CIR-Tree를 위한 효율적인 대량적재 알고리즘의 설계 및 구현
(Design and Implementation of an Efficient Bulk Loading Algorithm for CIR-Tree)

피 준 일 † 송 석 일 † 유 재 수 ††
(Jun Il Pee) (Seok Il Song) (Jae Soo Yoo)

요 약 이 논문에서는 고차원 색인 구조인 CIR-트리를 위한 효율적인 벌크로딩 알고리즘을 설계하고 구현한다. 벌크로딩 기법은 대량의 고차원 데이타가 색인 구성 시 함께 주어지는 경우 색인의 구성을 빠르게 하고 구축한 색인의 검색 성능을 향상시킨다. CIR-트리는 반드시 필요한 차원만 이용해서 비단말 노드의 엔트리를 구성하기 때문에 엔트리 크기가 일정하지 않다. 이 특성은 비단말 노드의 분기율을 높이고 탐색 성능을 향상시키는 효과가 있다. 기존에 다차원 및 고차원 색인구조를 위한 벌크로딩 기법이 제안되었지만 이러한 CIR-트리의 특징을 제대로 살릴 수 있는 방법은 없다.

이 논문에서는 기존의 벌크로딩 알고리즘을 개선하면서 CIR-트리의 특징을 효과적으로 색인 구성에 반영할 수 있는 알고리즘을 제안한다. 또한 이를 BADA-III의 하부 저장 시스템인 MiDAS-III에서 구현하고 다양한 실험을 통해 그 성능을 입증한다.

키워드 : 고차원 색인구조, 벌크로딩, CIR-Tree, 저장시스템

Abstract In this paper, we design and implement an efficient bulk-loading algorithm for CIR-Tree. Bulk-loading techniques increase node utilization, improve query performance and reduce index construction time. The CIR-tree has variable size of internal node entries since it only maintains minimal dimensions to decriminate child nodes. This property increases fan-out of internal nodes and improves search performance. Even though several bulk-loading algorithms for mutli/high-dimensional index structures have been proposed, we cannot apply them to CIR-tree because of the variable size of internal node entries. In this paper, we propose an efficient bulk- loading algorithm for CIR-tree that improves the existing bulk-loading algorithm and accomodates the property of CIR-tree. We also implement it on a storage system MiDAS-III and show superiority of our algorithm through various experiments.

Key workds : High-dimensional index structure, bulk-loading, CIR-Tree, Storage system

1. 서 론

최근 원격진료 시스템, 지리정보 시스템, 이미지 데이타베이스 시스템, 멀티 미디어 데이타베이스 시스템 등에서 내용기반 이미지검색을 보다 효과적으로 수행하기 위한 연구가 활발하게 진행되어 왔다. 내용기반 이미지 검색을 위해서는 대량의 이미지들을 고차원의 특징 벡터로 변환하고 그것을 고차원 색인구조를 이용해 색인하여야 한다. 고차원 색인구조를 구성하는 일반적인 방법은 주어진 데이타를 하나씩 개별적으로 삽입하는 것이다. 그러나 응용분야의 특성상 색인 구성 시 대량의 정적인 데이타가 주어지는 것이 보통이다. 이럴 경우 대량의 정적인 데이타를 하나씩 삽입하는 것은 그 속도가 매우 느리고 저장공간 활용률을 저하시킬 뿐 아니라 같은 데이타 집합이라도 그 삽입의 순서가 색인구조의 효

· 이 논문은 2001년도 한국학술진흥재단(KRF-2001-041-E00233)의 지원에 의하여 연구되었음.
† 비 회 원 : 충북대학교 정보통신공학과
pji@netdb.chungbuk.ac.kr
prince@netdb.chungbuk.ac.kr
†† 종신회원 : 충북대학교 정보통신공학과 및 컴퓨터정보통신연구소 교수
yjs@cbucc.chungbuk.ac.kr
논문접수 : 2001년 1월 15일
심사완료 : 2002년 3월 18일

율성에 큰 영향을 미치게 된다. 이에 따라 최근 색인을 구축할 객체들이 미리 모두 정해져 있을 때 데이타 집합을 알고 있다는 점을 충분히 이용하여 구성 시간이 보다 빠르고 탐색 성능을 향상시킬 수 있도록 색인을 구축하는 벌크로딩 알고리즘이 제안되었다.

벌크로딩이란 대량의 데이타를 다른 연산의 방해를 받지 않고 데이타의 특성을 충분히 고려하여 색인을 생성하는 것이다. 최근 십여 년 동안 R-트리를 위한 몇 몇 벌크로딩 알고리즘들이 제시되었으나, 고차원의 데이타를 위한 벌크로딩 알고리즘은 소수에 불과하다. 기존의 다차원 및 고차원 색인 구조를 위한 벌크로딩 알고리즘은 크게 정렬을 수행하는 방법과 정렬을 수행하지 않고 단순히 데이타를 이분하거나 버퍼를 통한 I/O시간을 개선하는 방법들로 구분해 볼 수 있다. 정렬을 수행하여 색인을 구성하는 벌크로딩 알고리즘은 검색 성능은 개선시킬 수 있지만 색인의 구성 시간이 증가한다는 단점이 있다. 정렬을 수행하지 않고 버퍼트리를 이용하는 방법은 구성시간은 짧지만 하나씩 삽입하는 방법과 비교해서 탐색성능을 개선시킬 수 없다.[7] 이런 문제점을 해결하고 있는 방법이 UBBT (Unbalanced Bisection Bulk-loading Technique)[8, 9]이다. 이 방법은 트리의 형태(topology)를 미리 계산하여 이를 색인 구성 시 적극 이용한다. 이 논문에서는 MiDAS-Ⅲ에 구현된 CIR-트리 관리기에 적합한 벌크로딩 기법을 설계한다. 그런데 CIR-트리는 분기율을 높이기 위해 비단말 노드의 엔트리 구성 시 활성차원과 비활성차원을 사용하게 되며, 이런 특성으로 인하여 엔트리들의 크기가 모두 다를 수 있다. 이런 특징으로 인하여 벌크로딩 수행 시 트리의 형태를 미리 정확하게 예측하는 것은 불가능하며, UBBT를 그대로 적용하는 것은 문제가 있다.

본 논문에서는 기존의 벌크로딩 알고리즘들 중 가장 성능이 우수한 UBBT를 개선하고, CIR-트리의 특징을 효과적으로 수용할 수 있는 벌크로딩 알고리즘을 설계한다. 또한 이를 BADA-Ⅲ의 하부 저장 시스템인 MiDAS-Ⅲ상에서 CIR-트리에 적용하여 구현하고 다양한 실험을 통해 우수성을 입증한다. 이 논문의 구성은 다음과 같다. 2장에서 CIR-트리와 기존의 벌크로딩 알고리즘에 대해서 살펴보고, 3장에서는 제안하는 벌크로딩 알고리즘에 대해서 설명한다. 4장에서는 CIR-트리를 위한 벌크로딩 알고리즘의 구현에 대한 내용과 성능평가 결과에 대해 기술하고 마지막으로 5장에서 결론을 맺는다.

2. 관련 연구

여기서는 먼저 CIR-트리에 대하여 알아보고 기존의 고차원 색인 구조를 위한 벌크로딩 알고리즘과 문제점에 대해서 살펴본다.

1.1 CIR-트리

CIR(Content based Image Retrieval)-트리[1, 2]는 내용 기반 이미지 검색을 지원하기 위해 제안된 고차원 색인구조이다. 이 색인구조는 기존의 R-트리 계열의 고차원 색인구조들의 문제점인 차원의 저주 현상을 개선하고 있다. CIR-트리의 특징은 다음과 같이 정리할 수 있다. 첫째, 비단말 노드들에는 분별력이 있는 차원만을 선정하여 MBR을 구성함으로써 분기율을 증가시킨다. 둘째, 겹침 영역을 최소화하기 위한 노드 분할 방법을 제공하며 수퍼 노드 개념을 도입한다. 셋째, 이미지 특징들의 군집화를 위해 무게 중심을 이용하는 보다 개선된 재삽입을 수행한다.

CIR-트리를 위한 벌크로딩 설계 시 고려해야할 점은 비단말 노드에 저장되는 엔트리의 크기가 모두 다를 수 있다는 점이다. 분별력이 있는 일부 차원만으로 MBR을 구성하기 때문에 엔트리마다 활성차원과 비활성 차원의 수가 다를 수 있다. 이 특징은 데이타 집합을 미리 알고 있다 하더라도 전체 트리의 형태를 정확하게 예측하는 것이 불가능하게 한다.

1.2 기존의 벌크로딩 알고리즘들

기존의 벌크로딩 알고리즘들은 그림 1에서와 같이 정렬을 수행하는 알고리즘과 정렬을 수행하지 않는 알고리즘으로 분류해볼 수 있으며 다시 상향식 구성과 하향식 구성으로 각각 분류할 수 있다.

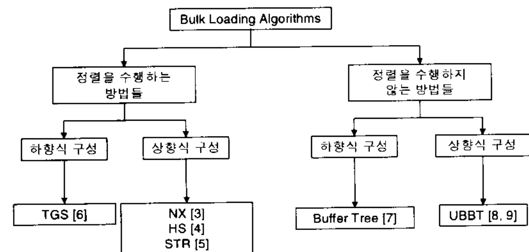


그림 1 벌크로딩 알고리즘의 분류

정렬을 수행하는 방법들로 NX[3], HS[4], STR[5], TGS[6]가 있다. 이들 중 상향식으로 색인을 구성하는 알고리즘들은 NX[3], HS[4], STR[5]가 있다. NX(Nearest-X) 알고리즘[3]은 구현이 매우 쉽고 포인트 질의에 대해서 좋은 검색성능을 발휘한다. 그러나 영역 질의에서는 성능이 저하되는 단점을 가지고 있다. HS(Hilbert Sort) 알고리즘[4]은 힐버트 커브(Hilbert curve)에 의해 결정된

저장 순서를 이용하는 알고리즘을 제시하였다. HS 알고리즘은 전체 영역의 크기를 적게 유지하면서 전체 영역의 둘레 길이를 감소시키는 결과를 발생시키기 때문에 NX 알고리즘에 비해 영역 질의에 대한 성능을 상당히 향상시켰다. STR(Sort-Tile-Recursive) 알고리즘[5]은 구현이 쉽고 이전의 알고리즘에 비해 상당한 성능 향상을 얻을 수 있었다. 그러나 HS 알고리즘이 VLSI 데이타에 대해서는 STR 알고리즘보다 더 좋은 성능을 발휘하기도 한다. 따라서 HS 알고리즘과 STR 알고리즘의 성능은 주어진 데이타의 형태에 의존한다고 말할 수 있다[5]. 지금까지 살펴본 NX, HS, STR 알고리즘은 고정된 전처리 과정을 사용하며 상향식으로 색인을 구성한다. 이에 반해 하향식으로 트리를 구성하는 TGS(Top-down Greedy Split) 알고리즘[6]도 제안되었다. TGS 알고리즘은 데이타베이스에 존재하는 데이타들을 분할하기 위해 최적의 분할 차원과 분할 위치를 선택하는 전처리 과정을 사용한다. TGS 알고리즘은 지금까지 설명한 방법들보다 효과적인 색인을 구성할 수 있으나 최적의 분할 차원과 분할 위치를 선택하기 위하여 드는 비용으로 인하여 색인의 구성 시간이 크게 증가한다는 단점이 있다. 정렬을 수행하는 벌크로딩 알고리즘들은 색인의 구축 후 탐색 성능이 향상된다는 장점이 있지만 대량의 데이타에 대해 정렬을 수행해야하므로 색인의 구성 시간이 오래 걸린다는 단점이 있다.

정렬을 수행하지 않는 벌크로딩 알고리즘으로는 버퍼 트리를 이용하는 방법[7]와 UBBT[8, 9]가 있다. [7]의 벌크로딩 알고리즘은 데이타 정렬 방식을 사용하지 않고 분할이나 합병기법을 사용하여 색인 구조를 만든다. 특히 색인 노드에 버퍼를 두는 버퍼 트리를 이용하여 색인을 생성하는 시간을 절감시켰다. 하지만 이 방법은 데이타 집합의 특성을 효과적으로 반영할 수 없어서 탐색 성능을 향상시킬 수 없다는 단점이 있다. UBBT는 데이타들을 정렬하지 않고 이분하는 방법을 제안하였다. UBBT는 색인 구성시간과 검색 성능을 종합적으로 고려해볼 때 기존 알고리즘들 중 가장 우수한 성능을 나타내며 이 논문에서 제안하는 벌크로딩 알고리즘도 이 방법에 기반한다.

UBBT의 특징은 다음과 같이 요약할 수 있다. 색인 시간을 단축시키기 위해서 데이타 집합을 분할할 때 정렬을 하지 않고 단지 기준 값보다 작은 데이타 집합과 크거나 같은 값을 갖는 데이타 집합으로 이분한다. 두 번째로 데이타 집합을 분할할 때, 그 분할 비율이 정해지지 않고 사용자에 의해 결정될 수 있으며 그 분할 비율을 불균등하게 적용하여 검색 성능을 향상시킨다. 마지막으로 단말 노드를 먼저 만들고 다시 이를 가리키는 색인 노드(비 단말 노드)를 구성하는 상향식 방법이다. UBBT는 ①트리 형태 결정, ② 분할 전략 결정, ③ 데이타 집합 분할, ④ 색인 노드 구성의 4단계를 거쳐서 색인을 구축한다. 첫 번째 단계에서는 색인을 구축할 데이타의 차원, 데이타의 개수, 색인 노드 크기로부터 색인 트리의 형태를 구한다. 계산되는 트리의 형태로는 트리의 전체 높이와 트리의 레벨에 따른 비단말 노드들의 분기율 등이다.

두 번째 단계에서는 데이타 집합을 분할하기 위한 분할 전략을 세운다. 데이타 집합을 어떤 차원을 기준으로 나눌 것이며, 또한 어떤 비율로 나눌 것인지를 결정한다. 분할 차원은 최대 길이를 갖는 차원이 되며, 분할 비율은 불균등(1:3, 1:9등)하게 선택한다. 선택된 분할 비율에 따라 데이타 집합의 이분 위치를 결정하게 되는데, 색인의 구성 시간을 단축시키기 위하여 정확한 분할 위치결정 대신 분할 범위를 둔다. 세 번째 단계에서는 두 번째 단계에서 결정한 분할 전략에 따라 데이타 집합을 이분한다. 이분을 할 때는 변형된 퀵 정렬을 수행하며 이때 분할 범위를 포함하는 부데이타 집합에 대해서는 단 한번의 순환호출을 수행한다. 데이타 집합이 주 기억공간에 적재될 수 있는 양인 경우에는 변형된 퀵정렬을 이용해 데이타 집합을 분할하며 주 기억공간에 적재되지 못하는 경우에는 데이타 집합에서 주기억공간 만큼의 데이타를 표본 추출하여 피벗값(Pivot)을 결정하고 그 값에 따라서 전체 데이타 집합을 이분한다. 마지막 단계에서는 앞의 단계에서 전체 데이타 집합을 계속 반복적으로 분할하여 단말 노드들이 생성되면 이들로부터 색인 노드들을 구성하는 과정을 거친다.

2.3 UBBT의 문제점 분석

이상의 기존 벌크로딩 알고리즘들 중 가장 우수한 성능을 발휘하는 것은 UBBT 이다. 하지만 UBBT에는 몇 가지 고려해야할 문제점이 있다. 먼저 분할 범위 계산 시 잘못된 분할 범위를 갖는 경우가 발생한다. UBBT는 트리 형태(트리의 높이) 계산 시 노드에 평균적으로 들어가는 엔트리 수($C_{eff,data}$, $C_{eff,dir}$)를 고려해서 계산하는데 반해 분할 범위 계산 시에는 노드에 최대로 들어갈 수 있는 엔트리 수($C_{max,data}$, $C_{max,dir}$)를 고려하여 계산하기 때문이다. 분할 범위를 구하는 방법은 다음과 같다. 왼쪽 데이타 집합이 가질 수 있는 최대 수는 각 노드마다 최대로 들어갈 수 있는 엔트리 수를 고려하여 계산하고, 최소 수는 총 엔트리 수와 오른쪽 집합이 가질 수 있는 최대 수의 차이로 계산하게 된다. 이때 왼쪽에 최대로 들어갈 수 있는 엔트리 수가 이분할 전체 엔트리 수보다 크거나 같은 경우가 생기거나 왼쪽 집합에 최소로 들어갈 수 있는 엔트리 수가 너무 작아서 높이 균등(height-balanced) 색인이 구축이 되지 않을 수도 있다. 간단한 예를 그림 2에서 보인다. 그림에서 보듯이 왼쪽의 트리는 최초의 트리 형태로 계산한 트리의

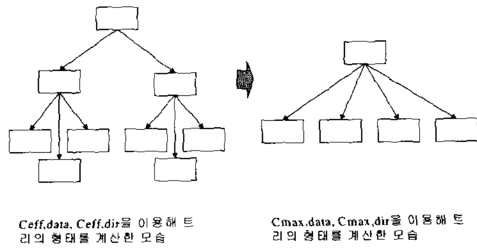


그림 2 트리 형태 결정의 문제점

모습이다. 이 때는 노드에 평균적으로 들어갈 수 있는 엔트리 수로 계산하여 높이가 3인 트리를 갖는 것으로 판단되었다. 하지만 이를 단말 노드와 색인 노드에 최대 개수의 엔트리가 들어가는 것으로 다시 계산하면 트리의 높이가 2로 줄어들 수도 있다. 이때 왼쪽에 최대로 들어갈 수 있는 데이타의 수가 전체 데이타 수보다 크게 된다.

또 다른 문제는 이분을 수행할 때 발생한다. UBBT에서 이분은 퀵 정렬 알고리즘을 변형시켜서 사용하고 있다. 데이타 집합 전체를 정렬하는 것이 아니고 단지 피벗값을 기준으로 특징 벡터의 분할 차원 값이 작은 것과 크거나 같은 것으로 이분한다. 이분하는 방법은 내·외부 이분 모두 공통적으로 임의로 피벗값을 정해서 피벗값을 기준으로 1차 이분을 수행한다. 이분 결과 피벗값이 분할 범위안에 들어오면 더 이상 이분을 수행하지 않는다. 피벗값이 분할 범위안에 들어오지 않는 경우에는 이분된 두 데이타 집합들 중에서 분할 범위를 포함하는 데이타 집합에 대해서 같은 방법으로 2차 이분을 수행한다. 이런 작업을 피벗값이 분할 범위에 들어올 때까지 반복적으로 수행한다. 하지만 그림 3의 (a)와 같은 불균등한 분포를 갖는 데이타 집합에 대해서 피벗값과 같은 값을 갖는 데이타들이 분할 범위에 걸쳐 있을 수 있다. 이 경우 피벗값은 결코 분할 범위에 포함될 수 없으며 이분은 멈출 수가 없다. 이를 해결하기 위해서는 적절히 분할 범위내의 한 지점으로 이분을 수행해야 하지만 UBBT는 정렬을 수행하지 않으므로 그림 3의 (b)처럼 분할차원의 값이 같은 데이타들이 모여있지 않다.

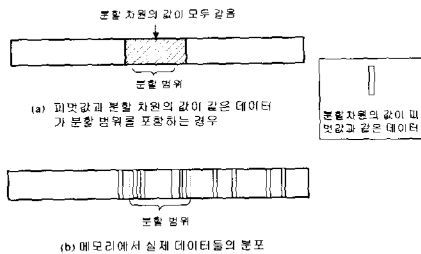


그림 3 이분 시 발생하는 첫번째 문제점

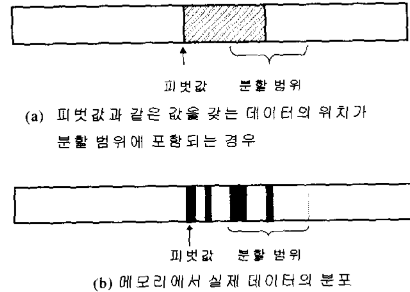


그림 4 이분 시 발생하는 두 번째 문제점

또한 그림 4의 (a)와 같은 경우에는 비록 피벗값이 분할 범위안에 들지는 않지만 피벗값과 같은 데이타의 개수가 분할 범위안에 들게 된다. 이런 경우에는 더 이상 이분을 수행하지 않고 피벗값의 위치에서 피벗값과 같은 값을 갖는 데이타의 개수를 더한 만큼의 지점에서 이분을 하고 멈추면 된다. 하지만 이것 역시 그림 4의 (b)처럼 정렬이 되어 있는 상태가 아니므로 불가능하다.

마지막으로 언급할 것은 UBBT 방법을 그대로 CIR-트리에 적용할 수 없다는 점이다. 앞의 2.1에서 제시했던 것처럼 CIR-트리는 색인 노드 엔트리가 전체 특징 차원 중 일부를 이용해 하위 노드에 대한 MBR을 구성한다. CIR-트리에서는 비활성차원과 활성 차원이라는 개념을 이용하는데 이로 인해 색인 노드 엔트리의 크기가 일정하지 않다. 활성 차원은 엔트리들을 구별할 수 있는 차원들을 말하며, 비활성 차원은 엔트리 전체가 같은 값을 갖는 차원을 말한다. UBBT에서는 트리의 형태를 추정하여 이를 기반으로 각 색인 노드의 분기율을 계산하여 이분을 수행하고 색인을 구성하기 때문에 CIR-트리의 경우처럼 정확한 트리의 형태를 계산하기가 어려운 경우에는 UBBT를 수정없이 그대로 적용하는 것은 문제가 있다.

본 논문에서는 UBBT 알고리즘을 기반으로 지적한 두 문제점을 해결하면서 CIR-트리의 특징을 효과적으로 지원하는 벌크로딩 기법을 설계하고 구현한다.

3. CIR-트리를 위한 벌크로딩 기법

3.1 제안하는 벌크로딩 기법의 특징

제안하는 벌크로딩 기법의 특징은 다음과 같이 4가지로 요약할 수 있다. 첫 번째, UBBT에서 제안한 트리 형태 계산을 통한 비균등 이분법을 기반으로 한다. 데이타 집합으로부터 미리 알 수 있는 정보를 통해 트리 형태를 계산하고 데이타 집합을 비균등하게 분할한다는 점은 기존의 UBBT 방법을 따른다. 이 방법을 기반으로 하여 보다 개

선된 그리고 CIR-트리의 특징을 수용하는 벌크로딩 기법을 설계한다. 두 번째, 분할 범위를 계산할 때 부 트리가 가져야 하는 최소 한계 엔트리 수를 고려한다. 이것은 2장의 기존 연구에서 언급했던 잘못된 분할 범위 계산의 문제점을 해결하기 위한 것이다. 여기서는 노드가 가져야 할 최소 한계 엔트리 수를 고려함으로써 잘못된 분할 범위를 갖는 문제를 해결한다. 세 번째, 이분을 수행할 때 피벗값과 같은 값을 갖는 데이타들을 별도로 관리한다. 피벗값과 같은 값을 갖는 데이타들을 피벗값의 바로 옆에 별도로 관리하면 그림 3과 그림 4의 경우에 문제없이 이분을 수행할 수 있다. 네 번째, CIR-트리가 갖는 가변 MBR 특징을 수용한다. CIR-트리의 특징 때문에 색인의 형태를 정확히 계산할 수 없다는 문제를 해결하기 위해서 활성 차원과 비활성 차원을 고려하여 각 색인 노드의 분기율을 계산한다. 또한 분할 차원의 선택도 활성차원으로 제한한다. 트리의 형태 계산 시 색인 노드의 정확한 엔트리 수를 알지 못하여 색인 노드 구성 시 넘침이 발생할 수 있는데, 이를 위해 이분 시 차등 분할 비율을 적용하며 넘침 발생 시 노드 분할과 수퍼 노드로의 확장을 통해 해결한다.

설계 시 이상의 내용들을 주로 고려하며, 이들이 바로 CIR-트리를 위한 벌크로딩 기법의 특징이 된다. 다음에서 설계한 벌크로딩 기법에 대한 보다 자세한 설명을 하도록 한다.

3.2 제안하는 알고리즘

CIR-트리를 위한 벌크로딩 알고리즘은 주어진 데이타 집합을 이분을 통해 단말 노드를 생성하고, 단말 노드가 생성되면 그 단말 노드의 MBR을 상위에 반영하여 상향식으로 색인을 생성하는 방식을 취한다.

CIR-트리를 위한 벌크로딩 알고리즘은 크게 3단계로 구성된다.

단계 1 : 트리의 형태 계산 (높이와 분기율 계산)

단계 2 : 부 데이타 집합으로 분할(분할 전략 결정과 이분)

단계 3 : 색인 노드 생성

루트 노드가 완성되어 색인이 구성될 때까지 위의 세 단계를 반복 수행한다. 먼저 트리의 형태를 계산하는 단계에서는 주어진 데이타 집합으로 구성될 색인에 대한 정보를 미리 파악하여 높이와 분기율을 계산하게 되는데, 이때 데이타 집합이 가지고 있는 비활성 차원을 고려하여 트리의 형태를 계산한다. 계산을 통해 얻은 트리 형태 정보는 실제 데이타를 이분할 때 적용하게 된다. 다음으로 주어진 데이타 집합을 부 데이타 집합으로 분할하는 과정을 거치게 된다. 부 데이타 집합으로 분할하는 과정은 분할 전략 결정과 이분 과정을 반복 수행하여 앞서 계산한 분기 수만큼의 부 데이타 집합으로 나누게 된다. 반복적인 분할과정을 통해 데이타 집합으로부터 단말 노드들이 생성되면 단말 노드의 MBR정보를 상위의 노드에 반영하게 된다. 이렇게 해서 하나의 색인 노드가 생성되면 다른 부 데이타 집합에 대해서도 위의 과정을 반복 수행하여 최종 루트 노드가 생성되면 벌크로딩 수행을 마치게 된다. 각각의 단계에 대해서 자세히 살펴보도록 하겠다.

3.1.1 트리 형태 (Tree Topology) 계산

벌크로딩을 통해 생성될 색인의 형태를 미리 계산하는 단계이다. 계산을 통해 얻어지는 형태로는 생성될 색인의 높이와 분기율 등이다.

높이와 분기율을 계산하기 위해서는 먼저 $C_{max,data}$ 와 $C_{max,dir}$을 결정해야 한다. $C_{max,data}$는 단말 노드에 들어갈 수 있는 최대 엔트리 수를 말하며, $C_{max,dir}$은 색인 노드에 들어갈 수 있는 최대 엔트리 수를 말한다. $C_{max,data}$는 수식 1을 통해 얻을 수 있다. ***DataObject***의 크기는 특징 벡터와 데이타 파일에서 레코드를 식별할 수 있는 RID (Record ID)의 크기의 합으로 이루어진다.

$$C_{\max,data} = \left\lfloor \frac{NodSize}{sizeof(DataObject)} \right\rfloor \qquad \text{(수식 1)}$$

$C_{max,dir}$은 $C_{max,data}$를 구하는 방법과 유사하지만, 색인 노드에 들어갈 수 있는 엔트리의 정확한 크기를 예측하기 어렵다. CIR-트리에서는 색인 노드 구성 시에 활성 차원과 비활성 차원을 이용한다. 활성 차원 수는 색인 구성 시 주어지지만 비활성 차원의 수는 노드마다 다르며, 색인 노드의 경우 각 엔트리마다 다른 수의 비활성 차원 수를 가질 수 있다. 트리의 형태 계산 시에는 색인 노드의 각 엔트리 크기를 알 수 없으므로, 색인 노드에 들어갈 엔트리들의 정확한 크기를 구하는 것이 아니라 현재 데이타 집합의 MBR 정보를 이용해 현재 노드의 비활성 차원의 수(***NonActiveDim***)와 활성 차원의 수(***ActiveDim***)를 구한다. 경우에 따라서는 노드의 비활성 차원수보다 많은 비활성 차원을 갖는 엔트리들도 존재하지만 형태 계산 시에는 고려하지 않는다. 색인 노드의 엔트리의 크기는 수식 2와 같다. 활성 차원(***ActiveDim***)은 각 차원별로 최소값과 최대값을 유지하므로 두 배의 크기가 필요하며 ***NodeID***는 하위 노드를 나타내는 노드 식별자를 의미한다.

$$NonLeafEntry = (ActiveDim * 2) + NonActiveDim + NodeID \qquad \text{(수식 2)}$$

$C_{max,dir}$은 수식 3을 통해 얻을 수 있다.

$$C_{\max,dir} = \left\lfloor \frac{NodeSize}{sizeof(NonLeafEntry)} \right\rfloor \qquad \text{(수식 3)}$$

이렇게 얻은 값에 저장 공간 활용률을 곱하면 $C_{eff,data}$와 $C_{eff,dir}$값을 얻을 수 있게 되는데, 저장 공간 활용률은 구성된 색인에서 노드들이 평균적으로 어느 정도 이상의 공간

이 사용되도록 보장하는 것이다. 노드들이 저장 공간 활용률을 만족하도록 보장하여 불필요하게 높이가 증가하는 것을 막을 수 있다.

위에서 얻은 값들을 통해 높이와 분기율을 결정할 수 있게 되는데 먼저 높이는 수식 4를 통해 얻을 수 있다.

$$h = \left\lceil \log_{C_{eff,dir}}\left(\frac{n}{C_{eff,data}}\right)\right\rceil + 1 \qquad \text{(수식 4)}$$

이와 같은 수식을 통해 생성될 색인의 높이가 결정되었다면, 현재 데이타 집합이 가질 수 있는 분기율을 결정해야 한다. 분기율은 각 레벨마다 별도로 재 계산을 한다. 만약 데이타 집합을 부 데이타 집합으로 나눈다면 각 부 데이타 집합은 원래 데이타 집합이 가지고 있던 비활성 차원 수보다 더 많은 비활성 차원을 가질 수 있다. 따라서 레벨마다 부 데이타 집합이 가질 수 있는 분기의 수는 달라질 수 있으며, 각 부 데이타 집합을 분할할 때마다 다시 계산해야 한다. $C_{max,dir}$을 구할 때처럼 분기율 계산 시에도 비활성 차원을 고려한다. 수식 5를 통해서 분기율을 구할 수 있다.

$$fanout(h, n) = \min\left(\left\lceil \frac{n}{C_{eff,data} \cdot C_{eff,dir}^{h-2}}\right\rceil, C_{max,dir}\right) \qquad \text{(수식 5)}$$

예를 들어, 만약 전체 차원이 100차원이고 활성 차원이 40인 특징 벡터 1000개가 있고, ***NodeSize***가 4K라고 하자. 이때 색인 노드의 엔트리에 비활성 차원이 없고 단말노드와 색인 노드의 엔트리에서 RID와 NodeID는 고려하지 않는다면, 위의 수식을 통해 $C_{max,data}$와 $C_{max,dir}$은 각각 10과 12가 되며, 저장공간 활용률을 80%라고 가정하면 $C_{eff,data}$와 $C_{eff,dir}$은 8과 9가 된다. 이 정보를 통해 높이는 4이고 루트 노드의 분기율이 2라는 것을 알 수가 있다. 하지만 만약 비활성 차원의 수가 40이라고 가정하면 $C_{max,dir}$와 $C_{eff,dir}$은 8과 6이 된다. 이로부터 높이가 4이고 루트의 분기 수가 4인 정보를 얻을 수 있다.

3.2.2 부 데이타 집합으로 분할

부 데이타 집합으로 분할하는 과정은 분할 전략 수립단계와 이분 과정을 반복 수행하며, 앞에서 구한 분기 수만큼의 부 데이타 집합으로 나누는 과정이다.

가) 분할 전략 수립

데이타 집합을 이분하기 위해서는 분할 차원과 분할 위치가 필요하다. 분할 차원은 최대 길이를 갖는 차원이 선택되는데, 이전의 많은 논문에서 증명되었던 것처럼 최대 길이를 갖는 차원을 분할 차원으로 선택하는 것이 질의 시 두 분할된 영역을 모두 찾아갈 확률이 더 적다.

그림 5에서 이를 보여준다. 지름이 r인 질의가 점선으로 된 실제 MBR을 접근할 확률을 구하기 위해서 ***Minkowski Sum***을 구한다. ***Minkowski Sum***은 실제 MBR

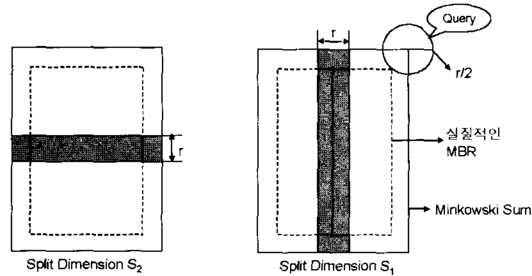


그림 5 Minkowski Sum

의 각 변을 각각 지름이 r인 범위 질의만큼 증가시켜서 구할 수 있다. 이 **Minkowski Sum**을 통해서 쉽게 범위 질의를 점 질의로 변환해서 범위 질의가 MBR을 접근할 확률을 구할 수 있다. 확률은 ***Minkowski Sum***만큼이 된다. 그림 5에서는 각각 MBR을 차원 S_2와 S_1에 대해서 분할한다. 각 그림의 중심부분의 회색영역은 MBR을 분할하고 분할된 각각에 대한 ***Minkowski Sum***을 구했을 때 겹치는 영역이다. 다시 말해서 MBR을 둘로 분할했을 때 지름이 r인 범위질의가 그 둘을 접근할 확률이 된다. 그림으로부터 쉽게 알 수 있듯이 길이가 긴 차원을 선택해서 분할했을 때 그 확률이 더 작아 진다.

이 때 분할 차원 결정은 활성 차원에 대해서 최대 길이를 갖는 차원을 선택하면 된다. 비활성 차원의 경우 엔트리 모두 같은 값을 갖기 때문에 분할 차원으로 선택될 수 없으며 활성 차원 이후의 나머지 차원들도 고려할 필요가 없다. 비활성 차원과 활성 차원으로 MBR이 구성되므로 활성 차원 이후의 차원들로 데이타들이 분할된다면 MBR간의 겹침이 많이 발생하여 탐색 성능은 떨어질 것이다. 분할 위치는 분할 비율에 따라 결정하게 되는데 비 균등 분할을 원칙으로 한다. 비 균등 분할 비율에 따라 분기율을 l : r로 나누고, 나누어진 데이타 집합이 가질 데이타의 수를 계산한다. 이때 정확한 분할 위치를 결정하는 것이 아니라 UBBT에서처럼 분할 범위를 구하게 된다. 그럼으로써 이 분에 드는 시간을 단축시켜 전체적인 벌크로딩 시간을 단축시킬 수 있다. 분할 범위를 구하는 수식은 수식 6과 수식 7이다. 여기서 N은 현재 분할할 데이타 집합의 데이타 수를 말하며, $N_{max,dataleft}$와 $N_{min,dataleft}$는 분할 후 왼쪽 데이타 집합이 가져야 하는 최대 데이타 수와 최소 데이타 수를 의미한다.

$$N_{max,dataleft} = l \cdot C_{max,tree}(h-1) \qquad \text{(수식 6)}$$

$$N_{min,dataleft} = N - r \cdot C_{max,tree}(h-1) \qquad \text{(수식 7)}$$

이 때 잘못된 분할 범위를 갖는 경우가 발생할 수 있다. $N_{min,dataleft}$가 0보다 작거나 분할 비율에 비해 너무 적은

수를 갖게 되는 경우와 $N_{max,dataleft}$가 N보다 커지는 경우가 생길 수 있다. 이것은 높이와 분기율을 구할 때는 저장공간 활용률을 곱한값으로 계산을 하고, 실제 분할 범위 계산시에는 저장공간 활용률을 고려하지 않기 때문에 발생한다. 이를 해결하기 위해 $N_{min,dataleft}$가 어느 한도 이하의 수를 갖게 되면 왼쪽 데이타 집합이 가질 수 있는 최소 개수는 부 트리가 최소한으로 가져야 하는 $C_{min,tree}$를 구하여 계산한다. $C_{min,tree}$는 단말노드와 색인 노드 모두 각 노드가 ***MinFill***을 만족하는 것으로 생각하여 구한다. ***MinFill***은 노드가 보유해야 할 최소한의 엔트리 수를 의미한다. $N_{max,dataleft}$가 N보다 커지는 경우와 같이 어느 한도 이상을 넘어서는 경우에는 오른쪽 데이타 집합이 가져야 하는 최소한의 데이타 수를 계산하고 나머지를 $N_{max,dataleft}$로 한다.

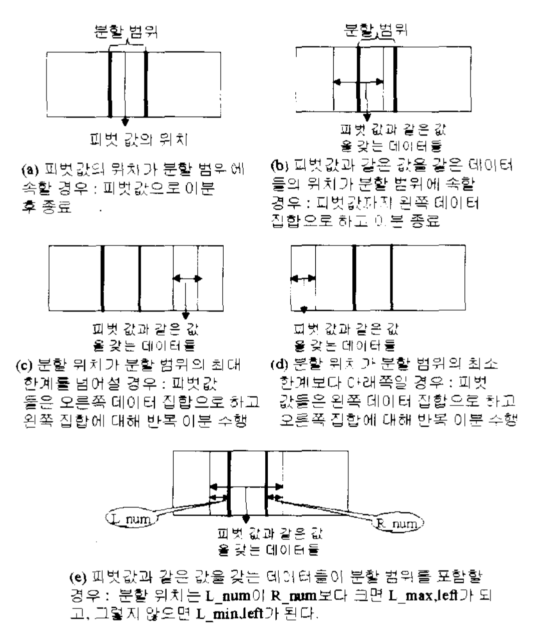
나) 이분(Bisection)

데이타의 이분 과정은 높이 우선으로 수행되며 같은 레벨일 경우 왼쪽 부 데이타 집합에 대해 먼저 반복적인 이분을 수행한다. 전 단계에서 구한 트리의 형태와 분할 전략을 바탕으로 주어진 데이타 집합을 선택된 차원으로 이분을 하게 되는데, 이때 외부 이분과정과 내부 이분 과정으로 나눌 수가 있다. 빠른 이분 수행을 위해 일정량의 메모리를 사용하게 되는데, 데이타들이 메모리로 모두 올라올 수 있는지 여부에 따라서 내부 이분인지 외부 이분인지 나뉘게 된다.

내부 이분은 이분할 데이타 집합이 메모리에 올라올 수 있는 경우를 말한다. 피벗값은 그 중 일부의 데이타를 표본 검출하여 정렬한 뒤 분할 비율을 적용하여 결정하게 되는데, 실제 데이타 중의 하나가 피벗값이 된다. 피벗값이 결정되면 데이타 집합은 피벗값보다 작은 데이타 집합과 크거나 같은 값을 갖는 데이타 집합으로 나뉘게 된다. 만약 분할 위치가 분할 범위에 들어온다면 이분 과정을 종료하고 그렇지 않으면 반복해서 이분 과정을 수행한다. 다시 수행할 때는 원래 이분하고자 하는 데이타 집합 전체에 대해 다시 수행하는 것이 아니고 분할 범위를 포함하는 쪽에 대해서만 다시 수행을 한다.

만약 데이타 집합이 메모리에 올라오지 못한다면 외부 이분을 수행한다. 외부 이분은 메모리에 올라올 수 있을 만큼의 데이타를 표본 추출하여 메모리로 로드한다. 메모리로 로드된 데이타에 대해 내부 이분을 수행하여 피벗값을 결정하고 메모리 내의 데이타를 이분한다.

내부이분 또는 외부이분을 수행하는 도중에 그림 3과 그림 4와 같은 경우처럼 정상적인 이분을 수행하지 못하고 무한 루프에 빠지는 경우가 발생할 수 있다. 기존의 UBBT 알고리즘에서는 이에 대한 고려를 전혀 하고 있지 않다. 이를 해결하기 위한 방법으로 분할 범위내의 임의의 지점에서 이분하는 방법이 있을 수 있다. 하지만, 전체 데이타를 정렬하지 않고 이분을 수행하기 때문에 임의의 지점으로 나눈다면 겹침 현상이 발생할 수 있으므로 좋은 방법이라고 할 수 없다. 본 논문에서 제안하는 알고리즘에서는 피벗값과 같은 값을 가지는 데이타들을 별도로 관리하고 만약 이런 데이타들의 위치가 분할 범위내에 들어온다면 분할 범위내의 한 지점에서 나누도록 한다.



(a) 피벗값의 위치가 분할 범우에 속할 경우 : 피벗값으로 이분 후 종료

(b) 피벗값과 같은 값을 갖은 데이터들의 위치가 분할 범위에 속할 경우 : 피벗값까지 왼쪽 데이터 집합으로 하고 이분 종료

(c) 분할 위치가 분할 범위의 최대 한계를 넘어설 경우 : 피벗값들은 오른쪽 데이터 집합으로 하고 왼쪽 집합에 대해 반복 이분 수행

(d) 분할 위치가 분할 범위의 최소 한계보다 아래쪽일 경우 : 피벗 값들은 왼쪽 데이터 집합으로 하고 오른쪽 집합에 대해 반복 이분 수행

(e) 피벗값과 같은 값을 갖는 데이터들이 분할 범위를 포함할 경우 : 분할 위치는 **L_num**이 **R_num**보다 크면 **L_max,left**가 되고, 그렇지 않으면 **L_min,left**가 된다.

그림 6 이분 수행 방법

피벗값과 같은 값을 갖는 데이타들을 별도로 관리함으로써 그림 6에서처럼 이분 시 발생할 수 있는 여러 가지 상황에 대해 능동적으로 대처할 수 있다. 먼저 (a)와 같은 경우 한번의 이분 수행으로 데이타 집합을 이분할 수 있다. (b)의 경우에도 피벗값과 같은 값을 갖는 데이타들을 별도로 관리하므로 한번의 이분 수행으로 둘로 나눌 수 있다. (c)와 (d)의 경우에는 다시 한번 수행을 해야 한다. (c)의 경우에는 왼쪽 데이타 집합에 대해 반복 이분 수행하고, (d)의 경우에는 오른쪽 데이타 집합에 대해 반복 이분을 수행하게 된다. 만약 이분 수행 후 분할 범위 내에서 이분이 가능하다면 그때 이분에 사용했던 값이 실제 피벗값이 되며, 이를 기준으로 두 개의 부 데이타 집합을 생성한다. (e)의 경우는 피벗값과 같은 값을 갖는 데이타들의 위치가 분할 범위를 포함하는 경우이다. 이 때에는 (e)에서 나타낸

것처럼 ***L_num***과 ***R_num***을 구한다. ***L_num***은 피벗값과 같은 값을 갖는 데이타 중 왼쪽 데이타 집합에 속한 데이타의 수로 분할 범위의 최소값에서 피벗값보다 작은 값을 갖는 데이타의 수를 뺀 값이 된다. 반대로 ***R_num***은 피벗값과 같은 값을 갖는 데이타들 중 오른쪽 데이타 집합에 속한 데이타 수를 의미한다. 만약 ***L_num***이 ***R_num***보다 크다면 분할 위치는 분할 범위의 최대값이 되고, 그렇지 않으면 최소값이 된다. 분할 위치가 결정이 되면, 오른쪽 데이타 집합으로부터 왼쪽 데이타 집합으로 결정된 개수만큼 이동시킨 후 이분을 종료한다.

분할 전략 수립과 이분을 통해 데이타 집합을 둘로 나누는 과정을 왼쪽 부 데이타 집합에 대해 반복 수행하고, 만약 부 데이타 집합이 갖는 분기 수가 하나면 하위 레벨로 내려가 분할을 수행한다. 그렇지 않으면 다시 왼쪽 부 데이타 집합에 대해서 반복 이분 과정을 수행한다. 이 때 부 노드의 높이가 1이라면 단말 노드를 생성한다. 단말노드를 생성하게 되면 그 정보를 상위 노드에 반영하게 되는데, 이 때 각 단말 노드마다 발생하는 비활성 차원의 수가 다를 수 있다. 각 단말 노드에 대한 엔트리들로부터 상위노드를 구성하였을 경우 상위 노드가 갖는 노드의 비활성 차원수보다 많은 비활성 차원을 갖는 엔트리들이 생길 수 있다. 따라서 색인 노드는 기존에 트리 형태 계산 시 결정된 분기 수보다 적은 엔트리만을 수용할 수 있다. 그렇게 되면 단말 노드의 엔트리들을 상위 노드에 반영할 시 넘침 현상이 발생할 수 있다. 또한 색인 노드들로부터 상위 레벨의 색인 노드를 구성할 때 역시 넘침 현상이 발생할 수 있다. 노드 분할을 통하여 넘침 현상을 해결할 수 있지만, 겹침이 없는 분할을 위해서는 많은 비용이 든다. 이것은 데이타 집합 이분 시에 비 균등 분할을 수행하기 때문으로, 넘침 현상이 발생했을 경우 색인의 효율을 위해 각 노드의 ***Min Fill***을 만족하면서 노드간의 겹침이 없는 분할 차원과 위치를 찾아야 한다. 하지만 그런 분할 차원과 위치를 찾기 위해서는 많은 비용이 들며, 그로 인해 색인 구성 시간이 증가할 수 있다. 본 논문에서는 이를 해결하기 위한 방법으로 데이타 집합의 이분 시에 차등 분할 비율을 사용한다.

그림 7에서처럼 한 색인 노드로 구성될 데이타 집합에 대해 처음 이분 시에는 ***Minfill***을 만족하는 범위내에서 비균등 분할을 하고 이때 사용된 분할 차원과 위치를 별도로 관리한다. 이후의 이분은 주어진 비 균등 분할 비율을 적용한다. 이 경우 색인 생성 시 넘침이 발생하면 ***MinFill***을 만족하면서 겹침이 없는 분할을 수행할 수 있다. 하지만 어느 한쪽에 비활성 차원이 많이 생긴다면, 비활성 차원이 많이 생긴 쪽은 여전히 넘침이 발생할 수 있다. 이 경우에는 수퍼 노드로의 확장을 통하여 해결한다.

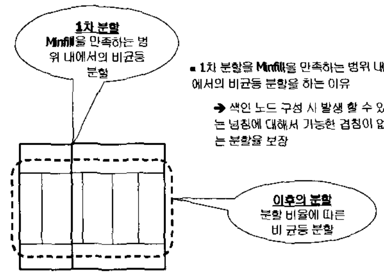


그림 7 차등 이분법

3.2.3 색인 노드(비 단말 노드) 생성

데이타 집합을 부 데이타 집합으로 반복 이분하면 단말 노드를 생성할 수 있다. 이렇게 단말 노드가 생성되면 그 정보를 상위 노드에 반영하여 루트 노드까지 생성하는 상향식 색인 구성 방식을 따른다.

반복적인 이분을 통해 최초의 데이타 집합은 여러 개의 부 데이타 집합으로 나뉘며, 각 부 데이타 집합은 다음 레벨에서 또 다시 여러 개의 부 데이타 집합으로 나뉜다. 어떤 부 데이타 집합의 높이가 1이되면 그 부 데이타 집합으로부터 단말 노드를 생성하고, 생성된 단말 노드의 영역정보로부터 엔트리를 구성하여 상위 노드에 반영한다. 이렇게 해서 높이가 1인 부 데이타 집합들로 단말 노드들을 구성하고 이로부터 색인 노드와 루트 노드를 차례로 완성하면 모든 색인 생성 과정을 모두 마치게 된다.

단말 노드로부터 MBR을 구성하고 이 엔트리들을 상위 노드에 반영할 때 넘침 현상이 발생할 수 있다. 이런 넘침 현상은 트리의 형태 결정 시에 모든 엔트리의 비활성 차원을 고려한 것이 아니므로 예기치 못한 비활성 차원의 증가로 인한 것이다. 넘침 현상이 발생하면 노드 분할과 수퍼 노드로의 확장을 통해서 해결한다. 먼저 노드 분할을 시도하게 되는데, 앞서 이를 위해 그림 7에서 데이타 집합 분할 시에 차등 분할 비율을 적용하였다. 데이타 집합을 부 데이타 집합으로 처음 분할할 때 적용된 분할 차원과 피벗값에 따라 넘침 현상이 발생한 노드에 대해 분할을 시도한다. 만약 노드 분할이 성공하면 분할된 노드와 새로 생성된 노드 모두를 상위에 반영하게 되지만, 그렇지 못할 경우에는 수퍼 노드로 확장하고 그에 대한 영역정보만을 상위 노드에 반영한다.

지금까지 본 논문에서 제안하는 벌크로딩의 알고리즘에 대해 수행 순서에 따라서 설명을 하였다. 그림 8은 벌크로딩 알고리즘에 따라 수행되어 색인을 구성한 예이다. 데이타 집합을 반복 이분하여 단말 노드를 생성한 후 그로부터 색인 노드를 생성하여 색인을 완성하게 된다. 여기서 두 번

째 레벨의 가장 오른쪽 노드를 구성할 때 넘침이 발생하였고, 분할을 통하여 해결하였다.

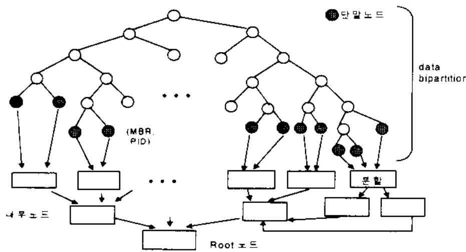


그림 8 색인 구조 생성의 예

3.2.4 의사 코드

각 모듈별로 수행 과정에 대한 의사 코드를 나타내었으며 각각을 간략히 설명한다.

가) CreateIndex()

베이스 파일로부터 특징 벡터를 추출하여 정수형으로 변환한 뒤 임시 화일을 생성하고 그것을 기초로 하여 CIR-트리 색인을 생성한다.

그림 9의 1행에서는 먼저 색인을 구성할 데이타의 수를 파악한다. 만약 데이타가 하나도 없다면 단지 빈 색인을 생성하게 된다. 그렇지 않으면 주어진 데이타 집합으로 제안하는 벌크로딩 알고리즘을 통해 색인을 구성하게 된다. 먼저 5행에서 효과적인 내부 이분을 수행하기 위해 필요한 메모리를 할당한다. 그런 후 주어진 데이타 집합으로부터 생성될 수 있는 색인의 높이를 계산하고, ConstructIndexNode( )를 호출하여 색인을 생성하게 된다.

```
CreateIndex (dataset)
1 : if ( 데이타 수 == 0 )
2 :     빈 색인 생성;
3 : else
4 : {
5 :     내부 이분을 위한 메모리 할당;
6 :     생성될 색인의 height 계산 ;
7 :     ConstructIndexNode(dataset, h, rootnode) ;
8 : }
9 : return ;
```

그림 9 CreateIndex 모듈의 의사코드

나) ConstructIndexNode()

실제 CIR-트리의 노드를 생성하는 모듈로서 만약 높이가 1이라면 단말 노드를 생성하여 상위 노드에 반영하고, 그렇지 않으면 분기율을 계산하여 SubDataPartition( )을 호출한다.

```
ConstructIndexNode (dataset, h, parentnode)
1 : If ( 주어진 데이타집합으로 하나의 단말 노드 생성
    가능)
2 : {
3 :     단말 노드를 생성 ;
4 :     leafmbr = 단말 노드의 MBR을 구성;
5 :     if ( leafmbr을 상위 노드(parentnode)에 삽입시
        오버플로 발생 )
6 :         수퍼 노드에 단말 노드 MBR 삽입 ;
7 :     else
8 :         단말 노드의 MBR을 parentnode에 삽입;
9 :     return ;
10 : }
11 : fanout 수 계산;
12 : nonleaf = 비단말 노드 생성 ;
13 : nonleaf 노드 구성 = SubDataPartition(dataset,
     nonleaf,h,fanout) ;
14 : if(nonleaf 노드에 오버플로 )
15 : {
16 :         분할이나 수퍼노드 생성으로 해결;
17 : }
18 : else nonleaf의 MBR을 구성하여 parentnode에 삽
입;
19 : return ;
```

그림 10 ConstructIndexNode 모듈의 의사코드

그림 10의 1~10행은 높이가 1일 때로 주어진 데이타 집합으로부터 하나의 단말 노드를 생성하는 경우이다. 이 경우에는 먼저 단말 노드 하나를 할당받고 주어진 데이타 집합으로부터 엔트리들을 단말 노드에 삽입한다. 단말 노드가 생성이 되면 이에 대한 MBR을 구하여 비단말 노드 엔트리 형태로 구성하며. 구성된 단말 노드의 MBR을 상위 노드에 삽입한다. 이 때 상위 노드에 삽입할 공간이 있는지 검사하여 삽입할 공간이 있으면 삽입하고 아니면, 수퍼 노드에 단말 노드의 MBR을 삽입한다. 단말 노드를 생성하고 그것의 MBR을 상위 노드에 삽입하였다면 모듈의 수행을 종료한다.

높이가 1이 아니면 하나의 비단말 노드를 생성하는 단계로 11행 이후의 연산을 수행하게 된다. 11행에서 주어진 데이타 집합으로 현재 레벨의 노드에서 가질 수 있는 분기율을 구한 후, 12행에서 비단말 노드를 생성하여 13행의 SubDataPartition을 호출하게 된다. 이를 통해 완성된 비단말 노드가 만약 넘침이 발생하면 분할이나 수퍼 노드로의 확장을 통해 해결한 후 상위 레벨에 반영한다. 그렇지 않다면 정상적으로 비단말 노드에 대한 MBR을 구성하여 상위 레벨의 노드에 삽입한다.

다) SubDataPartition()

주어진 데이타 집합을 반복해서 이분하여 분기 수만큼의 서브 데이타 집합으로 나눈다.

```
SubDataPartition (dataset, parentnode, h, fanout)

1 :  분할 차원 = 활성 차원 중 최대 길이를 갖는 차원 ;
2 :  if ( 최초 분할 )
3 :      MinFill을 만족하는 범위에서 비균등 분할 수행
;
4 :  else
5 :      비균등 분할을 수행한다.
6 :  splitinterval = 분할 범위 계산 ;
7 :  If (잘못된 분할 범위)
8 :  {
9 :      splitinterval = Minfill을 고려하여 분할 범위
         재계산;
10 : }
11 : 두 개의 데이타 집합 생성(ldataset, rdataset) =
     BiSection(dataset, splitinterval) ;
12 : if ( ldataset의 fanout == 1 )
13 :     cir_ConstructIndexNode(ldataset, h, parentnode)
;
14 : else
15 :     cir_SubDataPartition(ldataset, parentnode, h,
         left _fanout) ;
16 : if (rdataset의 fanout == 1 )
17 :     cir_ConstructIndexNode(rdataset, h,
         parentnode) ;.
18 : else
19 :     cir_SubDataPartition(rdataset, parentnode, h,
         right_fanout) ;
20 : return ;
```

그림 11 SubDataPartition 모듈의 의사코드

그림 11의 1행은 분할 차원을 선택하는 부분이다. 활성 차원 중 최대 길이를 갖는 차원을 분할 차원으로 선택한다. 2~5행은 분할 비율에 따라 분할 위치를 결정하는 부분이다. 앞서 예기치 못한 넘침을 해결하기 위하여 차등 분할 비율을 적용한다고 했는데 이 부분이 그 역할을 한다. 만약 현재 레벨의 노드에 대해 최초 분할이면 Minfill을 고려하여 분할 비율을 적용하고, 최초 분할이 아니면 비균등 분할 비율을 적용한다. 분할 비율에 따라 분할할 위치가 결정이 되면 이분의 수행 시간을 줄이기 위해 6~10행의 수행을 거치게 된다. 그런 후 11행의 cir_BiSection을 통해 데이타 집합을 둘로 나눈다. 12~19행을 통해 왼쪽, 오른쪽 데이타 집합에 대해 각각 만약 분기 수가 1이면 Construct IndexNode를 호출하고 그렇지 않으면 SubDataPartition을 재귀 호출한다.

라) BiSection()

주어진 데이타 집합이 메모리에 올라올 수 있거나 이미 올라와 있으면 내부 이분을 수행하고, 그렇지 않으면 외부 이분을 수행하여 데이타 집합을 둘로 분할한다.

그림 12의 1~4행은 파일로부터 메모리로 데이타를 로드하는 부분이다. 가능하다면 모두 로드하고 그렇지 않으면 일부만 표본 검출을 하게 된다. 데이타들이 이미 모두 메모리에 있다면 바로 다음의 5행을 수행한다. 5행의 InternalBiSection은 메모리내의 데이타들 중 하나를 피벗값으로 선택하여 내부 이분을 수행하는 모듈이다. Internal BiSection을 호출할 때 메모리 내의 데이타들에 대해서 분할 범위를 재계산하게 되는데, 이것은 전체 데이타 중 일부의 데이타만이 메모리에 있을 수 있기 때문이다. 만약 외부 이분이면 6~11행을 통해 나머지 데이타들에 대해 이분을 수행하여 파일에 기록한다. 12~23행은 그림 6의 각각의 경우에 따라 수행을 하여 이분을 수행하는 과정을 설명한 것으로 12행은 그림 6의 (a)에 해당하게 되며, 14~18행은 (b)와 (e)의 경우에 해당된다. 19행은 (c), 21행은 (d)의 경우를 고려한 것이다

```
BiSection (dataset,분할 범위)

1 : if( 메모리에 로드 가능 )
2 :      mdataset = 파일로부터 메모리로 로드 ;
3 : else if( 외부 이분 수행 )
4 :      mdataset = 메모리로 표본 검출 ;
5 : 내부 이분 수행 = InternalBiSection(mdataset,
    m_splitinterval) ;
6 : if ( 외부 이분 )
7 : {
8 :      내부 이분 결과를 파일에 작성 ;
9 :      나머지 데이타들에 대해 피벗값을 기준으로
         이분.
10 :     피벗값과 같은 값을 갖는 데이타들은 별도 관
         리 ;
11 : }
12 : if( 피벗값의 위치가 분할 범위에 포함 )
13 :    return ;
14 : if ( 피벗값과 같은 값을 갖는 데이타들이 분할 범
     위에 위치)
15 : {
16 :    일부 데이타들을 왼쪽 데이타 집합으로 이동 ;
17 :      retrun ;
18 : }
19 : if ( 피벗값의 위치 > 분할 범위의 최대값 )
20 :      피벗값과 같은 값을 갖는 데이타들은 오른쪽
         데이타 집합 ;
21 : else if ( 피벗값의 위치 < 분할 범위의 최소값 )
22 :      피벗값과 같은 값을 갖는 데이타들은 왼쪽으
         로 이동;
23 : 분할 범위를 포함하는 쪽에 대해 BiSection()을 반
     복 호출 ;
24 :      return ;
```

그림 12 BiSection 모듈의 의사코드

마) InternalBiSection

메모리에 있는 데이타들에 대해 피벗값을 구하고 피벗값에 따라 둘로 나누는 모듈이다.

그림 13의 1행에서는 먼저 이분의 기준이 되는 피벗값을 선택하게 되는데, 이때 분할 비율을 고려하여 실제 데이타 중의 하나로 선택한다. 2행에서 앞서 선택된 피벗값을 기준으로 이분을 수행하며, 4~10행은 이분의 종료조건을 검사하는 부분으로 이 역시 그림 6의 내용을 따르게 된다. 내부 이분 역시 항상 분할범위 내에서 정상적인 분할이 수행되도록 보장한다.

```
InternalBiSection (mdataset, m_splitinterval)

1 : pivot = 피벗값을 검출 ;
2 : 피벗값을 기준으로 이분 수행 ;
3 : 피벗값과 같은 데이타들은 별도로 관리 ;
4 : if ( 피벗값의 위치가 분할 범위내 위치)
5 :     return ;
6 : else if (피벗값과 같은 데이타들의 위치가 분할 범위
    내에 위치)
7 : {
8 :     피벗값과 같은 값을 갖는 데이타들을 일부 왼쪽
    으로 이동
9 :         return ;
10 : }
11 : 분할 범위를 포함하는 쪽에 대해 InternalBiSection()
수행
12 : return ;
```

그림 13 InternalBiSection 모듈의 의사코드

4. 구현 및 성능 평가

CIR-트리를 위한 벌크로딩 알고리즘은 바다-Ⅲ의 하부 저장 시스템인 MIDAS-Ⅲ에서 구현하여 다양한 성능 평가를 수행하였다.

4.1 구현 환경

구현에 사용된 시스템은 Solaris 2.7.x 운영 체제에 Sun Enterprise 250이며 메인 메모리의 크기는 1GByte로 벌크로딩 수행 시 내부 이분을 수행하기 위해 충분한 메모리를 획득할 수 있었다. 구현은 MiDAS-Ⅲ상에서 이루어졌으며 컴파일러는 gcc 2.8이다. 성능 평가에 사용된 데이타 집합은 10 차원의 100 만개의 실수형 균등 분포 데이타 집합과 9차원의 30만개 동영상 데이타 집합, 100 차원의 정규 분포 30 만개 데이타 집합이다. 제안한 알고리즘을 구현하여 성능 평가를 하기 위해 고려한 파라미터들은 표 1과 같다.

표 1 성능평가를 위한 파라미터들

factor	value
전체 차원 수 (차원)	9, 10, 20, 100, 120
활성 차원 수 (%)	30 ~ 40
페이지 크기 (KByte)	4, 8, 16
저장 공간 활용률 (%)	70 ~ 80
first partition-rate	1 : 1
second partition-rate	1 : 5
메모리 크기 (KByte)	320 ~ 1,920
데이타 수 (만개)	1400(개), 10 ~ 100

디스크 페이지의 크기는 MiDAS-III의 페이지 크기인 16 KByte를 기본으로 하되 일반 DBMS의 기본 페이지 크기인 4KByte, 8KByte에 대해서도 성능 평가를 수행하였다. 페이지를 4KByte, 8KByte로 하였을 경우에도 16 KByte로 실험을 수행한 결과와 유사하기 때문에 지면 관계상 페이지 크기가 16KByte인 경우의 성능평가 결과를 중심으로 실험 내용을 기술한다. 내부 이분을 수행하기 위해 사용되는 메모리의 크기는 320 KByte~1,920KByte로 변화시켜 성능 평가를 하였다. 앞서 그림 7에서 넘침 현상을 해결하기 위해 차등 분할 비율을 적용하였는데, 표 1에서 보듯이 1 : 1 과 1 : 5의 차등 비균등 분할 비율을 적용하였다. 또한 데이타의 수를 10만개에서 100 만개로 변화시키면서 성능 평가를 하였다.

4.2 성능 평가

성능 평가는 메모리 크기에 따른 색인 구축 시간, 개별 삽입과의 색인 구성 시간과 탐색 성능 비교, 그리고 UBBT와의 탐색 성능 비교 관점에서 하였다. 그러나 UBBT기법이 CIR-트리와 같이 색인 노드 구성 시 일부 차원만을 사용하는 고차원 색인 구조에 적합하지 않고, 제안하는 방법이 MiDAS-Ⅲ라는 환경에서 구현되었기 때문에, 색인 구성 시간 비교는 하지 않고 10개의 최근접 질의에 대한 탐색 성능만을 비교하였다

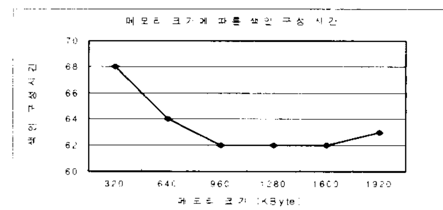


그림 14 메모리 크기에 따른 색인 구성 시간

먼저 내부 이분을 수행하기 위한 메모리의 크기가 색인 구성에 미치는 영향에 대하여 성능 평가를 하였다. 그림 14는 전체 차원과 활성 차원 모두 10 차원인 균등 분포 데이타 10만개에 대하여 내부 이분 시 사용되는 메모리 크기에 따른 색인의 구성 시간을 측정한 것이다. 그림에서 보듯이 메모리의 크기가 960KByte로 늘어날 때까지 색인의 구성 시간이 조금씩 단축되는 것을 알 수 있다. 그러나, 960 KByte이상의 크기에서는 내부 이분을 위해 필요한 메모리의 크기 증가로 인한 효과가 없음을 알 수 있다. 메모리가 전체 데이타 크기의 25 %일 때가 가장 우수한 성능을 나타내고 있다.

그림 15는 벌크로딩을 통한 CIR-트리의 구축과 개별 삽입을 통한 구축을 비교한 것이다. 전체 차원이 10이고 활성 차원이 5인 데이타 집합을 10만개에서 60만개로 변화시키면서 색인 구성 시간을 측정한 것이다. 그림에서 보듯이 개별 삽입보다 벌크로딩을 통한 색인의 구성 시간이 현저하게 줄어드는 것을 알 수 있다. 개별 삽입의 경우 루트 페이지부터 삽입할 해당 단말 페이지까지 디스크를 접근하

여야 하며, 이를 매번 삽입 때마다 수행하여야 하기 때문에 대량의 데이타로부터 색인을 구축할 경우 많은 시간이 걸리게 된다. 이 때문에 벌크로딩 기법을 통한 색인 구축이 개별삽입보다 더 좋은 성능을 보이는 것이다.

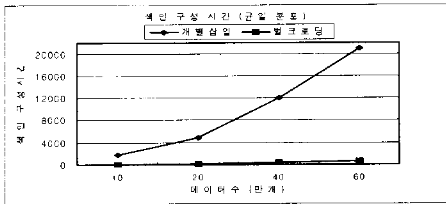


그림 15 색인 구성 시간

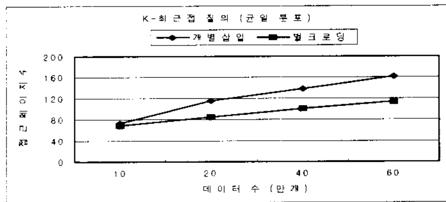


그림 16 K-최근접 질의(K=10)

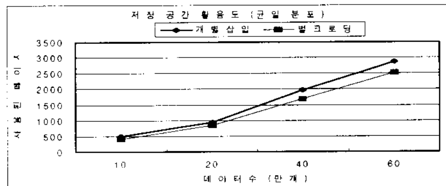


그림 17 저장 공간 활용도

그림 16은 그림 15에서 구축된 색인에 대하여 K-최근접 질의를 수행한 결과이다. 10개의 최근접 질의로 성능 평가를 하였으며 성능 평가의 척도는 접근하는 페이지(노드) 수이다. 그림 16에서 보듯이 개별 삽입을 통해 구축된 색인보다 K-최근접 질의에 대해 10~30% 정도의 성능 개선이 있음을 알 수 있다. 이것은 색인 구축 시 생성되는 페이지들간의 겹침이 발생하지 않고, 유사한 데이타들은 같은 페이지에 들어갈 확률이 높아졌기 때문이다. 또한 그림 17에서 보듯이 저장 공간활용도도 개별 삽입보다 높기 때문이다.

두 번째 데이타 집합인 9차원의 실제 동영상 데이타 집합에 대해서도 위와 같은 실험을 하였다. 먼저 벌크로딩 알고리즘으로 구축할 경우 저장 공간의 절약 효과를 알아보기 위해 CIR-트리 구축에 사용된 페이지 수를 측정하였다.

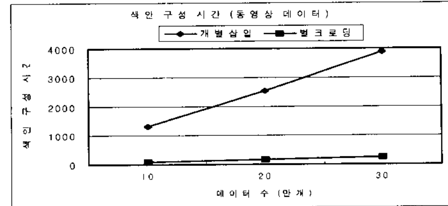


그림 18 색인 구성 시간

그림 18은 전체 9차원의 실제 동영상 데이타 집합에 대해 활성 차원 수를 5로 하고 데이타 수를 변화시켜 색인 구성 시간에 대한 성능 평가를 한 결과이다. 앞서 균등 분포 데이타의 실험과 같이 벌크로딩으로 색인을 구성할 경우 구성 시간 측면에서 탁월한 성능을 보였다. 그림 18에서 구성한 색인에 대한 K-최근접 질의를 수행한 결과가 그림 19이다. 이것 역시 균등 분포 데이타 집합에 대한 탐색 성능 평가와 같은 결과를 보였다. K-최근접 질의 수행 시 벌크로딩으로 색인을 구성하였을 경우 18~40 % 정도 페이지를 덜 접근함을 알 수 있다. 색인 구성 시 그림 20에서와 같이 벌크로딩 알고리즘이 개별 삽입에 비해 약 16 %정도의 저장 공간의 절약 효과가 있음을 알 수 있다. 이것은 저장공간 활용률을 80%로 보장하여 색인에 사용된 페이지들이 불필요하게 낭비되는 현상을 막기 때문이다.

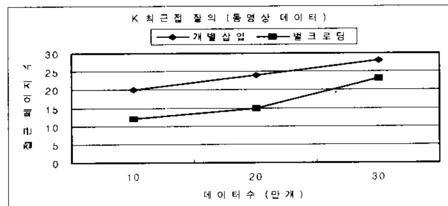


그림 19 K-최근접 질의(K=10)

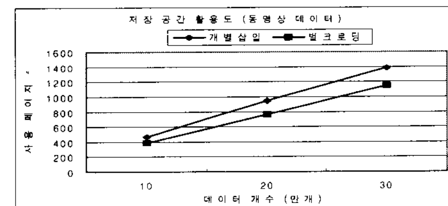


그림 20 저장 공간 활용도

그림 21은 120 차원의 1400 개 실제 이미지 데이타에 대한 질의 결과이다. 이때 활성 차원의 수는 40으로 하였

다. 그림 21의 성능 평가 결과에서 보듯이 최근접 질의와 범위 질의 모두 벌크로딩으로 색인을 구축하였을 경우 페이지 접근 수를 감소시킬 수 있었다.

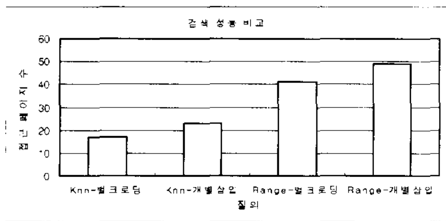


그림 21 검색 성능 평가

지금까지의 성능 평가에서 보듯이 실험 결과 개별 삽입에 비해 탁월한 성능을 보임을 알 수 있다. 먼저 저장공간의 사용량을 16% 정도 절감할 수 있었으며, 색인의 구성 시간 측면에서도 개별 삽입에 비해 탁월한 성능을 보였다. K-최근접 질의 수행 시 페이지 접근수가 균등분포의 경우 30%, 동영상 데이타의 경우 18~40%정도 덜 접근함을 알 수 있다. 이는 저장 공간 활용도를 어느 정도로 보장해 주었으며, 개선된 이분법을 적용하여 색인 구성 시간을 단축시킬 수 있었기 때문이다. 또한 차등 비균등 분할 비율을 적용하여 색인 구성을 효율적으로 함으로써 검색 성능을 향상시켰기 때문이다.

다음으로는 X-트리에 적용한 UBBT와의 성능 비교이다. 여기서는 K-최근접 질의에 대한 성능비교를 수행하였다. 제안하는 알고리즘이 MiDAS-Ⅲ라는 특수한 환경에서 구현되었고, UBBT는 UNIX의 파일 기반으로 X-트리에 구현되어 있기 때문에 색인 구성 시간 측면에서는 비교를 하지 않았다. K=10일 경우 최근접 질의에 대한 성능 비교 결과가 그림 22에 나타나 있다.

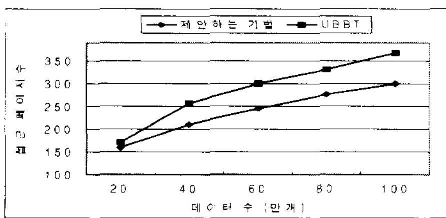


그림 22 UBBT와의 KNN 성능 비교(K=10)

색인 구성에 사용된 데이타 집합은 10차원의 정규분포 데이타 100만개이다. CIR-트리는 전체 차원의 수는 10차원이며, 활성 차원의 수는 4차원으로 하였다. 성능 평가는 데이타를 20만개에서 100만개까지 변화시키면서 K=10일 경우 최근접 질의 수행 시 접근하는 페이지 수로 하였다. 그림 22에서 보듯이 평균적으로 15%정도 제안하는 방법이 좋은 성능을 나타내고 있으며 데이타 수가 많은 경우에는 최고 약 20%정도 우수한 성능을 보이고 있다. 이것은 CIR-트리를 위한 벌크로딩 알고리즘이 차등 분할 비율의 적용과 데이타 집합 분할 시 피벗값과 같은 값을 갖는 데이타들을 별도로 관리함으로써 겹침이 없는 색인을 구성할 수 있게 보장하여 검색 성능을 향상시켰기 때문이다. 또한 분할 범위 결정 시 색인이 불균형하게 구성되는 점을 방지하여 UBBT가 갖는 문제를 해결하였기 때문이다.

마지막으로 100차원의 정규분포를 갖는 30만개 데이타 집합에 대해서 성능 평가를 수행하였으며, 지금까지의 실험에서처럼 100 차원의 데이타에 대한 성능 평가 역시 색인의 구성 시간 측면에서 개별 삽입보다 탁월한 성능을 보였으며 UBBT보다 우수한 질의 성능을 나타내었다.

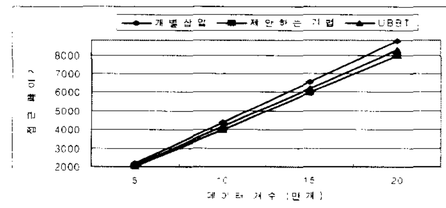


그림 23 K-최근접 질의(K=10)

그림 23은 100차원의 데이타에 대한 색인 구축 후 K-최근접 질의를 수행한 결과이다. CIR-트리는 전체 차원과 활성 차원을 각각 100, 40으로 하였으며, 10개의 최근접 질의를 수행하여 접근하는 페이지 수를 측정하였다. 그림 23에서 위에서부터 차례로 개별삽입, UBBT, 제안하는 기법을 통한 색인 구축 후 검색 성능 결과를 나타낸다. 성능 평가 결과 각 기법 모두 색인을 구성하고 있는 페이지 중 많은 페이지를 접근하는 문제점을 나타내었다. 이것은 고차원 색인 구조의 근본적인 문제점인 차원의 저주 현상 때문으로 차원이 증가할수록 검색 성능이 저하되는 결과를 보였다. 그러나 그림에서처럼 제안하는 기법이 가장 우수한 성능을 나타냈다.

5. 결 론

이 논문에서는 CIR-트리를 위한 벌크로딩 알고리즘을 제안하고 구현하였다. 그동안 고차원 색인 구조를 위한 벌

크로딩 알고리즘들이 일부 제안되었지만, 색인 노드 구성 시 일부 차원만 사용하는 고차원 색인 구조에 적합하지 못하고 검색 성능 측면이나 색인의 구성 시간 측면에서 좋은 성능을 보이지 못하였다.

제안하는 알고리즘에서는 UBBT 기법을 기반으로 하여 기존 알고리즘의 문제점을 해결하고 CIR-트리에 적합한 벌크로딩 기법을 설계하였다. 이 논문에서 CIR-트리를 위한 벌크로딩 기법을 설계할 때 고려하였던 점들을 네 가지로 요약할 수 있다. 첫 번째, UBBT에서 제안한 트리 형태 계산을 통한 비균등 이분법을 기반으로 한다. UBBT 기법을 기반으로 하며 보다 개선된 그리고 CIR-트리의 특징을 수용하는 벌크로딩 기법을 설계한다. 두 번째, 분할 범위를 계산할 때 부 트리가 가져야 하는 최소 한계 데이타 개수를 고려하여 잘못된 분할 범위가 계산되는 문제점을 해결하였다. 세 번째, 이분을 수행할 때 분할 차원의 값이 피벗 값과 같은 데이타들을 따로 모아서 관리함으로써 분할 범위내에서 이분이 가능하도록 보장한다. 네 번째, CIR-트리가 갖는 가변 MBR 특징을 수용하는 방법을 고려한다. 비 단말 노드에 저장되는 엔트리의 길이가 서로 다를 수 있다는 문제를 해결하기 위해서 각 비 단말 노드의 분기율을 계산할 때 활성차원과 비 활성 차원을 고려한다. 또한 분할 차원의 선택도 활성차원으로 제한하며, 색인 노드를 구성 시 발생할 수 있는 넘침에 대처하기 위해 차등 분할 비율을 적용한다.

제안한 벌크로딩 알고리즘은 바다 DBMS의 하부 저장 시스템인 MiDAS-III에서 구현하고 성능 평가를 하였다. 성능 평가에서 보듯이 개별 삽입보다 색인의 구성 시간, 저장 공간 활용률, 그리고 검색 성능 측면에서 모두 우수함을 나타내었으며 UBBT를 적용한 X-트리와의 탐색 성능 비교에서도 우수함을 타나내었다.

참 고 문 헌

[1] 이석희, 최길성, 유재수, 조기형, "CIR-트리 : 내용기반 이미지 검색을 위한 효율적인 고차원 색인기법", 한국정보과학회 97 가을 학술발표(I), pp. 349-352, 1997

[2] 이석희, 송석일, 유재수, "내용기반 이미지 검색을 위한 고차원 색인구조", 한국정보과학회 데이타베이스 연구회 논문지, 제 14권, 제 4호, pp. 53-68, 1998

[3] Roussopoulos N. and Keifker D., "Direct Spatial Search On Pictorial Databases Using Packed R-Trees," In Proc. ACM SIGMOD, pp. 17-31, 1985

[4] Kamel I. and Falousos C., "On Packing R-Trees," In Proc. CKIM, pp. 490-499, 1993

[5] Leutenegger S. T., Lopez M. A. and Edgington J., "STR : A Simple and Efficient Algorithm for R-Tree Packing," In Proc. ICDE, pp. 497-506, 1997

[6] Garcia Y. J., Lopez M. A. and Leutenegger S. T., "A Greedy Algorithm for Bulk Loading R-Trees," In Proc. ACM GIS, pp. 47-57, 1998

[7] Van Den Bercken J. and Seeger B., Widmayer, "A General Approach to Bulk Loading Mutidimensional Index Structures," In Proc. VLDB, pp. 406-415, 1997

[8] Berchtold S., Böhm C. and Kriegel H. P., "Improving the Query performance of High-Dimensional Index Structure by Bulk Load Operation," In Proc. EDBT, pp. 216-230, 1998

[9] Böhm C. and Kriegel H. P, "Efficient Bulk Loading of Large High-Dimensional Indexes," In Proc. Dawak, pp. 251-260, 1999

피 준 일

1999년 충북대학교 공과대학 컴퓨터 공학과(공학사). 2001년 충북대학교 정보통신공학과 (공학석사). 2002년 충북대학교 정보통신공학과(박사과정 재학 중). 관심분야는 고차원 색인 구조, 데이터 베이스 시스템, 메모리 상주형 데이터 베이스 시스템, 저장 시스템, 실시간 시스템 등

송 석 일

정보과학회논문지 : 데이타베이스 제 29 권 제 1 호 참조

유 재 수

정보과학회논문지 : 데이타베이스 제 29 권 제 1 호 참조