비공간 검색 조건이 포함된 k-최근접 질의 처리를 위한 R-트리와 시그니쳐 화일의 결합

(Combining R-trees and Signature Files for Handling k-Nearest Neighbor Queries with Non-spatial Predicates)

박 동 주 † 김 형 주 ††

(Dong-Joo Park) (Hyoung-Joo Kim)

요 약 멀티미디어 데이타베이스에서 k-최근접 질의는 가장 일반적이며, 비공간 검색 조건이 포함된 경우가 많다. 현재까지 이러한 질의를 위한 여러 기법 중에서 Hjaltason과 Samet이 제안한 점증적 최근접 알고리즘이 가장 유용하다고 알려져 있다[1]. 질의 처리를 위해 상위 연산자가 k보다 많은 객체를 요구할 때, 이 알고리즘은 처음부터 질의를 재실행하지 않고 다음 객체를 전달할 수 있기 때문이다. 그런데, 이 알고리즘에서 사용하는 R-트리는 결국에는 비공간 검색 조건을 만족시키지 않을 투플 후보들을 부분적으로 제거할 수가 없기 때문에 비효율적이다. 본 논문에서 우리는 이 알고리즘을 보완한 RS-트리 기반 점증적 최근접 알고리즘을 제안한다. RS-트리는 R-트리와, 그 보조 트리로서 계층적 시그니쳐 화일을 기반으로 하는 S-트리로 구성된다. S-트리는 R-트리를 탐색하는 과정에서 많은 불필요한 투플을 제거하는 역할을 수행한다. 본 논문에서는 실험을 통해 RS-트리가 Hjaltason과 Samet의 알고리즘의 성능을 향상시킬 수 있음을 보인다.

Abstract In multimedia databases, k-nearest neighbor queries are very popular and include commonly non-spatial predicates. Among the available techniques for such queries, the incremental nearest neighbor algorithm proposed by Hjatason and Samet is known as the most useful algorithm[1]. The reason is that if k′ > k neighbors are needed, it can provide the next neighbor for the upper operator without restarting query from scratch. However in their algorithm, the R-tree has no facility capable of partly pruning tuple candidates that will turn out not to satisfy the remaining predicates, leading their algorithm to inefficiency. In this paper, we propose an RS-tree-based incremental nearest neighbor algorithm complementary to their algorithm. The RS-tree in our algorithm is a hybrid of the R-tree and the S-tree, as its buddy tree, based on the hierarchical signature file, which participates in pruning a large portion of worthless candidates while traversing the R-tree. Experimental results show that our RS-tree enhances Hjaltason and Samet's algorithm.

1. 서 론

최근 지리정보시스템(GIS) 또는 이미지 검색 시스템 등의 멀티미디어 응용들은 다차원 공간 객체 집합을 대상으로 효율적인 "k-최근접 질의(k-nearest neighbor query)"의 처리를 요구한다. 이러한 질의를 위한 효율적인 최적화 기법들이 개발되어 왔다[1,2,3,4,5]. 그 중에서 Hjaltason과 Samet이 제안한 점증적 최근접 기법(Incremental Nearest Neighbor Algorithm)은 대화식 방식(interactive fashion)으로 동작하므로 "거리 브라우징(distance browsing)"을 요구하는 아래와 같은 k-최근접 질의 처리에 가장 적합하다고 알려져 있다[1].

SELECT *

· 본 연구는 정보통신부의 대학기초연구 과제, "공간 데이타베이스의 확장 및 공간 데이타 웨어하우징으로의 응용에 관한 연구"의 부분적인 지원과 2000년도 두뇌한국21사업의 부분적인 지원에 의한 것임.

† 비 회 원 : 서울대학교 컴퓨터공학부
djpark@oopsla.snu.ac.kr
†† 종신회원 : 서울대학교 컴퓨터공학부 교수
hjk@oopsla.snu.ac.kr

논문접수 : 1999년 12월 30일
심사완료 : 2000년 9월 8일

```
FROM DISC
WHERE artist = 'Beatles'
ORDER BY distance(color, 'red')
STOP AFTER k;
```

위의 질의는 Fagin의 멀티미디어 질의, "artist = 'Beatles' ∧ color = 'red'"[6]를 확장 SQL문으로 표현한 것으로, "Beatles의 음반(DISC) 중에서 음반의 색(color)이 red에 가장 가까운 k 개를 검색하라"는 의미를 가진다. 여기서 비공간 검색 조건(artist='Beatles)을 만족하는 k 개의 질의 결과를 얻기까지 질의 객체 red[1]로부터 거리 순서대로 공간 객체(color의 속성값)를 하나씩 검색해 낼 수 있는 거리 브라우징이 필요하다.

Hjaltason과 Samet의 점증적 최근접 알고리즘은 R-트리와 같은 계층적 인덱스와 순위 큐(priority queue)를 이용하여, 주어진 질의 객체로부터 거리순으로 다음 최근접 공간 객체를 포함하는 투플의 식별자[2](TID)를 하나씩 상위 연산자(예를 들면, artist='Beatles'를 처리하기 위한 *select* 연산자)에게 전달한다. 상위 연산자는 TID를 이용하여 해당 투플을 접근한 후 그 밖의 비공간 검색 조건을 처리하며, k 개의 질의 결과가 될 때까지 이 과정이 반복한다. 이 알고리즘의 가장 큰 장점은, 상위 연산자가 또다른 TID를 요구할 때 질의를 처음부터 재시작(restart)하지 않고 이전의 작업 상태에서 다음 최근접 공간 객체를 포함하는 투플의 TID를 제공할 수 있다는 것이다[1].

그러나, 이 알고리즘은 k 개의 질의 결과를 얻기까지 많은 수의 불필요한, 즉 비공간 검색 조건을 만족시키지 않는 투플들을 상위 연산자에게 전달한다는 면에서 그리 효율적이지 못하다. 불필요한 투플이 많은 경우 그만큼 사용자의 응답 시간이 길어지게 된다. 불필요한 투플이 발생하는 이유는, 점증적 최근접 알고리즘이 질의에 사용된 비공간 검색 조건과는 무관하게 작동하기 때문이다.

본 논문에서, 우리는 비공간 검색 조건이 포함된 k-최근접 질의를 처리할 때 Hjaltason과 Samet의 알고리즘의 성능을 개선할 수 있는 새로운 알고리즘을 제안한다. 우리의 알고리즘은 불필요한 투플들을 부분적으로 제거할 수 있다는 측면에서 그들의 알고리즘에 보완적이다. 우리의 알고리즘에서는, Hjaltason과 Samet의 알고리즘에서 사용되는 R-트리가 아닌 RS-트리를 이용한다. RS-트리는 R-트리와, 그 보조 트리(buddy tree)로서 계층적 시그니쳐 화일(hierarchical signature file)[7]을 기반으로 하는 S-트리의 조합을 일컫는다. S-트리는 비공간 속성을 키(key)로 사용하며, R-트리와는 달리 불필요한 투플들을 부분적으로 제거할 수 있는 기능을 제공한다.

본 논문에서, 우리는 다음과 같은 두 가지 가정을 한다. (1) 사용자 질의에 artist='Beatles'와 같은 형태의 *Equal Selection*(즉, =) 조건문이 포함되어 있다고 가정한다. 이러한 가정은, S-트리가 "키워드(keyword) 중심의 질의"를 지원하는 텍스트 검색 시스템에 인덱스로 많이 사용되는 시그니쳐 화일에 기반을 두기 때문이다. (2) 사용자 질의는 오직 하나의 비공간 검색 조건을 갖는다. 물론, artist = 'Beatles' OR(또는 AND) artist = 'Sting' 또는 artist ='B*' 형태의 검색 조건을 갖는 질의도 고려할 수 있다. 하지만, S-트리는 시그니쳐 화일을 기반으로 하며, 이를 이용한 텍스트 검색에서 이러한 질의의 처리가 가능하기 때문에[7], 본 논문에서는 artist = 'Beatles'와 같이 단순한 검색 조건을 갖는 질의만을 고려한다. 이와 같은 비공간 검색 조건들은 Paradise[8], Dedale[9] 등의 GIS 응용이나 Chabot[10], QBIC[11] 등의 이미지 검색 시스템에 많이 나타난다.

이후 논문의 구성은 다음과 같다. 제 2절에서는 관련 연구를 서술하며, 제 3절에서는 Hjatason과 Samet의 알고리즘을 살펴보고 그 문제점을 알아본다. 제 4절에서는 주요 아이디어를 설명하며, 제 5절에서는 RS-트리와 이것을 이용한 점증적 최근접 알고리즘을 제시한다. 제 6절에서는 실험 결과를 보이며, 마지막으로 제 7절에서 결론을 맺는다.

2. 관련 연구

멀티미디어 데이타베이스 또는 GIS 분야에서 k-최근접 질의를 처리하기 위한 많은 기법들이 소개되었으며, 다차원 인덱스에 기반을 둔 방법(branch & bound 알고리즘[5], 점증적 최근접 알고리즘[1,3,4], 점증적 최근접 기법을 이용하는 다단(multi-step) 알고리즘[12]), 효율적인 순차 스캔(sequential scan)을 위한 벡터 근사(Vector approximation) 기법[13], 병렬 처리 기법[14] 등이 있다. 어떤 응용에서는 정확한 k 개의 질의 결과보다 근사된(approximate) k 개의 질의 결과로도 충분한 경우가 많다. 이를 위한 k-근사 최근접 알고리즘[15]은 많은 비용을 줄일 수 있다.

[5]의 *mindist*와 *minmaxdist*를 이용한 branch & bound 알고리즘 또는 [13]의 벡터 근사 기법은 미리

1) 실제적으로는 d-차원 공간에서 red의 특성 벡터(feature vector)를 의미하지만 본 논문에서는 같은 의미로 사용한다.

2) R-트리와 같은 계층적 인덱스의 데이타 노드(leaf node)에 (공간 객체, TID) 쌍의 레코드가 저장된다고 가정한다.

주어진 k 값을 사용하기 때문에 "거리 브라우징"이 요구되는 질의를 처리하는 데 비효율적이다. 그 이유는 앞 절의 서론에서 설명한 재시작 비용을 부담해야 하기 때문이다. 반면, k-d 트리 또는 LSD-트리를 이용하거나 R-트리와 같은 계층적 인덱스에 일반적으로 적용될 수 있는 점증적 최근접 알고리즘[1,3,4]은 재시작 비용이 필요가 없기 때문에 이와 같은 질의를 처리하는데 적합하다. 이 중에서 Hjaltason과 Samet의 알고리즘은 단순하면서도 효율적이며 범용으로 사용될 수 있는 장점이 있다[1].

[6]에서, Fagin은 "artist = 'Beatles' ∧ color = 'red'"와 같은 질의를 처리할 수 있는 간단한 방법을 제시하였다. 질의에 포함된 각각의 조건문을 처리하기 위한 서브 시스템(예로, 관계형 데이타베이스 시스템과 QBIC과 같은 이미지 검색 시스템)이 필요하며, 비공간 검색 조건의 선택율이 매우 낮다는 가정에 기반한다. 그리고, Fagin의 방법은 R-트리와 같은 공간 인덱스를 전혀 사용하지 않는 순차 스캔 방법(sequential scan method)에 해당한다. 반면, 우리의 기법에서는 비공간 검색 조건의 선택율이 질의 최적기에서 공간 인덱스를 사용할 만큼 높으며[3], 또한 질의 결과의 수(k)에 제한을 두고 있다.

3. 점증적 최근접 알고리즘

Hjaltason과 Samet이 제안한 점증적 최근접 알고리즘에 대해서 자세히 살펴보며, 비공간 검색 조건이 포함된 k-최근접 질의를 처리할 때 발생하는 문제점을 제시한다.

3.1 Hjaltason & Samet의 알고리즘

[1]에서 Hjaltason과 Samet은 R-트리를 이용한 점증적 최근접 알고리즘을 제안하였다(이후, *RtreeINN* 알고리즘). R-트리의 루트 노드부터 탐색하면서, 다음 탐색 노드는 큐의 최전방 레코드에서 얻으며 그 노드의 자식 노드 또는 공간 객체를 큐에 삽입한다. 이때 질의 객체로부터 자식 노드 또는 공간 객체와의 거리를 함께 삽입한다. 항상 큐는 거리 순서대로 유지된다. 만약 큐의 최전방에서 꺼낸 데이타가 공간 객체일 때, 이것을 포함하는 튜플의 식별자(TID)를 비공간 검색 조건 처리를 위한 select 연산자에게 리턴한다. k 개의 질의 결과를 얻거나 또는 큐에 레코드가 없을 때까지 이 과정을 반복한다. 위에서 설명하였듯이, *RtreeINN* 알고리즘은 $k'(> k)$ 투플이 필요할 때 질의를 처음부터 재실행하지 않고 다음 TID를 상위 연산자에게 전달할 수 있다.

3.2 문제점

Berchtold 등은 R-트리와 같은 계층적 인덱스를 이용하는 k-최근접 알고리즘의 "최적성(optimality)"을 정의하고, *RtreeINN* 알고리즘이 최적성을 보장함을 증명하였다[2]. 최적성의 정의는 다음과 같다.

정의 1 *k-최적성(k-Optimality)*

최근접 질의를 처리할 때 접근된 전체 페이지 수와 $SP(Q, r)$와 겹치는(intersect) 페이지 수가 같을 때, 그 k-최근접 알고리즘은 최적성을 갖는다.

여기서 Q와 r은 각각 질의 객체와, Q로부터 k-번째 거리에 위치하는 공간 객체(O^k)와의 거리를 뜻한다 ($r=||Q - O^k||$). 그리고, $SP(Q, r)$는 중심이 Q이고 반지름이 r인 구면체(sphere)를 나타낸다.

정의 1에 의해 *RtreeNN* 알고리즘은 최적성을 갖는다(자세한 것은 [2]를 참조). 따라서, k 개의 질의 결과를 얻기까지 *RtreeNN* 알고리즘의 비용은 $SP(Q, r)$과 겹치는 R-트리의 페이지 수와 같다. 여기서의 비용은 비공간 검색 조건이 전혀 포함되지 않은 k-최근접 질의의 비용을 의미한다.

지금부터 비공간 검색 조건이 포함된 k-최근접 질의를 처리할 때 *RtreeNN* 알고리즘의 문제점을 살펴본다. *RtreeNN* 알고리즘을 이용하여 이러한 질의를 처리할 때 전체 비용 $C_{all} = C_{rtree} + C_{tuple}$과 같다. 여기서 C_{rtree}는 R-트리 비용, C_{tuple}은 비공간 검색 조건을 처리하기 위한 투플 접근 비용을 의미한다. 그 외 전체 비용에는 순위 큐 연산 비용와 CPU 비용 등이 포함되지만 다른 비용에 비해 작기 때문에 무시한다.

질의 객체를 Q, 비공간 검색 조건을 만족하는 k-번째 공간 객체를 $O^{k'}(k' \geq k)$, 반지름 $r = ||Q - O^{k'}||$이라고 할 때, C_{rtree}는 *RtreeNN* 알고리즘의 최적성에 따라 $SP(Q, r)$와 겹치는 R-트리 노드의 수와 같다. 다음으로, C_{tuple}을 구하기 위해 k'을 유추한다. 상위 연산자에게 전달되는 TID 집합의 크기는 k'과 동일하며, 질의 결과의 수 k가 주어질 때 k'을 다음과 같이 유추할 수 있다. 공간 속성(color)과 비공간 속성(artist)간에 연관성(correlation)이 전혀 없고, 질의에 사용된 속성값('Beatles')의 선택율(selectivity)을 S라고 하자. 그러면, i번째 질의 결과를 만족하는 공간 객체를 O^i라고 할 때, i+1 번째 질의 결과를 얻기 위해서는 $d_i \leq ||Q - O^i|| \leq d_{i-1}(d_i = ||Q - O^i||)$를 만족하는 $1/S$ 개의 공간 객체 O를 더 접근해야 한다. 따라서, $k' = \frac{1}{S} \cdot k$가 된다. 여기서 k'은 k에 대해 선형적으로 증가함을 알 수 있다. 이때, 평

3) Hjaltason과 Samet의 알고리즘은 공간 인덱스에 기반하고 있음을 염두해야 한다.

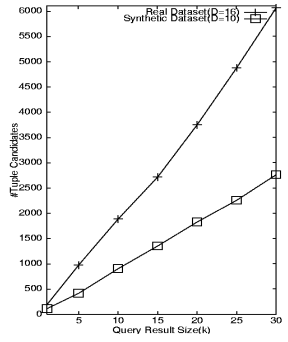


그림 1 후보 투플 수에 대한 관찰(N=100,000)

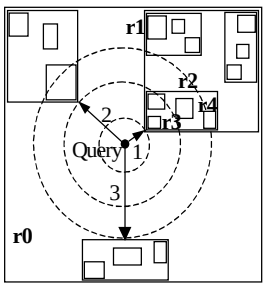


그림 2 RtreeINN 알고리즘의 탐색공간

균 접근 페이지 수를 구하는 Yao의 방정식[16]을 사용하면 $c_{tuple}=k(\frac{N\cdot T}{B},\frac{B}{T},\frac{k}{S})$과 같다. 여기서 N은 투플의 수, B는 페이지 크기, T는 투플 크기를 의미한다.

우리는 위에서 구한 k'의 근사값을 확인하기 위해 [1]의 *RtreeINN* 알고리즘을 구현하여 "artist = 'Beatles' ∧ color = 'red'" 형태의 질의를 여러 번 수행하여 그림 1의 결과를 얻었다(자세한 것은 6절의 실험 환경을 참조). 그림 1에서 우리가 위에서 예측한 대로 그래프는 선형적으로 증가함을 알 수 있다. 그러나, 비공간 검색 조건을 만족하는 작은 수(k)의 질의 결과에 비해 접근해야 하는 투플의 수(k')가 너무 많다. 이 사실은 k' 개의 후보 투플 집합에는 너무 많은 불필요한 투플들을 포함하고 있다는 것을 나타낸다. 결국, k'가 커지면 C_{tuple} 비용이 커지게 되고 사용자 응답 시간이 길어진다.

4. 주요 아이디어(Key Ideas)

앞의 3.2 절에서 제기되었듯이, *RtreeINN* 알고리즘에서 불필요한 후보 투플이 많이 발생하는 이유는, R-비트리가 공간 검색 조건을 만족시키지 않을 투플을 미리 제거할 장치가 없기 때문이다. 우리의 전략을 설명하기 전에 먼저 간단한 예를 하나 든다. 그림 2에서, 질의 객체 *Query*로부터 각 R-트리 노드 r_i까지의 거리를 d_i, 거리 순서 $d_i<d_j(i<j)$를 만족한다고 가정한다.

RtreeINN 알고리즘에 따르면, r_0, r_1 다음으로 r_2를 처리할 때, r_2 노드의 각 자식 노드를 미리 계산한 거리와 함께 큐에 삽입할 것이다. 여기서, r_3과 r_4의 하위 트리에 비공간 검색 조건을 만족시키는 투플이 하나도 없다고 가정한다. 가정에 따라 r_3과r_4가 쓸모없는 노드임에도 불구하고, *RtreeINN* 알고리즘은 그 노드들로 인해 미래에 많은 불필요한 투플들이 생성된다는 것을 고려하지 않고 그 노드들을 큐에 삽입할 것이다.

반면, 우리의 전략은 *RtreeINN* 알고리즘의 매 삽입 단계전에 각 자식 노드의 서브트리(subtree)가 비공간 검색 조건을 만족하는지를 검사하는 연산을 추가하는 것이다. 예를 들면, 우리의 전략을 따르는 알고리즘은 r_3 또는 r_4와 같은 노드를 큐에 삽입할 가능성을 줄일 것이며, 그로부터 파생되는 모든 불필요한 투플들이 생겨나는 것을 방지할 수 있다.

어떤 R-트리 노드의 서브트리가 바람직한 투플을 포함하고 있는지를 검사하기 위해서, R-트리와 같은 계층적 구조(hierarchical structure)를 지원하는 자료 구조가 필요하다. 또한, 그러한 자료 구조의 각 엔트리(entry)는, R-트리의 MBR(Minimun Bounding Rectangle)이 공간 객체를 계층적으로 경계를 지우듯(bound), 그 엔트리의 서브 트리에 속한 모든 비공간 속성값들을 표현할 수 있는 집합값(set value)를 가져야 한다. 우리는 위와 같은 속성들을 갖는 자료 구조로서 계층적 시그니처 화일에 기반한 S-트리와, 더나아가 R-트리와 S-트리로 구성된 RS-트리를 제안한다.

우리가 제안하는 알고리즘은 R-트리가 아닌 RS-트리에 기반한 점증적 최근접 알고리즘이며, 우리의 알고리즘의 장점은 다음과 같은 S-트리의 속성에 기인한다.

- S-트리는 "시그니처 집합"으로서 각 시그니처는 비공간 속성값들의 집합을 표현하는 집합 값(set value)이다.
- S-트리는 계층적 시그니처 화일에 기반하므로 R-트리와 똑같은 계층적 구조를 지원한다.
- S-트리는 R-트리와 독립적인 자료 구조를 가지며, 또한 독립적인 저장 구조를 가진다.
- S-트리도 R-트리의 MBR과 같이 레벨에 따른 포함 관계(hierarchical inclusion relationship)를 잘 지원한다.
- S-트리의 시그니처는 0 또는 1로 이루어진 비트 스트림(bit stream)이므로 AND 또는OR과 같은 연산 비용이 싸다.

5. RS-트리를 이용한 점증적 최근접 알고리즘

본 절에서는 RS-트리의 구조, S-트리의 제거 효과(pruning effect)와 이를 높이기 위한 시그니쳐 분할(signature chopping) 기법, RS-트리를 이용한 점증적 최근접 알고리즘에 대해서 서술한다.

5.1 RS-트리

RS-트리는 R-트리와 계층적 시그니쳐 화일 구조를 갖는 S-트리의 조합을 의미한다. R-트리는 color와 같은 공간 속성에 대한 인덱스로 사용되며, S-트리는 artist와 같은 비공간 속성에 대한 인덱스로 사용된다. 일반적으로 하나의 투플은 여러 개의 비공간 속성으로 구성될 수 있으므로, RS-트리를 하나의 공간 속성과 m 개의 비공간 속성을 위한 인덱스, 즉 RS^m-트리로 확장할 수 있다. 새로운 인덱스는 m=1인 RS-트리를 중심으로 설명한다.

RS-트리를 구성하는 R-트리는 기존의 구조와 동일하므로 S-트리를 중심으로 전체 인덱스를 설명한다. S-트리는 R-트리와 같이 계층적 구조의 노드의 집합으로 구성되며, 각 노드는 여러 개의 엔트리들로 채워진다. S-트리의 각 노드와 그 노드를 구성하는 엔트리는 각각 R-트리의 노드와 엔트리에 일대일 대응된다. 그러나, S-트리는 R-트리와 달리 인덱스 노드 레벨만을 가진다. 그 이유는, R-트리는 인덱스 노드에 비해 데이타 노드의 저장 공간 비율이 매우 높으므로 R-트리 데이타 노드와 일대일 대응되는 S-트리의 데이타 노드를 두면 저장 공간의 오버헤드(overhead)가 너무 커지기 때문이다.

S-트리 노드의 j 번째 엔트리는, 해당 R-트리 노드의 j 번째 엔트리[4]의 서브트리에 포함되는 모든 비공간 속성값(예를 들면, 'Beatles')의 집합을 표현하는 집합 값을 가진다. S-트리 노드의 엔트리가 집합 형태의 값을 갖는 이유는 "비공간 검색 조건을 만족시키지 않는 R-트리 노드"를 쉽게 제거하기 위해서다.

S-트리는 R-트리가 일괄적재(bulk-loading) 방식으로 생성되는 시점에서 동시에 생성된다고 가정한다. 이 때 S-트리 노드의 엔트리가 가지는 시그니쳐의 생성은 크게 S-트리의 레벨 l = 1(S-트리의 맨 하단층)과 l >1로 구분된다. 레벨 l > 1에서 S-트리 노드의 엔트리가 갖는 시그니쳐는, 1) 엔트리 자신과 대응되는 R-트리 노드의 엔트리의 서브트리가 가질 수 있는 모든 비공간 속성값의 집합을 구한 다음, 2) 그 집합에 속하는 각 비공간 속성값을 해쉬(hash) 함수를 써서 고정길이의 비트 스트림 즉, 시그니쳐로 변환하고, 3) 생성된 시그니쳐 집합의 모든 원소에 대해 비트 단위로 합연산(OR)한 결과와 같다. 그 이외의 레벨에서는 S-트리 노드의 엔트리가 가리키는 자식 노드의 모든 시그니쳐를 합연산한 결과를 그 엔트리의 시그니쳐로 사용한다.

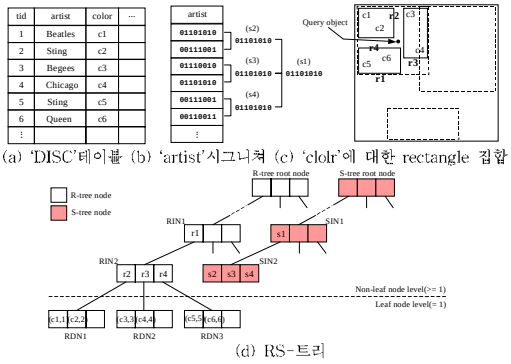


그림 3 RS-트리의 생성

그림 3에서 테이블 DISC는 공간 속성으로 color를, 비공간 속성으로 artist를 가지며, 각각 R-트리와 S-트리의 키로 사용된다. 그림 3(b)는 DISC 테이블의 각 artist 속성값에 대해 해쉬 함수를 이용하여 변환된 고정 길이의 시그니쳐 테이블을 의미한다. color 속성에 대한 d-차원 공간이 그림 3(c)와 같을 때, 해당 R-트리는 그림 3(d)의 왼쪽 부분의 트리와 같다. 각 R-트리 인덱스 노드에 일대일 대응되는 노드(예: $RIN_2 \leftrightarrow SIN_2$)의 집합으로 구성된 트리가 S-트리이다. 레벨 l=1에서 SIN_2 노드의 s_2 시그니쳐를 생성하는 과정은, 먼저 대응되는 R-트리 노드RIN_2의 엔트리 r_2를 찾고, r_2가 가리키는 데이타 노드 RIN_1에 포함된 TID=1, 2를 통해 테이블 DISC에서 artist의 속성값 {'Beatles', 'Sting'}을 얻고, 해쉬 함수를 통해 시그니쳐 {01101010, 00111001}로 변환한 후, 두 개의 시그니쳐를 합연산한다(01101010 ∧ 00111001 = 01111011). 시그니쳐 s_3와 s_4도 s_2와 같은 방식으로 만들어진다. 레벨 l>1인 경우, 시그니쳐 s_1은 자신의 하위 노드 SIN_2가 가지는 모든 시그니쳐 s_2, s_3, s_4의 합연산 01111011 ∧ 01111010 ∧ 00111011 = 01111011과 같다. 이런 식으로 최상위 S-트리 노드의 시그니쳐가 생성되고, 전체 S-트리가 만들어진다.

5.1.1 단점

RS-트리는 추가된 S-트리에 의해 저장 공간의 오버

4) R-트리 인덱스 노드의 엔트리는 (pointer, mbr) 값을 가지며, pointer는 엔트리 자신의 다음 하위 노드를 가리키고 mbr은 그 하위 노드를 둘러싸는 최소 사각형을 의미한다.

헤드와 갱신 비용(update cost) 등의 단점을 가진다. 저장 공간을 줄이기 위해서 R-트리와는 달리 S-트리는 인덱스 페이지(index page)만을 갖도록 설계되었다. S-트리에 의한 저장 공간의 증가량은 모든 R-트리 인덱스 페이지의 수와 같다. 그런데, R-트리 전체 페이지 수에 대한 인덱스 페이지의 수의 비율은 대략 $\frac{c^{l-1}}{c^l}=\frac{1}{c}$($c$는 R-트리 노드의 용량이며 l은 R-트리의 높이)에 해당되므로, 요구되는 저장 공간은 아주 적다고 할 수 있다.

공간 데이타가 삽입 또는 삭제되거나 비공간 데이타의 값이 수정되는 경우 S-트리의 갱신 작업이 필요하다. 첫째, 공간 데이타가 삽입 또는 삭제되는 경우 R-트리 갱신 알고리즘에 의해 해당 말단 페이지(leaf page)에 그 데이타가 삽입 또는 삭제된다. 삽입의 경우, 해쉬 함수를 이용하여 추가된 비공간 데이타를 시그니처로 만든 후, S-트리의 상위 레벨로 이동하면서 새로 생성된 시그니처를 수정이 요구되는 시그니처와 합연산한다. 삭제의 경우, 수정된 R-트리 말단 페이지에 저장되어 있는 각 TID를 이용하여 투플을 접근한 후, 각 비공간 데이타에 대해 시그니처를 생성하고 모든 시그니처를 합연산한다. 그리고 S-트리의 상위 레벨로 이동하면서 새로 생성된 시그니처를 수정이 요구되는 시그니처에 반영시킨다. 공간 데이타의 삽입 또는 삭제 연산 시, R-트리 페이지 분할(split) 또는 합병(merge)이 발생할 수 있다. 이런 경우, 동시에 S-트리에도 분할 또는 합병 작업을 수행한다. 그리고 수정된 R-트리 페이지에 대해서 새로운 시그니처를 생성하고 S-트리에 반영시킨다. 둘째, 비공간 데이타의 값이 수정되는 경우, 앞에서 설명한 공간 데이타의 삭제와 비슷한 작업을 거친다. 해당 투플의 공간 데이타가 저장되어 있는 R-트리 말단 페이지를 찾은 후, 그 페이지의 모든 TID를 이용하여 새로운 시그니처를 생성하고 이것을 S-트리에 반영시킨다.

위와 같은 RS-트리의 단점은 S-트리에 의한 성능 향상을 통해 충분히 보상되어질 수 있다(6 절의 실험 결과). 멀티미디어 데이타베이스에서 중요한 것은 공간 데이타라 할 수 있으며, 공간 데이타는 자주 갱신되지 않는 특징이 있다. 이러한 특징도 RS-트리의 단점을 희석시키는 데 일조한다.

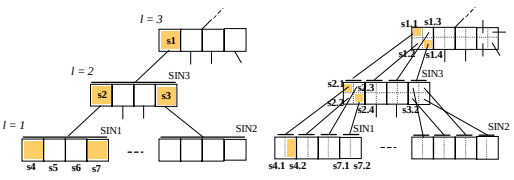
5.2 S-트리의 제거 효과(Pruning Effect of the S-tree)

본 절에서는 S-트리가 첨가된 *RtreeINN* 알고리즘의 관점에서 S-트리에 의해 불필요한 R-트리 노드와 객체(또는 TID)가 어떻게 제거되는지 설명한다.

큐의 최전방에 존재하는 다음 탐색 노드(next target node)를 R_i, 그 노드의 자식 노드를 $r_j(0\le j<c)$이라 하면, 새로운 알고리즘은 각각의 r_j로부터 질의 객체까지의 거리를 계산하기 전에 R_i와 대응되는 S-트리 노드 S_i의 시그니처 $s_j(0\le j<c)$를 이용하여 r_j의 서브트리가 비공간 질의값(예를 들면, 'Beatles')을 포함하는지를 먼저 검사한다. 즉, 비공간 질의값의 시그니처(예로 'Beatles'의 시그니처 01101010)를 질의 시그니처(query signature) s_q라고 하면, $s_q \wedge s_j = s_q$를 만족하는지 검사한다. 여기서 연산자 $\wedge$는 비트 단위의 곱연산(AND)을 의미하며, 이러한 검사를 시그니처 검사(Signature Checking)라고 부른다. 시그니처 검사를 통과한다는 것은, 해당 R-트리의 자식 노드가 가질 수 있는 모든 데이타 노드에 질의에 사용된 비공간 속성값을 갖는 투플이 있을 수 있다는 것을 의미하며, 그렇지 않는 경우는 그 반대의 의미를 지닌다. 따라서, S-트리의 노드 S_i에 포함되는 모든 시그니처 중 시그니처를 통과하는 s_j에 대응되는 R-트리의 자식 노드를 큐에 삽입하면 된다. 물론, 시그니처 검사를 위해 S-트리 노드 I/O가 발생하지만 이 비용보다 S-트리의 노드 제거에 의한 이익이 실제로 더 크다(제 6절의 실험 결과).

그림 3에서, 현재 큐에서 꺼낸 탐색 노드가 RIN_2, 엔트리 r_2, r_3, r_4이 각각 가리키는 자식 노드가 RIN_1, RIN_2, RIN_3, 질의 객체로부터 각 자식 노드까지의 거리가 각각 d_1, d_2, d_3, 거리 관계가 $d_2 < d_1 < d_3$을 만족시킨다고 하자. 그리고 질의 시그니처를 'Beatles'의 시그니처, s_q = 01101010이라고 가정한다. S-트리를 이용하는 알고리즘에서는 RIN_2의 엔트리 r_3가 가리키는 자식 노드 RIN_2를 큐에 삽입하기 전에 먼저 S-트리를 이용한 시그니처 검사를 수행한다. 시그니처 검사에서 s_3(= 01111010)은 $s_q \wedge s_3 = s_q$를 만족시키지 않으므로 RIN_2는 큐에 삽입되지 않는다.

상위 레벨로 갈수록 S-트리의 시그니처는 모든 비트가 '1'로 채워질 "시그니처 포화(signature saturation)" 상태에 이를 수 있다. 그 이유는 5.1절의 S-트리의



(a) 시그니처 분할 전 (b) 시그니처 분할 후

그림 4 Signature Chopping

생성에서 레벨이 증가하면 시그니쳐를 생성하기 위해 합연산되는 자식 시그니쳐의 수가 증가하기 때문이다. 이때 발생하는 문제가 "false drop"이 많이 발생한다는 것이다. 어떤 투플이 false drop이라는 것은 시그니쳐 검사를 통과했음에도 불구하고 결국에는 질의 조건을 만족시키지 않는다는 것을 의미한다. 예를 들면, 그림 3(d)에서 s_q=01101010일 때 RIN_2의 TID = 3, 4인 투플은 s_3이 시그니쳐 검사를 통과하지만, 결국 artist ='Beatles'를 만족하지 않으므로 false drop이 된다. false drop의 문제는 다수의 시그니쳐가 합연산될 때 발생하는 "팬텀 효과(phantom effect)" 때문이다. 다음 절에서는 시그니쳐 포화로 인해 발생하는 다량의 false drop의 수를 줄일 수 있는 "시그니쳐 분할 방식"에 대해서 설명한다.

5.3 시그니쳐 분할(Signature Chopping)

우리는 수식적으로 "시그니쳐 포화"를 확인할 수 있다. 자식 시그니쳐와 부모 시그니쳐 각각의 i번째 비트 값이 '1'일 확률이 각각 α_L, α_P이고, 그룹에 속하는 자식 시그니쳐 수가 N_L일 때, $\alpha_P = 1-(1-\alpha_L)^{N_L}$과 같이 주어진다[7]. 여기서, 비교적 작은 α_L에 대한 α_P는 작은 N_L에도 불구하고 거의 '1'에 가까와진다는 것을 추정할 수 있다. 이로부터 S-트리의 레벨이 증가할수록 시그니쳐 포화에 급격하게 도달한다는 사실을 알 수 있다. 왜냐 하면, S-트리의 레벨이 증가하면 동시에 N_L도 증가하기 때문이다. 따라서, 팬텀 효과로 인해 더 많은 false drop이 발생할 것이다.

우리는 α_L 또는 N_L를 작게 만듬으로써 시그니쳐 포화 속도를 지연시킬 수 있다. 작은 α_L를 선택하면 확실히 그 속도를 지연시킬 수 있지만, 작은 α_L를 위해 매우 큰 크기의 시그니쳐가 필요함에도 불구하고 그 효과는 그리 크지 않다. 대신, 우리는 N_L을 작게 만들 수 있는 방법을 제안하며, 이것을 "시그니쳐 분할(signature chopping)"이라고 부른다.

시그니쳐 분할의 목표는, S-트리 노드의 각 시그니쳐를 시그니쳐 분할 함수 $f(l)$(예를 들면, $f(l)=2^l$)에 의해 여러 개의 시그니쳐로 분할함으로써, N_L을 가능하면 작게 만들고 더나아가 팬텀 효과를 줄이는 데 있다[5]. 레벨 l 상에 존재하는 어떤 S-트리 노드가 갖는 각각의 시그니쳐 $s_i(0\le i<c)$에 대해서, s_i는 $f(l)$ 개의 분할 시그니쳐(chopped-signature), $s_{i,j}(0\le j<f(l))$로 나누어진다 (레벨 l이 높은 경우 $f(l)\ge c$가 발생할 수 있으므로 이런 경우의 $f(l)=c$가 된다). S-트리의 생성시(제 5.1절), s_i의 자식 노드가 갖는 c 개의 시그니쳐들은 $f(l)$ 개의 버킷(bucket)에 $\lceil \frac{c}{f(l)} \rceil$씩 배분된다. 그런 후, j-번째 버킷에 포함된 모든 시그니쳐를 합연산함으로써 $s_{i,j}$가 생성된다. 여기서, $\vee$를 OR 연산자라고 할 때 $s_i = \bigvee_{j=1}^{f(l)} s_{i,j}$을 만족시킨다. 따라서, 팬텀 효과는 시그니쳐 분할 전에 비해 약 $\lceil \frac{c}{f(l)} \rceil$배만큼 감소한다. 그림 4에서 c=4이고 $f(l)=2^l$이라고 가정할 때, s_2는 $f(2)=2^2$개의 $s_{2,1}$, $s_{2,2}$, $s_{2,3}$, $s_{2,4}$로 분할된다(s_1의 경우는 $f(3)(=2^3)\ge c$이므로 $f(l)=4$ 개로 분할된다). 시그니쳐 분할 전의 s_2는 자신의 자식 노드가 갖는 네 개의 s_4, s_5, s_6, s_7을 합연산하여 생성되는 반면, 시그니쳐 분할 후의 s_2는 네 개의 시그니쳐로 분할되며 각각은 s_2의 자식 노드 중에서 단 하나의 시그니쳐를 자식 시그니쳐로 갖는다.

시그니쳐 분할 방식에서, 시그니쳐 검사는 $f(l)$ 개의 분할 시그니쳐 각각에 대해 수행된다. R을 큐에서 꺼낸 다음 탐색 노드 S를 R와 대응되는 S-노드라고 정의하자. $f(l)$ 개의 분할 시그니쳐 $s_{i,j}$에 대해서, 어떤 분할 시그니쳐도 시그니쳐 검사를 통과하지 못하면, 시그니쳐 분할을 적용하지 않는 방식과 같이 R의 i-번째 엔트리의 자식 노드(=$CHILD_i$)는 제거된다. 그 이유는 다음과 같다. 시그니쳐 분할 전 $CHILD_i$를 큐에 넣을 수 있기 위해서는 $s_i \wedge s_q = s_q$가 만족되어야 한다. 그런데 시그니쳐 분할 후 s_i의 어떤 분할 시그니쳐도 시그니쳐 검사를 통과하지 못하면 즉, $s_{i,j} \wedge s_q \neq s_q$, $0\le j<f(l)$이면, $s_i \wedge s_q = (s_{i,1} \vee s_{i,2} \vee \ldots \vee s_{i,f(l)}) \wedge s_q = (s_{i,1} \wedge s_q) \vee \ldots \vee (s_{i,f(l)} \wedge s_q) \neq s_q$이기 때문에 $CHILD_i$는 제거된다. s_i의 분할 시그니쳐 중 하나라도 시그니쳐 검사를 통과하는 경우, $CHILD_i$는 추가적인 힌트(hint), 예를 들면, 비트맵(bitmap)과 함께 큐에 삽입된다. 여기서 힌트는, 시그니쳐 검사를 통과하지 못한 각 분할 시그니쳐에게 배분된, $CHILD_i$ 노드의 $\lceil \frac{c}{f(l)} \rceil$개의 엔트리의 모든 서브트리들은 더이상 접근될 필요가 없으므로 제거된다는 의미를 갖는다. 여기서, s_i의 각 분할 시그니쳐는 $CHILD_i$ 노드의 $\lceil \frac{c}{f(l)} \rceil$개의 엔트리를 책임진다. 그림 4에서 시그니쳐 검사를 통과한 시그니쳐를 진한 영역으로 표시하며, S-트리 노드 SIN_i에 대응되는 R-트리 노드를 RIN_i라고 하자. s_3의 분할 시그니쳐 $s_{3,1}$, $s_{3,2}$, $s_{3,3}$, $s_{3,4}$가 어느 것도 시그니쳐 검사를 통과하지 않는다면, R-트리 노드 RIN_2는 큐에 삽입되지 않는다. 반면, s_2의 분할 시그니쳐 중에서 $s_{2,1}$과 $s_{2,4}$는 시그니쳐 검사를 통과하므로 해당 R-트리 노드 RIN_1을 큐에 삽입한다. 이때, $s_{2,2}$와 $s_{2,3}$은 시그니쳐 검사를 통과하지 못하였기 때문에 RIN_1의 엔트리 중

5) $f(l)$이 커지면, S-트리에 오버플로우 페이지가 발생할 수도 있으므로 적절한 $f(l)$을 고려해야 한다.

```
Algorithm 1 RStreeINN(Qs, Qr)
/* Qs and Qr denote a given spatial query object and a non-spatial query value, respectively */
1  Sq := Hash(Qr)                                   /* a query signature */
2  Queue := PriorityQueue()
3  NewElm.ptr := RtreeRootNode; NewElm.dist := 0; NewElm.bitmap.Setall()
4  Enqueue(Queue, Elm)
5  while Queue is not empty
6     Elm := Dequeue(Queue)          /* the next target node is fetched from the queue */
7     if Elm.ptr is a non-leaf node          /* the next target node is a non-leaf node */
8          for each entry(child, mbr) in Elm.ptr
               /* check if current entry was pruned by the previous signature checking or not */
9              if Elm.bitmap.Isset(EntryIndex) is True
                    /* the corresponding S-tree page is read, ChoppedSignArray is filled with
                  the f(l) chopped signatures in that page, and signature checking is performed */
10                  NewBitmap := SignChecking(Sq, ChoppedSignArray)
                    /* check if at least one chopped signature passed signature checking or not */
11                  if at least one bit in NewBitmap is not zero
12                       NewElm.ptr:=child; NewElm.dist:=DIST(Qs, mbr); NewElm.bitmap:=NewBitmap
13                       Enqueue(Queue, NewElm)
14                  end if
15             end if
16         end for
17    else if Elm.ptr is a leaf node    /* the next target node is a leaf node */
18         for each entry(object, mbr, tid) in Elm.ptr
19             if Elm.bitmap.Isset(EntryIndex) is True
20                  NewElm.ptr := object; NewElm.dist := DIST(Qs, object); NewElm.tid := tid
21                  Enqueue(Queue, NewElm)
22             end if
23         end for
24    else                                          /* Elm.ptr is an object */
25         return Elm.tid
26    end if
27 end while

Algorithm 2 SignChecking(Sq, ChoppedSignArray)}
1  S := ChoppedSignArray
2  for i := 0 to f(l)-1                          /* l is the current level of the S-tree */
3     if (Sq ∧ Si) == Sq
4          for j := i*AllocSize(l) to (i+1)*AllocSize(l)-1
5               NewBitmap.Set(j)
6          end for
7     end if
8  end for
9  return NewBitmap
```

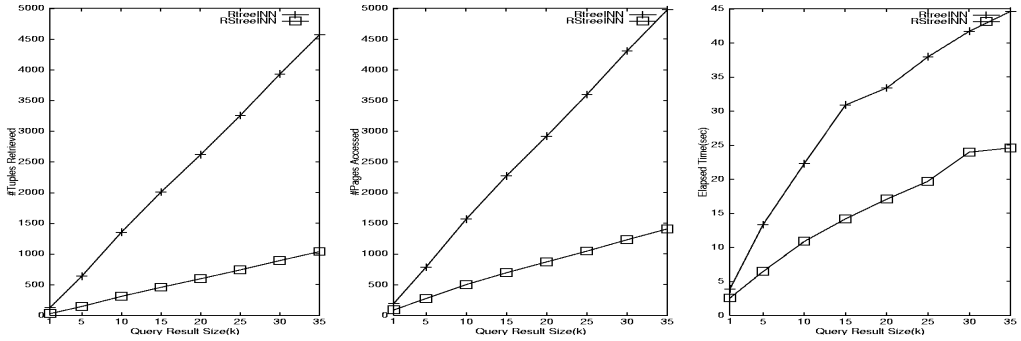
에서 두 번째와 세 번째 엔트리에 대해서는 어떤 작업(예: 거리의 계산)도 수행할 필요가 없다는 것을 표시하는 비트맵을 큐에 삽입한다.

5.4 RS-트리 기반 점증적 최근접 알고리즘

본 절에서는, 제 3절에서 설명한 *RtreeINN* 알고리즘을 보완한 RS-트리 기반 점증적 최근접 알고리즘(줄여서 *RStreeINN* 알고리즘)을 서술한다. 알고리즘 1과 2는 각각 *RStreeINN* 알고리즘과 시그니쳐 검사 루틴을 나타낸다. *RStreeINN* 알고리즘에서 추가적인 부분은 큐에서 꺼낸 다음 탐색 노드가 갖는 자식 노드 중 어떤 노드가 제거되었는지 확인하는 단계(Algorithm 1의 단계 9과 19)와, 큐에 그것들을 삽입하기 전에 시그니쳐 검사를 수행하는 단계(Algorithm 1의 단계 10)이다. 알고리즘 2에서, *AllocSize(l)*은 각각의 분할 시그니쳐에 배분된, 레벨 l 상의 R-트리 노드의 엔트리의 수를 의미한다. *RStreeINN* 알고리즘은 비교 연산자(> 또는 <)가 포함된 비공간 검색 조건문이 포함된 질의에 적합하지 않다. 그러나, 이러한 질의의 처리는 RS-트리의 R-트리를 이용한 *RtreeINN* 알고리즘을 그대로 적용함으로써 가능하다. 왜냐 하면, RS-트리에서의 R-트리는 S-트리와 독립적인 저장 구조를 갖기 때문이다.

6. 실험 결과

본 절에서는 다양한 실험을 통해 *RtreeINN* 알고리즘과 *RStreeINN* 알고리즘의 성능을 비교 평가한다. 실험을 위해 기본 테이블 DISC를 생성하였으며, 그 스키마는 DISC(artist, type, country, color)와 같다. color는 공간 속성으로서 d-차원의 점(point)으로 표현되며, 그 이외는 비공간 속성으로서 30 바이트 크기의 텍스트 형식의 값을 가진다. color 속성값은 균일(uniform) 분포를 따르는 인위 데이타와 실세계 데이타를 나누어 실험하였다. color 속성 이외의 속성들은 Zipfian 분포를 따르는 속성값을 가지며, 또한 각각의 속성은 전체 투플 수의 0.5%에 해당하는 이산(distinct) 값을 가진다. DISC 테이블을 저장할 때의 페이지의 크기는 4K 바이트이다.



(a) 투플 접근수 (b) 페이지 접근수 (c) 질의 수행 시간

그림 5 질의 결과 k에 따른 실험 결과(d=6, N=10^5, z=0.5, $f(l)$=6^l, F=64)

color 속성값을 저장하는 인덱스 구조로서 R*-트리[17]를 사용하였으며, R*-트리의 노드 크기는 4K 바이트이다. *RStreeINN* 알고리즘의 실험을 위한 S-트리는 R*-트리를 일괄적재 방식으로 생성할 때 동시에 생성하였으며, 이때 여러 시그니쳐 분할 함수 $f(l)$을 적용하였다(예를 들면, $f(l)$=2^l, 3^l, ...). 비공간 속성값에 대한 시그니쳐 생성을 위한 해쉬 함수는 [18]을 따르며, 각 실험에서 시그니쳐 크기 F=64비트(=8바이트)를 사용하였다. 두 알고리즘에서 사용하는 순위 큐는 모두 주 메모리에서 동작하도록 구현되었다.

실험 측정값은 질의 수행시간, 페이지 접근 수, 투플 접근 수로 나뉘며, 모든 실험에서 실험 질의는 (d-차원의 점, 비공간 속성값)으로 구성되며, 각 실험 결과는 50개의 실험 질의를 수행한 결과의 평균값이다. d-차원 점은 무작위 방식으로, 비공간 속성값은 Zipfian 분포에 따른 선택율이 가장 높은 상위 50개의 속성값을 선택하였다. 그 이유는, 선택율이 높으면 동일한 속성값을 갖는 투플의 수가 증가하며, 따라서 질의 최적화시 정렬(sort) 비용이 큰 순차 스캔(scan) 방식보다 공간 인덱스를 사용할 가능성이 커지기 때문이다. 질의 결과 k는 1, ..., 35 범위를 가지며, k가 비교적 작은 이유는, k가 상대적으로 큰 경우 순차 검색이 유리하고 실제적으로 작은 k가 사용되기 때문이다[1]. R*-트리의 캐쉬의 크기는 하나의 페이지이며, 테이블 접근 때 사용되는 캐쉬의 크기는 전체 테이블의 크기의 1%를 사용하였다. 매 질의 수행시 시스템의 OS 캐쉬의 영향을 최소화하기 위해 시스템의 메모리보다 큰 파일 스캔 작업을 먼저 수행하였다.

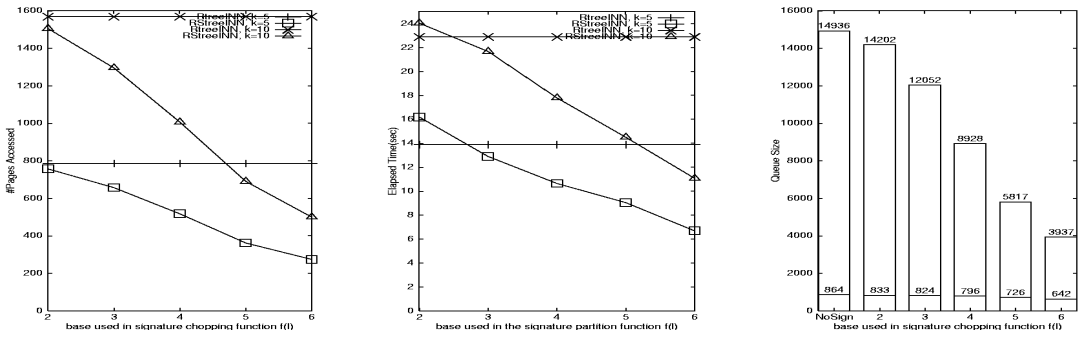
두 알고리즘의 구현 프로그램을 수행시키기 위해서 Pentium 133MHz 프로세서, 128MB의 메모리, 6GB 디스크 드라이브를 갖는 시스템과 운영 체제 PowerLinux 6.0을 사용하였다.

6.1 인위 데이타(Synthetic Dataset)

본 절에서의 실험은 인위 데이타를 사용하였으며, 실험 결과는 R*-트리가 중저차원에서 효율적이므로d=6인 공간 데이타를 중심으로 수행된 것이다.

질의 결과 k에 따른 성능 평가: 테이블 DISC의 color 속성값의 차원 d=6, artist의 속성값의 Zipfian 변수 z=0.5, 전체 투플의 수 N=10^5, 시그니쳐 분할 함수 $f(l)$=6^l, 시그니쳐의 비트 수 F=64일 때, 질의 수행시간, 페이지 접근 수, 투플 접근 수는 그림 5와 같다.

그림 5(a)에서 투플 접근 수는 k개의 질의 결과를 얻기까지 전체 후보 투플의 수와 같다. 두 알고리즘 모두 k가 커질수록 접근해야 하는 투플 수가 선형적으로 증가하지만, 직선의 기울기에서 크게 차이가 난다. 그 이유는 *RStreeINN*이 제 5.2절에서 설명한 S-트리의 제거 효과를 가지기 때문이다. 그림 5(b)에서 두 알고리즘의 페이지 접근 수는 그림 5(a)의 투플 접근 수보다 약간 큰 것을 알 수 있다. 그 이유는 두 알고리즘이 R-트리 또는 S-트리의 페이지를 추가적으로 접근하기 때문이다. 두 알고리즘 모두 접근 페이지 수가 k가 커짐에 따라 선형적으로 증가하며, 그림 5(a)의 그래프와 거의 닮은 꼴을 유지한다. 그 이유는, 실험 데이타가 상대적으로 작은 경우 접근된 페이지 수는 공간 인덱스보다 투플 접근 비용에 더 영향을 받기 때문이다. 그림 5(b)의 페이지 접근 수에서의 차이는 그림 5(c)의 질의 수행 시간에 그대로 반영된다. 그러나, 그림 5(c)에서 그림 5(b)의 페이지 접근 수의 차이에서 예상할 수 있는 두



(a) 페이지 접근수 (b) 질의 수행 시간 (c) 큐 크기

그림 6 시그니쳐 분할 함수 $f(l)$에 따른 실험 결과(d=6, N=10^5, z=0.5, $f(l)$=6^l, F=64)

알고리즘의 성능 차이의 폭이 줄어든 것을 알 수 있다. 그 이유는, *RStreeINN*은 R-트리와 S-트리를 교대로 접근해야 하므로 랜덤 디스크 I/O가 발생하여 추가적인 비용을 더 지불해야 하기 때문이다.

시그니쳐 분할 함수 $f(l)$에 따른 성능 평가: *RStreeINN*에서 S-트리 노드의 각 엔트리가 가질 수 있는 시그니쳐의 수에 따라 전체 성능이 달라질 수 있다. 전체 성능은 시그니쳐의 수에 비례한다는 것을 예상할 수 있다. 우리는 오버플로우 페이지가 생기지 않는 범위내에서 시그니쳐 분할 함수 $f(l)$에 따른 실험을 수행하였으며, 그 결과는 그림 6과 같다.

그림 6(a)에서 *RStreeINN* 기법의 경우 시그니쳐 분할 수가 증가함에 따라 페이지 접근 수가 상당히 감소함을 알 수 있으며, *RtreeINN* 기법과 비교할 때 큰 폭의 차이를 보인다. 이러한 차이는 그림 6(b)의 질의 수행 시간에 그대로 반영된다. 그러나, $f(l)$=2^l인 경우 *RStreeINN*이 *RtreeINN*보다 성능이 나쁘다. 그 이유는, *RStreeINN*에서는 S-트리를 통해 얻는 이익보다 S-트리의 접근으로 발생하는 랜덤 디스크 I/O에 의한 손해가 더 크기 때문이다. 그림 6(c)는 k=10일 때 $f(l)$에 따른 큐의 크기를 나타내며, *NoSign*은 *RtreeINN*에, *base*>2인 경우는 *RStreeINN*에 해당한다. 그림에서 아래쪽과 윗쪽 박스의 숫자는 각각 큐에 삽입된 R-트리 노드의 수와 전체 큐 크기를 의미하며, 두 값의 차는 큐에 삽입된 객체(또는 TID)의 수를 의미한다. 그림 6(c)에서 시그니쳐 분할 수가 커짐에 따라 큐에 삽입된 R-트리 노드와 객체의 수가 줄어듬을 알 수 있으며, S-트리에서 시그니쳐 포화와 팬텀 효과가 시그니쳐 분할에 의해 완화되었음을 의미한다. 그리고, 큐 크기의 차이는 성능의 차이로 나타남을 알 수 있다(그림 6(a-b)).

데이타 크기에 따른 성능 평가: 이 실험에서 실험 변수값은 d=6, z=0.5, $f(l)$=6^l, F=64를 사용하였다. 그림 7에서, 데이타 크기가 커짐에 따라 *RStreeINN*이 *RtreeINN*보다 성능이 우수함을 알 수 있다. 그리고, *RtreeINN*에서는 데이타 크기가 커짐에 따라 질의 수행 시간도 비교적 큰기울기로 증가하지만, 반면 *RStreeINN*에서는 그래프의 기울기가 완만하다. 이 사실에서 *RStreeINN*은 데이타 크기에 상관없이 그 성능을 보장함을 알 수 있다.

공간 데이타의 차원(d)에 따른 성능 평가: 이 실험에서 실험 변수 값은 N=10^5, z=0.5, $f(l)$=6^l, F=64를 사용하였으며, d=2인 경우 S-트리에서 각 노드마다 하나의 오버플로우 페이지가 발생하였다. 그림 8에서, 각 차원에 대해 *RStreeINN*이 *RtreeINN*보다 우수한 것을 알 수 있다. 그러나, 차원이 증가할수록 성능의 차이는 감소한다. 그 이유는, 고차원 R-트리에서는 저차원과 달리 데이타 노드의 용량이 작아서[6] 질의의 비공간 속성 값이 전체 데이타 노드에 고르게 분포하여 "시그니쳐 검사"를 통과할 확률이 높게 되어 *RStreeINN*가 접근하는 R-트리 노드의 수가 증가하기 때문이다.

6.2 실세계 데이타(Real Dataset)

이 실험에서는 공간 데이타는 실세계 데이타로서 16차원의, CAD 모델에 사용하는 고차원 푸리에(Fourier) 점을 사용하였으며[2], 비공간 데이타는 인위 데이타로서 Zipfian 변수 z=0.5을 갖는 분포를 따른다. 그 이외의 실험 변수 값은 N=200000, $f(l)$=6^l, F=64와 같다. 차원 d=16일 때 R*-트리의 성능은 효율적이지 못하지만, 두

6) 16 차원 R-트리의 데이타 노드 용량은 2 차원 경우의 1/4 배 이하이다.

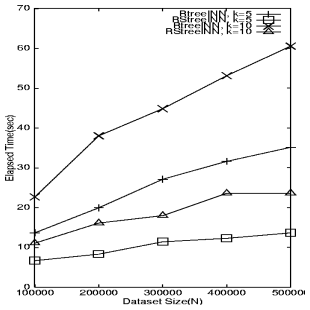


그림 7 데이타 크기에 따른 질의 수행 시간

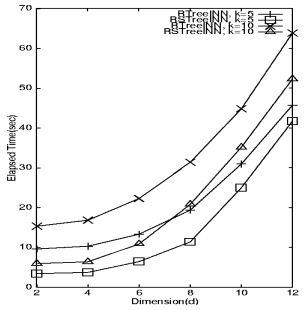


그림 8 데이타 차원(d)에 따른 질의 수행 시간

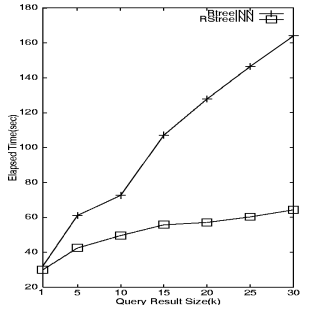


그림 9 실세계 데이타에 대한 질의 수행 시간

알고리즘의 성능 비교에는 별 무리가 없으리라 생각한다. 고차원에서는 R-트리 노드의 용량이 작아지기 때문에 두 알고리즘이 중저차원에서와 같은 성능은 보이지 않지만, *RStreeINN*이 *RtreeINN*보다 우위에 있음을 알 수 있다(그림 9).

7. 결 론

본 논문에서 Hjaltason과 Semet이 제안한 R-트리를 이용한 점증적 최근접 알고리즘을 사용하여 비공간 검색 조건이 포함된 *k*-최근접 질의를 처리할 때 발생하는 불필요한 후보 투플의 수를 줄이기 위해서 RS-트리를 이용한 점증적 최근접 알고리즘을 제안하였다. 우리의 알고리즘을 이용하면 S-트리의 시그니쳐 검사를 통하여 비공간 검색 조건을 만족시킬 가능성이 낮은 R-트리 노드나 객체를 큐에 삽입하지 않으므로써 불필요한 후보 투플을 부분적으로 제거하여 전체 질의 처리 성능을 높일 수 있다. 그리고, S-트리의 각 레벨에서 "시그니쳐 포화"를 완화시킬 수 있는 "시그니쳐 분할" 기법을 통해 S-트리의 제거 효과를 높였다. 마지막으로, 다양한 실험 결과를 통해 우리의 알고리즘이 Hjatalson과 Samet의 알고리즘보다 우수함을 보였다.

참 고 문 헌

[1] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems*, 24(2), June 1999.

[2] S. Berchtold, C. Bohm, D. A. Keim, and H.-P. Kriegel. A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space. In *Proc. of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, May 1997.

[3] A. J. Broder. Strategies for Efficient Incremental Nearest Neighbor Search. *Pattern Recognition, 23(1-2)*, January 1990.

[4] A. Henrich. A Distance-scan Algorithm for Spatial Access Structures. In *Proc. of the Second ACM Workshop on Geographic Information Systems*, December 1994.

[5] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proc. of the ACM SIGMOD Conf.*, May 1995.

[6] Ronald Fagin. Fuzzy Queries in Multimedia Database Systems. In *Proc. of the 16th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, June 1998.

[7] Walter W. Chang and Hans J. Schek. A Signature Access Method for the Startbust Database System. In *Proc. of the 15th Int'l Conf. on Very Large Data Bases*, August 1989.

[8] J. M. Patel et al. Building a Scaleable Geo-Spatial DBMS: Technology, Implement ation, and Evaluation. In *Proc. of the ACM SIGMOD Conf.*, June 1997.

[9] S. Grumbach, P. Rigaux, and L. Segoufin. The DEDALE System for Complex Spatial Queries. In *Proc. of the ACM SIGMOD Conf.*, June 1998.

[10] V. E. Ogle and M. Stonebraker. Chabot: Retrieval from a Relational Database of Images. *IEEE Computer*, 28(9), September 1995.

[11] M. Flickner et al. Query by Image and Video Content: The QBIC System. *IEEE Computer*, 28(9), September 1995.

[12] T. Seidl and H.-P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *Proc. of the ACM SIGMOD Conf.*, June 1998.

[13] R. Weber, H.-J. Schek, and S. Blott. A Quantitative Analysis and Performance Study for

Similarity-Search Methods in High-Dimensional Spaces. In *Proc. of 24rd International Conf. on Very Large Data Bases*, August 1998.
[14] S. Berchtold, C. Bohm, B. Braunmuller, D. A. Keim, and H.-P. Kriegel. Fast Parallel Similarity Search in Multimedia Databases. In *Proc. of the ACM SIGMOD Conf.*, June 1997.
[15] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. *Journal of the ACM*, 45(6), Nov. 1998.
[16] S.B. Yao. Approximating Block Accesses in Database Organization. *Communications of the ACM*, 20(4), April 1977.
[17] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient And Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD Conf.*, June 1990.
[18] C. Faloutsos and S. Christodoulakis. Optimal Signature Extraction and Information Loss. *ACM Transactions on Database Systems*, 12(3), September 1987.

박 동 주
1995년 서울대학교 컴퓨터공학과 학사. 1997년 서울대학교 컴퓨터공학과 석사. 1997년 ~ 현재 서울대학교 컴퓨터공학과 박사과정. 관심분야는 데이타베이스, 멀티미디어 데이타베이스, 공간 데이타베이스

김 형 주
정보과학회논문지 : 컴퓨팅의 실제
제 6 권 제 1 호 참조