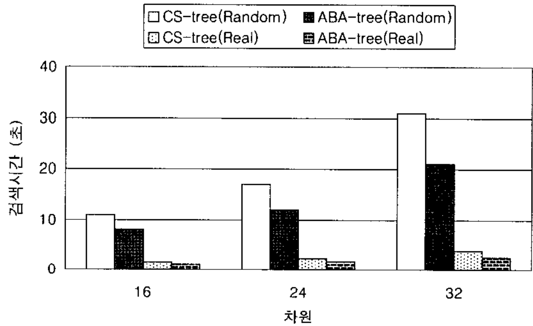


그림 20 범위 검색에 따른 검색 시간

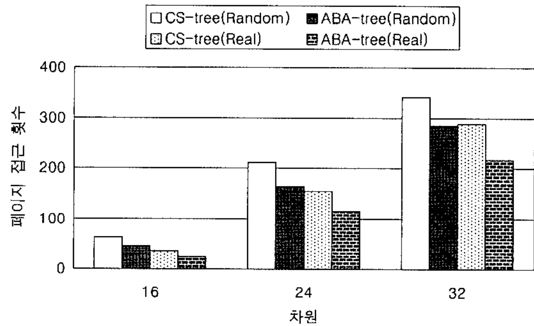


그림 21 K-최근접 검색에 따른 페이지 접근 횟수

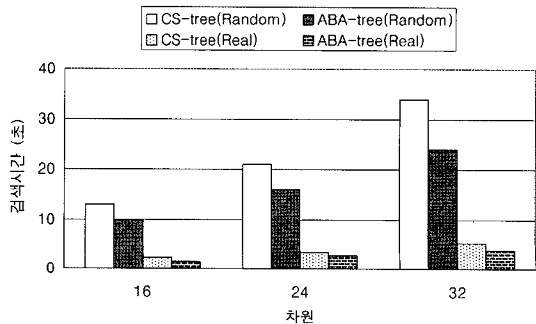


그림 22 K-최근접 검색에 따른 검색 시간

제안하는 색인 구조는 동적 비트 할당을 통해 데이타들이 존재하는 실제 영역에 대한 근사화를 수행한다. 자식 노드에 대한 *BMBR*을 표현하기 위한 비트 수를 결정하기 위해 식 (1)를 이용하여 *RMBR*과 *BMBR*의 유사도가 임계치 D 이하가 될 때까지 *BMBR*을 위한 비트를 할당한다. 그림 23과 24는 동적 비트 할당을 위한 임계치 D에 따른 삽입 시간과 검색 성능을 비교한 것이다. 동적 비트 할당을 위한 임계치가 증가할수록 삽입 시간은 향상되지만 검색 성능은 저하되는 것을 알 수 있다. 임계치가 클수록 비트 할당을 위한 계산 시간을 빠르게 하여 삽입 성능은 증가되지만 근사화된 영역과 실제 영역과 많은 차이를 보이기 때문에 중간 노드에서 근사화된 영역들 사이에 선별력이 저하된다. 따라서, 검색 과정에서 탐색해야할 노드 수를 증가시켜 검색 성능을 저하시킨다.

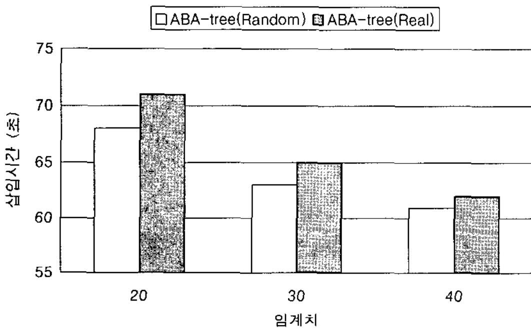


그림 23 동적 비트 할당에 따른 삽입 시간

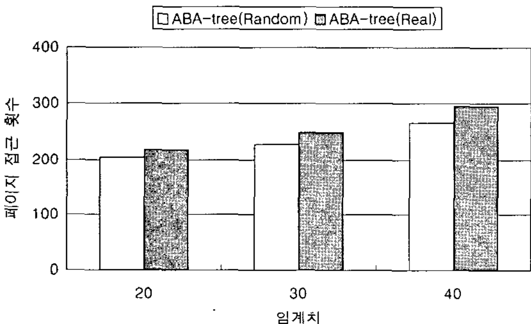


그림 24 동적 비트 할당에 따른 K-최근접 검색

6. 결론 및 향후 연구

본 논문에서는 다차원 색인 구조의 문제점을 해결하기 공간 분할 방식과 벡터 근사화 기법을 이용한 새로운 색인 구조를 제안하였다. 제안하는 색인 구조는 색인 구조의 계층적 구조에 의해 자식 노드에 대한 영역을 상대적으로 표현하고 분할된 각 영역은 데이타의 분포 특성에 따라 동적인 비트를 할당한다. 또한 분할된 영역들 사이의 겹침 영역을 제거하기 위해 공간 분할 방식을 이용하여 분할을 수행하고 실제 데이타들이 존재하는 영역에 대해서만 근사화를 수행한 영역을 표현한다. 제안하는 색인 구조는 일반적인 R-트리 기반 색인 구조와 유사하게 데이타의 삽입 순서에 따라 색인 구조의 모양이 변경될 수 있으며 분할된 영역이 길이가 다를 경우 길이가 짧은 축은 다른 축에 비해 조밀하게 영역이 표현된다. 성능 평가 결과 제안하는 색인 구조는 기존의 제안된 CS-트리에 비해 삽입 성능은 약간 저하되지만 검색 성능은 40% 이상 향상됨을 보였다. 향후 연구 방향으로 제안하는 색인 구조에 적합한 벌크 연산에 대한 연구를 수행할 예정이다.

참 고 문 헌

[1] S. Berchtold, D. A. Keim and H. P. Kriegel, "The X-Tree : An Index Structure for High-Dimensional Data," Proc. the 22nd International Conference on Very Large Data Bases, pp.28-39, 1996.
[2] V. Gaede and O. Gunther, "Multidimensional Access Methods," ACM Computer Survey, Vol.30, No.2, pp.170-231, 1998.
[3] K. Chakrabarti and S. Mehrotra, "The Hybrid Tree : An Index Structure for High Dimensional Feature Spaces," Proc. the 15th International Conference on Data Engineering, pp.440-447, 1999.
[4] J. T. Robinson, "The K-D-B-Tree : A Search Structure For Large Multidimensional Dynamic Indexes," Proc. the ACM SIGMOD International Conference on Management of Data, pp.10-18, 1981.
[5] A. Henrich, H. W. Six and P. Widmayer, "The LSD tree : Spatial Access to Multidimensional Point and Nonpoint Objects," Proc. the Fifteenth International Conference on Very Large Data Bases, pp.45-53, 1989.
[6] A. Henrich, "The LSDh-Tree : An Access Structure for Feature Vectors," Proc. the Fourteenth International Conference on Data Engineering, pp.362-369, 1998.
[7] O. Procopiuc, P. K. Agarwal, L. Arge and J. S. Vitter, "Bkd-Tree : A Dynamic Scalable kd-Tree," Proc. 8th International Symposium on Spatial and Temporal Databases, pp.46-65, 2003.
[8] B. Yu, R. Orlandic, T. Bailey and J. Somavaram, "KDBKD-Tree : A Compact KDB-Tree Structure for Indexing Multidimensional Data," Proc. International Conference on Information Technology : Coding and Computing, pp.676-680, 2003.
[9] A. Guttman, "R-trees : A Dynamic Index Structure for Spatial Searching," Proc. ACM SIGMOD Conference, pp.47-57, 1984.

[10] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-Tree : An Efficient and Robust Access Method for Points and Rectangles," Proc. ACM SIGMOD International Conference on Management of Data, pp.322-331, 1990.
[11] K. I. Lin, H. V. Jagadish and C. Faloutsos, "The TV-tree : An Index Structure for High-Dimensional Data," VLDB Journal, Vol.3, No.4, pp.517-542, 1994.
[12] R. Weber, H. J. Schek and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," Proc. 24rd International Conference on Very Large Data Bases, pp.194-205, 1998.
[13] J. An, Y. P. Chen, Q. Xu, and X. Zhou, "A New Indexing Method for High Dimensional Dataset," Proc. International Conference on Database Systems for Advanced Applications, pp.385-397, 2005.
[14] E. Tuncel, H. Ferhatosmanoglu, and K. Rose, "VQ-index: an index structure for similarity searching in multimedia databases," Pro. the 10th ACM International Conference on Multimedia, pp.543-552, 2002.
[15] J. S. Lee, J. S. Yoo, S. H. Lee, and M. J. Kim, "An Efficient Content-Based High-Dimensional Index Structure for Image Data," ETRI Journal, Vol.22, No.2, pp.32-43, 2002.
[16] J. Goldstein, R. Ramakrishnan and U. Shaft, "Compressing Relations and Indexes," Proc. the Fourteenth International Conference on Data Engineering, pp.370-379, 1998.
[17] K. T. Song, H. J. Nam and J. W. Chang, "A Cell-based Index Structure for Similarity Search in High-dimensional Feature Spaces," Proc. the 2001 ACM Symposium on Applied Computing, pp.264-268, 2001.
[18] Y. Sakurai, M. Yoshikawa, S. Uemura and H. Kojima, "The A-tree : An Index Structure for High-Dimensional Spaces Using Relative Approximation," Proc. 26th International Conference on Very Large Data Bases, pp.516-526, 2000.
[19] Y. Sakurai, M. Yoshikawa, S. Uemura and H. Kojima, "Spatial indexing of High-dimensional Data based on Relative Approximation," VLDB Journal, Vol.11, No.2, pp.93-108, 2002.
[20] G. H. Cha and C. W. Chung, "The GC-Tree : A High-Dimensional Index Structure for Similarity Search in Image Databases IEEE Transactions on Multimedia, Vol.4, No.2, pp.235-247, 2002.
[21] N. Roussopoulos, S. Kelley and F. Vincent, "Nearest Neighbor Queries," Proc. ACM SIGMOD Conference, pp.71-79, 1995.
[22] J. Kuan, and P. Lewis, "Fast k nearest neighbour search for R-tree family," Proc. International Conference on Information, Communications and Signal Processing, pp.924-928, 1997.
[23] R. Orlandic and B. Yu, "Implementing KDB-Trees to Support High-Dimensional Data," Proc. International Database Engineering and Applications Symposium, pp.58-67, 2001.

복 경 수

1998년 2월 충북대학교 수학과(이학사) 2000년 2월 충북대학교 정보통신공학과(공학석사). 2005년 2월 충북대학교 정보통신공학과(공학박사). 2005년 3월~현재 한국과학기술원 전산학과 Postdoc. 관심분야는 내용 기반 검색, 자료 저장 시스템, 멀티미디어 데이타베이스, 이동 객체 데이타베이스, 고차원 색인 구조, 시공간 색인구조 등

김 은 재

2004년 2월 충북대학교 정보통신공학과(공학사). 2004년 3월~현재 충북대학교 정보통신공학과 석사과정. 관심분야는 XML 질의 처리, 자료 저장 시스템, 데이타베이스 시스템, 이동 객체 데이타베이스 등

유 재 수

정보과학회논문지 : 데이타베이스
제 32 권 제 3 호 참조