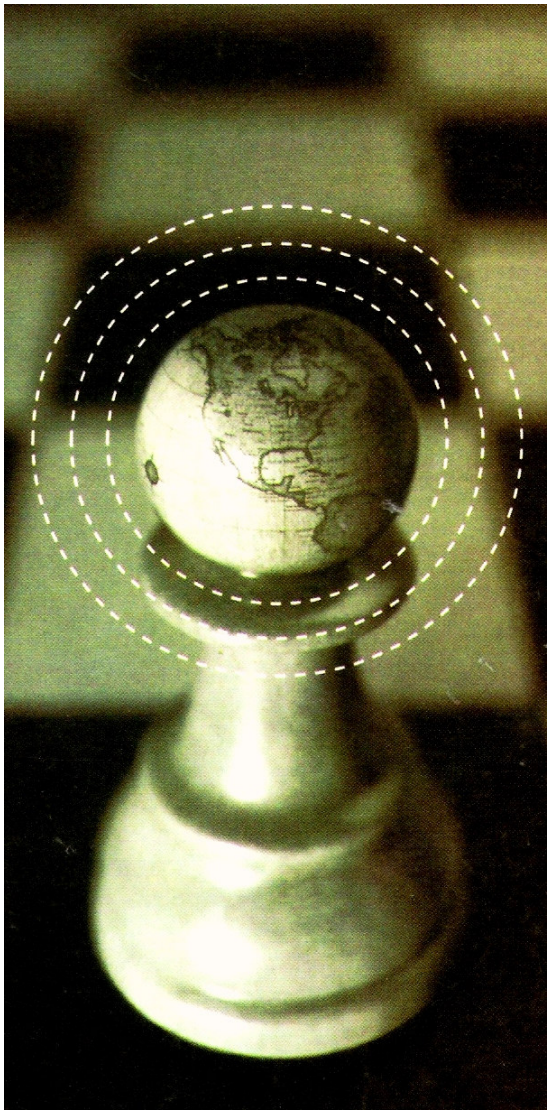


Universidad Católica
"Nuestra Señora de la Asunción"

Teoría y Aplicación de la Informática II

Bases de Datos Orientadas a Objetos

Daniel F. Cáceres
dcaceres@knowhow.com.py

Asunción, 21 de setiembre de 2006

Objetos, Objetos, Objetos por Todos Lados

Bases de Datos Orientadas a Objetos

¿Qué es OO?

En esos mundos OO, el conocimiento se descentraliza en todos los objetos que lo componen, cada objeto sabe hacer lo suyo y no le interesa saber cómo el vecino hace su trabajo, pero sabe que lo hace y qué es lo que puede hacer. Como bien lo definió Dan Ingalls de Smalltalk con las siguientes palabras: [3]

"La orientación a objetos proporciona una solución que conduce a un Universo de Objetos "bien educados" que se piden de manera cortés, concederse mutuamente sus deseos". [3]

¿Por qué OO?

La meta es dejar la etapa en la que la construcción del software es una labor de artesanos, y pasar a la etapa en la que se pueda tener fábricas de software, con gran capacidad de reutilización de código y con metodologías eficientes y efectivas que se apliquen al proceso de producción. [3]

¿Qué es una BDOO?

Es una base de datos que soporta el paradigma orientado a objetos, almacenando datos y métodos, y no sólo datos. Está diseñada para almacenar objetos complejos que anteriormente existían solo en tiempo de ejecución de los programas. Es más segura ya que no permite tener acceso a los datos (objetos) directamente; el acceso se debe realizar a través de los métodos que el programador creó para tal efecto. [3]

Un Modelo Conceptual Unificado

Las técnicas OO utilizan los mismos modelos conceptuales para el análisis, diseño y construcción. La tecnología de las BDOO da un paso más hacia la unificación. El modelo conceptual de las bases de datos OO es igual al del resto del mundo OO, en lugar de utilizar tablas independientes por relación. [3]

El uso del mismo modelo conceptual para todos los aspectos del desarrollo simplifica dicho trabajo, particularmente con las herramientas CASE OO, mejora la comunicación entre usuarios, analistas y programadores, además de que reduce las posibilidades de error. [3]

Necesidad de Evolución

Los modelos de bases de datos tradicionales (relacional, red y jerárquico) han sido capaces de satisfacer con éxito las necesidades, en cuanto a bases de datos, de las aplicaciones de gestión tradicionales. Sin embargo, presentan algunas deficiencias cuando se trata de aplicaciones más complejas o sofisticadas como, por ejemplo, el diseño y fabricación en ingeniería (CAD/CAM, CIM -se explican más adelante en este texto-), los experimentos científicos, los sistemas de información geográfica o los sistemas multimedia. Los requerimientos y las características de estas nuevas aplicaciones difieren en gran medida de las típicas aplicaciones de gestión: la estructura de los objetos es más compleja, las transacciones son de larga duración, se necesitan

nuevos tipos de datos para almacenar imágenes y textos, y hace falta definir operaciones no estándar, específicas para cada aplicación. [4]

Las bases de datos orientadas a objetos se crearon para tratar de satisfacer las necesidades de estas nuevas aplicaciones. La orientación a objetos ofrece flexibilidad para manejar algunos de estos requisitos y no está limitada por los tipos de datos y los lenguajes de consulta de los sistemas de bases de datos tradicionales. Una característica clave de las bases de datos orientadas a objetos 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. [4]

Otro motivo para la creación de las bases de datos orientadas a objetos es el creciente uso de los lenguajes orientados a objetos para desarrollar aplicaciones. Las bases de datos se han convertido en piezas fundamentales de muchos sistemas de información y las bases de datos tradicionales son difíciles de utilizar cuando las aplicaciones que acceden a ellas están escritas en un lenguaje de programación orientado a objetos como C++, Smalltalk o Java. Las bases de datos orientadas a objetos se han diseñado para que se puedan integrar directamente con aplicaciones desarrolladas con lenguajes orientados a objetos, habiendo adoptado muchos de los conceptos de estos lenguajes. [4]

El *modelo orientado a objetos* se basa en el paradigma de la programación orientada a objetos. [1] El desarrollo del paradigma orientado a objetos aporta un gran cambio en el modo en que vemos los datos y los procedimientos que actúan sobre ellos. Tradicionalmente, los datos y los procedimientos se han almacenado separadamente: los datos y sus relaciones en la base de datos y los procedimientos en los programas de aplicación. La orientación a objetos, sin embargo, combina los procedimientos de una entidad con sus datos. [4]

Esta combinación se considera como un paso adelante en la gestión de datos. Las entidades son unidades auto-contenidas que se pueden reutilizar con relativa facilidad. En lugar de ligar el comportamiento de una entidad a un programa de aplicación, el comportamiento es parte de la entidad en sí, por lo en cualquier lugar en el que se utilice la entidad, se comporta de un modo predecible y conocido. [4]

El modelo orientado a objetos también soporta relaciones de muchos a muchos, siendo el primer modelo que lo permite. [4]

A modo de obtener un marco general de referencia es oportuno resaltar que las primeras bases de datos se desarrollaron a partir de sistemas de gestión de archivos. Algunas de las características comunes de estas aplicaciones "antiguas" son las siguientes: [1]

- **Orientación a registros:** los elementos de datos fundamentales consisten en registros de longitud fija. [1]

- **Elementos de datos de pequeño tamaño:** Cada registro es pequeño (los registros rara vez superan los centenares de bytes de longitud). [1]

- **Campos atómicos:** Los campos de cada registro son cortos y de longitud fija. No hay estructuras en el interior de los campos. [1]

En los últimos años, sin embargo, la tecnología de las bases de datos se ha aplicado a nuevos territorios. Estas nuevas aplicaciones incluyen las siguientes: [1]

- **Diseño asistido por computadora** (CAD, *Computer-Aided Design).* Las bases de datos CAD guardan los datos correspondientes a los diseños de ingeniería, incluyendo los componentes del elemento que se diseña, la interrelación de los componentes y las versiones antiguas de los diseños. [1]

- **Ingeniería de software asistida por computadora** (CASE, *Computer-Aided Software Engineering).* Las bases de datos CASE guardan los datos necesarios para ayudar a los desarrolladores de software. Estos datos incluyen el código fuente, las dependencias entre los módulos de software, las definiciones y usos de las variables, y el historial de programación del sistema de software. [1]

- **Bases de datos multimedia.** Las bases de datos multimedia contienen imágenes, datos de sonido, de vídeo y similares. Las bases de datos de este tipo surgen de la necesidad de guardar fotografías, datos geográficos, datos de los sistemas de correo de voz, de las aplicaciones gráficas y de los sistemas de vídeo bajo demanda. [1]

- **Sistemas de información para oficinas** (SIO). La automatización de las oficinas incluye herramientas para la creación y recuperación de documentos, herramientas para la conservación y organización de los calendarios de citas, etc. Las bases de datos SIO permiten consultas referentes a las agendas, a los documentos y al contenido de los documentos. [1]

Los objetos han entrado en el mundo de las bases de datos de formas: [4]

- SGBD orientados a objetos puros: son SGBD basados completamente en el modelo orientado a objetos. [4]

- SGBD híbridos u objeto-relacionales: son SGBD relacionales que permiten almacenar objetos en sus relaciones (tablas). [4]

El modelo orientado a objetos

A continuación se definen los conceptos del paradigma orientado a objetos en programación, ya que el modelo de datos orientado a objetos es una extensión del mismo. [4]

- **Objeto:** es un elemento auto-contenido utilizado por el programa. Los valores que almacena un objeto se denominan atributos, variables o propiedades. Los objetos pueden realizar acciones, que se denominan métodos, servicios, funciones, procedimientos u operaciones. Los objetos tienen un gran sentido de la privacidad, por lo que sólo dan información sobre sí mismos a través de los métodos que poseen para compartir su información. También ocultan la implementación de sus procedimientos, aunque es muy sencillo pedirles que los ejecuten. Los usuarios y los programas de aplicación no pueden ver qué hay dentro de los métodos, sólo pueden ver los resultados de ejecutarlos. A esto es a lo que se denomina ocultación de información o encapsulamiento de datos. Cada objeto presenta una interface pública al resto de objetos que pueden utilizarlo. Una de las mayores ventajas del encapsulamiento es que, mientras que la interface pública sea la misma, se puede cambiar la implementación de los métodos sin que sea necesario informar al resto de los objetos que los utilizan. Para pedir datos a un objeto o que éste realice una acción se le debe enviar un **mensaje**. Un programa orientado a objetos es un conjunto de objetos que tienen atributos y métodos. **Los objetos interactúan enviándose mensajes.** La clave, por supuesto, es averiguar qué objetos necesita el programa y cuáles deben ser sus atributos y sus métodos. [4]

- **Clase:** Es un patrón o plantilla en la que se basan objetos que son similares. Cuando un programa crea un objeto de una clase, proporciona datos para sus variables, y el objeto puede entonces utilizar los métodos que se han escrito para la clase. Todos los objetos creados a partir de la misma clase comparten los mismos procedimientos para sus métodos, también tienen los mismos tipos para sus datos, pero los valores pueden diferir. Una clase también es un tipo de datos. De hecho una clase es una implementación de lo que se conoce como un tipo abstracto de datos. El que una clase sea también un tipo de datos significa que una clase se puede utilizar como tipo de datos de un atributo. [4]

- **Tipos de clases.** En los programas orientados a objetos hay tres tipos de clases: **clases de control, clases entidad y clases interface.** Las **clases de control** gestionan el flujo de operación de un programa (por ejemplo, el programa que se ejecuta es un objeto de esta clase). Las **clases entidad** son las que se utilizan para crear objetos que manejan datos (por ejemplo, clases para personas, objetos tangibles o eventos). Las **clases interface** son las que manejan la entrada y la salida de información (por ejemplo, las ventanas gráficas y los menús utilizados por un programa). En los programas orientados a objetos, las clases entidad no hacen su propia entrada/salida. El teclado es manejado por objetos interface que recogen los datos y los

envían a los objetos entidad para que los almacenen y los procesen. La salida impresa y por pantalla la formatea un objeto interface para obtener los datos a visualizar de los objetos entidad. Cuando los objetos entidad forman parte de la base de datos, es el SGBD el que se encarga de la entrada/salida a ficheros. El resto de la entrada/salida la manejan los programas de aplicación o las utilidades del SGBD. **Muchos programas orientados a objetos tienen un cuarto tipo de clase: la clase contenedor**. Estas clases contienen, o manejan, múltiples objetos creados a partir del mismo tipo de clase. **También se conocen como agregaciones.** Las clases contenedor mantienen los objetos en algún orden, los listan e incluso pueden permitir búsquedas en ellos. **Muchos SGBD orientados a objetos llaman a sus clases contenedor extents (extensiones)** y su objetivo es permitir el acceso a todos los objetos creados a partir de la misma clase. [4]

- **Tipos de métodos.** Hay varios tipos de métodos que son comunes a la mayoría de las clases: **los constructores** son métodos que tienen el mismo nombre que la Clase y se ejecutan cuando se crea un objeto de una clase, por lo tanto, un constructor contiene instrucciones para inicializar las variables de un objeto. **Los destructores** son métodos que se utilizan para destruir un objeto. No todos los lenguajes orientados a objetos poseen destructores. **Los accesores** son métodos que devuelven el valor de un atributo privado de otro objeto. Así es como los objetos externos pueden acceder a los datos encapsulados. **Los mutadores** son métodos que almacenan un nuevo valor en un atributo. De este modo es cómo objetos externos pueden modificar los datos encapsulados. Además, cada clase tendrá otros métodos dependiendo del comportamiento específico que deba poseer. [4]

- **Sobrecarga de métodos.** Una de las características de las clases es que pueden tener métodos sobrecargados, que son métodos que tienen el mismo nombre pero que necesitan distintos datos para operar. Ya que los datos son distintos, las interfaces públicas de los métodos serán diferentes. Por ejemplo, consideremos una clase contenedor, *TodosLosEmpleados*, que agrega todos los objetos creados de la clase Empleado. Para que la clase contenedor sea útil, debe proporcionar alguna forma de buscar objetos de empleados específicos. Se puede querer buscar por número de empleado, por el nombre y los apellidos o por el número de teléfono. La clase contenedor *TodosLosEmpleados* tendrá tres métodos llamados **encuentra**. Uno de los métodos requiere un entero como parámetro (el número de empleado), el segundo requiere dos cadenas (el nombre y los apellidos) y el tercero requiere una sola cadena (el número de teléfono). Aunque los tres métodos tienen el mismo nombre, poseen distintas interfaces públicas. La ventaja de la sobrecarga de los métodos es que presentan una interfase consistente al programador: siempre que quiera localizar a un empleado, debe utilizar el método **encuentra**. [4]

- **Polimorfismo.** En general, las subclases heredan los métodos de sus superclases y los utilizan como si fueran suyos. Sin embargo, en algunas ocasiones no es posible escribir un método genérico que pueda ser usado por todas las subclases. La clase *ObjetoGeometrico* posee un método *area* que deberá tener distinta implementación para sus subclases *Circulo*, *Rectangulo* y *Triangulo*. La superclase contendrá un prototipo para el método que calcula el área, indicando sólo su interfase pública. Cada subclase redefine el método, añadiendo las instrucciones necesarias para calcular su área. Nótese que polimorfismo no es lo mismo que sobrecarga: la sobrecarga se aplica a métodos de la misma clase y el polimorfismo se aplica a métodos de varias subclases de la misma superclase. [4]

- **Herencia.** Será explicada con mayor profundidad y con ejemplos más adelante.

Comparación con los modelos de BD tradicionales

Como los modelos **entidad-relación** y **relacional** se fundamentan básicamente sobre el mismo principio de tablas y relaciones entre las mismas por medio de claves ajenas nos referiremos indistintamente a cualquiera de ellos como modelo relacional haciendo algunas aclaraciones donde sea útil.

Las bases de datos relacionales representan las relaciones mediante las claves ajenas. No tienen estructuras de datos que formen parte de la base de datos y que representen estos enlaces entre tablas. Las relaciones se utilizan para hacer concatenaciones (join) de tablas. Por el contrario, las bases de datos orientadas a objetos implementan sus relaciones incluyendo en cada objeto los identificadores de los objetos con los que se relaciona. [4]

Un identificador de objeto es un atributo interno que posee cada objeto. Ni los programadores, ni los usuarios que realizan consultas de forma interactiva, ven o manipulan estos identificadores directamente. Los identificadores de los objetos los asigna el SGBD y es él el único que los utiliza. [4]

El modelo orientado a objetos permite los atributos **multivaluados, agregaciones a las que se denomina conjuntos (sets) o bolsas (bags)**. Para crear una relación de uno a muchos, se define un atributo en la parte del uno que será de la clase del objeto con el que se relaciona. Este atributo contendrá el identificador de objeto del padre. La clase del objeto padre contendrá un atributo que almacenará un conjunto de valores: los identificadores de los objetos hijo con los que se relaciona. Cuando el SGBD ve que un atributo tiene como tipo de datos una clase, ya sabe que el atributo contendrá un identificador de objeto. [4]

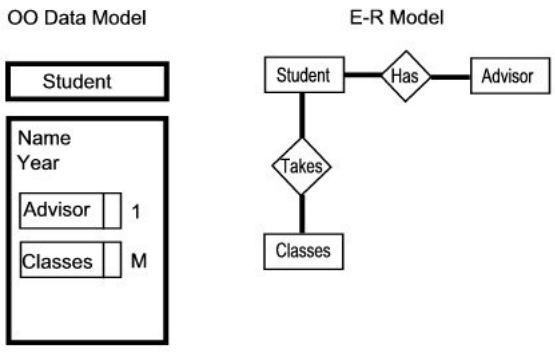
En el modelo relacional los datos se organizan en filas (tuplas) y columnas (campos). A su vez, las tuplas se organizan en tablas. Las filas son instancias específicas de una entidad (registro). Las columnas son características de esa entidad (registro). Una tabla separada modelaría la relación entre dos entidades (registros). En esta tabla, cada fila sería una relación, y cada columna sería el identificador único (clave primaria) de las dos entidades (registros) en la relación. La unión, a través de la tabla, provee una relación entre las dos entidades. [2]

Las relaciones de muchos a muchos se pueden representar directamente en las bases de datos orientadas a objetos, sin necesidad de crear entidades intermedias. Para representar la relación, cada clase que participa en ella define un atributo que contendrá un conjunto de valores de la otra clase con la que se relacionará. [4]

Ya que el paradigma orientado a objetos soporta la herencia, una base de datos orientada a objetos también puede utilizar la relación "es un" entre objetos. Por ejemplo, en una base de datos para un departamento de recursos humanos habrá una clase genérica Empleado con diversos atributos: nombre, dirección, número del seguro social, fecha de contrato y departamento en el que trabaja. Sin embargo, para registrar el modo de pago de cada empleado hay un dilema. No a todos los empleados se les paga del mismo modo: a algunos se les paga por horas, mientras que otros tienen un salario mensual. La clase de los empleados que trabajan por horas necesita unos atributos distintos que la clase de los otros empleados. En una base de datos orientada a objetos se deben crear las dos subclases de empleados. Aunque el SGBD nunca creará objetos de la clase Empleado, su presencia en el diseño clarifica el diseño lógico de la base de datos y ayuda a los programadores de aplicaciones permitiéndoles escribir sólo una vez los métodos que tienen en común las dos subclases (serán los métodos que se sitúan en la clase Empleado). Esta capacidad no está presente en los modelos tradicionales. Sobre este ejemplo se profundizará más adelante. [4]

En teoría, una base de datos orientada a objetos debe soportar dos tipos de herencia: la relación "es un" y la relación "extiende". La relación "es un", que también se conoce como generalización/especialización, crea una jerarquía donde las subclases son tipos específicos de las superclases. Con la relación "extiende", sin embargo, una clase expande su superclase, en lugar de estrecharla en un tipo más específico. Por ejemplo, en la jerarquía de la clase Empleado, al igual que son necesarias clases para los empleados que realizan cada trabajo específico, hace falta guardar información adicional sobre los directores, que son empleados pero que también tienen unas características específicas. La base de datos incluirá una clase Director con un atributo para los empleados a los que dirige. En este sentido un director no es un empleado más específico sino un empleado con información adicional. [4]

Figure 1:



El modelo de datos orientado a objetos comparado con el modelo relacional. [2]

La **Figura 1** compara el modelo OODM con el relacional. Los dos modelos representan un Estudiante quien *tiene un* Consejero y *toma* Clases. El OODM de una clase contiene los atributos Nombre y Año. Sirve como una clase contenedora que contiene el objeto/entidad Consejero. Por su parte, el modelo relacional tiene tres tablas de entidades: Estudiante, Consejero, y Clases. Además, el modelo relacional debe tener dos tablas de relaciones para reunir la información de manera que tenga sentido. Como uno puede ver, la técnica relacional (E-R específicamente en la figura) necesita de dos entidades (tablas) más para proveer las relaciones que la aproximación OODM sugiere implícitamente. [2]

La idea del OODM de una clase se asemeja a la idea de entidad de un conjunto de entidades o tabla del modelo E-R. Como un objeto, las clases del OODM son más potentes que la idea de conjunto de entidades o tabla del modelo E-R. Una clase no solo describe la estructura de datos, sino que describe el comportamiento de los objetos de esa clase. Características tales como métodos, los cuales permiten la descripción del comportamiento de los objetos, dan a una OODB capacidades completas para Tipos de Datos Abstractos (ADT por sus iniciales en inglés - Abstract Data Type) permitiendo un incremento en la semántica del objeto que se está modelando. Esta capacidad ADT completa también permite soporte de encapsulación y herencia. Estas capacidades proveen mecanismos para *triggers* (restricciones de tipo) en las bases de datos, los cuales sólo soportan unas pocas bases de datos SQL. [2]

Otra diferencia entre las bases de datos tradicionales y los OODBs es el *soporte de versiones*. Así como un documento puede tener múltiples versiones también puede tenerlas una base de datos. Este es especialmente un rasgo característico deseado para un sistema tal como CAD o CASE. Uno es capaz de cargar un sistema y cambiar aspectos para determinar como esto afectaría al sistema en general. Luego el sistema original puede ser cargado nuevamente intacto. [2]

OODBMS

Uno no puede abordar el tema de OODB sin brindar una solución de administración, la cual es un sistema de administración de bases de datos orientadas a objetos (OODBMS). Un OODBMS administra realmente el acceso a los datos y la consulta de los datos desde las bases de datos. Administra múltiples usuarios tratando de acceder a los datos al mismo tiempo, brinda servicio de recuperación de desastres, y administra todas las bases de datos en el sistema. Un OODBMS es un componente importante, ya que provee y administra las restricciones de las bases de datos. Esto es el "Caché" en nuestro caso.

Características como persistencia, control de concurrencia y transacciones y seguridad/administración todavía no están en un estándar establecido. Para responder a este dilema se presentó el "Manifiesto de Sistemas Orientados a Objetos" en la *Primera Conferencia Internacional sobre Bases de Datos Deductivas y Orientadas a Objetos* en Kyoto, Japón. Brinda los siguientes "debería" para un OODBMS: [2]

1. *Los OODBMS deberían soportar objetos complejos.* [2]
2. *Los OODBMS deberían soportar identidad de objetos.* [2]

3. *Los OODBMS deberían encapsular vuestros objetos.* [2]
4. *Los OODBMS deberían soportar tipos o clases.* [2]
5. *Las clases o tipos del OODBMS types deberían heredar de sus ancestros.* [2]
6. *Los OODBMS no deberían vincular en forma prematura.* Soportarán vinculación tardía. [2]
7. *Los OODBMS deberían ser computacionalmente completos.* Nociones básicas de programación se soportan en el Lenguaje de Manipulación de Bases de Datos. [2]
8. *Los OODBMS deberían ser extensibles*. [2]
9. *Los OODBMS deberían recordar vuestros datos.* Los objetos mantienen sus datos en memoria, a diferencia de una base de datos. Esos datos se pierden al apagar los equipos, pero esto no es aceptable. Se debe brindar una forma para almacenar los objetos en almacenamiento secundario. [2]
10. *Los OODBMS deberían administrar bases de datos muy grandes.* [2]
11. *Los OODBMS deberían aceptar usuarios concurrentes*. Los OODBMS deberían soportar el mismo nivel de concurrencia que las bases de datos presentes. [2]
12. *Los OODBMS deberían recuperarse de fallas de hardware y software.* Los OODBMS deben soportar el mismo nivel de protección de fallas que las bases de datos presentes. [2]
13. *Los OODBMS deberían tener una forma sencilla de consultar los datos.* debe disponerse de un método eficiente para consultar, similar a SQL. [2]

El Manifiesto es el primer intento de describir una norma en la cual deberían basarse los OODBMS. Es un primer paso importante hacia el acuerdo de los requisitos mínimos que un OODBMS debería soportar. Una vez que una norma esté en uso, los OODBMS y OODB se convertirán en corriente dominante mayor. Además, ya que es la lista de requisitos más significativa a ser compilada, la mayoría de los OODBMS se medirán contra ella. [2]

OODB

Mensajes

Todas las interacciones entre cada objeto y el resto del sistema se realizan mediante *mensajes*. Por lo tanto, la interfaz entre cada objeto y el resto del sistema se define mediante un conjunto de mensajes permitidos. [1]

En general, cada objeto está asociado con: [1]

a. Un conjunto de *variables* que contienen los datos relacionados con el objeto; **las variables** se corresponden con **los atributos** (campos) del modelo E-R (relacional). [1]

b. Un conjunto de *mensajes* a los que responde; cada mensaje puede tener ninguno o varios parámetros. [1]

c. Un conjunto de *métodos*, cada uno de los cuales es código que implementa un mensaje; el método devuelve un valor como respuesta al mensaje. [1]

El término *mensaje* en un entorno orientado a objetos hace referencia al intercambio de solicitudes entre los objetos, independientemente de los detalles concretos de su implementación. Se utiliza a veces la expresión *invocar un método* para denotar el hecho de enviar un mensaje a un objeto y la ejecución del método correspondiente. [1]

Se puede explicar la razón del uso de este enfoque con el ejemplo de los empleados introducido anteriormente. Consideremos las entidades EMPLEADO de una base de datos bancaria. Supóngase que el sueldo anual de cada empleado se calcula de manera diferente para los distintos empleados. Por ejemplo, puede que los jefes obtengan una prima en función de los resultados obtenidos, mientras que los cajeros reciben una prima en función de las horas trabajadas. Se puede encapsular el código para calcular el sueldo de cada empleado en forma de método que se ejecute en respuesta a un mensaje de *sueldo-anual*. [1]

Todos los objetos EMPLEADO responden al mensaje *sueldo-anual*, pero lo hacen de manera diferente. Al encapsular con el objeto EMPLEADO la información sobre el cálculo de su sueldo anual, todos los objetos EMPLEADO presentan la misma interfaz. Dado que la única interfaz externa presentada por cada objeto es el conjunto de mensajes a los que responde, resulta posible modificar las definiciones de los métodos y de las variables sin afectar al resto del sistema. La posibilidad de modificar la definición de un objeto sin afectar al resto del sistema se considera una de las mayores ventajas del paradigma de la programación orientada a objetos. [1]

Si queremos ser estrictos, en el modelo orientado a objetos hay que expresar cada atributo de las entidades como una variable y un par de mensajes del objeto correspondiente. La variable se utiliza para guardar el valor del atributo, uno de los mensajes se utiliza para leer dicho valor y el otro mensaje se utiliza para actualizarlo, sin embargo, en aras de la sencillez, muchos modelos orientados a objetos permiten que las variables se lean o se actualicen de manera directa, sin necesidad de definir los mensajes para ello. [1]

Clases de objetos

Como se dijo anteriormente, si una entidad (tabla) en el modelo E-R agrupa un conjunto de registros similares (cantidad, tipo y nombres de campos iguales), una **clase** en el modelo OO agrupa un conjunto de objetos similares (responden a los mismos mensajes, utilizan los mismos métodos y tienen variables del mismo nombre y del mismo tipo). [1]

Cada uno de los objetos en el modelo OO se denomina *ejemplar* de su clase. Todos los objetos de una clase comparten una definición común, pese a que se diferencien en los valores asignados a sus variables. [1]

A continuación se muestra un ejemplo de la definición de la clase EMPLEADO escrita en pseudocódigo. La definición muestra las variables y los mensajes a los que responden los objetos de la clase; no se muestran aquí los métodos para el tratamiento de los mensajes. [1]

```
class empleado {
        /* Variables */
        string nombre;
        string dirección;
        date fecha-alta;
        int sueldo;

        /* Mensajes */
        int sueldo-anual();
```

```
        string obtener-nombre();
        string obtener-dirección();
        void definir-nombre(string nuevo_nombre);
        void definir-dirección(string nueva_dirección);
        int antigüedad();
}; [1]
```

En la definición anterior, cada objeto de la clase *empleado* contiene las variables: *nombre, dirección, fecha-alta, y sueldo*. Cada objeto responde a los seis mensajes mostrados, es decir, *sueldo-anual, obtener-nombre, obtener-dirección, definir-nombre, definir-dirección y antigüedad*. [1]

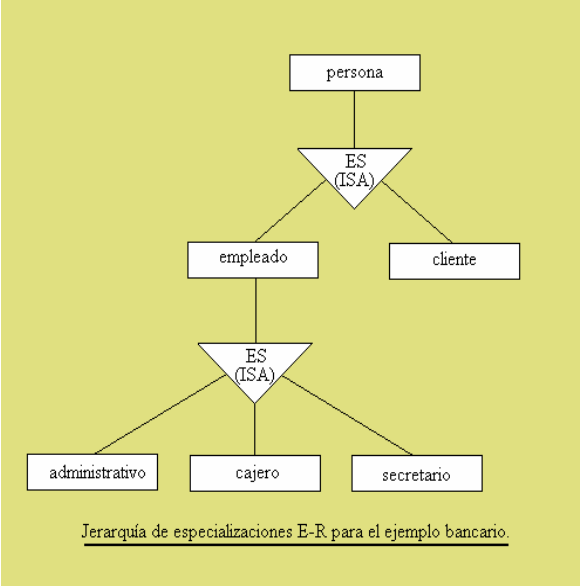


Figura 1 [1]

Herencia

Los esquemas de las bases de datos OO suelen necesitar gran número de clases. Frecuentemente, sin embargo, varias de las clases son parecidas entre sí. Por ejemplo, supóngase que se tiene una base de datos orientada a objetos en la aplicación bancaria. Cabe esperar que la clase de los clientes del banco sea parecida a la clase de los empleados en que ambas definan variables para *nombre, dirección, etc*. Sin embargo, hay algunas variaciones específicas de los empleados (*sueldo*, por ejemplo) y otras específicas de los clientes (*interés-préstamo*, por ejemplo). Es conveniente definir una representación de las variables comunes en un solo lugar. Esto sólo puede hacerse si se combinan los empleados y los clientes en una sola clase. [1]

Para permitir la representación directa de los parecidos entre las clases, hay que ubicarlas en una jerarquía de especializaciones (la relación <<ES>> o <<ISA>> en el modelo E-R). Por ejemplo, se puede decir que EMPLEADO es una especialización de PERSONA, dado que el conjunto de los *empleados* es un subconjunto del conjunto de *personas*. Es decir, todos los empleados son personas. De manera parecida, CLIENTE es una especialización de PERSONA. El concepto de jerarquía de clases, como se puede ver, es análogo al de especialización en el modelo E-R. Los detalles se pueden ver en la figura 1 de esta misma página. [1]

Las variables y los métodos específicos de los clientes se asocian con la clase CLIENTE. Las variables y los métodos que se aplican tanto a empleados como a clientes se asocian con la clase PERSONA. [1]

En la figura 2 se muestra un diagrama con una jerarquía de especializaciones que representa a las personas implicadas en la operación del ejemplo bancario. [1]

La jerarquía de clases correspondiente al modelo OO se define según la misma estructura de árbol. Las especializaciones de las clases se denominan *subclases* y dichas clases se denominan *superclases*. Esta propiedad de que los objetos de una clase contengan las variables y métodos definidos en sus superclases se denomina *herencia*. [1]

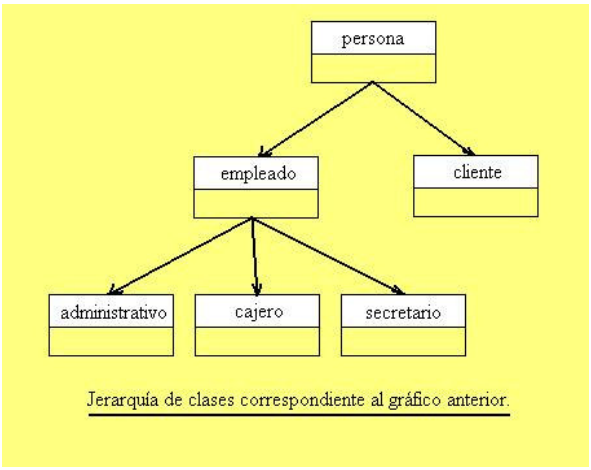


Figura 2 [1]

Los métodos se heredan de modo idéntico a las variables. Una ventaja importante de la herencia en los sistemas orientados a objetos es el concepto de *posibilidad de sustitución*: cualquier método de una clase dada -por ejemplo, A (o una función que reciba un objeto de la clase A como argumento)- puede ser llamado de igual modo mediante cualquier objeto perteneciente a cualquier subclase B de A. Esta característica lleva a la reutilización del código, dado que no hace falta volver a escribir los métodos y las funciones para los objetos de la clase B. Por ejemplo, si se define el método *obtener_nombre* para la clase PERSONA, se lo puede llamar con un objeto *persona* o con cualquier objeto perteneciente a cualquiera de las subclases de PERSONA, como CLIENTE o ADMINISTRATIVO. [1]

Figura 3 [1]
Definición en seudocódigo de una jerarquía de clases

```
class persona {
        string nombre;
        string dirección;
};

class cliente isa persona {
        int interés_préstamo;
};

class empleado isa persona {
        date fecha_alta;
        int sueldo;
};

class administrativo isa empleado {
        int número_despacho;
        int número_cuenta_corriente;
};
```

```
class cajero isa empleado {
        Int horas_semana;
        string jefe;
};
```

Resulta sencillo determinar los objetos que se hallan asociados con las clases en las hojas de la jerarquía. Por ejemplo, se asocia con la clase CLIENTE el conjunto de los *clientes del banco*. Para las clases no hojas, sin embargo, el problema resulta más complejo. En la jerarquía de la **Figura 2** hay dos maneras posibles de asociar los objetos con las clases: [1]

a. Se pueden asociar todos los *objetos empleado*, incluyendo aquellos que sean elementos de ADMINISTRATIVO, de CAJERO o de SECRETARIO, con la clase EMPLEADO. [1]

b. Sólo se pueden asociar con la clase empleado los *objetos empleado* que no sean elementos de ADMINISTRATIVO ni de CAJERO ni de SECRETARIO. [1]

La mayor parte de los sistemas orientados a objetos permiten que la especialización sea parcial: es decir, permiten objetos que pertenezcan a una clase, como EMPLEADO, pero que no pertenezcan a ninguna de las subclases de la misma. [1]

Herencia múltiple

En la mayor parte de los casos una organización de clases con estructura de árbol resulta adecuada para describir las aplicaciones. Sin embargo, hay situaciones que no pueden representarse bien en una jerarquía de clases con estructura de árbol de este tipo. [1]

Supóngase que en el ejemplo de la **Figura 2** se desea distinguir entre cajeros y secretarios que trabajan tiempo completo y los que trabajan por horas. Supóngase además que se necesitan variables y métodos diferentes para representar a los empleados que trabajan tiempo completo y los que trabajan a tiempo parcial. Por tanto, cada uno de los empleados se clasifica de dos maneras diferentes. Se pueden crear las subclases: [1]

1. CAJERO_POR_HORAS [1]
2. CAJERO_A_TIEMPO_COMPLETO [1]
3. SECRETARIO_POR_HORAS [1]
4. SECRETARIO_A_TIEMPO_COMPLETO [1]

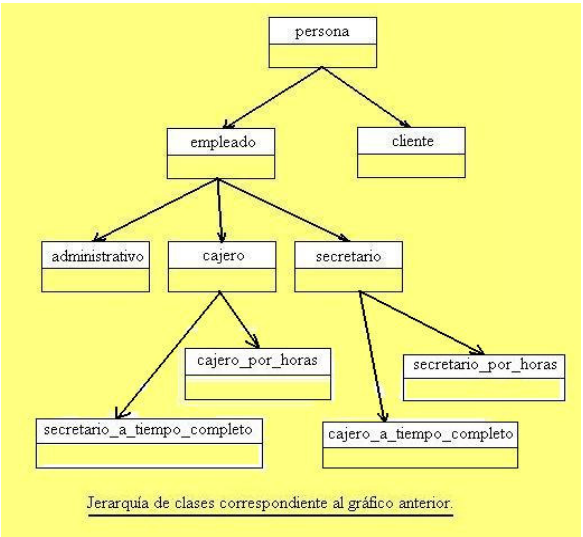


Figura 4 [1]

La jerarquía resultante, que se muestra en la **Figura 4**, presenta algunos inconvenientes: [1]

a. Como ya se observó, hay ciertas variables y ciertos métodos específicos de los contratos a tiempo completo y otros específicos de los contratos temporales (por horas). En la **Figura 4** hay que definir dos veces las variables y los métodos de los empleados a tiempo completo: una para los *secretarios_a_tiempo_completo* y otra para los *cajeros_a_tiempo_completo*. Existe una redundancia parecida en los empleados por horas. Este tipo de redundancia no es deseable, dado que hay que realizar en dos sitios distintos cualquier cambio de las propiedades de los empleados a tiempo completo o por horas, lo que lleva a una posible inconsistencia. Además, la jerarquía no puede aprovechar la posible reutilización del código. [1]

b. La jerarquía no tiene ningún medio de representar a los empleados que no sean administrativos, cajeros o secretarios, a menos que se la siga extendiendo para incluir las clases EMPLEADO_A_TIEMPO_COMPLETO y EMPLEADO_POR_HORAS. [1]

Si tuviéramos varias clasificaciones de puestos de trabajo en lugar de dos, como en el ejemplo, las limitaciones del modelo se harían aun más patentes. [1]

Las dificultades que se acaban de observar se resuelven en el modelo orientado a objetos mediante el concepto de herencia múltiple, que es la posibilidad que tienen las clases de heredar variables y métodos de varias superclases. La relación entre clases y subclases se representa mediante un grafo acíclico dirigido (GAD), en el que las clases pueden tener más de una superclase. Volviendo ahora al ejemplo bancario, mediante un GAD se pueden definir las propiedades de los *contratos a tiempo completo* y de los *contratos temporales* en un solo sitio. Como se muestra en la **Figura 5** de la siguiente página, se define una clase TEMPORAL, que define las variables y los métodos específicos de los contratos por horas, y otra clase TIEMPO_COMPLETO, que define las variables y los métodos específicos de los *contratos a tiempo completo*. [1]

La clase CAJERO_POR_HORAS es una subclase de CAJERO y de TEMPORAL. Hereda de CAJERO las variables y los métodos correspondientes a los *cajeros* y de TEMPORAL las variables y los métodos correspondientes a los *contratos temporales*. Se elimina la redundancia de la jerarquía de la **Figura 4. [1]**

Mediante el GAD de la **Figura 5** se pueden representar *empleados a tiempo completo* y a *empleados por horas* que no son ni *cajeros* ni *secretarios* como elementos de las clases TIEMPO_COMPLETO o TEMPORAL. [1]

Cuando se utiliza la herencia múltiple, aparece una ambigüedad potencial si se puede heredar la misma variable o el mismo método de más de una superclase. En el ejemplo bancario, supóngase que en lugar de definir sueldo para la clase empleado, se define de la manera siguiente: una variable *paga* para cada una de las clases TIEMPO_COMPLETO, TEMPORAL, CAJERO y SECRETARIO. [1]

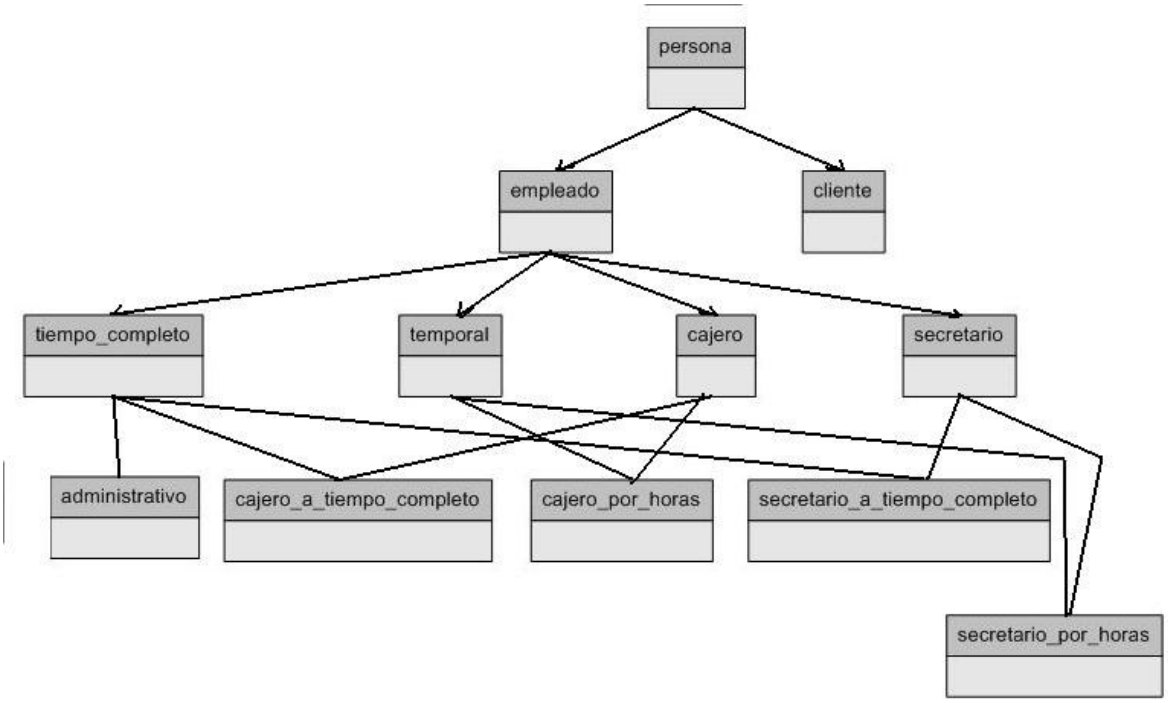


Figura 5 [1]

Las siguientes son las distintas variaciones del atributo *paga:*

a. TIEMPO_COMPLETO: *paga* es un entero de 0 a 20.000.000 que contiene el sueldo anual. [1]

b. TEMPORAL: *paga* es un entero de 0 a 4.000 que contiene el sueldo por hora. [1]

c. CAJERO: *paga* es un entero de 0 a 4.000.000 que contiene el sueldo anual. [1]

d. SECRETARIO: *paga* es un entero de 0 a 5.000.000 que contiene el sueldo anual. [1]

Considérese la clase SECRETARIO_POR_HORAS. Puede heredar la definición de *paga* desde TEMPORAL o desde SECRETARIO. El resultado es diferente. Según la opción elegida. Entre las opciones escogidas por varias de las implementaciones del modelo orientado a objetos están las siguientes: [1]

a. Incluir las dos variables, generando nuevos nombres para cada atributo, por ejemplo: *paga_por_horas* y *paga_secretario.* [1]

b. Escoger una de las dos según el orden en que se crearon las clases TEMPORAL y SECRETARIO. [1]

c. Hacer que el usuario escoja de manera explícita una de las opciones en la nueva definición. [1]

d. Tratar la situación como si fuera un error. [1]

Ninguna de estas soluciones se ha aceptado como óptima y los diferentes sistemas implementan opciones diferentes. [1]

No todos los casos de herencia múltiple llevan a ambigüedad. Si en lugar de definir *paga* se conserva la definición de la variable *sueldo* en la clase EMPLEADO y no se define en ningún otro sitio, todas las clases: TEMPORAL, TIEMPO_COMPLETO, SECRETARIO y CAJERO heredan sueldo de empleado. Dado que estas cuatro clases comparten la misma definición de sueldo, no surge ninguna ambigüedad de la herencia de *sueldo* por SECRETARIO_POR_HORAS, SECRETARIO_A_TIEMPO_COMPLETO, etc. [1]

Se puede utilizar la herencia múltiple para modelar el concepto de *roles*. Para comprender este concepto se puede considerar una base de datos universitaria. Puede que la base de datos tenga varias subclases de PERSONA, como ESTUDIANTE, PROFESOR y JUGADOR_DE_FUTBOL. Los objetos pueden pertenecer a varias de las categorías de manera simultánea y cada una de ellas se denomina *rol*. Se puede utilizar la herencia múltiple para crear subclases como ESTUDIANTE_PROFESOR, ESTUDIANTE_JUGADOR_DE_FUTBOL, etc., para modelar la posibilidad de que un objeto tenga varios *roles* de manera simultánea. Los problemas de la herencia múltiple en los lenguajes de definición de datos. [1]

Identidad de los objetos

A pesar de que ya se habló sobre este tema en apartados anteriores profundizaremos un poco sobre este concepto. Los objetos de las bases de datos orientadas a objetos generalmente corresponden a entidades del sistema modelado por la base de datos. Las entidades conservan su identidad aunque algunas de sus propiedades cambien con el tiempo. De manera parecida, los objetos conservan su identidad aunque los valores de las variables o las definiciones de sus métodos cambien total o parcialmente con el tiempo. Este concepto de identidad no se aplica a las tuplas de las bases de datos relacionales. En los sistemas relacionales, las tuplas de una relación sólo se distinguen por los valores que contienen. [1]

La *identidad* de los objetos es un concepto de identidad más potente que el que suele hallarse en los lenguajes de programación o en los modelos de datos que no se basan en la programación orientada a objetos. A continuación se ilustran varios ejemplos de identidad: [1]

a. **Valor**. Se utiliza un valor de datos como identidad. Esta forma de identidad se utiliza en los sistemas relacionales. Por ejemplo, el valor

de la clave primaria de una tupla identifica a la tupla. [1]

b. **Nombre**. Se utiliza como identidad un nombre proporcionado por el usuario. Esta forma de identidad suele utilizarse para los archivos en los sistemas de archivos. Cada archivo recibe un nombre que lo identifica de manera unívoca, independientemente de su contenido. [1]

c. **Incorporada**. No hace falta que el usuario proporcione ningún identificador. Cada objeto recibe del sistema de manera automática un identificador en el momento en que se crea. [1]

La identidad de los objetos es una noción conceptual; los sistemas reales necesitan un mecanismo físico que identifique los objetos de manera unívoca. Para los seres humanos se suelen utilizar como identificadores los nombres, junto con otra información, como la fecha y el lugar de nacimiento. Los sistemas orientados a objetos proporcionan el concepto de *identificador del objeto*. Los identificadores de los objetos son únicos; es decir, cada objeto tiene un solo identificador y no hay dos objetos que tengan el mismo identificador. Los identificadores de los objetos no tienen por qué estar en una forma con la que los seres humanos se encuentren cómodos, pueden ser números grandes, por ejemplo. La posibilidad de guardar el identificador de un objeto como un campo de otro objeto es más importante que tener un nombre que resulte fácil de recordar. [1]

Como ejemplo del uso de los identificadores de los objetos, uno de los atributos de los objetos *persona* puede ser el atributo *cónyuge*, que es en realidad un identificador del objeto *persona* correspondiente al cónyuge de la primera persona. Por tanto, el objeto persona puede guardar una referencia del objeto que representa al cónyuge de esa persona. [1]

Generalmente, al identificador lo genera el sistema de manera automática. Por el contrario, en las bases de datos relacionales el campo *cónyuge* de la relación *persona* será generalmente un identificador unívoco (quizás un número de CI) del cónyuge de esa persona generado de manera externa al sistema de bases de datos. Hay muchas situaciones en las que hacer que el sistema genere identificadores de manera automática resulta una ventaja, dado que libera a los seres humanos de llevar a cabo esa tarea. Sin embargo, esta posibilidad debe utilizarse con precaución. Los identificadores generados por el sistema suelen ser específicos del mismo, y si se desplazan los datos a un sistema de bases de datos diferente, hay que traducirlos. Los identificadores generados por el sistema pueden resultar redundantes si las entidades que se modelan ya disponen de identificadores unívocos externos al sistema. Por ejemplo, los números de CI suelen utilizarse como identificadores unívocos de las personas. [1]

Lenguajes Orientados a Objetos

Hasta el momento se han abordado los conceptos básicos de la orientación a objetos en un nivel abstracto. Para poder utilizarlos en la práctica en un sistema de bases de datos hay que expresarlos mediante algún lenguaje. Esta expresión puede realizarse de dos maneras: [1]

a) Los conceptos de la programación orientada a objetos se utilizan simplemente como herramientas de diseño y se codifican, por ejemplo, en una base de datos relacional. Se sigue este enfoque cuando se utilizan los diagramas entidad-relación para modelar los datos y luego se convienen de manera manual en un conjunto de relaciones. [1]

b) Los conceptos de la programación orientada a objetos se incorporan en un lenguaje que se utiliza para trabajar con la base de datos. Con este enfoque, hay varios lenguajes posibles en los que se pueden integrar los conceptos. [1]

   i) Una opción es extender un lenguaje para el tratamiento de datos como SQL añadiendo tipos complejos y la programación orientada a objetos. Los sistemas que proporcionan extensiones orientadas a objetos a los sistemas relacionales se denominan *sistemas relacionales orientados a objetos*. [1]

   ii) Otra opción es tomar un lenguaje de programación orientado a objetos ya existente y extenderlo para que trabaje con las bases de datos. Estos lenguajes se denominan *lenguajes de programación persistentes*. [1]

Lenguajes de programación persistentes

Los lenguajes de las bases de datos se diferencian de los lenguajes de programación tradicionales en que trabajan directamente con datos que son persistentes, es decir, los datos siguen existiendo una vez que el programa que los creó ha concluido. Las relaciones de las bases de datos y las tuplas de las relaciones son ejemplos de datos persistentes. Por el contrario, los únicos datos persistentes con los que los lenguajes de programación tradicionales trabajan directamente son los archivos. [1]

El acceso a las bases de datos es sólo un aspecto de las aplicaciones del mundo real. Mientras que los lenguajes para el tratamiento de datos como SQL son bastante efectivos en el acceso a los datos, hace falta un lenguaje de programación para implementar otros aspectos de las aplicaciones, como las interfaces de usuario o la comunicación con otras computadoras. [1]

La manera tradicional de realizar las interfaces de las bases de datos con los lenguajes de programación es incorporar SQL dentro del lenguaje de programación sin extender el lenguaje propiamente dicho con nueva sintaxis para el tratamiento de bases de datos. [1]

Los lenguajes de programación persistentes son lenguajes de programación extendidos para el tratamiento de datos persistentes (sin utilizar el lenguaje SQL). [1]

Los lenguajes de programación persistentes pueden distinguirse de los lenguajes con SQL incorporado de al menos dos maneras: [1]

1) En los lenguajes con SQL incorporado, el sistema de tipos del lenguaje anfitrión suele ser diferente del sistema de tipos del lenguaje para el tratamiento de los datos (SQL en este caso). Los programadores son responsables de las conversiones de tipos entre el lenguaje anfitrión y el lenguaje para la manipulación de los datos. Exigir que los programadores ejecuten esta tarea presenta varios inconvenientes: [1]

a. El código para la conversión entre objetos y tuplas opera fuera del sistema de tipos orientado a objetos, y por tanto tiene más posibilidades de presentar errores no detectados; [1]

b. La conversión en la base de datos entre el formato orientado a objetos y el formato relacional de las tuplas necesita gran cantidad de código. El código para la conversión de formatos, junto con el código para cargar y descargar los datos de la base de datos, puede suponer un porcentaje significativo del código total necesario para la aplicación. Por el contrario, en los lenguajes de programación persistentes el lenguaje de consulta se halla totalmente integrado con el lenguaje anfitrión y ambos comparten el mismo sistema de tipos. Los objetos se pueden crear y guardar en la base de datos sin ningún tipo explícito ni cambios de formato: los cambios de formato necesarios se realizan de manera transparente. [1]

2) Los programadores que utilizan lenguajes de consulta incorporados son responsables de la escritura de código explícito para la búsqueda de los datos de la base de datos en la memoria. Si se realizan actualizaciones, los programadores deben escribir código de manera explícita para volver a guardar los datos actualizados en la base de datos. Por el contrario, en los lenguajes de programación persistentes los programadores pueden trabajar con datos persistentes sin tener que escribir de manera explícita código para buscarlos en la memoria o para volver a guardarlos en el disco. [1]

Se han propuesto versiones persistentes de los lenguajes de programación como Pascal. En los últimos años han recibido mucha atención las versiones persistentes de los lenguajes orientados a objetos como C++ o Smalltalk. Permiten a los programadores trabajar directamente con los datos desde el lenguaje de programación, sin tener que pasar por un lenguaje para el tratamiento de los datos como SQL. Por tanto, proporcionan una mayor integración de los lenguajes de programación con las bases de datos que, por ejemplo, SQL incorporado. [1]

Sin embargo, los lenguajes de programación persistentes presentan ciertos inconvenientes que hay que tener presente al decidir si conviene utilizarlos: [1]

1) Dado que los lenguajes de programación suelen ser potentes, resulta relativamente sencillo cometer errores de programación que dañen las bases de datos. [1]

2) La complejidad de los lenguajes hace que la optimización automática de alto nivel, como la reducción de E/S de disco, resulte más difícil. [1]

3) En muchas aplicaciones, la posibilidad de las consultas declarativas resulta de gran importancia, pero los lenguajes de programación persistentes no permiten actualmente las consultas declarativas sin que aparezcan problemas de algún tipo. [1]

Persistencia de los objetos

Los lenguajes de programación orientados a objetos no contemplan este concepto. Los objetos creados por ellos son transitorios, desaparecen en cuanto se termina el programa, igual que ocurre con las variables de los programas en Pascal o en C. Si se desea transformar uno de estos lenguajes en un lenguaje para la programación de bases de datos, el primer paso consiste en proporcionar una manera de hacer persistentes a los objetos. Se han propuesto varios enfoques: [1]

a. **Persistencia por clases**. El enfoque más sencillo, pero el menos conveniente, consiste en declarar que una clase es persistente. Todos los objetos de la clase son, por tanto, persistentes de manera predeterminada. Todos los objetos de las clases no persistentes son transitorios. Este enfoque no es flexible, dado que suele resultar útil disponer en una misma clase tanto de objetos transitorios como de objetos persistentes. En muchos sistemas de bases de datos orientados a objetos, la declaración de que una clase es persistente se interpreta como si se afirmara que los objetos de la clase pueden hacerse persistentes en vez de que se interprete como que todos los objetos de la clase son, si o si, persistentes. Estas clases se podrían haber denominado con más propiedad clases «que pueden ser persistentes». [1]

b. **Persistencia por creación**. En este enfoque se introduce una sintaxis nueva para crear los objetos persistentes mediante la extensión de la sintaxis para la creación de los objetos transitorios. Por tanto, los objetos son persistentes o transitorios en función de la manera de crearlos. Este enfoque se sigue en varios sistemas de bases de datos orientados a objetos. [1]

c. **Persistencia por marcas**. Una variante del enfoque anterior es marcar los objetos como persistentes después de haberlos creado. Todos los objetos se crean como transitorios, pero si un objeto tiene que persistir más allá de la ejecución del programa, hay que marcarlo de manera explícita antes de que el programa concluya. A diferencia del enfoque anterior, la decisión sobre la persistencia o la transitoriedad se retrasa hasta después de la creación del objeto. [1]

d. **Persistencia por referencia**. Uno o varios objetos se declaran objetos persistentes (objetos raíz) de manera explícita. Todos o algunos de los demás objetos serán persistentes si (y sólo si) se hace referencia a ellos de manera directa o indirecta desde un objeto persistente (objeto raíz). Por tanto, todos los objetos a los que se haga referencia desde los objetos persistentes (objetos raíz) serán persistentes. Pero también lo serán todos los objetos a los que se haga referencia desde este último y los objetos a los que estos últimos hagan referencia serán también persistentes, etc., etc. Este esquema tiene la ventaja de que resulta sencillo hacer que sean persistentes estructuras de datos completas con sólo declarar como persistente la raíz de las mismas. Sin embargo, el sistema de bases de datos sufre la carga de tener que seguir las cadenas de referencias para detectar los objetos que son persistentes, y eso puede resultar costoso. [1]

La identidad de los objetos y los punteros

Cuando se crean objetos persistentes se les asignan identificadores de objetos persistentes. En los lenguajes

de programación orientados a objetos que no se han extendido para tratar la persistencia, al crear objetos se obtienen identificadores de objetos transitorios. La única diferencia entre los dos tipos de identificadores es que los identificadores de objetos transitorios sólo son válidos mientras se ejecuta el programa que los creó; después de que concluya ese programa, el objeto se borra y el identificador pierde su sentido. [1]

El concepto de la identidad de los objetos tiene una relación interesante con los punteros de los lenguajes de programación. Una manera sencilla de conseguir una identidad intrínseca es utilizar los punteros a las ubicaciones físicas de almacenamiento. En concreto, en muchos lenguajes orientados a objetos como C++, los identificadores de los objetos son en realidad punteros internos de la memoria. [1]

Sin embargo, la asociación de los objetos con ubicaciones físicas de almacenamiento puede variar con el tiempo. Hay varios grados de permanencia de las identidades: [1]

a. **Dentro de los procedimientos**. La identidad sólo persiste durante la ejecución de un único procedimiento. Un ejemplo de la identidad dentro de los procedimientos son las variables locales del interior de los procedimientos; [1]

b. **Dentro de los programas**. La identidad sólo persiste durante la ejecución de un único programa o una única consulta. Un ejemplo de la identidad dentro de los programas son las variables globales de los lenguajes de programación. Los punteros de la memoria principal o de la memoria virtual sólo ofrecen identidad dentro de los programas. [1]

c. **Entre programas**. La identidad persiste de una ejecución del programa a otra. Los punteros a los datos del sistema de archivos del disco ofrecen identidad entre los programas, pero pueden cambiar si se modifica la manera en que los datos se guardan en el sistema de archivos. [1]

d. **Persistente**. La identidad no sólo persiste entre las ejecuciones del programa, sino también entre las reorganizaciones estructurales de los datos. Es la forma persistente de la identidad necesaria para los sistemas orientados a objetos. [1]

Integridad de las relaciones

Para que las relaciones funcionen en una base de datos orientada a objetos pura, los identificadores de los objetos deben corresponderse en ambos extremos de la relación. Por ejemplo, si los ingenieros de una empresa de control de calidad se deben relacionar con las obras de construcción que supervisan, debe haber algún modo de garantizar que, cuando un identificador de un objeto *Obra* se incluye en un objeto *Ingeniero*, el identificador de este mismo objeto *Ingeniero* se debe incluir en el objeto *Obra* anterior. Este tipo de integridad de relaciones, que es de algún modo análogo a la integridad referencial en las bases de datos relacionales, se gestiona especificando relaciones inversas. [4]

La clase *Ingeniero* tiene un atributo de tipo conjunto llamado "supervisa". Del mismo modo, la clase *Obra* tiene un atributo llamado "es_supervisada". Para garantizar la integridad de esta relación, un SGBD orientado a objetos puro deberá permitir que el diseñador de la base de datos pueda especificar dónde debe aparecer el identificador del objeto inverso, como por ejemplo: [4]

relationship Obra supervisa
    inverse Obra::es_supervisada [4]

en la clase Aparejador, y;

relationship Ingeniero es_supervisada
    inverse Ingeniero::supervisa [4]

en la clase *Obra*.

Siempre que un usuario o un programa de aplicación inserta o elimina un identificador de objeto de la relación supervisa en un objeto *Ingeniero*, el SGBD actualizará automáticamente la relación es_supervisada en el objeto *Obra* relacionado. Cuando se hace una modificación en el objeto *Obra*, el SGBD lo propagará automáticamente al objeto *Ingeniero.* [4]

Del mismo modo que en las bases de datos relacionales es el diseñador de la base de datos el que debe especificar las reglas de integridad referencial, en las bases de datos orientadas a objetos es también el diseñador el que debe especificar las relaciones inversas cuando crea el esquema de la base de datos. [4]

Bases de datos relacionales orientadas a objetos

Los lenguajes de programación persistentes añaden la persistencia y otras características de las bases de datos a los lenguajes de programación existentes con sistemas de tipos orientados a objetos. Por el contrario, los modelos de datos relacionales orientados a objetos extienden el modelo de datos relacional proporcionando un sistema de tipos más rico que incluye la programación orientada a objetos y añade constructores a los lenguajes de consulta relacionales como SQL para trabajar con los tipos de datos añadidos y las clases. Los sistemas de tipos extendidos permiten que los atributos de las tuplas tengan tipos complejos. Estas extensiones intentan conservar las bases relacionales (en concreto, el acceso declarativo a los datos) al tiempo que extienden la capacidad de modelado. Los sistemas de bases de datos relacionales orientados a objetos (es decir, los sistemas de bases de datos basados en el modelo objeto-relación) proporcionan un modo de cambio adecuado para los usuarios de las bases de datos relacionales que deseen utilizar características orientadas a objetos de manera que la migración no sea tan brusca. [1]

Demostración práctica para la exposición

Para las demostraciones prácticas en la exposición se utilizará una herramienta denominada **"Caché, The post-relational database"**. Esta herramienta incluye las siguientes capacidades.

a. Un poderoso motor de transacciones que incluye la habilidad de crear bases de datos OO distribuidas. [5]

b. Una arquitectura de datos que une el poder de los objetos con el alto *performance* del lenguaje SQL. [5]

c. Un conjunto de tecnologías y herramientas que permiten el rápido desarrollo de bases de datos OO y aplicaciones web. [5]

d. Soporte de XML object-based y servicios web. [5]

e. Interoperatividad vía Java, EJB, JDBC, ActiveX, .NET, C++, ODBC, XML, SOAP y otros. [5]

¿Qué queremos decir con Post-Relational?

Caché está diseñado para trascender sobre las limitaciones del modelo relacional otorgando al mismo tiempo un camino para la actualización/evolución de las miles de aplicaciones de bases de datos relacionales existentes así como soporte para muchas herramientas de generación de reportes basados en SQL existentes en el mercado. [5]

La parte "relacional" de "post-relational" se refiere al hecho de que Caché posee todas las características de las bases de datos relacionales. Todos los datos dentro de una base de datos de Caché están disponibles como si estuvieran en verdaderas tablas relacionales y pueden ser consultados y modificado utilizando el SQL Standard vía ODBC, JDBC, o con métodos orientados a objetos. [5]

La parte "post" de "post-relational" se refiere al hecho de que Caché ofrece características que van más allá de los límites de las bases de datos relacionales además de seguir soportando las características de las bases de datos relacionales normales. Algunas de estas características son: [5]

a. La habilidad de modelar los datos como objetos (cada uno de los cuales son creados y sincronizados con una representación relacional nativa). [5]

b. Un modelo object-based concurrente simple. [5]

c. Tipos de datos definidos por el usuario. [5]

d. Se dispone de una extensión del SQL que permite manejar identidades de objetos y relaciones. [5]

e. Habilidad para mezclar accesos SQL y accesos basados en objetos dentro de una sola aplicación, utilizando cada uno en donde resulte más conveniente. [5]

Objetos

Caché incluye todas las características de la siguiente generación de las bases de datos, las bases de datos orientadas a objetos. Está especialmente diseñado para cumplir con las necesidades de las complejas aplicaciones orientadas a transacciones. Se incluyen las siguientes características: [5]

a. **Clases**. Se pueden definir clases que representen el estado (datos) y el comportamiento (código) de los componentes de nuestras aplicaciones. Las clases son utilizadas para crear instancias de objetos que pueden ser, tanto componentes de tiempo de ejecución como elementos almacenados dentro de una base de datos. [2]

b. **Propiedades**. Las clases pueden incluir propiedades, los cuales especifican los datos asociados con cada instancia de objeto. Las propiedades pueden ser datos simples (ej.: strings o enteros), tipos definidos por el usuario (definidos utilizando clases), objetos complejos (o incrustados), colecciones o referencias a otros objetos. [5]

c. **Relaciones**. En las clases se pude definir cómo los objetos instanciados se relacionan unos con otros. El sistema provee métodos de navegación a través de las relaciones así como también proporciona integridad referencial dentro de las bases de datos. [5]

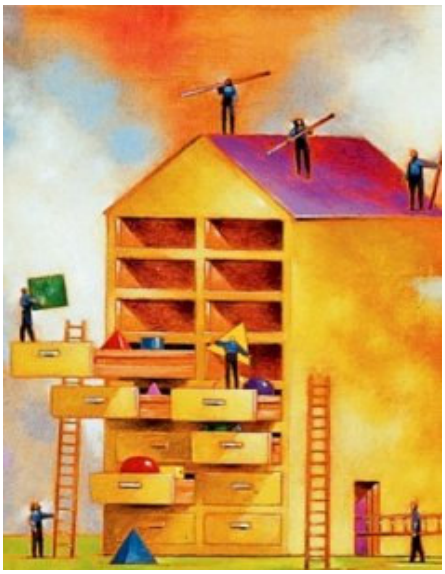
d. **Métodos**. Las clases pueden definir comportamientos por medio de métodos: código ejecutable asociado a un objeto. Los métodos de objetos pueden ser escritos utilizando el lenguaje del sistema (Caché ObjectScript), SQL, Basic o pueden ser generados utilizando el generador de métodos. Esta última característica no está disponible en la versión "free" del Caché. [5]

e. **Objetos persistentes**. Los objetos persistentes tienen la habilidad de almacenarse/recargarse ellos mismos en/desde una base de datos. El soporte de persistencia incluye un administrador de transacciones, control de concurrencia, mantenimiento de índices y validación de datos. Los objetos persistentes son automáticamente visibles por medio de consultas SQL. [5]

f. **Herencia.** Sobre este tema ya se ha hablado bastante. [5]

g. **Polimorfismo**. Esto significa que las aplicaciones pueden utilizar las interfaces de los objetos (conjuntos de métodos y propiedades disponibles para otros objetos) y dejar que el sistema invoque la implementación adecuada de la interfaz, basado en el tipo de cada objeto. [5]

h. **"Lazy loading".** Caché recarga, desde el disco duro a la memoria, cualquier objeto persistente cuando este es referenciado por otro objeto. [5]



Una breve introducción

Definición de clases

El Caché ObjectScript es un lenguaje de programación orientado a objetos diseñado para el desarrollo rápido y eficiente de aplicaciones que utilicen bases de datos orientadas a objetos. [5]

A continuación se presenta un breve resumen de los principales aspectos del Caché ObjectScript para que se tenga una idea inicial sobre su sintaxis. [5]

La forma más simple y común de definir clases en Caché es utilizando su interfaz gráfica de desarrollo, el "Caché Studio". A continuación se presenta un ejemplo de definición de la clase COMPONENT como se vería en el Caché Studio: [5]

```
Class MyApp.Component Extends %Persistent [ClassType = persistent]
{
Property TheName As %String;
Property TheValue As %Integer;
}
```

[5]

Esta clase "COMPONENT" se define como una clase persistente (esto es, se puede almacenar dentro de una base de datos). En este caso, la clase %PERSISTENT, proveída por Caché (los nombres de las clases proveídas por el sistema comienzan con "%" para distinguirlas de las clases de la aplicación, creadas por el desarrollador) provee todo el código persistente necesario por medio de la herencia. La clase se agrupa en el paquete, "MyApp". Los "paquetes" agrupan clases relacionadas, lo que simplifica el desarrollo de grandes aplicaciones. La clase define dos propiedades: *TheName*, que contiene un valor tipo string, y *TheValue*, el cual almacena un valor entero. [5]

También se puede utilizar la sintaxis del Basic para, por ejemplo, manipular las instancias de los objetos: [5]

```
' Create a new component
component = New Component()
component.TheName = "Widget"
component.TheValue = 22
' Save the new Component to the database
component.%Save()
```

[5]

En este punto, una nueva instancia de COMPONENT está almacenada dentro de la base de datos con un identificador de objeto asignado por el sistema. Posteriormente se puede recuperar el objeto utilizando su identificador de objeto, por ejemplo, de la siguiente manera: [5]

```
' Open an instance and double its value:
component = OpenId Component(id)

component.TheValue = component.TheValue * 2
component.%Save()
```

[5]

Se puede realizar exactamente las mismas operaciones utilizando Java, C++, ActiveX o .NET. El compilador de clases puede generar, y sincronizar, cualquier código adicional necesario para acceder a los objetos de la base de datos externamente. Por ejemplo, si estamos utilizando Caché con Java, podemos especificar que el compilador de clases genere automáticamente y mantenga *clases proxy de Java* que permiten acceso remoto a los objetos y clases persistente de las bases de datos OO. En un programa de Java se puede hacer, por ejemplo, lo siguiente: [5]

```
// Get an instance of Component from the database
component = (MyApp.Component)MyApp.Component._open(database, new Id(id));
```

```
// Inspect some properties of this object
System.out.println("Name: " + component.getTheName());
System.out.println("Value: " + component.getTheValue());
```

[5]

Extensión del SQL para manejar orientación a objetos

Para hacer más fácil la vida a las personas que están acostumbradas a utilizar el SQL como lenguaje de consulta, Caché incluye algunas extensiones para el SQL que permiten manejar aspectos de la orientación a objetos. [5]

Una de las más interesantes de dichas extensiones es la habilidad para seguir referencias a objetos utilizando el operador de referencias ("→"). Por ejemplo, supongamos que tenemos una clase VENDEDOR con referencias a otras dos clases: CONTACTO y REGION. Podemos referirnos a las propiedades de objetos relacionados utilizando el operador de referencia mencionado: [5]

```
SELECT ID,Name,ContactInfo->Name
FROM Vendor
WHERE Vendor->Region->Name = 'Antarctica'
```

[5]

Hasta aquí la introducción. La empresa que desarrolla esta herramienta es InterSystems.

RESUMEN GENERAL Y CONCLUSIONES

La metodología y programación orientada a objetos (*OOP por sus iniciales en inglés - Object Oriented Programming*) ha alterado radicalmente la forma en que programamos. Prácticamente todas las clases iniciales de las Ciencias de la Computación se enseña ahora en un lenguaje de programación orientado a objetos como Java o C++. Las ideas de orientación a objetos están influenciando no sólo cómo programamos, sino cómo pensamos acerca de otros conceptos tales como los de las **bases de datos**. [2]

Pros y Contras de los OODBMS

Pros

Los OODBMS proveen muchas ventajas. Estas ventajas son importantes porque resuelven muchos de los problemas que los sistemas tradicionales no pueden. Primero, la cantidad de información que puede modelarse con un OODBMS se incrementa, y también es más fácil modelar esta información. Aún más, los OODBMS brindan objetos complejos que permiten fácil integración de multimedia, CAD, y otras bases de datos especializadas. Los OODBMS también son capaces de tener mayores capacidades de modelado por medio de la extensibilidad. Con un OODBMS, uno sería capaz de agregar más capacidades de modelado, permitiendo de este modo modelar sistemas aún más complejos. Esta extensibilidad brinda una solución para incorporar bases de datos existentes y futuras en un solo entorno. [2]

Además de ventajas de modelado, OODBMS un también tienen ventajas de sistema. En un OODBMS, el manejo de versiones está disponible para ayudar a modelar cambios diversos a los sistemas. Con el manejo de versiones, uno sería capaz de volver a conjuntos de datos previos, y comparar los conjuntos actuales con los anteriores. [2]

La reutilización de clases juega un rol vital en el desarrollo y mantenimiento más rápido de aplicaciones. Las clases genéricas son potentes, pero más importante es que ellas pueden ser usadas nuevamente. Ya que las clases pueden reutilizarse, no se necesita diseñar material redundante. Esto lleva a la más rápida producción de aplicaciones y más fácil mantenimiento de dichas aplicaciones y bases de datos. [2]

Está su flexibilidad, y soporte para el manejo de tipos de datos complejos. Por ejemplo, en una base de datos convencional, si una empresa adquiere varios clientes gracias a referencias de otros clientes, y se desea registrar esta información, pero la base de datos existente, que mantiene la información de clientes y sus compras, no tiene un campo para registrar quién proporcionó la referencia, de qué manera se realizó dicho contacto, o si debe compensarse con algún descuento, crédito u otro beneficio, sería necesario reestructurar la base de datos para añadir este tipo de modificaciones. Por el contrario, en una BDOO, el usuario puede añadir una "subclase" de la clase de clientes para manejar las modificaciones que representan los clientes por referencia. [3]

La subclase heredará toda la estructura de la definición original, además se especializará en especificar los nuevos campos que se requieren, así como los métodos para manipular solamente estos campos. Naturalmente se reservan los espacios para almacenar la información adicional de los nuevos campos. Esto presenta la ventaja adicional de que una BDOO puede ajustarse para usar solo el espacio que sea necesario para los campos existentes, liberando espacio desperdiciado en registros con campos que se usan poco. [3]

Las bases de datos tradicionales almacenan sólo datos, mientras que las bases de datos orientadas a objetos almacenan objetos, con una estructura arbitraria y un comportamiento. Una simple metáfora (Esther Dyson) ayuda a ilustrar la diferencia entre ambos modelos. Consideremos el problema de almacenar un coche en un garaje al final del día. En un sistema de objetos el coche es un objeto, el garaje es otro objeto, y hay una operación simple que es *almacenar_coche_en_garaje*. En un sistema relacional, todos los datos (objetos) deben ser traducidos a tablas, de esta forma el coche debe ser desarmado, y todos los pistones almacenados en una tabla, todas las ruedas en otra, etc. Por la mañana, antes de irse a trabajar hay que componer de nuevo el coche

para poder conducir (problema: al componer piezas puede salir una moto en vez de un coche). [3]

Contras

Mientras hay muchas ventajas para los OODBMS, también hay muchas desventajas. Los sistemas ER tradicionales han estado en el mercado por un largo tiempo y un cambio se apartaría de las ideas establecidas. Requeriría que la gente piense diferente, y en algunos casos los usuarios relacionales no tendrían la base de OO necesarias para trabajar con OODBMS. La educación de la gente en las bases de OO es un proceso muy minucioso. Requeriría gran cantidad de tiempo, dinero y otros recursos. También, y con motivo del cambio, se requeriría más tiempo para mover los datos a los nuevos OODBMS. [2]

Otra desventaja es que habrá necesidad de haya una forma de que los sistemas tradicionales y los OODBMS se comuniquen y trabajen juntos. Los sistemas tradicionales y los OODBMS deben entenderse cada uno y las relaciones que representan. Al momento no hay tal sistema de comunicación y entendimiento. [2]

Además, no existe un lenguaje de consulta ad hoc como SQL. Mientras es más fácil realizar consultas complejas con OODBMSs, no existe un lenguaje de consulta. Aún más, no hay actualmente normas para el diseño e implementación y ninguna parece probable en un futuro cercano. Los OODBMS podrían resolver problemas de los sistemas tradicionales, pero se necesita acordar una norma. [2]

Desarrollos Futuros

Los desarrollos futuros para las OODB deben incluir un lenguaje de consulta ad hoc para el usuario promedio. Este lenguaje debería proveer a las OODB lo que el SQL provee a las bases de datos tradicionales también, debe acordarse una norma para el diseño, la notación y la implementación. Los desarrollos futuros para las OODB podrían incluir un método más fácil de acceder desde Internet y la integración de ideas tales como XML o algo similar. Una iniciativa en este sentido es W3QL (por sus siglas en inglés - World Wide Web Query Language). Esta iniciativa permitiría que uno consulte la web como si fuera una base de datos. Por las enormes cantidades de información, la aproximación orientada a objetos podría resultar útil. [2]

Conclusión final

No hay duda que las ideas orientadas a objetos llegaron para quedarse y han influenciado los modelos relacionales tradicionales. La unión de estos dos pensamientos, programación orientada a objetos y sistemas de administración de bases de datos, brindan las bases para las bases de datos orientadas a objetos. El ser capaces de representar datos y relaciones, manejo de versiones, simplificación del acceso a datos son algunas de las características principales de las OODB. Los OODBMS deben incluir las trece características principales delineadas en el "Manifiesto de Sistemas Orientados a Objetos". Mientras hay muchos pros y contras para los OODBMS, están brindando formas para solucionar problemas y mecanismos por los cuales soluciones tradicionales pueden incorporarse en soluciones futuras. El desarrollo futuro es un campo abierto ya que todavía debe establecerse una norma para las OODB. Una vez que una norma se haya establecido para los OODBMS, podría convertirse en una norma de bases de datos. Como con todo cambio, no vendrá rápidamente, pero con la introducción de Internet y la enorme cantidad de datos que se examinan cuidadosamente, necesitarán alistarse soluciones como las OODB para manejarlos de manera satisfactoria. [2]

Otros tipos de Bases de Datos

Base de datos deductivas

Un sistema de base de datos deductivas, es un sistema de base de datos pero con la diferencia de que permite hacer deducciones a través de inferencias. Se basa principalmente en reglas y hechos que son almacenados en la base de datos. También las bases de datos deductivas son llamadas base de datos lógicas, a raíz de que se basan en lógica matemática.

Los sistemas han comenzado a realizarse hace algunos años, inspirándose inicialmente en las técnicas desarrolladas en Inteligencia Artificial en el marco de los sistemas "Pregunta. Respuesta", adaptándolas a las limitaciones específicas de las Bases de Datos.

Un SGBD deductivo es un Sistema que permite derivar nuevas informaciones a partir de las introducidas explícitamente en la Base por el usuario. Este maneja la perspectiva según la teoría de las demostraciones de una base de datos, y en particular es capaz de deducir hechos a partir de la base de datos extensional, es decir, las relaciones base, aplicando a esos hechos axiomas deductivos o reglas de inferencias especificados. Esta función deductiva se realiza mediante la adecuada explotación de ciertos conocimientos generales relativos a las informaciones de la Base.

Definición: Un sistema de bases de datos que tenga la capacidad de definir reglas con las cuales deducir o inferir información adicional a partir de los hechos almacenados en las bases de datos se llama Sistema de Bases de Datos Deductivas. Puesto que parte de los fundamentos teóricos de algunos sistemas de ésta especie es la lógica matemática, a menudo se les denomina Bases de Datos Lógicas. Una base de datos deductiva es, en esencia, un programa lógico; mapeo de relaciones base hacia hechos, y reglas que son usadas para definir nuevas relaciones en términos de las relaciones base y el procesamiento de consultas.

Los sistemas de Bases de Datos Deductivas intentan modificar el hecho de que los datos requeridos residan en la memoria principal (por lo que la gestión de almacenamiento secundario no viene al caso) de modo que un SGBD se amplíe para manejar datos que residen en almacenamiento secundario.

En un sistema de Bases de Datos Deductivas por lo regular se usa un lenguaje declarativo para especificar reglas. Con lenguaje declarativo se quiere decir un lenguaje que define lo que un programa desea lograr, en vez de especificar los detalles de cómo 1ograrlo. Una máquina de inferencia (o mecanismo de deducción) dentro del sistema puede deducir hechos nuevos a partir de la base de datos interpretando dichas reglas. El modelo empleado en las Bases de Datos Deductivas está íntimamente relacionado con el modelo de datos relacional, y sobre todo con el formalismo del cálculo relacional. También esta relacionado con el campo de la programación lógica y el lenguaje Prolog. Los trabajos

sobre Bases de Datos Deductivas basados en lógica han utilizado Prolog como punto de partida. Con un subconjunto de Prolog llamado Datalog se definen reglas declarativamente junto con un conjunto de relaciones existentes que se tratan como literales en el lenguaje. Aunque la estructura gramatical se parece a la de Prolog, su semántica operativa (esto es, la forma como debe ejecutarse un programa en Datalog) queda abierta.

Una Base de Datos Deductiva utiliza dos tipos de especificaciones: hechos y reglas. Los hechos se especifican de manera similar a como se especifican las relaciones, excepto que no es necesario incluir los nombres de los atributos. Recordemos que una tupla en una relación describe algún hecho del mundo real cuyo significado queda determinado en parte por los nombres de los atributos. En una Base de Datos Deductiva, el significado del valor del atributo en una tupla queda determinado exclusivamente por su posición dentro de la tupla.

Las reglas se parecen un poco a las vistas relacionales. Especifican relaciones virtuales que no están almacenadas realmente, pero que se pueden formar a partir de los hechos aplicando mecanismos de inferencia basados en las especificaciones de las reglas. La principal diferencia entre las reglas y las vistas es que en las primeras puede haber recursión y por tanto pueden producir vistas que no es posible definir en términos de las vistas relacionales estándar.

Las BDD buscan derivar nuevos conocimientos a partir de datos existentes proporcionando interrelaciones del mundo real en forma de reglas. Utilizan mecanismos internos para la evaluación y la optimización.

Características

Una Base de Datos Deductiva debe contar al menos con las siguientes características:

- Tener la capacidad de expresar consultas por medio de reglas lógicas.
- Permitir consultas recursivas y algoritmo s eficientes para su evaluación.
- Contar con negaciones estratificadas.
- Soportar objetos y conjuntos complejos.
- Contar con métodos de optimización que garanticen la traducción de especificaciones dentro de planes eficientes de acceso.
- Como característica fundamental de una Base de Datos Deductiva es la posibilidad de inferir información a partir de los datos almacenados, es imperativo modelar la base de datos como un conjunto de fórmulas lógicas, las cuales permiten inferir otras fórmulas nuevas.

Reglas de Deducción

Las relaciones de una Base de Datos Relacional se define por "intención" y por "extensión". Para una Base particular, la intención de las relaciones que la constituyen se define por un conjunto de leyes generales, mientras que cada estado de la Base proporciona una extensión (conjunto de tuplas) para cada una de las relaciones. Las tuplas constituyen, de hecho, informaciones elementales.

En un SGBD convencional, todas las leyes generales se explotan para mantener la coherencia de las informaciones elementales; a estas leyes se las denomina entonces restricciones de integridad. Por el contrario, en un Sistema deductivo, algunos (o todas) de estas leyes se utilizan como reglas de deducción para deducir nuevas informaciones elementales a partir de las introducidas explícitamente en la Base.

Hardware

Físicamente, las bases de datos deductivas casi siempre se almacenan en medios de acceso directo, por lo regular discos magnéticos de cabeza móvil, aunque en algunos sistemas pudieran utilizarse otros medios (tambores, discos ópticos) en vez de discos o además de discos. Los tiempos de acceso a disco son mucho más largos que los de acceso a la memoria principal: 400 milisegundos o más para un disco flexible, y 30 mili segundos o menos para un disco "rápido" grande. El acceso a la memoria principal será con toda probabilidad cuatro o cinco órdenes de magnitud más rápido que el acceso a disco en un sistema dado. Por 10 tanto, un objetivo prioritario de desempeño en sistemas de Bases de Datos Deductivas es reducir al mínimo el número de accesos a disco (E/S a disco). Cualquier organización de los datos en el disco se denomina estructura de almacenamiento, la cual debe ser elegido el proceso de diseño, a esto se le conoce como diseño físico de Bases de Datos Deductivas.

Algunas de las estructuras de almacenamiento utilizadas con mayor frecuencia en los sistemas actuales son la indexación, hash, cadenas de apuntadores y técnicas de compresión.

Se han hecho varias sugerencias alternativas para crear hardware especial enfocado a las funciones de gestión de datos. Estas alternativas cuyo nombre genérico es máquinas (o computadores) de Bases de Datos, incluyen procesado res dorsales, dispositivos inteligentes, sistemas multiprocesadores, sistema de memoria asociativa y procesadores de propósito general.

Arquitectura de una SABD Deductivo

SABD Deductivo posee dos componentes fundamentales: un módulo deductivo y un SABD relacional.

Ejemplo

La interpretación por la teoría de demostraciones ofrece un enfoque por procedimientos o computacional para calcular una respuesta a la consulta Datalog. Al proceso de demostrar si un determinado hecho (teorema) se cumple se le conoce también corno demostración de teoremas.

Sistema LDL

El proyecto Logic Data Languaje (Lenguaje Lógico de Dato: LDL) de Microelectronics and Computer

Corporation (MCC) se inició en 1984 con dos objetivos primarios:

- Crear un sistema que extendiera el modelo

relacional y a la vez aprovechara algunas de las características positivas de un SGBDR (Sistema de Gestión de Base de Datos Relacionales).

- Mejorar la funcionalidad de un SGBD de un modo que opera como un SGBD deductivo y además permitiera la creación de aplicaciones de propósito general.

Ahora el sistema resultante es un SGBD deductivo que se encuentra en el mercado. Aplicaciones de LDL :

Aplicaciones LDL

El sistema LDL se ha utilizado en los siguientes dominios de aplicación:

- **Modelado de empresas**: este dominio implica modelar la estructura, los procesos y las restricciones dentro de una empresa. Los datos relacionados con ella pueden resultar en modelo ER extendido que contiene cientos de entidades y vínculos y miles de atributos. Es posible desarrollar varias aplicaciones útiles para los diseñadores de nuevas aplicaciones (así corno para los gerentes) a partir de esta metabase de datos, que contiene información tipo diccionario a cerca de toda la empresa.

- **Prueba de hipótesis o dragado de datos**: este dominio implica formular una hipótesis, traducirla a un conjunto de reglas LDL y una consulta, y luego ejecutar la consulta contra los datos para probar la hipótesis. El proceso se repite reformulando las reglas y la consulta. Esto se ha aplicado al análisis de datos de genoma en el campo de la microbiología. El dragado de datos consiste en identificar las secuencias de DNA a partir de autorradiografias digitalizadas de bajo nivel obtenidas de experimentos con bacterias E. coli.

- **Reutilización de software**: el grueso del software para una aplicación se desarrolla en código estándar por procedimientos, y una pequeña fracción se basa en reglas y se codifica en LDL. Las reglas dan origen a una base de conocimientos que contienen los siguientes elementos:

    * Una definición de cada módulo C empleado en el programa.

    * Un conjunto de reglas que define las formas en que los módulos pueden exportar importar funciones, restricciones, etc.

La base de conocimientos puede servir para tornar decisiones referentes a la reutilización de subconjuntos del software. Los módulos pueden recombinarse para satisfacer tareas específicas, en tanto se satisfagan las reglas pertinentes. Se está experimentando con esto en el software bancario.

Conclusión

Se ha presentado una introducción a una rama relativamente nueva de la gestión de Bases de Datos: los Sistemas de Bases de Datos Deductivas. Este campo acusa la influencia de los lenguajes de programación de lógica, sobre todo de Prolog. Un subconjunto de Prolog llamado Datalog que contiene cláusulas de Horn libres de funciones, es el que se usa primordialmente como fundamento de los trabajos con Bases de Datos Deductivas en la actualidad.

El área de las Bases de Datos Deductivas todavía está en una etapa experimental. Su adopción por parte de la industria impulsará su desarrollo.

Otras bases de Datos

Bases de datos jerárquicas

Las bases de datos jerárquicas son bases de datos que, almacenan información en una estructura jerárquica; los datos se organizan en una forma similar a un árbol invertido. Estas particularmente útiles en el caso de aplicaciones que manejan un gran volumen de información y datos muy compartidos permitiendo crear estructuras estables y de gran rendimiento.

Una de las principales limitaciones de este modelo es su incapacidad de representar eficientemente la redundancia de datos.

La base de dato de red

La base de datos de red es un modelo apenas distinto del jerárquico; su diferencia fundamental es la modificación del concepto de nodo: se permite que un mismo nodo tenga varios padres (posibilidad no permitida en el modelo jerárquico). Fue una gran mejora con respecto al modelo jerárquico, ya que ofrece una solución eficiente al problema de redundancia de datos; pero, aun así, la dificultad que significa administrar la información en una base de datos de red ha significado que sea un modelo utilizado en su mayoría por programadores más que por usuarios finales.

Bibliografía:

[1]

Fundamentos de Bases de Datos. Abraham Silberschatz, Henry F. Korth, S. Sudarshan.

[2]

www.acm.org/crossroads/espanol/xrds7-3/objects.html

[3]

Ver en Internet la dirección exacta.

[4]

Bases de datos orientadas a objetos. Diseño de sistemas de bases de datos. Marche Marqués. 12 de abril de 2002.-

[5]

Ayuda de Caché PCKit. Desarrollado por InterSystems. Web: http://www.intersystems.com