Electro-21 객체지향 데이터베이스 시스템의 객체관리기 설계 및 구현*

(The Design and Implementation of Object Manager in Electro-21 ODBMS)

송 하 주† 이 상 원‡ 김 형 주§

May 6, 1998

요약

객체지향 데이터베이스 시스템은 기존의 데이터베이스 시스템들과는 달리 시스템 수준에서 객체지향 개념을 지원하기 때문에 데이터들간의 관계가 주를 이루는 디자인 응용프로그램을 개발하기에 용이할 뿐아니라 객체식별자를 이용 이들 객체들을 고속으로 탐색할 수 있게 한다.

객체관리기는 객체지향 데이터베이스 시스템의 서브시스템으로서 응용프로그램 수준에서 객체 개념에 의한 프로그램이 가능하게 한다. 본 논문은 ODMG'93의 C++인터페이스를 따르는 객체지향 데이터베이스를 개발함에 있어서 객체관리기 시스템을 객체지향 디자인 방법으로 설계하고 C++언어를 사용하여 구현함으로서 확장성과 재사용성을 높이고 C++언어에 적합한 객체캐쉬 구조를 사용하여 고속의 객체 접근이 가능하도록 하였다.

주요어: 객체지향 데이터베이스, 객체관리기, 객체식별자, 확장성, 재사용성, ODMG

Abstract

Powerful semantics and various advanced functionalities of the object-oriented database management system(OODBMS) provide the desirable storage system for the design-oriented systems such as CAD, CAE, CIM and CASE. It is the object manager (a subsystem of the OODBMS) that provides the users with the uniform view of objects and their relationships. In this paper, we represent a design and implementation of an object manager which has the characteristics of high performance, extensibility, and ODMG-compliance.

*상공부 Electro-21 프로젝트의 일환으로 객체지향 데이터베이스(ODBMS) 개발을 목적으로 한다. 기업체로서 현대전자, 동양 SHL이 위탁연구 기관으로서 서울대학교 컴퓨터공학과 객체지향 시스템 연구실과 KAIST 데이터베이스 시스템 연구실이 참여하고 있다.

†서울대학교 컴퓨터공학과 박사과정

‡서울대학교 컴퓨터공학과 박사과정

§서울대학교 컴퓨터공학과 부교수

1 서론

1.1 객체지향 데이터베이스 시스템의 등장

컴퓨터의 발달과 더불어 은행업무나 기업의 회계업무와 같은 비교적 단순한 형태의 데이터를 대량으로 취급하는 분야이외에 CASE, CAM, CAD, AI 응용 프로그램 그리고 근래들어 각광받는 멀티미디어 등에 광범위하게 컴퓨터가 사용되게 되었다. [Cat91, Kim90] 이들 시스템은 저장 시스템으로서 주로 화일 시스템을 사용해 왔었으나 점차 데이터베이스 시스템에 의해 대체되는 추세이다. 그러나 기존 관계형 데이터베이스 시스템은 단순한 레코드들의 모임인 테이블 개념과 값에의한 접근 [KS91] 방법만을 제공하기 때문에 이와 같은 응용프로그램이 요구하는 데이터 모델이나 데이터간의 관계를 지원해 주기는 역부족이며 온라인 트랜잭션만을 대상으로 설계되어온 관계형 데이터베이스 시스템의 동시성 제어 메카니즘은 이들 시스템이 사용되는 환경의 특징인 장기간에 걸친 트랜잭션의 처리에는 불합리하다. 따라서 차세대 데이터베이스 시스템은 위와 같은 응용분야를 고려하여 설계되어야 하는데 이에 필요한 특징들을 간추리면 다음과 같다.

- 서로 다른 객체들의 조합에 의해 객체를 표현하는 능력 즉 복합 객체의 지원
- 가변적이며 임의의 크기를 가지는 데이터의 저장과 검색 기능
- 다양한 데이터 타입을 사용자가 정의하고 사용할 수 있는 기능, 또한 이들을 응용프로그램에서 모델간의 불일치(impedance mismatch)가 없이 자연스럽게 사용할 수 있는 기능
- 객체또는 데이터베이스 스키마의 버젼 기능
- 지식추론 지원
- 장기간에 걸친 협동작업을 위한 트랜잭션 지원

현재 차세대 데이터베이스 시스템으로서 객체지향 데이터베이스 시스템(object database system), 추론 데이터베이스(deductive database), 시간지원 데이터베이스(temporal database)등이 주목받고 있다. 객체지향 데이터베이스 시스템는 그 중에서도 위의 기능들을 가장 잘 구현하고 있다. 객체지향 데이터베이스 시스템은 상태를 나타내는 속성(attribute)과 작동을 나타내는 메소스(method)로 이루어진 객체(object), 그리고 공통된 속성과 메소드를 가지는 객체를 생성하기 위한 클래스(class), 클래스들간의 계승(inheritance), 그리고 객체들을 구별하기 위한 객체식별자(object identifier), 객체의 메소드를 호출 실행하기 위한 메시지 등의 개념을 중심으로 구성된다. 또한 이와 같은 중심 개념에 외에 복합객체(composite object), 버젼(version) 등의 추가된 개념을 지원함으로서 객체지향데이터베이스 시스템은 위에서 언급된 디자인 시스템이나 멀티미디어에 적합한 기능을 제공한다.

위와 같은 객체지향 데이터베이스 시스템의 특징에도 불구하고 최근까지 공통된 데이터모델이 없었으며 이는 또한 객체지향 데이터베이스 시스템의 성장의 장해요인 이었다. 이로인해 공통된 데이터모델의 정립을 위한 다각적인 노력이 있어왔으며 Object Data Management Group이 이에 상당히 근접한 안을 발표하게 되었다. [Cat94] 이는 비록 참여한 기존 ODBMS 업체들의 모델을 적당히 절충한 것에 불과하다는 비판을 받기도 했지만 점차 학계와 업계의 표준으로 자리잡아 가고 있다.

1.2 ODBM C++ 바인딩

ODMG는 표준 데이터 모델뿐만 아니라 적용 언어별로 각각의 ODL(Object Definition Language)과 OML(Object Manipulation Language)을 제시하고 있는데 본 시스템은 C++바인딩을 따르고 있다. 여기에서는 객체가 지속성을 가지기 위해서는 시스템에 의해 미리 정의된 Persistent_Object 클래스로부터 계승을 받은 클래스의 인스턴스일 경우만 가능하다.

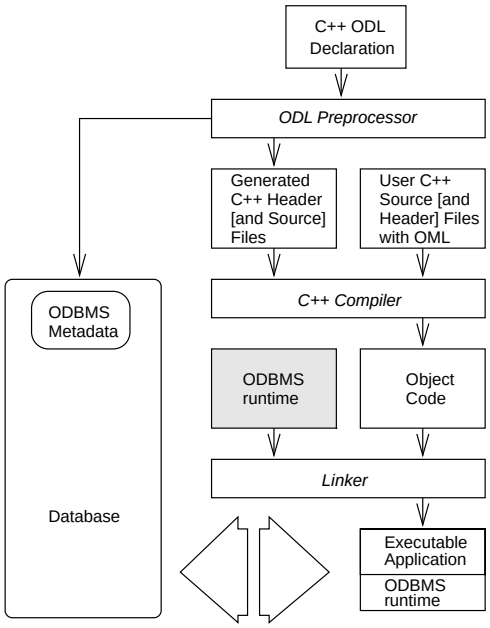

Figure 1: ODMG C++ 바인딩

그림 1은 C++바인딩을 이용해서 객체지향 데이터베이스 시스템상에서 응용프로그램을 작성하는 절차를 보인다. C++ODL로 정의된 지속성 클래스는 ODL전처리기에 의하여 데이터베이스에 해당 클래스의 스키마를 반영하며 응용프로그램들이 사용할 수 있도록 C++용 헤더화일을 생성한다. 응용프로그램은 클래스 라이브러리인 Ref 변수를 이용하여 객체를 생성하거나 삭제, 또는 변경을 가할 수 있다.

1.3 객체관리기의 기능

객체 관리기는 ODBMS의 하위 시스템의 하나이며 저장시스템과 밀접하게 결합되어 하나의 모듈로 통합되거나 서로 독자적인 모듈을 형성하기도 한다. 그 주요한 기능은 결합될 응용프로그램이 필요로 하는 모든 객체 개념과 관련한 기능을 수행하는 것으로 구체적으로 객체 버퍼링, 객체 접근 방법제공, 객체식별자관리 등이며 빠른 객체 접근을 위한 포인터 스위즐링, 객체를 직접적으로 접근하기 위한 수단인 객체 이름관리를 들 수 있다. 다음은 이들 중 중요한 개념만을 간략하게 나타내었다. [Cat91, Pop91]

- 객체식별자(object identifier)
  시스템내의 수많은 객체들 중 한 객체를 지정하는 수단이다. [KC86] 이는 사람에게 출생시 주어지는 주민등록번호와도 같은 것으로 객체의 생성시에 시스템에의하여 부여되며 보통 객체가 삭제되더라도 삭제된 사실을 기록할 뿐 다른 객체를 위하여 재사용되지는 않는다. 이는 또한 객체들을 구분하는 수단이된다. 기존의 관계데이터베이스는 레코드를 그 값에 따라 구분하였지만 객체지향 데이터베이스 시스템에서는 값에 의한 구분보다는 식별자를 통해 구분된다. 객체식별자는 객체에 대한 접근이 이루어질 때마다 사용되기 때문에 시스템의 특징이나 성능면에서 크게 영향을 미치게 되는데 구현 방법에 따라 물리적위치법(physical OID), 구조화된 주소법(structured physical OID), 서로게이트(surrogate or logical OID), 타입서로게이트(typed surrogate)등이 있는데 이들은 그 생성방법이나 객체 접근 성능면에서 차이를 보인다.

- 객체 버퍼링(object buffering)
  일반적으로 RDBMS의 응용 프로그램은 필요한 레코드들을 자신의 메모리 영역으로 복사한 후 사용하는데 반해 ODBMS에서는 응용프로그램에서 사용하는 객체들을 시스템 수준에서 관리하므

로 별도의 객체버퍼(객체캐쉬)를 필요로 한다. [WD92] 따라서 객체관리기는 응용프로그램에대해서는 요구를 파악하여 필요한 경우 해당 객체를 객체캐쉬에 적재하고 이를 접근할 수 있는 방법을 제공하며 캐쉬메모리의 사용과 반환에 따른 메모리 관리를 수행한다.

- 포인터 스위즐(pointer swizzle)
  객체캐쉬내의 객체를 접근하려면 객체식별자 해쉬테이블을 거쳐 해당 객체의 위치를 찾게 된다. 따라서 객체에 대한 첫번째 접근시에 객체식별자 대신 해당 객체의 메모리내 위치(주소)로 대치시키면 이후에는 해쉬테이블을 거치지 않고 객체를 바로 찾을 수 있으므로 빠른 객체접근이 가능하다. 이러한 기능을 포인터 스위즐이라 하며 자주 접근되는 객체식별자의 경우 수행성능의 향상을 기대할 수 있다. [Mos92, WD92]

- 객체 이름 관리(object naming)
  특정 객체를 사용자에 의해 주어진 이름을 통해 접근할 수 있게 하며 이는 데이터베이스에 접근하기 위한 시스템 진입점(entry point)이 된다. 객체관리기는 주어진 이름과 객체식별자와의 대응관계를 유지하며 기타 이름의 변경이나 삭제에 따른 정보를 유지한다.

- 객체 클러스터링(object clustering)
  동시에 자주 사용되는 객체들 끼리는 물리적으로도 서로 인접하게 저장하도록 해야 한다. 이는 이들 객체를 한 페이지내에 또는 디스크의 동일한 실린더상에 존재하는 페이지에 저장되도록 한다.

2 객체관리기의 설계와 구현

2.1 Electro-21 ODBMS 실행모듈의 구조

그림 2은 객체지향 데이터베이스 런타임(runtime library)과 결합된 응용프로그램의 구조를 나타내었다. 객체관리기는 라이브러리 형태로 제공되며 응용프로그램과 저장시스템의 사이에 존재한다. 클라이언트-서버형으로 확장되는 경우에는 객체관리기와 저장시스템사이에 통신모듈이 삽입된다.

응용프로그램은 객체 캐쉬에 적재된 객체를 **Ref** 변수를 통해 접근할 수 있다. 저장시스템으로서 ODS시스템[1]을 사용한다. 저장시스템은 실재 객체의 저장과 동시성제어(concurrency control), 고장회복(recovery)를 담당하며 이름관리를 위한 별도의 인덱스 인터페이스를 제공한다[2] 따라서 시스템 전체는 그림 2와 같이 응용프로그램층, 객체관리기 층, 저장시스템 층의 세단계로 이루어지며 응용프로그램은 객체관리기 층을 통해 모든 데이터베이스 관련 서비스를 받도록 하였다. 물론 이들 인터페이스들을 사용자가 바로 사용할 수도 있지만 시스템의 일관성, 안정성 그리고 사용상의 편의를 위해 **Database** , **Transaction** , **Ref** , **RefAny** , **Set(collection object)** 등과 같은 별도의 시스템 클래스들을 제공하고 있다. 다음에 이어질 장들에서는 객체관리자의 구조 및 중요한 시스템 클래스에 대하여 설명한다.

2.2 객체식별자

본 시스템은 다중 데이터베이스지원, 물리적인 위치와의 독립성과 타입정보의 이용을 위해서 타입화된 서로게이트형의 객체식별자를 사용했다. 객체식별자는 **SOID** 라는 이름의 클래스로 구현되며 다음과 같은 멤버를 갖는다.

- 제어 바이트(1 바이트)
  - transientobj(1 비트)

[1] KAIST 데이터베이스 연구실에서 개발한 다중사용자용의 저장시스템
[2] 일반적으로 관계형 데이터베이스 시스템에서는 이에 대한 명시적인 인터페이스가 요구되지 않는다.

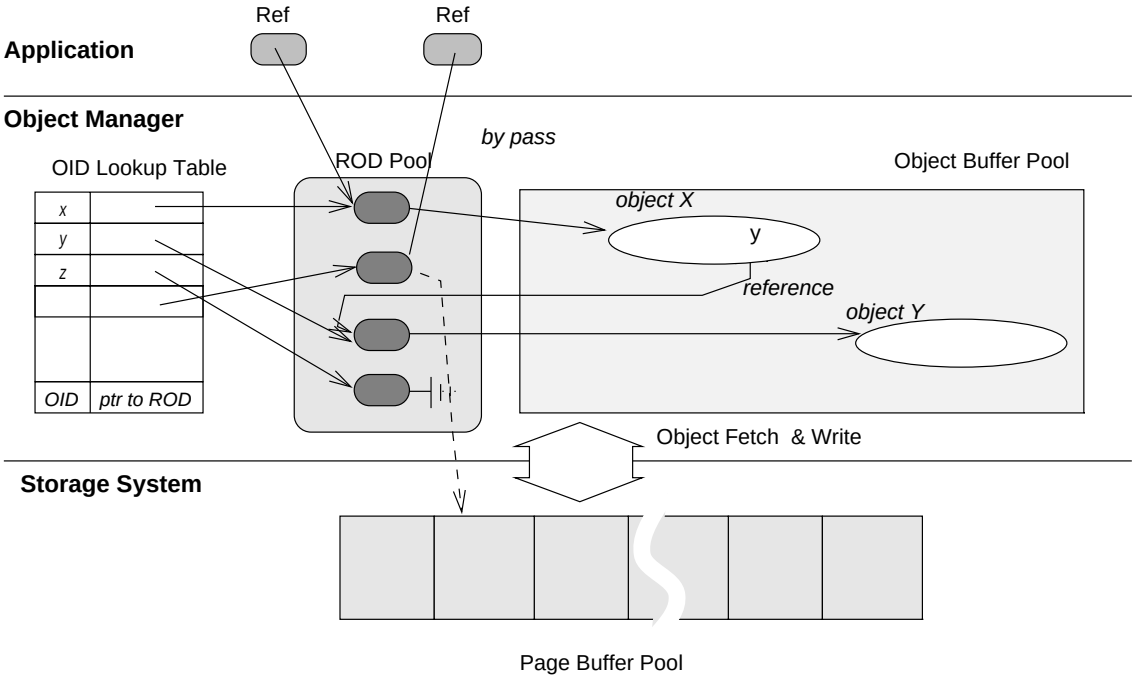

Figure 2: 응용프로그램에 라이브러리 형태로 결합된 객체관리자

  - swizzled(1 비트)
  - padd(6 비트) : 미사용

- **dbid** : 데이터베이스 식별자(1 바이트)
- **classid** : 클래스 식별자(2 바이트)
- **tid** : 인스턴스 식별자(4 바이트)

여기서 **transentobj** 는 현재 참조하고 있는 객체가 지속성 객체인지 아니면 임시 객체인지를 나타내며 **swizzled** 는 객체식별자가 스위즐 되었는지를 나타낸다. **dbid** 는 객체가 생성된(존재하는) 데이터베이스의 식별자이며, **classid** 는 객체의 클래스, **tid** 는 클래스내에서 객체의 일련번호를 나타낸다. 따라서 본 시스템은 최대 256개의 데이터베이스에 존재하는 객체를 동시에 사용할 있으며 각 데이터베이스내에는 최대 65556개의 클래스를 가질 수 있다. 그리고 각 클래스는 최대 2^{32}개의 인스턴스를 가질 수 있다.[3] 객체식별자의 생성, 삭제, 물리적위치와의 대응은 저장시스템이 담당한다.

2.3 객체구조

객체관리기는 크게 자신이 다루고 있는 객체의 내부 구조를 인식할 수 있는 형태의 것과 그렇지 않은 것으로 나뉘는데 전자의 경우 객체단위의 인터페이스 외에 객체내 속성(attribute) 단위의 제거, 이동, 생성과 같은 기능을 제공할 수 있다. 후자의 경우를 바이트 서버라고도 하는데 이는 전자보다도 낮은 수준의 객체관련 시멘틱을 제공하지만 그때문에 다양한 객체지향 데이터베이스 시스템의 서브시스템으로 응용될 수 있다.

본 객체관리기는 응용프로그램과 라이브러리 형태로 결합되기 때문에 객체 내부 구조의 해석은 컴파일러나 스키마관리기와 같은 별도의 모듈이 담당하며 응용 프로그램과 객체관리기 사이에 별도의 언

[3] 이 것은 모두 이론적으로 가능한 수치이며 객체의 크기와 OS상의 제한을 고려하면 이보다 크게 줄어든다.

어 시스템이 존재하지 않기 때문에 바이트서버형을 택했다. 따라서 객체는 디스크나 메모리상에서 모두 동일한 C++ 객체 포맷(object format)을 유지해야 한다. 따라서 모든 객체는 반드시 동일한 언어, 컴파일러, 동종의 CPU를 사용한 동일한 시스템에 의해서만 사용할 수 있다는 단점을 가진다. 반면 전처리기(preprocessor)를 사용하지 않고서도 클래스 라이브러리만(class library)으로서도 응용프로그램을 작성할 수 있으며 다른 객체지향 데이터베이스 시스템에도 쉽게 이식될 수 있고 빠른 객체접근이 가능하다.

2.4 객체관리기(ObjectManager)

객체관리기는 SysObjectManager라는 이름의 전역변수로 정의되며 내부적으로 객체관련 기능의 객체캐쉬 관련 모듈, 데이터베이스 관련 인터페이스 모듈, 트랜잭션 관련 인터페이스 모듈, 이름관리 관련모듈의 4가지 모듈로 이루어진다. 이들 중 객체캐쉬 관련 모듈은 별도의 객체캐쉬 클래스를 통해 구현되는데 이는 객체의 적재와 저장, 접근방법제공에 관한 기능을 담당하며 세부 구조는 다음 장에서 설명된다. 데이터베이스 관련 모듈은 데이터베이스의 개방과 폐쇄에 관련한 기능과 이에 따른 시스템 상태정보의 기록을 담당한다. 트랜잭션 관련 인터페이스 모듈은 트랜잭션의 시작과 완료, 중단에 관련한 기능을 담당한다. 이름관리 모듈에 이름관리를 위한 화일의 입출력을 담당한다. 그림 3은 객체관리기를 포함한 시스템의 전반적인 제어구조이며 표 1는 객체관리기의 주요한 인터페이스이다.

2.5 객체캐쉬(ObjectCache)

일반객체[4]에 대한 접근 수단을 제공한다. 응용 프로그램[5] 상에서 디스크에 저장된 객체를 접근하기 위해서는 객체를 캐쉬 메모리에 적재하고 적절한 접근 방법을 제공해야 한다. 캐쉬의 크기는 시스템의 구동시에 결정된다. 미사용 캐쉬메모리[6]의 할당은 객체 단위로 이루어지며 연속적인 메모리 영역을 가지게된다. C++ 언어와 결합된다는 점과 시스템 부하의 분산을 위하여 쓰레기수집(garbage collection)에 의한 메모리관리는 수행하지 않는다.

캐쉬는 객체식별자 해쉬테이블과 ROD풀(Resident Object Descriptor Pool), 캐쉬뱅크로 이루어진다. 해쉬테이블은 SOIDLookUpTbl과 DatabaseLookUpTbl의두가지가 존재하는데 전자는 객체식별자를 통해 만일 해당 객체대한 ROD(Resient Object Descriptor)를 반환하고 후자는 ROD를 데이터베이스별로 해쉬된 리스트를 유지하여 데이터베이스가 폐쇄되는 경우 해당 데이터베이스에 이 리스트를 탐색하면서 해당 객체를 디스크에 저장한다. 각 ROD는 캐쉬된 객체에 대하여 하나씩 존재하며 객체식별자, 캐쉬내의 위치, 크기, 접근빈도, 스위즐 여부, 갱신 여부등과 같은 객체의 상태 정보를 유지한다. 트랜잭션이 종료되는 경우 갱신되었거나 새로이 생성된 객체는 모두 디스크로 저장되며 캐쉬상에서는 무효화 된다.[7] 이는 시스템이 클라언트 프로세스마다 독립된 캐쉬를 가지며 동시성제어가 서버에서만 수행되는 분산형으로 확장될 경우에 대비하기 위함이다.

2.6 객체캐쉬뱅크(ObjectCacheBank)

객체캐쉬뱅크는 객체캐쉬로 객체를 적재할 메모리의 관리를 담당한다. 하나의 ObjectCacheBank는 객체가 적재될 공간인 캐쉬메모리와 미사용중인 캐쉬메모리에 대한 정보를 기록하는 FreeCBlockList로 이루어진다

캐쉬메모리는 AUNIT[8]의 정수배로 할당되는데 이를 CBlock이라 한다. 미사용중인 CBlock은 각각 하나의 FreeBlkDesc를 가지며 내부에 존재하는 유효한 객체들 간에는 이중 링크에 의하여 연결된다. 이는 객체가 반환되더라도 버퍼상에서 제거하지 않고 접근경로를 남겨둠으로서 캐슁기능을 수행한다.

[4] 한 페이지 크기보다 큰 객체인 거대객체(large object)와 구분하기 위해 붙인 이름이다.
[5] 스키마관리기, 질의어 처리기, 사용자 프로그램과 같은 객체관리기를 사용하는 상위 모듈을 뜻한다.
[6] 버퍼의 메모리중 객체를 적재하고 있지 않은 부분, 이는 아직 할당되지 않았거나 반환된 메모리이다.
[7] intra-transaction chache라고도 한다.
[8] 컴파일시에 결정되며 일반적으로 32 또는 64바이트의 크기이다. 즉 1 AUNIT == 32 바이트

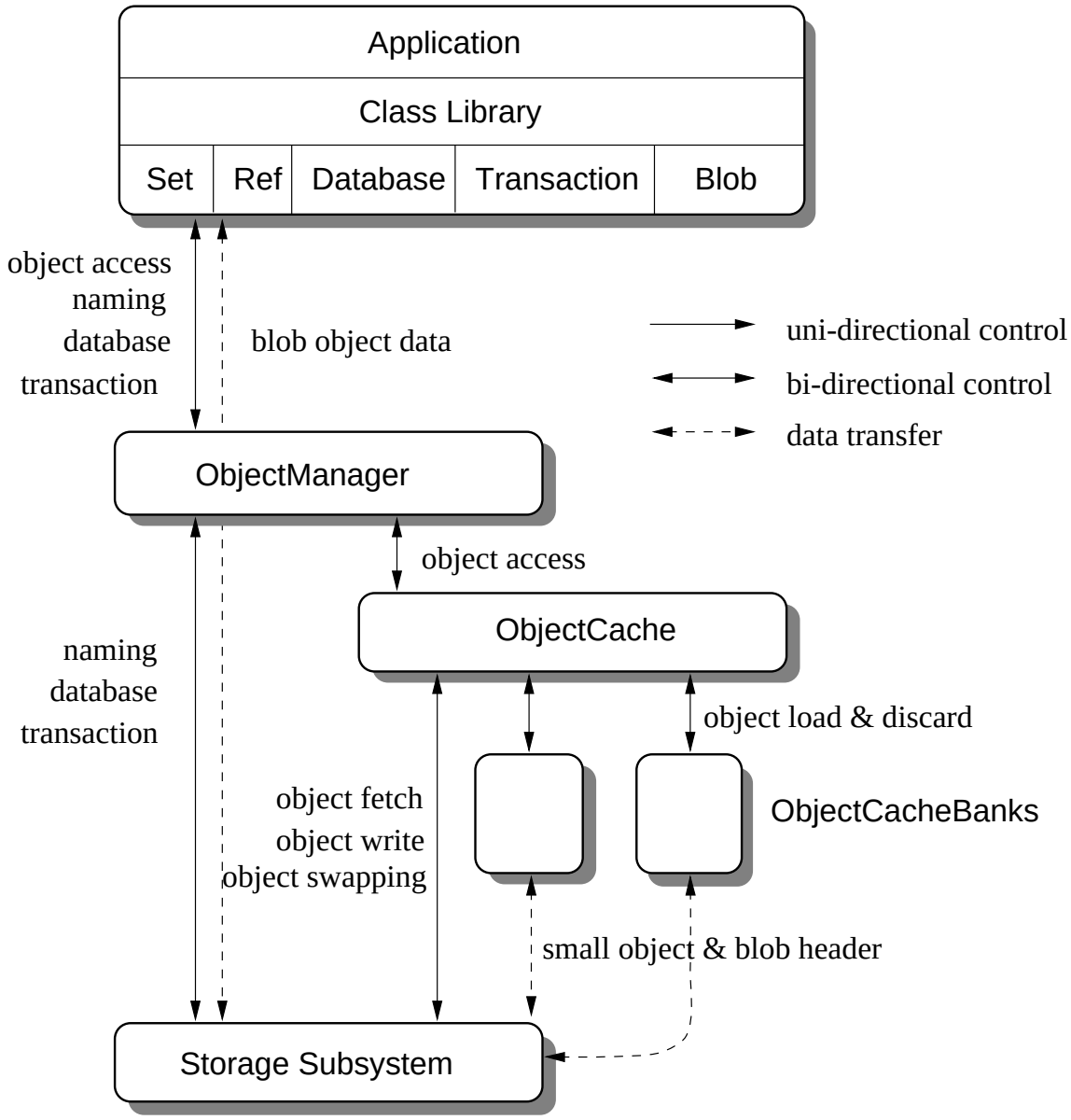

Figure 3: 시스템 제어 구조

인터페이스	동작
`opendb dbname()`	*dbname* 의 데이터베이스를 개방한다.
`closedb dbid()`	*dbid* 식별자를 가진 데이터베이스를 폐쇄한다.
`begintrans mode()`	*mode* 로 트랜잭션을 시작한다.
`endtrans()`	트랜잭션이 완료되었음을 알린다.
`rollback()`	트랜잭션이 취소되었음을 알린다.
`isintrans()`	현재 트랜잭션이 진행중임을 알려준다.
`map dbid(oid objname)`	*oid* 객체에 *objname* 의 이름을 부여한다.
`name dbid(objname oid)`	*objname* 에 해당하는 객체식별자를 알려준다.
`(unmap , dbid , objname)`	*objname* 이름을 데이터베이스에서 삭제한다. 이름에 해당하는 객체는 삭제되지 않는다.
`remap dbid(oldname newname)`	객체의 이름을 *oldname* 에서 *newname* 으로 변경한다.
`createobj dbid(cname size)` *udesc*)	*cname* 타입의 지속성 객체를 생성한다.
`createobj cobj(cname size)` *udesc*)	*cobj* 지속성 객체를 *cobj* 에 인접하게 생성한다.
`destroy udesc()`	*udesc* 가 가리키는 객체를 데이터베이스에서 삭제한다.
`destroy soid()`	객체식별자 *soid* 에 해당하는 객체를 데이터베이스에서 삭제한다.
`(fixobj , oid , size)`	객체식별자 *oid* 에 해당하는 객체를 객체 캐쉬에 적재한다.
`unfixobj udesc()`	*udesc* 가 가리키는 객체가 차지하고 있는 캐쉬메모리를 반환한다.
`touchobj udesc()`	객체의 멤버값을 변경될 것임을 알린다.
`isincache addr()`	*addr* 가 캐쉬상의 주소인지를 확인한다.
`iscached soid()`	객체가 캐쉬상에 존재하는 지를 확인한다.

Table 1: 객체관리기의 인터페이스

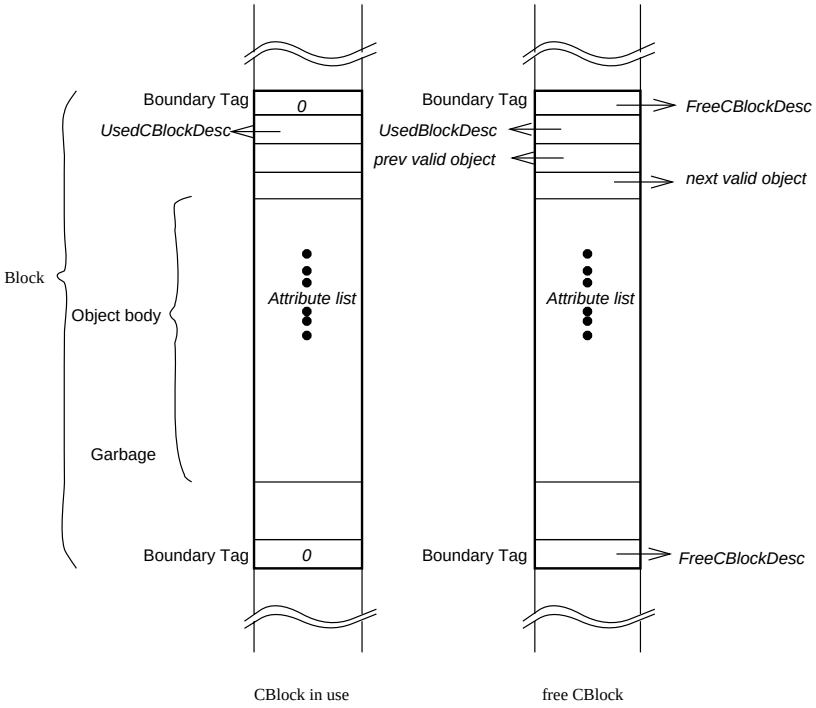

Figure 4: CBlock의 구조

그림 4은 사용중인 CBlock과 미사용중인 CBlock의 구조를 보인다. CBlock 양끝의 Boundary Tag는 CBlock이 사용중인 경우 0값을, 미사용중인 경우 해당 CBlock의 FreeBlkDesc에 대한 포인터로 사용된다. 이는 CBlock이 free 되는 경우 인접한 CBlock의 Boundary tag상태를 점검하여 Coleascing을 가능하게 한다. ROD 포인터는 사용중인 CBlock이나 두 포인터는 미사용중인 CBlock의 경우 CBlock내에 존재하는 유효한 객체들을 연결한다. 유효 객체 연결 리스트는 FreeBlkDesc에 의하여 접근 가능하다.

FreeBlkDesc는 미사용의 CBlock에 대해 하나씩 존재하며 CBlock의 크기, CBlock내의 유효 객체 리스트에 대한 포인터등을 유지한다. FreeBlkDesc는 FreeCBlock의 크기별로 리스트를 형성하고 FreeCBlock이 재사용되거나, 분리가 이루어지면 리스트에서 삭제 또는 다른 크기의 리스트로 이동이 이루어진다.

2.7 객체 이름 관리

객체지향 데이터베이스에서는 모든 객체는 고유한 식별자를 가지며 이를 통해서 접근이 가능하다. 따라서 응용 프로그램이 객체를 사용하기 위해서는 먼저 접근하고자 하는 객체의 식별자를 알아내야 하는데 이는 보통 먼저 접근한 객체로 부터 얻을 수 있다. 그러나 아직 접근한 객체가 전혀 없는 경우 즉 데이터 베이스를 개방하고 처음으로 객체를 접근하고자 하는 경우에는 사용자가 직접 객체식별자를 지정해야 하는데 이것을 응용프로그램 내에 일일이 저장하는 것은 프로그램의 확장성이나 응용프로그래머의 입장에서 많는 제약점을 가져온다. 따라서 진입 객체(entry point object)에 대한 식별자를 좀더 유연하고 사용자가 인식하기에 편리한 표현에 의해 알아낼 수 있는 방법이 필요하며 이에 적합한 방법이 특정

객체에 사용자가 지정한 문자열 즉 이름을 대응 시키는 것이다 이를 도식화하여 나타내면 다음과 같다.

이름관리 : 객체 이름 → 객체식별자

이 방법에 의하면 응용프로그래머는 데이터베이스 탐색의 시작이 될 몇몇 객체에 이름을 부여하고 이후의 접근은 객체의 이름을 이용, 진입객체의 식별자를 찾아 이 객체를 접근한 후 다시 이것에 의해 참조되는 객체를 차례로 방문하는 형태를 취한다. 이름과 객체식별자와의 대응을 저장하기 위하여 시스템에서는 내부적으로 인덱스를 이용한다.

2.8 은닉된 포인터의 처리

C++의 객체내에는 프로그래머가 정의한 멤버외에 컴파일러에 의하여 사용자에게는 명시적으로 나타나지는 않지만 삽입되는 포인터 변수들이 있다. 이들은 각각 가상함수(virtual function) 기능과 가상상위클래스(virtual base class) 개념을 지원하기 위한 자료 구조이다. 이들은 모두 포인터로서 메모리 주소를 그 값으로 가진다. 이들 포인터는 해당 객체를 생성한 프로세스가 끝남과 동시에 그 값은 무효화된다. 따라서 다른 프로세스가 이 객체를 사용하는 경우 은닉된 포인터는 오작동을 유발하게 된다. 그러므로 은닉된 포인터를 가진 객체를 디스크에서 메모리로 읽어 들일 경우에는 이들 포인터의 값을 초기화 시키는 과정을 거쳐야 한다.

본 시스템에서는 시스템에 새로운 모듈을 첨가하지 않고서도 간결하게 이를 해결하기 위하여 `new` 연산자를 사용한 은닉 포인터 처리법을 사용하였다. [BDG93] 이는 `new` 연산자 호출이 새로 생성되는 객체의 생성자(constructor)를 호출하는 점을 이용한 것으로 자세한 것은 다음과 같다.

```
// new operator overloading
inline void* new(__HIDDENFIX__* obj) { return obj; }
  .....
// tap has start address of object body
hiddenfix = !0;
new((__HIDDENFIX__*) tap) TA;
  .....
```

`tap`는 객체 캐쉬에 적재된 객체를 가리키는 포인터이다. `new` 연산자 중첩에 의해 `new` 는 `tap`이 가진 주소값을 그대로 반환한다. 여기서 컴파일러는 반환된 포인터 값에 대해 해당 객체의 디폴트 생성자를 호출한다. 호출된 디폴트 생성자는 해당 객체 내에 존재하는 은닉 포인터들의 값을 초기화한다. 이때 주의할 것은 사용자 정의의 디폴트 생성자가 존재하는 경우 전역변수로 정의된 **hiddenfix** 변수를 검사하여 이 변수가 세트된 경우에는 자신이 사용자에 의해 정의된 코드가 수행되지 않도록 해주어야만 한다.[9] 이 방법은 O++ 시스템에서 사용하는 것으로서 간단한 방법으로 은닉포인터를 교정할 수 있는 반면 객체를 가리키는 Ref 타입의 포인터와 실재 객체는 반드시 그 타입이 동일해야 하는 제약점을 가지기 때문에 C++ 본래의 가상함수 기능을 충분히 살리지 못하는 단점이 있다.

2.9 시스템 정의 클래스

p_Object 클래스

일반적으로 객체의 지속성을 가리는 방법은 두 가지 방법이 존재한다. 첫째 방법은 클래스 계승구조상에서 객체의 지속성 여부를 결정짓는다. 즉 한 클래스의 인스턴스가 지속성 객체가 되기 위해서는 해당 클래스가 시스템의 정의한 특정 클래스로 부터 직접 계승을 받거나 이로 부터 계승의 받은 클래스로 부

[9] 데이터 정의 모듈에 대해 전처리기나 프로그래머에 의하여 수행된다.

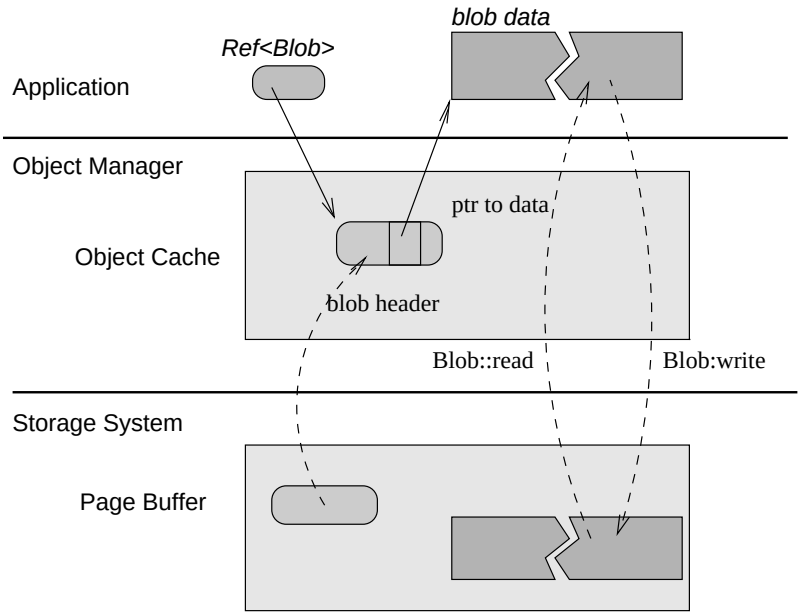

Figure 5: Blob의 입출력

터 계승을 받는 방법이다. 또다른 방법은 일단 지속성을 가지는 객체가 생성되면 이 객체로 부터 참조가 능한 모든 객체를 지속성 객체로 만드는 것이다.

ODMG의 모델은 전자의 형태를 취하여 클래스가 지속성을 가지기 위해서는 반드시 p_Object로 부터 직접 상속받거나 이로 부터 상속받은 클래스로 부터 상속을 받아야한다. 즉 p_Object의 하위 클래스가 되어야 한다. 그러나 지속성 클래스의 인스턴스라고 해서 모두 지속성을 가지는 것은 아니며 이것 또한 Free Store(또는 Heap)이 아닌 객체 캐쉬에 객체를 생성해야만 지속성을 가지는 객체가 될 수 있다.[10] p_Object는 멤버로서 자신의 객체식별자를 하위 클래스에서 자신의 객체식별자가 필요한 경우 이를 이용할 수 있고[11] 객체를 위치를 찾기위한 인덱스의 키로 사용된다.

Ref와 RefAny 클래스

Ref와 RefAny는 객체를 가리키기위한 지속성 포인터이다. Ref는 참조되는 객체의 타입을 인자로 가지는 템플릿 클래스로서 C++의 포인터와 같이 사용된다. RefAny는 타입정보를 유지하지 않고 모든 객체들이 가질 수 있는 공통적인 기능만을 인터페이스로 가진다. Electro-21에서는 ODMG의 RefAny 인터페이스외에 read, write 의 두가지 인터페이스를 추가해서 객체의 일부분을 읽거나 쓸 수 있도록 하였으며 이는 주로 객체질의어 처리기(OQL processor)가 사용한다.

Blob 클래스

Blob 클래스는 비정형 거대 객체를 위한 다루기 위한 클래스로서 이미지, 음향, 텍스트등의 멀티미디어 시스템에서 주로 사용되는 데이터의 저장에 유리하다. 객체관리기의 측면에서 Blob의 특징은 일반 객체와는 달리 객체캐쉬에는 헤더만이 적재되며 데이터 영역은 응용프로그램의 프리스토어(free store)에 직접 복사되는 것이다.

Blob 클래스의 멤버는 **size** 가 전부이며 Blob및 Blob으로 부터 상속받은 클래스의 객체는 그 멤버들의 값이 지속적으로 보존되지 않는 점에 주의해야 한다. 즉 Blob 타입의 객체가 디스크에 저장된 후

[10] 이는 사용자가 객체 생성시에 해당 데이터베이스나 클러스터될 객체를 명시함으로서 이루어 진다.
[11] Blob의 경우에는 사용할 수 없다.

다시 읽혀질 때는 그 멤버의 값이 보존되지 않는다. Blob은 read와 write의 두 멤버함수를 인터페이스로 가진다. read는 블럽데이터를 사용자 메모리 영역으로 복사하며 write는 주어지 크기의 데이터를 사용자 영역으로 부터 시스템으로 복사한다. write시 기존의 데이터는 사라지게 된다. 이들은 모두 데이터 전체를 일시에 메모리로 적재하거나 디스크에 저장하는데 부분적으로도 읽기와 쓰기가 가능한 인터페이스도 제공할 예정이다.

3 결론

그동안 객체지향 데이터베이스의 성장의 저해 요소이던 통일된 데이터 모델의 부재는 ODMG 데이터 모델의 출현에 힘입어 사라지게 될 것이다. Electro-21의 ODBMS는 ODMG의 C++ 바인딩에 충실하도록 구현되었으며 Ref, RefAny 등과 Database, Transaction, Collection 등과 같은 시스템 클래스들을 통해 응용프로그램에 대한 전처리(preprocessing) 과정을 거치지 않고 사용할 수 있도록 했다.

객체관리기는 객체캐쉬, 캐쉬뱅크 등의 객체지향 개념을 이용해서 설계되고 C++을 사용해서 구현되었기 때문에 확장성과 재사용성이 뛰어나며 LRU를 적용한 객체캐쉬 알고리즘을 사용하여 캐쉬의 효과를 높일 수 있었다. 현재 클라이언트-서버형으로의 확장과 swizzling과 같은 성능향상을 위한 연구가 진행되고 있다.

감사의 글

본 시스템을 개발하는데 많은 도움을 주신 현대전자의 ODBMS 팀과 동양SHL의 서버팀, KAIST 데이터베이스 시스템 연구원들께 진심으로 감사드립니다.

References

[BDG93] A. Biliris, S. Dar, and N. H. Gehani. "Making C++ Objects Persistent: The Hidden Pointers". *Software—Practice & Experience*, 1993.

[Cat91] R. G. G. Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Addison-Wesley Publishing Company, Inc., 1991.

[Cat94] R.G.G. Cattel, editor. *The Object Database Standard: ODMG-93(release 1.1)*. Morgan Kaufman Publishers, Inc., 1994.

[KC86] Setrag Khoshafian and George P. Copeland. "Object Identity". In *ACM OOPSLA Conference Proceedings*, pages 406–416, September 1986.

[Kim90] Won Kim. *Introduction to Object-Oriented Databases*. The MIT Press, 1990.

[KS91] Henry F. Korth and Abraham Silberschatz. *Database System Concepts*. McGraw-Hill, Inc, second edition, 1991.

[Mos92] J. Eliot B. Moss. "Working with Persistent Objects: To Swizzle or Not to Swizzle". *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[Pop91] Steven S. Popovich. An architectural framework for object management systems. Technical report, September 1991.