BASES DE DATOS ORIENTADAS A OBJETOS

IES SAN VICENTE
2º, ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS

José Piqueres Torres

INDICE

INTRODUCCIÓN

Los modelos de bases de datos tradicionales presentan deficiencias en cuento a aplicaciones más complejas o sofisticadas. Además son difíciles de utilizar cuando las aplicaciones que acceden a ellas están escritas en un lenguaje de programación orientado a objetos.

La orientación a objetos ofrece flexibilidad, no está limitada por los tipos de datos y los lenguajes de consulta de los sistemas de bases de datos tradicionales. La característica clave es la potencia que proporcionan al diseñador al permitirle especificar tanto la estructura de objetos complejos, como las operaciones que se pueden aplicar sobre dichos objetos.

Las BDOO se han diseñado para que se puedan integrar directamente con aplicaciones desarrolladas con lenguajes orientados a objetos. También están diseñadas para simplificar la POO. Almacenan los objetos en la BD con las mismas estructuras y relaciones que los lenguajes de POO.

Una SGBDOO es una SGBD que almacena objetos incorporando así todas las ventajas de la OO. Pueden tratar directamente con objetos, no teniendo que hacer la traducción a tablas o registros. Sus objetos se conservan, pueden ser gestionados aunque su tamaño sea grande, pueden ser compartidos entre múltiples usuarios y mantienen su integridad como sus relaciones.

ODMG (Object Database Mangement Group) es el grupo de fabricantes de SGBDOO que propuso el estándar ODM-93 en 1993; en 1997 evolucionó a ODMG-2.0 y en enero de 2000 se publicó la última versión ODMG 3.0. El uso del estándar proporciona portabilidad (que se pueda ejecutar sobre sistemas distintos), interoperabilidad (que la aplicación pueda acceder a varios sistemas diferentes) y además permite que los usuarios puedan comparar entre distintos sistemas comerciales.

VENTAJAS E INCONVENIENTES

Las ventajas de un SGBDOO son:
- Mayor capacidad de modelado:

Un objeto permite encapsular tanto un estado como un comportamiento.

Un objeto puede almacenar todas las relaciones que tenga con otros objetos.

Los objetos pueden agruparse para formar objetos complejos (herencia).
- Ampliabilidad:

Se pueden construir nuevos tipos de datos a partir de los ya existentes

Agrupar propiedades comunes de diversas clases e incluirlas en una superclase, lo que reduce la redundancia.

Reusabilidad de clases, lo que repercute en una mayor facilidad de mantenimiento y un menor tiempo de desarrollo.
- Lenguaje de consulta más expresivo.

El acceso navegacional desde un objeto al siguiente es la forma más común de acceso a datos en un SGBDOO. Mientras que SQL utiliza el acceso asociativo.

El acceso navegacional es más adecuado para gestionar operaciones como los despieces, consultas recursivas, etc.
- Adecuación a las aplicaciones avanzadas de base de datos.

Hay muchas áreas en las que los SGBD tradicionales no han tenido excesivo éxito como el CAD, CASE, OIS, sistemas multimedia, etc. en los que las capacidades de modelado de los SGBDOO han hecho que esos sistemas sí resulten efectivos para este tipo de aplicaciones.
- Mayores prestaciones.

Los SGBDOO proporcionan mejoras significativas de rendimiento con respecto a los SGBD relacionales.

Los inconvenientes de un SGBDOO son:
- Carencia de un modelo de datos universal.

No hay ningún modelo de datos que esté universalmente aceptado para los SGBDOO y la mayoría de los modelos carecen una base teórica.
- Carencia de experiencia.

Todavía no se dispone del nivel de experiencia del que se dispone para los sistemas tradicionales.
- Carencia de estándares.

Existe una carencia de estándares general para los SGBDOO.
- Competencia. Con respecto a los SGBDR y los SGBDOR.

Estos productos tienen una experiencia de uso considerable. SQL es un estándar aprobado y ODBC es un estándar de facto. Además, el modelo relacional tiene una sólida base teórica y los productos relacionales disponen de muchas herramientas de soporte que sirven tanto para desarrolladores como para usuarios finales.
- La optimización de consultas compromete la encapsulación.

La optimización de consultas requiere una compresión de la implementación de los objetos, para poder acceder a la base de datos de manera eficiente. Sin embargo, esto compromete el concepto de encapsulación.

- El modelo de objetos aún no tiene una teoría matemática coherente que le sirva de base.

MANIFIESTOS ACERCA DE LAS SGBDOO

- Manifiesto de los Sistemas de Bases de Datos al Objeto puras, ATKINSON, 1989:

Enfoque purista que sostiene que los SGBO deben soportar una modelo de objetos puros y no basarse en extensiones semánticas de modelos clásicos como el relacional.

- Manifiesto de los SBD de Tercera Generación, STONEBRAKER, 1990:

SGBD Relacionales extendidos que sean capaces de soportar los conceptos de orientación al objeto. Postura que propugnan los principales vendedores de productos relacionales.

- Tercer manifiesto, DARWEN y DATE 1995:

Reinterpretan el modelo relacional bajo la visión orientada al objeto.

EL MODELO ESTÁNDAR DE ODMG

Modelo de objetos:

Permite que los diseños y las implementaciones sean portables entre los sistemas que lo soportan. Primitivas de modelado:

- Componentes básicos son objetos y literales. Un objeto es una instancia autocontenida de una entidad de interés del mundo real. Tienen identificador único. Literal es un valor específico. No tiene identificadores. Pude ser una estructura o un conjunto de valores relacionados.
- Se categorizan en tipos. Cada tipo tiene un dominio específico compartido por todos los objetos y literales de ese tipo. Los tipos también pueden tener comportamientos, que también comparten todos los objetos del mismo.
- Lo que un objeto sabe hacer son sus operaciones. Puede requerir datos de entrada y devolver algún valor de un tipo conocido.
- Las propiedades son sus atributos y las relaciones. El estado viene dado por los valores actuales de sus propiedades.
- Una base de datos es un conjunto de objetos almacenados que pueden ser accedidos por múltiples usuarios y aplicaciones.
- La definición de una base de datos está contenida en un esquema que se ha creado mediante el lenguaje de definición de objetos ODL.

Objetos:

Los tipos de tipo colección (para las clases contenedor) son:

Set<tipo>: Grupo desordenado de objetos del mismo tipo. No se permiten duplicados.

Bag<tipo>: Grupo desordenado de objetos del mismo tipo. Se permiten duplicados.

List<tipo>: Grupo ordenado de objetos del mismo tipo. Se permite duplicados.

Array<tipo>: Grupo ordenado de objetos del mismo tipo que se puede acceder por su posición. El tamaño es dinámico.

Dictionary<clave,valor>: índice. Formado por las clavez ordenadas, emparejada con un solo valor.

De tipo estructurado:
Date: fecha del calendario
Time: hora.
Timestamp: hora de una fecha.
Interval: período de tiempo.

Literales:

No tienen identificadores y no pueden aparecer solos como objetos, están embebidos en objetos y no pueden referenciarse de modo individual. Los atómicos son:

Boolean: verdadero o falso.
Short: entero con signo de 8 o 16 bits.
Long: entero con signo de 32 o 64 bits.
Unsigned short: entero sin signo de 8 o 16 bits.
Unsigned long: entero sin signo de 32 o 64 bits.
Float: valor real en coma flotante de simple precisión.
Double: valor real en coma flotante de doble precisión
Octet: almacén de 8 bits.
Char: carácter ASCII o UNICODE.
String: cade de caracteres.
Enum: tipo enumerado donde los valores se especifican explícitamente cuando se declara el tipo.

Literales estructurales contienen un número fijo de elementos heterogéneos. Cada elemento es un par <nombre,valor> donde valor puede ser cualquier tipo literal. Tipos estructurados son: date, time, timestamp, interval y struct.

Los de colección son: set<tipo>, bag<tipo>, listo<tipo>, array<tipo> y dictionaru<clave,valor>.

Tipos

Una interface es una especificación del comportamiento abstracto de un tipo de objeto y contiene las asignaturas de las operaciones. No se pueden crear objetos a partir de ella (equivalente a una clase abstracta).

Un clase es una especificación del comportamiento abstracto y del estado abstracto de un tipo de objeto. A partir de ellas se pueden crear instancias de objetos individuales (equivalente a una clase concreta).

Soporta la herencia simple y la herencia múltiple mediante las interfaces. Se suelen utilizar para especificar operaciones abstractas que pueden ser heradas por las clases o por otras interfaces: Herencia de comportamiento. Requiere que el supertipo sea una interface y el subtipo sea una clase o un interface.

Se puede hacer referencia a los subtipo como su supertipo. Los subtipo se pueden especializar como sea necesario añadiéndoles comportamientos. Los subtipos de un subtipo especializado heredan también los comportamientos añadidos.

La relación extiende (extends) para iindicar la herencia de estado y de comportamiento. La clase puede extender a otra clase. Las clases de más abajo en la jeraquía heredan todo lo que sus supertipos heredan de las clases que tienen por encima.

Extension (extent) tiene un nombre e incluye todas las instancias de objetos persistentes creadas a partir de dicho tipo.

Puede tener una o más claves (key), es un identificador único. Con una solo propieda es una clave simple y con varias propiedades es una clave compuesta.

La representación es una estructura de datos dependiente de un lenguaje de programación que contiene las propiedades del tipo.

Los detalles de las operaciones de un tipo se especifican mediante un conjunto de métodos. Un tipo puede incluir métodos que nunca se ven desde fuera del tipo. Los métodos se escribirán en el mismo lenguaje de programación utilizado.

Propiedades

Atributos: Define del tipo de un objeto. No tiene identificador, toma como valor un literal o el identificador de un objeto.

Relaciones: Se definen entre tipos. Sólo soporta relaciones binarias (1:1, 1:n, n:m). Define caminos transversales en la interface de cada dirección. En el lado del muchos los objetos pueden estar desordenados (set o bag) u ordenados (list).

Transacciones

Son unidades lógicas de trabajo que llevan a la base de datos de un estado consistente a otro estado consistente. Todos los accesos, creación, modificación y borrado de objetos persistentes se deben realizar dentro de una transacción.

ODL:

ODL es un lenguaje de especificación para definir tipos de objetos para sistemas complejos compatibles con ODMG. Es el equivalente de DDL (Data Definition Languaje o lenguaje de definición de datos) de los DBMS tradicionales. Define los atributos y las relaciones entre tipos y especifica la signatura de las operaciones. La sintaxis de ODL extiende el lenguaje de definición de interfaces (IDL) de la arquitectura CORBA (Common Object Request Broker Architecture).

Las declaraciones de atributos son sintácticamente idénticas las declaraciones de miebros de C++.

Ejemplo:

```
class Persona
    (extent personas key dni)
    {
    /* Definición de atributos */
            attribute struct Nom_Persona {string nombre pila, string apellido1,
    string apellido2} nombre;
            attribute string dni;
            attribute date fecha nacim;
            attribute enum GenerofF,Mg sexo;
            attribute struct Direccion {string calle, string cp, string ciudad}
            direccion;
    /* Definición de operaciones */
```

```
        float edad();
    }

    class Profesor extends Persona
        (extent profesores)
        {
        /* Definición de atributos */
                attribute string categoria;
                attribute float salario;
                attribute string despacho;
                attributo string telefono;
        /* Definición de relaciones */
                relationship Departamento trabaja en
                        inverse Departamento::tiene profesores;
                relationship Set<EstudianteGrad> tutoriza
                        inverse EstudianteGrad::tutor;
                relationship Set<EstudianteGrad> en comite
                        inverse EstudianteGrad::comite;
        /* Definición de operaciones */
                void aumentar salario(in float aumento);
                void promocionar(in string nueva categoria);
    }

    class Estudiante extends Persona
        (extent estudiantes)
        {
        /* Definición de atributos */
                attribute string titulacion;
        /* Definición de relaciones */
                relationship set<Calificacion> ediciones cursadas
                        inverse Calificacion::estudiante;
                relationship set<EdicionActual> matriculado
                        inverse EdicionActual::estudiantes matriculados;
        /* Definición de operaciones */
                float nota media();
                void matricularse(in short num edic) raises(edicion no valida,
                edicion llena);
                void calificar(in short num edic; in float nota)
                raises(edicion no valida, nota no valida);
    };

    class Calificacion
        (extent calificaciones)
        {
        /* Definición de atributos */
                attribute float nota;
        /* Definición de relaciones */
                relationship Edicion edicion inverse Edicion::estudiantes;
                relationship Estudiante estudiante
                        inverse Estudiante::ediciones cursadas;
```

```
};

class EstudianteGrad extends Estudiante
    (extent estudiantes graduados)
    {
    /* Definición de atributos */
            attribute set<Titulo> titulos;
    /* Definición de relaciones */
            relationship Profesor tutor inverse Profesor::tutoriza;
            relationship set<Profesor> comite inverse Profesor::en comite;
    /* Definición de operaciones */
            void asignar tutor(in string apellido1; in string apellido2)
            raises(profesor no valido);
            void asignar miembro comite(in string apellido1; in string
    apellido2)
            raises(profesor no valido);
};

class Titulo
    {
    /* Definición de atributos */
            attribute string escuela;
            attribute string titulo;
            attribute string a~no;
};

class Departamento
    (extent departamentos key nombre)
    {
    /* Definición de atributos */
            attribute string nombre;
            attribute string telefono;
            attribute string despacho;
            attribute string escuela;
            attribute Profesor director;
    /* Definición de relaciones */
            relationship set<Profesor> tiene profesores
                    inverse Profesor::trabaja en;
            relationship set<Curso> oferta
                    inverse Curso::ofertado por;
};

class Curso
    (extent cursos key num curso)
    /* Definición de atributos */
        attribute string nombre;
        attribute string num curso;
        attribute string descripcion;
    /* Definición de relaciones */
        relationship set<Edicion> tiene ediciones
```

```
            inverse Edicion::de curso;
        relationship Departamento ofertado por inverse Departamento::oferta;
};

class Edicion
    (extent ediciones)
    {
    /* Definición de atributos */
        attribute short num edic
        attribute string a~no;
        attribute enum SemestrefPrimero,Segundog semestre;
    /* Definición de relaciones */
        relationship set<Calificacion> estudiantes
            inverse Calificacion::edicion;
        relationship Curso de curso inverse Curso::tiene ediciones;
};

class EdicionActual extends Edicion
    (extent ediciones actuales)
    {
    /* Definición de relaciones */
    relationship set<Estudiante> estudiantes matriculados
            inverse Estudiante::matriculado;
    /* Definición de operaciones */
            void matricular estudiante(in string dni)
        raises(estudiante no valido,edicion llena);
};
```

OQL:

Permite realizar consultas de modo eficiente sobre bases de datos orientadas a objetos. Basado en SQL-92.

La sintaxis básica de OQL es una structura SELECT… FROM… WHERE… como en SQL:

```
SELECT d.nombre
FROM d in departamentos
WHERE d.escuela=”Ingeniería”;
```

En las consultas se necesita un punto de entrada que es la extensión de una clase. Además es necesario utilizar una variable iteradora que vaya tomando valores en los objetos de la colección. Estas se puede especificar de estas formas:

```
D in departamentos
Departamentos d
Departamentos as d
```

Una consulta no debe seguir la estructura SELECT obligatoriamente, el nombre de cualquier objto persistente es una consulta de por sí:

```
Departamentos;
```

Si se da nombre a un objeto concreto. Por ejemplo: departamentoinf; al departamento de informatica, es devuelve una referencia a ese objeto individual. Entonces cuando se establece un punto de entrada se pueden utilizar expresiones de caminos para especificar un camino a atributos y objetos relacionados. Empieza con un nombre de objeto persistente o una variable iterador, seguida de ninguno o varios nombres de ralciones o de atributos conectados mediante un punto:

Departamentoinf.director;
departamentoinf.director.categoria:
departamentoinf.tiene_profesores;

Una consulta OQL puede devolver un resultado con una estructura compleja especificada en la misma consulta utilizando struct.

Ejemplo que muestra los nombres y apellidos de los estudiantes y lo títulos que tiene cada uno:

```
Select struct(nombre:struct(ape1: e.nombre.apellido1,
                            ape2 e.nombre.apellido2,
                            nom: e.nombre.nombre_pila),
              titulos:(Select struct(tit: t.titulo,
                                     año: t.año,
                                     esc: t.escuela)
                       From t in e.titulos)
 From e in departamentoinf.director.tutoriza;
```

OQL es ortogonal respecto a la especificación de expresiones de caminos: atributos, relaciones y operaciones pueden ser utilizados en estas expresiones, siempre que el sistema de tipos de OQL no se vea comprometido

OQL tiene además otras características:

- Especificación de vistas dando nombres a consultas.
- Obtención como resultado de un solo elemento.
- Uso de operadores de colecciones: funciones agregados y cuantificadores.
- Uso de group by.

ORACLE

Tipo de objetos y referencias

Para crear tipo de objetos utilizamos CREATE TYPE. Ejemplo:

```
CREATE TYPE persona AS OBJECT(
        Nombre VARCHAR2(30),
        Telefono VARCHAR2(20)
);
```

Un tabla relacional tiene una columna cuyo tipo es un objeto: objetos coulmna. Ejemplo:

```
CREATE TABLE contactos(
        Contacto persona,
        Fecha DATE
);
```

Se puede incluir una clausula default:

```
CREATE TABLE departamento(
        Num_dept VARCHAR2(5) PRIMARY KEY,
        Nombre_dept VARCHAR2(20),
        Director persona DEFAULT persona(1,'Pepe Pérez',NULL),
        Empleados gente DEFAULT gente(
                                persona(2,'Ana López','C/ del pez, 5')
                                persona(3,'Eva García',NULL) )
)
NESTED TABLE empleados STORE AS empleados_tab
```

Las columnas que son tablas anidadas y los atributos que son tablas de objetos requieren una table a parte donde almacenar las filas de dichas tablas. Se especifica con NESTED TABLE… STORE AS…

Definir una clave primaria sobre una tabla de objetos:

```
CREATE TYPE persona AS OBJECT(
        Id NUMBER,
        Nombre  VARCHAR2(30),
        Direccion VARCHAR2(30),
        Oficina ubicación
);
CREATE TABLE empleados OF persona(
        Id PRIMARY KEY
);
```

También se pueden definir disparadores:

```
CREATE TRIGGER disparador
        AFTER UPDATE OF despacho ON empleados
        FOR EACH ROW
        WHEN new.despacho.ciudad='Castellon'
        BEGIN
                IF (:new.despacho.num_edificio=600) THEN
                        INSERT INTO traslado (id, despacho_antiguo,
                despacho_nuevo)
                                VALUES (:old.id, :old.despacho,
                        :new.despacho);
                END IF;
        END;
```

Métodos

Son funciones o procedimeinto que se pueden declarar en la definición de un tipo de objeto para implementar el comportamiento que se desea para dicho tipo de objeto. El método miembro se utiliza para ganar acceso a los datos de una instancia de un objeto. El parámetro SELF denota a la instancia del objeto sobre la que se está invocando el método:

```
CREATE TYPE BODY racional AS
        MEMBER PROCEDURE normaliza IS
                g INTEGER;
        BEGIN
                g := gcd(SELF.num, SELF.den);
                g := gcd(num, den); -- equivale a la linea anterior
                num :=num/g;
                den :=den/g;
        END normaliza;
        ….
END;
```

Colecciones

Dos tipos: Un varray es una collección ordenada de elementos. Se debe especificar el número máximo (se puede cambiar). Una tabla anidada puede tener cualquier número de elementos. No se mantiene el orden de estos. Y los elementos se almacenan en una tabla a parte en la que hay una columna llamada NESTED_TABLE_ID:

```
CREATE TYPE precios AS VARRAY(10) OF NUMBRE (12,2);
CREATE TYPE lineaped_tabla AS TABLE OF lineaped;
```

Se pueden consultar con resultados anidados y sin anidar:

```
SELECT e.nombre, e.proyectos
FROM empledos e
```

```
NOMBRE      PROYECTOS
----------  --------------------
'Pedro'     tab_proyecto(67,82)
'Juan'      tab_proyecto(22,67,97)
```

```
SELECT e.nombre, p.*
FROM empleados e, TABLE(e.proyectos) p;
```

```
NOMBRE      PROYECTOS
----------  --------------------
'Pedro'     67
'Pedro'     82
'Juan'      22
'Juan'      67
'Juan'      97
```

Herencia de tipos

Cuando se crea un subtipo a partir de un tipo, el subtipo hereda todos los atributos y los métodos del tipo padre. Cualquier cambio en los atributos o métodos del tipo padre se reflejan automáticamente en el subtipo. Para crear un subtipo se utiliza la cláusula UNDER:

```
CREATE TYPE estudiante UNDER persona(…,
        Titulacion VARCHAR(30),
        Fecha-ingreso DATE
) NOT FINAL;
```

Funciones y predicados útiles con objetos

VALUE: devuelve instancias de objetos correspondientes a las filas de la tabla.

```
SELECT VALUE(p)
FROM personas_tab p
WHERE p.direccion LIKE 'C/Mayor%';
```

REF: toma como parámetro un alias de tabla y devuelve una referencia a una instancia de un objeto de dicha tabla.

IS OF comprobar el nivel de especialización

```
SELECT VALUE(p)
FROM personas_tab p
WHERE VALUE(p) IS OF (estudiante);
```

TREAT: trata a una instancia de un supertipo como una instancia de uno de sus subtipos.

```
SELECT TREAT(VALUE(p) AS estudiante)
FROM personas_tab p
WHERE VALUE(p) IS OF (ONLY estudiante);
```

BIBLIOGRAFÍA

www.odbms.org

http://es.wikipedia.org/wiki/Base_de_datos

http://es.wikipedia.org/wiki/Base_de_datos_orientada_a_objetos

http://www.itescam.edu.mx/principal/sylabus/fpdb/recursos/r23897.PDF

http://tejo.usal.es/~fgarcia/docencia/poo/02-03/trabajos/S1T3.pdf

http://www3.uji.es/~mmarques/e16/teoria/cap2.pdf

http://www.cs.cinvestav.mx/BDChapa/Beto/Blanco.htm

http://tejo.usal.es/~fgarcia/docencia/poo/03-04/Trabajos/SGBDOO.pdf

http://basesdatos.uc3m.es/fileadmin/Docencia/BDA-II/teoria/BDOO.pdf

www.bdoo.wordpress.com