객체관계형 DBMS에서 타입수준 액세스 패턴을 이용한 선인출 전략

(Prefetching based on the Type-Level Access Pattern in Object-Relational DBMSs)

한 욱 신 † 문 양 세 † 황 규 영 ††
(Wook-Shin Han) (Yang-Sae Moon) (Kyu-Young Whang)

요 약 선인출은 데이타베이스 관리 시스템에서 클라이언트와 서버 사이에 발생하는 라운드트립을 줄일 수 있는 효과적인 방법이다. 본 논문에서는 타입수준 액세스 패턴과 타입수준 지역성이라는 새로운 개념을 제시하고, 이 개념에 기반한 새로운 선인출 방법을 제시한다. **타입수준 액세스 패턴**이란 항해에 사용된 애트리뷰트들의 패턴이며, **타입수준 액세스 지역성**이란 항해 응용에서 타입수준 액세스 패턴이 반복적으로 나타나는 현상이다. 기존의 선인출 방법은 항해 응용에서 액세스된 객체 혹은 페이지 식별자들간의 패턴인 객체수준 혹은 페이지수준 액세스 패턴을 선인출에 이용하는데, 이 방법은 동일한 객체 혹은 페이지들이 반복적으로 액세스될 때에만 선인출 효과를 가지는 문제점이 있다. 이에 반해 제안하는 방법은 항해 응용에서 같은 객체들이 반복적으로 액세스되지 않더라도 같은 애트리뷰트들이 반복적으로 참조되는 경우, 즉, 타입수준 액세스 지역성이 존재하면, 효과적인 선인출을 수행하는 장점이 있다. 객체관계형 DBMS(ORDBMS)의 많은 항해 응용들은 타입수준 액세스 지역성이 있다. 따라서, 제안하는 방법을 ORDBMS에 적용하면 라운드트립의 횟수를 효과적으로 줄일 수 있고 성능을 크게 향상시킬 수 있다. 제안하는 방법의 우수성을 증명하기 위해, ORDBMS 프로토타입에 구현하여 많은 종류의 실험을 수행하였다. 실험결과, 복잡한 구조를 탐색하는 OO7 벤치마크나 실제 GIS 응용에서, 제안하는 선인출 방법은 단순한 요구인출 방법 및 최근의 문맥 기반 선인출 방법과 비교하여 라운드트립 횟수를 수십배에서 수백배까지 줄이고 성능을 수배까지 향상시켰다. 이와 같은 결과로 볼 때, 제안하는 방법은 객체지향 항해 응용의 성능을 크게 향상시키는 결과로서, 상용 ORDBMS에 구현될 수 있는 실용적인 결과라 믿는다.

Abstract Prefetching is an effective method to minimize the number of roundtrips between the client and the server in database management systems. In this paper, we propose new notions of the type-level access pattern and the type-level access locality and develop an efficient prefetching policy based on these notions. The *type-level access pattern* is a sequence of attributes that are referenced in accessing the objects; the *type-level access locality* a phenomenon that regular and repetitive type-level access patterns exist. Existing prefetching methods are based on object-level or page-level access patterns, which consist of object-ids or page-ids of the objects accessed. However, the drawback of these methods is that they work only when exactly the same objects or pages are accessed repeatedly. In contrast, even though the same objects are not accessed repeatedly, our technique effectively prefetches objects if the same attributes are referenced repeatedly, i.e., if there is type-level access locality. Many navigational applications in Object-Relational Database Management Systems(ORDBMSs) have type-level access locality. Therefore, our technique can be employed in ORDBMSs to effectively reduce the number of roundtrips thereby significantly enhancing the performance. We have conducted extensive experiments in a prototype ORDBMS to show the

· 본 연구는 첨단정보기술연구센터를 통하여 한국과학재단의 지원을 받았음.

† 학생회원 : 한국과학기술원 전자전산학과
wshan@mozart.kaist.ac.kr
ysmoon@mozart.kaist.ac.kr

†† 종신회원 : 한국과학기술원 전산학과 교수
kywhang@cs.kaist.ac.kr

논문접수 : 2000년 7월 27일
심사완료 : 2001년 8월 20일

effectiveness of our algorithm. Experimental results using the OO7 benchmark and a real GIS application show that our technique provides orders of magnitude improvements in the roundtrips and several factors of improvements in overall performance over on-demand fetching and context-based prefetching, which is a state-of-the-art prefetching method. These results indicate that our approach provides a new paradigm in prefetching that improves performance of navigational applications significantly and is a practical method that can be implemented in commercial ORDBMSs.

1. 서 론

ORDBMS 응용은 서로 관련 있는 객체들을 참조 애트리뷰트나 컬렉션 애트리뷰트를 사용하여 복합 객체로 모델링한다. 그리고, 항해 응용(navigational application)에서는 이 애트리뷰트 값에 따라 객체를 한번에 하나씩 액세스하며 복합 객체를 항해하는 특징을 가진다. 이런 항해 특징으로 인해, 클라이언트/서버 ORDBMS 환경에서 클라이언트와 서버간의 잦은 라운드트립이 발생하며, 이는 시스템 성능을 저하시키는 주된 원인으로 알려져 있다. C/S ORDBMS 아키텍쳐에서는 클라이언트측에 캐시를 둠으로써 항해 응용에서 발생하는 서버와의 라운드트립 횟수를 줄여 성능을 향상시킨다. 즉, 항해 응용에서 한번 액세스한 객체들을 캐싱하여 동일한 객체들이 다시 요구될 때에는 클라이언트 캐시로부터 액세스 되게 한다.

서버의 객체들을 클라이언트 캐시로 인출하는 방식은 크게 요구인출(on-demand fetch)[4]과 선인출(prefetch) [2, 6, 8, 10, 14, 17]로 분류할 수 있다. 요구인출 방법은 항해과정에서 액세스되는 객체를 요구시마다 서버에서 인출하는 방법이다. 이 요구인출 방법은 꼭 필요한 객체만 인출한다는 장점이 있으나, 객체의 인출시마다 서버와의 라운드트립이 발생하므로 그 횟수가 많은 단점이 있다. 이에 반해, 선인출 방법은 향후 액세스될 객체들을 서버에서 미리 인출하는 방법이다. 선인출 방법은 미리 인출된 객체들이 응용에서 실제로 액세스되는 경우 서버와의 라운드트립 횟수가 줄고 성능이 향상되는 장점이 있다. 그러나 선인출된 객체들이 응용에서 실제로 액세스되지 않으면 불필요한 객체들의 선인출 비용으로 인해 시스템의 성능이 오히려 저하될 수 있다. 따라서, 효과적인 선인출 방법을 구현하기 위해서는 향후 액세스될 객체들의 액세스 패턴을 정확하게 예측하는 것이 매우 중요하다.

객체지향 응용에서의 일반적인 항해 액세스 패턴은 루트 객체들을 구한 후, 루트 객체들로부터 항해를 시작하여 서로 관련된 객체들의 그래프를 탐색하는 패턴이다[3, 12]. 예를 들어, 봉급이 $100,000 이상인 교수의 주소와 차를 구하는 항해 응용을 생각해 보자. 응용에서는 먼저 질의를 통해 항해를 시작하는 루트 객체들에 대한 참조(reference)를 구한다. 즉, 질의 select * from Professors where salary 100,000을 통해서 루트 객체인 교수 객체들에 대한 참조를 구한다. 다음으로, 각 루트 객체들을 액세스하면서 주소와 차에 대한 정보를 구하는 항해를 수행한다.

객체지향 응용의 액세스 패턴은 항해시에 액세스되는 객체의 식별자들의 패턴이 아니라 액세스하기 위해 사용된 애트리뷰트들의 패턴으로서 표현될 수 있다. 그림 1은 앞서의 교수의 차와 주소를 구하는 예에서 액세스되는 객체들의 참조 순서를 나타낸다. 즉, 항해시 액세스되는 객체들의 순서는 o_1, o_2, o_3, o_4, …, o_{18}이다. 여기서, 루트 객체로부터 해당 객체를 액세스 하기 위해 사용된 애트리뷰트들을 점(.)을 사용하여 순서대로 연결한 시퀀스를 **타입수준 패스**(*type-level path*)라 부른다. 예를 들어, 객체 o_4의 타입수준 패스는 owns.company이고, 객체 o_6의 타입수준 패스는 owns.drivetrain.engine이다. 그림 1에서, 객체들의 참조 순서인 o_1, o_2, o_3, o_4, …로부터는 어떠한 반복적인 패턴도 나타나지 않는 반면, 객체들을 액세스하기 위해 사용된 타입수준 패스들인 address, owns, owns.company, owns.drivetrain, owns.drivetrain.engine은 반복적으로 나타남을 주목할 수 있다. 본 논문에서는 이렇게 객체를 액세스하기 위해 사용된 타입수준 패스들이 항해중에 반복되어 나타나는 현상을 **타입수준 액세스 지역성**(*type-level access locality*)이라 정의한다. 그리고, 항해에 사용된 타입수준 패스들의 패턴을 **타입수준 액세스 패턴**(*type-level access pattern*)이라 정의한다. 따라서 타입수준 액세스 지역성을 가지는 타입수준 액세스 패턴을 항해시에 포착(capture)할 수 있다면, 이를 사용하여 향후 액세스될 객체들을 선인출 할 수 있게 된다.

액세스 패턴에 기반한 기존의 선인출 방법은 사용하는 액세스 패턴의 종류에 따라 객체수준 액세스 패턴 기반 선인출 방법[14]과 페이지수준 액세스 패턴 기반 선인출 방법[6]이 있다. 그러나 이 방법들은 동일한 객체 혹은 페이지들이 반복해서 인출될 때에만 선인출 효

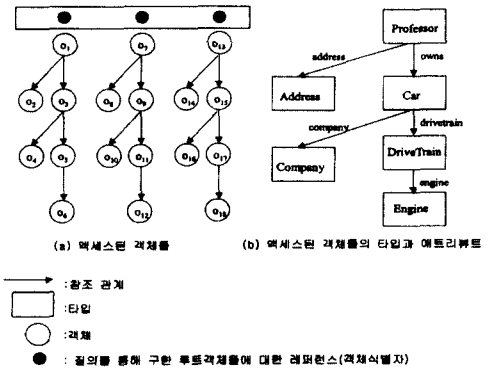


그림 1 봉급이 $100,000 이상인 교수의 주소와 차를 구하는 항해 응용의 예

과를 가지는 문제점을 가진다[2]. 즉, 이 방법들은 객체지향 응용에서 많이 발생하는 타입수준 액세스 지역성이 선인출 전략에 반영되지 않은 문제점이 있다.

본 논문에서는 우선 타입수준 액세스 지역성을 가지는 타입수준 액세스 패턴들을 분석한다. 그리고, 항해 과정에서 발생하는 객체참조로부터 타입수준 액세스 패턴을 포착하는 방법과 포착된 정보를 활용하여 선인출하는 방법을 제안한다. 본 논문에서는 타입수준 액세스 패턴을 포착하기 위해서, 객체의 참조 시 어떠한 애트리뷰트들을 사용하여 객체를 참조하는지에 대한 정보를 사용한다. 예를 들어, 그림 1의 예에서 o_1에서 o_6까지 항해하면, 타입수준 패스들의 리스트인 [address, owns, owns.company, owns.drivetrain, owns.drivetrain.engine]이 포착되므로, 객체 o_7 액세스 시에 이 정보를 활용하여 향후 액세스될 객체들인 o_8 ~ o_{18}을 선인출 할 수 있다.

본 논문의 공헌은 다음과 같다. 첫째, 타입수준 액세스 패턴 지역성과 타입수준 액세스 패턴의 개념을 제안한다. 둘째, 객체 참조 시퀀스들로부터 타입수준 액세스 패턴을 포착하고, 포착된 패턴을 사용하여 향후 액세스될 객체들을 선인출하는 알고리즘을 제안한다. 셋째, 제안한 선인출 방법의 유용성을 보이기 위해 성능 평가를 수행하여, 제안한 방법이 기존의 요구인출 방법 및 문맥기반 선인출 방법[2]과 비교하여 성능을 크게 향상시킴을 보인다.

본 논문의 구성은 다음과 같다. 제 2절에서는 선인출에 관한 기존연구를 설명하고 장단점을 논한다. 제 3절에서는 타입수준 액세스 패턴을 설명하고, 제 4절에서는 타입수준 액세스 패턴을 포착하고 선인출하는 알고리즘을 제안한다. 제 5절에서는 성능평가 결과에 대해 설명하며, 제6절에서는 본 논문의 결론을 내린다.

2. 관련 연구

객체지향 응용을 위한 기존의 선인출 방법은 선인출 대상을 선정하는 방식에 따라 크게 1) 페이지기반 선인출 방법, 2) 객체수준/페이지수준 액세스패턴에 기반한 선인출 방법, 3) 사용자 힌트 기반 선인출 방법,그리고 4) 문맥 기반 선인출 방법의 네 가지로 분류할 수 있다.

첫째, 페이지기반 선인출 방법은 요청된 객체를 서버에서 가져올 때, 그 객체가 속한 페이지에 있는 다른 객체들을 함께 인출하는 방법이다[8, 11]. 이 방법은 페이지 내의 객체들이 연속해서 액세스 되는 경우에 좋은 성능을 보인다. 그러나, 페이지내의 객체들이 연속적으로 액세스되지 않으면 선인출 효과가 없게 된다. 이 방법의 유효성은 전적으로 객체들의 클러스터링 방식에 의존하게 되므로, 응용에서 클러스터링 순서와 다른 방식으로 객체들을 액세스하는 경우에 선인출 효과를 보지 못하는 단점을 가진다.

둘째, 객체수준/페이지수준 액세스 패턴에 기반한 선인출 방법은 최근에 발생한 객체/페이지 참조에 기반하여 향후 객체/페이지 액세스 패턴을 예측하는 방법이다[6, 14]. Palmer와 Zdonik[14]은 학습 알고리즘을 사용하여 객체 참조로부터 반복되는 객체 참조 패턴을 찾은 후 이를 바탕으로 객체들을 선인출하는 방법을 제안하였다. Curewitz 등[6]은 압축 알고리즘을 사용하여 페이지 참조로부터 반복되는 페이지 액세스 패턴을 찾은 후 이를 바탕으로 페이지들을 선인출을 하는 방법을 제안하였다. 그러나 이 방법들은 동일한 객체와 페이지들이 반복해서 액세스되지 않으면 선인출 효과를 보지 못하는 단점을 가진다[2].

셋째, 사용자 힌트 기반 선인출 방법은 사용자 힌트를 사용하여 선인출할 객체들을 결정하는 방법이다[7, 12]. Chang과 Katz[7]는 "응용의 주요 액세스는 컨피규레이션 관계를 통하여 이루어진다"와 같은 사용자 힌트를 사용하여 필요한 객체들을 선인출하는 방법을 제안하였다. 또한, 상용 ORDBMS에서도 이와 유사한 방법으로 선인출하는 전략을 취한다[12]. 그러나, 이 방법들은 사용자가 힌트를 제시해야 하는 어려움이 있으며, 현재의 흐름인 자동튜닝 DBMS로의 추세에 맞지 않는다.

넷째, 최근에 발표된 문맥 기반 선인출 방법은 객체를 서버에서 가져올 때, 그 객체의 구조 문맥(structure context)이 가리키는 구조의 모든 객체들도 함께 인출하는 방법이다[2]. 구조의 예로는 질의 결과나 컬렉션 등이 있으며, 구조 문맥은 각 객체가 어떤 구조에서 인

출되었는지에 대한 정보이다. 그림 2는 문맥 기반 선인출 방식에서 선인출하는 객체들의 예를 나타내고 있다. 그림에서 o_2의 구조 문맥은 'o_1의 애트리뷰트 A의 값'이다. 따라서, o_2를 액세스하면, o_2의 구조 문맥의 모든 객체들인 $o_2 \sim o_n$을 선인출한다. 즉, 컬렉션의 한 원소가 가리키는 객체(o_2)를 인출할 때 다른 나머지 원소들이 가리키는 객체들($o_3 \sim o_n$)도 함께 선인출 하는 방법이다. 이 방법은 항해 응용이 객체 계층 구조를 넓이 우선 탐색(bread first search: BFS)방식으로 항해하는 경우에 효과적인 선인출 방법으로, 상용 DBMS에 구현되어 최대 70까지의 성능 향상을 보였다[2].

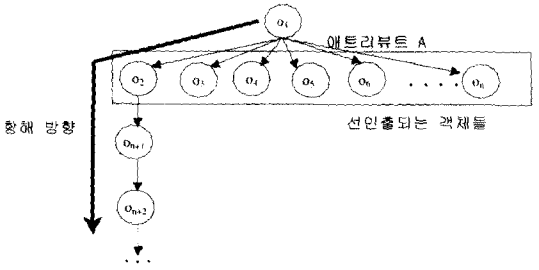


그림 2 문맥 기반 선인출 방법의 예

그러나, 문맥 기반 선인출 방법은 깊이 우선 탐색(depth first search:DFS) 방식으로 객체 계층 구조를 탐색하는 경우 선인출된 객체들이 액세스도 되기전에 캐시에서 교체되는 문제점을 가진다[2]. 예를 들어, 그림 2와 같이 굵은 화살표 방향으로 DFS 방식의 항해가 이루어진다고 하자. 이때, o_2와 연결된 객체들(예: o_{n-1}, o_{n-2})의 수가 매우 많다면, 선인출된 $o_3 \sim o_n$은 액세스도 되기 전에 캐시에서 교체되는 문제점이 발생하게 된다. 이러한 문제점은 질의 결과의 경우와 같이 구조 문맥의 크기가 매우 큰 경우에는 더욱 심각하게 된다. 문맥 기반 선인출 방법의 또 다른 단점은 컬렉션이 아닌 참조 타입(non-collection reference type)에 의해 연결된 객체들은 선인출하지 못한다는 것이다. 예를 들어, 그림 1의 교수의 주소와 차를 구하는 항해 응용의 예에서 문맥기반 선인출 방법은 o_1을 액세스할 때, o_7과 o_{13}만을 선인출할 뿐, 객체 o_1로부터 컬렉션이 아닌 참조 타입의 애트리뷰트에 의해서 직간접으로 연결된 $o_2 \sim o_6$, $o_8 \sim o_{12}$, $o_{14} \sim o_{18}$은 선인출하지 못하는 문제점이 있다.

이 외에도, 서버에서 객체에 대한 시맨틱스(클래스, 릴레이션쉽 등)를 해석하지 못하는 페이지기반 클라이언트/서버 객체지향 DBMS 구조에서 선인출 기법을 다루는 연구가 있었다[1]. 이 방법은 휴리스틱을 사용하여 어떤 페이지내의 많은 객체들이 앞으로 액세스될 것으로 예상되면, 그 페이지의 모든 객체들을 객체 캐시로 선인출한다. 그러나, 객체 관계형 DBMS는 서버에서 객체들에 대한 시맨틱스를 해석할 수 있는 장점이 있으므로 이를 활용하는 효과적인 선인출 방법이 필요하다.

3. 타입수준 액세스 패턴

본 장에서는 객체지향 항해 응용에서 자주 발생하는 타입수준 액세스 패턴을 정의하고, 그 종류를 분석한다. 그림 3은 대학 데이타베이스 스키마를 나타내며, 본 논문의 예제들은 이 스키마를 사용한다.

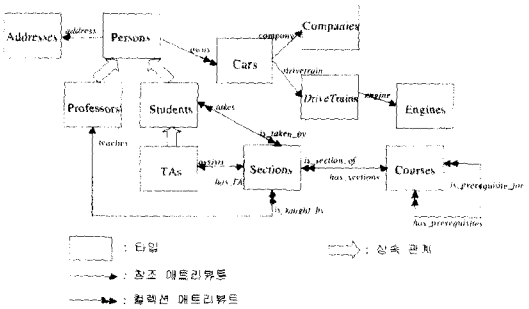


그림 3 대학 데이타베이스 스키마

3.1 용어 정의

본 절에서는 본 논문에서 사용하는 몇 가지 용어와 기호를 정의한다.

항해루트집합(*set of navigational root objects*)이란 항해 응용에서 항해를 시작하기 위해 구한 루트 객체들의 집합으로, 이를 Ω로 표시한다. 예를 들어, 그림 1에서 항해루트집합은 $\{o_1, o_7, o_{13}\}$이다(편의상 이를 Ω_1이라 하자). ORDBMS에서는 항해루트집합을 구하는 방법으로 질의기능을 제공하며[12, 19], 하나의 응용에서는 여러 번의 질의를 통해 여러 개의 항해루트집합을 구할 수 있다.

객체 o의 **타입수준 패스**(*type-level path*)란 루트 객체로부터 객체 o를 액세스 하기 위해 사용된 애트리뷰트들을 점(.)을 사용하여 순서대로 연결한 시퀀스이다. 객체의 타입수준 패스를 표현하기 위해, 항해루트집합은 가상의 객체에 대한 가상의 컬렉션 애트리뷰트의 값이라고 가정하고, 항해루트집합의 객체를 액세스하는 방법도 컬렉션 애트리뷰트 값을 액세스 하는 방법과 동일하게 취급한다. 그리고 본 논문에서는 이 가상의 컬렉션 애트리뷰트 이름으로 a_0를 사용한다. 예를 들어, 그림 1

에서 o_1의 타입수준 패스는 a_0, o_2의 타입수준 패스는 a_0.address, o_3의 타입수준 패스는 a_0.owns, 그리고 o_4의 타입수준 패스는 a_0.owns.company이다. 타입수준 패스 상에서, 애트리뷰트 a_i 바로 다음에 a_j가 나오면, a_i는 a_j의 **부모 애트리뷰트**(*parent attribute*)라 정의하고, a_j는 a_i의 **자식 애트리뷰트**(*child attribute*)라 정의한다. 예를 들어, o_4의 타입수준 패스 a_0.owns.company에서 company의 부모 애트리뷰트는 owns이고 owns의 자식 애트리뷰트는 company가 된다.

항해루트집합 Ω로부터 시작한 항해의 **객체 참조 스트링**(*object reference string*)이란 Ω로부터 참조관계에 의해 직간접적으로 연결되어 참조된객체들의 시퀀스이다. 예를들어, 그림 1의 항해루트집합 Ω_1로부터 시작한 항해의 객체참조스트링은 o_1, o_2, o_3,…, o_{18}이다. 항해루트집합 로부터 시작한 항해의 **타입수준 패스 스트링**(*type-level path string*)이란 Ω로부터 시작한 항해의 객체참조스트링의 각 객체를 타입수준 패스로 대체한 스트링이다. 예를들어, 그림 1의 항해루트집합 Ω_1로부터 시작한 항해의 타입수준 패스 스트링은 o_1, o_2, o_3, o_4, o_5,…, o_{17}, o_{18}의 각 객체를 타입수준 패스로 대체한 a_0, a_0.address, a_0.owns, a_0.owns.company, a_0.owns.drivetrain,, a_0.owns.drivetrain, a_0.owns.drivetrain. engine이다.

항해루트집합 Ω로부터 시작한 항해의 **타입수준 패스 리스트**(*type-level path list*)란 타입수준 패스 스트링에서 중복을 제거한 타입수준 패스들의 리스트이다. 예를 들어, 그림 1의 항해루트집합 Ω_1로부터 시작한 항해의 타입수준 패스 리스트는 [a_0,a_0.address, a_0.owns, a_0.owns.company, a_0.owns.drivetrain, a_0.owns.drivetrain.engine]이다. 본 논문에서는 타입수준 패스 리스트를 그림 1 (b)와 같이 타입을 노드로 하고 애트리뷰트를 에지로 하는 그래프 형태로 표현하되, 리스트임을 표현하기 위해 추가되는 에지에 순서대로 번호를 부여한다. 그리고, 이 그래프를 **타입수준 패스 그래프**(*type-level path graph*)라 부르며, 그 예는 제 4.1절에서 제시한다.

본 논문에서는 지금까지 정의한 용어들에 대해서 표 1의 표기법을 사용한다.

표 1 주요 표기법

기 호	정의/의미
o	객체
o_i	항해에서 i번째로 액세스한 객체
TLP(o)	객체 o의 타입수준 패스(type-level path)
Ω	항해루트집합
SNRO(o)	객체 o의 항해루트집합(set of navigational root objects)
CTLP(Ω)	Ω로부터 시작한 항해에서 현재까지 포착된 타입수준 패스들의 리스트(list of captured type- level paths)

3.2 타입수준 액세스 지역성

객체지향 응용은 서로 관련 있는 객체들을 그래프 형태의 복합객체로 모델링한다[12]. ORDBMS에서는 참조와 컬렉션 타입의 애트리뷰트들을 사용하여 복합객체들을 모델링하며, 항해 응용에서는 이러한 애트리뷰트 정보를 사용하여 복합객체를 항해한다. 이와 같은 ORDBMS 항해 응용에서는 같은 타입들의 복합객체들을 액세스 하는 경우 액세스되는 객체들에서는 규칙적이고 반복적인 패턴이 발생하지 않더라도, 액세스되는 객체들의 타입수준 패스들에서는 규칙적이고 반복적인 패턴이 발생하게 된다. 이런 현상인 타입수준 지역성은 정의 1과 같이 정의할 수 있다.

정의 1 타입수준 액세스 지역성은 항해의 타입수준 패스 스트링에 나타난 타입수준 패스들이 규칙적이고 반복적인 패턴을 보이는 현상이다.

본 논문에서는 타입정보를 사용하여 표현한 액세스 패턴을 객체수준의 액세스 패턴과는 구별되게 타입수준 액세스 패턴이라 하며 다음과 같이 정의한다.

정의 2 타입수준 액세스 패턴은 액세스되는 객체들의 타입수준 패스를 사용하여 표현한 액세스 패턴이다.

제 3.3절에서는 타입수준 액세스 지역성을 가지는 중요한 타입수준 액세스 패턴들을 설명한다.

3.3 타입수준 액세스 패턴

객체지향 항해 응용에서 액세스하는 복합객체들은 참조 및 컬렉션 타입으로 모델링되며[17], 많은 경우 재귀적인 구조를 가진다[15]. 본 절에서는 컬렉션 타입으로 모델링된 복합객체를 항해하는 경우 발생하는 반복 패턴(iterative pattern)과 재귀적 구조를 가진 복합객체를 항해하는 경우 발생하는 재귀 패턴(recursive pattern)에 대해 설명한다.

본 논문에서는 타입수준 액세스 패턴을 표현하는 방법으로 생성 규칙(production rules)[18]을 사용한다. 그리고 타입수준 패스의 간결한 표현을 위해 식 (1)과 같이 정의되는 연산자 $\odot$을 문맥 자유 문법에 포함시켜 사용한다. 식 (1)에서 $P_i(0\le i\le k)$는 타입수준 패스의 서브 패스이며, 콤마(,)는 타입수준 패스들을 구별하기 위해서 사용된다.

$$P_0\odot(P_1,P_2,\cdots,P_k)=P_0.P_1,P_0.P_2,\cdots,P_0.P_k \quad (1)$$

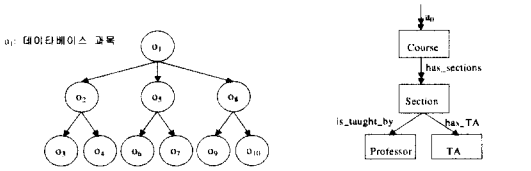
3.3.1 반복 패턴

ORDBMS에서는 항해루트집합의 객체들을 액세스하

거나, 컬렉션 애트리뷰트의 값을 액세스하는 방법으로서 반복 기능(iteration method)을 제공한다. 반복이란 컬렉션의 원소가 가리키는 객체들을 하나씩 순차적으로 액세스 하는 방법으로, 이 과정에서 액세스되는 객체들은 동일한 타입수준 패스를 가진다. 이와 같이 반복적으로 나타나는 동일한 타입수준 패스들이 구성하는 타입수준 액세스 패턴을 정의 3에서와 같이 반복 패턴이라 부른다.

정의 3 반복 패턴은 타입수준 패스 스트링에서 컬렉션 애트리뷰트로 인해 동일한 타입수준 패스가 반복되어 나타나는 타입수준 액세스 패턴이다.

그림 4는 질의 "select * from Courses where name='데이타베이스' "로 구한 루트 객체인 데이타베이스 과목에 대해 이 과목의 섹션들과 각 섹션들을 가르치는 교수와 조교를 구하는 항해 응용에서 반복 패턴이 발생하는 예를 나타낸다. 그림에서는 데이타베이스 과목 객체 o_1의 컬렉션 애트리뷰트 has_sections의 값을 순회하며 한번에 하나씩 섹션 객체 o_2, o_5, o_8을 액세스한다. 그리고, 각 섹션에 대해 애트리뷰트 is_taught_by와 has_TA 값을 통해 각 섹션을 가르치는 교수와 조교를 구한다. 여기에서, o_1이후의 객체 참조 스트링은 o_2, o_3, o_4, ···, o_{10}이며, 이에 해당하는 타입수준 패스 참조 스트링은 a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA, a_0.has_sections, a_0.has_sections.is_taught_by, ···, a_0.has_sections.has_TA가 된다. 결국, a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA가 타입수준 패스 스트링에서 반복되어 나타나고, 이에 대한 반복 패턴은 (a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA)$^+$으로 표현할 수 있으며, 루트 객체가 하나 이상 나타날 수 있으므로 이를 반영한 반복 패턴은 (a_0, (a_0.has_sections, a_0.has_sections.is_taught_by, a_0.has_sections.has_TA)$^+$)$^+$으로 표현할 수 있다.[1)]



(a) 액세스된 객체들 (b) 액세스된 객체들의 타입과 애트리뷰트

그림 4 반복 패턴이 발생하는 예

1) A+는 A가 1번 이상 반복되어 나타남을 의미한다.

반복 패턴은 여러 레벨로 중첩된 형태를 가질 수 있다. 왜냐하면, 반복 패턴은 하나의 컬레션 애트리뷰트의 원소들이 가리키는 객체들을 액세스할 때마다 발생하고, 컬렉션 애트리뷰트는 타입수준 패스상의 여러 곳에서 나타날 수 있기 때문이다. 항해 응용이 반복 기능을사용하여 항해루트집합으로부터 애트리뷰트들의 패스 $a_0.a_1.a_2..a_n$을 따라 항해하였고, 이들 n개의 애트리뷰트중 j($j \leq n$)개가 컬렉션 애트리뷰트, a_{c1}, a_{c2}, ···, a_{cj} ($c_1 < c_2 \cdots < c_j$)라 하자. 그러면, 타입수준 패스 스트링으로부터 각 컬렉션 애트리뷰트에 의한 반복 패턴인($a_0..a_{c1}$, ···)$^+$, ($a_0..a_{c2}$, ···)$^+$, ···, ($a_0..a_{cj}$, ···)$^+$가 중첩된 (··· ($a_0..a_{c1}$, ··· ($a_0..a_{c2}$, ··· ($a_0..a_{cj}$, ···)$^+$ ···)$^+$ ···)$^+$ ···)형태의 반복 패턴이 발생한다. 즉, 항해 응용이 애트리뷰트들의 패스 $a_0.a_1..a_n$을 따라 항해한다면, 이에 대한 반복 패턴을 표현하는 생성 규칙은 식 (2)와 같으며, 이를 간단히 하면 식(3)과 같다.

$$\left.\begin{array}{l} A_0 \rightarrow (a_0, a_0 \odot A_1)^{iter(a0)} \\ A_1 \rightarrow (a_1, a_1 \odot A_2)^{iter(a1)} \\ \cdots \\ A_k \rightarrow (a_k, a_k \odot A_{k-1})^{iter(ak)} \\ \cdots \\ A_n \rightarrow (a_n)^{iter(an)} \end{array}\right\} \quad (2)$$

$$\left.\begin{array}{ll} A_k \rightarrow (a_k,\ a_k \odot A_{k+1})^{iter(ak)} & ,\ 0<k\leq n-1 \\ A_k \rightarrow (a_k)^{iter(ak)} & ,\ k=n \end{array}\right\} \quad (3)$$

위 식에서 시작 심볼은 A_0이고, iter(a_k)는 a_k가 컬렉션 애트리뷰트이면 +이고, 그렇지 않으면 1이다.

그런데, 타입수준 패스 상의 각 애트리뷰트는 두 개 이상의 자식 애트리뷰트를 가질 수 있으므로, 식 (3)의 반복 패턴은 이 경우를 포함하도록 확장되어야 한다. 즉, 항해 응용이 타입수준 패스 그래프를 따라 항해하는 경우에 발생하는 반복 패턴을 표현할 수 있어야 한다. 본 절에서는 타입수준 패스 그래프에서도 사이클이 발생하지 않는 경우인 타입수준 패스 트리에 대해서만 다루며, 사이클이 발생하는 경우는 제 3.3.2절의 재귀 패턴에서 설명한다. 항해 응용이 타입수준 패스 트리를 따라 항해하는 경우, 이때 발생하는 반복 패턴은 위 식 (3)이 확장된 다음의 두 가지 생성규칙을 사용하여 표현할 수 있다. 즉, 타입수준 패스 트리의 각 노드들을 다음 두 가지 생성규칙으로 매핑함으로써 반복 패턴을 문맥 자유 문법으로 표현할 수 있다.

규칙 1 항해 응용이 애트리뷰트 a를 통해 객체(o_a)를 액세스한 후, a의 자식 애트리뷰트 b_1, b_2, ···, b_j($j \geq 1$)를 차례대로 따라가며 객체들(o_{b1}, o_{b2}, ..., o_{bj})을 액세스한다고 하자. 이때, 타입수준 패스 트리에서 a를 통해 연

결된 노드가 A이고, b_1, b_2, ..., b_j와 연결된 노드가 B_1, B_2, ..., B_j라 하자. 그러면, 노드 A에 대응되는 생성 규칙은 다음과 같이 표현된다.

$$A \rightarrow (a, a\odot(B_1, B_2, ..., B_j))^{iter(a)} \quad (4)$$

식 (4)에서 단독으로 있는 a는 객체 o_a를 액세스 했을 때 나타나는 타입수준 패스를 표현하기 위해 사용되며, $a\odot(B_1, B_2, ..., B_j)$에서의 a는 객체 o_{b1}, o_{b2}, ..., o_{bj}들이 a를 통해 연결되므로 이들의 타입수준 패스에 a가 나타나도록 하기 위해 사용된다.

규칙 2 항해 응용이 애트리뷰트 a를 통해 객체를 액세스하고, a의 자식 애트리뷰트가 없다고 하자. 이때, 타입수준 패스 트리에서 a를 통해 연결된 노드를 A라 하자. 그러면 노드 A에 대응되는 생성 규칙은 다음과 같이 표현된다.

$$A \rightarrow (a)^{iter(a)} \quad (5)$$

애트리뷰트 a가 자식 애트리뷰트가 없는 경우는 규칙 1의 식 (4)에서 $a\odot(B_1, B_2, ..., B_j)$이 필요 없으므로 노드 A에 대응되는 생성 규칙은 식 (5)와 같이 간단히 표현된다.

규칙 1과 2에 따르면, 그림 4의 예에서 발생하는 반복패턴을 나타내는 생성 규칙은 식 (6)과 같이 표현할 수 있다.

$$\left.\begin{array}{l} A_0 \rightarrow (a_0,\ a_0\odot A_1)^* \\ A_1 \rightarrow (\text{has\_sections}, \text{has\_sections}\odot(B_1,\ B_2))^* \\ B_1 \rightarrow (\text{is\_taught\_by}) \\ B_2 \rightarrow (\text{has\_TA}) \end{array}\right\} \quad (6)$$

식 (6)에서 시작 심볼은 A_0이다.

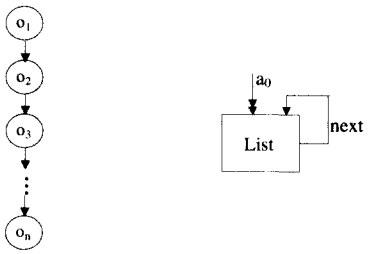
문맥 기반 선인출 방법[2]은 반복 패턴에서 하나의 레벨에 대해서만 초점을 맞추어 선인출 하는 방법으로 볼 수 있다. 즉, 컬렉션 애트리뷰트 a_i의 원소가 가리키는 객체를 액세스할 때, a_i의 모든 원소들이 가리키는 객체를 선인출할 뿐, a_i 다음 레벨의 다른 애트리뷰트들과 연결된 객체는 선인출 대상으로 삼지 않는다. 만일 어떠한 패스를 따라가는지의 힌트가 있다면, a_i 다음 레벨의 다른 애트리뷰트들과 연결된 객체들도 선인출 할 수 있게 되는데, 참고문헌 [2]에서는 이런 힌트를 자동적으로 제시하는 방법이 향후 연구로 필요하다고 지적하고 있다. 본 논문에서 제안하는 방법은 어떠한 패스들을 따라가며 액세스하는지를 항해중에 자동적으로 포착하고 여러 레벨로 중첩된 반복 패턴을 고려한 선인출을 수행한다.

3.3.2 재귀 패턴

객체지향 항해 응용에서는 재귀적 구조(recursive structure)를 가지는 객체들을 항해하기 위해 재귀 기능(recursion method)을 사용한다. 재귀적 구조를 가지는 객체들이란 스키마에서 사이클을 구성하는 타입들의 객체들이다. 따라서 이러한 재귀적 구조를 가지는 객체들을 액세스하는 경우, 액세스되는 객체의 타입수준 패스 내에는 사이클이 발생한다. 이와 같이 사이클을 포함하는 타입수준 패스들에서 나타나는 타입수준 액세스 패턴을 재귀 패턴이라 정의한다.

정의 4 재귀 패턴은 재귀적 구조를 항해함으로 인해 타입수준 패스 스트링에서 사이클을 포함하는 타입수준 패스들이 발생하는 타입수준 액세스 패턴이다.

그림 5는 연결 리스트(linked list)를 액세스하는 항해 응용에서 재귀 패턴이 발생하는 예를 나타낸다. 그림에서는 루트 객체인 o_1을 액세스한 후, next 애트리뷰트의 값을 사용하여 o_1과 연결된 객체들을 재귀적으로 액세스한다. 따라서, o_1 이후의 객체 참조 스트링은 o_2, o_3, ⋯, o_n이며, 타입수준 패스 참조 스트링은 a_0.next, a_0.next.next, ⋯, a_0.(next)$^{n-2}$.next가 된다. 결국, 재귀 타입을 구성하는 next 애트리뷰트가 타입수준 패스내에서 반복되어 나타나는 재귀 패턴이 발생한다.



(a) 액세스된 객체들 (b) 액세스된 객체들의 타입과 애트리뷰트

그림 5 재귀 패턴이 발생하는 예

재귀 패턴에서 반복적으로 나타나는 애트리뷰트의 개수는 스키마에서 사이클을 구성하는 애트리뷰트의 개수와 동일하다. 항해 응용이 애트리뷰트들의 패스 ..a_i..a_j를 따라 항해하였고, 패스 a_i..a_j가 사이클을 구성한다면, 타입수준 패스 스트링에는 이 사이클이 반복되는 타입수준 패스들이 나타나게 된다. 즉, ⋯, ..a_i, ..$a_i.a_{i+1}$, ⋯, ..a_i..a_j, ..a_i..$a_j.a_i$, ..a_i..$a_j.a_i.a_{i+1}$, ⋯, ..$(a_i..a_j)^2$, ..$(a_i..a_j)^2.a_i$, ⋯의 타입수준 패스들이 나타난다. 이 같은 형태의 재귀 패턴은 다음의 생성 규칙을 사용하여 표현할 수 있다.

규칙 3 항해 응용이 타입수준 패스 ..a_i..a_j($i \le j$)를 따라 항해하였고, 패스 a_i..a_j가 사이클을 구성한다고 하자. 이때, 타입수준 패스 그래프에서 a_k를 통해 연결된 노드

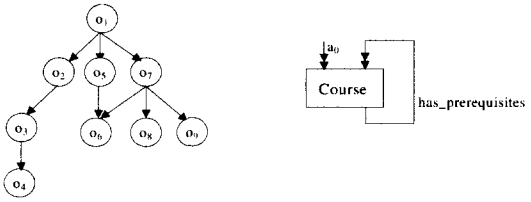
를 A_k라 하자. 그러면, 노드 A_k에 대응되는 생성 규칙은 다음과 같이 정의된다.

$$\left.\begin{array}{l} A_k \rightarrow (a_k, a_k \odot A_{k+1})^{iter(ak)}\ ,\ i \le k < j \\ A_k \rightarrow (a_k, a_k \odot A_i)^{iter(ak)} \mid (a_k)^{iter(ak)}\ ,k=j \end{array}\right\} \quad (7)$$

스키마에서 사이클을 구성하는 애트리뷰트들 중에 컬렉션 애트리뷰트가 존재하면, 타입수준 액세스 패턴은 반복 패턴과 재귀 패턴이 함께 존재하는 복잡한 형태가 된다. 예를 들어, 그림 6은 각 과목의 선수과목을 구하는 항해 응용에서 액세스하는 객체들을 보여주고 있다. 항해에 사용된 has_prerequisites 애트리뷰트는 컬렉션 타입이므로 타입수준 패스 스트링에서는 반복 패턴이 나타나며, has_prerequisites 애트리뷰트는 사이클을 구성하므로 재귀 패턴이 나타난다. 따라서, 이와 같은 타입수준 액세스 패턴을 표현하는 생성 규칙은 식 (8)과 같이 반복 패턴과 재귀 패턴이 함께 나타나는 형태가 된다.

$$\left.\begin{array}{l} A_0 \rightarrow (a_0,\ a_0 \odot A_1)^* \\ A_1 \rightarrow (\text{has\_prerequisites}, \text{has\_prerequisites} \odot A_1)^* \\ \quad \mid (\text{has\_prerequisites})^* \end{array}\right\} \quad (8)$$

지금까지 설명한 재귀는 한 개의 사이클만을 나타내는 경우인데, 이론적으로 여러 개의 사이클을 함께 활용하는 복잡한 형태의 재귀가 가능하다. 그러나, CAD의 부품 계층구조나 GIS의 공간 객체 스키마 등에서와 같은 실제 응용에서 사용되는 재귀는 비교적 간단한 형태의 재귀이므로 본 논문에서는 한 개의 사이클을 사용하는 재귀만을 다루도록 한다. 복잡한 형태의 재귀 패턴의 분석과 포착은 향후 연구 토픽으로 미룬다.



(a) 액세스된 객체들 (b) 액세스된 객체들의 타입과 애트리뷰트

그림 6 반복 패턴과 재귀 패턴이 함께 나타나는 예

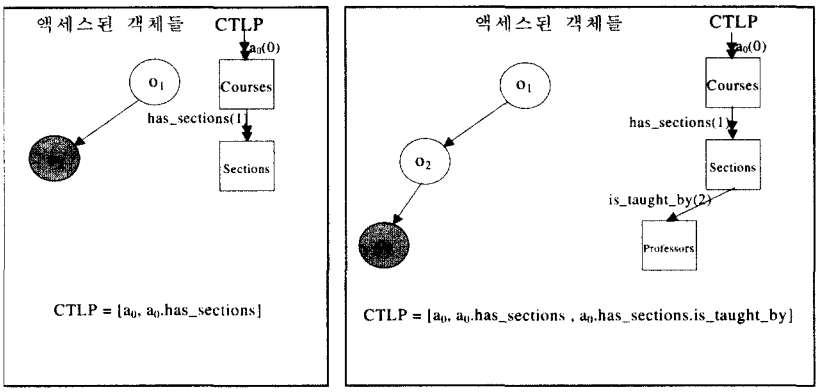
4. 타입수준 액세스 패턴의 포착과 선인출

4.1 개요

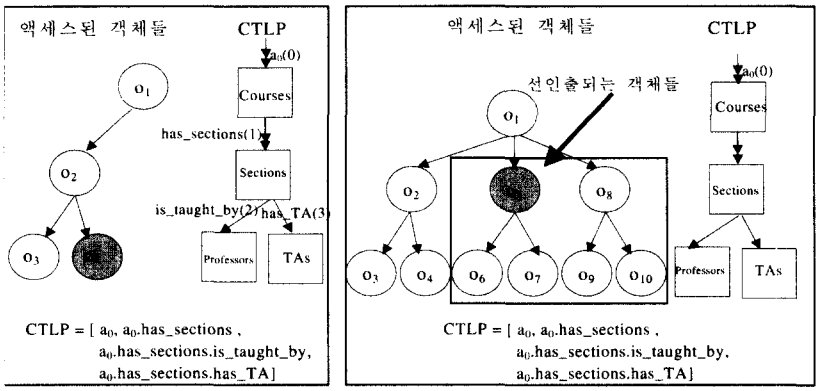
제안하는 선인출 알고리즘은 각 객체를 참조할 때마다 해당 객체의 타입수준 패스(TLP)를 그때까지 발생한 타입수준 패스들의 리스트인 CTLP에 삽입하고, CTLP의 타입수준 패스들을 사용하여 반복 패턴과 재귀 패턴을 포착하며, 포착된 패턴에 따라 선인출을 수행한다. 본 절에서는 이 방법의 개념을 설명하고, 제 4.2절에서는 알고리즘을 제시한다. 그리고, 제 4.3절에서는 제안한 선인출 기법을 ORDBMS에 구현하기 위해 발생할 수 있는 구현 이슈에 대해서 설명한다.

먼저, 컬렉션 애트리뷰트에 의해 발생하는 반복 패턴의 포착과 선인출에 대해 설명한다. 항해 응용에서 애트리뷰트들의 패스 $a_0..a_i..$를 따라 항해하였고, a_i가 컬렉션 애트리뷰트라 하자. 그러면, 애트리뷰트 a_i와 관련된 반복 패턴은 규칙 1, 2에 의해 $(a_0..a_i,\ a_0..a_i \odot A_{i+1})^*$의 형태가 되며, 항해 과정에서는 $a_0..a_i$, …, $a_0..a_i$, 의 타입수준 패스들이 나타난다. 여기에서 A_{i+1}은 항해에서 사용된 a_i의 자식 애트리뷰트들에 의해 결정된다. 제안하는 알고리즘에서는 타입수준 패스 $a_0..a_i$가 처음 나타날 때, 이 패스를 타입수준 패스 리스트에 삽입한다. 그리고, 다음에 동일한 타입수준 패스인 $a_0..a_i$가 나타나면 컬렉션 애트리뷰트 a_i에 의한 반복이 발생함을 인식하여, $A_i \rightarrow (a_i, a_i \odot A_{i+1})^*$형태의 생성 규칙이 적용된 반복 패턴 $(a_0..a_i, a_0..a_i \odot A_{i+1})^*$을 타입수준 패스 리스트로부터 포착한다. 다음으로, 이 반복 패턴에 의한 타입수준 패스들을 따라가며 객체들을 선인출하는 것이다. 이 과정을 요약하면, 컬렉션 애트리뷰트에 의한 첫번째 반복에서는 어떠한 패스들을 따라 항해하였는지를 저장하고, 두번째 반복에서는 반복 패턴이 포착되어 반복 패턴에 의해 생겨날 수 있는 타입수준 패스들을 따라가며 객체들을 선인출한다.

그림 7은 그림 4의 항해응용에 대한 반복 패턴의 포착과 선인출 과정을 보여준다. 그림 7 (a)~(d)는 객체 o_2~o_5를 액세스하는 각 과정을 나타내며, 각 과정에서 왼쪽 부분은 액세스된 객체들을 나타내고, 오른쪽 부분은 각 과정까지 발생한 타입수준 패스들의 리스트인 CTLP를 그래프 형태로 표현한 타입수준 패스 그래프를 나타낸다. 각 과정에서 새롭게 액세스되는 객체는 그림자가 있는 원으로 표시하였으며, 객체 타입수준 패스 a_0는 o_1 액세스 시에 CTLP에 삽입한 것으로 가정한다. 먼저, 과정 (a)에서는 객체 o_2의 타입수준 패스인 a_0.has_sections가 CTLP에 추가된다. 이와 유사하게, 과정 (b)와 (c)에서는 객체 o_3와 o_4의 타입수준 패스인 a_0.has_sections.is_taught_by와 a_0.has_sections.has_TA가 각각 CTLP에 추가된다. 그리고, 과정 (d)에서는 객체 o_5의 타입수준 패스인 a_0.has_sections가 CTLP에 이미 존재하므로 규칙 1에 의해 (a_0, (a_0.has_section, a_0.has_sections.is_taught_by, a_0.has_section.has_TA)*), 즉



(a) 객체 o_2의 액세스 과정 (b) 객체 o_3의 액세스 과정



(c) 객체 o_4의 액세스 과정 (d) 객체 o_5의 액세스 과정

그림 7 반복 패턴을 가지는 항해 응용의 선인출 과정

$(a_0,(a_0\odot(has_section,\ has_sections\odot(is_taught_by,\ has_TA)))^*)$와 같은 반복 패턴을 포착한다. 그리고, 이 반복 패턴에 의해 생겨날 수 있는 타입수준 패스들을 따라가며 o_5에서 o_{10}까지의 객체들을 선인출한다.[2)]

재귀 패턴의 포착과 이를 이용한 선인출도 반복 패턴과 유사한 방법으로 이루어진다. 재귀 패턴은 액세스되는 객체들의 타입수준 패스 내에 사이클이 발생하는지를 검사함으로써 포착할 수 있다. 응용에서 애트리뷰트들의 패스 $a_0..a_i..a_j$를 따라 항해하였고, $a_i..a_j$가 사이클을 구성한다고 하자. 그러면, 항해 과정에서 $a_0..a_i$, …, $a_0..(a_i..a_j)$, $a_0..(a_i..a_j).a_i$, …, $a_0..(a_i..a_j)^2$, $a_0..(a_i..a_j)^2.a_i$, … 의 타입수준 패스들이 나타날 것이다. 제안하는 알고리즘에서는 타입수준 패스 $a_0..(a_i..a_j)$가 발생했을 때, a_{i-1}를 통하여 액세스한 객체와 a_j를 통하여 액세스한 객체의 타입이 동일하므로 사이클을 구성하는 패스 $a_i..a_j$가 나타남을 인식한다. 따라서, 규칙 3에 의해 $A_k(i \le k \le j)$가 $\{A_k\rightarrow(a_k,\ a_k\odot A_{k+1})$ if $i \le k \le j-1,\ A_k\rightarrow(a_k,\ a_k\odot A_i)|(a_k)$ if $k = j\}$로 매핑되는 재귀 패턴 $(a_0..a_{k-1}\odot A_i)$이 포착된다. 다음으로 포착된 재귀 패턴에 의해 생겨날 수 있는 타입수준 패스들을 따라가며 객체들을 선인출한다.

그림 8은 그림 5의 항해 응용에 대한 재귀 패턴의 포착과 선인출 과정을 나타낸다. 그림에서는 객체 o_2의 타입수준 패스 a_0.next가 CTLP에 추가된다. 그리고, next에 의해 액세스된 o_2의 타입이 o_1의 타입과 동일하므로, next에 의해 사이클이 구성됨을 인식한다. 따라서, A가 $\{A\rightarrow(next,\ next\odot A)|(next)\}$로 매핑되는 재귀 패턴 $(a_0\odot A)$가 포착된다. 알고리즘에서는 이 재귀 패턴으로 나

2) 실제 구현에서는 타입수준 패스 그래프의 에지를 삽입된 순서대로 따라가며 에지에 해당하는 애트리뷰트 값이 가리키는 객체들을 선인출하는데, 이는 타입수준 액세스 패턴에 따라 선인출하는 것과 동일하다.

타날 수 있는 타입수준 패스들을 따라가며 객체 $o_2 \sim o_n$을 선인출한다. 여기에서 n은 선인출할 객체의 개수에 의해 제한되는데, 이 개수에 대해서는 제 4.2절에서 자세히 설명한다.

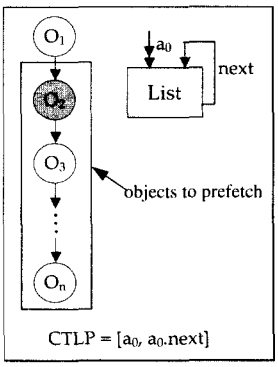


그림 8 재귀 패턴을 가지는 항해 응용의 선인출 과정

타입수준 액세스 패턴은 제3.3.2절에서 설명한 바와 같이 반복 패턴과 재귀 패턴이 함께 나타나는 복잡한 형태가 될 수 있다. 이와 같은 복잡한 형태의 패턴도 앞서 설명한 반복 패턴과 재귀 패턴의 포착 및 선인출 방법을 함께 적용하여 해결할 수 있다. 예를 들어, 그림 6의 경우 o_2를 액세스할 때 has_prerequisites으로 인해 사이클이 나타남을 인식하여 A_0와 A_1이 {$A_0 \rightarrow (a_0, a_0 \odot A_1)$, $A_1 \rightarrow$(has_prerequisites, has_prerequisites$\odot A_1$) | (has_prerequisites)}로 매핑되는 재귀패턴 (A_0)를 포착하여, $o_2 \sim o_4$를 선인출한다. 그리고, o_5를 액세스할 때에는 has_prerequisites가 반복해서 나타나는 반복 패턴임을 인식하여 {$A_0 \rightarrow (a_0, a_0 \odot A_1)$, $A_1 \rightarrow$(has_prerequisites, has_prerequisites$\odot A_1)^*$ | (has_prerequisites)*}로 매핑되는 패턴 (A_0)를 포착하며, 이에 따라 객체 $o_5 \sim o_9$를 선인출한다.

4.2 타입수준 액세스 패턴 기반 선인출 알고리즘

본 논문에서는 항해 기본 연산 함수를 객체의 참조 애트리뷰트의 값이 가리키는 객체 혹은 컬렉션 애트리뷰트의 원소들이 가리키는 객체를 반환하는 함수로 정의하는데, 기존 ORDBMS 에서도 이와 유사한 형태의 항해 기본 연산 함수를 제공한다[13, 19]. 응용 프로그래머는 이와 같은 항해 기본 연산 함수를 사용하여 항해 기능을 구현한다. 본 절에서는 제4.1절에서 설명한 선인출 방법을 항해 기본 연산 함수의 알고리즘에 어떻게 반영하는지에 대해서 설명한다. 이를 위해, 우선 선인출 기능이 없는 항해 기본 연산 함수의 알고리즘을 설명한 후, 선인출 기능을 포함하도록 항해 기본 연산 함수의 알고리즘을 확장한다.

그림 9는 선인출 기능이 없는 항해 기본 연산 함수 Basic-Navigate의 알고리즘을 나타낸다. 함수 Basic-Navigate는 이미 액세스한 객체 o_{curr}와 액세스하고자 하는 애트리뷰트 a를 입력으로 하여, o_{curr}.a가 가리키는 객체 o_{next}를 반환한다. 알고리즘에서는 우선 액세스하고자 하는 객체 o_{next}의 객체 식별자인 next_oid를 애트리뷰트 a의 타입에 따라 다르게 구한다(라인 1~4). 애트리뷰트 a가 컬렉션 애트리뷰트인 경우에는 컬렉션 원소들의 반복적인 액세스를 위해 유지하는 커서를 사용하여 next_oid를 구한다. 알고리즘의 라인 2에서 사용된 nextElement() 함수는 호출될 때 마다 커서를 이동시켜 가며 커서가 가리키는 원소를 반환하는 함수로, 이 함수가 최초로 호출될 때 커서가 열린다고 가정한다. 애트리뷰트 a가 컬렉션 애트리뷰트가 아닌 경우, 즉 참조 애트리뷰트인 경우에는 라인 4에서와 같이 o_{curr}.a 값을 next_oid로 삼는다. 다음으로, next_oid를 구한 후에는 next_oid가 가리키는 객체가 캐시에 있는지를 확인한 후, 없는 경우 이 객체를 서버로부터 읽어 오는 기능을 수행한다(라인 5~6). 마지막으로, o_{next}를 반환하는 기능을 수행한다(라인 7).

```
Basic-Navigate
Input:
  ocurr : 이미 액세스한 객체
  a     : 액세스하고자 하는 애트리뷰트 이름
Output:
  onext : 반환하는 객체

begin
1:  if a is of collection type then
2:      next_oid = nextElement(ocurr.a's cursor);
3:  else
4:      next_oid = ocurr.a;
5:  if onext, whose oid is next_oid, is not in cache then
6:      fetch onext from server;
7:  return onext;
end
```

그림 9 선인출 기능이 없는 항해 기본 연산 함수 Basic-Navigate의 알고리즘

그림 10은 함수 Basic-Navigate에 타입수준 액세스 패턴을 포착하는 기능과 이를 사용하는 선인출 기능이 추가된 항해 기본 연산 함수 Prefetch-Navigate의 알고리즘을 나타낸다. Prefetch-Navigate에서 객체 o_{next}의 객체 식별자인 next_oid를 구하는 과정(라인 1~4)은 Basic-Navigate와 동일하다. 그러나 next_oid를 구한 직후 o_{next}를 서버나 캐시로부터 읽어 반환하는 Basic-

Navigate와 달리, Prefetch-Navigate에서는 타입수준 패스를 저장하고 반복 및 재귀 패턴의 포착하는 기능과 포착된 패턴을 활용하여 선인출을 수행하는 기능을 추가로 수행한다. 알고리즘에서 나타나는 TLP(o_{curr})을 구하는 방법은 제 4.3.2절에서 자세히 설명한다.

```
Prefetch-Navigate
Input:
    ocurr : 이미 액세스한 객체
    a     : 액세스하고자 하는 애트리뷰트 이름
Output:
    onext : 반환하는 객체

begin
/* obtaining the next oid */
1:  if a is of collection type then
2:    next_oid = nextElement(ocurr.a's cursor);
3:  else
4:    next_oid = ocurr.a;
/* capturing the pattern */
5:  if TLP(ocurr).a is not in CTLP then
6:    begin
7:    insert TLP(ocurr).a into CTLP;
8:    if subpath p in TLP(ocurr).a makes a cycle then
9:      capture the recursive pattern using the cycle;
10:   end
11: else  /* TLP(ocurr).a is in CTLP */
12:     capture the iterative pattern consisting of paths in
        CTLP starting with TLP(ocurr).a;
/* fetching or prefetching the objects */
13: if onext, whose oid is next_oid, is not in cache then
14:   begin
15:     if there is a captured pattern used for prefetching
          then
16:       prefetch objects connected from ocurr according to
          the captured pattern;
17:   else
18:     fetch onext from server;
19:   end
20: return onext;
end
```

그림 10 타입수준 액세스 패턴 기반 선인출 기능을 포함한 항해 기본연산 함수 Prefetch-Navigate의 알고리즘

타입수준 패스를 저장하고 반복 및 재귀 패턴을 포착하는 기능은 라인 5~12에서 수행한다. 먼저, 액세스하고자 하는 o_{next}의 타입수준 패스 TLP(o_{next})(=TLP(o_{curr}.a))가 CTLP에 존재하지 않는다면(라인 5), 새로운 타입수준 패스인 TLP(o_{next})을 CTLP에 추가하고(라인 7), 이 패스를 조사하여 사이클이 포함된 경우에는 재귀 패턴이 발생했음을 의미하므로 재귀 패턴을 포착한다(라인 8~9). 반면에, 타입수준 패스 TLP(o_{next})가 CTLP에 이미 존재한다면(라인 11), 반복 패턴이 발생했음을 의미하므로 TLP(o_{next})와 TLP(o_{next}) 이후에 발생한 타입수준 패스들을 사용하여 반복 패턴을 포착한다(라인 12).

라인 13~20에서는 포착된 패턴을 사용하여 선인출을 수행하고 o_{next}를 반환하는 기능을 수행한다. 액세스하고자 하는 o_{next}가 캐시에 있지 않는 경우(라인 13)에는 o_{next}을 포함한 객체들을 서버로부터 인출해야 한다. 이 때, 앞서의 라인 5~12에서 포착된 패턴이 있는 경우(라인 15)에는 이 패턴에 따라 o_{next}을 포함하여 o_{curr}와 연결된 객체들을 선인출한다(라인 16). 반면에 포착된 패턴이 없는 경우(라인 17)에는 o_{next}만을 인출한다(라인 18). 마지막으로, 라인 20에서는 객체 o_{next}를 반환한다.

예 1: Prefetch-Navigate 함수에서 반복 패턴을 활용한 선인출 과정의 예는 그림 7을 사용하여 설명한다. 그림의 각 과정 (a)~(d)에서는 각각 Prefetch-Navigate 함수가 호출된다. 즉, 그림 7의 (a)는 Prefetch-Navigate(o_1, has_sections), (b)는 Prefetch-Navigate (o_2, is_taught_by), (c)는 Prefetch-Navigate(o_2, has_TA), 그리고 (d)는 Prefetch-Navigate(o_1, has_sections)를 호출한 경우이다. (a), (b), (c)에서는 o_2, o_3, o_4의 타입수준 패스인a_0.has_sections, a_0.has_sections.is_taught_by와 a_0.has_sections.has_TA가 CTLP에 존재치 않으므로(라인 5), 이들을 CTLP를 추가한다(라인 7). 그러나, 그림 7 (d)에서는 객체 o_5의 타입수준 패스인 has_sections가 CTLP에 이미 존재하므로(라인 11), 규칙 1에 의해 컬렉션 애트리뷰트 has_sections에 의한 식 (6)의 반복 패턴 (a_0, (a_0⊙(has_section, has_sections ⊙(is_taught_by, has_TA)))$^+$)을 포착한다(라인 12). 그리고, o_5가 캐시에 없는 경우, 포착한 반복 패턴에 기반하여 o_5를 포함하여 객체 o_6~o_{10}들을 선인출한다(라인 14~19). □

예 2: Prefetch-Navigate 함수에서 재귀 패턴을 활용한 선인출 과정의 예는 그림 8을 사용하여 설명한다. 그림 8은 Prefetch-Navigate(o_1, next)를 호출한 경우이다. 그림 8에서는 o_2의 타입수준 패스 a_0.next가 CTLP에 존재치 않으므로(라인 5), 이를 CTLP에 추가한다(라인 7). 그리고, 추가된 타입수준 패스의 next가 사이클을 구성하므로, 규칙 3에 의해 A가 {A→(next, next⊙A) | (next)}로 매핑되는 재귀 패턴 (a_0.A)가 포착된다(라인 8~9). 그리고, o_2가 캐시에 없는 경우, 포착한 재귀 패턴에 기반하여 객체 o_2~o_n들을 선인출 한다(라인 14~19). □

Prefetch-Navigate에서 사용하는 타입수준 패스들의 리스트인 CTLP는 항해루트집합별로 유지된다. 즉, CTLP는 항해루트집합을 구하는 질의가 수행될 때 생성되고, 항해루트집합으로부터 항해가 더 이상 수행되지 않을 때 삭제된다. 그리고, CTLP의 크기는 하나의 트

랜잭션에서 액세스하는 객체의 수가 아니라 액세스하는 타입과 애트리뷰트의 수에 비례하고, 일반적으로 그 수는 그다지 많지 않으므로, CTLP는 주기억 장치에서 관리가 가능하다.

객체들을 선인출하기로 결정했을 때, 얼마만큼의 객체들을 선인출을 하여야 하는지가 하나의 이슈가 된다. 본 논문에서는 이전에 선인출된 객체들이 얼마나 사용되었는지를 다음 선인출할 객체들의 개수에 반영함으로써 잘못된 선인출 확률을 줄이는 휴리스틱 방법을 사용하였다. 즉, 이전에 선인출한 객체가 X% 이하로 액세스되면 선인출할 객체의 최대 개수를 Y%만큼 감소시키고, X% 이상 액세스되면 선인출 할 객체의 최대 개수를 Y%만큼 증가시킨다. 단, 선인출 버퍼의 최대 크기를 결정하여 이를 넘지 않도록 하였고, 최초의 선인출할 객체의 수는 선인출 버퍼의 반으로 설정하였으며, 선인출 할 최소의 객체 수는 1로 하였다. 본 논문에서는 반복적인 실험을 통해 선인출 버퍼의 최대 크기를 1 M 바이트로 하였고, X=50, Y=50을 찾아 이용하였다. 이 값에 대해서는 디스크 액세스 비용과 네트워크 전송비용등을 고려하여 자동적으로 설정하는 방법이 향후 연구로 필요하다.

4.4 구현 이슈

4.4.1 클라이언트/서버 선인출 아키텍처

액세스 패턴에 기반한 선인출의 방식은 액세스 패턴을 포착하는 기능과 포착 정보를 사용한 선인출 기능으로 구분할 수 있다. 따라서, 액세스 패턴의 포착기능과 선인출 기능을 클라이언트와 서버의 어느 곳에 둘 것인가가 하나의 이슈가 된다.

포착 기능은 클라이언트측에 두어야 정확한 액세스 패턴을 포착할 수 있다. 왜냐하면, 항해는 ORDBMS 클라이언트에서 항해 기본 연산 함수를 호출함으로써 수행되는데, 항해에서 액세스되는 객체가 클라이언트의 객체 캐시에 이미 있는 경우에, 서버로는 메시지가 전송되지 않으므로, 서버에서는 어떠한 객체들이 클라이언트에서 액세스되는지를 알 수 없기 때문이다. 반면에 선인출 기능은 선인출될 객체들이 존재하는 서버에 구현하여야 한다. 왜냐하면 선인출할 객체들의 객체 식별자는 타입 수준 패턴에 따라 서버의 객체들을 항해해야만 알 수 있기 때문이다. 그러나, 포착된 액세스 패턴은 클라이언트에 존재하므로, 선인출할 때 포착된 액세스 패턴을 서버로 전송하는 기능이 필요하다.

4.4.2 타입수준 패스 유지 방법

본 절에서는 타입수준 패스 리스트인 CTLP의 구현 방법과 액세스되는 각 객체의 타입수준 패스를 구하는 방법(Prefetch-Navigate 함수에서 TLP(o_{curr})을 구하는 방법)에 대해서 설명한다.

먼저, 객체들의 타입수준 패스들의 리스트인 CTLP를 간결하게 유지하기 위해 알고리즘 Prefetch-Navigate에서는 그래프 구조를 사용한다. 항해란 애트리뷰트 값에 의해 서로 연결된 객체들을 따라가는 것이므로, 항해에서 액세스된 객체들의 타입수준 패스들은 타입을 노드로 하고, 애트리뷰트를 에지로 하는 그래프 구조로 표현할 수 있다. 따라서 CTLP를 그래프 구조로 표현하기 위해, Prefetch-Navigate에서는 새로운 패스 TLP(o_{curr}).a가 CTLP에 추가될 때마다 새로운 타입 노드를 하나 생성하며 TLP(o_{curr})에 대응되는 타입노드로부터 이름이 a인 에지를 사용하여 생성된 노드와 연결한다. 예를 들어, 그림 7 (a)에서는 Sections이라는 타입 노드를 하나 생성하고, Courses 노드로부터 has_sections라는 에지를 사용하여 Sections 노드와 연결한다.

그리고, 알고리즘 Prefetch-Navigate에서 현재 액세스하려는 객체 o_{curr}의 타입수준 패스 TLP(o_{curr})를 구하기 위해서는 지속성 포인터를 이용하는 방법을 사용하며, 이를 위해 Prefetch-Navigate의 입출력 매개 변수로 객체(o_{curr}와 o_{next})가 아니라 객체에 대한 지속성 포인터를 사용하도록 Prefetch-Navigate를 변경한다. 대부분의 ORDBMS에서 지속성 포인터를 사용하여 객체를 액세스하는 API를 제공하므로 지속성 포인터에 타입수준 패스를 유지하는 것이 기존 API와의 호환성을 위해 자연스럽다. 따라서, 지속성 포인터에 객체의 타입수준 패스를 저장할 수 있도록 그 구조를 확장한다. 지속성 포인터란 ORDBMS 클라이언트에서 객체를 가리키기 위해 객체 식별자 대신에 사용하는 데이타 구조로서, 그 예로는 ORRef[13], odb_Ref[19]등이 있다.

그림 11은 그림 7 (b)의 경우에서 Prefetch-Navigate (r, is_taught_by)가 호출될 때, 지속성 포인터를 사용하여 타입수준 패스를 유지하는 방법을 나타낸다. 그림에서 r은 o_2에 대한 지속성 포인터로서 타입수준 패스 has_sections를 저장하고 있다. 항해 연산을 수행하는 Prefetch-Navigate 함수에서는 r의 타입수준 패스 has_sections와 입력으로 받은 애트리뷰트 이름 is_taught_by를 연결하여 반환되는 객체 o_3의 지속성 포인터 r'의 타입수준 패스에 저장한다. 이와 같은 방식으로 지속성 포인터를 통해 액세스하는 모든 객체에 대해 타입수준 패스를 유지할 수 있다. 타입수준 패스를 유지하기 위한 방법으로, 실제 구현에서는 그림 11에서와 같이 지속성 포인터내에 타입수준 패스에 대응되는 CLTP의 노드에 대한 메모리 포인터만을 기록함으로써 저장 공간을 최

소화하고 성능 부담을 없앨 수 있다.

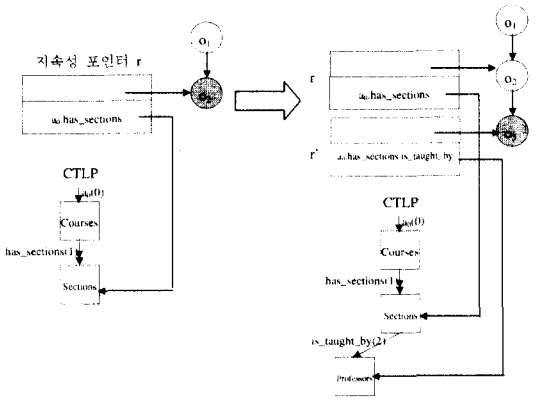


그림 11 지속성 포인터를 사용한 타입수준 패스의 유지 방법

5. 성능 평가

본 절에서는 제안하는 타입수준 액세스패턴 기반 선인출 방법(*TypePrefetch*라 표기), 요구 인출 방법(*OnDemandFetch*라 표기), 그리고 문맥 기반 선인출 방법(*ContextPrefetch*라 표기)의 성능 평가 결과를 설명한다. 제 5.1절에서는 성능 평가를 수행한 실험 종류와 실험 환경을 설명하고, 제 5.2절에서는 실험 결과를 설명한다.

5.1 실험 종류 및 실험 환경

본 논문에서는 제안하는 TypePrefetch의 우수성을 입증하기 위하여 세 가지 종류의 실험을 수행하였다. 첫 번째 실험에서는 본 논문에서 예제로 설명한 그림 1과 그림 5의 항해 응용에 대해 성능 평가를 수행하였다. 이 실험의 목적은 반복 패턴과 재귀 패턴이 나타나는 전형적인 예에서 제안하는 선인출 알고리즘이 유용함을 보이기 위해서다. 편의상 그림 1과 그림 5의 항해 응용을 각각 **반복응용**, **재귀응용**이라고 부른다. 두 번째 실험에서는 객체관계형/객체지향 DBMS의 대표적 벤치마크인 OO7 벤치마크[5, 17]의 탐색 연산에 대해 성능 평가를 수행하였다. 이 실험의 목적은 객체지향 항해 응용의 대표적 워크로드(workload)인 OO7 벤치마크에서도 제안하는 선인출 알고리즘이 유용함을 보이기 위해서다. 이 실험에서는 데이타베이스가 커지더라도 제안하는 알고리즘의 성능이 나빠지지 않음을 보이기 위해 작은 크기, 중간 크기, 큰 크기의 데이타베이스를 모두 실험하였다. 세 번째 실험에서는 실제 GIS 응용 프로그램을 사용하여 성능 평가를 수행하였다. 세 번째 실험의 목적은 실제 응용에서도 제안한 알고리즘이 유용함을 보이기 위해서다.

실험을 위해 각 인출 기법들을 한국과학기술원에서 개발해오고 있는 오디세우스 ORDBMS 프로토타입상에 구현하였으며, 실험은 실제 LAN으로 연결된 클라이언트/서버 환경에서 수행되었다. 클라이언트로는 Sun Ultra-2 워크스테이션을 사용하였으며, 서버로는 Sun Ultra-60 워크스테이션을 사용하였다. 그리고, 클라이언트의 객체 캐시 크기는 8M 바이트, 서버의 페이지 버퍼 크기는 16M 바이트로 하였으며, 원시 디스크(raw device)를 사용한 자체 디스크 관리자를 사용하였다.[3]

하나의 클라이언트 캐시를 여러 트랜잭션(사용자)들이 사용하는 환경인 경우에는 다중 사용자 실험이 필요하다. 그러나, 본 논문에서는 기존 연구와 같이 각 클라이언트 캐시마다 하나의 사용자만을 가정한 경우만을 다루므로, 다중 사용자 경우의 추가적인 이슈와 실험은 향후 연구로 남겨둔다.

실험 결과로는 TypePrefetch와 OnDemandFetch의 라운드트립 횟수[4] 비율과 수행시간 비율을 측정하고, TypePrefetch와 ContextPrefetch의 라운드트립 횟수 비율와 수행시간 비율을 측정하였다. 그리고, 노이즈(noise) 효과를 피하고 실험 결과의 정확성을 높이기 위해, 다섯번 실험한 후 평균을 취한 값을 실험 결과로 하였다.

5.2 실험 결과

반복응용과 재귀응용

반복응용에서 사용한 데이타의 자세한 내용은 다음과 같다. 전체 교수의 수는 1000명으로 하였고, 봉급이 $100,000 이상인 교수의 수는 200명으로 하였다. 그리고, 각 교수는 한 대의 자동차를 소유하는 것으로 하였고, 자동차 회사의 수는 5개로 하였다. 다음으로 재귀응용에서는 그림 5와 같은 연결 리스트구조에서 루트 객체로부터 시작하여 500개의 객체가 재귀적으로 연결된 데이타를 사용하였으며, 항해 응용이 액세스하는 순서대로 객체들을 클러스터링하였다.

표 2는 반복응용과 재귀응용에 대해 각 방법의 실험 결과를 나타낸다. 표에서 보듯이, TypePrefetch는 OnDemandFetch는 물론 ContextPrefetch에 비해 훨씬 우

3) 캐쉬 크기와 페이지 버퍼의 크기를 현실적인 범위 내에서 변경하여 실험해 보았지만, 성능 변화의 차이가 크지 않은 것으로 나타났다.

4) 라운드트립의 횟수는 클라이언트와 서버간의 RPC 횟수로 측정하였다.

수한 결과를 보이고 있다. 반복응용의 경우, TypePrefetch는 OnDemandFetch보다 라운드트립 횟수와 수행시간을 각각 64.7배와 2.49배 줄였으며, ContextPrefetch보다는 각각 50.8배와 1.95배 줄였음을 알 수 있다. 반복응용에서 수행시간 비율이 라운드트립 시간 비율보다 작은 이유는 수행시간이 라운드트립 시간뿐 아니라 객체를 디스크에서 읽기 위한 시간인 디스크 액세스 시간도 포함하기 때문이다. 즉, 디스크 액세스 시간은 선인출 알고리즘에서 줄이지 못하기 때문에 수행시간 비율이 라운드트립 횟수 비율보다는 작게 나타난다.

표 2 반복응용과 재귀응용에서의 각 인출 방법의 실험 결과

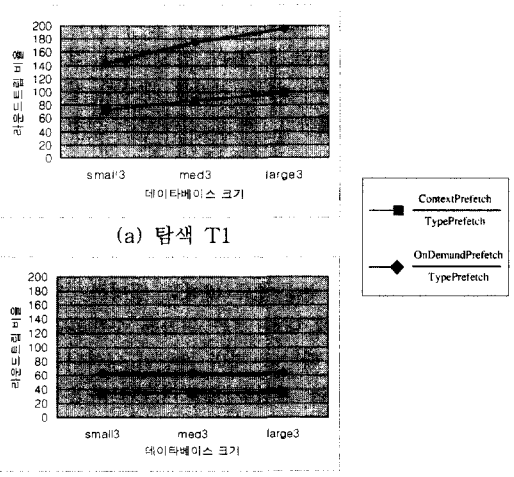
		반복응용	재귀응용
OnDemandFetch / TypePrefetch	라운드트립	67.4	56.2
	수행시간	2.49	6.12
ContextPrefetch / TypePrefetch	라운드트립	50.8	56.2
	수행시간	1.95	6.12

재귀응용에서도 TypePrefetch는 다른 두 방법에 비해 훨씬 우수한 성능을 보이고 있다. 재귀응용에서는 TypePrefetch의 수행시간이 반복응용에서보다 상대적으로 향상되었다. 그 이유는, 객체들의 저장 방식을 분석하여 본 결과, 재귀응용에서는 액세스하는 순서대로 객체들이 클러스터링되어 있어 서버에서의 디스크 액세스 비용이 반복응용에 비해 상대적으로 작기 때문이다. 따라서, 클러스터링이 잘 되어 있다면 제안하는 선인출 방법은 더 큰 효과를 볼 수 있다.

OO7 벤치마크

본 실험에서는 OO7 벤치마크의 탐색 연산을 수행하였다. OO7의 탐색 연산에 대한 자세한 소개는 참고문헌 [5]에 잘 나와 있으며, 여기서는 간략하게 소개한다. 데이타베이스는 루트객체인 하나의 모듈 객체, 모듈 객체와 연결되어 있는 트리 형태의 어셈블리 객체 계층구조, 어셈블리 객체 계층구조의 리프와 연결되어 있는 복합부품(composite part) 그래프 등으로 구성된다. 복합부품 그래프는 원자부품(atomic part)과 연결부품(connection)의 그래프로 구성된다. 탐색 T1, T2a, T2b, T2c, T3a, T3b, T3c는 하나의 루트로부터 연결된 어셈블리 계층구조를 탐색하고, 각 어셈블리와 연결된 복합부품 그래프를 탐색한다. 탐색 T6는 하나의 루트로부터 연결된 어셈블리 계층구조를 탐색하고 각 어셈블리의 루트 원자 부품만을 탐색한다.

그림 12는 OO7 벤치마크에 대한 각 인출방법의 라운드트립 횟수 비율을 보여준다. 그림 12의 T1에서 TypePrefetch 방식은 OnDemandFetch 방식과 비교하여, 라운드트립을 최대 195배, ContextPrefetch 방식과 비교하여 최대 97.8배까지 라운드트립 횟수를 줄였다. T2a~T3c에 대해서도 T1과 유사한 형태의 성능을 보였다. 그림 12 (b)의 T6에서 TypePrefetch 방식은 OnDemandFetch 방식과 비교하여 최대 63.3배 줄였고, ContextPrefetch 방식과 비교하여 최대 35.4배까지 라운드트립을 줄였다. TypePrefetch 방식이 T1~T3c에서 라운드트립 횟수를 크게 줄이는 이유는 어셈블리 계층구조와 복합부품 그래프에서 재귀 패턴과 여러 레벨로 중첩된 반복 패턴이 나타나기 때문이다. T1에서는 어셈블리 계층 및 복합 부품 그래프에 나타나는 타입수준 패턴을 모두 이용하는 반면, T6에서는 어셈블리 계층구조에 나타나는 타입수준 패턴만을 선인출에 활용하므로 OnDemandFetch와 ContextPrefetch에 대한 TypePrefetch의 라운드트립의 횟수의 감소비율이 T1보다 T6에서 상대적으로 작게 나타났다.



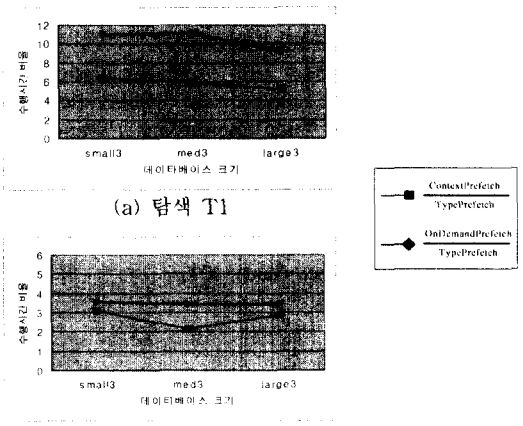
(a) 탐색 T1

(b) 탐색 T6

그림 12 OO7 벤치마크에서의 라운드트립 횟수 비율

그림 13에서 보듯이, 라운드트립을 줄인 효과는 성능에 반영되어 수행시간을 크게 감소시켰다. 그림 13 (a)의 T1에서 TypePrefetch 방식은 OnDemandFetch 방식과 비교하여 최대 11.1배, ContextPrefetch 방식과 비교하여 최대 6.39배까지 수행시간을 줄였다. T2a~T3c에 대해서도 T1과 유사한 형태의 성능을 보였다. 그림

13 (b)의 T6에서, TypePrefetch 방식은 OnDemand Fetch 방식과 비교하여 최대 3.57배, ContextPrefetch 방식과 비교하여 최대 3.10배까지 수행시간을 줄였다.[5]



(a) 탐색 T1

(b) 탐색 T6

그림 13 OO7 벤치마크에서의 성능 비율

실제 GIS 응용

본 절에서는 실제 GIS 응용을 사용한 성능 평가 결과를 제시한다. 사용한 GIS 응용은 지도 데이타를 탐색하는 지도 브라우징 시스템이며, 이 시스템은 화면에 원하는 지도 객체들을 빠르게 인출하는 것을 요구한다. 실험에서 사용한 지도 데이타는 서울 강남구의 실제 데이타로서 총 객체의 개수는 8만개이고, 데이타베이스의 크기는 10M 바이트이다. 응용에서 도로는 폴리라인(polyline)으로, 건물들은 폴리곤(polygon)으로 모델링하였으며, 폴리라인과 폴리곤은 여러 개의 라인으로 구성하는 것으로 모델링하였다.

표 3은 실제 GIS 응용에 대한 성능 결과를 나타낸다. TypePrefetch는 OnDemandFetch와 비교하여 라운드트립을 최대 402배 줄이고 수행시간을 최대 9.50배 향상시켰으며, ContextPrefetch와 비교하여 라운드트립을 최대 51.3배 줄이고 수행시간을 최대 2.38배 향상시켰다. 이와 같은 결과를 볼 때, 실제 응용에서도 타입수준 액세스 지역성이 빈번하게 발생함을 알 수 있으며, 이들 타입수준 액세스 패턴을 활용하는 TypePrefetch가 보다 효과적인 선인출 방법이라 할 수 있다.

5) ContextPrefetch에 대해서 참고문헌 [2]에서는 RDBMS를 사용하여 실험한 반면, 본 논문에서는 ORDBMS를 사용하여 실험을 수행하였다. ORDBMS는 RDBMS에 비해 서버에서 참조를 따라 연결된 객체들을 액세스하는 묵시적 조인 연산 비용이 상대적으로 작다. 따라서, 선인출에 의한 라운드트립 횟수를 줄인 효과가 성능에 많이 반영되므로, Context Prefetch의 성능이 참고문헌 [2]보다 더 우수하게 나타났다.

표 3 실제 GIS 응용에서의 각 인출 방법의 실험 결과

		GIS 응용
OnDemandFetch / TypePrefetch	라운드트립	402
	수행시간	9.50
ContextPrefetch / TypePrefetch	라운드트립	51.3
	수행시간	2.38

6. 결 론

본 논문에서는 객체지향 항해 응용의 항해 특징으로 선인출에 도움이 되는 타입수준 액세스 패턴과 타입수준 액세스 지역성이라는 새로운 개념을 제안하였다. 그리고, 객체지향 항해 응용에서 자주 발생하는 액세스 패턴인 반복 패턴과 재귀 패턴을 정형적으로 정의하였다. 또한 항해시 이러한 타입수준 액세스 패턴을 동적으로 포착하고, 이를 활용하여 선인출하는 알고리즘을 제시하였다. 마지막으로 제안한 방법의 유용성을 증명하기 위해 제안하는 방법을 오디세우스 ORDBMS에 구현하여 성능 분석을 위한 실험을 수행하였다.

실험은 1) 반복 패턴과 재귀 패턴이 나타나는 대학 데이타베이스 예제, 2) 객체관계형/객체지향 DBMS의 대표적 벤치마크인 OO7 벤치마크, 3) 실제 GIS 응용으로 구성하였다. 실험 결과, 제안하는 방법의 성능은 대부분의 경우에 있어서 요구 인출 방법은 물론 문맥 기반 선인출 방법보다 매우 우수하게 나타났다. 제안하는 방법은 요구 인출 방법과 비교하여 라운드 트립 횟수를 최대 402배 줄이고, 성능을 최대 11.1배 향상시켰으며 문맥 기반 선인출 방법과 비교하여 라운드 트립 횟수를 최대 97.8배 줄이고, 성능을 6.39배 향상시켰다. 특히, 복잡한 구조를 탐색하는 OO7 벤치마크나 실제 GIS 응용에서 요구 인출 방법 및 문맥 기반 선인출 방법과 비교하여 라운드트립 회수를 크게 줄이고 성능을 크게 향상 시켰다.

이와 같은 결과를 볼 때, 타입수준 액세스 패턴 기반 선인출 방법은 객체지향 항해 응용의 성능을 크게 향상시킬 수 있는 연구 결과로, 상용 ORDBMS에 구현될 수 있는 실용적인 결과라 믿는다.

참 고 문 헌

[1] Ahn, J. and Kim, H., SEOF: An Adaptable Object

Prefetch Policy For Object-Oriented Database Systems, In *Proc. IEEE 17th International Conference on Data Engineering*, Birmingham, U.K., pp. 4-13, April, 1997.

[2] Bernstein, P. A., Pal, S., and Shutt, D., Context-Based Prefetch for Implementing Objects on Relations, In *Proc. 21st Int'l Conf. on Very Large Data Bases*, Edinburgh, Scotland, pp. 327-338, Sept. 1999.

[3] Cattel, R. G. G. and Barry, D. K., *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.

[4] Coffman, E. G. Jr. and Denning, P. J., *Operating Systems Theory*, Prentice-Hall, 1973.

[5] Carey, M. J., DeWitt, D. J., and Naughton, J. F., The OO7 benchmark, In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., pp. 12-21, May 1993.

[6] Curewitz, K. M., Krishnan, P., and Vitter, J. S., Pratical Prefetching via Data Compression, In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., pp. 257-266, May 1993.

[7] Chang, E. E., Katz, R. H., Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object-Oriented DBMS, In *Proc Int'l Conf. on Management of Data*, ACM SIGMOD, Portland, Oregon, pp. 348-357, May 1989.

[8] Kim, W. et al., Architecture of the ORION Next-Generation Database System, *IEEE Trans. on Knowledge and Database Engineering*, Vol. 2, No. 1, pp. 109-124, Mar. 1990.

[9] Kim, W., *Introduction to Object-Oriented Databases*, The MIT press, 1990.

[10] Liskov, B. et al., Safe and Efficient Sharing of Persistent Objects in Thor, In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Montreal, Canada, pp. 318-329, June 1996.

[11] Lamb, C. et al., The ObjectStore System, *Comm. ACM*, Vol. 34, No. 10, pp. 50-63, 1991.

[12] Oracle Corp., Oracle Call Interface Programmer's Guide Release 8.0, 1997.

[13] Park, C. M., Carey, M. J., and Dessloch, S., MAJOR: A Java Language Binding for Object-Relational Databases," *In Proc. 8th Int'l Conf. Workshop on Persistent Object Systems*, Tiburon, California, pp. 112-122, Aug. 1998.

[14] Palmer, Z. and Zdonik, S. B., Fido: A Cache That Learns to Fetch, In *Proc. 17th Int'l Conf. on Very Large Data Bases*, Barcelona, Catalonia, Spain, pp. 255-264, Sept. 1991.

[15] Rosenthal, A. et al., Traversal Recursion: A Practical Approach to Supporting Recursive Applications, In *Proc. Int'l Conf. on Management of Data*, ACM SIGMOD, Washington, D.C., pp. 166-176, May 1986.

[16] Stonebraker, M. and Brown, P., *Object-Relational DBMSs*, Morgan Kaufmann, 1999.

[17] Subramanian, M. and Krishnamurthy, V., Performance Challeges in Object-Relational DBMSs, *IEEE Data Engineering Bulletin*, Vol. 22, No. 2, pp. 27-31, 1999.

[18] Taylor, R. G., *Models of Computation and Formal Languages*, Oxford University Press, 1998.

[19] UniSQL, Inc., UniSQL/X Application Program Interface Reference Guide, 1995.

한 욱 신

1994년 2월 경북대학교 컴퓨터공학과 학사. 1996년 2월 한국과학기술원 전산학과 석사. 2001년 8월 한국과학기술원 전산학과 박사. 2001년 9월 ~ 현재 한국과학기술원 첨단정보기술연구센터 Post-doc. 관심분야는 객체 관계형 DBMS, XML DBMS, 캐쉬 관리 및 선인출

문 양 세

정보과학회논문지 : 데이타베이스 제 28 권 제 1 호 참조

황 규 영

정보과학회논문지 : 데이타베이스 제 28 권 제 1 호 참조