노드 이용률과 검색 속도 개선을 위한 비트 벡터 기반 공간 분할 색인 기법

(Bit-Vector-Based Space Partitioning Indexing Scheme for Improving Node Utilization and Information Retrieval)

여 명 호 † 성 동 욱 †
(Myung Ho Yeo) (Dong Ook Seong)

유 재 수 ††
(Jae Soo Yoo)

요 약 KDB-트리는 다차원 데이터를 검색하기 위한 전통적인 색인 기법이다. 많은 연구에서 낮은 저장 공간 사용과 검색 성능이 KDB-트리군의 두 병목현상이라고 언급되고 있다. 데이터 삽입 순서와 데이터의 편향으로 인한 불필요한 공간 분할이 그 원인이다. 본 논문에서는 편향 데이터를 효율적으로 처리하고, 검색 성능을 향상시키기 위한 새로운 색인 구조인 KDB_{CS}^{+}-트리를 제안한다. KDB_{CS}^{+}-트리는 분할 정보를 비트벡터로 표현하는 압축 기법과 노드의 그룹화를 통한 포인터 제거 기법을 활용하여 중간 노드의 팬-아웃을 증가시키고, 중간 노드의 엔트리를 계층적으로 표현함으로써 중간 노드의 사용율을 높인다.

키워드 : 공간 분할, 색인 구조, KDB-트리, 정보 검색

· 이 논문은 2009년 교육과학기술부로부터 지원(지역거점연구단육성사업/충북BIT연구중심대학육성사업단)과 교육과학기술부와 한국산업기술재단의 지역혁신인력양성사업으로 수행된 연구결과임
· 이 논문은 제36회 추계학술발표회에서 '효율적인 노드 사용과 빠른 정보 검색을 위한 비트벡터 기반 공간 분할 색인 기법'의 제목으로 발표된 논문을 확장한 것임

† 학생회원 : 충북대학교 정보통신공학과
mhyeo@netdb.cbnu.ac.kr
seong.do@gmail.com
†† 종신회원 : 충북대학교 정보통신공학과 교수
yjs@chungbuk.ac.kr
(Corresponding author임)

논문접수 : 2009년 12월 28일
심사완료 : 2010년 3월 29일

정보과학회논문지: 컴퓨팅의 실제 및 레터 제16권 제7호(2010.7)

Abstract The KDB-tree is a traditional indexing scheme for retrieving multidimensional data. Much research for KDB-tree family frequently addresses the low storage utilization and insufficient retrieval performance as their two bottlenecks. The bottlenecks occur due to a number of unnecessary splits caused by data insertion orders and data skewness. In this paper, we propose a novel index structure, called as KDB_{CS}^{+}-tree, to process skewed data efficiently and improve the retrieval performance. The KDB_{CS}^{+}-tree increases the number of fan-outs by exploiting bit-vectors for representing splitting information and pointer elimination. It also improves the storage utilization by representing entries as a hierarchical structure in each internal node.

Key words : space partitioning, index structure, KDB-tree, information retrieval

1. 서 론

지리정보시스템(GIS), 교통/환경 시스템, CAD/CAM과 같은 대부분의 데이터베이스 응용 분야에서 데이터는 다중 속성으로 표현된다. 이러한 데이터를 빠르게 접근하기 위한 목적으로 데이터의 색인을 구성한다. 데이터 색인 기법은 크게 공간 분할 색인 기법과 데이터 분할 색인 기법으로 나눌 수 있다[1]. 대표적인 데이터 분할 색인 기법인 R-트리는 검색 시 데이터가 존재하는 공간만 검색이 이루어지므로 공간 분할 색인 기법에 비해 검색 성능이 우수하지만, 삽입과 갱신에 있어서 빈번한 MBR의 갱신이 발생하므로 공간 분할 색인 기법에 비해 성능이 좋지 못하다[2]. 대표적인 공간 분할 색인 기법인 KDB-트리[3]는 삽입과 갱신이 비교적 간단하지만, 검색 시에 데이터가 존재하지 않는 공간도 검색이 이루어질 가능성이 있기 때문에 검색 부분에 있어서 데이터 분할 색인 기법에 비해 성능이 나쁘다는 단점이 있다[4]. 또한, 분포가 고르지 않은 편향 데이터(Skewed Data)를 효과적으로 색인하지 못하는 문제로 인해 R-트리에 비해 널리 연구되지 못했다. 하지만, 구현이 비교적 간단하고, 넓은 분포를 보이는 데이터를 다루는데 있어서 R-트리보다 더 안정적이고, 강인하다는 점에서 여전히 많은 응용 분야에서 사용되고 있다[5,6].

본 논문에서는 단말 노드와 중간 노드의 노드 사용율을 함께 향상시킴으로써 공간 분할 색인 기법의 편향 데이터 처리 문제를 해결하고, 데이터 탐색 성능을 향상시키는 KDB_{CS}^{+}-트리를 제안한다. KDB_{CS}^{+}-트리는 캐시를 고려한 다차원 색인 구조인 KDB_{CS}-트리를 확장한 형태로 노드 사용율을 향상시키기 위한 접근 방법으로 세가지 방법을 함께 사용한다. 첫 번째, 분할축 정보를 압축하기 위해서 미리 결정된 분할 정보를 비트벡터로 유지하며, 분할이 필요한 시점에서 분할축 집합으로부터

분할이 유효한 분할축을 선택한다. 이때, 노드는 해당 분할에 참여한 분할축이 분할축 집합의 몇 번째인지를 비트벡터에 기록하게 된다. 기존에 R-트리나 격자 파일에서 MBR을 비트맵이나 비트벡터를 표현하는 기법이 제안된 바 있으나 KDB_{CS}^{+}-트리는 MBR 대신 분할축 정보를 비트로 표현한다는 차이점을 가지고 있다. 이를 통해 KDB-트리가 차원에 독립적인 팬-아웃을 제공하는 본래 장점을 그대로 유지시킨다. 두 번째, 포인터 정보를 제거하기 위해서 하위 노드를 그룹화하고, 하위 노드 그룹에 대한 하나의 포인터와 개별 하위 노드의 접근을 위한 오프셋 정보를 유지한다. 세 번째, 단말 노드의 overflow 또는 underflow 발생시, 새로운 분할을 수행하기 이전에 분할축 집합으로부터 분할이 유효한 분할축이 재선정함으로써 노드의 사용율를 높인다.

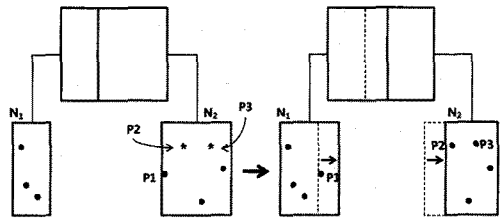
본 논문의 구성은 다음과 같다. 제2장에서는 기존에 제안된 KDB-트리의 특징을 기술한다. 제3장에서는 제안하는 색인구조의 특징과 분할 및 포인터 제거기법을 기술한다. 제4장에서는 성능평가와 분석을 통해 제안하는 색인 구조의 우수성을 보이고, 제5장에서 결론과 향후 연구방향을 제시한다.

2. 관련연구

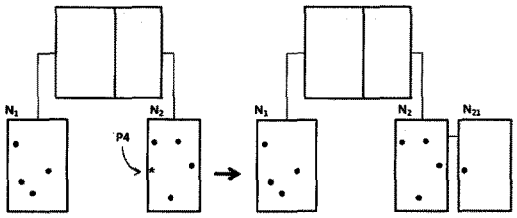
KDB_{HD}-트리[7]는 FD(First Division)-분할 정책과 MBR Descriptor로부터 중복 정보를 제거한 KDB-트리의 변형이다. KDB_{KD}-트리[8]는 중간 노드의 색인 엔트리 구조를 변경함으로써 KDB_{HD}-트리에서 여전히 남아있는 중복된 정보를 제거하는 방법을 제안했다. KDB_{CS}-트리는 KDB_{KD}-트리를 주기억장치에서 사용 가능하도록 분할축 정보를 압축하고, 동시에 물리적으로 연속된 공간에 하위 노드들을 할당하여 포인터 정보를 제거함으로써 팬-아웃을 크게 증가시켰다. KDB_{CPD}-트리는 편향 데이터 문제를 효과적으로 해결하기 위해서 분할축 재조정 기법과 복제 노드 기법을 제안하였다. 그림은 KDB_{CPD}-트리의 분할축 재조정 및 노드 복제의 예를 나타낸다. 그림 1(a)처럼 P2와 P3가 N_2에 삽입되는 경우, N_2의 overflow와 함께 N_2에 데이터가 집중되는 편향이 발생하게 된다. 기존 KDB-트리는 공간을 추가로 분할하지만, KDB_{CPD}-트리의 경우, 분할축을 재조정하여 overflow를 해결한다. 이를 통해 N_1과 N_2의 편향도 함께 해결된다. 만약 그림 1(b)와 같이 재조정이 불가피한 경우, N_2의 복제 노드 N_{21}를 연결 리스트형태로 연결하여 N_2의 데이터를 분산해서 저장한다. 복제 노드의 수가 일정 수 이상이 되면, 공간 분할을 조정하게 된다.

3. 제안하는 공간 분할 색인 기법

3.1 노드 구조



(a) 분할축 재조정



(b) 복제 노드

그림 1 KDBCPD-트리

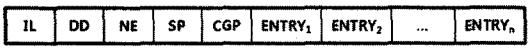


그림 2 KDB_{CS}^{+}-트리의 중간 노드 구조

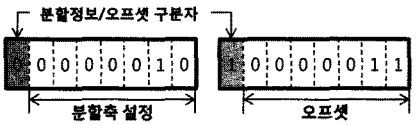
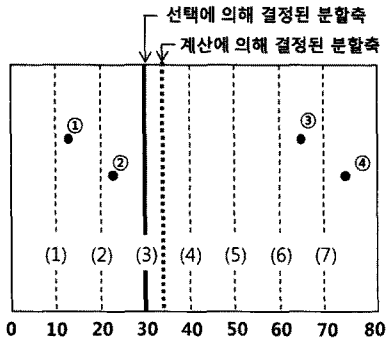


그림 3 KDB_{CS}^{+}-트리의 엔트리(Entry) 구조

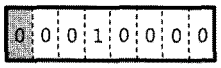
그림 2와 그림 3은 각각 제안하는 KDB_{CS}^{+}-트리의 중간 노드와 엔트리의 구조를 나타낸다. 중간 노드는 중간 노드와 단말노드를 구별하기 위한 플래그 IL(internal/leaf node identifier), 분할차원을 나타내는 플래그 DD(division dimension), 엔트리의 수 NE(the number of entry), 공간정보 SP(space), 자식 노드 그룹의 주소 CGP(Child Group Pointer), 공간 분할정보, 자식 노드에 접근하기 위한 오프셋 정보를 담고 있는 엔트리(Entry) 집합으로 구성된다. 각 엔트리 의 첫 번째 비트는 분할축과 오프셋 정보를 구별하기 위한 구분자로 사용된다. 분할축에 저장하는 방법과 오프셋을 통해 자식 노드에 접근하는 방법은 각각 3.2절과 3.3절에서 자세히 기술한다.

3.2 공간 분할 및 재조정

그림 4는 단말 노드가 가질 수 있는 최대의 데이터 수가 3인 KDB-트리에서 KDB-트리와 제안하는 기법의 분할축을 선정하는 예를 나타낸다. KDB-트리의 경우, overflow가 발생하게 되면, 축을 중심으로 데이터를



(a) 공간 분할의 예



(b) 분할정보를 저장하는 비트벡터

그림 4 공간 분할의 예

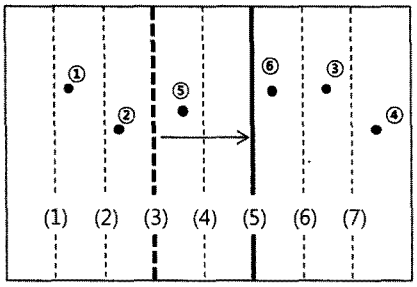


그림 5 분할축의 재조정

정렬하고, 데이터의 수를 1/2로 구분할 수 있는 분할축을 계산한다. 예를 들면, 그림과 같이 R1의 공간이 분할하기 위해서 x=33인 분할축을 결정하게 되고, 분할 정보를 중간 노드에 저장한다. 분할이 계속 발생하게 되면, 분할 정보의 크기는 선형적으로 증가한다. 제안하는 분할 기법은 분할 정보의 크기를 줄이기 위해서 공간을 미리 n등분하고, 분할축 집합으로부터 실제 데이터를 분할할 수 있는 축을 선택하게 된다. 이 과정에서 다수의 분할 정보는 하나의 비트벡터로 표현 가능하다. 0~80의 범위를 가진 x 축을 미리 양자화 레벨(=8로 가정)만큼 등분하고, overflow가 발생하여 분할이 필요한 경우, 분할축 집합 = *{10, 20, 30, 40, 50, 60, 70, 80}* 으로부터 분할이 가능한 축을 선택한다. 따라서, 분할축(3) 의해 2개의 공간으로 분할되고, 분할 정보는 그림 4(b)와 같은 비트벡터로 저장된다. 비트벡터는 미리 할당된 분할축의 사용여부를 나타내며, 분할축 집합의 모든 분할축 정보를 동시에 저장하게 된다. 따라서, 제안하는 기법은 비트벡터를 통해 분할 정보의 크기를 효과적으로 줄인다. 더 많은 데이터가 삽입되어 overflow가 발생하게 되면, 추가적인 분할축을 선택하기 이전에 현재 분할축을 조정하여 데이터가 저장 가능한지 확인하게 된다. 따라서, 그림 5와 같이 6번째 데이터의 삽입으로 인해 overflow가 발생하면, 기존 분할축(3)을 분할축(6)으로 재조정하여 공간을 분할한다. 이를 통해 단말 노드의 사용 정도(node utilization)이 높아지게 된다.

3.3 포인터 제거

색인 구조에서 포인터 제거하는 기법은 색인 기법의 최적화 측면에서 매우 중요한 기술이다. 포인터 제거는 포인터를 포함하고 있는 관련키의 크기와 관련이 있다. 일반적으로 색인 구조에 키 엔트리는 포인터와 함께 영역 정보나 데이터 ID값 등을 저장하며 포인터의 크기는 일반적으로 Integer의 크기와 같다. 32-bit 머신에서 64-bit 머신으로 변경되면, 포인터의 크기는 4바이트에서 8바이트로 증가하여 잠재적으로 동일한 수의 키 엔트리 정보를 저장하기 위해서 더 많은 공간을 필요로 한다. 따라서, 색인 구조의 설계에 있어 포인터 제거 기법은 반드시 고려되어야 할 필수적인 요소이다. 포인터를 제거하기 위한 대표적인 접근 방법으로 CSB^+-트리[9]가 있다. CSB^+-트리는 연속적인 공간에 자식 노드들을 할당하고 첫 번째 자식 노드에 대한 포인터만 저장함으로써 나머지 포인터를 제거하는 방법을 사용한다. 즉, 키 엔트리의 순서와 첫 번째 자식 노드의 포인터를 연산하여 자식 노드에 접근하게 된다. 하지만, KDB-트리 다차원 데이터를 색인하고 있고, 공간의 분할 순서에 따라 자식 노드의 생성 순서가 달라질 수 있기 때문에 1차원 색인 구조인 CSB^+-트리의 기법을 그대로 적용할 수 없다. 제안하는 기법은 CSB^+-트리처럼 자식 노드를 물리적으로 연속된 공간에 할당하는 방법과 함께 연속적인 공간에서 자식 노드의 순서를 명시적으로 나타내는 오프셋 정보를 함께 저장하는 기법을 제안한다. 예를 들어, 그림 6과 같이 분할된 공간이 있다고 가정하자. 분할축의 숫자는 분할축이 결정된 순서를 의미한다. 분할 정보와 자식 노드의 정보를 계층적으로 표현하면 그림 7과 같다. 자식 노드의 정보를 연속적인 공간에 저장한다고 가정할 때, 분할 순서에 따라 오프셋 값을 결정할 수 있다. 즉, 분할축(1)에 의해 분할된 R1과 R2는 자식 노드 그룹중 첫번째와 두번째 노드에 할당되고, 분할축(2)에 의해 R2와 R3가 분할되기 때문에 R3는 세번째 노드에 할당된다. 그 다음, 분할축(3)에 의해 R1과 R4가 분할되기 때문에 R4는 네번째 노드에 할당된다. 따라서, {R1, R2, R3, R4} 의 오프셋값은 {0, 1, 2, 3}이 된다. 중간 노드는 자식 노드 그룹에 대한 포인터, 즉, CGP를 그림 8과 같이 유지하고 있기 때문에 자식 노드에 대한 모든 포인터를 유지하지 않고도, 식 (1)에 의해

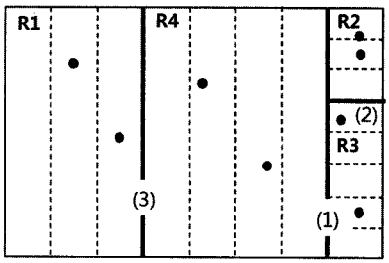


그림 6 분할의 예

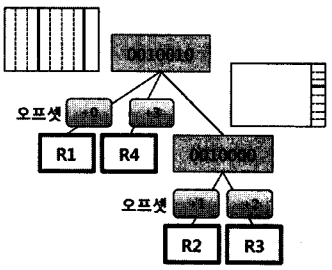


그림 7 분할의 계층적인 표현

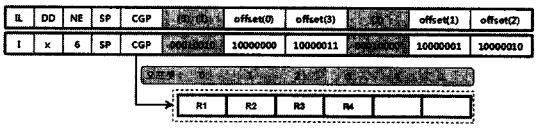


그림 8 KDB_{CS}^{+}-트리의 예

접근 포인터의 계산이 가능하다.

{접근 포인터} = CGP+*{오프셋}***{노드의크기}* (1)

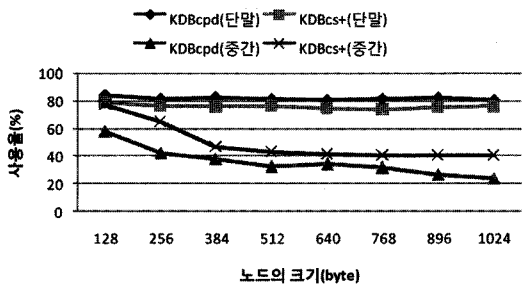
4. 성능 평가 및 분석

4.1 실험 환경

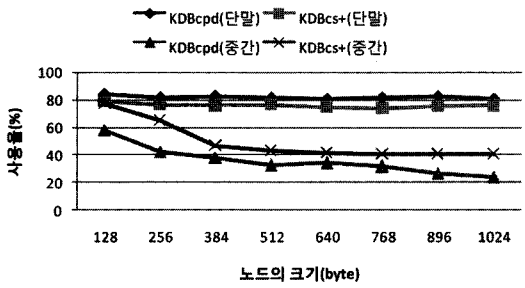
제안하는 기법의 우수성을 보이기 위해서 기존 KDB_{CPD}-트리와 성능을 비교평가 한다. KDB_{CPD}-트리는 DT-분할 정책만 사용하고, 1-복제 정책은 사용하지 않는다. 제안하는 기법은 공간을 8등분하며 중간 노드의 각 엔트리는 1byte의 크기를 갖는다. 균등분포, 가우시안 분포, 편향 분포 특성을 보이는 2차원 데이터 100,000개를 1,000 × 1,000 공간에 삽입하며, 데이터 삽입후 노드의 사용율과 데이터 탐색에 필요한 평균 노드 접근수를 비교한다. 편향 분포 데이터는 균등 분포를 보이는 10,000개 데이터와 100 × 100 공간에 집중된 90,000개 데이터의 조합으로 구성된다.

4.2 노드 사용율

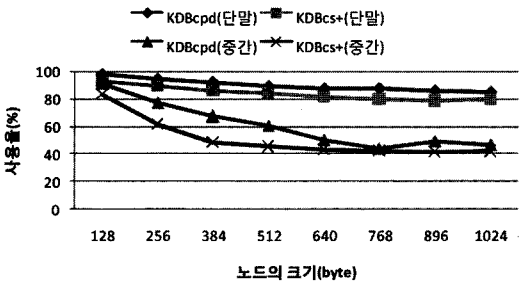
그림 9는 100,000개의 데이터를 삽입 후, 중간 및 단말 노드의 사용율을 나타낸다. 단말 노드의 사용율은 모든 분포 데이터에 대해서 두 색인 기법이 유사한 결과



(a) 균등분포 데이터



(b) 가우시안분포 데이터



(c) 편향분포 데이터

그림 9 노드의 사용율

를 보여준다. 중간 노드의 경우, 균등 분포와 가우시안 분포 데이터를 삽입했을 때 제안하는 기법이 약 10~20% 사용율의 향상된 결과를 보여주며, 편향 데이터에 대해서 오히려 저하된 성능을 보여준다. 제안하는 기법이 더 많은 팬-아웃을 보여주지만, 7개의 분할축을 이용하기 때문에 상대적으로 더 많은 분할이 발생하고, 사용율이 낮은 중간 노드가 많이 생성되기 때문이다.

4.3 평균 노드 접근수

그림 10은 각 분포 데이터에 대한 평균 노드 접근수를 나타낸다. 제안하는 기법의 경우, 모든 데이터의 분포에 대해서 약 3정도의 평균 노드 접근 수를 보여준다. KDB_{CPD}-트리의 경우, 5~16의 평균 노드 접근 수를 보여주며, 노드의 크기가 증가함에 따라 팬-아웃이 증가하

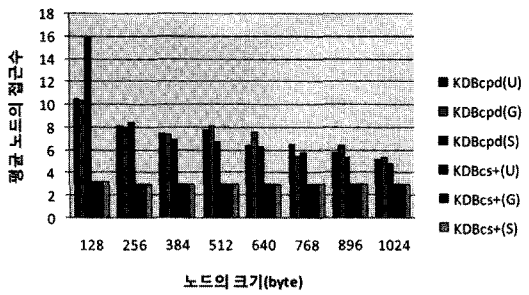


그림 10 평균 노드 접근수

기 때문에 평균 노드의 접근수가 줄어든다. 즉, 제안하는 기법은 분할축 선정 방법과 포인터 제거 기법을 통해 중간 노드의 팬-아웃을 늘리고, 중간 노드 엔트리의 계층적인 표현을 통해 중간 노드의 사용율을 높임으로써 평균 노드의 접근수를 줄였다.

5. 결 론

본 논문에서는 공간 분할 색인 기법의 편향 데이터 처리 문제를 해결하고, 노드의 탐색 성능을 향상시키기 위한 새로운 색인 기법인 KDB_{CS}^{+}-트리를 제안하였다. KDB_{CS}^{+}-트리는 분할 정보를 비트벡터로 표현하는 압축 기법과 노드의 그룹화를 통한 포인터 제거 기법을 활용하여 중간 노드의 팬-아웃을 증가시키고, 중간 노드의 엔트리를 계층적으로 표현함으로써 중간 노드의 사용율을 높인다. 그 결과, 균등 분포와 가우시안 분포 데이터에 대해서는 KDB_{CPD}-트리보다 약 10~20% 향상된 노드 사용율을 보이며, 편향데이터에 대해서 약 10~20% 저하된 노드 사용율을 보인다. 하지만, 중간 노드의 팬-아웃을 크게 증가시킴으로써 모든 분포 데이터에 대하여 트리의 높이를 감소시키고, 평균 노드 접근수를 줄였다. 향후 연구로는 노드 복제 기법과 함께 벌크 데이터 삽입을 위한 기법을 적용하는 것이다.

참 고 문 헌

[1] K. Chakrabarti and S. Mehrotra "The Hybrid Tree: An Index Structure for High Dimensional Feature Spaces," *Proceedings of the International Conference on Data Engineering*, pp.440-447, 1999.

[2] R. Orlandic and B. Yu, "Estimating the Probability of Overlap between Multi-Dimensional Rectangles in the Analysis of Spatial Structures," *Information Sciences*, 2001.

[3] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proceedings of the ACM SIGMOD Conference*, pp.10-18, 1981.

[4] A. Guttman, "R-trees: A Dynamic Index Structure for Spatial Searching," *Proceedings of ACM SIGMOD Conference*, pp.47-57, 1984.

[5] H. Lin and S. Chen "High Indexing Compression for Spatial Databases," *Proceedings of the IEEE 8th International Conference on Computer and Information Technology Workshops*, pp.20-25, 2008.

[6] M. Yeo, Y. Min, K. Bok and J. Yoo, "The Optimization of In-Memory Space Partitioning Trees for Cache Utilization," *IEICE Transaction on Information and Systems*, vol.E91-D, no.2, pp.243-250, 2008.

[7] R. Orlandic and B. Yu, "Implementing KDB-Trees to Support High-Dimensional Data," *Proceedings of the International Database Engineering & Applications Symposium*, pp.58-67, 2001.

[8] B. Yu, T. Bailey, R. Orlandic and J. Somavaram, "KDBKD-Tree: A Compact KDB-Tree Structure for Indexing Multidimensional Data," *Proceedings of the International Conference on Information Technology: Coding and Computing*, pp.676-680, 2003.

[9] J. Rao and K. A. Ross, "Making B+-Trees Cache Conscious in Main Memory," *Proceedings of the ACM SIGMOD Conference*, pp.475-486, 2000.