고차원 색인 구조를 위한 효율적인 벌크 로딩

복 경 수[†] · 이 석 희[††] · 조 기 형[†††] · 유 재 수[††††]

요 약

다차원 색인 구조를 위한 기존의 벌크 로딩 알고리즘은 색인 구성 시간과 검색 성능 모두를 향상시키지 못하는 문제점을 갖는다. 이 논문은 이와 같은 문제점을 해결한 대량의 고차원 데이터에 대한 색인 구조를 위한 새로운 벌크 로딩 알고리즘을 제안한다. 제안하는 알고리즘은 색인을 구성하는 시간을 단축시키기 위해 전체 데이터 집합을 정렬하는 것이 아니라 데이터의 특성을 파악하여 피벗 값에 따라 분할하는 기법을 이용한다. 또한 검색 성능을 향상시키기 위해 데이터들의 분포 특성에 따라 분할 위치를 선택한다. 실험을 통해 제안하는 알고리즘이 기존의 알고리즘보다 색인 구성 시간과 검색 성능 측면에서 우수함을 보인다.

An Efficient Bulk Loading for High Dimensional Index Structures

Kyoung-Soo Bok[†] · Seok-Hee Lee[††] · Ki-Hyung Cho[†††] · Jae-Soo Yoo[††††]

ABSTRACT

Existing bulk loading algorithms for multi-dimensional index structures suffer from satisfying both index construction time and retrieval performance. In this paper, we propose an efficient bulk loading algorithm to construct high dimensional index structures for large data set that overcomes the problem. Although several bulk loading algorithms have been proposed for this purpose, none of them improve both construction time and search performance. To improve the construction time, we don't sort whole data set and use bisection algorithm that divides the whole data set or a subset into two partitions according to the specific pivot value. Also, we improve the search performance by selecting split positions according to the distribution properties of the data set. We show that the proposed algorithm is superior to existing algorithms in terms of construction time and search performance through various experiments.

1. 서 론

최근 원격 진료 시스템, 지리 정보 시스템, 이미지 데이터베이스 시스템, 멀티미디어 데이터베이스 시스템 등에서 이미지에 대한 검색을 효과적으로 수행하기 위한 연구가 활발하게 이루어지고 있다. 특히 내용기반 이미지 검색에 대한 관심은 점점 더 증가되고 있다. 내용기반 이미지 검색을 수행하기 위하여 이미지에서 추출되는 특징 벡터들은 고차원의 특징 벡터로 구성된다. 그러므로 기존에 존재하는 B-트리와 같은 텍스트 검색을 위한 색인을 이용하는 것은 많은 어려움이 있다. 이를 해결하기 위해 고차원 특징 벡터를 이용하여 색인을 구성하는 R-트리[1], R*-트리[2], TV-트리[3], X-트리[4]와 같은 고차원 색인 구조들이 제안되었다. 이러한 색인 구조들은 색인을 구성하기 위해서 주어진 데이터가 삽입하게 될 가장 적절한 노드를 탐색하여 해당 노드에 데이터를 하나씩 삽입하면

※ 본 연구는 과학재단 특정기초과제(과제번호 : 1999-1-303-007-3) 연구비지원에 의하여 수행되었음.
† 준 회 원 : 충북대학교 대학원 정보통신공학과
†† 정 회 원 : 동아방송대학 인터넷방송과 교수
††† 정 회 원 : 충북대학교 전기전자공학부 교수
†††† 종신회원 : 충북대학교 정보통신공학과 교수
논문접수 : 1999년 11월 12일, 심사완료 : 2000년 7월 27일

서 색인을 구성한다. 그러나 대량의 데이터가 한꺼번에 주어져 있을 때 데이터를 하나씩 삽입하면서 색인을 구성한다면 색인을 구성하기 위하여 많은 시간이 소요하게 된다. 또한 데이터의 삽입으로 계속적인 분할 과정을 반복하면서 저장 공간 활용률을 저하시키게 되고 이로 인하여 트리의 높이도 증가하게 된다. 특히 고차원 데이터는 분할을 수행하는 과정에서 겹침 영역이 증가하게 되어 색인을 이용한 검색 성능을 저하시키게 된다. 최근 이러한 성능 저하 문제를 해결하면서 대량의 데이터들을 이용하여 효과적인 색인을 구성하기 위한 벌크 로딩 기법들이 제시되었다.

다차원 색인 구조에 대한 최초의 벌크 로딩 알고리즘은 Roussopoulos에 의해 제안된 NX(Nearest-X) 알고리즘[5]이다. NX 알고리즘은 전체 차원 중 특정한 하나의 차원에 대해서만 데이터를 정렬하여 색인을 구성하기 때문에 MBR(Minimum Bounding Region)의 둘레 길이가 증가되어 유사도 검색에 대한 성능을 저하시키는 문제점을 갖고 있다. 이러한 문제점을 해결하고자 Kamel과 Faloutsos은 힐버트 값을 이용하여 데이터를 정렬하여 색인을 구성하는 HS(Hilbert Sorting) 알고리즘[6]을 제안하였고, Leutnengger와 Lopez 등은 특정 차원에 대해 데이터를 정렬하여 분할을 수행한 후 또 다른 차원에 대해 정렬을 수행하여 분할하는 과정을 반복하면서 색인을 구성하는 STR(Sort Tile Recursive), TGS(Top down Greedy Split) 알고리즘[7, 8]을 제안하였다. 그러나 이러한 알고리즘들은 전체 데이터 집합을 정렬하여 색인을 구성하기 때문에 저장공간 활용률은 증가시킬 수 있으나 색인을 구성하기 위한 시간이 많이 소요된다. 그러나 Bercken와 Seeger[9] 등은 Arge에 의해 제안된 버퍼 트리[10]라는 개념을 이용하여 주어진 전체 데이터를 정렬하지 않고 색인을 구성하는 벌크 로딩 알고리즘을 제안하였다. 또한 Berchtold, Bohm, Kriegal [11]등은 전체 데이터 집합을 정렬하는 것이 아니라 변형된 퀵 정렬을 이용하여 분할을 수행할 데이터들이 특정 분할 비율을 만족하면 데이터를 직접 양분하면서 색인을 구성하는 알고리즘을 제안하였다. 버퍼 트리을 이용하는 [9]와 변형된 퀵 정렬을 이용하여 데이터를 양분하는 [11]과 같은 알고리즘들은 전체 데이터 집합을 정렬하지 않기 때문에 STR, TGS와 같은 알고리즘에 비해 색인 구성 시간이 단축된다. 그러나 데이터를 정렬하지 않고 색인을 구성하기 때문에 데이터의 특성을 제대로 고려하지 못하여 전체 데이터 영역의 일부에 군집화가 되어 있는 데이터들에 대해서는 검색 성능이 저하된다는 문제점이 있다.

기존의 알고리즘들 중 데이터의 특성을 고려하기 위하여 데이터 집합을 정렬하여 색인을 구성하는 STR, TGS과 같은 알고리즘들은 검색 성능은 향상되지만 색인을 구성하기 위한 시간이 증가되고 전체 데이터 집합을 정렬하지 않고 색인을 구성하는 [11, 9]와 같은 방법들은 색인을 구성하기 위한 시간은 단축되지만 검색 성능이 저하된다는 문제점이 있다. 제안하는 알고리즘은 색인을 구성하는 시간을 단축시키기 위해 전체 데이터 집합을 정렬하지 않고 데이터의 특성을 파악하여 분할하는 기법을 이용한다. 또한 색인을 구성시간을 단축시키면서도 검색 성능이 저하되면 문제점을 해결하기 위하여 영역에 대한 분할 비율과 분할되는 데이터의 수에 대한 비를 고려하여 전체 영역 중 한쪽으로 군집화되어 있는 데이터에 대한 특성을 파악할 수 있도록 한다.

본 논문의 구성은 다음과 같다. 2장에서 기존에 제안되었던 벌크 로딩에 관해 살펴보고 고차원 색인 구조에 대한 벌크 로딩을 수행하기 위해 고려한 사항들을 기술하고, 3장에서 제안하는 알고리즘에 대한 구체적인 사항들을 기술한다. 4장에서는 제안하는 알고리즘에 대한 성능 평가를 통해 기존에 존재하는 알고리즘과 성능을 비교 분석하고, 5장에서는 본 논문에 대한 결론과 향후 연구 방향에 대해 기술한다.

2. 기존의 벌크 로딩 알고리즘

주어진 대량의 데이터들을 이용하여 효과적인 색인을 구성하기 위한 기법을 벌크 로딩이라 한다. 이러한 벌크 로딩은 주어진 대량의 데이터들을 다른 연산에 방해받지 않고 한꺼번에 삽입하여 색인을 구성한다. 벌크 로딩을 이용하여 색인을 구성하면 대량의 데이터를 하나씩 삽입하여 색인을 구성하는 것보다 색인을 구성하는 시간을 단축시킬 수 있다. 또한 저장공간 활용률을 증가시켜 트리의 높이가 증가되는 것을 방지하고 데이터들에 대한 클러스터링 효과를 증가시켜 검색 성능을 향상시킬 수 있다.

최근 이러한 대량의 데이터에 대한 효과적인 색인을 구성하기 위한 벌크 로딩 알고리즘들이 제시되었다. Roussopoulos은 주어진 데이터들을 전체 차원 중 하나

의 특정 차원에 대해서만 데이터을 정렬하여 색인을 구성하는 Nearest-X(NX) 알고리즘[5]을 제안하였다. NX 알고리즘은 하나의 특정 차원에 대해서만 정렬을 수행하기 때문에 부모 노드는 적은 영역을 포함하여 포인트 질의에 대해서는 좋은 성능을 발휘한다. 그러나 다른 차원의 특징들을 색인을 구성하는데 고려하지 않기 때문에 부모 노드의 둘레 길이가 증가되는 결과를 발생시켜 영역 질의에 대해서는 성능이 현저히 저하된다는 단점이 있다.

Faloutosos와 Kamel은 주어진 데이터에 대한 클러스터링 효과를 향상시키기 위하여 Space filling Curve 중 가장 우수한 성능을 발휘하는 힐버트 커브[12]에서 사용되는 힐버트 값에 의해 오름차순으로 정렬을 수행하여 색인을 구성하는 HS(Hilbert Sorting) 알고리즘[6]을 제안하였다. 힐버트 값에 의해 정렬된 데이터를 이용하여 색인을 구성하는 HS 알고리즘은 노드들이 적은 영역을 포함하고 있어 포인트 질의에 좋은 성능을 갖고 있고 영역의 둘레 길이가 작기 때문에 NX 알고리즘보다 영역 질의에 대해 성능이 향상된다. 그러나 HS 알고리즘은 사용되는 데이터의 차원이 증가할수록 영역들 간에 겹침 영역이 증가된다는 단점이 있기 때문에 고차원에 대해서는 영역 질의에 대한 성능을 저하시킨다는 단점이 있다.

Leutenegger, Edgington, Lopez는 색인에 대한 검색을 수행하는 동안 방문하는 노드의 수를 최소화하기 위하여 주어진 데이터를 정렬하여 필요한 분할을 수행하는 과정을 반복하여 색인을 구성하는 STR(Sort Tile Recursive) 알고리즘을 제안하였다[7]. STR 알고리즘은 주어진 데이터들을 보다 효과적으로 클러스터링하기 위하여 주어진 데이터 집합을 조그마한 조각으로 계속 분할하면서 노드를 생성하고 있다. STR 알고리즘은 이전의 알고리즘인 NX 알고리즘에 비해 포인트 질의나 영역 질의 모두에 좋은 성능을 보이고 있다. 그러나 VLSI 데이터에 대한 성능은 HS 알고리즘이 우수한 것으로 알려져 있다. 따라서 힐버트 알고리즘과 STR 알고리즘의 성능은 주어진 데이터의 특성에 의존한다고 말할 수 있다.

지금까지 살펴본 NX, HS, STR 알고리즘들은 주어진 데이터를 정렬하여 단말 노드를 생성하는 고정된 전처리 과정을 사용한다. 또한 색인을 생성하기 위하여 단말 노드를 먼저 생성한 후 생성된 단말 노드를 이용하여 인덱스 노드를 만들어 가는 상향식 방법으로 색인을 구성한다. 이에 반해 Leutnengger, Lopez, Garcial은 루트 노드에서부터 단말 노드를 생성하는 하향식 방법으로 색인을 구성하는 TGS(Top down Greedy Split) 알고리즘[8]를 제안하였다. TGS 알고리즘에서는 데이터를 분할하기 위해 사용될 수 있는 모든 차원에 대해 분할 가능성을 고려하여 데이터를 분할한다. TGS 알고리즘은 하향식으로 트리를 생성하며 분할 가능성을 최대한 고려하기 때문에 이전의 알고리즘보다 매우 뛰어난 성능의 향상을 보인다. 특히 TGS 알고리즘을 이용하여 생성한 색인 구조에 대해 검색을 수행한다면 이전의 STR이나 HS 알고리즘에 비해 디스크 접근 수를 1/3로 줄일 수 있다. 그러나 TGS 알고리즘의 트리 구성 시간은 이전의 알고리즘보다 2.6배나 더 소비되기 때문에 벌크 로딩이 자주 발생하지 않는 경우 적합하다.

이전의 대부분의 알고리즘들은 주어진 데이터를 특정 차원에 대해 정렬을 수행하고 정렬된 순서에 따라 데이터를 노드에 저장하는 방식을 사용하였다. 그러나 실제적인 고차원 색인구조에 대하여 벌크 로딩을 수행한 경우 대량의 데이터들을 정렬하기 위해서는 너무 많은 시간을 소요한다는 문제점이 있다. 이러한 문제점을 해결하기 위하여 Bercken, Seeger, Widmayer는 주어진 데이터를 정렬하지 않고 Arge에 의해 제안된 버퍼 트리[10]를 이용하여 벌크 로딩을 수행하는 알고리즘을 제안하였다[9]. 버퍼 트리를 이용하여 벌크 로딩을 수행하는 알고리즘은 특정 차원에 대하여 데이터를 정렬하지 않고 분할이나 합병 기법을 사용하여 데이터를 분할한다. 트리의 인터널 노드들은 데이터들이 임시 저장되는 (외부) 버퍼를 포함하고 있고 버퍼 트리에는 동시에 다중 삽입이 가능하게 한다. 트리 내의 노드는 삽입 프로세스가 도착하면 노드의 버퍼 내에 레코드를 저장한다. 만약 버퍼 내의 레코드의 수가 미리 정해놓은 임계치를 넘는다면, 버퍼 내에 존재하는 모든 레코드들의 삽입 프로세스는 트리의 다음 레벨로 내려간다. 버퍼 트리를 이용한 벌크 로딩의 접근방식은 seeded trees[13]를 기반으로 하는 기법을 일반화한 것이라고 말할 수 있다. 버퍼 트리를 이용한 벌크 로딩 방식은 색인을 구성하는 시간은 단축시킬 수 있지만 색인을 구성하기 위해 주어진 전체 데이터에 대한 특성을 이용하지 못하기 때문에 검색 성능이 저하된다.

Berchtold, Bohm, Kriegel등은 전체 데이터 집합을 정렬하지 않고 효과적인 데이터 공간의 분할을 수행하

기 위하여 데이터 집합에 대한 데이터 수를 분할 비율로 선택하여 데이터를 단지 양분하면서 색인을 구성하는 알고리즘을 제안하였다[11]. [11]에서는 색인을 구성하기 위한 저장공간 활용률을 고정시키지 않고 사용자에 의해 주어진 임의의 저장 공간 활용률을 사용하여 색인을 구성하는데 사용할 수 있도록 한다. 트리를 생성하기 위하여 [11]에서는 사용자에 의해 제시된 저장공간 활용률에 따라 생성할 트리의 높이와 각 노드의 팬아웃 수 등을 결정하고 분할을 수행할 차원을 선택한다. 분할을 수행할 차원에 대해 변형된 퀵 정렬 알고리즘을 이용하여 데이터들에 대한 특정 분할 비율을 만족하도록 데이터를 양분한다. [11]에서는 벌크 로딩을 수행하기 위해 주어진 데이터 집합을 균등하게 분할하는 것이 아니라 데이터를 불균등하게 분할하여 검색 성능이 저하되는 것을 방지하였다.

3. 설계시 고려사항

벌크 로딩을 수행하기 위하여 주어진 데이터에 대한 특성을 파악하는 것은 검색 성능을 향상시킬 수 있는 중요한 요소이다. 내용 기반 이미지 검색을 지원하는 여러 가지 색인 구조들은 데이터의 삽입으로 인하여 오버플로우가 발생하면 오버플로우가 발생한 노드를 분할하게 된다. 오버플로우가 발생한 노드에 대한 분할을 효과적으로 수행하기 위하여 각각의 색인 구조들은 분할을 수행할 위치를 선택하기 위하여 여러 가지 분할 기준을 사용하고 있다. 그러나 이러한 분할 기준의 선택은 주어진 데이터 전체에 대한 특성을 반영하는 것이 아니라 일부 데이터 집합에 대한 특성만을 고려하여 분할을 수행하기 때문에 트리 내에 거의 비어있는 노드가 존재하여도 계속적인 분할을 수행하면서 겹침 영역을 증가시키고 트리의 높이 또한 증가시키는 문제점을 갖고 있다. 따라서 벌크 로딩에 의해 색인을 구성하기 위해서는 주어진 데이터 집합 전체에 대한 특성을 이용하여 효과적인 분할을 수행하기 위한 기준을 선택하고 분할 기준을 만족하는 경우에 대해서만 분할을 수행하고 있다. 그러나 주어진 데이터를 하나씩 삽입하여 색인을 구성하는 방법과 달리 벌크 로딩에 의해 색인을 구성하는 방법은 많은 양의 데이터를 이용하기 때문에 분할을 수행할 기준을 선택하는 것이 상당히 어렵고 검색 성능에 커다란 영향을 미친다.

벌크 로딩에 의해 구성할 색인에 대한 검색 성능을 향상시키기 위하여 데이터를 정렬하여 분할할 위치를 선택하는 STR과 TGS 알고리즘은 정렬된 데이터를 이용하여 분할을 수행할 가장 효과적인 위치를 선택하기 때문에 검색 성능을 향상시킬 수 있고 100% 저장공간 활용률을 보장할 수 있어 트리의 높이를 최소로 구성할 수 있다. 그러나 데이터를 정렬하여 색인을 구성하는 알고리즘들은 색인을 구성하기 위한 시간이 너무 많이 소요된다. 주어진 데이터에 대한 정렬을 수행하지 않고 색인을 구성하는 [11]과 [9]의 방법은 색인을 구성하는 시간은 단축시킬 수 있지만 데이터를 정렬하지 않기 때문에 주어진 데이터 전체에 대한 특성을 반영하지 못하고 일부 데이터에 대한 특성만을 이용하기 때문에 검색 성능을 저하시킨다. 또한 정렬을 수행하여 색인을 구성하는 알고리즘에 비해 저장공간 활용률이 저하되고 이로 인하여 트리의 높이를 증가시킬 수 있다. 제안하는 벌크 로딩 알고리즘에서는 기존의 벌크 로딩 알고리즘에 대한 문제를 해결하기 위해 다음과 같은 사항들을 고려한다.

첫째, 벌크 로딩에 의해 생성할 트리의 높이를 최소로 구성한다. 트리의 높이가 증가되면 검색을 수행하는 과정에서 데이터를 읽기 위한 I/O 시간이 증가하기 때문에 트리의 높이를 최소로 하여 색인을 구성하는 것이 색인에 대한 검색 시간을 단축시킬 수 있다. 따라서 트리의 높이는 100% 저장공간 활용률을 만족하도록 구성했을 때의 색인의 높이를 초과하지 않도록 한다.

둘째, 사용자에 의하여 정의된 저장공간 활용률로 인하여 트리의 높이가 증가되는 것을 방지한다. 만약 색인을 구성하기 위해 사용자에 의해서 저장공간 활용률을 결정할 수 있도록 한다면 이로 인하여 벌크 로딩에 의해 생성한 트리의 높이가 증가될 수 있다. 따라서 제안하는 알고리즘은 트리의 높이를 변화시키지 않는 범위 내에서 저장공간 활용률을 결정하여 사용자에 의해 정의된 저장공간 활용률로 인하여 트리의 높이가 증가되는 것을 방지한다.

셋째, 색인을 구성하기 위한 시간을 단축하기 위하여 전체 데이터 집합을 정렬하지는 않는다. 주어진 데이터 집합에 대해 정렬을 수행한다면 색인을 구성하는 시간이 증가되는 단점이 있다. 그러나 정렬을 수행하지 않고 색인을 구성한다면 데이터에 대한 특성을 제대로 반영하지 못하여 검색 성능을 저하시킬 수 있다. 따라서 제안하는 알고리즘은 벌크 로딩을 수행할 데이

터에 대해 영역의 분할 비율에 따른 데이터들간의 평균적인 거리를 이용하여 데이터를 양분하고 분할 기준을 만족하는 영역 주변에 존재하는 일부 데이터들에 한해서 정렬을 수행하여 분할을 수행할 가장 적절한 위치를 선택한다.

넷째, 하향식으로 색인을 구성한다. 주어진 데이터들에 대한 분할을 수행하여 필요한 단말 노드들이 모두 생성되면 이를 이용하여 인터널 노드를 생성하는 상향식 방식은 인터널 노드를 생성하기 위한 연산들이 필요하고 불필요한 I/O를 발생시킬 수 있다. 제안하는 알고리즘은 루트 노드에서부터 트리의 각 레벨에 대해 팬아웃 만큼의 필요한 노드들을 생성하고 하위 노드를 생성해 가는 하향식 방식을 사용한다.

4. 제안하는 벌크 로딩 알고리즘

이 장에서는 제안하는 벌크 로딩 알고리즘을 수행하는 과정에 대해 상세히 기술한다. 먼저 알고리즘의 전체적인 흐름을 기술한 후, 생성할 색인의 높이와 저장공간 활용률을 결정하는 과정에 대해 기술하고, 색인 구성시간을 단축시키면서도 검색 성능을 향상시키기 위해 분할 위치를 선택하는 방법을 기술한다. 마지막으로 색인을 구성하는 과정을 자세히 기술한다.

4.1 개 요

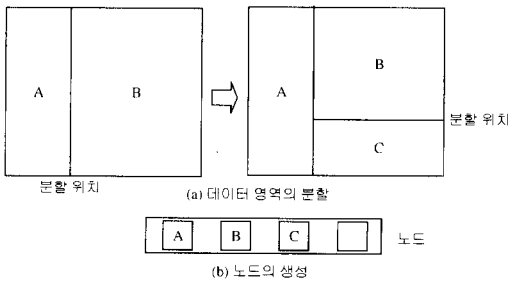
제안하는 벌크 로딩 알고리즘은 트리를 구성하기 위해 (그림 1)과 같은 과정을 수행한다. 먼저 1행과 2행에서 주어진 전체 데이터 수를 n에 대하여 생성할 트리의 높이(h)와 저장공간 활용률(*Storage Utilization*)을 결정한다. 저장공간 활용률이란 트리을 구성하였을 때 노드의 크기에 대해 각 노드에 얼마만큼의 데이터들이 저장하고 있는지를 나타낸다. 이러한 저장공간 활용률은 생성할 트리의 높이에 의해 결정되기 때문에 먼저 트리의 높이를 결정하고 이에 따른 저장공간 활용률을 결정한다. 3~7행은 주어진 데이터를 이용하여 트리의 각 레벨에서 몇 개의 하위 노드들을 포함하고 있는지를 나타내는 팬아웃 수를 결정하여 필요한 팬아웃 수만큼의 분할된 영역을 생성한다. 제안하는 알고리즘에서는 트리를 구성하기 위하여 루트 노드에서부터 단말 노드를 생성할 때까지 노드를 구성하는데 필요한 저장공간 활용률을 만족할 수 있도록 데이터를 분할하여 분할된 영역에 대한 MBR 정보를 노드에 기록한다.

```
알고리즘 Create_Index( )
입력 : 데이터 집합
{
1 :   h = ⌈ log_{C_div,max} ( n / C_data,max ) ⌉ + 1;
2 :   Storage_utilization = h-1√( n / ( C_data,max · C_div,max^(h-1) ) );
3 :   for(Level= h; Level=1; Level--)
4 :   {
5 :       Create_Level( );
6 :       분할된 영역에 대한 MBR을 노드에 기록;
7 :   }
}
```

(그림 1) 제안하는 벌크 로딩 알고리즘

(그림 2)는 2차원의 데이터에 대해 3개의 분할된 영역을 생성한 과정을 나타낸 것이다. 현재 데이터를 분할하여 생성해야할 색인의 레벨에서 각 노드들이 저장공간 활용률을 만족하기 위해 몇 개의 분할 영역을 생성해야 하는지를 판별하고 데이터를 분할하기 위한 차원과 위치를 선택한다. 분할을 수행할 분할 차원과 위치가 선택되면 저장공간 활용률을 만족하는 경우에 한해서만 데이터를 분할한다. (그림 2(a))는 데이터 영역을 분할하는 과정을 나타내는 것이다. 먼저 주어진 데이터를 1차원에 대해 계속적인 분할을 수행하면서 저장공간 활용률을 만족할 수 있도록 데이터를 분할한다. 1차원에 의해 분할된 영역들 중 저장공간 활용률을 만족하기 위해 분할되어야 할 데이터 영역이 있다면 다시 분할 차원과 위치를 선택하여 분할을 수행한다. (그림 2(a))에서는 1차원에 대하여 분할된 영역 중 B부분이 하나의 노드를 생성하기에는 너무 많은 데이터를 포함하고 있기 때문에 다시 분할 차원으로 2차원을 선택하여 데이터를 분할한다. 제안하는 알고리즘에서는 가장 적절한 위치를 선택하기 위하여 주어진 데이터들에 대하여 분할을 수행해야 할 차원의 구간에 대해 데이터들간의 평균적인 거리를 나타내는 *meandist*를 계산하여 *meandist*의 변화량에 따라 주어진 데이터의 분포 형태를 파악하여 영역 내에 데이터들이 존재하지 않는 구간을 최소화하기 위한 분할을 수행한다. 현재 분할을 수행하고 있는 데이터에 대해 필요한 팬아웃 만큼 데이터 영역이 생성되면 (그림 2(b))와 같이 분할된 데이터 영역에 대한 MBR 정보를 생성하여 노드에 저장한다. 이러한 과정을 반복하면서 필요한 단말 노드가 모두 생성되면 분할 과정을 중지하고 트리를 완성

한다. 제안하는 벌크 로딩 알고리즘을 수행하기 위한 트리의 높이, 저장공간 활용률, 분할 축의 선택과 같은 구체적인 내용은 다음절에서부터 기술한다.



(그림 2) 데이터 분할 과정

4.2 트리의 높이

벌크 로딩에 의해 색인을 구성하기 위하여 제일 먼저 수행해야 할 일은 생성할 트리의 높이와 저장공간 활용률을 결정하는 것이다. 데이터를 하나씩 삽입하여 색인을 구성하는 방법에서는 데이터의 삽입으로 계속적인 분할이 발생하여 트리의 높이가 증가되기 때문에 생성할 색인의 높이를 미리 결정할 수 없다. 그러나 벌크 로딩을 이용하여 색인을 구성할 때에는 색인을 구성할 전체 데이터들이 이미 주어져 있기 때문에 이를 이용하여 가장 최적의 색인을 구성하기 위한 색인의 높이를 미리 결정한다. 기존 벌크 로딩 알고리즘들은 트리의 높이를 결정하기 위하여 먼저 주어진 데이터를 각 노드에 얼마만큼 저장할 것인지 결정하여 노드의 저장공간 활용률을 결정한다. 이러한 저장공간 활용률이 결정되면 저장공간 활용률을 만족하면서 생성할 트리의 높이를 결정한다. 그러나 제안하는 알고리즘에서는 트리의 높이를 먼저 결정하고 트리의 높이를 변화시키지 않는 범위에서 유연하게 저장공간 활용률을 결정한다.

생성할 트리의 높이는 주어진 데이터의 수와 각 노드에 저장될 수 있는 최대 엔트리의 수 등에 의해 결정된다. 트리의 높이를 결정하기 위하여 벌크 로딩을 수행할 데이터의 수를 n이라고 가정하자. 실제 데이터를 저장하는 단말 노드와 단말 노드를 찾기 위한 영역 정보를 포함하고 있는 인터널 노드는 최대로 저장할 수 있는 엔트리의 수가 서로 다를 수 있기 때문에 이를 각각 계산하여 보자. 단말 노드와 인터널 노드에 최대로 저장될 수 있는 최대 엔트리의 수를 각각 $C_{data,\max}$, $C_{dir,\max}$라 하면 $C_{data,\max}$는 식 (1)과 같이 계산될 수 있고 $C_{dir,\max}$도 $C_{data,\max}$과 유사한 방법으로 구할 수 있다.

$$C_{data,\max} = \left\lfloor \frac{\text{페이지의 크기}}{\text{하나의 데이터의 크기}} \right\rfloor \qquad (1)$$

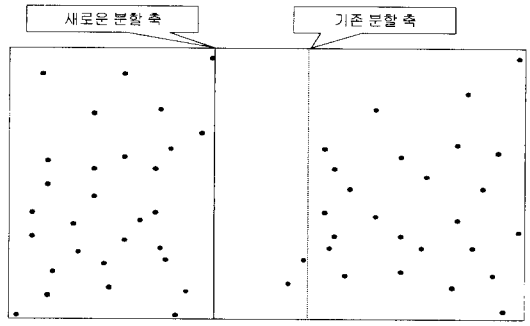
이제 생성할 트리의 높이를 결정하기 위하여 트리의 높이를 h라 하자. 만약 100% 저장공간 활용률을 보장하여 트리를 생성하면 생성될 트리의 높이는 식 (2)와 같이 계산된다.

$$h = \left\lceil \log_{C_{dir,\max}}\left(\frac{n}{C_{data,\max}}\right) \right\rceil + 1 \qquad (2)$$

제안하는 벌크 로딩 알고리즘은 생성할 트리의 높이를 최소화하기 위하여 100% 저장공간 활용률을 보장하는 트리의 높이 h를 생성할 트리의 높이로 결정한다. 이렇게 결정된 트리의 높이는 주어진 데이터를 이용하여 구성할 수 있는 트리의 최소 높이이다.

4.3 저장공간 활용률

트리의 높이가 결정되면 저장공간 활용률을 결정해야 한다. 만약 벌크 로딩을 수행하기 위해 100% 저장공간 활용률을 보장한다면 각 노드에는 $C_{data,\max}$, $C_{dir,\max}$의 수 만큼의 엔트리들이 저장될 수 있을 것이다. 그러나 100% 저장공간 활용률 만을 보장하여 색인을 구성한다면 트리의 높이는 최소로 생성할 수 있지만 데이터를 포함하는 영역 내에 데이터를 포함하고 있지 않은 구간을 증가시킬 수도 있다. 즉 영역 내에 Dead Space를 증가시킬 수 있다. 예를 들어 (그림 3)과 같이 저장공간 활용률을 100%으로 결정하여 데이



(그림 3) 저장공간 활용률을 보장하기 위한 데이터 분할

터를 분할하는 [기존의 분할 축]보다 100%의 저장공간 활용률을 보장하지 않고 데이터를 분할하는 [새로운 분할 축]이 영역 내에 데이터들이 존재하지 않는 구간을 최소화시킬 수 있다. 따라서 제안하는 벌크 로딩 알고리즘에서는 트리의 높이는 최소화하면서 저장공간 활용률을 100%로 고정하지 않고 트리의 높이를 변화시키지 않는 범위 내에서 주어지는 데이터의 수에 따라 저장공간 활용률을 결정한다.

벌크 로딩에 의해 생성할 노드에 대한 저장공간 활용률을 결정하기 위해 다음을 생각해 보자. 만약 트리의 높이가 결정되고 저장공간 활용률(storage utilization)이 결정되었다고 가정하면 저장공간 활용률에 따라 단말 노드와 인터널 노드에 저장될 수 있는 평균적인 엔트리의 수 $C_{data,avg}$, $C_{dir,avg}$는 식 (3)과 식 (4)와 같이 결정된다.

$$C_{data,avg} = (Storage\ utilization) \cdot C_{data,\max} \quad (3)$$

$$C_{dir,avg} = (Storage\ utilization) \cdot C_{dir,\max} \quad (4)$$

단말 노드에 저장될 수 있는 평균적인 엔트리의 수 $C_{data,avg}$가 결정되면 이에 따라 벌크 로딩에 의해 할당될 단말 노드의 수 S는 식 (5)와 같이 결정된다. 또한 트리 내에서 각 인터널 노드가 평균적으로 $C_{dir,avg}$의 엔트리를 포함하고 있다면 주어진 데이터를 이용하여 생성한 색인 구조에는 ${C_{dir,avg}}^{h-1}$에 해당하는 단말 노드가 할당될 것이다.

$$S = \frac{n}{C_{data,avg}} \quad (5)$$

따라서 S와 ${C_{dir,avg}}^{h-1}$는 색인을 위해 할당될 최대 단말 노드의 수로 그 수는 식 (6)과 같이 동일해야 한다.

$$\frac{n}{C_{data,avg}} = {C_{dir,avg}}^{h-1} \quad (6)$$

저장공간 활용률을 계산하기 위해 식 (6)에 식 (3)과 식 (4)를 대입하면 식 (7)과 같이 된다.

$$\frac{n}{(Storage\ utilization) \cdot C_{data,\max}} = \{(Storage\ utilization) \cdot C_{dir,\max}\}^{h-1} \quad (7)$$

식 (7)을 정리하면 식 (8)과 같이 저장공간 활용률이 결정된다.

$$Storage\ utilization = \sqrt[h-1]{\frac{n}{(C_{data,\max} \cdot {C_{dir,\max}}^{h-1})}} \quad (8)$$

식 (8)에서 계산된 저장공간 활용률은 트리의 높이를 최소로 했을 때 트리의 높이를 변화시키지 않는 범위내에서 계산된 저장공간 활용률이다. 벌크 로딩에 의해 생성할 트리의 높이와 저장공간 활용률이 결정되면 이를 이용하여 주어진 데이터 집합에 대해 분할을 수행하여 색인을 구성하게 된다. 데이터 집합을 분할하는 과정에서 분할된 각각의 데이터 집합들이 저장공간 활용률을 만족하지 못한다면 이는 트리의 높이를 증가시킬 수 있는 가능성이 있기 때문에 항상 저장공간 활용률을 만족하도록 분할을 수행한다.

4.4 분할 축의 선택

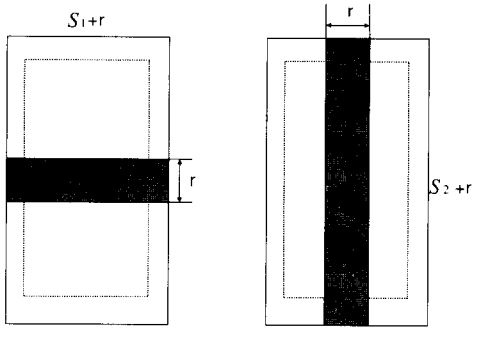
트리의 높이와 저장공간 활용률이 결정되면 이제 데이터를 분할해야 한다. (그림 4)는 트리의 각 레벨에서 데이터를 분할하는 알고리즘이다. 1~4행은 데이터를 양분을 수행하고 5~15행은 분할된 데이터가 현재 레벨에서 더 이상의 분할을 수행해야 하는지를 판별하여 MBR을 생성한다. 필요한 데이터 영역을 생성하기 위해 1행에서 분할을 수행할 차원을 선택한다. 2행에서

```
알고리즘 Create_Level( )
입력 : 트리의 레벨, 데이터 집합
{
 1 : 분할 차원 i 를 선택;
 2 : meandist = (B_i − A_i) / n ;
 3 : Split_position = A_i + (B_i − A_i) / Part_num ;
 4 : Split_data( );
 5 : if (분할된 모든 영역 내에 포함된 데이터 수 <=
 6 :     현재 Level에서 하나의 노드에 포함할 수 있는 최대 데이터 수)
 7 : {
 8 :         분할된 영역에 대한 MBR을 계산;
 9 :         return;
10 : }
11 : while(분할을 수행할 영역이 존재)
12 : {
13 :         분할을 수행할 데이터 영역을 결정;
14 :         Create_Level( );
15 : }
}
```

(그림 4) 트리의 각 레벨에 필요한 데이터 집합 생성 알고리즘

분할을 수행하기 위해 선택된 차원에서 평균적으로 데이터들간에 얼마만큼의 거리를 유지하고 있는지를 나타내는 *meandist*을 계산하고 3행에서 분할 위치 *Split_position*을 선택한다. 그러나 *Split_position*은 최적의 분할 위치라는 것을 보장할 수 없기 때문에 4행에서 *Split_position*보다 분할하기 더 적절한 위치가 있는지를 판별하여 데이터를 분할한다. 데이터에 대한 분할을 수행하기 위하여 주어진 데이터 집합에 대한 특성을 알고 있다면 분할 축를 선택하기가 용이할 것이다. 그러나 제안하는 알고리즘에서는 주어진 데이터 집합 전체를 정렬하지 않기 때문에 데이터에 대한 특성을 파악하기 어렵다.

분할을 수행하기 위해 (그림 5)와 같이 2차원의 데이터 공간이 존재한다고 생각해 보자. 주어진 2차원의 데이터 공간에 대해 1차원, 2차원의 길이가 각각 s_1, s_2이고 $s_1 < s_2$일 때 (그림 5(a))는 영역의 분포가 큰 2차원에 대해 분할을 수행하였고 (그림 5(b))는 영역의 분포가 작은 1차원에 대해 분할을 수행하였다. 그림에서 알 수 있듯이 영역의 분포가 큰 2차원에 대해 분할하는 것이 검색을 수행하는 과정에서 두 개의 데이터 영역을 모두 탐색할 확률이 더 감소된다. 또한 데이터의 분포가 큰 차원은 데이터들이 존재하지 않는 영역인 Dead Space을 포함하고 있을 가능성이 다른 차원보다 더 많기 때문에 데이터 영역의 분포가 큰 차원을 선택하여 데이터를 분할하는 것이 데이터 분포가 작은 차원을 분할하는 것보다 더 효과적이라고 할 수 있다. 분할을 수행하기 위한 차원으로 영역의 분포가 가장 큰 차원을 선택하여 데이터를 분할하는 것이 검색 성능을 향상시킬 수 있다는 것을 이미 증명하였다[14].

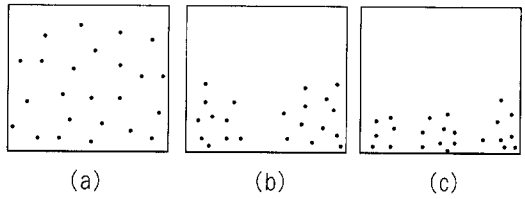


(a) 2차원에 대해 분할 (b) 1차원에 대해 분할

(그림 5) 분할 차원의 선택

따라서 제안하는 알고리즘에서는 데이터에 대한 분할을 수행할 차원으로 데이터의 분포가 가장 긴 차원을 선택한다. 만약 데이터의 영역의 길이가 같은 차원이 두 개 이상 존재하여 데이터를 분할하기 위한 차원을 선택할 수 없다면 주어진 데이터에 대한 분산이 가장 큰 차원을 선택한다.

이제 주어진 데이터에 대한 분할을 수행하기 위해 마지막으로 결정해야 하는 것은 선택된 차원에서 분할하기에 가장 적당한 위치를 선택하는 것이다. 분할하기 위한 위치를 선택하는 방법을 설명하기 위하여 (그림 6)과 같이 분할 차원으로 1차원을 선택하였지만 서로 다른 데이터 분포 특성을 갖고 있는 세 가지의 데이터 집합이 있다고 가정하자.



(그림 6) 데이터의 분포

(그림 6)의 (a)는 1차원에 균일한 데이터 분포를 갖고 있지만 (그림 6(b))와 (그림 6(c))는 1차원에 대해 데이터 일부가 한쪽으로 몰려 있는 데이터 분포를 갖고 있다. 분할을 수행하기 위해 선택된 차원에 대해 *(그림 6(a))와 같이 균일한 분포를 갖고 있는 경우에는 저장공간 활용률 만을 만족하는 위치에서 데이터를 분할하여도 검색 성능에 큰 영향을 미치지는 않는다. 그러나 (그림 6(b))와 (그림 6(c))처럼 데이터의 분포가 특정 영역에 집중되어 있는 경우에는 저장공간 활용률만을 보장하여 데이터를 분할하면 데이터가 군집되어 있는 영역의 중간을 분할하게 되어 클러스터링 효과를 저하시키거나 또는 데이터의 분할로 영역 내에 데이터들이 존재하지 않는 구간을 포함하여 Dead Space를 증가시켜 검색 성능을 저하시키게 된다. 따라서 어느 특정 제약 조건만을 만족한다고 하여 데이터를 분할하는 것은 검색 성능을 저하시킬 수 있기 때문에 데이터들의 분포에 따라 가능한 모든 분할 가능성을 고려하여 데이터를 분할하여야 한다. 제안하는 알고리즘은 분할을 수행할 위치를 선택하기 위하여 저장공간 활용률 만을 이용하거나 또는 데이터의 수를 분할 비율로 선택하는 것이 아니라 영역의 분할 비율에 따른 데이

터의 수를 이용하여 데이터들의 분포 특성을 파악하여 분할 위치를 선택한다.

데이터에 대한 분할을 수행하기 위하여 선택된 차원에 대한 데이터 분포가 $[A_i, B_i]$라 하고 벌크 로딩을 수행하기 위해 데이터를 분할하고 있는 색인의 레벨이 h라고 가정하자. 제안하는 벌크 로딩 알고리즘은 트리를 구성하기 위하여 루트 노드에서부터 시작하여 단말 노드를 만들 때까지 데이터를 분할한다. 트리의 각 레벨에서 필요한 팬아웃 수만큼 데이터 영역을 생성하면 현재 레벨에 대한 분할을 중지한다. 다시 하위 레벨에 필요한 데이터 영역을 생성하기 위하여 상위 레벨에서 생성한 데이터 영역을 분할하는 과정을 반복하면서 필요한 단말 노드를 모두 생성하면 데이터 분할 과정을 중지한다.

먼저 분할을 수행하기 위한 적절한 위치를 선택하기 위하여 데이터들의 분포 형태를 파악할 수 있도록 데이터들간에 평균적으로 얼마만큼의 거리를 유지하고 있는지를 나타내는 *meandist*를 식 (9)와 같이 계산한다.

$$meandist = \frac{B_i - A_i}{n} \tag{9}$$

분할 대상의 데이터 집합에 대한 *meandist*가 계산되면 현재 분할을 수행할 데이터 집합이 몇 번의 분할을 더 수행해야 하는지를 판별한다. 주어진 데이터의 수가 n이고 현재 분할을 수행하고 있는 레벨이 h라면 현재 레벨에 분할을 수행해야 할 횟수 *Part_num*은 식 (10)과 같다.

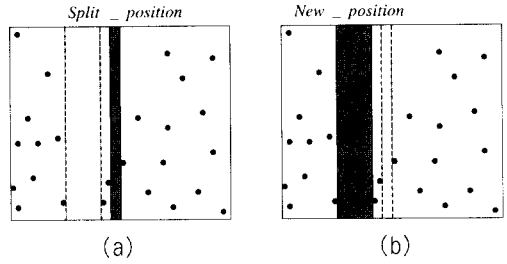
$$Part\_num = \left\lfloor \frac{n}{C_{dir,avg}^{\,h-2} \cdot C_{data,avg}} \right\rfloor \tag{10}$$

분할을 수행하기 위한 초기 위치를 선택하기 위한 분할 위치 *Split_position*을 결정하는 식은 식 (11)이다.

$$Split\_position = A_i + \frac{(B_i - A_i)}{Part\_num} \tag{11}$$

만약 주어진 분할을 수행할 차원에 대해 데이터 집합이 정규 분포(normal distribution)로 되어있다면 *Split_position*에 의해 데이터를 분할하면 적은 영역을 포함하고 있는 부분은 현재 분할을 수행하고 있는 트리의 레벨에서 하나의 부 노드를 생성할 수 있는 데이터 수를 포함하고 있을 것이다. 그러나 주어진 데이터가 데이터 영역의 한쪽으로 몰려 있는 데이터 분포라면 *Split_position*에서는 저장공간 활용률을 만족하지 않을 수도 있다. 따라서 *Split_position*에 따라 분할된 데이터 집합들이 모두 저장공간 활용률을 만족한다고는 말할 수 없다. 그렇기 때문에 *Split_position*에 저장공간 활용률을 만족하는지를 판별해야만 한다. 만약 분할을 수행하여 저장공간 활용률을 만족하지 못한다면 다시 *Split_position*을 결정하여 데이터를 양분한다.

데이터의 분할로 양분된 각각의 데이터 집합에 대해 분할을 수행하고 있는 현재 레벨에서 몇 번의 분할을 더 수행할 수 있는지를 계산하여 주어진 데이터를 분할될 수로 나누어 저장공간 활용률을 만족하는지를 판별한다. 예를 들어 현재 분할을 수행해야 하는 레벨에서 저장공간 활용률을 만족하기 위해서 50개의 데이터 집합이 필요하고 최대 60개의 데이터들이 포함될 수 있다고 할 때 분할을 수행한 데이터 집합이 100개의 데이터를 포함하고 있다면 주어진 데이터 집합은 두 개의 부 노드를 생성해야 한다. 두 개의 부 노드에는 각각 50개씩의 데이터를 포함할 수 있기 때문에 저장공간 활용률을 만족한다. 그러나 데이터를 분할한 데이터 집합이 135개의 데이터를 포함하고 있다면 두 개의 노드에 다 포함될 수 없기 때문에 세 개의 부 노드를 생성하여 데이터를 저장해야 한다. 세 개의 부 노드들에는 각각 45의 데이터를 포함하기 때문에 저장공간 활용률을 만족하지 못하여 데이터를 분할해서는 안 된다.



(그림 7) 분할 위치의 선택

분할 차원에서 초기의 *Split_position*을 기준으로 데이터를 양분하여 각각의 데이터 집합에 대해 *meandist*를 계산한다. 만약 *Split_position*에 따라 데이터를 양분하여 양분된 데이터 집합 중 좁은 영역을 포함하고 있는 데이터 집합에 대한 *meandist*가 분할을 수행하기 전의 *meandist*보다 감소되었다면 이는 좁은 영역을 포함하고 있는 데이터 집합 내에 데이터들이 집중되어 있다는 것을 의미한다. 그러나 *Split_posi-*

*tion*에 따라 양분된 데이터 집합 중 좁은 영역을 포함하고 있는 데이터 집합에 대한 *meandist*가 분할을 수행하기 전의 *meandist*보다 증가되었다면 하나의 부 노드를 생성할 수 있을 정도의 저장공간 활용률을 만족하지 못한다는 의미한다.

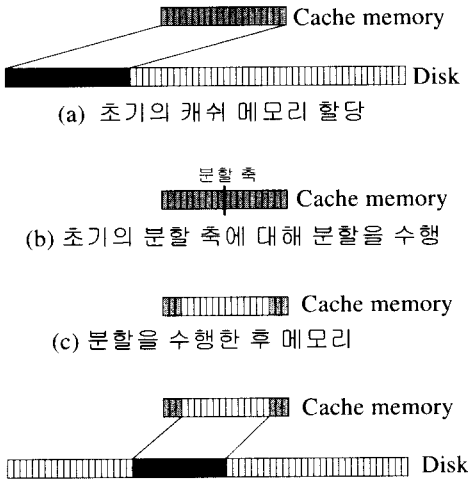
*Split_position*에 따라 분할된 각 데이터 집합들에 대해 *meandist*의 변화량을 이용하여 현재 분할 위치에서 분할된 데이터 집합들이 분할을 수행한 차원에 대한 영역에 어떠한 데이터 분포 형태를 갖고 있는지를 판별할 수 있다. 그러나 (그림 7)에서 보는 것과 같이 데이터들이 특정 영역 내에 집중되어 있을 경우에는 저장공간 활용률을 만족하더라도 (그림 7(a))와 같이 현재의 분할 위치 *Split_position*에 의해 분할하는 것보다는 (그림 7(b))에서와 같이 *New_position*에서 데이터를 분할하여 저장공간 활용률을 만족한다면 *New_position*에서 데이터를 분할한다. 이는 분할된 데이터 집합에 대한 영역을 생성하였을 때 Dead Space을 줄일 수 있기 때문에 보다 효과적이다. 따라서 제안하는 알고리즘은 *Split_position*에서 분할된 데이터 집합들이 저장공간 활용률을 만족한다 하더라도 무조건적으로 데이터를 분할하는 것이 아니라 데이터 집합에 대한 영역을 보다 효과적으로 생성할 수 있는 위치를 판별하여 데이터를 분할한다.

4.6 분할 과정

이제 주어진 데이터 집합을 분할하는 과정을 자세히 살펴보자. 분할을 수행하기 위한 분할 차원과 초기 분할 위치가 선택되면 주어진 데이터를 분할하여야 한다. (그림 8)은 분할 차원에 따라 데이터를 분할하는 알고리즘이다. 1행는 초기 분할 위치 *Split_position*에 따라 데이터를 양분한다. 그러나 *Split_position*이 최적의 분할 위치라 할 수 없기 때문에 2~22행은 이전의 *Split_position*보다 분할하기에 더 적절한 위치가 존재하는지를 판별하여 만약 초기 분할 위치보다 분할하기 더 적절한 위치가 있다는 이를 이용하여 데이터를 분할한다. 벌크 로딩을 수행하기 위해 주어진 데이터 집합들은 데이터의 양이 많기 때문에 주기억장치에 모든 데이터가 적재되지 못할 수 있다. 이러한 데이터에 대한 분할을 수행할 때 디스크 I/O 수를 감소시키기 위하여 제안하는 알고리즘에서는 일정한 크기의 캐쉬 메모리를 할당하여 사용한다. 초기에 주어진 데이터 집합은 그 양이 너무나 크기 때문에 메모리에 모두 적재될 수는 없을 것이다. 그러나 분할 과정을 반복하는 과정에서 데이터 집합은 양이 줄어들기 때문에 어느 순간에 모두 메모리에 적재될 수 있을 정도의 데이터 양이 될 것이다. 주어진 데이터 집합을 분할하기 위하여 먼저 주어진 데이터 집합이 벌크 로딩을 위해 할당된 캐쉬 메모리에 모두 적재될 수 있는지를 판별한다. 만약 분할을 수행할 데이터 집합이 캐쉬 메모리에 모두 적재될 수 있다면 메모리 내에서 데이터를 분할한다. 만약 분할을 수행할 데이터 집합이 캐쉬 메모리에 모두 적재될 수 없다면 (그림 9)과 같이 보조기억 장치인 디스크에 존재하는 데이터 집합 중 일부를 캐쉬 메모리에 적재할 수 있는 데이터 수만큼을 메모리에 할당하고 분할 축에 따라 데이터를 양분한다. 양분된 데이터 집합에서 분할 축에 가까운 일정 양의 데이터를 메모리에 계속 할당한다. 다시 데이터 집합 중 일부를 메모리에 할당하여 분할 축에 따라 데이터를 양분하고 이전에 메모리에 남아있던 데이터와 비교하여 분할 축에 더 가까운 특정 양의 데이터를 다시 계산하여 메모리에 할당한다.

```
알고리즘 Split_data( )
입력 : 분할을 수행할 데이터 집합, Split_position, 분할 차원
{
 1 : Bipartition( ); /* Split_position에 따라 데이터를 양분*/
 2 : Right_meandist = Right_region의 meandist;
 3 : Left_meandist = Left_region의 meandist;
 4 : if ( Left_region < Right_region )
 5 : {
 6 :       mean_dist = Left_meandist;
 7 :       dataset[ ] = Left_region;
 8 : }
 9 : else
10 : {
11 :       mean_dist = Right_meandist;
12 :       dataset[ ] = Right_region;
13 : }
14 : if ( mean_dist ≤ Meandist )
15 : {
16 :       Sort_data( ); /*메모리 내에 존재하는 데이터
                          집합에 대한 정렬*/
17 :       Select_position( );
18 : }
19 : else if ( mean_dist > Meandist )
20 : {
21 :       Split_position += (B_i − A_i) / Part_num
22 :       Split_data( )
}
```

(그림 8) 데이터 집합의 분할 알고리즘



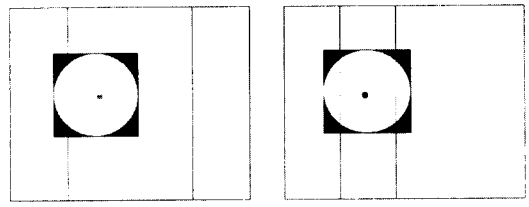
(그림 9) 캐쉬 메모리의 할당

이와 같은 방법을 계속하여 데이터를 분할하면 초기의 분할 축을 기준으로 하여 데이터들이 양분되어 있을 것이다. 또한 분할 축에 가까운 특정 양의 데이터들은 메모리에 계속 적재되어 있다. 분할 축에 따라 데이터를 분할하면서 각각의 차원 i에 대한 영역의 최대, 최소 값을 $C_{i,\max}$, $C_{i,\min}$에 기록하여 다음에 분할을 수행할 차원을 선택하는 데 사용한다.

이제 *Split_position*보다 분할하기 적절한 위치를 선택하기 위하여 양분된 각 데이터 집합에 대해 *meandist*을 계산한다. 만약 좁은 영역에 대한 *meandist* 값이 분할을 수행하기 전의 *meandist* 보다 감소되었다면 넓은 영역을 포함하고 있는 데이터 집합이 하나의 부 노드를 만들 수 있는 데이터 수 이상인지를 판별한다. 만약 하나의 부 노드를 생성할 수 있는 데이터 수 이상이라면 분할 축에 가까운 데이터들이 메모리에 존재하기 때문에 메모리에 존재하는 데이터 중에서 분할 축보다 큰 데이터들을 정렬하여 저장공간 활용률을 만족하면서 영역의 크기를 최소로 만들 수 있는 분할 위치를 선택한다. 그러나 넓은 영역을 포함하고 있는 데이터 집합이 하나의 부 노드를 만들 수 있는 데이터 수보다 작다면 현재 분할 축에 따라 데이터를 분할하면 저장공간 활용률을 만족하지 못하기 때문에 메모리에 존재하는 데이터 중 분할 축보다 작은 데이터들을 정렬하여 분할을 수행할 적절한 위치를 선택한다.

이와 달리 양분된 데이터 집합 중 좁은 영역을 포함하고 있는 데이터 집합에 대한 *meandist*가 분할을 수행하기 전의 *meandist*에 비해 증가되었다면 좁은 영역을 포함하고 있는 데이터 집합이 저장공간 활용률을 만족하지 못한다는 것을 의미한다. 따라서 현재 메모리에 존재하는 데이터 중에서 분할 축보다 더 큰 데이터에 대해 저장공간 활용률을 만족할 수 있고 영역의 크기를 작게 만들 수 있는 분할 위치를 선택한다. 또한 초기의 분할 위치에서 $\frac{(B_i - A_i)}{Part\_num}$ 만큼의 크기를 증가시킨 위치를 분할 위치로 선택하여 분할을 수행한 후 다시 *meandist*를 계산하여 위와 동일한 방법으로 분할을 수행할 위치를 선택한다. 이전에 분할을 수행하기 위해 적절하다고 생각했던 위치와 현재 분할을 수행하기 위해 결정한 분할 위치를 비교하여 적절한 위치를 선택하여 데이터를 분할한다.

데이터 집합을 분할하는 과정에서 한번 분할을 수행한 차원이 다시 분할 차원으로 선택되었을 때 처음 분할을 수행한 위치의 바로 이웃에서 계속적으로 분할이 일어나면 (그림 10)과 같이 분할된 영역에 대한 범위 탐색을 수행할 때 검색 성능이 떨어지는 문제점을 갖는다. 따라서 처음 분할 위치의 반대 위치에서 분할을 수행하는 것이 검색해야 할 노드 수를 감소시킨다. 따라서 한번 분할을 수행한 차원에 대해 다시 분할을 수행해야 한다면 이전 분할 위치의 반대편에서 분할 위치를 선택하여 분할한다. 이렇게 함으로써 검색해야 할 노드 수를 감소시킨다.



(그림 10) 계속된 분할 차원의 선택

이러한 분할 과정을 통해 양분된 데이터 집합 중 많은 양의 데이터를 포함하고 있는 데이터 집합에 대해 다시 위와 같은 분할과정을 반복하여 분할을 수행한다. 현재 레벨에 대한 저장공간 활용률을 만족할 수 있는 부 노드들이 생성되면 현재 레벨에서 분할을 수행하여 만들어진 데이터 집합에 대한 영역 정보를 생성하여 현재 레벨에 필요한 노드를 생성하고 트리의

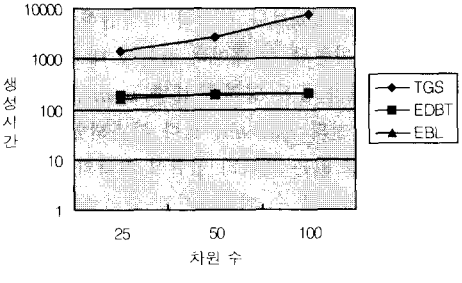
다음 레벨에 대한 분할 과정을 수행하게 된다. 분할 과정을 반복하여 단말 노드에 대한 분할을 수행하고 필요한 데이터 집합을 생성하면 위와 동일한 방법으로 노드를 생성하고 색인을 구성한다.

5. 성능 평가

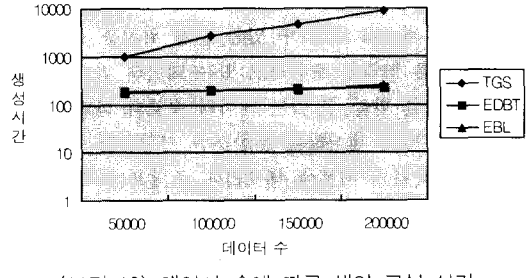
제안한 벌크 로딩 알고리즘에 대한 성능 평가를 위하여 기존에 제안된 벌크 로딩 알고리즘 중에서 색인 구성시간이 가장 적은 [11]에서 제안된 알고리즘과 100% 저장공간 활용률을 보장하면서 모든 차원에 대해 정렬을 수행하여 색인을 구성하는 TGS 알고리즘[8]에 대해 색인을 구성하였다. 벌크 로딩 알고리즘에 대한 모든 프로그램은 C++로 구현하였고 벌크 로딩에 의해 생성할 색인 구조로는 X-트리[4]로 선택하였다 또한 하나의 노드는 4KBytes로 하였다. 성능 평가를 위해 사용된 시스템은 Solaris 2.6.x의 운영 체제에서 Sun UltraSPARC-II의 366MHz CPU 2개를 장착하고 있으며 컴파일러로는 g++ 2.7.1을 사용한다. 벌크 로딩을 수행하기 위해 실제 영상에서 칼라 정보를 추출한 실제 데이터에 대하여 색인을 구성하는 시간과 검색 성능에 대해 비교, 분석한다. 제안하는 알고리즘과 [11]에서 사용되는 알고리즘은 벌크 로딩을 수행하기 위하여 캐쉬 메모리를 할당하여 사용하고 있다. 이러한 캐쉬 메모리로 32KBytes를 사용하여 구현하였고 [11]에서 제안한 알고리즘에 대한 분할 비율로 1 : 3을 사용하였다. 편의상 [11]에서 제안된 알고리즘을 EDBT 알고리즘이라 하고 제안하는 알고리즘을 EBL(Efficient Bulk Loading) 알고리즘이라 한다.

각 벌크 로딩 알고리즘에 대한 색인을 구성하는 시간을 비교하기 위하여 50차원의 실제 데이터를 50,000에서 200,000개까지 증가시키면서 각 벌크 로딩 알고리즘에 대한 색인 구성 시간을 비교하여 벌크 로딩을 수행하기 위한 데이터의 증가에 따른 색인 구성 시간을 비교하였다. 또한 차원의 변화에 따른 색인 구성 시간에 대한 변화를 측정하기 위하여 25차원, 50차원, 100차원 데이터에 대한 100,000개 데이터를 삽입하여 색인을 구성하는 시간을 비교하였다.

(그림 11)은 차원의 변화에 따른 색인 구성 시간의 변화를 나타낸다. (그림 11)에서 보는 바와 같이 TGS 알고리즘은 차원의 증가에 비례하여 색인을 구성하는 시간이 증가되지만 EDBT 알고리즘과 제안하는 EPS 알고리즘은 차원이 증가하여도 색인을 구성하는 시간은 거의 변화하지 않는다. (그림 12)는 데이터 수의 증가에 따른 색인 구성 시간의 변화를 나타낸다. 데이터를 정렬하지 않고 색인을 구성하는 EDBT 알고리즘과 제안하는 EBL 알고리즘은 데이터 수가 증가하여도 색인을 구성하기 위한 시간의 변화가 거의 없다. 그러나 TGS 알고리즘은 데이터 수의 변화에 비례하여 색인 구성 시간이 증가되는 것을 알 수 있다. 이러한 이유는 TGS 알고리즘이 주어진 데이터의 모든 차원을 정렬한 후 분할하여 색인을 구성하기 때문에 차원의 증가와 데이터의 변화에 따라 정렬을 수행해야 할 시간이 증가되지만 EDBT 알고리즘과 제안하는 알고리즘은 벌크 로딩을 수행하기 위해 모든 차원을 선택하는 것이 아니라 특정 차원만을 이용하여 분할을 수행하기 위한 위치를 선택하기 때문에 색인을 구성하기 위한 데이터 수의 변화에 대해서 큰 영향을 받지 않는다. 제안하는 알고리즘은 분할 위치를 선택하기 위해 일부 데이터를 정렬하여 색인을 구성하기 때문에 EDBT 알고리즘보다는 색인 구성 시간이 다소 증가한다. 그러나 이러한 시간은 색인을 구성하는 전체 시간에 비해 거의 무시해도 되는 시간이기 때문에 색인을 구성하는 시간은 [11]에서 제안된 벌크 로딩 알고리즘과 거의 비슷하다고 할 수 있다.



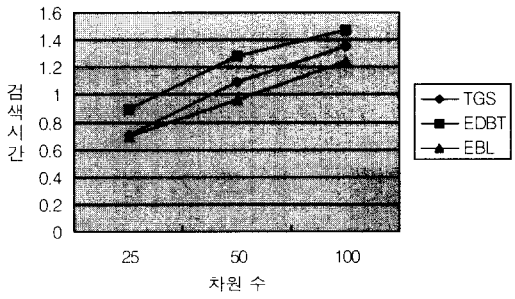
(그림 11) 차원에 따른 색인 구성 시간



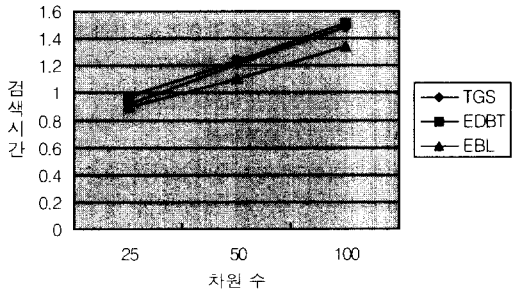
(그림 12) 데이터 수에 따른 색인 구성 시간

각각의 벌크 로딩 알고리즘을 이용하여 생성한 색인 구조에 대한 검색 성능을 비교하기 위하여 25차원, 50차원, 100차원의 10,000개의 데이터를 이용하여 트리를 생성하여 범위 질의(range query)과 k-NN(k nearest neighbor)질의에 대한 검색 시간을 비교한다. 검색 성능을 비교하기 위하여 모든 질의에는 주어진 데이터 영역에서 임의의 10개 데이터를 추출하고 이에 대한 평균적인 검색 시간을 비교한다. 범위 질의를 수행하기 위한 질의 범위를 0.1로 하였고 k-NN 질의를 수행하기 위하여 k를 10으로 정하여 탐색 성능을 비교한다.

(그림 13)은 차원의 변화에 따른 범위 질의에 대한 검색 시간의 변화를 나타낸다. (그림 13)에서 보는 것과 같이 EDBT 알고리즘은 차원이 증가함에 따라 검색 시간이 현저히 증가하지만 TGS 알고리즘과 제안하는 알고리즘은 차원이 증가하여도 검색 시간에 대한 증가는 크지 않다. TGS 알고리즘과 제안하는 알고리즘에서는 데이터의 분포 특성에 따라 데이터를 분할하지만 [11]에서는 데이터를 분할하기 위해 고정된 분할 비율을 선택하여 데이터를 분할하기 때문에 차원의 증가에 따라 영역 내에 데이터들이 존재하지 않는 Dead Space의 양이 점점 증가하기 때문이다. (그림 14)는 차원의 변화에 따른 k-NN 질의에 대한 검색 시간의 변화를 나타낸다. 저차원에 대해서는 TGS 알고리즘이 다른 벌크 로딩 알고리즘에 비해 성능이 우수하지만 차원이 증가할수록 제안하는 알고리즘에 대한 검색 시간이 TGS 알고리즘에 대한 검색 시간보다 우수한 것으로 나타났다. 실험 결과의 분석을 통해 본 논문에서 제안한 벌크 로딩 알고리즘이 색인을 구성하기 위한 시간을 단축하면서도 검색 성능이 우수함을 확인할 수 있었다.



(그림 13) 차원에 따른 범위 질의



(그림 14) 차원에 따른 k-NN 질의

5. 결 론

이 논문에서는 고차원 데이터에 대한 색인을 효과적으로 구성하기 위한 새로운 벌크 로딩 기법을 제시하였다. 제안하는 알고리즘은 주어진 데이터에 대해 최소한의 정렬을 수행하여 색인을 구성하는 시간을 단축하였고 전체 데이터 집합에 대한 특성을 파악하기 위해 분할을 수행해야 할 데이터에 대한 영역의 비와 영역 내에 존재하는 데이터들 간의 평균적인 거리를 이용하여 분할된 영역 내에 데이터들이 존재하지 않는 구간을 최소화하여 검색 성능을 향상시킬 수 있도록 하였다. 또한 검색을 수행하는 과정에서 불필요한 I/O 수를 줄이기 위해 트리에 대한 높이를 최소화하였고 이에 따른 저장공간 활용률 결정하는 방법을 제시하였다. 제안하는 알고리즘이 색인을 구성하는 시간을 단축하면서 우수한 검색 성능을 나타냄을 실험을 통해 증명하였다. 향후 과제로 실제 데이터베이스 하부 구조에 제안한 벌크 로딩 알고리즘을 구현하여 상용 데이터베이스 시스템에서 사용할 수 있도록 하는 연구가 진행되어야 한다.

참 고 문 헌

[1] Guttman A., "R-trees : A Dynamic Index Structure for Spatial Searching," ACM SIGMOD, pp.47-57, 1984.

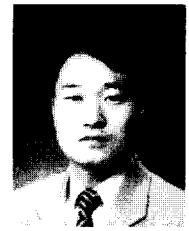
[2] Beckmann N., Kriegel H. P., Schneider R., Seeger B., "The R*-tree : An Efficient and Robust Access Method for Points and Rectangles," ACM SIGMOD, pp.322-331, May, 1990.

[3] K.I. Lin, H. Jagadish, and C. Faloutsos, "The TV-tree - An Index Structure for High Dimensional Data," VLDB Journal, Vol.3, pp.517-542, 1994.

[4] Berchtold S., Keim D. A., Kriegel H. P., "The X-tree : An Index Structure for High-Dimensional Data," VLDB Conference, pp.28-39, 1996.
[5] Roussopoulos N., Keifker D., "Direct Spatial Search on Pictorial Databases Packed R-trees," Proc. ACM SIGMOD Conference, pp.17-31, 1985.
[6] Kamel I., Falousos C., "On Packing R-trees," CIKM, pp.490-499, 1993.
[7] Leutenegger S. T., Lopez M. A., Edgington J., "STR : A Simple and Efficient Algorithm for R-Tree Packing," ICDE, pp.497-506, 1997.
[8] Garcia Y. J., Lopez M. A., Leutenegger S. T., "A Greedy Algorithm for Bulk Loading R-Trees," ACM GIS, pp.163-164, 1998.
[9] Van den Bercken J., Seeger B., Widmayer., "A General Approach to Bulk Loading Multidimensional Index Structures," VLDB Conference, pp. 406-415, 1997.
[10] Arge L., "The Buffer Tree : A New Technique for Optimal I/O-Algorithms," WADS, pp.334-345, 1995.
[11] Berchtold S., Bohm C., Kriegel H. P., "Improving the Query Performance of High-Dimensional Index Structures by Bulk-Load Operations," EDBT, pp. 216-230, 1998.
[12] Bially T., "Space-Filling Curves : Their Generation and Their Application to Bandwidth Reduction," IEEE Trans. on Information Theory, Vol.IT-15, No.6, pp.658-664, 1969.
[13] Lo M. N., Ravishankar C. V., "Generating Seeded Trees from Data Sets," SSD, pp.328-347, 1995.
[14] Berchtold S., Bohm C., Keim D. A., Kriegel H. P., "A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space," PODS, pp.78-86, 1997.

복 경 수

e-mail : ksbok@netdb.chungbuk.ac.kr
1998년 충북대학교 수학과(이학사)
2000년 충북대학교 정보통신공학과 (공학석사)
2000년~현재 충북대학교 정보통신공학과 박사과정
관심분야 : 데이터베이스 시스템, 자료 저장 시스템, 멀티미디어 데이터베이스, 정보검색 등



이 석 희

e-mail : seoklee@dab-c.ac.kr
1994년 충북대학교 정보통신공학과(공학사)
1998년 충북대학교 정보통신공학과(공학석사)
2000년 충북대학교 정보통신공학과 박사과정 수료
2000년~현재 동아방송대학 인터넷방송과 전임강사
관심분야 : 데이터베이스 시스템, 정보검색, 인터넷 방송, 분산 객체 컴퓨팅 등

유 재 수

e-mail : yjs@cbucc.chungbuk.ac.kr
1989년 전북대학교 컴퓨터공학과 (학사)
1991년 한국과학기술원 전산학과 (공학석사)
1995년 한국과학기술원 전산학과 (공학박사)
1995년~1996년 목포대학교 전산통계학과 전임강사
1996년~현재 충북대학교 공과대학 전기전자공학부 조교수
관심분야 : 데이터베이스 시스템, 정보검색, 멀티미디어 데이터베이스, 분산 객체 컴퓨팅 등

조 기 형

e-mail : khjoe@cbucc.chungbuk.ac.kr
1966년 인하대학교 전기공학과 (공학사)
1984년 청주대학교 산업공학과 (공학석사)
1992년 경희대학교 전자공학과 (공학박사)
1981년~1988년 충주공업전문대학 조교수
1996년~1999년 전기전자공학부장
1988년~현재 충북대학교 정보통신공학과 교수
관심분야 : 데이터베이스시스템, 화상처리 및 통신, 통신 프로토콜, 분산 객체 컴퓨팅 등