Soprano:객체 저장 시스템의 설계 및 구현

(Soprano : Design and Implementation of Object Storage System)

안 정 호 † 이 강 우 †† 송 하 주 † 김 형 주 †††
(Jung-Ho Ahn) (Kang-Woo Lee) (Ha-Joo Song) (Hyoung-Joo Kim)

요 약 객체 저장 시스템은 지속성 객체의 저장 및 접근 등 객체를 다루는데 필요한 모든 기본적인 기능을 통해 객체지향 DBMS의 상위 모듈에 추상적인 객체 개념을 제공하는 모듈이다. 본 연구에서는 여러 다양한 응용 프로그램에서 사용할 수 있는 고성능 객체 저장 시스템, Soprano를 설계, 구현하였다. 본 객체 저장 시스템은 일관된 객체 개념을 지원하고 효율적인 객체 캐쉬 관리 및 스위즐링/언스위즐링 등을 통해 빠른 객체 저장 및 접근을 제공한다. 또한 다양한 지속성 클래스를 제공하여 사용자가 쉽게 지속성 프로그래밍을 할 수 있도록 도와주며, 객체지향 설계 및 구현을 통한 시스템의 확장성 및 재사용성도 제공한다. 본 논문에서는 Soprano의 전체 구조와 특징을 소개하고 주요 구현 알고리즘을 설명하였다. 그리고 OO1 벤치마크를 수행하여 Soprano의 성능을 평가하였다.

Abstract The object storage system provides object abstraction to higher modules of object-oriented DBMS with all of the primitive operations on objects including storing and accessing objects. In this research, we designed and implemented a high-performance object storage system for various advanced database applications, called Soprano. Soprano provides an uniform object abstraction and supports efficient object cache management and pointer swizzling/unswizzling for fast object manipulations. And Soprano gives us the ease of persistent programming through many useful persistent classes. Soprano was designed with an object-oriented fashion, which naturally provides extensibility and code reusability. This paper presents the overall architecture and features of Soprano and shows the details of implementation algorithms. We also evaluate its performance via OO1 benchmark which is a standard OODB benchmark.

1. 서 론

객체지향 DBMS는 객체지향 데이타 모델의 강력한 표현 능력을 바탕으로 CAD/CAM/CASE, 인공지능 전문가 시스템 쉘, 멀티미디어 사무 정보 시스템 등 차세대 응용 분야의 기반 DBMS로서 그 위치를 점차 확고히 하고 있다. 객체지향 DBMS는 차세대 응용 분야에서 요구하는 복잡한 형태의 내포형 구조나 버젼 제어 기능을 비롯하여 동적 스키마 변경과 같은 새로운 기능도 제공하고 있다.

· 본 연구는 서울대학교 반도체공동연구소의 교육부 반도체분야 학술연구조성비(과제번호: ISRC 95-E-2010)에 의해 수행되었습니다

† 비 회 원 : 서울대학교 컴퓨터공학과
†† 학생회원 : 서울대학교 전산과학과
††† 종신회원 : 서울대학교 컴퓨터공학과 부교수

논문접수 : 1995년 9월 12일
심사완료 : 1996년 1월 28일

저장 시스템은 관계형 DBMS나 계층형 DBMS 등 기존의 DBMS의 하부 구조로 물리적 장치에서의 화일 관리, 페이지 관리, 레코드 관리 등을 통해 데이타의 저장, 삭제, 검색 등의 기능을 담당한다. 저장 시스템은 DBMS의 핵심 구성 모듈로 DBMS의 성능에 매우 큰 영향을 미친다. 그러나 객체지향 DBMS에서는 이러한 레코드 중심의 저장 시스템을 바로 이용하기 어렵다. 그 이유는 기존의 저장 시스템에서 제공하는 것은 단순한 레코드 구조의 저장과 검색인 반면, 객체지향 DBMS에서는 객체지향 데이타 모델의 강력하고 풍부한 특성들을 지원해야 하기 때문이다. 즉, 객체지향 DBMS가 기존의 저장 시스템 상에서 객체지향 데이타 모델을 지원하기 위해서는 저장 시스템의 레코드 중심의 저장 모델과 객체지향 데이타 모델 사이의 의미적 격차를 스스로 극복해야 한다. 이는 DBMS 시스템의 복잡도를 증가시키고 질의 최적화 등을 어렵게 하기 때문에 성능에도 악 영향을 미칠 수 있다[3].

Copyright (C) 2005 NuriMedia Co., Ltd.

객체 관리자란 객체지향 DBMS 상위 모듈과 하부 저장 시스템 사이에 존재하는 모듈로, 두 구조 사이의 개념적 불일치를 극복하는 것을 그 목적으로 하고 있다. 다시 말하면, 객체 관리자란 하부 저장 시스템의 기능을 이용하여 상위 구조에 객체 개념을 제공하는 모듈이다. 객체 관리자는 객체를 다루는데 필요한 기본적인 모든 기능을 제공하여 상위 모듈의 복잡도를 떨어뜨리고 성능 향상에 기여한다. 따라서 객체지향 DBMS의 성능 향상을 위해서는 효율적인 객체 관리자가 반드시 요구된다.

기본적인 객체 관리자의 기능은 다음과 같은 것이 있다[3].

1. 지속성 객체의 생성 및 저장, 삭제 기능
2. 지속성 객체 식별자 관리 기능
3. 지속성 객체의 공유 기능
4. 지속성 객체 접근 기능
5. 지속성 객체의 이름 관리 기능

이 밖에도 객체 버전 관리 기능이나 클러스터링 기능 등도 포함한다.

지금까지 개발된 객체 관리자로는 Massachusetts 대학의 Mneme[15], Exodus 저장 시스템[9] 상에서 구현된 EPVM[18], Brown 대학의 ObServer[11] 등 연구 목적의 객체 관리자를 비롯하여, O2 객체지향 DBMS의 객체 관리자[3] 등 많은 상업용 OODBMS의 객체 관리자가 존재한다. 그리고 국내 관련 연구로는 Obase[2] 시스템과 ODYSSEUS[1] 시스템의 객체 관리자를 들 수 있다. 그러나 이들 객체 관리자는 단순한 형태의 객체 캐쉬 관리만을 지원하고 있으며 스위즐링/언스위즐링[20]도 제한적으로 이루어지고 있다. 또한 객체 저장을 위해서 관계형 DBMS의 저장 시스템을 그대로 사용함으로써 일관된 객체 개념의 지원이 어렵고 불필요한 오버헤드를 초래할 수 있다. 그리고 가상 함수(virtual function)나 가상 베이스클래스(virtual base class) 등 C++의 주요 개념들을 완벽하게 지원하지 못하고 있다.

본 논문에서 소개하고자 하는 Soprano(SNU Object Persistent Repository with Advance Novel Operations)는 여러 다양한 응용 프로그램에서 사용할 수 있는 고성능 객체 저장 시스템[1]으로 시스템내의 모든 개념을 객체로서 접근할 수 있도록 일관된 객체 개념을 지원하고 효율적인 객체 캐쉬 관리 및 포인터 스위즐링/언스위즐링 등을 통해 빠른 객체 저장 및 접근을 제공한다. 또한 Soprano는 다양한 지속성 클래스를 제공하고 가상 함수나 가상 베이스클래스 등 C++의 모델링 개념을 지원함으로써[2] 사용자가 쉽게 지속성 프로그래밍을 할 수 있도록 도와준다. 아울러 Soprano는 객체지향 설계 및 구현을 통한 시스템의 확장성도 제공한다.

본 논문의 제 2 절에서는 먼저 Soprano의 전체적인 구조를 소개하고 제 3 절에서는 주요 모듈에 대한 특징 및 구현 알고리즘을 자세히 설명하겠다. 그리고 제 4 절에서는 지금까지 개발된 주요 객체 관리자들을 Soprano의 특징과 비교 소개하겠다. 끝으로 제 5 절에서는 OO1 벤치마크[5]를 수행한 결과를 소개하고 이를 분석하겠다.

2. Soprano의 구조

Soprano의 기본 목표는 사용자에게 일관된 객체 개념을 지원하고 여러 다양한 응용에서 사용할 수 있도록 만족할 만한 성능을 제공하는 것이다.

그림 1은 Soprano의 전체 구조를 보여주고 있다. 그림에서와 같이 Soprano는 물리적 장치를 다루는 ObjectBase와 페이지 단위의 버퍼 관리를 위한 페이지 캐쉬, 그리고 지속성 객체의 버퍼 관리를 담당하는 객체 캐쉬가 있다. 또한 동시성 제어 및 고장 회복을 위한 로크 관리자, 로그 관리자, 트랜잭션 관리자 등이 있다.

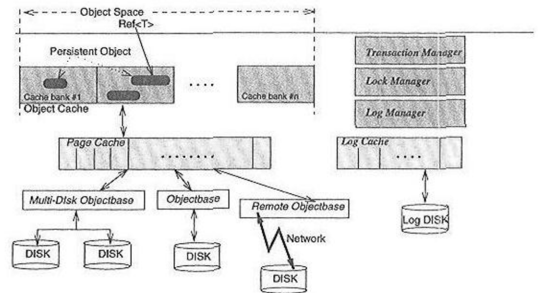


그림 1 Soprano의 전체 구조

O2 객체지향 DBMS[3] 등과 같은 객체 관리자는 기존의 관계형 DBMS의 저장 시스템상에서 구현되었다. 그러나 이러한 방식은 이미 존재하는 저장 시스템을 그대로 사용할 수 있다는 장점은 있으나, 화일과 같은 불필요한 개념으로 인한 성능 저하 및 시스템의 복잡성을 그 단점으로 들 수 있다. 반면에 Soprano는 물리적 저

1) 본 논문에서 객체 저장 시스템이란 객체 관리자와 저장 시스템의 기능을 모두 가진 하나의 통합된 시스템을 말한다.

2) 이를 위해서는 스키마 정보가 필요하다.

Copyright (C) 2005 NuriMedia Co., Ltd.

장 장치의 관리부터 객체 캐쉬 관리에 이르기까지 모든 지속성 객체의 관리가 하나의 일관된 흐름상에서 이루어지는 통합된 구조로 되어 있다. 이러한 구조는 시스템의 유지 보수의 용이성과 같은 소프트웨어 공학적인 측면의 장점 외에도 시스템에 새로운 지속성 객체를 쉽게 추가할 수 있는 확장성을 제공하며 불필요한 오버헤드를 제거하여 성능의 향상을 꾀할 수 있다.

한 예로 Soprano에서는 기존의 저장 시스템과는 달리 화일이라는 개념을 직접 제공하지 않는다. 그 이유는 객체 저장 시스템이 객체를 중심으로 읽기/쓰기를 수행해야 하기 때문이다. 즉, 화일의 open/close 와 같은 부수적인 연산 없이 객체를 바로 접근할 수 있어야 한다는 것이다. 따라서 기존의 화일 개념은 관계된 객체들 간의 클러스터링 수단이나 순차적인 접근의 수단으로서만 요구된다. 또한 다양한 객체 클러스터링을 위해 서로 다른 타입의 객체도 같은 화일에 저장할 수 있어야 하는데, 이는 하나의 화일 내에는 주로 같은 타입의 데이타를 저장한다는 일반적인 화일의 개념과는 대치된다. 따라서 Soprano 시스템에서는 객체 그룹이라는 개념을 통해 클러스터링을 제공하며 기존의 화일 시스템에서 제공하는 화일 연산은 지속성 객체로서 주어진다.

또한 Soprano는 '모든 것이 객체' 라는 일관된 객체 개념을 지원한다. 즉, 일반 지속성 객체는 물론 멀티미디어 데이타를 저장하기 위한 BLOB 구조나 B+트리와 같은 인덱스 구조 및 화일 개념도 지속성 객체로서 제공한다. 뿐만 아니라 객체의 이름을 저장하기 위한 디렉토리도 지속성 객체로서 제공된다. 이와 같은 일관된 객체 개념을 지원함으로써 사용자는 객체 저장 시스템과 이를 사용하는 프로그래밍 언어 사이에 어떠한 개념적 불일치 없이 쉽게 지속성 프로그래밍을 할 수 있다.

2.1 클라이언트-서버 구조

Soprano는 각 클라이언트마다 하나의 서버 프로세스가 연결되어 서비스를 제공하는 다중 프로세스 구조이다. Soprano의 클라이언트-서버 구조는 그림 2와 같다. 이때 각 서버 또는 클라이언트 프로세스의 내부 구조는 그림 1과 같다. 먼저 서버 프로세스는 페이지 캐쉬의 요구를 수행하는 페이지 캐쉬 요구 중개자(request broker), 디스크 연산을 대신하는 ObjectBase 요구 중개자, 로크 요구를 처리해주는 로크 요구 중개자 등 여러 요구 중개자들로 구성된다. 즉, 서버 프로세스는 클라이언트 프로세스에게 페이지의 할당 및 반환, 읽기, 쓰기, 로크의 획득 및 반환 등의 연산을 대신 처리하고 그 결과를 전달한다. 따라서 서버와 클라이언트 사이의 데이타 전송 단위는 페이지 또는 페이지의 집합이 된다. 이는 페이지를 직접 다루어야 하는 인덱스나 BLOB 구조에 대한 연산을 클라이언트에서도 수행할 수 있도록하기 위함이다. 서버 프로세스들은 페이지 캐쉬, 로그 캐쉬, ObjectBase 등은 물론 전역 시스템 객체 및 데이타도 함께 공유한다. 클라이언트 프로세스는 초기화시 Broker Init Daemon 에 의해 생성된 요구 중개자와 연결하여 클라이언트에서 수행할 수 없는 모든 서비스를 요구 중개자를 통해 수행한다. 그리고 같은 노드 내에서 수행되는 클라이언트 프로세스들은 서버 프로세스들과 마찬가지로 페이지 캐쉬를 비롯한 여러 전역 시스템 객체 및 데이타를 공유한다. 그리고 각 클라이언트 노드마다 소환 데몬(callback daemon)을 수행시켜 트랜잭션간 캐싱(intertransaction caching)을 지원한다[3]. 이러한 클라이언트 프로세스간 데이타 공유 및 트랜잭션간 캐싱은 네트워크를 통한 데이타 전송을 최소화하여 시스템의 처리량을 높일 수 있다.

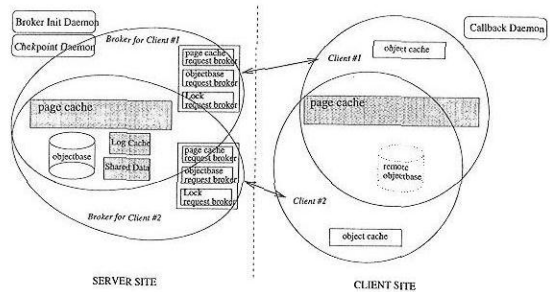


그림 2 Soprano의 클라이언트-서버 구조

3. Soprano의 구현

이번 절에서는 Soprano의 주요 모듈에 대한 특징과 구현에 대해서 자세히 소개하겠다.

3.1 ObjectBase

Soprano의 ObjectBase는 물리적 장치를 하나의 연속된 페이지들의 집합으로 추상화시킨 개념이다. ObjectBase는 디스크 상에서의 페이지 할당 및 반환을 담당하며, 클러스터링을 위한 세그먼트 그룹(segment group)이라는 개념을 지원한다. 세그먼트 그룹은 디스크 상에서 관계된 객체들을 서로 가까운 곳에 저장하기 위한 수단으로 객체를 저장할 때 클러스터링 힌트로서 사용자가 제공한다. 세그먼트 그룹은 세그먼트들의 집합이며, 세그먼트는 다시 연속된 페이지들의 집합으로 정의된다. Soprano의 ObjectBase는 실제 물

3) 트랜잭션간 캐싱을 위한 소환 기능은 현재 구현 중에 있다.

Copyright (C) 2005 NuriMedia Co., Ltd.

리적 디스크 장치의 종류에 상관없이 동일한 연산과 개념을 지원함으로써 시스템의 확장성을 증가시킨다. 즉, 새로운 물리적 장치를 추가하고자 하는 경우에는 ObjectBase의 하위 클래스로서 이 물리적 장치를 다루는 클래스를 정의하고 이에 해당하는 연산들만을 재정의 하면 된다[4].

이와 같이 ObjectBase는 연속된 페이지들의 집합으로서 페이지의 할당/반환과 같은 페이지 단위의 연산을 제공한다. 그러나 사용자는 데이타베이스를 지속성 객체들의 집합으로 바라볼 수 있어야한다. OBase는 ObjectBase 상에서 객체의 생성/삭제와 같은 연산을 구현함으로써 사용자에게 ObjectBase를 지속성 객체들의 집합으로서 추상화시킨 개념이다.

OBase는 데이타베이스 상에서 객체의 생성 및 반환을 담당하며, 관련된 객체들 사이의 클러스터링을 위해서 객체 그룹(object group)[5]이라는 개념을 지원한다. 객체 그룹은 데이타베이스 상에서 관계된 객체들을 서로 가까운 장소에 저장하기 위한 수단으로 객체의 생성시 객체 그룹을 명시한다. 객체 그룹은 클러스터링외에도 관계있는 객체들의 집합체로서 역할을 하며, 따라서 객체 그룹 내에 속하는 지속성 객체들을 순차적으로 검색할 수 있는 기능도 제공한다

3.2 객체 식별자 및 스위즐링

Soprano는 물리적 저장 장치 내의 주소 값을 기반으로 하는 물리적 객체 식별자를 채택하였다. 그 이유는 비록 물리적 식별자가 객체의 이동을 어렵게한다는 단점은 있으나, 실제 주소 값을 얻기 위해 변환 연산이 요구되는 논리적 객체 식별자보다 빠른 객체 접근을 제공하기 때문이다[12]. 그리고 지속성 객체의 이동은 새로 이동된 장소의 물리적 주소를 갖는 객체를 이전 위치에 저장하고 이를 통해 간접 접근함으로써 해결할 수 있다[3, 15]. 반면에 논리적 객체 식별자의 경우, 논리적 주소를 물리적 주소로 변환하기위한 추가적인 연산이 필요하며, 객체의 이동이 많아질수록 이 변환 연산의 비용도 증가하게 된다[7]. 그리고 식별자 내에 클래스 정보를 기록하는 경우에는 스키마 변환이 매우 어렵게 된다.

일반적으로 지속성 객체에 대한 참조는 객체 식별자로 이루어진다. 그러나 객체 식별자를 통해서 메모리내의 객체를 접근하는 것은 많은 오버헤드를 유발한다. 즉, 객체 식별자를 사용하여 해당 객체의 메모리내 주소 값을 얻은 후, 이 값을 사용하여 객체를 접근해야 하기 때문이다. 대개의 경우 객체 저장 시스템은 객체 식별자를 메모리 주소 값으로 변경하기 위한 해쉬 테이블을 유지한다. 이러한 방법은 간편하나 동일한 객체를 여러 번 접근하는 경우에도 매번 해쉬 테이블을 검색하여 주소 값을 얻어야 하므로 일반 메모리 객체에 비해 접근 속도가 상당히 뒤떨어진다. 이를 극복하기 위한 방법이 바로 포인터 스위즐링이다[16]. 포인터 스위즐링은 처음 지속성 객체를 참조하는 경우에는 객체 식별자를 사용하나, 일단 메모리로 적재된 객체에 대해서는 해당 객체의 주소 값으로 객체 식별자를 대신하는 방법이다. 따라서 같은 객체를 여러 번 접근하는 경우 처음 한번의 해쉬 테이블 검색만을 요구하고, 이후 모든 객체 접근은 스위즐링된 주소 값을 사용하여 바로 접근한다.

이러한 포인터 스위즐링을 통하여 지속성 객체에 대한 접근 속도를 일반 메모리 객체와 같은 수준으로 향상시킬 수 있다. 스위즐링의 기법에는 포인터의 스위즐링 여부를 검사하는 방법에 따라 하드웨어에 의한 방법 및 소프트웨어에 의한 방법으로 구분하며, 또 스위즐링 시기에 따라서도 EAGER 스위즐링과 LAZY 스위즐링으로 나뉜다[16, 20]. 먼저 하드웨어의 페이지 폴트 검사(page fault detection) 기능에 의존하는 스위즐링 방법은 지속성 객체에 대한 접근이 일반 메모리 객체와 같은 속도로 이루어진다는 점과 객체 접근의 투명성을 가장 큰 장점으로하고 있다. 그러나 페이지 폴트 검사의 제한과 운영체제의 폴트 핸들러의 비용으로 인하여 주로 Texas[19]나 Quickstore[20]와 같은 단일-레벨(single-level) 객체 저장 시스템에서 사용하고 있다. 반면 소프트웨어에 의한 스위즐링은 객체 접근시 매번 지속성 포인터의 스위즐링 여부를 검사하는 방법으로 순수한 메모리 객체의 접근보다는 속도가 다소 떨어지나 객체 식별자의 크기나 구조에 제한이 없어 Soprano와 같은 이중-레벨(two-level) 객체 저장 시스템에 적합하다. 따라서 Soprano는 포인터의 스위즐링 여부를 소프트웨어로 검사하는 소프트웨어 방식의 스위즐링을 사용하며, 불필요한 포인터 스위즐링을 없애기 위해서 포인터를 사용하는 마지막 순간에 스위즐링을 수행한다(LAZY 스위즐링). 뿐만 아니라 Soprano에서는 객체에 대한 포인터가 더 이상 사용되지 않거나 객체가 객체 캐쉬로부터 방출될 때 스위즐링된 메모리 주소 값을 다시 객체 식별자로 복원시키는 언스위즐링(unswizzling)을 수행함으로써 객체 캐쉬내의 객체를 임의의 시기에 방출할 수 있다.

4) 현재는 원격 디스크를 사용할 수 있는 RemoteBase 클래스가 제공된다.

5) 객체 그룹은 ObjectBase의 세그먼트 그룹을 OBase 입장에서 바라본 개념이다.

Copyright (C) 2005 NuriMedia Co., Ltd.

Soprano에서는 객체 접근을 위해 ODMG-93[4]에서 제시한 Ref 핸들러를 제공한다. Ref 핸들러는 스위즐링된 메모리 주소 값을 저장하는 memptr와 실제 참조하는 객체 식별자로 구성되어 있다. 이러한 구조는 일반적으로 스위즐링된 메모리 주소 값을 객체 식별자가 저장되어 있던 위치에 기록하여 사용하는 방법과는 달리 두 값을 함께 갖는 방법이다. 이러한 방법은 두 값을 같은 위치에서 공유하여 사용하는 것 보다 저장 공간이 많이 드는 단점이 있다. 그러나 저장 공간의 차이가 단지 스위즐링된 주소 값을 저장하기위한 4 바이트뿐이고, 이러한 공간 부담은 현재의 하드웨어 기술 수준으로는 별다른 부담으로 작용하지 않는다. 반면에 두 값을 분리 저장함으로써 얻는 장점은 다음과 같다.

첫째, 스위즐링 여부의 비교를 빠르게 할 수 있다는 것이다. 객체 식별자와 스위즐링된 주소 값을 공유하여 사용하는 경우 대개 객체 식별자의 특정 비트를 통해 스위즐링 여부를 판별한다. 또한 객체 접근을 위해서 추가적인 비트 연산이 요구되기도 한다. 그러나 Soprano의 경우 스위즐링된 주소 값을 기록하는 memptr의 NULL 여부만을 판별하면되고 추가적인 연산도 요구되지 않는다. 특히 RISC 구조에서는 워드 단위의 비교 연산이 2 개정도의 기계어로 표시되며, 더욱이 스위즐링 여부를 검사하기 위해 읽어들인 memptr의 값은 바로 객체 접근을 위해서 사용하므로 실제 소프트웨어의 검사를 통해 요구되는 오버헤드는 수 사이클 정도로 작다[6].

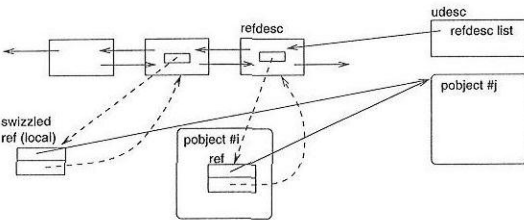


그림 3 역 참조 리스트의 관리

둘째, 객체 식별자와 스위즐링된 주소 값을 공유하는 경우, 스위즐링된 객체를 다시 물리적 장치에 저장하기 위해서는 스위즐링된 주소 값을 해당 주소 값에 위치한 객체의 식별자로 복원해주어야 한다. 이때 주소 값을 통해 객체 식별자를 구하는 비용으로 인하여 언스위즐링이 어려워진다. 반면 Soprano에서는 객체 식별자가 저장된 곳을 언스위즐링을 위한 역 참조 리스트를 관리하는 데 사용함으로써 객체의 언스위즐링을 용이하게 한다. Soprano에서는 역 참조 리스트를 관리하기 위해서 그림 3과 같은 구조를 유지하고 있다.

즉, 그림에서와 같이 객체 캐쉬내의 모든 지속성 객체는 자신을 가리키고 있는 스위즐링된 Ref 핸들러들을 유지한다. 이 역 참조 리스트는 Ref 핸들러가 스위즐링될 때 대상 객체의 역 참조 리스트에 해당 Ref 핸들러를 등록하고 언스위즐링될 때 Ref 핸들러를 리스트로부터 삭제함으로써 유지 관리한다. 이때 각 Ref 핸들러는 객체 식별자를 역 참조 리스트내의 자신에 대한 노드를 가리키고 그 노드도 해당 Ref 핸들러를 가리킨다. 이 값을 통해 Ref 핸들러가 언스위즐링될 때 자신을 역 참조 리스트로부터 쉽게 삭제할 수 있으며, 또한 한 객체가 객체 캐쉬로부터 방출할 때도 역 참조 리스트를 따라가면서 모든 Ref 핸들러들을 언스위즐링 시킬 수 있다.

한편 Soprano는 C++의 가상 함수와 가상 베이스클래스를 지원하기 위해서 클래스의 스키마 정보를 사용한다. 먼저 Soprano는 지속성 객체를 디스크로부터 처음 메모리로 읽어들일때, 생성자(constructor)를 호출하여 C++ 객체의 은닉된 포인터[8] — 가상 함수 포인터 테이블을 가리키는 포인터와 가상 베이스클래스의 객체가 위치한 곳을 가리키는 포인터 — 를 초기화시킨다[7].

그러나 은닉된 포인터 값의 초기화만을 통해서는 가상 베이스 클래스를 지원할 수 없다. 예를 들면 클래스 A가 클래스 B의 가상 베이스 클래스인 경우, A에 대한 Ref 핸들러를 통해서 B의 객체를 가리킬 수 없다. 그 이유는 가상 베이스 클래스인 경우 A 객체의 내용이 B 객체의 시작 위치가 아닌 B 객체의 내부에 저장되므로 단순히 핸들러의 memptr 값을 반환하는 것으로는 정확한 A 객체의 위치를 접근할 수 없기때문이다. 이를 해결하기 위해서 Soprano에서는 스키마 정보를 이용하여 스위즐링시 B 객체 내에서의 A 객체의 시작 위치를 Ref 핸들러 내에 캐쉬해두고, 객체 접근시 이 값을 사용하여 실제 A 객체의 위치를 계산하여 반환한다.

3.3 페이지 캐쉬

Soprano에서는 페이지 단위 로킹과 페이지 단위 REDO-ONLY 물리적 로깅 방법을 제공한다. 페이지 로크가 획득되는 시점은 처음으로 트랜잭션이 페이지를 페이지 캐쉬를 통해 읽으려는 시점이고, 로그 레코드가 생성되는 시점은 트랜잭션이 완료(commit)될 때로, 트랜

6) 40MHz Sparc에서 Soprano를 사용하여 벤치마크를 수행한 결과, 소프트웨어 검사 방법의 오버헤드는 4.3 사이클이다.

7) 은닉된 포인터의 초기화를 위한 생성자는 클래스 등록시 자동으로 생성된다.

Copyright (C) 2005 NuriMedia Co., Ltd.

잭션이 변경시킨 페이지의 이미지를 로그 장치에 REDO-ONLY 로그 레코드로 기록한다. 페이지 캐쉬는 REDO-ONLY 로깅 기법을 지원하기 위해 ¬STEAL/¬FORCE 캐쉬 관리 기법[10]을 사용한다.

Soprano에서의 페이지 캐쉬는 로그 관리자와 로크 관리자와 통합되어 작동한다. 트랜잭션이 페이지 캐쉬를 이용하여 특정 페이지를 접근할 때, 캐쉬는 해당 페이지에 대한 로크를 획득하고, 트랜잭션 완료시에는 변경된 페이지에 대한 REDO 로그를 생성하여 로그 장치에 저장한다. 즉, 로크 획득 및 로그 레코드 생성은 페이지 캐쉬를 통해 자동으로 수행된다. 이와 같이 통합된 형태의 페이지 캐쉬, 로그 관리자, 로크 관리자는 다음과 같은 장점을 제공한다.

첫째, 페이지 캐쉬의 인터페이스만을 이용하여 로크 획득 및 로그 레코드 생성이 자동적으로 이루어지기 때문에, 페이지 캐쉬를 이용하여 데이타베이스를 접근하는 모든 상위 모듈들은 데이타의 일관성 및 지속성을 구현하기 위한 추가적인 노력이 필요없게 된다.

둘째, 페이지 단위 로킹은 레코드 단위 로킹보다 로크 요청 횟수가 적기때문에 로크 획득에 필요한 서버와의 통신 횟수를 줄일 수 있다. 이로써 시스템의 성능을 높일 수 있으며, 또한 캐쉬의 페이지 단위와 로크의 단위가 같기 때문에, 서버로의 페이지 요청에 로크 획득 요청을 피기백(piggyback) 시킴으로써 로크에 필요한 추가적인 메시지 전송을 없앨 수 있다[17].

셋째, ¬STEAL/¬FORCE 캐쉬 관리를 통한 REDO-ONLY 고장복구 기법은 페이지 서버 형태의 구조에서 효율적인 클라이언트 트랜잭션 철회 및 서버 재가동 고장복구 방법을 제공한다. 수행 중인 트랜잭션에 의해 갱신된 클라이언트 페이지는 해당 트랜잭션이 완료되기 전에는 서버로 전송되지 않기 때문에, 클라이언트에서 트랜잭션이 철회되는 경우에는 단순히 클라이언트 내의 변경된 페이지들을 무효화시킴으로써 철회 처리가 끝나게 된다. 서버의 디스크에는 오직 완료된 트랜잭션이 변경시킨 내용만 기록되기 때문에 재가동 고장복구시 REDO 단계만을 수행하면 된다. 뿐만 아니라, 여러 번 변경된 각 데이타 페이지에 대해서도 오직 한번의 REDO 처리만 필요하기 때문에(3.5 절 참조), 빠르게 REDO를 처리할 수 있다[8]. 또한 트랜잭션 완료나 철회시 트랜잭션이 변경시킨 데이타를 디스크에 저장시킬 필요가 없어 일반적인 트랜잭션의 처리 속도가 좋다. 그리고, 클라이언트에서 생성된 로그 레코드는 트랜잭션이 완료되는 시점에 모두 서버로 전송되어 로그 장치에 집중적으로 기록된다. 이는 네트워크의 대역폭(bandwidth)과 로그 장치의 I/O 대역폭을 충분히 활용할 수 있도록 하여 신속한 트랜잭션 완료처리를 가능하게 한다[9].

그러나 이러한 로킹 및 로깅 기법은 다음과 같은 단점을 지적할 수 있다. 첫째, 페이지 단위 로킹은 시스템의 동시성 정도를 저하시키는 단점을 갖는다. 그러나 관계형 DBMS의 응용과는 달리, 응용 프로그램간 데이타의 공유도가 상대적으로 낮은 객체지향 데이타베이스 응용에서는 페이지 단위 로킹의 로크 획득을 위한 서버 호출 횟수를 줄임으로써 로크 획득 부하를 줄일 수 있다. 또한 Soprano와 같은 페이지 서버 형식의 클라이언트-서버 구조에서는 페이지 단위 로킹이 비교적 잘 어울리는 기법으로 알려져 있다[6]. 그리고 Soprano에서는 중첩-최상위 연산(nested-top action)[14]을 제공하여, 일부 자주 접근되는 데이타(예를 들어 시스템 디렉토리 객체나 스키마 관리자)에 대해서는, 중첩-최상위 연산을 통해 접근하게 하여 로크 유지 시간을 가능한 줄이게 하고 있다.

둘째, 페이지 단위 로깅 기법은 로그 공간의 낭비를 초래할 수 있다는 점과 많은 양의 로그 레코드 생성으로 인해 트랜잭션 완료 기간이 길어지는 점을 들 수 있다. 그러나 앞서 언급한 바와 같이, 반복적으로 같은 데이타(혹은 같은 페이지 내에 속해 있는 데이타)를 갱신하는 경우에 페이지의 변경 횟수와는 관계없이 오직 하나의 페이지 로그 레코드를 생성하므로 객체 단위의 로그 레코드 생성 방법에 비해 로그 공간 사용량을 줄일 수 있다[21]. 또한 트랜잭션이 실패되는 경우에는 데이타 갱신에 의한 로그 레코드가 전혀 생성되지 않기 때문에, 로그 장치 공간을 절약 할 수 있다.

마지막으로 ¬STEAL/¬FORCE 캐쉬 관리 기법은 페이지 캐쉬의 성능을 저하시키거나 캐쉬가 꽉차는 현상을 발생시킬 수 있다[10]. Soprano에서는 이러한 단점을 보완하기 위해 스왑(swap) 기능을 제공한다. 페이지 캐쉬는 아직 종료되지 않은 트랜잭션이 변경한 페이지를 스왑 장치에 임시 저장하므로 페이지 캐쉬가 꽉 차거나, 페이지 캐쉬의 효율이 저하되는 것을 막는다. 페이지 캐쉬내의 페이지에는 객체가 디스크에 저장된 형태 — 즉, 언스위즐링된 형태로서만 존재하기때문에 페이지 스왑은 포인터 스위즐링에 영향을 미치지 않는다.

8) 페이지 전체를 로그 레코드로 하지 않는 경우에는 REDO가 필요한 페이지를 일관된 페이지로 전이시키기 위해서는 여러 번의 REDO 처리를 필요로 하게 된다.

9) 일반적으로 이러한 장치의 오버헤드는 사용 횟수에서 오는 부하가 사용 양에 기인한 오버헤드보다 크다.

Copyright (C) 2005 NuriMedia Co., Ltd.

3.4 객체 캐쉬

Soprano의 객체 캐쉬 관리자는 하나 이상의 객체 캐쉬 뱅크로 구성되어 있다[10]. 객체 캐쉬 관리자는 메모리 공간의 할당 및 반환 등 객체 캐쉬 메모리를 관리하는 객체 캐쉬 뱅크의 서비스를 제공받아 객체의 fix/unfix 연산과 같은 메모리 내에서의 객체 관리를 담당한다.

객체 캐쉬 관리자는 현재 사용중인 모든 메모리 슬롯에 대한 기술자를 유지 관리한다. 이 기술자는 그림 4와 같이 트랜잭션 식별자, 객체 식별자, OBase에 대해서 각각 이중 연결 리스트로써 연결되어 있다.

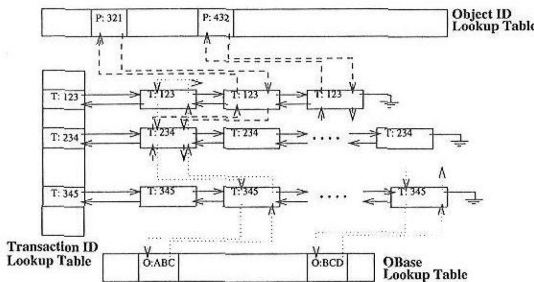


그림 4 사용중인 메모리 슬롯 기술자의 관리

객체 캐쉬 뱅크는 실제 객체를 읽어들일 메모리 공간(이하 RBlock)의 할당 및 반환을 담당하는 모듈이다(그림 5). 이때 RBlock의 할당은 특정 상수 값[11]의 정수배 크기로 이루어진다.

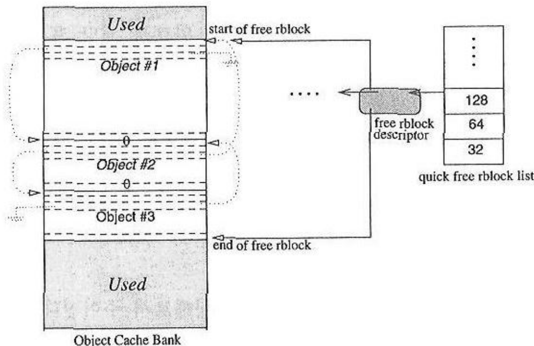


그림 5 객체 캐쉬 뱅크의 구조

미사용중인 RBlock은 각각 하나의 FreeBlkDesc를 가지며, 내부에 존재하는 유효한 객체들은 이중 연결 리스트로 연결한다. 이는 객체가 반환되더라도 캐쉬상에서 제거하지 않고 접근 경로를 남겨둠으로서 캐쉬의 기능을 제공하자는 것이다(그림 5, 6).

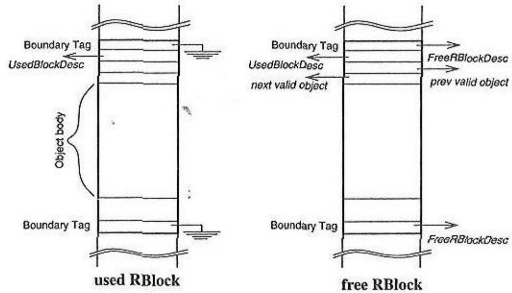


그림 6 RBlock의 구조

RBlock 양끝의 BoundaryTag는 RBlock이 사용중인 경우 NULL 값을 가지며, 미사용중인 경우에는 해당 RBlock의 FreeBlkDesc에 대한 주소 값을 갖는다. 이로써 RBlock이 반환되는 경우 인접한 RBlock의 BoundaryTag상태를 점검하여 미사용 공간의 병합(coalescing)이 가능해 진다. UsedBlockDesc 포인터는 사용중인 RBlock에 대한 UsedBlockDesc을 가리키며, 다음 두 포인터는 미사용중인 RBlock의 경우 RBlock 내에 존재하는 유효한 객체들을 연결한다.

그림 5에 나타난 바와같이 FreeBlkDesc는 미사용 공간의 크기별로 리스트를 형성하여 FreeBlkDescPool를 구성한다. 이는 특정한 크기의 객체 공간을 할당할 때 신속히 최적의 공간을 찾을 수 있으며, 또한 공간의 단편화(fragmentation)도 최소화시킨다.

3.5 트랜잭션 관리

ObjectBase의 경우에는 페이지 캐쉬를 사용하지 않기 때문에, 자체적으로 동시성 제어 모듈과 고장복구을 위한 로그 생성 코드를 갖게 된다. Soprano의 트랜잭션 관리자와 로그 관리자는 ObjectBase를 위해 연산 로깅(operational logging) 기능 및 연산 로깅을 위한 REDO/UNDO 고장복구 기능을 제공한다. 트랜잭션 관리자가 제공하는 REDO/UNDO 고장복구 기법은 ARIES[14] 기법을 사용한다. 이때 ObjectBase가 다루는 데이타는 페이지 캐쉬를 통해 관리되는 데이타와는 완전히 구별되기 때문에 서로 상이한 고장복구 기법으로

10) 객체 캐쉬를 여러 개의 캐쉬 뱅크로 구분하는 이유는 객체 캐쉬의 공유시 캐쉬내의 메모리 단편화를 줄이기 위함이다.

11) 컴파일시에 결정되며 일반적으로 32 바이트 또는 64 바이트로 결정된다.

Copyright (C) 2005 NuriMedia Co., Ltd.

비롯되는 문제점은 발생하지 않는다.

트랜잭션 관리자는 또한 페이지 반환과 같이 정확한 고장 회복이 어려운 연산을 지원하기 위해 지연 연산(pending action) 기능[14]을 제공한다. 그리고 중첩-최상위 연산은 완료하였으나 그 상위 연산이 철회되는 경우에 나타날 수 있는 비일관성 현상을 제거하기 위해 보상 처리(compensation action) 기능을 제공한다[12]. 예를 들면 시스템 디렉토리에 객체 이름을 등록하는 경우, 디렉토리 연산은 중첩-최상위 연산으로 수행하므로 상위 트랜잭션의 철회시 보상 처리를 수행하여 등록된 객체 이름을 다시 삭제한다.

또한 트랜잭션 관리자는 퍼지(fuzzy) 체크 포인트 기능을 제공하여, 체크 포인트 연산 중 시스템이 정지하는 현상을 제거하였다.

서버에서 트랜잭션이 완료되는 경우에는 먼저 페이지 캐쉬의 완료 작업을 수행한다. 페이지 캐쉬의 트랜잭션 완료 작업은 해당 트랜잭션이 변경시킨 모든 페이지의 내용을 로그 장치에 기록하고, 기록된 로그 레코드에 해당하는 LSN[13]을 각 페이지 헤더의 commitLSN에 기록한다. 페이지 헤더의 commitLSN은 가장 최근에 해당 페이지를 변경시킨 트랜잭션이 생성한 로그 레코드의 LSN을 말한다[14]. 이때 만일 페이지가 스왑-아웃되어 있다면, 로그 장치에 기록하기 전에 메모리 페이지를 할당받아 읽어들인다. 페이지 캐쉬에 대한 완료 작업을 수행한 뒤, 트랜잭션 관리자는 지연된 연산을 트랜잭션 완료 로그 레코드에 기록하고, 트랜잭션이 생성한 모든 로그 레코드들을 로그 장치에 저장시킨다. 그리고 나서 지연된 연산들을 수행하고, 트랜잭션이 갖고 있던 모든 로크를 반환한다. 만일 완료된 트랜잭션이 중첩-최상위 연산이라면, 이전 상위 트랜잭션으로 돌아간다. 읽기 전용 트랜잭션(read-only transaction)의 완료 작업을 빠르게 하기 위해서, 트랜잭션 수행 중 로그 레코드를 생성하지 않은 경우에는 완료 로그 레코드의 생성없이 가지고 있던 로크를 반환함으로써 완료 작업을 마친다.

트랜잭션이 철회되는 경우에는 먼저 지연된 연산들을 모두 무효화시키고, 페이지 캐쉬의 철회 작업을 수행한다. 이때 페이지 캐쉬는 트랜잭션이 변경시킨 모든 페이지에 대해서 각 페이지의 commitLSN이 가리키는 로그 레코드를 읽어 이전 상태로 복구한다. commitLSN이 -1인 경우에는 이전 상태가 디스크에 기록되어 있다는 것을 의미하므로, 단순히 페이지 캐쉬에서 해당 페이지를 제거하면 된다. 그리고 나서 트랜잭션 관리자는 지금까지 생성한 로그 레코드를 역순으로 읽어가면서 UNDO 작업을 수행한다[15]. 이러한 작업이 완료되면, 철회 로그 레코드를 생성하고 트랜잭션이 생성한 모든 로그 레코드들을 로그 장치에 플러시시킨다. 그리고 트랜잭션이 가지고 있던 모든 로크를 반환한다.

로그 장치에 일정 양의 로그 레코드가 쌓이게 되면, 트랜잭션 관리자는 체크포인트 데몬을 가동시킨다. Soprano의 체크포인트 작업은 ARIES[14]와 유사하다. 또한 Soprano에서는 체크포인트 작업시 페이지 캐쉬 내에 오랫동안 변경된 상태로 남아 있는 페이지를 강제로 디스크에 저장시킴으로써 한정된 로그 장치를 효과적으로 재사용할 수 있도록 한다.

재가동 고장복구는 ARIES와 같이 3단계를 거친다. 첫번째 단계는 분석 단계로 시스템 고장시 수행 중인 트랜잭션을 알아내고, 비완료된 페이지를 찾는다. 이를 위해 분석 단계에서는 트랜잭션 테이블(TT)과 변경된 페이지 테이블(DPT)을 관리한다. DPT는 REDO가 필요한 페이지의 식별자와 그 페이지의 commitLSN을 기록한다. 두번째 REDO 단계에서는 두 가지 작업을 수행한다. 먼저 일반 페이지 데이타에 대해서는 DPT에 기록된 commitLSN이 가리키는 로그 레코드를 읽어 REDO를 수행한다. 그리고 ObjectBase가 생성한 연산 로그에 대해서는 repeating history[14]를 한다. 마지막 UNDO 단계에서는 ObjectBase가 생성한 연산 로그에 대해서만 UNDO를 수행한다. 자세한 작동 방법은 ARIES 기법과 유사하다.

클라이언트에서의 트랜잭션 관리는 앞서 설명한 서버에서의 트랜잭션 관리와 유사하다. 그 차이점만을 설명하면, 먼저 클라이언트는 트랜잭션 완료시 해당 트랜잭션이 변경시킨 모든 페이지를 완료 로그 레코드에 포함시켜 서버로 전송한다. 그리고 완료된 트랜잭션이 가지고 있던 로크는 서버에게 바로 반환하는 것이 아니라, 데이타와 함께 캐쉬한다. 트랜잭션 철회시에는 트랜잭션의 철회 사실을 서버에게 알리고 캐쉬내의 페이지를 무효화시킨 뒤, 보상 연산을 수행한다.

3.6 메소드 호출

Soprano는 사용자 라이브러리 형태로서 주어진다. 따라서 사용자는 응용 프로그램을 직접 Soprano 라이브러리와 링크하여 사용한다. 이때 응용 프로그램에서 사용하는 모든 지속성 클래스에 대한 메소드가 컴파일시 정

12) 보상 연산은 Soprano 시스템 내부적으로만 사용한다.

13) LSN은 생성된 모든 로그 레코드에 부여되는 유일한 식별자이다.

14) 여기서의 commitLSN은 [13]와는 다르다.

15) UNDO 작업은 ObjectBase에 대해서만 수행한다.

Copyright (C) 2005 NuriMedia Co., Ltd.

적으로 링크되는 경우에는 객체에 대한 메소드 호출이 컴파일러에 의해 이루어지므로 별다른 메소드 관리가 필요하지 않다.

그러나 질의 처리기와 같이 동적으로 메소드 호출을 해야하는 경우에는 메소드 호출을 컴파일러에 의존할 수 없다. 그래서 Soprano에서는 운영체제가 제공하는 동적 라이브러리 기능을 사용하여 동적 메소드 호출을 지원하고 있다. 즉, 데이타베이스에 등록된 클래스의 모든 메소드는 PIC(position independent code)로서 컴파일하여 동적 라이브러리로 만든다. 그리고 실제 메소드를 호출하는 경우 동적 라이브러리 내에서 해당 메소드의 텍스트 위치를 찾아 호출한다. 그러나 이러한 경우에도 모든 메소드의 인자 타입이나 갯수에 대해서 동적으로 정확하게 인자를 전달하는 것은 어렵다. 이를 해결하기 위해 Soprano에서는 클래스 등록시 클래스내의 모든 메소드에 대해서 각 인자와 반환 값을 'void*' 타입으로 변형하여 전달할 수 있도록 대리 함수를 자동으로 생성하고, 이를 동적 라이브러리에 등록한다. 예를 들면, 클래스 A의 메소드 'int f(int, double)'에 대해서는 'void f'(void*, void*, void*, void*)' 를 생성하는데, 이때 함수 f'의 첫번째 인자는 객체에 대한 포인터 값이며, 두번째 인자는 반환 값에 대한 포인터 값이다. 그리고 세번째와 네번째는 각각 메소드 f의 첫번째와 두번째 인자에 대한 포인터이다[16].

3.7 지속성 객체 클래스

Soprano에서 객체의 지속성은 해당 객체가 속한 클래스로부터 결정된다. 즉, 한 클래스의 인스턴스가 지속성 객체가 되기 위해서는 해당 클래스를 시스템이 제공하는 특정 클래스로부터 직접 유도하거나 이로부터 유도된 클래스를 그 상위 클래스로 하면 된다.

PObject 클래스는 바로 객체에 지속성을 제공하기 위해 주어진 시스템 클래스이다. 따라서 Soprano의 모든 지속성 클래스는 PObject 클래스의 하위 클래스이어야 한다. 그러나 지속성 클래스의 인스턴스가 반드시 지속성을 가져야 하는 것은 아니다. 객체를 생성할 때 사용자는 객체를 저장할 데이타베이스를 지정하는 데, 이때 시스템이 제공하는 TransientBase를 데이타베이스로서 제공하면 새로운 객체는 임시적으로만 존재한다[17]. 즉, 임시적 객체는 트랜잭션이 끝남과 동시에 소멸한다.

Soprano에서는 그림 7과 같이 여러 유용한 지속성 클래스들을 제공하고 있다. 먼저 기존의 저장 시스템에서 화일과 인덱스의 개념을 지원하기 위한 FileObject 클래스와 IndexObject 클래스를 제공하고 있는데, FileObject 클래스는 순차적인 데이타의 삽입과 검색을 지원하는 SequentialFile 클래스를 하위 클래스로 하고 있다.

그리고 SequentialFile 클래스의 하위 클래스로는 다루는 데이타의 크기가 가변적인지 아닌지 여부에 따라서 각각 FixedSizeSeqFile 클래스와 VarSize-SeqFile 클래스가 있다. 현재 Soprano에서 지원하고 있는 인덱스 클래스로는 B+Tree 인덱싱을 제공하는 BTreeIndex 클래스가 있다. Soprano의 모든 지속성 객체는 데이타베이스의 물리적 페이지 크기[18]보다 작아야한다.

그러나 Soprano에서는 멀티미디어 데이타와 같이 임의의 길이의 데이타를 자유롭게 저장할 수 있는 LargeObj 클래스를 지원하고 있으며, 집합 및 백, 리스트, 배열 등의 기반 클래스로서 사용할 수 있는 콜렉션 클래스들도 제공한다.

Soprano의 사용자는 시스템에서 제공하는 여러 지속성 클래스들을 직접 이용하거나 PObject 클래스 또는 그 하위 클래스로부터 각 응용 프로그램에서 필요로 하는 클래스를 유도하여 사용하면 된다.

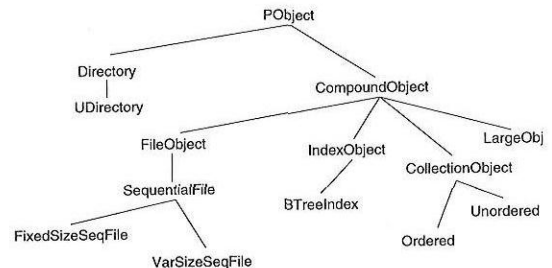


그림 7 PObject 클래스 계층 구조

객체지향 데이타베이스 응용 프로그램에서는 일반적으로 하나 또는 그 이상의 객체를 접근함으로써 데이타베이스 사용을 시작한다. 응용 프로그램은 "root" 객체라 불리는 이러한 객체를 출발점으로 하여 객체에 연결된 다른 객체를 재귀적으로 탐색해 나감으로써 전체 데이타

16) 인자의 갯수에 제한받지 않기 위해서는 인자를 'void*'의 리스트 형식으로 전달해야 한다. 즉, 대리 함수는 'void f'(void*, void*, void*[])'가 된다.

17) TransientBase는 현재 구현중에 있다.

18) Soprano에서 다루는 물리적 페이지의 크기는 컴파일시 결정되며 4K 바이트 또는 8K 바이트로 지정할 수 있다.

Copyright (C) 2005 NuriMedia Co., Ltd.

베이스를 접근한다. 따라서 객체에 이름을 부여하고 부여된 이름을 통해서 객체를 접근할 수 있는 기능은 객체지향 데이타베이스 응용 프로그램에서 데이타베이스 접근을 시작할 수 있도록 해주는 필수적인 기능이다[4]. Soprano는 각 ObjectBase 마다 하나의 시스템 디렉토리 객체를 유지하며, 이를 통해 객체 및 객체 그룹의 이름 서비스와 관련된 여러 기능을 제공한다. Soprano에서는 객체와 객체 그룹에 대해서 각각 별도의 이름 공간을 제공한다. 각 이름 공간 내에서의 이름은 유일해야 하며, 각 객체와 객체 그룹은 1개의 이름만을 가질 수 있다. 시스템 디렉토리에 등록된 이름은 객체나 객체 그룹이 삭제될 때 자동으로 함께 지워진다. 이와는 별도로 각 사용자가 자신의 이름 디렉토리를 유지할 수 있도록 UDirectory라는 지속성 클래스도 지원한다. 사용자 디렉토리는 시스템 디렉토리와는 달리 하나의 객체나 객체 그룹에 여러 개의 이름을 부여할 수 있다. 그러나 객체 이름의 자동적인 삭제는 지원하지 않는다.

4. 관련 연구와의 비교

본 절에서는 지금까지 개발된 주요 객체 관리자들에 대해서 그 특징을 중심으로 Soprano 와 비교 설명하겠다.

Mneme Massachusetts 대학의 Mneme는 데이타베이스 기능을 갖는 지속성 프로그래밍 언어의 저장 시스템을 목표로 구현된 시스템이다[15]. Mneme의 객체는 다양한 응용 프로그래밍과의 접속을 위하여 어떠한 시맨틱이나 타입을 갖지 않는다. Mneme의 객체는 유일한 객체 식별자를 갖는 바이트 스트림으로서 제공된다. Mneme 객체의 식별자는 빠른 객체 접근 및 저장의 효율성을 위해서 일반 메모리 포인터와 같이 32비트 크기를 갖는다. 객체 식별자의 크기 제한으로 인해 객체 식별자는 반드시 하나의 화일 내에서만 유효하다. 여기서 화일이란 관계된 객체들을 묶는 단위가 된다. 만약 다른 화일에 속한 객체를 접근하려면 외부 화일로의 경로를 유지하는 forwarder 객체를 통해 간접적으로 접근해야 한다. 또한 객체의 이름 관리는 화일을 단위로 이루어지는데, 각 화일은 객체 이름을 화일 내의 객체 식별자로 변환하기 위한 테이블을 유지 관리한다.

객체에 대한 접근은 시스템이 제공하는 객체 핸들러를 통해서 접근하는 것을 원칙으로 한다. 객체 핸들러는 직접적인 메모리 포인터를 통한 객체 접근보다는 그 속도가 느리나 시스템의 안정성을 향상시킨다. 이러한 객체 핸들러의 성능 결점을 보완하기 위해 Mneme에서는 직접 지속성 객체를 접근할 수 있도록 메모리 포인터를 제공하기도 한다. 그러나 메모리 포인터를 사용하여 직접 객체를 접근하는 경우에는 사용자가 시스템에 객체의 수정 여부 등 여러 정보를 직접 제공해야하는 불편이 있다. Mneme는 그러나 Soprano 와는 달리 별도의 객체 캐쉬를 관리하지 않기 때문에 효율적인 메모리 자원의 활용이 어렵다.

O2 객체 관리자 O2 OODBMS의 객체 관리자는 WiSS[22]를 저장 시스템으로 하여 메인 메모리와 물리적 저장 장치 사이의 객체 전송을 담당한다[3]. O2의 객체 관리자는 이러한 기본 기능 외에 메소드의 수행, 클라이언트-서버 환경에서의 분산, 인덱싱 및 일부의 질의 최적화, 버전 관리 등의 기능을 제공한다. O2 객체 관리자는 물리적 객체 식별자를 사용하고 있으며, 객체의 이동은 하부 저장 시스템인 WiSS에서 제공하는 레코드 이동 기법을 이용하고 있다. 또한 O2는 새로 생성되는 객체에 대해서는 임시 객체 식별자를 부여하고, 이 임시 객체 식별자는 트랜잭션의 완료시에 실제 객체 식별자로 재 할당된다.

한편 O2의 객체 관리자는 수행의 최적화를 위해서 여러 다양한 수행 모드를 지원한다. 수행 모드에는 개발자를 위한 개발 모드와 일반 사용자가 개발된 응용 프로그램을 실제 사용하기 위한 실행 모드가 있으며, 또한 빠른 프로토타입의 개발을 위한 모드도 제공한다[19]. O2 객체 관리자는 클라이언트 프로세스마다 하나씩의 서버 프로세스가 생성되어 클라이언트의 요구를 처리한다. 그리고 클라이언트에서 서버에게 객체를 요구하면 서버는 객체가 저장되어 있는 페이지내의 모든 객체를 클라이언트에게 보내는 페이지 서버 방식을 채택하고 있다. 그러나 O2 객체 관리자는 Soprano와는 달리 포인터 스위즐링을 지원하고 있지 않기 때문에 모든 객체 접근은 객체 식별자를 통한 해쉬 테이블 검색을 통해 이루어진다.

E Persistent Virtual Machine Exodus[9]를 하부 저장 시스템으로 사용하는 EPVM은 12바이트의 물리적 주소를 객체 식별자로 사용하며 핸들러를 통해 지속성 객체를 접근한다[18]. 그러나 EPVM은 객체 핸들러 중 지역 변수에 한해서만 제한적으로 포인터 스위즐링을 지원하고 있다. 따라서 지속성 객체에 내포된 객체 핸들러의 경우에는 항상 객체 식별자를 통해 객체를 접

19) 빠른 프로토타입을 위한 모드는 앞의 두 모드와는 독립적이다.

Copyright (C) 2005 NuriMedia Co., Ltd.

근해야 한다.

ObServer ObServer는 클라이언트-서버 구조를 가지며 binder라는 프로세스를 통해 클라이언트와 사용하고자하는 데이타베이스를 연결시킨다[11]. ObServer의 객체는 단순한 데이타의 바이트 스트림으로 인식하며, 세그먼트라는 개념을 통해 관계있는 객체들의 집합을 논리적으로 클러스터링시킨다. 세그먼트는 또한 클라이언트와 서버 사이의 데이타 전송 단위가 된다.

ObServer는 하나의 객체를 여러 클러스터링 방법에 따라 저장 할 수 있도록 하기 위해 객체를 복사하여 저장하는 기능을 제공한다. 이러한 기능은 클러스터링의 유연성을 제공하는 한편, 객체의 내용을 갱신에 많은 비용이 든다. 또한 복사된 객체의 식별자를 유지하기 위해서 Duplicate Object Table이라는 별도의 테이블도 관리하고 있다.

Texas Texas는 C++ 언어를 위한 단일 레벨 저장 시스템으로, 하드웨어에 의한 스위즐링 방법을 택하고 있다[19]. 하드웨어에 의한 스위즐링 방식은 기존의 C++ 라이브러리를 수정없이 그대로 사용할 수 있다는 장점과 지속성 객체의 접근 속도가 일반 메모리 객체와 같다는 장점을 갖고 있으나, 다룰 수 있는 데이타베이스의 크기가 가상 메모리 주소 공간에 제한받는다는 단점이 있어, 일반적인 데이타베이스 시스템의 사용하기에는 어렵다.

5. OO1 벤치마크

이번 절에서는 Soprano의 OO1 벤치마크[5] 결과를 소개하겠다.

OO1 벤치마크는 응용 프로그램간 데이타 공유도가 낮은 공학이나 과학 응용 프로그램을 대상으로해서 만들어진 벤치마크이다. OO1 벤치마크는 객체의 삽입과 검색, 그리고 객체간 연결을 따라 객체 데이타베이스를 탐색하는 연산을 측정한다. OO1 벤치마크의 데이타베이스는 Part와 Connection 두 객체로 구성되어 있으며, 각 Part 객체는 Connection 객체를 통해 3개의 Part 객체와 연결되어 있다. 이때 Part 객체 사이의 연결 중 90%는 1% 이내의 가까운 Part 객체와 연결시킨다. OO1 벤치마크는 기본적으로 네트워크를 통한 원격 접근을 요구한다.

본 벤치마크에서는 20,000개의 Part 객체와 60,000개의 Connection 객체를 갖는 데이타베이스에 대해서 Traversal 연산과 Insert 연산을 수행하였다. Traversal 연산은 임의로 선택된 Part 객체를 시작으로 해서 그 객체와 연결된 모든 Part 객체를 재귀적으로 탐색해 나가는 연산이다. 이때 탐색은 7 홉(hop)까지 이루어지며, 총 30번을 수행하였다. Insert 연산은 100개의 Part 객체를 삽입하고 각 Part 객체에 대해서 다른 3개의 Part 객체와 Connection 객체를 통해서 연결시키는 연산이다.

벤치마크 환경은 다음과 같다. 먼저 서버 노드는 Sparc2로 32 MB의 메인 메모리와 1 GB의 디스크를 가지고 있으며, 클라이언트 노드는 16 MB의 메인 메모리와 200 MB의 디스크를 가진 Sparc IPC이다. 운영체제는 모두 SunOS 4.1.3이다. Soprano의 페이지 캐쉬는 서버와 클라이언트 모두 400 KB 크기이며, 클라이언트는 5 MB 크기의 객체 캐쉬를 갖는다.

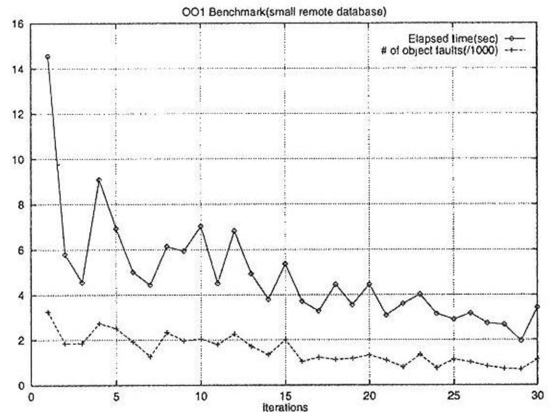


그림 8 Traversal 연산의 결과

먼저 Traversal 연산의 결과는 그림 8과 같다.

이 그래프에서 점선은 각 수행에서 걸린 시간을 표시하고 있으며, 실선은 객체 폴트 횟수(즉, 객체 캐쉬 내에 없는 객체를 접근한 횟수)를 표시한 것이다. 객체 폴트 횟수는 1/1000 값으로 표시하였다. 여기서 Traversal 연산의 수행 시간은 객체 폴트와 비례하여 증감됨을 보여주고 있으며, 따라서 객체 캐쉿의 효과가 탐색 시간의 주요한 결정 요인임을 알 수 있다. 또한 수행 회수가 늘어날수록 캐쉿된 객체가 늘어나므로 탐색에 소요되는 시간은 전반적으로 줄어들게 된다. 특히 캐쉿 효과가 나타나는 시점에서는 3 초이하의 수행 시간을 보여주고 있다. 이는 Traversal 연산의 각 수행 당 3280개의 Part 객체를 접근하므로, 1초에 1000개 이상의 객체를 접근 할 수 있음을 보여주는 것이다. 이러한 결과는 대 다수

Copyright (C) 2005 NuriMedia Co., Ltd.

의 과학이나 공학 응용 프로그램이 요구하는 수준을 만족하는 것이다[5]. 그림에서 그 증감폭이 일정하지 않는 것은 객체들간의 클러스터링과 임의 탐색에 의한 결과이다.

그러나 Insert 연산에서는 기존의 객체를 참조해야 할 필요가 없기 때문에 객체 캐싱 효과는 거의 나타나지 않는다. 그러므로 Insert 연산의 수행 결과(그림 9)에 나타난 바와 같이 콜드-타임과 웜-타임간의 수행 결과는 비슷하다.

	Iterations									
	1	2	3	4	5	6	7	8	9	10
수행 시간	5.83	5.38	5.01	5.40	4.99	5.01	5.00	5 19	4.80	4 81

그림 9 Insert 연산의 결과

6. 결 론

Soprano는 여러 다양한 응용 프로그램에서 사용할 수 있는 고성능 객체 저장 시스템으로 일관된 객체 개념을 지원하고 효율적인 객체 캐쉬 관리 및 스위즐링 등을 통해 빠른 객체 저장 및 접근을 제공한다. 아울러 객체 지향 설계 및 구현을 통한 시스템의 확장성도 갖는다.

Soprano는 모든 것이 객체라는 일관된 객체 개념을 지원함으로써 사용자는 객체 저장 시스템과 사용하는 프로그래밍 언어 사이에 어떠한 개념적 불일치 없이 쉽게 지속성 프로그래밍을 할 수 있다.

Soprano에서는 클라이언트마다 하나의 요구 중개자, 즉 서버 프로세스가 연결되어 클라이언트 프로세서의 요구를 처리하고 그 결과를 반환한다. 같은 노드내의 클라이언트 프로세서는 페이지 캐쉬는 물론 시스템 객체도 함께 공유한다. 또한 소환 데몬을 통해 트랜잭션간 캐싱을 지원하여 클라이언트간 데이타 공유를 최대화함과 동시에 네트워크를 통한 데이타 전송을 최소화하여 시스템의 처리량을 높인다.

Soprano의 페이지 캐쉬는 스왑 기능을 제공하여 REDO-ONLY 고장 회복 방식의 단점을 보완하며, 완전한 객체 캐쉬 관리 기능을 통해 효율적인 메모리의 사용을 제공한다.

앞으로 Soprano의 연구 발전 방향은 페이지 캐쉬 소환 기능을 추가하고, 임시 객체의 저장을 위한 TransientBase 클래스를 제공하는 것이다. 그리고 좀 더 효율적인 객체 캐쉬 관리를 위해서 새로운 객체 캐쉬 관리 알고리즘의 개발도 요구된다.

참 고 문 헌

[1] 박종목, 조완섭, 황규영, "C++ 기반 객체지향 데이타베이스 시스템에서 직교적 지속성을 제공하기 위한 강제계승방법", 정보과학회논문지, 제22권, 제11호, 1995.

[2] 유석인 외, "Obase: 객체지향 데이타베이스 관리 시스템", 정보과학회논문지, 제221권, 제10호, 1994.

[3] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O2*, Morgan Kaufmann Publishers, Inc., 1992.

[4] R. G. G. Cattel, editor. *The Object Database Standard: ODMG-93 Release 1.1*, Morgan Kaufmann Publishers, 1993.

[5] R. G. G. Cattell and J. Skeen, "Object Operations Benchmark," ACM Trans. Database Syst., Vol.17, No.1, Mar. 1992.

[6] D. DeWitt, D. Maier, P. Futtersack, and F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems," In Proc. of the Conf. on VLDB, Brisbane, Australia, Aug. 1990.

[7] A. Eickler, C. A. Gerlhof, and D. Kossmann, "A Performance Evaluation of OID Mapping Techniques," In Proc. of the Conf. on VLDB, Zurich, Switzerland, 1995.

[8] M. A. Ellis and B. Stroustrup. *The Annotated C++ Reference Manual*, Addison Wesley, 1990.

[9] EXODUS Project Group, "EXODUS Storage Manager Architectural Overview," Technical report, Computer Sciences Department University of Wisconsin-Madison, Nov. 1991.

[10] T. Haerder and A. Reuter, "Principles of Transaction-Oriented Database Recovery," ACM Comput. Survey, Vol.15, No.4, Dec. 1983.

[11] M. F. Hornick and S. B. Zdonik, "A Shared, Segmented Memory System for an Object-Oriented Database," ACM Trans. Office Inf. Syst., Vol.5, No.1, Jan. 1987.

[12] S. N. Khoshafian and G. P. Copeland, "Object Identity," In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications(OOPSLA), Sept. 1986.

[13] C. Mohan, "Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems," In Proc. of the Conf. on VLDB, Brisbane, Australia, Aug. 1990.

[14] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging," ACM Trans. Database Syst., Vol.17, No.1, Mar. 1992.

[15] J. E. B. Moss, "Design of the Mneme Persistent

Copyright (C) 2005 NuriMedia Co., Ltd.

Object Store," ACM Trans. Information Syst., Vol.8, No.2, Apr. 1990.

[16] J. E. B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle," IEEE Trans. Softw. Eng., Vol.18, No.8, Aug. 1992.

[17] E. Rahm, "Concurrency and Coherency Control in Database Sharing Systems," Technical Report ZRI 3/91, Universitat Kaiserslautern, Dec. 1991.

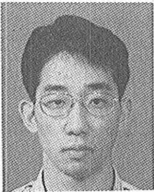
[18] D. Schuh, M. Carey, and D. DeWitt, "Persistence in E Revisited — Implementation Experiences," Technical Report #957, Computer Science Dept., University of Wisconsin-Madison, Aug. 1990.

[19] V. Singhal, S. V. Kakkad, and P. R. Wilson, "Texas: An Efficient, Portable Persistent Store," In Proc. Fifth Int'l. Workshop on Persistent Object Systems, Sept. 1992.

[20] S. J. White, "Pointer Swizzling Techniques for Object-Oriented Database Systems," PhD thesis, University of Wisconsin - Madison, 1994.

[21] S. J. White and D. J. Dewitt, "Implementing Crash Recovery in QuickStore: A Performance Study," In Proc. of the ACM SIGMOD Conf. on Management of Data, 1995.

[22] WiSS Implementation Team, "Wisconsin Storage System Version 3.0," Technical report, Computer Sciences Department University of Wisconsin, June 1985.



안 정 호

1991년 2월 서울대학교 컴퓨터공학과 졸업. 1993년 2월 서울대학교 컴퓨터공학과 졸업(공학석사). 1993년 3월 - 현재 서울대학교 컴퓨터공학과 박사과정. 관심분야는 객체지향 시스템, 데이타베이스.

이 강 우

1991년 2월 서울대학교 계산통계학과 졸업. 1993년 2월 서울대학교 계산통계학과 졸업(이학석사). 1993년 3월 - 현재 서울대학교 전산과학과 박사과정. 관심분야는 트랜잭션 시스템, 객체지향 시스템

송 하 주

1993년 2월 서울대학교 컴퓨터공학과 졸업. 1995년 2월 서울대학교 컴퓨터공학과 졸업(공학석사). 1995년 3월 - 현재 서울대학교 컴퓨터공학과 박사과정. 관심분야는 트랜잭션 시스템, 객체지향 시스템

김 형 주

1982년 2월 서울대학교 컴퓨터공학과 졸업. 1985년 8월 Univ. of Texas at Austin 전자계산학 석사. 1988년 5월 Univ. of Texas at Austin 전자계산학 박사. 1988년 5월 - 9월 Univ. of Texas at Austin Post-Doc. 1988년 9월 - 1990년 12월 Georgia Institute of Technology 조교수. 1991년 1월 - 현재 서울대학교 컴퓨터공학과 부교수. 관심분야는 객체지향 시스템, 사용자 인터페이스, 데이타베이스

Copyright (C) 2005 NuriMedia Co., Ltd.