질의 결과를 이용한 거리 브라우징 질의의 처리

박 동 주[†] · 박 상 원[††] · 정 태 선[†††] · 이 상 원[††††]

요 약

k-최근접 질의와 같은 거리 브라우징 질의는 지리정보시스템(GIS)과 같은 공간 데이터베이스 응용에서 아주 중요한 질의이다. 최근 GIS 응용은 웹과 같은 다중 사용자 환경으로 확장되고 있는 추세이다.이러한 질의를 처리하기 위한 많은 기법들 중에서 Hjaltason과 Samet이 제안한 알고리즘이 가장 우수하지만, 하나의 질의 처리에 대해서만 최적화가 이루어졌다. 따라서 다중 사용자 환경에 적합하도록 이러한 기법들을 보완할 필요성이 있다. 이전에 처리된 질의 결과를 캐쉬에 저장해 두고(즉, 질의 결과 캐싱 기법) 후속 질의를 처리할 때 질의 결과를 이용하는(즉, 질의 결과 매칭 기법) 것은 하나의 좋은 접근 방법이라 할 수 있다. 본 논문은 다중 사용자 GIS 환경에서 거리 브라우징 질의를 효율적으로 처리하기 위해서 캐쉬된 이전 질의 결과를 재사용할 수 있도록 보완된 Hjaltason & Samet의 알고리즘을 제안한다. 실험 결과를 통해 우리의 접근 방법이 효율적임을 보인다.

키워드 : 거리 브라우징 질의, k-최근접 질의, GIS, 질의 결과 캐싱

Distance Browsing Query Processing using Query Result Set

Dong-Joo Park[†] · Sangwon Park[††] · Tae-Sun Chung[†††] · Sang-Won Lee[††††]

ABSTRACT

Distance browsing queries, namely k-nearest neighbor queries, are the most important queries in spatial database applications, e.g., Geographic Information Systems(GISs). Recently, GIS applications trends to extend themselves toward wide multi-user environments such as the Web. Since many techniques for such queries, where Hjaltason and Samet's algorithm is the most efficient one, were optimized for only one query, we need to complement them suitable for multi-user environments. It can be a good approach that we store many individual query results in a cache, i.e., query result caching and reuse them in evaluating incoming queries, i.e., query result matching. In this paper, we propose a complementary Hjaltason and Samet's algorithm capable of reusing previous query results in a cache for answering distance browsing queries in multi-user GIS environments. Our experimental results confirm the efficiency of our approach.

Key Words : Distance Browsing Query, k-Nearest Neighbor Query, GIS, Query Result Caching

1. 서 론

공간 데이터베이스 응용, 예를 들면 지리정보시스템(GIS)과 같은 분야에서 "주어진 질의 객체로부터 가장 가까운 k개의 최근접 객체를 구하라"와 같은 질의는 공간 질의 중에서 가장 일반적이다. 최근까지 이러한 질의를 처리하기 위한 많은 기술들이 개발되어 왔다[6, 15, 16, 20]. 웹의 등장으로 GIS 응용은 웹과 같은 다중 사용자 환경으로 확장되고 있으므로[1, 2, 3, 9]서버는 더 많은 사용자로부터 질의를 받게 되며, 따라서 효율적으로 처리할 수 있어야 한다. 거리 브라우징 질의는 하나의 질의를 처리하는데도 상당한 질의 비용을 요구하기 때문에 웹과 같은 환경에서는 전체적인 성능이 저하될 수 있다. 따라서 이러한 문제를 해결할 수 있는 새로운 기법이 요구된다.

사용자는 주어진 질의점에 대한 질의 결과에 대해서 만족하지 않을 수 있다. 이후 사용자는 질의 결과의 크기(k)를 늘리거나 새로운 질의점을 가지는 또다른 질의를 생성할 수 있다. 사용자가 새로운 질의점을 찾을 때 이전의 질의점들 중의 하나에 가까운 질의점을 선택할 수 있다. 또한, 서로 다른 사용자로부터 선택된 질의점이 서로 일치하거나 공간적으로 서로 인접할 수 있다. 이러한 경우, 이전 질의 결과를 서버의 캐쉬에 저장해 두면 다음 질의 처리에 재사용할 수 있기 때문에 질의 비용을 줄일 수 있다.

거리 브라우징 질의를 효율적으로 처리하기 위해서 일반적으로 R-트리와 같은 공간 인덱스를 사용한다. 공간 인덱스를 기반으로 하는 기법들 중에서 Hjaltason & Samet의 알고리즘이 가장 우수한 것으로 알려져 있다[16] 그러나 이

※ 본 연구는 숭실대학교 교내연구비 지원으로 이루어졌음.
† 정 회 원 : 숭실대학교 컴퓨터학부 전임강사
†† 종신회원 : 한국외국어대학교 컴퓨터및정보통신공학부 조교수
††† 정 회 원 : 아주대학교 정보및컴퓨터공학부 조교수
†††† 정 회 원 : 성균관대학교 정보통신공학부 조교수
논문접수 : 2005년 5월 18일, 심사완료 : 2005년 7월 12일

알고리즘은 하나의 질의에 대해서 최적화가 되어 있기 때문에 웹 기반 다중 사용자 GIS 환경에서 좀더 성능을 개선시킬 여지가 있다. Hjaltason & Samet 알고리즘은 다중 사용자로부터 만들어진, 질의 결과가 겹칠 수 있는 여러 개의 질의들에 대해 서로 독립적으로 처리하기 때문에 그 비용은 개개의 질의 처리 비용의 합과 같다. 이것은 대단한 자원의 낭비이며, 이전 질의의 결과를 재사용하지 않기 때문에 발생하는 것이다. 본 논문에서는, 웹 기반 GIS 응용에서 Hjaltason & Samet의 알고리즘의 성능을 개선시키기 위해서 캐쉬에 저장된 이전 질의 결과를 질의 처리에 이용하는 기법을 제안한다.

관계형 데이터베이스에서 어떤 테이블에 대한 질의를 처리할 때 접근한 페이지를 디스크나 메인 메모리에 캐쉬해 두고, 다음 질의를 처리할 때 캐쉬 데이타를 이용하는 질의 처리 기법은 아주 일반적이다. 이러한 캐쉬의 관점에서 공간 질의 결과를 버리지 않고 디스크에 저장해 두고 다음 질의 처리시 재사용하는 것도 전체 시스템의 성능을 높일 수 있는 방법이 될 수 있다. 서로 다른 점은, 전자는 질의 처리시 접근한 테이블 페이지를 저장해 두지만, 후자는 질의 결과를 저장해 둔다는 것이다. 이러한 질의 결과를 재사용하는 기술은 이미 데이타 웨어하우스(Data Warehouse) 분야에서 뷰(View)를 통해 구체화되어 왔다[11, 14, 17]. 본 논문에서는 일반적인 캐쉬의 관점에서 거리 브라우징 질의 결과의 재사용성에 관해 알아보며, 또한 이것을 이용하기 위한 Haltason & Samet의 알고리즘의 확장에 관해 서술한다.

본 논문은 다음과 같이 구성되어 있다. 2장에서는 관련 연구에 대해서 알아 보며, 3장에서는 거리 브라우징 질의, R-트리, Hjaltason & Samet의 알고리즘에 대해서 서술한다. 4장에서는 우리의 기본 아이디어를 서술하며 5장에서는 질의 결과를 캐쉬에 저장, 관리하는 방법에 대해서 설명한다. 6장에서는 Hjaltason & Samet의 알고리즘을 보완한 캐쉬 기반 알고리즘을 설명하며, 7장에서는 실험 결과를, 마지막 장에서는 결론을 맺는다.

2. 관련 연구

k-최근접 질의는 이미지 검색 시스템, 지리 정보 시스템, 전자 상거래 시스템과 같은 데이터베이스 응용에서 아주 일반적인 질의 형태이다. 이러한 질의 처리를 위한 대부분의 기법들은 오직 하나의 질의에 대해서만 최적화가 되어 있을 뿐, 질의 처리에 이전 질의 결과를 이용할 수 있다는 사실을 간과하고 있다. [7]은 이전보다 더 나은 질의 결과를 얻기 위해서 질의를 수정한 후 재실행시키는 경우, 수정 질의(Refined query)의 성능을 개선하기 위한 기법을 제안하고 있다. 수정 질의를 처리할 때 R-트리에 기반한 Hjaltason & Samet 알고리즘을 이용하는데 이전 질의 처리시에 접근된 R-트리 페이지를 재사용한다는 특징이 있다. 그러나 우리의 캐쉬 기반 접근 방식은 질의 결과를 캐쉬에 저장하고 재사용한다는 점에서 [7]과 다르다.

[5]는 데이타 마이닝(Data Mining) 응용에서 사용자가 주어진 데이터베이스를 분석할 때 새로운 질의를 생성시 이전의 질의점들과 가까운 질의점을 선택한다는 가정을 하고 있다. 이러한 가정하에, [5]는 다중 유사 질의(Multiple similarity query)를 제안했으며, 개개의 유사 질의를 순차적으로 처리하는 것이 아니라 여러 개의 유사 질의를 하나의 다중 유사 질의로 취급하여 한번에 처리하는 기법을 제안하였다. [5]의 가정이 사용자가 다음 질의를 생성할 때 이전의 질의점과 이웃하는 질의점을 선택한다는 점에서 우리의 접근 방법과 유사하지만 질의 결과를 캐쉬에 저장하고 재사용한다는 점에서는 다르다.

후속 질의 처리에 질의 결과를 재사용하는 기법은 관계형 데이터베이스 분야[8, 12, 18]나 데이타 웨어하우스 분야[11, 14, 17]에서 다루어져 왔다. 하지만, 아직까지 GIS 응용에서 거리 브라우징 질의를 처리할 때 질의 결과를 캐쉬에 저장하고 재사용하는 기법은 제시되지 않았다. 따라서 우리가 제안하는 기법은 공간 데이터베이스 응용, 특히 GIS 응용에서 유용하게 사용될 수 있을 것이다.

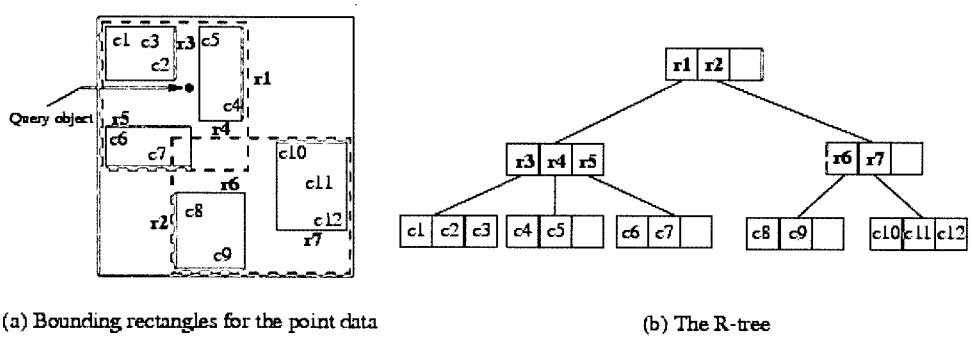
3. 예비 사항

3.1 거리 브라우징 질의

거리 브라우징 질의, 즉 k-최근접 질의는 d-차원 데이타 공간에서 주어진 질의점에서 거리 순서대로 k개의 객체를 구하는 질의를 의미한다. 본 논문에서 우리는 GIS 응용에 초점을 맞추기 때문에 질의는 2차원의 유클리디언(Euclidean) 데이타 공간에 기반함을 가정한다. 질의 결과 크기 k는 일반적으로 작지만, 질의문에 거리 브라우징을 위한 검색 조건뿐만 아니라 비공간 검색 조건이 포함될 수 있다[19]. 예를 들면, "백만명 이상의 인구를 가지는 서울에서 가장 가까운 k개의 도시를 검색하라(nearest(City=Seoul, k) ∧ Population ≥ 1M)"는 질의가 있을 수 있다. 이러한 경우, 각각의 질의 결과 객체는 비공간 검색 조건(즉, Population ≥ 1M)을 만족해야 하므로 질의 처리기는 k보다 큰 k'개의 객체를 탐색해야 한다. 여기서 k'는 수 천의 값을 가질 수 있다[19]. 위와 같은 경우, 질의 결과를 캐쉬에 저장할 때 k가 아닌 k'개의 질의 결과 객체를 저장한다.

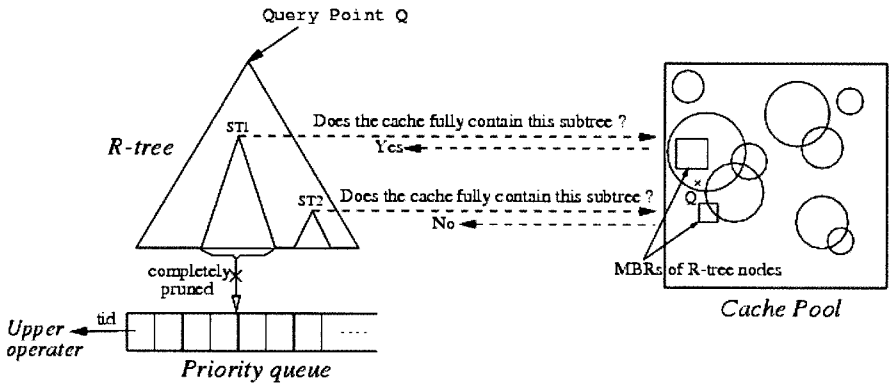
3.2 R-트리

(그림 1)과 같이, R-트리[13]는 B+-트리[10]와 같이 균형 구조(Balanced structure)를 갖는 객체 계층 구조를 말한다. R-트리 노드는 데이타 노드와 인덱스 노드로 구분되며, 각 노드는 (*key, pointer*) 쌍의 배열을 가진다. 여기서 *key*는 *pointer*가 가리키는 서브트리에 포함되는 모든 공간 객체를 둘러싸는 최소 사각형(MBR : Minimum Bounding Rectangle)을 의미한다. 인덱스 노드인 경우의 *pointer*는 다음 레벨의 자식 노드를 가리키는 포인터이며, 데이타 노드인 경우는 R-트리 외부에 저장되어 있는 실제 공간 객체를 가리키는 객체 식별자(ID)를 의미한다. R-트리 노드가 가질 수 있는 엔트리의 수는 노드 용량(Node capacity) 또는 팬아웃



(a) Bounding rectangles for the point data (b) The R-tree

(그림 1) R-트리 인덱스



(그림 2) 질의 결과를 이용한 질의 처리

(Fan-out)이라 하며, 데이타 노드와 인덱스 노드에 따라 달라질 수 있다. 노드 용량은 보통 디스크 페이지의 크기에 따라 결정된다.

3.3 Hjaltason & Samet 알고리즘

[16]에서 Hjaltason과 Samet은 R-트리를 이용한 점증적 최근접 알고리즘을 제안하였다. 이후, 이것을 RtreeINN 알고리즘이라 부른다. R-트리의 루트 노드부터 탐색하면서, 다음 탐색 노드는 큐의 최전방 레코드에서 얻으며, 그 노드를 탐색한 후 그 노드의 자식 노드 또는 공간 객체를 큐에 삽입하는데 이때 질의 객체로부터 자식 노드 또는 공간 객체와의 거리를 함께 삽입한다. 항상 큐는 거리 순서대로 유지된다. 만약 큐의 맨 앞에서 꺼낸 데이터가 공간 객체일 때, 이것을 포함하는 튜플의 식별자(TID)를 나머지 검색 조건을 처리를 위해 질의 처리 연산자에게 반환한다. k개의 질의 결과를 얻거나 또는 큐에 레코드가 없을 때까지 이 과정을 계속 반복한다.

4. 주요 아이디어

본 절에서는 거리 브라우징 질의를 성능을 향상시키기 위한 주요 아이디어를 설명한다. 우선 Hjaltason & Samet 알고리즘에 의해 매번 처리된 질의 결과는 캐쉬에 저장되고 주어진 캐쉬 교체 알고리즘에 의해 관리된다. 이후 캐쉬내의 각 질의 결과를 청크(Chunk)라 한다. Hjaltason & Samet 알고리즘에 의해 큐(Queue)에서 가져온 객체의 타입이 R-트리 노드인 경우, 알고리즘은 그 노드의 각 자식 노드에 대해서 자식 노드의 MBR을 완전히 포함하는 청크를 찾는다. 이것을 "포함 검사(Containment test)"라 부른다. 만약 이러한 청크가 존재하면, 해당 자식 노드의 서브트리는 완전히 제거되고 큐에 삽입되지 않는다((그림 2)의 ST1). 대신, 그 서브트리 아래에 존재하는 모든 공간 객체, 즉 해당 청크내의 공간 객체들 중에서 주어진 MBR에 포함되는 부분을 큐에 바로 삽입한다. 따라서 알고리즘은 제거된 서브트리에 해당하는 탐색 공간(Search space)을 줄일 수 있다. 만약 주어진 MBR를 포함하는 어떤 청크도 존재하지 않으면, 이전의 Hjaltason & Samet 알고리즘과 동일하게 해당 자식 노드를 큐에 삽입한다((그림 2)의 ST2).

새로운 청크를 캐쉬에 삽입할 때 주어진 캐쉬 공간이 부족한 경우 캐쉬 관리자가 어떤 청크를 교체할 것인가는 아주 중요하다. 위에서 설명한 포함 검사를 수행할 때, 캐쉬내의 모든 청크를 검사하는 것은 비효율적이기 때문에 좀더 효율적인 기법이 요구된다. 다음 절에서는 이러한 이슈에 대해서 좀더 자세히 설명한다.

5. 질의 결과 캐싱 & 매칭 기법

시스템은 질의 결과 캐싱과 질의 결과 매칭을 위한 두 개

의 모듈(Module)을 가진다. 질의 처리 후 질의 결과는 메인 메모리 캐쉬에 저장되고 캐쉬 모듈에 의해 관리된다. 캐쉬 모듈은 캐쉬 공간이 부족할 때 캐쉬 교체 알고리즘을 수행한다. 후속 질의는 매칭 모듈에서 처리되며, 매칭 모듈은 질의 처리시 Hjaltason & Samet이 필요로 하는 유익한 캐쉬 청크를 찾아서 재사용하는 작업을 수행한다.

5.1 질의 결과 캐싱(Query Result Caching)

아래에서는 캐쉬 관리에 대한 주요 이슈에 대해서 설명한다. 앞에서도 언급하였듯이, 각각의 질의 결과를 청크라 부르며, 간단히 c라 표기한다.

청크를 저장할 때 메인 메모리 캐쉬를 사용한다. 디스크 캐쉬는 그 용량이 크기 때문에 더 많은 청크를 저장할 수 있다. 그러나, 질의 처리시에 청크의 내용을 읽기 위한 추가적인 디스크 I/O 비용이 필요하기 때문에 질의 처리에 청크를 이용할 때 얻는 이익이 상쇄될 수 있다. 따라서, 청크를 읽을 때 추가적인 디스크 I/O 비용을 없애기 위해서 메인 메모리 캐쉬를 사용한다.

새로운 청크를 캐쉬에 저장할 때 저장 공간이 부족할 경우 캐쉬 내의 어떤 청크를 교체할 것인가를

결정해야 한다. 이를 위해서 몇 개의 캐쉬 교체 알고리즘을 제시한다.

- LRU(Least Recently Used) : 참조 시간이 가장 오래된 청크를 교체한다.
- LFU(Least Frequently Used) : 참조 횟수가 가장 작은 청크를 교체한다.
- SIZE : 청크의 크기, 즉 청크를 표현하는 원의 반지름이 가장 작은 청크를 교체한다.
- SPF(Small Penalty First) : 청크 c에 대해서 $\frac{frequency(c)\cdot \cos t(c)}{size(c)}$ 의 값이 가장 작은 청크를 교체한다. 여기서 *frequency*(c), *size*(c)는 각각 청크 c에 대한 참조 횟수, 크기를 나타내며, *cost*(c)는 청크 c가 교체된 후 다시 같은 c를 만들어 낼 때의 비용을 의미한다. 이 캐쉬 교체 기법은 [17]에 있는 기법과 유사하다.

테이블의 어떤 공간 객체가 수정된 경우 캐쉬의 청크도 같이 수정해야 한다. 어떤 공간 객체가 테이블로부터 삭제된 경우, 그 객체를 포함하고 있는 청크들을 찾은 후 해당 객체를 삭제한다. 공간 객체가 수정된 경우, 그 객체를 포함하고 있는 청크의 크기도 수정되어야 하는지 검사를 해야 한다. 빠른 수정을 위해서, 캐쉬 관리자는 캐쉬내의 청크 뿐만 아니라 청크가 가지고 있는 각 공간 객체의 TID 정보에 대한 해쉬 테이블을 유지하고 있다. 하지만, 공간 데이터베이스에서의 수정은 자주 발생하지 않으므로, 본 논문에서는 청크를 수정하는 비용은 고려하지 않는다.

5.2 질의 결과 매칭(Query Result Matching)

아래에서는 Hjaltason & Samet 알고리즘을 이용하여 질의를 처리할 때, 후보 청크의 선택과 이용에 대해서 알아본다.

주어진 R-트리 노드가 캐쉬 내의 어떤 청크에 포함되는지 검사할 때, 모든 청크를 탐색하는 것은 비효율적이다. 따라서 후보 청크의 수(m)에 제약을 두고 주어진 질의점으로부터 가장 가까운 m개의 청크를 선택한다. 최적의 m을 찾는 것은 어려우므로, 질의로 주어진 k의 두배 또는 세배의 공간 객체를 포함할 수 있을 정도의 m개의 후보 청크를 선택하는 휴리스틱(heuristic)을 사용한다.

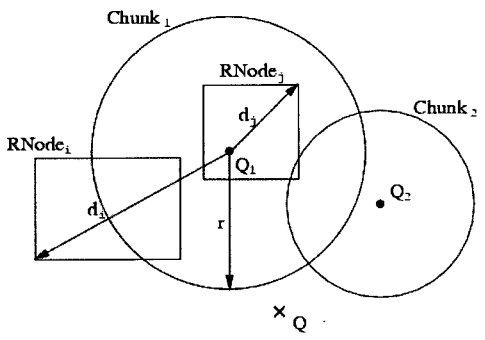
질의점에서 가장 가까운 m개의 후보 청크를 찾을 때 디렉토리 인덱스(Directory Index)를 이용한다. 디렉토리 인덱스는 R-트리를 이용하여 청크를 저장하며, Hjaltason & Samet 알고리즘을 이용하여 후보 청크를 찾는다.

5.2.1 디스크 I/O 줄이기

질의 처리시 후보 청크를 이용하는 방법에 대해서 설명한다. (그림 3)에서 후보 청크는 Chunk1과 Chunk2이며, 각 청크의 질의점은 Q1과 Q2임을 가정한다. 또한, 현재 질의점 Q에 대해서 Chunk1이 Chunk2보다 가깝다는 것을 가정한다. 따라서 "포함 검사"노드인 어떤 R-트리 노드가 주어지면, 포함 검사는 Chunk1부터 시작된다. 만약 주어진 R-트리 노드가 Chunk1에 포함되지 않으면, 포함 검사는 Chunk2에 대해 수행된다. R-트리 노드의 어떤 청크에 대한 포함 검사는 다음과 같이 수행된다. 먼저, 주어진 R-트리 노드의 MBR의 네 개의 꼭지점 중 청크의 질의점에서 가장 멀리 떨어진 꼭지점까지의 최대 거리 d를 계산한다. 청크의 반지름을 r이라고 할 때, 만약 $d \le r$ 조건을 만족시키면, 주어진 R-트리 노드가 청크에 완전히 포함된다고 한다((그림 2)의 RNodej). 그렇지 않은 경우, 청크는 R-트리 노드를 포함하지 않는다고 한다((그림 3)의 RNodej).

RNodej와 같이 R-트리 노드가 포함 검사를 만족하면, 그 노드의 MBR에 포함되는 객체들을 구한 후 각 객체에 대해 질의점과의 거리를 계산한 후 거리 순서대로 객체를 큐에 삽입한다. 만약 R-트리 노드가 어떤 후보 청크와도 포함 관계가 성립하지 않으면 이전의 Hjaltason & Samet 알고리즘과 동일하게 다음 연산을 수행한다.

R-트리 노드가 캐쉬 내의 어떤 청크와 포함 관계를 만족시키면 그 노드의 서브트리를 더 이상 탐색할 필요가 없기 때문에 그만큼 R-트리 탐색 공간을 줄일 수 있다. 물론, 포

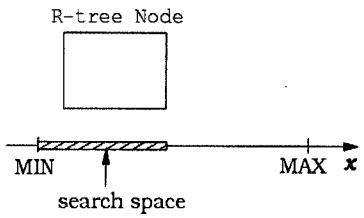


(그림 3) 후보 청크 집합에서의 포함 검사 수행

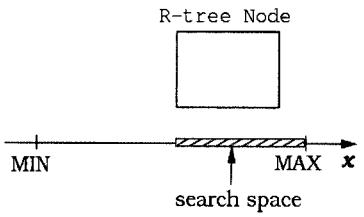
함 검사를 수행하거나 R-트리 노드의 MBR에 포함되는 객체를 찾을 때 추가적인 CPU 비용이 든다. 하지만 메인 메모리 캐쉬를 사용하기 때문에 R-트리 노드의 MBR에 포함되는 객체를 구할 때 추가적인 디스크 I/O는 발생하지 않는다. 아래에서는 위의 CPU 비용을 줄일 수 있는 방법에 대해서 서술한다.

5.2.2 CPU 비용 줄이기

R-트리 노드의 MBR에 포함되는 청크 내의 객체를 찾는데 상당의 CPU 비용이 소요된다. 최악의 경우, 청크 내의 모든 객체에 대해서 해당 MBR에 포함되는지를 검사해야 한다. 검사해야 할 객체의 수를 줄이기 위해서 정렬(sorting) 기법을 사용한다. 우선 질의를 처리한 후 질의 결과를 캐쉬에 저장할 때, 미리 청크 내의 객체들을 한 차원(x 또는 y)에 대해 오름차순으로 정렬한다. (그림 4)(a)과 같이, R-트리 노드가 정렬된 x 값들 중 최대값보다 최소값에 가까우면 최소값부터 전진 탐색(forward search)이 이루어진다. 탐색은 x 차원의 값이 R-트리 노드의 최대 x 값보다 같거나 작을 때까지 수행된다. (그림 4)(b)과 같이, R-트리 노드가 정렬된 x 값들 중 최소값에 가까우면 x의 최대값부터 R-트리 노드의 최소 x 값까지 후진 탐색(backward search)이 이루어진다. 결과적으로, 탐색 공간을 평균적으로 절반까지 줄일 수 있다.



(a) 전진 탐색

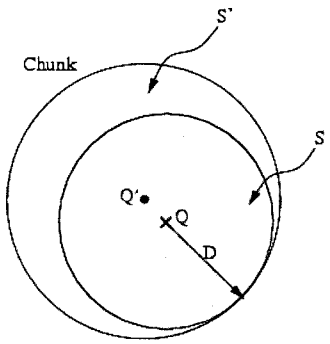


(b) 후진 탐색

(그림 4) 정렬 기법을 이용한 CPU 비용 절감

5.2.3 그 밖의 고려 사항

현재의 질의점이 이전의 질의점들 중 어느 하나에 아주 가까이 있거나, 서로 일치하는 경우를 생각해 볼 수 있다. 이러한 경우에 질의 최적화가 가능하다. (그림 5)와 같이 현재 질의점이 캐쉬 내의 어떤 청크에 포함된 경우를 고려하자. (그림 5)에서 Q와 Q'을 각각 현재 질의점과 청크의 중심점이라고 하자. 먼저 청크에 완전히 포함되는, 중심이 Q이고 반지름이 D인 내접원을 구한다. 그런 후, 위에서 설명



(그림 5) 질의점이 청크 내부에 포함되는 경우

한 정렬 기법을 사용하여 내접원에 포함되는 모든 객체를 구하여 사용자나 상위 연산자에게 전달한다. 만약 전달된 객체 집합이 질의 결과로 충분하면 알고리즘은 더이상 R-트리를 탐색할 필요가 없다.

그러나 전달된 객체 집합의 크기가 작은 경우, 질의 결과로 충분하지 않을 수 있다. 따라서 완전한 질의 결과를 찾기 위해 알고리즘은 R-트리를 처음부터 탐색하지 않을 수 없다. 이러한 불행한 상황을 줄이기 위해서 우리는 다음과 같은 휴리스틱을 제안한다.

(그림 5)와 같이, S'과 S를 각각 청크와 내접원의 면적이라 하자. 또한, k'과 k를 각각 청크 내의 객체의 수와 사용자가 요구하는 질의 결과의 크기라고 가정한다. 만약 $\frac{S}{S'} \cdot k' \geq k$를 만족하면, 질의를 처리하는데 내접원에 포함된 객체들로 충분할 수 있기 때문에, 일단은 청크를 이용하여 질의를 처리한다. 청크로 질의 처리를 완전히 수행하지 못하거나 위의 조건을 만족시키지 않는 경우, 이전의 방식대로 캐쉬 기반 알고리즘을 이용하여 질의를 처리한다.

6. 캐쉬 기반 점증적 최근접 알고리즘

본 절에서는 다중 사용자 GIS 환경에서 전체 질의 성능의 향상을 위한 캐쉬 기반 점증적 최근접 알고리즘을 설명한다. 이 알고리즘은 Hjaltason & Samet 알고리즘을 보완한 것이다.

알고리즘 1과 2는 각각 질의 엔진에서의 거리 브라우징 질의 처리 루틴과 캐쉬 기반 알고리즘을 나타낸다. 알고리즘 1의 함수 ProximitySearchOperator는 알고리즘 2의 함수 CacheBasedINN를 호출한다. 알고리즘 1에서 순위 큐(Priority queue)가 생성되고 루트(root) R-트리 노드를 포함하는 엘리먼트(element)로 초기화된다. 이 루트 R-트리 노드는 이후 첫 번째 대상(target) R-트리 노드가 된다. 또한, 포함 검사를 통과한 R-트리 노드의 MBR에 포함되는 캐쉬 객체를 저장하기 위한 리스트 PreoccupiedObjList가 생성된다(이것에 대해서는 아래에서 자세히 설명한다). 루프(loop) 문은 사용자가 요구한 k개의 질의 결과를 얻기까지 반복적으로 실행된다(단계 7-17). 3 절에서 설명했듯이, 질의에는

거리 조건문뿐만 아니라 다른 비공간 검색 조건문도 포함할 수 있다. 이러한 경우, 연산자 CacheBasedINN가 점증적으로 만들어 낸 TID를 질의 그래프의 해당 연산자에게 전달된다. 질의에 포함된 모든 비공간 검색 조건을 만족하는 k개의 질의 결과는 최종적으로 사용자에게 전송된다(물론, 어떤 경우에는 최종 질의 결과 크기가 k보다 작을 수도 있다).

```
Algorithm 1 ProximitySearchOperator(Q, k)
 1 Queue := PriorityQueue()
 2 PreoccupiedObjList := SortedLinkedList()
 3 NewElm.pointer := RtreeRootNode
 4 NewElm.distance := 0.0
 5 Enqueue(Queue, NewElm)
 6 i := 0
 7 while i is less than k
 8    tid := CacheBasedINN(Q)
 9    if there exist other non-spatial predicates in the query
10       send tid to the corresponding upper operators of the
         query graph
11       if tid satisfies all the remaining predicates
12             i++
13       end if
14    else
15       i++
16    end if
17 end while
```

알고리즘 2에서, Hjaltason & Samet 알고리즘과 다른 추가적인 부분은 포함 검사를 수행하는 부분이다. 어떤 R-트리 노드가 포함 검사를 만족시키면, 이 노드의 MBR에 포함되는 캐쉬 객체들은 거리나 객체 포인터와 같은 다른 정보와 함께 큐에 삽입된다(단계 6-14). 그런데 포함 검사를 수행한 후 얻은 캐쉬 객체들 중 어떤 것은 포함 검사를 통과한 다른 R-트리 노드의 MBR에도 중복적으로 포함될 수 있다. 따라서 그러한 캐쉬 객체들은 중복적으로 큐에 삽입하는 것을 방지하는 작업이 필요하다. 어떤 캐쉬 객체를 큐에 삽입하기 전에, PreoccupiedObjList에 그 객체가 존재하는지를 검사를 한 후 만약 리스트에 존재하지 않으면 그것을 큐에 삽입한다(단계 8).

7. 실험 결과

본 절은 기존의 Hjaltason & Samet 알고리즘과 캐쉬 기반 알고리즘의 성능 비교를 다룬다. 이후의 실험에서는 실세계(reallife) 데이타와 조작(synthetic) 데이타를 사용하였다.

실험을 위해서 5.1 절에서 설명한 캐쉬 교체 알고리즘과 메인 메모리에 기반한 캐쉬 관리자를 구현하였다. 모든 실험은 펜티엄 133MHz 프로세서와 128MB 메인 메모리를 장

```
Algorithm 2 CacheBasedINN(Q)          /* Q denotes a given query object */
 1 while Queue is not empty
 2    Elm := Dequeue(Queue)           /* a next target R-tree node is fetched from the queue */
 3    if Elm.pointer is a non-leaf node /* the next target node is a non-leaf node */
 4       for each entry(child, mbr) in Elm.pointer
         /* the child R-tree node is to be given a inclusion test by the chunk locator */
         /* if the answer is "yes", it returns the objects inside mbr */
 5          ObjSet := ChunkLocator.TestInclusion(mbr)
            /* in the case of the child being fully included in one of candidate chunks in a cache */
 6          if ObjSet.IsEmpty() is False
 7             for each object in ObjSet
 8                if PreoccupiedObjList.Exist(object) is False
 9                   PreoccupiedObjList.Insert(object)
10                   NewElm.distance :=DIST(Q, mbr)
11                   NewElm.tid := object.TID()
12                   Enqueue(Queue, NewElm)
13                end if
14             end for
15          else /* in the case of the child not being included inside any candidate chunk */
16             NewElm.pointer :=child
17             NewElm.distance :=DIST(Q, mbr)
18             Enqueue(Queue, NewElm)
19          end if
20       end for
21    else if Elm.pointer is a leaf node  /* the next target node is a leaf node */
22       for each entry(object, tid) in Elm.pointer
23          if PreoccupiedObjList.Exist(object) is False
24             PreoccupiedObjList.Insert(object)
25             NewElm.distance := DIST(Q, object)
26             NewElm.tid := tid
27             Enqueue(Queue, NewElm)
28          end if
29       end for
30    else  /* Elm.pointer is an object */
31       return Elm.tid
32    end if
33 end while
```

착한, PowerLinux 6.1로 운영되는 시스템에서 수행되었다.

7.1 실험 환경

7.1.1 실험 데이타

실험에서 실세계 데이타로서 Sequoia 2000 벤치마크[21]에 쓰이는 캘리포니아 지역 지도 데이타와 조작 데이타를 사용하였다. Sequoia 데이타는 그 크기가 65,000 개의 점을 가지며 한 점을 중심으로 집중화(clustering)되어 있다. 반면, 조작 데이타는 균일하게 분포하며 1,000,000 개의 점으로 구성되어 있다.

7.1.2 실험 질의

다중 사용자 GIS 환경에서는, 지도 상의 특정한 영역에 대해서 사용자들의 관심이 집중될 수 있으며, 따라서 이러한 핫스팟(hot spot)에 사용자의 질의점이 집중될 수 있다. 이런 사용자 질의 패턴을 모델링하기 위해서, 지도 상에 전체 데이타 공간의 20%에 해당하는 하나의 핫스팟을 만든다. 생성한 실험 질의 집합은 크게 핫스팟에 질의점을 갖는 것과 나머지 영역에 질의점을 갖는 두 종류의 질의로 나눌 수 있다. 사용자가 현재의 질의 결과에 만족하지 않는 경우 다른 질의를 수행시키게 되는데, 이때 그 질의점을 바로 전의 질의점과 이웃하게 선택할 확률이 높을 수 있다. 이런 사용자 질의 패턴을 모델링하기 위해서 근접질의(proximity query)를 만들어 내는데, 즉 어떤 질의점을 중심으로 공간적으로 이웃하는 1~3개의 질의들을 생성한다. 각 실험 데이타에 대해서, 균일하게(random) 생성한 질의들과 근접 질의들을 혼합하여 실험 질의 집합(query stream)을 생성하였다.

핫스팟에 포함되는 질의의 비율에 따라 다섯 종류의 질의 스트림(stream), 즉 QRandom, Q30, Q50, Q70, Q100을 생성하였다. QRandom은 핫스팟과는 상관없이 전체 데이타 공간 상에서 균일하게 생성된 질의들로 구성이 되며, 나머지 질의 집합들은 핫스팟에 포함되는 질의의 비율에 서로 다르다.

7.1.3 자료 구조

실험하려는 두 알고리즘은 R-트리에 기반하지만, 실험을 위해서 변형된 R*-트리[4]를 사용한다. 각 R*-트리의 깊이(depth)를 크게 하기 위해서, R*-트리의 페이지 크기를 1KByte로 고정하였다. 또한, 각 R*-트리의 캐시 크기를 한 페이지이다.

7.1.4 성능 척도

본 논문의 두 질의 처리 방식에 대한 성능 평가 시에 그 척도로서 페이지 I/O 발생 수를 사용하며, 이것은 R*-트리의 페이지를 읽을 때 발생하는 비용과 같다. 물론, CPU 비용도 중요할 수 있으나, 성능 척도로서 고려하지 않는다. 그 이유는, 대부분의 CPU 비용은 5.2.2 절에서 설명한 "포함 검사"를 수행한 후 청크에서 PreoccupiedObjList 객체들을 찾을 때 소요되며, 이 비용은 정렬 기법과 전진 및 후진 검색 기법을 통해 크게 줄일 수 있기 때문이다. 5.2절의 디렉토리 인덱스에서 R*-트리를 사용하기 때문에 페이지 I/O가 발생할 수 있다. 그러나, 캐쉬 풀에 저장되어 있는 청크의 수가 수십에서 수백 개밖에 되지 않으므로 디렉토리 인덱스는 메인 메모리에 상주시킬 수 있다. 따라서, 디렉토리 인덱스에서 발생하는 페이지 I/O 비용은 전체 페이지 I/O 비용에 포함시키지 않는다.

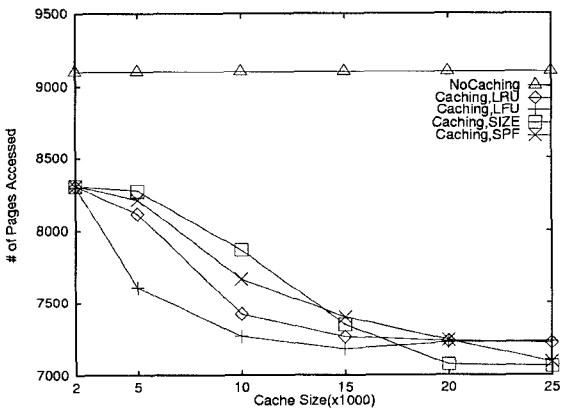
7.2 실험 결과

실험은 크게 조작 데이타와 실세계 데이타에 대해서 수행했으며, 실험결과로서 질의를 처리하는 동안 접근한 R-트리 페이지의 수를 측정하였다. 캐쉬 기반 방식에서는 5.1절에서 언급한 네 개의 캐쉬 교체

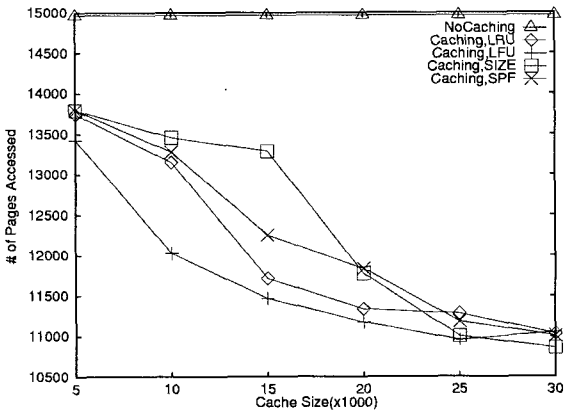
알고리즘을 사용했으며, 또한 여기서 캐쉬 크기로 사용한 m은 m 개의 질의 결과 객체를 저장할 수 있는 메인 메모리 공간을 의미한다.

7.2.1 조작 데이타

(그림 6)은 캐쉬 크기에 따른 캐쉬 기반 알고리즘과 기존의 Hjaltason & Samet 알고리즘의 성능을 보여준다. 그림의 각 곡선은 Q50 타입의 질의 스트림을 수행시키는데 접근한 모든 R-트리의 페이지의 수를 의미한다. 그림에서 질의 결과 크기 k=2,000일 때 캐쉬 기반 방식이 기존의 방식보다



(a) 최대 질의 결과 크기 k=2,000



(b) 최대 질의 결과 크기 k=4,000

(그림 6) 캐쉬 크기에 따른 페이지 I/O의 수

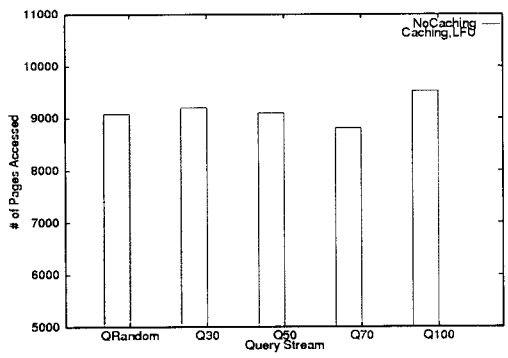
우수하다는 것을 알 수 있다((그림 6)(a)). SIZE 캐쉬 교체 방식을 사용하고 캐쉬 크기 m=20,000인 경우에는 기존 방식에 대한 캐쉬 기반 방식의 성능 증가는 1.3에 이른다. k=4,000의 (그림 6)(b)에서도 캐쉬 기반 방식이 기존 방식보다 우수함을 알 수 있으며, 또한 성능 증가는 최대 1.35에 달한다.

(그림 6)에서 캐쉬 크기가 증가함에 따라 캐쉬 기반 방식의 성능 또한 좋아짐을 알 수 있다. 또한, 캐쉬 크기가 어느 정도에 이르면, 예를 들면, (그림 6)(a)에서 x=15,000일 때 캐쉬 기반 방식의 성능은 최대에 도달함을 알 수 있다. 조작 데이터에 대한 실험에서 우리는 LFU 캐쉬 교체 방식을 따르는 캐쉬 기반 방식이 가장 좋은 성능을 보인다는 결론을 조심스럽게 내릴 수 있다. 그 이유는 다음과 같다. 실험에서 Q50 질의 타입을 사용하였으므로 질의점이 지도 상의 특정 영역에 집중된다고 할 수 있으며, 이러한 질의들을 R-트리를 통해 처리할 때 특정 인덱스 노드들이 자주 참조될 확률이 아주 높다. 따라서 참조 횟수 기반의 LFU 교체 알고리즘을 사용하는 것이 유리하다고 할 수 있다. SIZE 교체 알고리즘의 경우, 캐쉬 크기가 작으면 다른 알고리즘에 비해 성능이 떨어지는데 몇 개의 큰 청크가 캐쉬를 대부분 차지해 버리기 때문에 그만큼 캐쉬 적중률이 낮아지기 때문이다. 하지만 캐쉬 크기가 충분히 커지면 다른 알고리즘에 비해 캐쉬 적중률이 높아지는 것을 알 수 있다((그림 6)).

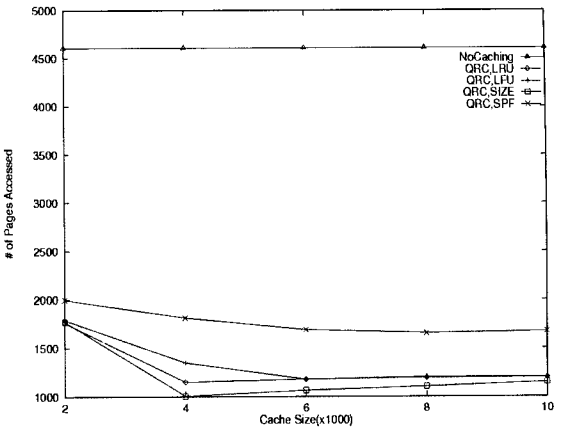
우리는 핫스팟에 포함되는 질의의 비율이 얼마나 두 방식의 성능에 영향을 미치는지 실험을 수행하였다. (그림 7)은 질의 스트림을 달리 했을 때 실험 결과를 보여준다. 그림에서 각 질의 스트림에 대한 왼쪽 막대와 오른쪽 막대는 각각 기존 Hjaltason & Samet 방식과 캐수 기반 방식의 성능을 의미한다. 캐쉬 기반 방식에서 사용한 캐쉬 교체 알고리즘은 LFU을 사용하였다. 그림에서 알 수 있듯이, 핫스팟에 포함되는 질의의 비율이 높아질수록 두 방식간의 성능차이가 증가함을 알 수 있다. 이런 사실로부터 캐쉬 기반 방식은 핫스팟에 집중되는 질의 스트림에 대해서 특히 우수한 성능을 보임을 알 수 있다.

7.2.2 실세계 데이타

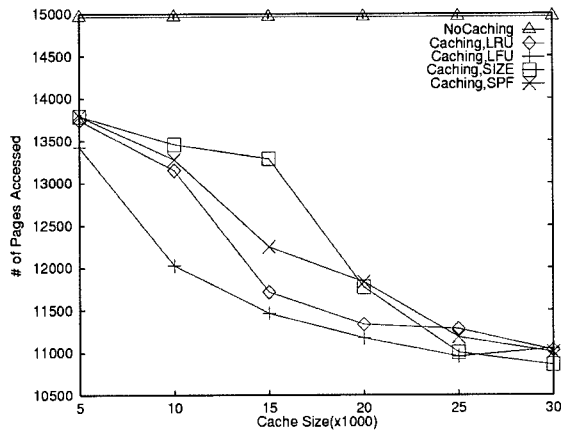
캘리포니아 지역 지도(California Area Map) 데이타를 이용하여 실세계 데이타에 대한 실험을 수행하였다.



(그림 7) 핫스팟에 포함되는 질의 비율에 따른 페이지 I/O의 수



(a) 최대 질의 결과 크기 k=2,000



(b) 최대 질의 결과 크기 k=4,000

(그림 8) 캐쉬 크기에 따른 페이지 I/O의 수

(그림 8)은 캐쉬 크기에 따른 캐쉬 기반 방식과 기존 Hjaltason & Samet 방식 간의 성능을 보여준다. 여기서 질의 스트림 Q50을 사용하였으며, 질의 결과 크기는 k=1,000과 k=2,000으로 설정하였다. 조작 데이타에 대한 실험 결과와 마찬가지로, 실세계 데이터에 대해서도 캐쉬 기반 방식이 기존 방식에 비해서 우수한 성능을 보임을 알 수 있다. 재미있는 사실은 조작 데이타에 대한 실험 결과보다 실세계 데이타의 실험에서 두 방식 간의 성능 차이가 더 크다는 것이다. 예를 들면, (그림 8)(a)에서 NoCaching 방식에 대한 Caching, SIZE 방식의 성능 증가는 최대 2.63에 달한다. 그 이유는 실험에서 사용한 실세계 데이타의 크기(65,000 개의 점)는 조작 데이타의 크기(1,000,000 개의 점)에 비해 매우 작아서 핫스팟 또한 작아지기 때문이다. 핫스팟의 크기가 작아질수록 캐쉬 적중률이 높아지게 된다. 따라서, 실세계 데이타를 이용한 실험에서 캐쉬 기반 방식과 기존 방식과의 성능 차이는 조작 데이타를 이용한 실험 결과와 비교할 때 확률적으로 커지게 된다.

(그림 8)에서 캐쉬 크기가 증가하면 모든 캐쉬 기반 방식들의 성능이 좋아짐을 알 수 있다. 그러나, 조작 데이타의 실험과는 달리 상대적으로 작은 캐쉬 크기에서 최대 성능에

도달함을 알 수 있다 ((그림 8) vs. (그림 6)). 그 이유는, 실세계 데이타의 경우 실험에서 사용한 핫스팟의 크기가 작고 따라서 심지어 작은 크기의 캐쉬에도 대부분의 핫스팟의 데이타를 채울 수 있기 때문이다. 이를 잘 뒷받침할 수 있는 근거는, (그림 8)에서 SIZE 캐쉬 교체 알고리즘을 사용한 캐쉬 기반 방식이 다른 캐쉬 기반 방식보다 더 좋은 성능을 보여준다는 사실이다.

8. 결 론

본 논문에서 우리는 웹과 같은 다중 사용자 환경에 기반한 GIS 응용에서 거리 브라우징 질의를 효율적으로 처리할 수 있는 캐쉬 기반 알고리즘을 제시하였다. 우리의 알고리즘은 심지어 하나의 거리 브라우징 질의 결과도 그냥 버리는 것이 아니고 다음 질의 처리에 계속 재사용해야 한다는 취지에 기반한다. 이러한 목표를 위해서, 메인 메모리 캐쉬에 질의 결과를 저장하고(즉, 질의 결과 캐싱) 다음 질의 처리시에 재사용할 수 있는 방법(즉, 질의 결과 매칭)을 제시하였다. 또한, 질의 결과 매칭에서 CPU 비용을 줄일 수 있는 방법을 설명하였다. 조작 데이타 및 실세계 데이터를 이용한 실험 결과를 통해 우리가 제안한 캐쉬 기반 방식이 기존의 Hjaltason & Samet 방식보다 성능이 우수함을 보였다.

본 논문에서는 거리 브라우징 질의에 대해서만 그 결과를 캐쉬에 저장하고 재사용하는 방법을 제안하였다. 하지만 GIS 응용에서는 이러한 질의뿐만 아니라 다양한 형태의 질의, 예를 들면 영역 질의(Range query), 포인트 질의(Point query), 공간 조인 질의(Spatial join query) 등을 다루고 있으며, 따라서 GIS 서버의 캐쉬에 이러한 질의 결과를 저장하고 재사용할 수 있는 확장된 질의 처리 방법이 필요하다. 또한, 고차원 데이타를 다루는 이미지 검색 응용에서도 본 논문에서 제안하는 방법을 적용시키는 연구가 필요하다.

참 고 문 헌

[1] GeoWeb Project. http://wings.buffalo.edu/geoweb.

[2] The ArcExplorer GIS data viewer. http://www.esri.com/software/arcexplorer/index.html.

[3] T. Barchlay, D. Slutz, and J. Gray. TerraServer: A Spatial Data Warehouse. In Proceedings of ACM SIGMOD 2000 Intl. Conf. on Management of Data, May, 2000.

[4] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An Efficient And Robust Access Method for Points and Rectangles. In Proceedings of the ACM SIGMOD Conference, June, 1990.

[5] B. Braunmuller, M. Ester, H.-P. Kriegel, and J. Sander. Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases. In Proceedings of the 2000 IEEE International Conference on Data Engineering, February, 2000.

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

[7] K. Chakrabarti, K. Porkaew, and S. Mehrotra. Efficient Query Refinement in Multimedia Databases. In Proceedings of the 2000 IEEE International Conference on Data Engineering, February, 2000.

[8] C. M. Chen and R. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. In 4th International Conference on Extending Database Technology, March, 1994.

[9] S. Coetzee and J. Bishop. A New Way to Query GISs on the Web. IEEE Software, 15(3), May/June, 1998.

[10] D. Comer. The Ubiquitous B-tree. ACM Computing Surveys, 11(2), June, 1979.

[11] P. M. Deshpande, K. Ramasamy, and A. Shukla. Caching Multidimensional Queries Using Chunks. In Proceedings of ACM SIGMOD Conference, June, 1997.

[12] S. Finkelstein. Common Expression Analysis in Database Applications. In Proceedings of ACM SIGMOD Conference, June, 1982.

[13] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In Proceedings of the ACM SIGMOD Conference, June, 1984.

[14] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data·cubes efficiently. In Proceedings of ACM SIGMOD Conference, June, 1996.

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

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

[17] Y. Kotidis and N. Roussopoulos. DynaMat: A Dynamic View Management System for Data Warehouses. In Proceedings of ACM SIGMOD Conference, June, 1998.

[18] L.-A. Larson and H. Z. Yang. Computing Queries From Derived Relations. In Proceedings of VLDB Conference, August, 1985.

[19] D.-J. Park and H.-J. Kim. An Enhanced Technique for k-Nearest Neighbor Queries with Non-spatial Selection Predicates. Multimedia Tools and Applications, 19(1), January, 2003.

[20] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In Proceedings of the ACM SIGMOD Conference, May, 1995.

[21] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredith. The Sequoia 2000 Benchmark. In Proceedings of ACM SIGMOD Conference, June, 1993.

박 동 주

e-mail : djpark@ssu.ac.kr
1995년 서울대학교 컴퓨터공학과(학사)
1997년 서울대학교 컴퓨터공학과 (공학석사)
2001년 서울대학교 컴퓨터공학부 (공학박사)
2001년~2003년 삼성전자 책임연구원
2004년~현재 숭실대학교 컴퓨터학부 전임강사
관심분야 : 플래시 메모리, 내장형 소프트웨어, 멀티미디어 데이터베이스 등

박 상 원

e-mail : swpark@hufs.ac.kr
1994년 서울대학교 컴퓨터공학과(학사)
1997년 서울대학교 컴퓨터공학과(석사)
2002년 서울대학교 컴퓨터공학과(박사)
2002년~2003년 세종사이버대학교 디지털콘텐츠학과 조교수
2003년~현재 한국외국어대학교 컴퓨터및정보통신공학부 조교수
관심분야 : DBMS on Flash, 데이터베이스, XML, 객체지향 시스템 등

정 태 선

e-mail : tschung@mju.ac.kr
1995년 KAIST 전산학과(학사)
1997년 서울대학교 전산과학과(석사)
2002년 서울대학교 전기컴퓨터공학부(박사)
2002년~2004년 삼성전자 소프트웨어센터 책임연구원
2004년 3월~2005년 8월 명지대학교 컴퓨터소프트웨어학과 조교수
2005년 9월~현재 아주대학교 정보및컴퓨터공학부 조교수
관심분야 : 플래시 메모리, 데이터베이스 등

이 상 원

e-mail : wonlee@ece.skku.ac.kr
1991년 서울대학교 컴퓨터공학과(학사)
1994년 서울대학교 컴퓨터공학과(석사)
1999년 서울대학교 컴퓨터공학과(박사)
1999년~2001년 한국오라클
2001년~2002년 이화여대 컴퓨터학과
2002년~현재 성균관대학교 정보통신공학부 조교수
관심분야 : XML, DB 튜닝, Data Warehouse/Data Mining