# $B^+$-트리 기반의 이동객체 색인 기법

## $B^+$-Tree based Indexing Method for Moving Object

서동민*, 유재수*, 송석일**
충북대학교 정보통신공학과*, 충주대학교 컴퓨터공학과**

Dong Min Seo(dmseo@cbnu.ac.kr)*, Jae Soo Yoo(yjs@cbnu.ac.kr)*,
Seok Il Song(sisong@cjnu.ac.kr)**

**요약**

이동객체 응용은 빈번하게 변경되는 이동객체의 위치정보를 효과적으로 처리할 수 있는 색인구조를 필요로 한다. 이동객체의 위치를 색인하기위해 제안된 색인기법들은 대부분 R-트리를 기반으로 하고 있다. R-트리는 변경 보다는 검색 연산의 성능에 초점이 맞추어진 색인구조이어서 잦은 변경을 다뤄야 하는 이동객체의 응용에 적합하지 않은 측면이 있다. 일부 연구에서는 R-트리의 변경 연산 성능을 향상시키기 위한 연구를 진행한 바 있다. 하지만, 변경 연산의 성능이 개선되었다 하더라도 R-트리가 기본적으로 내재하고 있는 동시성 제어기법 문제(동시성 제어 기법의 비효율성과 안정성) 때문에 R-트리 기반의 색인 기법을 실제 응용에서 쓰는 데는 여전히 문제가 있다. 이 논문에서는 B+-트리와 힐버트 곡선 (Hilbert Curve)를 기반으로 하는 새로운 이동객체 색인 기법을 제안한다. 기존에 제안된 B+-트리 기반의 색인 기법과는 다르게 이 논문에서는 힐버트 커브의 해상도(또는 차수, order)를 객체의 분포도와 개수에 따라서 가변적으로 적용하는 방법을 제안한다. 실험을 통해서 제안하는 색인 기법이 응답시간과 처리율 측면에서 기존 색인기법에 비해 우수함을 보인다.

■ 중심어 : | 이동객체 | 색인 | 빈번한 변경 |

**Abstract**

Applications involving moving objects require index structures to handle frequent updates of objects' locations efficiently. Several methods to index the current, the past and the future positions of moving objects have been proposed for the applications. Most of them are based on R-tree like index structures. Some researches have made efforts to improve update performance of R-trees that are actually focused on query performance. Even though the update performance is improved by researchers' efforts, the overhead and immaturity of concurrency control algorithms of R-trees makes us hesitate to choose them for moving objects. In this paper, we propose an update efficient indexing method that can be applicable for indexing the past, the current and the future locations. The proposed index is based on B+-Trees and Hilbert curve. We present an advanced Hilbert curve that adjusts automatically the order of Hilbert curve in subregions according to the data distribution and the number of data objects. Through empirical studies, we show that our strategy achieves higher response time and throughput.

■ keyword : | Moving Object | Index | Frequent Update |

* 본 논문은 2004년 정부(교육인적자원부)의 재원으로 한국학술진흥재단의 지원을 받아 수행된 연구입니다. (KRF-2004-003-D00300)

접수번호 : #070702-001
접수일자 : 2007년 07년 02일
심사완료일 : 2007년 07년 13일
교신저자 : 송석일, e-mail : sisong@cjnu.ac.kr

## I. 서 론

무선 통신 및 유비쿼터스 컴퓨팅 기술이 빠르게 발전하면서 이동객체 위치 추적을 기반으로 하는 응용이 급격히 성장하였다. 이에 따라 데이터베이스 분야에서는, 이동객체 응용들의 요구조건을 만족하기 위한 연구들이 진행되었다. 주요 연구 쟁점들 중 하나가 효과적인 이동객체 색인을 이용해서 이동객체 응용의 성능을 향상시키는 것이다.

일반적으로, 이동객체 응용은 이동객체의 위치가 지속적으로 변경되어 대량의 변경 연산을 필요로 한다. 이동객체 응용에서 대량의 변경연산을 반영하기 위해서는 색인구조 설계 시에 기존과 다른 쟁점이 발생한다. 그 중 하나가 대량의 변경연산을 처리하면서 동시에 다차원 질의를 효과적으로 처리하는 것이다. 이와 같은 이동객체 색인구조에 대한 요구사항은 다차원 데이터 색인기술에서 해결하기 쉽지 않은 문제로 대두되었다. 현재 저차원의 다차원 데이터를 색인하는 주요 색인기법은 R-트리 계열의 색인구조이다[1]. R-트리 계열은 주로 대용량의 정적인 데이터 집합에 최적화되었으며 변경연산 처리에 있어서는 낮은 성능을 보인다.

이에, R-트리 기반 색인구조들의 변경 성능을 향상시키기 위한 연구들이 제안되었다[2][3]. 이들 색인기법들은 변경 성능 향상을 위해서 각각 변경 지연 (lazy update) 기술과 상향(bottom-up)식 변경기법을 제시하였다. 하지만 여전히 변경 성능을 더욱더 향상해야 하는 과제가 남아 있었다. 특히, 대량의 변경 연산이 동시에 수행할 때 효과적으로 처리하지 못하는 $R^{link}$-트리와 같은 동시성 제어 알고리즘이 문제를 더욱더 악화 시킨다. 노드 분할과 MBR (Minimum Bounding Region) 변경의 반영을 위해서 트리를 거슬러 올라가는 연산이 빈번하게 되면 잠금 충돌(lock conflict)역시 빈번해 진다. 이 문제는 다차원 색인구조에서 공통적으로 나타나는 문제이다. 또 다른 문제는, 제안된 색인구조들이 기존의 데이터베이스 관리 시스템(DBMS)에 쉽게 통합되기가 어렵다는 점이다. 그 이유는 대부분의 상용 DBMS에서 R-트리 계열의 색인구조를 지원하지 않아서 R-트리 계열의 이동객체 색인구조를 통합하기 위해서는 DBMS 커널을 상당 부분 수정해야 한다.

이런 배경으로 공간 채움 곡선(Space Filling Curve)를 기반으로 이동객체의 위치데이터를 1차원으로 변환하여 $B^+$-트리를 통해 색인하는 방법들이 제안되었다[4][5]. $B^+$-트리를 사용하면 다음과 같은 장점을 얻을 수 있다. 첫 번째, $B^+$-트리는 기존의 상용 DBMS에서 폭넓게 사용되고 있을 뿐 아니라, 질의 및 변경에 대한 효율성과 안정성이 이미 증명되었다. 두 번째로, 1차원 색인구조를 이용하게 되므로 앞에서 언급한 기존의 R-트리 계열의 색인구조의 문제가 발생하지 않는다.

기존에 제안된 공간 채움 곡선을 기반으로 하는 색인기법들에서는 고정된 차수의 공간 채움 곡선을 사용한다. 즉, 색인 구성 시점에 결정된 차수는 변경이 되지 않는다. 하지만, 실세계에서 이동객체는 균등하게 분포되지 않을 뿐만 아니라 이동객체의 수 역시 변경된다. 공간 채움 곡선이 이런 데이터 분포에 유연하게 대처해서 특정 영역의 공간 채움 곡선의 차수가 다르게 적용된다면 보다 효과적인 색인이 가능 할 것이다. 이런 점에 착안하여 이 논문에서는 $B^+$-트리와 공간 채움 곡선을 기반으로 하되 차수가 영역별로 다르게 적용될 수 있는 새로운 색인 기법을 제안한다. 이를 통해 전체적인 검색 성능 저하를 가져오지 않으면서 색인구조의 크기를 효과적으로 줄여 변경 성능 향상을 기대할 수 있다.

이 논문은 다음과 같이 구성되어 있다. 2장에서는 기존에 제안된 관련 연구를 분석한다. 3장에서는 제안하는 색인구조를 설명하고 4장에서는 성능평가 결과를 통해 제안하는 색인 기법의 우수성을 보여준다. 마지막으로 5장에서 결론을 맺는다.

## II. 관련 연구

일반적으로, R-트리나 $R^*$-트리와 같은 기존의 다차원 색인구조들은 변경 연산 보다는 검색 성능 향상에 초점이 맞추어져 있다. 이런 종류의 색인구조들은 변경연산 보다는 검색 연산이 훨씬 더 많은 응용에는 적합하지만 이동객체 응용 분야와 같이 지속적으로 위치 정보가 변경되고 동시에 빈번한 질의가 발생하는 상황에는 부적합

하다.

이런 이유 때문에, 이동객체를 색인하기 위한 새로운 색인 구조들이 제안되었다. 이 색인구조들은 이동 객체의 과거 위치를 색인하는지 현재 또는 가까운 미래의 위치를 색인하는지에 따라서 분류해 볼 수 있다. 과거의 위치는 여러 선분으로 구성된 폴리라인을 통해서 추정할 수 있다. 폴리라인을 구성하는 선분은 R-트리와 같은 다차원 색인구조를 통해서 색인할 수 있지만, 어떤 선분들이 모여서 한 이동객체의 궤적이 되는지는 표현하기는 어렵다. 이를 해결하기 위해서 제안된 시공간 (Spatio-Temporal) R-트리는 한 이동객체의 궤적을 이루는 선분들을 그룹으로 관리할 수 있도록 제안되었으며 동시에 위치 정보도 표현이 가능하도록 되어 있다.

이동 객체의 현재 또는 가까운 미래의 위치를 색인하는 것은 과거의 위치를 색인하는 것과는 다르다. 이 경우에 이동 객체의 위치는 점이나 시간에 대한 함수로 표현된다. 점 데이터를 색인하는 색인구조로는 [2][3]을 들 수 있다. 변경 지연 R-트리는 이동객체의 빈번한 위치 변경을 효과적으로 처리하기 위해 제안된 색인구조이다[2]. R-트리에서 이동 객체의 위치정보 변화가 반드시 단말노드의 MBR 변경으로 이어지는 것은 아니다. 변경 지연 R-트리는 이동객체의 위치정보가 변경할 때 마다 삽입과 삭제연산으로 이루어지는 변경연산을 수행하지 않는다. 이동객체의 위치정보 변경으로 이동객체가 포함된 단말노드의 MBR을 벗어나지 않으면 간단히 단말 노드에 저장된 위치 정보만 변경한다. MBR을 벗어나더라도 일정한 수준이상 MBR이 변경 되지 않으면 MBR 변경만 수행하여 변경연산의 효율성을 높인다.

Lee 의 [3]에서도 R-트리의 변경연산 효율성을 높이기 위해서 상향식 변경 기법을 제안하고 있다. 변경을 수행하기 위해서는 색인구조에서 해당 이동객체를 찾아서 삭제하고 삽입을 하거나, 또는 위치 변경을 수행해야 한다. Lee 의 [3]은 보조 색인구조를 이용해서 R-트리를 순회하지 않고 바로 단말노드의 이동객체를 찾을 수 있는 방법을 제안하여 트리순회 횟수를 줄이고 있다.

시간에 대한 선형함수를 색인하는 대표적 색인구조로 [6]을 들 수 있다. Tao 의 [6]에서는 R-트리를 확장하여 시간에 대한 선형함수를 색인하는 TPR-트리를 제안하고 있다. 이동객체의 현재 위치는 선형함수에 현재시간을 적용해서 바로 구할 수 있다. MBR 역시 시간에 대한 함수를 통해서 표현할 수 있다. MBR의 하부 경계(lower bound)는 MBR에 포함된 모든 객체들 중에서 최대 속도로 하향 이동하는 객체를 통해 정해지고, 상부 경계(upper bound)는 최대 속도로 상향 이동하는 객체를 통해서 결정된다.

이상 언급한 기법들은 모두 R-트리를 기반으로 하고 있다. R-트리의 경우 이를 지원하는 동시성 제어 알고리즘이 동시에 수행되는 대량의 변경연산에 효과적이지 못하고 아직 안정화가 되지 않아서 상용 DBMS에서 사용하기에 어려운 점이 있다. 또한, 제안하는 기법들을 상용 DBMS에서 사용하기 위해서는 DBMS 커널을 많은 부분 수정해야 하는 어려움이 있다. 이런 점에 착안해서 상용 DBMS에서 지원하는 색인구조인 $B^+$-트리를 기반으로 하는 기법들이 제안되었다. [4]에서 제안하는 $B^x$-트리는 이동객체의 위치정보를 힐버트 곡선을 이용해 1차원으로 변환하여 $B^+$-트리를 통해서 색인한다. Yiu의 [5]에서 제안하는 $B^{dual}$-트리는 $B^x$-트리의 허위 적중(false hit)을 없애서 질의 성능을 향상시킨다.

이들 방법은 힐버트 곡선을 이용해서 2차원 이상의 데이터를 1차원으로 변환하고 있다. 변환을 위해서는 먼저 힐버트 곡선의 차수를 결정해야 한다. 이때, 차수를 너무 크게 주면 저장 공간이 많이 소요되고 너무 작게 주면 같은 값을 갖는 객체들이 많아져서 검색 성능이 저하된다. 따라서, 데이터 분포에 따라서 차수가 변경될 수 있다면 색인구조의 성능이 향상 될 것으로 예측할 수 있다. 결정된 차수는 변경이 될 경우 색인구조를 다시 구성해야 하므로 색인구조를 유지하는 동안 변경하기 어렵다.

## III. 제안하는 색인 기법

제안하는 색인기법의 이해를 돕기 위해서 제안하는 색인 기법의 기본적인 접근방법 설명을 통해서 힐버트 곡선을 이용해서 이동객체를 색인 할 때 어떤 문제가 있을 수 있는지 먼저 설명하고 최종적으로 이 논문에서 제안하는 색인 기법을 설명한다.

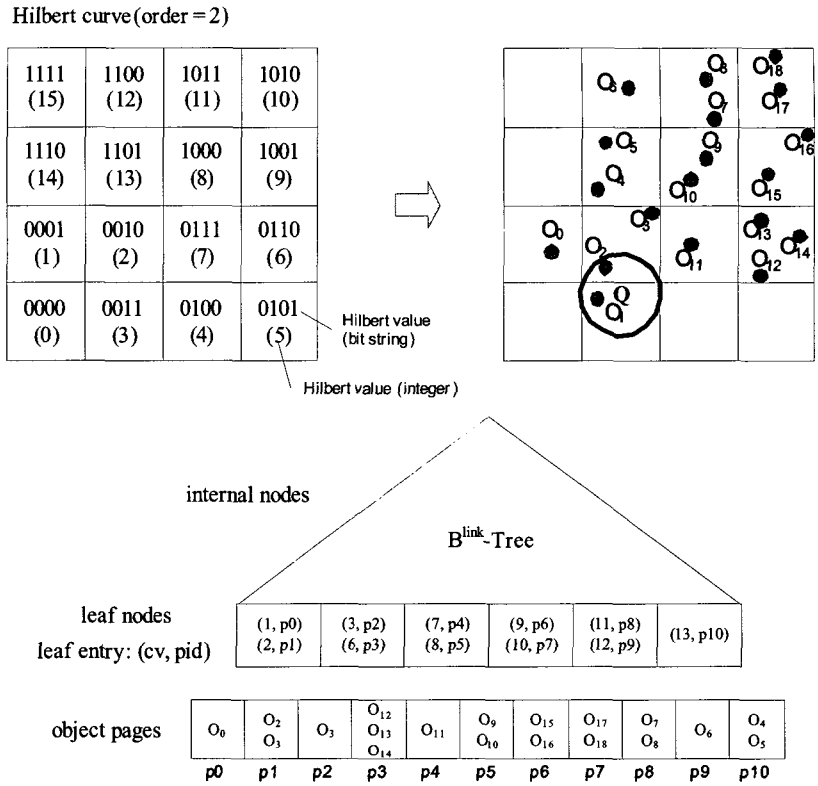


**그림 1. 제안하는 색인구조의 기본 구성**

## 1. 힐버트 곡선 기반의 기본적인 이동객체 색인

제안하는 색인 기법은 $B^{link}$-트리를 기반으로 한다[7]. $B^{link}$-트리는 $B^{+}$-트리가 동시성 제어기법을 지원하도록 수정된 버전이다. $B^{+}$-트리와는 다르게 $B^{link}$-트리는 단말 노드뿐아니라 중간노드들도 링크로 연결이 되어 있다. 중간노드는 디렉토리 노드 역할을 수행하고 오른쪽 형제 노드를 가리키는 포인터를 포함한다. 제안하는 색인 기법에서 $B^{link}$-트리의 단말 노드에는 색인대상이 되는 이동객체의 위치에 대한 힐버트 곡선 값을 저장한다.

힐버트 곡선은 여러 공간 채움 곡선중 하나이다. 공간 채움 곡선은 비연속적인 다차원 공간을 정확히 한 번씩 그리고 교차하지 않도록 방문하는 연속적인 경로를 말한다. 여러 종류의 공간 채움 곡선들이 존재하지만 피노 곡선 (Peano Curve, 또는 Z-Curve) 과 힐버트 곡선이 다차원 공간에서 객체들간의 관계를 1차원 상에서 가장 잘 유지 시켜주는 것으로 알려져 있다.

이미 밝힌바와 같이 제안하는 색인기법에서는 힐버트 곡선을 이용한다. 기존연구 [5][8]에서 피노 곡선보다 힐버트 곡선이 조금 더 좋다는 것이 밝혀진 바 있다. 제안하는 색인 기법에서 $B^{link}$-트리의 단말 노드 엔트리는 (cv, pid) 쌍으로 구성된다. cv는 이동 객체의 위치 정보에 대한 힐버트 곡선 값이고 pid 는 객체 페이지에 대한 페이지 ID(identifier)이다. 객체 페이지란 이동 객체들의 실제 정보를 저장하는 페이지 이다.

제안하는 색인 기법에서는 cv가 같은 이동객체들이 되도록 한 객체 페이지에 저장될 수 있도록 힐버트 곡선의 차수를 결정한다. 이를 통해서 $B^{link}$-트리의 크기를 줄일 수 있을 뿐 아니라 객체의 위치가 변경되더라도 힐버트 곡선의 값이 변경되지 않을 확률이 높아져서 잦은 변경에도 유연하게 대처할 수 있다. [그림 1]에서 보는 것

처럼 제안하는 색인 기법을 구성하는 것은 매우 간단하다. 그림에서 힐버트 곡선의 차수는 2 이다. 차수를 2로 하게 되면 2차원의 데이터 공간은 총 16개의 영역으로 나눠지고 각 영역에 하나의 cv 값이 할당 된다. $B^{link}$-트리의 단말 노드에는 cv 와 pid 쌍이 저장되며 단말 노드에는 최대 2개의 엔트리가 저장될 수 있다고 가정한다. 객체 페이지는 $B^{link}$-트리를 통해서 접근할 수 있으며 삽입하려는 객체의 cv를 키로 하여 객체를 삽입할 객체 페이지를 검색할 수 있다.

범위질의는 다음과 같이 수행된다. 먼저, 질의 범위에 해당하는 cv들을 구한다. [그림 1]에서 Q 가 범위 질의이다. 이 질의를 처리하기 위해서는 먼저 Q와 겹치는 영역들을 계산해 내야 한다. 그리고 각 영역의 cv 값을 구한다. [그림 1]에서 Q와 겹치는 영역의 cv 값은 2진수로 0010b과 0011b이다. 0010b과 0011b를 키로 하여 $B^{link}$-트리를 순회하여 객체 페이지 p1 과 p2를 디스크로부터 읽어 온다. p1 과 p2 에 저장된 객체들의 실제 위치정보를 이용해서 Q에 포함되는 객체들을 걸러낸다. [그림 1]에서 Q에 대한 최종 결과는 o1 과 o2 가 된다.

이상 설명한 접근 방법에서는 힐버트 곡선의 차수를 공간 활용도를 고려해서 결정하기 때문에 $B^{link}$-트리의 크기를 매우 작게 유지할 수 있다. 또한, $B^{link}$-트리의 변화가 매우 작다. 힐버트 곡선의 차수는 고정되고 $B^{link}$-트리의 단말 노드 엔트리에는 cv와 객체ID 가 아닌 cv와 페이지 ID가 저장되기 때문에 객체의 위치가 변한다고 하더라도 $B^{link}$-트리의 단말 노드 엔트리의 변경은 드물게 발생할 것이다. 객체의 위치정보 변경으로 발생하는 cv 의 변화는 객체 페이지들 사이에 객체를 이동시키기만 하면 된다.

하지만, 실제 응용에서는 이동객체가 균등하게 분포되지는 않는다. 특정 시점에는 특정 영역으로 객체들이 몰릴 수도 있고 전체 데이터 공간의 객체 개수가 증가하거나 줄어들 수도 있다. 이런 상황은 같은 cv를 갖는 이동객체의 숫자가 너무 많아져서 변별력을 떨어뜨리고 검색성능의 저하로 이어지게 될 것이다. 이런 문제는 힐버트 곡선의 차수를 최대로 하면 해결할 수 있다. 하지만, 그렇게 된다면 공간 활용도가 떨어져서 제안하는 색인 기법의 장점을 살릴 수 없게 된다. 특히, 최대 차수를 사용하게 될 경우 객체의 위치 이동으로 인한 cv의 변경이 더 빈번해 지게 될 것이다. [그림 2]는 각각 2차원 데이터 공간에서 힐버트 곡선의 차수를 1과 2로 했을 때 객체 o1의 cv 값 변경 회수를 보여주기 위한 예제 이다. 그림에서 보는 것처럼 차수가 2일 때 cv 값이 2번 변경되는데 반해서 차수가 1일 때는 변경되지 않는다.

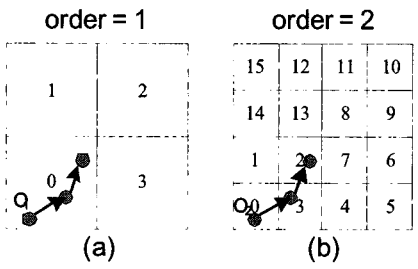


**그림 2. 힐버트 곡선의 차수에 따른 cv 의 변경 확률**

만일 힐버트 곡선의 차수가 특정 지역별로 데이터의 분포도나 객체의 수에 따라서 다르게 적용될 수 있다면 저장 공간 활용률을 높이면서도 검색성능을 향상할 수 있을 것이다. 하지만, 지역별로 다른 차수를 적용해서 힐버트 곡선을 구한다면 보조 자료구조를 이용하지 않고는 위치 정보를 cv로 변환할 수 없다. 별도의 자료구조는 또 다른 문제를 발생 시키므로 바람직하지 않다. 이 논문에서는 지역별로 다른 차수의 힐버트 곡선을 적용하면서도 별도의 자료구조 없이 $B^{link}$-트리의 순회만으로 필요한 객체 페이지를 찾을 수 있도록 하는 새로운 방법을 제안한다.

## 2. 제안하는 이동객체 색인 기법

[그림 3]에서 제안하는 색인구조의 전체적인 구축과정을 보여주고 있다. 색인을 구축하기 전에 기본 힐버트 곡선의 차수를 미리 결정한다. 기본 힐버트 곡선 차수는 사용자에 의해서 결정된다. 이 차수는 객체가 삽입되면서 지역적으로 차수가 데이터 분포에 따라서 변경된다. [그림 3]을 통해서 구축과정을 먼저 설명하고 이를 알고리즘화 해서 한번 더 설명한다.

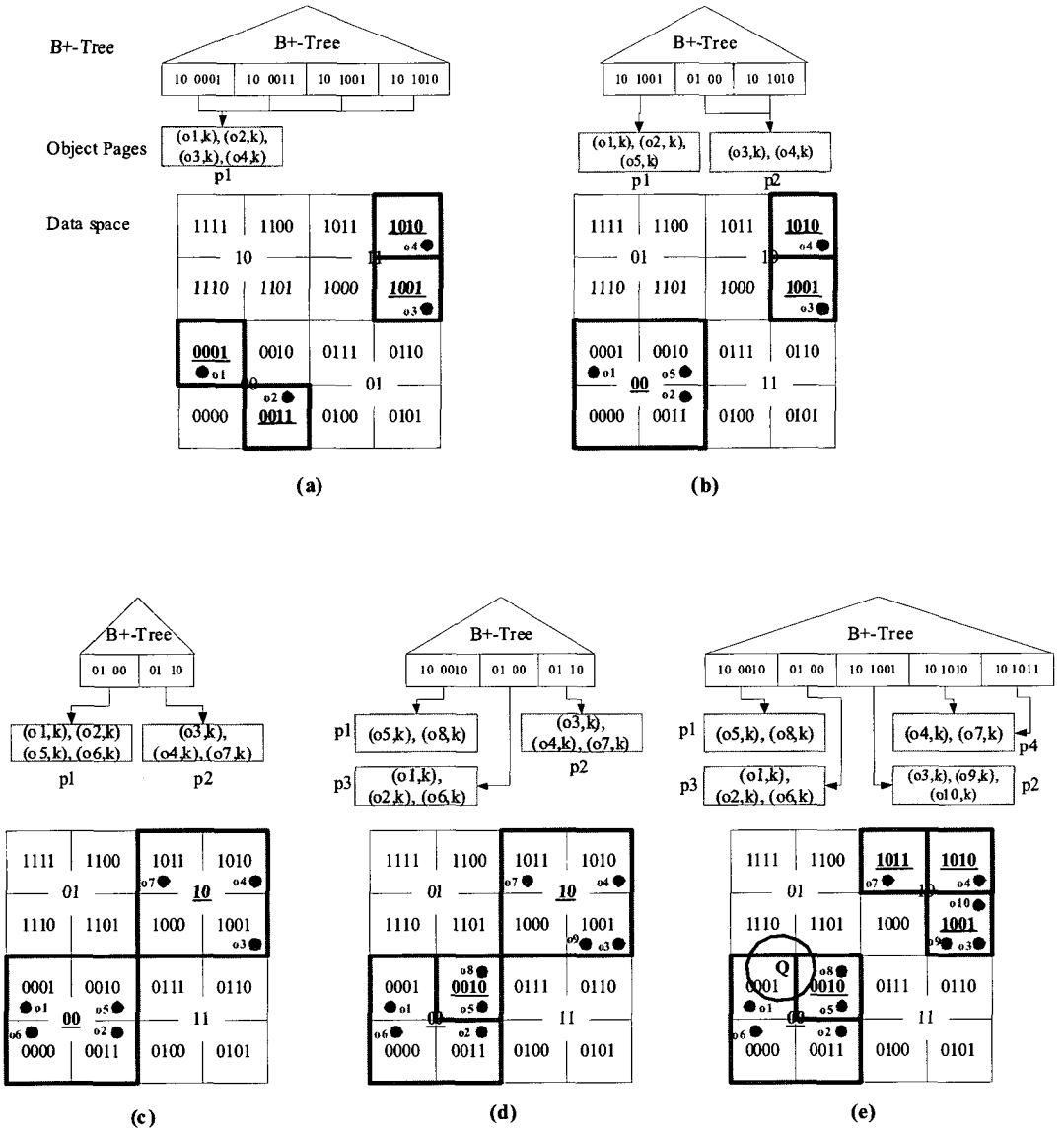


그림 3. 제안 하는 색인 구조의 구축 과정

### 2.1 제안하는 색인 구조의 구축과정

[그림 3]에서는 기본 힐버트 곡선 차수는 2, 객체페이지에 저장될 수 있는 최대 객체수는 4로 가정한다. [그림 3]의 (a)는 객체 o1, o2, o3, o4 가 삽입된 모습을 보여준다. o1을 삽입하기 위해서는 먼저 기본 힐버트 곡선 차수 2를 기준으로 cv 값을 계산한다. 이때 cv 값은 0001b 이다. [그림 1]에서는 이 cv를 객체 페이지 ID와 같이 $B^{link}$-트리의 단말노드에 저장을 했다. 하지만, 제안하는 색인기법에서는 지역마다 힐버트 곡선의 차수를 다르게 적용하기 위해서 차수를 같이 저장한다. 저장되는 key는 o+cv 로 표현할 수 있다. 여기서 o 는 차수를 말한다. 이 표현 방식에 따르면 o1의 key는 10b+0001b 가 된다.

[그림 3]의 (b) 는 (a) 의 색인구조에 o5 가 추가로 삽입된 모습을 보여주고 있다. 제안하는 색인기법에서는 객체의 키를 $B^{link}$-트리에 삽입할 때 단말노드에 이웃하는 엔트리들의 키 값을 확인하여 영역을 합병해서 부분적으로 차수를 1 감소시킬 수 있는지 확인한다. [그림 3]의 (b)에서 o5의 cv 는 0010b 이고 차수는 10b 이므로 키 값은 10b+0010b 가 된다. 키 값을 삽입할 단말 노드에 도달하게 되면 기본 차수보다 한 단계 낮은 차수의 키 (o5의 경우에는 01b+00b 가 됨)와 같은 값을 갖는 키를 찾는다. 그림에서는 10b+0001b, 10b+0011b 이 이에 해당한다. 만일 같은 값을 갖는 키가 2개 이상이면 병합 단계로 들어간다.

같은 값을 갖는 키가 2개 이상이라는 것은 그림에서 00b 에 해당하는 차수 1의 영역의 하위에 차수 2인 4개의 영역 중 3개의 영역에 해당하는 키가 존재 한다는 것을 의미한다. 만일 이 3개의 영역에 포함된 객체가 같은 객체페이지에 저장된다면 항상 같이 읽혀지기 때문에 굳이 각 영역마다 별도의 곡선 값을 할당할 필요가 없다. 이런 이유로 제안하는 색인기법에서는 이 영역의 차수를 기본차수에서 1 감소시켜서 o1, o2, o5 에 같은 키 01b+00b 를 할당해서 $B^{link}$-트리에 반영한다. 이를 반영하기 위해서 단말 노드에서 10b+0001b, 10b+0011b 을 삭제 한 후 새로 할당한 키를 삽입한다. 영역 병합의 결과로 검색성능에 영향을 주지 않으면서 $B^{link}$-트리의 크기를 줄일 수 있다.

다음으로 새로운 객체 o5를 객체 페이지 p1 에 삽입하기 위해서 페이지를 읽어온다. 먼저 객체 페이지에 새로운 객체를 삽입할 여유 공간이 있는지 확인한다. 여유 공간이 있을 경우 삽입하고 연산을 종료한다. 반면에 여유 공간이 없을 경우에는 객체 페이지를 분할하는 단계로 넘어간다. 검색성능을 위해서 객체 페이지의 분할은 같은 키 값을 갖는 객체들을 되도록 한 페이지에 저장할 수 있도록 수행한다. 같은 키 값을 갖는 객체들을 한 페이지 유지하는 것이 어렵다면 다시 차수를 높여서 분할하는 방법을 택한다. 이 부분에 대해서는 [그림 3]의 (d) 와 (e)에서 추가로 설명한다. [그림 3]의 (b) 에서는 같은 키를 갖는 객체들을 한 페이지에 저장하도록 하는 원칙에 따라서 o1, o2, o5 를 p1에 o3, o4를 p2에 저장한다. 이에 따라서 단말 노드의 엔트리에 pid 를 조정한다.

[그림 3]의 (c)는 객체 o6 와 o7이 삽입 된 후의 모습을 보여준다. o6를 삽입하기 위해서는 먼저 키 값을 구해야 한다. o6의 키는 차수 와 커브 곡선 값을 결합해서 10b+0000b 이 된다. 이 키 값을 가지고 o6를 삽입할 객체 페이지를 결정해야 한다. [그림 3]의 (b)에서 본 것처럼 제안하는 색인 기법에서는 더 이상 같은 차수의 키 값으로만 색인구조가 구축되지 않는다. 따라서, 객체 페이지를 결정하기 위해서 $B^{link}$-트리를 순회할 때 키 값들 사이의 단순 비교로는 원하는 페이지를 찾을 수 없다. 이에 따라, 제안하는 색인 기법에서는 [그림 4]와 같이 $B^{link}$-트리를 순회할 때 사용할 새로운 키 비교 함수를 제안한다.

```
Function : Compare
  Input : key1, key2
  Output : result (compResult, orderDiff)
Begin
  result.orderDiff = false;
  if ( key1.order < key2.order )
    key2.cv = key2.cv 를 Key1.order 에 맞추어 변환;
    Result.orderDiff = true;
  else If ( key1.order > key2.order )
    key1.cv = key1.cv 를 key2.order 에 맞추어 변환;
    Result.orderDiff = true;
  end If
  result.compResult = key1.cv - key2.cv;
  result를 반환하고 Compare 함수 종료;
End
```

**그림 4. 제안하는 색인기법의 키 비교 함수**

[그림 4]의 비교함수의 출력 result 는 두 키 값의 대소 비교 결과를 0(같음), +(key1이 더 큼), -(key2가 더 큼)로 알려주는 comResult 와 두 키의 차수 차이 여부를 true 와 false 로 알려주는 orderDiff로 구성된다. 삽입할 객체 페이지를 찾기 위해서는 우선적으로 orderDiff가 false 이면서 값이 같은 키를 찾는다. 즉, 차수가 같으면서 곡선 값도 같은 완전히 일치하는 키를 말한다. 이를 발견하지 못하면 다음 순서로 orderDiff가 true 면서 값이 같은 키를 찾는다. 즉, 차수가 1 작을때 키 값이 일치하는 키를 찾는다. [그림 3]의 (c)에서 o6의 키를 가지고 트리를 순회하면 [그림 4]의 비교 함수에 따라서 01b+00b이 같은 값을 갖는 키로 결정이 될 것이다. 객체

페이지에 삽입을 위해서 p1 페이지를 읽고 삽입할 여유 공간이 있는지 확인한다. 여유 공간이 있으므로 삽입을 하고 삽입 연산을 끝낸다.

계속해서 o6를 삽입한 후에 o7을 삽입한다. o7의 키는 10b+1011b 이다. $B^{link}$-트리의 단말 노드에 도착했을 때 그림 4의 비교함수를 통해서 10b+1010b 보다 크다는 것을 알 수 있다. 다음에 할 일은 o7을 어떤 객체 페이지에 삽입할지를 결정하는 것이다. 하지만, 앞에서 설명한 것처럼, 먼저 영역 병합이 가능한지 확인한다. 이를 위해 기본 차수에서 1 감소시킨 키 값을 구해서 이웃하는 엔트리들 중에서 같은 키를 갖는 엔트리가 2개 있는지 확인한다. 확인을 위해서는 최대 앞/뒤로 2개의 엔트리를 확인하면 된다. [그림 3]의 (c)에서는 10b+1010b 과 10b+1001b의 (기본차수 - 1) 에 대한 키 값이 일치한다. 따라서, 10b+1010b, 10b+1001b, 10b+1011b 을 병합해서 01b+10b 의 키를 갖는 엔트리를 생성해서 $B^{link}$-트리에 삽입한다. o7을 삽입할 객체페이지는 되도록 같은 키를 갖는 객체를 같은 객체페이지에 삽입하는 원칙에 따라서 p2에 삽입한다.

[그림 3]의 (d)는 o8과 o9을 삽입한 후의 모습을 보여주고 있다. o8의 키는 10b+0010b 이다. 이 키를 가지고 트리를 순회하면 단말 노드에서 orderDiff가 true 이면서 키 값이 같은 엔트리를 찾을 수 없으므로 orderDiff가 false 이면서 키 값이 같은 01b+00b 을 키로 하는 객체 페이지에 삽입할 것을 결정한다. 하지만 p1에는 객체를 삽입할 여유 공간이 없으므로 p1을 분할해야 한다. 분할의 최우선 원칙은 되도록 같은 키 값을 갖는 객체들이 같은 객체 페이지에 저장되도록 하는 것이다.

[그림 3]의 (d)에서 보듯이 힐버트 곡선 값 00b 에 해당하는 영역을 분할하려면 4개의 하위 영역으로 분할하고 키를 구하기 위한 힐버트 곡선 차수를 1 증가해야 한다. 그러나 분할된 4개의 영역을 보면 0010b 에 해당하는 영역을 제외한 나머지 영역들은 1개의 객체가 위치한다. 따라서, p1을 4개의 페이지로 분할하면 공간 활용률이 저하된다. 이 경우에는 가장 많은 객체를 포함하고 있는 영역을 하나의 객체 페이지에 저장하고 나머지 3 영역의 객체들을 다른 페이지에 저장한다. 즉, 최대한 같은 키를 갖는 객체를 한 페이지에 저장하면서 동시에 되도록 비슷한 개수의 그룹으로 나눠지도록 객체들을 분리한다. 각 영역의 키는 o5와 o8이 포함된 영역에 대해서는 10b+0010b, o1, o2, o6 가 포함된 나머지 세 영역은 앞서 설명한 원칙에 따라서 하나의 영역으로 하고 키를 01b+00b 로 한다. o9 의 삽입은 o6의 삽입과 같은 경우이므로 설명을 생략한다.

[그림 3]의 (e)는 또 다른 경우의 객체페이지 분할을 보여준다. 키가 10b+1001b 인 객체 o10을 삽입하기 위해서 트리를 순회하면 orderDiff 가 false 이면서 같은 키 값을 갖는 엔트리가 없으므로 orderDiff 가 true 이면서 같은 키 값을 갖는 엔트리를 찾으면 01b+10b 이 선택 된다. 하지만 역시 p2 에 여유 공간이 없으므로 분할을 수행해야 한다. 10b에 해당하는 영역을 4개의 하위 영역으로 나누어 보면 1001b 에 3개의 객체가 포함되어 있고, 1011 과 1010 에 각각 1개씩의 객체가 포함되어 잇다. 분할 원칙은 되도록 같은 키를 갖는 객체들을 같은 페이지에 저장되도록 하면서 두 그룹에 속하는 객체의 수가 되도록 같게 하는 것이다. 이 원칙에 따르면 가장 많은 객체를 포함하는 영역에 포함된 객체를 하나의 객체페이지에 저장한다. 나머지 영역이 둘 뿐이므로 병합을 하지 않고 각각의 영역에 대해서 키를 생성하고 두 영역의 객체들을 같은 페이지에 저장한다.

### 2.2 삽입 알고리즘

이상 각각의 경우에 대해서 설명한 삽입 연산을 알고리즘 형태로 정리한 것이 [그림 5]이다. 삽입 연산은 Insert 함수로부터 시작한다. 가장 먼저 Locate-ObjectPage 함수를 호출하여 새로운 객체를 삽입할 객체 페이지를 결정하고 객체페이지에 여유 공간이 있으면 삽입하고 여유 공간이 없을 경우에는 SplitObjectPage를 호출하여 객체페이지를 분할한다.

LocateObjectPage는 먼저 삽입하려는 객체의 위치정보를 이용해서 키 값을 구한다. 키 값을 이용해서 $B^{link}$-트리를 순회하여 삽입할 객체페이지를 결정해 나간다. $B^{link}$-트리를 순회하는 알고리즘은 별도로 설명하지 않는다. 기존 $B^{link}$-트리와의 유일한 차이점은 키 값을 비교할 때 [그림 5]에 제시한 비교함수를 사용한다는 것이다. 중간 노드에서는 Compare 함수가 반환하는 result의

```
Function : Insert
Input : obj
Begin
    entry = 함수 LocateObjectPage 호출;
    objectPage = entry.pid 페이지를 디스크로 부터 읽어 옴;

    if (objectPage 에 obj를 삽입할 여유공간 없음)
        objetPage, obj를 입력으로 하여 함수 SplitObjectPage 호출;
        Insert 함수 종료;
    else
        objectPage에 obj 삽입;
        Insert 함수 종료;
    end if
End

Function : LocateObjectPage
Input : obj
Output : pid, leafPage
Begin
    key.cv = obj의 위치정보를 이용해서 힐버트 곡선 값을 계산;
    key.order = initOrder;  //initOrder 는 사용자가 지정하는 기본
                힐버트 곡선 차수;
    leafPage = Blink-트리 순회 결과로 획득한 단말노드;
    for (entry = leafPage 의 i 번째 엔트리)
        if (key.order와 차수가 일치하고 cv가 같은 단말 엔트리 ||
              key.order와 차수가 일치하지 않지만 cv가
              같은 단말 엔트리)
            entry, leafPage를 반환하고 LocateObjectPage 종료;
        else if (key 보다 큰 키를 갖는 엔트리)
            entry = i, initOrder, key, leafPage 와 함께
                    MergeArea 호출;
            entry, leafPage를 반환하고 LocateObjectPage 종료;
        end if
    end for
End

Function : MergeArea // 영역병합, 재귀함수
Input : i, order, key, leafPage
Output : entry
Begin
    entry = leafPage의 i번째 엔트리;
    order = order - 1;
    tempCV = order 에 대한 key.cv의 힐버트 곡선 값 재 계산;
    if (leafPage의 entry 앞/뒤의 엔트리에 대해서 key.order 와
            차수가 같으면서 order 에 대한 힐버트 곡선 값이
            tempCV 와 같은엔트리 수가 2 ‖ 2개의 엔트리가
            같은 pid 를 가짐 )
        tempEntry.key.cv = tempCV; tempEntry.key.order = order;
        tempEntry.pid = pid;
        조건을 만족하는 2개의 엔트리를 단말노드에서 삭제하고
            tempEntry를 단말노드에 삽입;
        entry = leafPage에서 tempEntry의 위치, order,
                tempEntry.key, leafPage 와 함께 MergeArea 호출;
        entry를 반환하고 LocateObjectPage 종료;

    else
        entry를 반환하고 LocateObjectPage 종료;
    end if
End

Function : SplitObjectPage
Input : leafPage, objectPage, obj
Begin
    tempArr[].entry = leafPage에서 objectPage를 pid로 하는
                      엔트리들을 찾아 낸다;
    tempArr[].cnt = objectPage에서 tempArr[] 의 각
  엔트리에
                    해당하는 객체들의 개수를 계산;
    while (객체를 골고루 나눌때 까지)
        tempArr[]을 cnt를 기준으로 정렬하고 최대한 두 페이지에
            같은 수의 객체가 포함될 수 있도록 나눔;
        if (한 페이지의 객체수가 저장가능한 최대 객체수의
            30%~70% 가 되지 않는다면)
            tempArr[] 의 첫번째 엔트리의 차수를 감소시켜
                영역을 분할하고 생성되는 엔트리와 각 영역에

                포함되는 객체의 수로 tempArr[] 재구성;
        else
            나누어진 객체를 새로운 객체페이지를 할당해서 나누고,
                leafPage에서 엔트리 변경내용을 조정함;
        end if
    end while
End
```

**그림 5. 삽입 알고리즘**

orderDiff 가 false 인 경우에는 compResult 의 결과를 그대로 따르지만 orderDiff 가 true 인 경우 compResult 가 0 이면 key1 이 더 작다고 판단한다. 단말 노드에서는 [그림 6]의 알고리즘에 따라서 판단하게 된다.

$B^{link}$-트리를 순회해서 단말노드 leafPage에 도달하면 leafPage의 각 엔트리와 삽입하는 객체의 key 를 비교하면서 객체를 삽입할 객체 페이지를 결정한다. 가장 우선이 되는 기준은 Compare 함수의 결과 orderDiff 가 true 이면서 compResult 가 0 인 것을 찾는다. 찾지 못하면 orderDiff가 false 이면서 compResult가 0 인 엔트리를 찾는다. 이에 해당하는 엔트리도 없으면 orderDiff 와 관계없이 보다 큰 키를 갖는 엔트리를 만나면 멈춘다. 이 과정은 이웃하는 단말 노드로 이동하면서 이루어질 수 있다. compResult가 0 인 엔트리 entry를 찾으면 leafPage와 함께 entry를 반환하면서 함수를 종료한다. 찾지 못하면 영역의 병합 여부를 확인하기 위해서 MergeArea를 호출한다.

함수 MergeArea는 현재 위치에서 앞뒤로 이동하면서 병합이 가능한지 확인하고 병합시 이에 따른 엔트리 조정을 한다. 병합이라는 것은 부분적으로 힐버트 곡선의

차수를 1 감소시키고 하위 영역에 포함된 객체들의 커브 값을 하나로 하는 과정을 말한다. 이를 위해서 먼저 현재 위치에서 앞뒤로 이동하면서 차수를 1 감소시켰을 때 삽입하려는 객체의 키와 같은 키를 갖는 엔트리가 있는지 확인한다. 병합은 차수를 1 감소 시켰을 때 같은 엔트리가 2개 존재하며 이들이 모두 같은 객체 페이지에 저장될 때 수행한다. 엔트리가 2개라는 것은 [그림 3]의 (b)에서 설명한 경우이다. 즉, 해당 영역의 75% 에 객체가 존재하고 이들이 같은 객체 페이지에 저장되는 경우에는 질의를 처리하기 위해서 어짜피 읽어야 하는 객체들이므로 다른 키를 부여할 필요 없이 하나의 키를 부여해서 Blink-트리의 저장 공간 효율성을 높이려는 것이다. 병합을 해야 하는 경우에는 병합에 따른 엔트리를 새로 생성하고 이를 단말 노드에 반영한다. MergeArea 함수는 바로 종료되지 않고 차수를 줄인 새로운 엔트리와 병합될 엔트리가 있는지 다시 재귀 호출로 확인한다. 병합이 완료되면 새로운 객체에 대한 엔트리를 반환하고 종료한다.

범위질의 처리는 상대적으로 간단하다. [그림 3]의 (e)의 범위질의 Q를 처리하는 과정을 예로 설명한다. 가장 먼저 범위질의 Q와 겹치는 모든 영역에 대해서 차수가 기본차수 인 힐버트 커브 값을 구하고 키를 생성한다. 그림에서 Q는 10b+0010b, 10b+0010b, 10b+0100b, 10b+0111b 의 키들로 변환될 수 있다. 범위질의 Q를 처리하기 위해서는 변환된 4개의 키에 대해서 4개의 단순 정확한 비교(exact match) 질의를 수행함으로써 처리할 수 있다. 하지만 불필요하게 이미 접근한 노드를 반복해서 접근할 수 있어 비효율적이다.

제안하는 색인 기법에서는 4개의 키를 분석해서 하나 또는 두 개 이상의 범위질의로 변환해서 처리한다. 키들을 분석해서 가장 최소 키 값과 최대 킷값의 범위가 두 개의 $B^{link}$-트리 단말노드를 접근하면 처리할 수 있다고 예상될 때 하나의 범위질의로 처리한다. 두 개의 단말노드를 접근할 것으로 예상되는 범위질의를 둘로 분리하더라도 두 번째 범위질의 처리시에 현재 단말 노드의 이웃 단말 노드를 접근하게 되어서 결과적으로 비용이 같아지기 때문이다. 범위를 분석해서 세 개 이상의 단말노드를 접근해야 하면 두 개 이상의 범위질의로 분리해서 처리한다. [그림 3]의 (e)에서 $B^{link}$-트리의 단말노드에 최대 4개의 엔트리를 저장할 수 있다고 가정한다면 최소 키 와 최대 키의 범위가 8이 되는 경우까지 하나의 범위 질의로 처리하게 된다. 10b+0010b 과 10b+0111b의 범위가 6이므로 Q는 $B^{link}$-트리에서 하나의 범위질의로 처리된다. 먼저 10b+0010b 과 일치하는 키를 갖는 엔트리 또는 큰 키를 갖는 엔트리를 찾는다. 첫 번째 엔트리가 10b+0010b을 키로 하기 때문에 이 키가 가리키는 p1을 읽는다, 두 번째 엔트리는 01b+00b을 키로 갖는데 그림 4의 비교함수에 의하면 같은 키가 되므로 이 키가 가리키는 p3를 읽는다. 다음은 0100b부터 0111b까지 인데 일치하는 키를 갖는 엔트리가 없으므로 10b+1001b을 키로 하는 엔트리를 읽게 되면 범위질의를 멈춘다. 읽어온 p1 과 p3 의 객체들의 위치정보를 이용해서 범위 질의 Q에 해당되는 객체들을 최종적으로 걸러낸다.

## IV. 성능평가

### 1. 실험 환경

이 논문에서는 가장 최근에 발표된 B-트리기반의 이동객체 색인 기법인 $B^{dual}$-트리, 그리고 R-트리 기반의 색인구조인 TPR-트리와 제안하는 색인구조를 다양한 실험을 통해서 비교한다. 모든 실험은 펜티엄 IV 3GHz CPU, 1GB 의 주기억장치를 갖는 컴퓨터에서 수행되었다. 구현하는 색인구조의 페이지 크기는 모두 1Kbyte로 고정하였다. 이 논문에서는 데이터 집합을 [5][6]에 따라서 생성하였다. 데이터 공간은 2 차원이며 각 차원의 영역은 0 ~ 1000 으로 하였다. [9]에서 제공하는 128,000 개의 데이터 집합으로부터 5000 개의 직사각형을 뽑아내었다. 이 사각형은 공항을 의미한다. 이동 객체는 항공기로 가정하였으며 임의의 공항에서 임의로 선택된 다른 공항으로 이동하는 것으로 가정하였다. 이동한 항공기는 다시 임의의 다른 공항을 선택해서 이동한다. 범위 질의는 0.01% ~ 0.05% 의 크기로 생성되며 균등 분포의 특성을 갖는다. 범위 질의의 성능 측정은 100 회의 질의의 평균값을 구한다.

## 2. 저장 공간 비교

[그림 4]는 제안하는 색인구조와 TPR-트리, $B^{dual}$-트리의 저장 공간을 비교한 것이다. 저장 공간의 크기는 전체 객체의 개수를 변경시켜 가면서 측정하였다. 그림에서 보이는 것처럼, 제안하는 색인기법이 사용하는 저장 공간의 크기가 훨씬 작음을 알 수 있다. 제안하는 색인기법은 힐버트 곡선의 차수가 항상 가능한 최댓값으로 고정되는 것이 아니고, 데이터의 분포에 따라서 차수가 더 낮아지므로 저장 공간의 사용도가 훨씬 작다. 저장 공간 측면의 특징은 이동객체 색인 기법에서 중요하다. 갱신 및 질의 처리 성능을 향상시키기 위해서 색인구조 전체를 주기억장치에 상주시키는 선택을 할 수도 있기 때문이다.

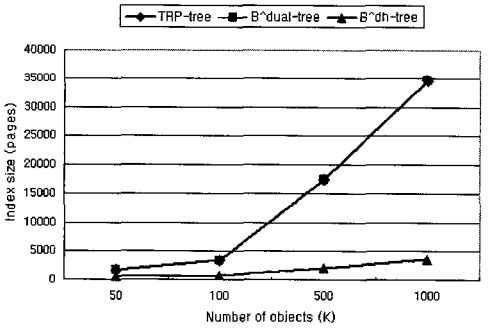


**그림 6. 인덱스 크기 비교**

## 3. 변경연산 성능 비교

[그림 7]은 제안하는 색인기법과 $B^{dual}$-트리와 TPR-트리의 변경 연산 성능을 비교한 것이다. 실험에서는 객체의 개수를 50,000 개에서 1,000,000 개로 증가시키면서 페이지 접근 회수를 측정하였다. 총 5000개의 변경연산을 수행하였고 제시된 결과는 평균 페이지 접근회수 이다. 그림에서 볼 수 있는 것처럼 TPR-트리보다 제안하는 색인 기법이나 $B^{dual}$-트리의 성능이 월등히 뛰어 났다. 또한, 제안하는 색인 기법은 $B^{dual}$-트리에 비해서도 약 1.5배에서 2 배의 성능향상을 보였다. 기본적으로 R-트리 계열의 색인 구조에 비해서 겹침이 없다는 것이 B-트리 계열의 성능향상 요인이라고 볼 수 있다. $B^{dual}$-트리에 비해서 제안하는 색인기법의 전체 색인구조의 크기가 작으므로 소요되는 페이지 접근 회수도 작아진다는 것을 알 수 있다.

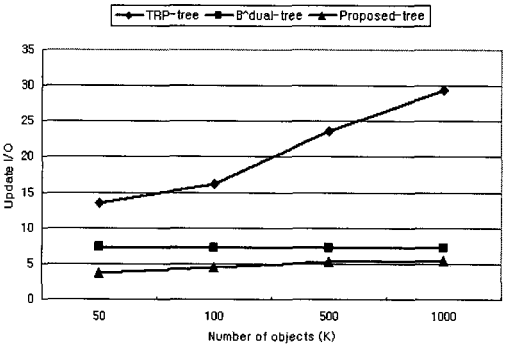


**그림 7. 데이터 크기에 따른 변경 비용**

## 4. 범위질의 성능 비교

[그림 8]은 제안하는 색인구조의 범위질의 성능을 비교하고 있다. 객체의 개수를 5,000 개에서 1,000,000 까지 증가시키면서 범위 질의의 페이지 접근 회수를 측정한 결과 이다. 범위질의의 크기는 0.03%로 하였으며 100개의 질의 수행결과 측정된 페이지 접근회수의 평균을 내었다. 그림에서 보는 것처럼 제안하는 색인구조가 객체의 개수가 증가할수록 다른 색인구조에 비해서 우수한 성능을 보인다. 그 이유는 객체의 수가 증가하더라도 실제 색인구조의 크기는 가변적인 차수의 적용으로 증가의 폭이 작기 때문이다.

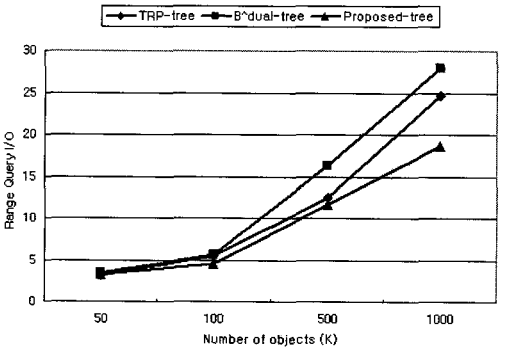


**그림 8. 데이터 크기에 따른 범위 질의 비용**

## V. 결론

이 논문에서는 $B^+$-트리 기반의 새로운 이동객체를 위한 색인구조를 제안하였다. 제안하는 색인구조는 기존의 B-트리 계열의 색인구조와는 다르게 공간 채움 곡선의 차수를 데이터 분포에 따라서 부분별로 다르게 적용할 수 있다. 그 결과 색인구조의 크기가 줄어들고 이에 따른 변경성능이나 검색성능이 모두 향상되었다. 또한, 특정 영역의 객체들이 한 페이지에 저장되는 경우 차수를 하향 조정하여 되도록 객체들의 공간 채움 곡선 값을 같도록 하여 객체가 해당 영역 내에서 이동할 때는 색인구조에 별도의 변경연산을 수행할 필요가 없어 변경 연산의 비용이 더 줄어들게 된다. 실험을 통해서 제안하는 색인구조의 우수성을 보였다. 비교 대상으로 가장 최근에 제안된 B-트리 기반의 색인구조인 $B^{dual}$-트리와 대표적인 R-트리 계열의 색인구조인 TPR-트리를 사용하였다. 실험 결과 제안하는 색인구조가 변경 연산이나 질의 성능 모두 다른 색인구조에 비해 우수함을 볼 수 있었다.

## 참 고 문 헌


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

[2] D. Kwon, S. Lee, and S. Lee, "Indexing the Current Positions of Moving Objects Using the Lazy Update R-tree," In Proceedings of MDM, pp.113-120, 2002.

[3] M. L. Lee, W. Hsu, C. S. Jensen, and K. L. Teo, "Supporting Frequent Updates in R-trees: A Bottom-up Approach," In Proceedings of VLDB, pp.608-619, 2003.

[4] C. S. Ensen, D. Lin, and B. C. Ooi, "Query and Update Efficient $B^+$-Tree Based Indexing of Moving Objects," In Proceedings of VLDB, pp.768-779, 2004.

[5] M. L. Yiu, Y. Tao, and N. Mamoulis, "The $B^{dual}$-tree: Indexing Moving Objects by Space Filling Curves in the Dual Space," submitted to VLDB Journal, 2006.

[6] Y. Tao, D. Papadias, and J. Sun, "The TPR-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries," In Proceedings of VLDB, pp.790-801, 2003.

[7] P. L. Lehmann and S. B. Yao, "Efficient Locking for Concurrent Operations on B-Trees," Journal of Acm TODS, Vol.6, No.4, pp.650-670, 1981.

[8] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," TKDE, Vol.13, No.1, pp.124-141, 2001.

[9] http://www.rtreeportal.org


## 저 자 소 개


**서 동 민(Dong Min Seo)** 정회원

- 2002년 2월 : 충북대학교 정보통신공학과 (공학사)
- 2004년 2월 : 충북대학교 정보통신공학과 (공학석사)
- 2004년 3월 ~ 현재 : 충북대학교 정보통신공학과 박사과정

<관심분야> : 데이터베이스 시스템, 에이전트 시스템, XML, 이동 객체 데이터베이스, 시공간 색인구조

**유 재 수(Jae Soo Yoo)** 종신회원

- 1989년 2월 : 전북대학교 컴퓨터공학과 (공학사)
- 1991년 2월 : 한국과학기술원 전산학과 (공학석사)
- 1995년 2월 : 한국과학기술원 전산학과 (공학박사)
- 1995년 3월 ~ 1996년 8월 : 목포대학교 전산통계학과 전임강사
- 1996년 8월 ~ 현재 : 충북대학교 전기전자컴퓨터공학부 교수

<관심분야> : 데이터베이스 시스템, 정보검색, 멀티미디어 데이터베이스 분산 객체 컴퓨팅, 센서 네트워크

**송 석 일(Seok Il Song)** 정회원

- 1998년 2월 : 충북대학교 정보통신공학과 (공학사)
- 2000년 2월 : 충북대학교 정보통신공학과 (공학석사)
- 2003년 2월 : 충북대학교 정보통신공학과 (공학박사)
- 2003년 8월 ~ 현재 : 충주대학교 컴퓨터공학전공 교수

<관심분야> : 데이터베이스 시스템, 센서 네트워크, 이동객체