1. ESTADO DEL ARTE DE LAS BASES DE DATOS ORIENTADAS A OBJETOS EMBEBIDAS

Las Bases de Datos constituyen una parte fundamental para gestionar los datos en diferentes grados de complejidad con el fin de cumplir en su totalidad con el modelamiento que se realiza de un contexto determinado.

Surgen entonces varias tecnologías para llevar a cabo dicho modelamiento que facilita un mejor manejo y mayor conocimiento de la forma como se accede a los datos y la manera como se puede mantener las aplicaciones por un determinado tiempo aumentando su calidad.

Antes de tratar las Bases de Datos Orientadas a Objetos Embebidas se hizo una revisión del entorno que las rodean, Bases de Datos Orientadas a Objetos y Bases de Datos Objeto - Relacionales, las cuales servirán como punto de comparación de las bases de datos en estudio.

1.1. BASES DE DATOS ORIENTADAS A OBJETOS (BDOO)

Una Base de Datos Orientada a Objetos (BDOO) es una base de datos donde los elementos de datos son **objetos**. Las entidades se representan como **clases**. "Las BDOO contienen un método sistemático de representación de relación, y la interfaz uniforme de usuario es un sistema de mensajes que puede explorar los objetos y sus interconexiones" **(4)**.

En una BDOO, las entidades de aplicación son las clases, las instancias de entidad son objetos creados desde las clases, y las relaciones se mantienen por medio de inclusión lógica. Un sistema de señales y métodos para procesarlas contiene una interfaz para la base de datos.

1.1.1. Características generales de las Bases de Datos Orientadas a Objetos (BDOO)

- Solución a problemas más complejos, ofreciendo un ambiente de desarrollo de aplicaciones donde se requiere un almacenamiento y utilización ágil de la información.

- Agrupamiento de datos relacionados, es decir, no se realiza un ensamble de los datos, lo cual afectaría notablemente el rendimiento de cualquier aplicación.

- Mejoras en la calidad del Software (SW).

- Disminución de los tiempos de desarrollo y costos de mantenimiento del SW.

- Facilidad para manejar aplicaciones web dinámicas, CAD, CAM, CASE, sistemas multimedia, sistemas de red, Sistemas de Información Geográfica (GIS), entre otras.

1.1.2. Conceptos fundamentales de las Bases de Datos Orientados a Objetos

Las BDOO se basan en los siguientes conceptos:

1.1.2.1. Identidad de objetos

En un sistema de BDOO se provee una identidad única a cada objeto independiente almacenado en la base de datos. Esta identidad única suele implementarse con un identificador de objeto único, generado por el sistema, u OID. El valor de un OID (***Identificador de objeto)*** no es visible para el usuario externo, sino que el sistema lo utiliza a nivel interno para identificar cada objeto de manera única y para crear y manejar las referencias entre objetos.

La principal propiedad que debe tener un OID es la de ser inmutable; es decir, el valor del OID para un objeto en particular nunca debe cambiar. Esto

preserva la identidad del objeto del mundo real que se está presentando. También es preferible que cada OID se utilice sólo una vez; esto es aunque un objeto se elimine de la base de datos, su OID no se deberá asignar a otro objeto. Estas dos propiedades implican que el OID no debe depender del valor de ningún atributo del objeto, pues estos valores pueden cambiar.

1.1.2.2. Constructores de tipos

En las BDOO, los valores (o estados) de los objetos complejos se pueden construir a partir de otros objetos mediante ciertos constructores de tipos. Una forma de representar tales objetos es considerar a cada objeto como tripleta (i, c, v), donde i es un identificador de objeto único (el OID), c es un constructor (esto es, una indicación de cómo se construye el valor del objeto) y v es el valor (o estado) del objeto. Puede haber varios constructores según el modelo de datos y el sistema OO. Los tres constructores básicos son:

- Constructores de átomos.
- Constructores de tuplas.
- Constructores de conjuntos.

Otros constructores de uso más común son los de listas y de arreglos. También existe un dominio que contiene todos los valores atómicos básicos que están disponibles directamente en el sistema. Por lo regular estos incluyen los enteros, los números reales, las cadenas de caracteres, los tipos booleanos, las fechas y cualesquiera otros tipos de datos que el sistema maneje directamente.

1.1.2.3. Encapsulamiento

Tanto la estructura de los objetos como las operaciones que se pueden aplicar a ellos se incluyen en las definiciones de clases de los objetos.

1.1.2.4. Compatibilidad con los lenguajes de programación

Existen varios lenguajes de programación con los que se pueden integrar los conceptos orientados a objetos:

- Se puede extender un lenguaje para el manejo de datos como el SQL añadiendo tipos complejos y la programación orientada a objetos. Los sistemas proporcionan extensiones orientadas a objetos a los sistemas relacionales llamados *sistemas relacionales orientados a objetos*.

- 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*. Estos lenguajes 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 datos como SQL. Se denominan persistentes porque los datos siguen existiendo una vez que el programa que los creó ha concluido.

A la hora de decidir que opción utilizar se debe tener en cuenta que los *lenguajes Persistentes* suelen ser potentes y resulta relativamente **sencillo** cometer errores de programación que dañen las bases de datos.

1.1.2.5. Jerarquías de tipos y herencias

Los esquemas de BDOO necesitan un gran número de clases. Sin embargo, varias clases son parecidas entre sí. Para permitir la representación directa de parecidos entre las clases, hay que ubicarlas en una jerarquía de especializaciones. El concepto de jerarquía de clases es parecido al de especialización del modelo E-R. Las especializaciones de las clases son denominadas subclases; lo cual especifica atributos y métodos adicionales para una clase existente. Los objetos creados por medio de una subclase heredan todos los atributos y métodos de la clase padre. Algunas de estas características heredadas pueden ellas mismas haber sido heredadas de clases más altas en la jerarquía.

1.1.2.6. Manejo de Objetos Complejos

Los objetos se consideran complejos porque requieren un área de almacenamiento sustancial y no forman parte de los tipos de datos estándar que suele ofrecer un Sistema Gestor de Base de Datos (SGBD). Puesto que el tamaño de los objetos es considerable, un SGBD podría obtener una porción del objeto y proporcionarla al programa de aplicación antes de obtener todo el objeto. El SGBD podría también usar técnicas de almacenamiento intermedio y caché para obtener por anticipado porciones del objeto, antes de que el programa de aplicación necesite tener acceso a ellas.

En una BDOO se permite a los usuarios crear nuevos tipos, y como un tipo incluye tanto estructura como operaciones, podemos considerar que un SGBOO tiene un sistema de ***tipos extensibles***. Podemos crear bibliotecas de nuevos tipos definiendo su estructura y operaciones, incluso con tipos complejos.

1.1.2.7. Polimorfismo y sobrecarga de operadores

El polimorfismo se refiere a que un método tenga diferentes implementaciones las cuales se aplican según a qué tipo de objeto se le pasa en la llamada al método. Cuando el programador envía una señal a un objeto, el método de la clase de objeto, posiblemente heredado, procesa la señal. Un método puede tener acceso directamente a atributos de un objeto destino, al incluir cualquier atributo heredado de clases padres, pero debe tener acceso a atributos de otros objetos con señales secundarias. Este concepto permite enlazar el mismo nombre o símbolo de operador a dos o más implementaciones diferentes del operador, dependiendo del tipo de objetos a los que éste se aplique.

1.1.2.8. Creación de versiones

Muchas aplicaciones de bases de datos que usan sistemas OO requieren la existencia de varias versiones del mismo objeto.

Regularmente, se aplican actividades de mantenimiento a un sistema de software conforme sus requerimientos evolucionan. Por lo regular, el mantenimiento implica modificar algunos de los módulos de diseño y de implementación. Si el sistema ya está en operación, y si es preciso modificar uno o más módulos, el diseñador deberá crear una nueva versión de cada uno de ellos para efectuar cambios. Cabe señalar que cuando existen más de dos versiones de un objeto, además del módulo original, se puede actualizar concurrentemente las propias versiones del mismo módulo del software. Esto se llama ***ingeniería concurrente***. Sin embargo, siempre llega el momento en que es preciso combinar (fusionar) estas dos versiones para que la versión híbrida incluya los cambios realizados. Es necesario de que sus cambios sean compatibles.

1.2. BASES DE DATOS OBJETO-RELACIONAL (BDOR)

El término Base de Datos Objeto Relacional (BDOR) se usa para "describir una base de datos que ha evolucionado desde el modelo relacional hacia otra más amplia que incorpora conceptos del paradigma orientado a objetos" **(5)**.

1.2.1. Conceptos fundamentales de las BDOR

Una base de datos objeto relacional es aquella base de datos que combina el modelo relacional junto con los conceptos de la programación orientada a objetos. Una gran ventaja es la flexibilidad y la seguridad que ofrecen las BDOR ya que el usuario puede crear sus propios tipos de datos para que puedan ser utilizados de forma transparente en la implementación en un lenguaje orientado a objetos. Esto permite la creación de métodos para estos tipos de datos así como también la creación de funciones miembro desde el Sistema Gestor de Base de Datos (SGBD) para los tipos de datos definidos por el usuario. Lo anterior quiere decir que los SGBDOR permiten almacenar parte de la aplicación en el servidor de base de datos debido a que almacenan datos complejos de una aplicación dentro de la base de datos objeto relacional (BDOR) sin

necesidad de forzar los tipos de datos tradicionales de una base de datos relacional (BDR) permitiendo trabajar funciones orientadas a objetos sobre la base de datos.

En las BDOR el modelo de datos sigue conservando la estructura de tablas mediante filas y columnas de tal forma que el usuario puede acceder a la base de datos de una forma relacional pero a la vez permitiendo definir y acceder a los datos mediante los diferentes mecanismos orientados a objetos. En consecuencia, la base de datos tiene la capacidad cuando se recuperan los datos, de reconstruir nuevamente los datos simples a objetos, lo que implica una gran desventaja ya que el desempeño y la utilización de recursos tiene un costo más grande con respecto a las bases de objetos embebidas las cuales son de igual forma transaccionales pero los objetos siempre se van a encontrar dentro de la aplicación.

La figura 1 ilustra un esquema de las Base de Datos Orientada a Objetos (BDOO) vs Base de Datos Relacional (BDR) con el fin de observar la forma como se realiza el proceso de manejo de los datos en estos tipos de tecnologías:

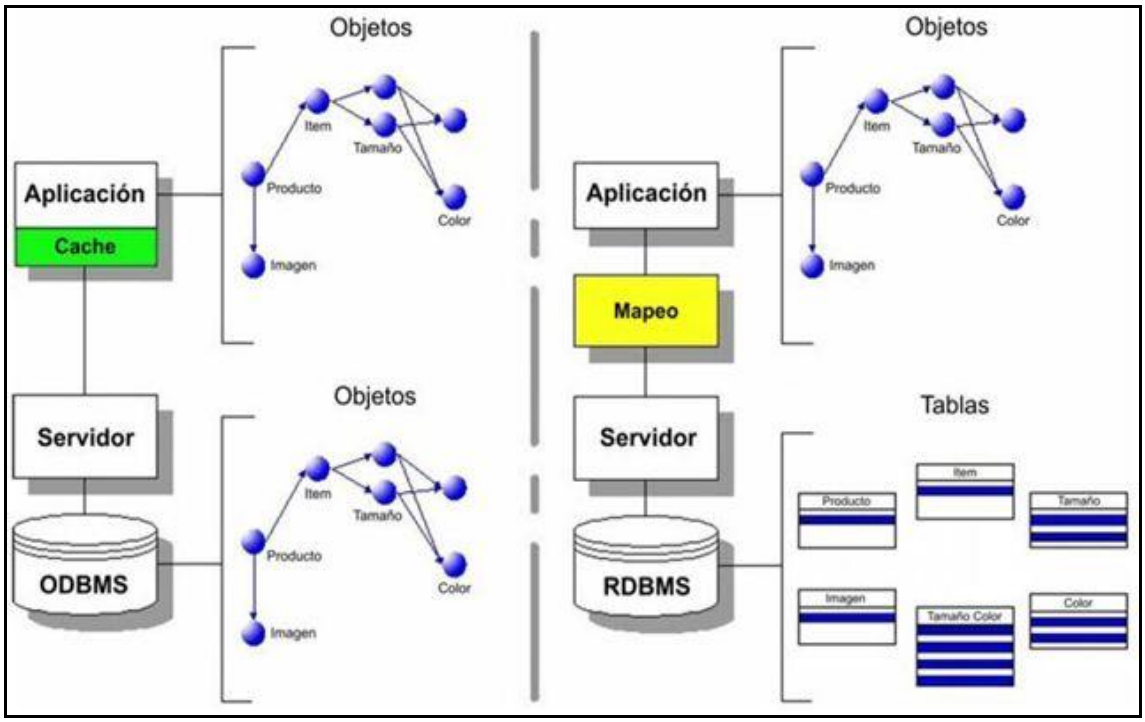

Figura 1. BDOO vs. BDR.

1.2.2. Características generales de las BDOR

- Gran capacidad de compatibilidad, con las Bases de Datos Relaciones (BDR) permitiendo pasar las aplicaciones de una Base de Datos Relacional al **nuevo** modelo sin tener que rescribirlas.

- Gran capacidad para adaptar aplicaciones y Bases de Datos para que se utilicen las funciones orientadas a objetos de una forma progresiva e íntegra.

- Se pueden compartir varias bibliotecas de clases ya existentes, esto es lo que se conoce como reusabilidad.

- Soporte adicional para seguridad y activación de la versión cliente-servidor.

- Se pueden crear funciones que tengan un código en algún lenguaje de programación como por ejemplo: SQL, Java, C, etc.

- Al igual que en las BDOO mencionadas anteriormente, cada objeto tiene un identificador único (OID) el cual no puede ser modificable y tiene el propósito de hacer referencia a otros objetos para representar relaciones de agregación y asociación características del modelo objeto relacional[1].

1.3. BASES DE DATOS EMBEBIDAS

1.3.1. Definiciones

- "Una **Base de datos embebida** es aquella que NO inicia un servicio en nuestra máquina independiente de la aplicación, pudiéndose enlazar directamente a nuestro código fuente o bien utilizarse en forma de librería" **(1)**.

[1] Este tipo de tecnología de base de datos gestiona eficazmente tipos de datos complejos dejando parte de la aplicación en el servidor de bases de datos utilizado.

- “The expression **"Embedded Database**" requires an explanation. A "database" is an application that allows the targeted retrieval of stored data -- a log-file is not a database. By "embedded" I mean a database that does not run in a separate process, but instead is directly linked ("embedded") into the application requiring access to the stored data. This is in contrast to more conventional database management systems (such as Postgres or Oracle), which run as a separate process, and to which the application connects using some form of Inter Process Communication (such as TCP/IP sockets, for instance)” **(2)**.

- “Database software that is included with an application rather than offered as a separate database management system. Tending to be compact and efficient, an embedded database generally includes fewer features than a full-blown DBMS” **(3)**.

- “Database software that is used in cellphones, PDAs and other dedicated devices. It may be software built from the ground up or a slimmed-down version of a larger, mainstream database management system (DBMS). For example, DB2 Everywhere is an embedded version of IBM's flagship DBMS” **(3)**.

1.3.2. *Características generales de las Bases de Datos Embebidas*

- Pequeño tamaño en términos del número de líneas de código en las aplicaciones que se utilizan y en los recursos que consume.

- Utilización solamente pensada para el acceso multiusuario.

- Transparencia para el usuario final.

- Simplicidad para la administración de los datos.

1.3.3. *Bases de Datos Embebidas Comerciales*

A continuación se presentan algunas Bases de Datos Embebidas en el mercado. Estas son:

1.3.3.1. SQLite

SQLite es una Base de Datos Relacional Embebida utilizada como una librería (escrita en C) que implementa un motor de base de datos SQL embebido en diferentes lenguajes de programación como C, Java y Python. Sus principales características son: su completo soporte de tablas e índices en un único archivo por base de datos ofreciendo soporte transaccional, rapidez y completa portabilidad debido a su escaso tamaño, y permite extender su funcionalidad, es decir, definir funciones que operarán sobre los datos devueltos como si se trataran de sentencias SQL.

En el manejo multiusuario, SQLite no permite acceder en modo escritura a la base de datos, ya que el mecanismo de bloqueo que utiliza es muy radical debido a que lo realiza en la base de datos. Así esta librería está especialmente diseñada cuando se quiere una gran rapidez en las consultas pero tomando en cuenta que un único usuario es el que puede realizar modificaciones, es decir, es ideal como base de datos mono usuario y para guardar configuraciones y logs de scripts.

Un de las grandes ventajas que presenta SQLite es que no se hace necesario configurar la red, es decir, tanto el cliente como el servidor corren juntos en el mismo proceso lo cual reduce considerablemente los gastos de la red por el mínimo de llamadas remotas al servidor permitiendo un mejor despliegue de la aplicación que se está ejecutando.

En la Figura 2 se puede observar la forma como SQLite maneja la ejecución de varias aplicaciones **(10)**.

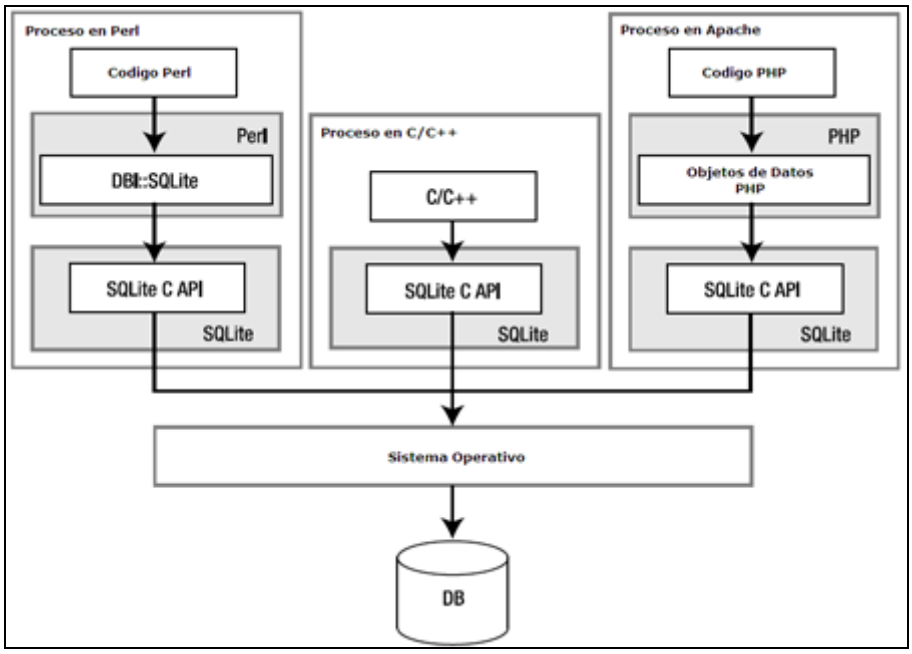

Figura 2. Arquitectura SQLite.

Los procesos mostrados en la figura anterior pertenecen a un script en Perl, otro es un programa en C/C++, y el otro es un proceso de Apache con PHP, todos utilizando SQLite. El script en Perl importa el modulo DBI::SQLite, que a su vez está relacionado con la API en C de SQLite, incluida en la librería de SQLite. La librería en PHP funciona de manera similar al igual que la aplicación en C++. En última instancia, los tres procesos están interactuando con la API en C de SQLite. Por otra lado, aunque cada proceso representa un servidor independiente, aún pueden operar sobre el(los) archivo(s) de una misma base de datos, ya que SQLite utiliza el sistema operativo para llevar a cabo las operaciones de sincronización y bloqueo.

Adicionalmente se puede decir que SQLite presenta muy buen rendimiento trabajando de modo asíncrono en relación con mySQL o Postgresql, resaltando que se trata de motores de base de datos para propósitos distintos.

En la figura 3 se presenta un ejemplo donde se utiliza SQLite para crear una base de datos **(12)**.

Para crear y abrir una base de datos se utiliza la función **$db=sqlite_open("nombre_basedatos")** donde se pasa como parámetro el nombre de la base de datos. Si la base de datos existe, entonces la abrirá, y si no existe la creará y por consiguiente la abrirá. Se puede tener tantas bases de datos como se deseen. La única restricción es la capacidad de almacenamiento que haya en el servidor. Para cerrar la base de datos se utiliza la función **sqlite_close($db)** pasándole como parámetro la variable de identificación de la conexión a la base de datos.

Para ejecutar cualquier sentencia SQL se utiliza la función **sqlite_query()** donde se pasa como parámetros la variable de identificación de la base de datos y la sentencia SQL que se quiere aplicar. Cuando se realiza una consulta SQL el resultado se puede almacenar en una variable de tipo array (en este caso "**$result**") la cual se puede manejar libremente para imprimir resultados en pantalla:

```
$db = sqlite_open("nombre.bd");

sqlite_query($db , "CREATE TABLE tabla_prueba (id INTEGER PRIMARY
KEY, nombre CHAR(255))");

sqlite_query($db, "INSERT INTO tabla_prueba(nombre) VALUES
('Antonio')");
sqlite_query($db, "INSERT INTO tabla_prueba(nombre) VALUES
('Pedro')");
sqlite_query($db, "INSERT INTO tabla_prueba(nombre) VALUES
('Sara')");

$result = sqlite_query($db, "SELECT * FROM tabla_prueba ");

while ($row = sqlite_fetch_array($result)) {
  echo "ID --> ".$row["id"]." Nombre --> ". $row["nombre"]."<br>";
}

sqlite_close($db);
?>

//Resultado del ejemplo anterior
```

```
ID --> 16 Nombre --> Antonio
ID --> 17 Nombre --> Pedro
ID --> 18 Nombre --> Sara
```

Figura 3. Ejemplo de SQLite.

1.3.3.2. Berkeley DB Java Edition

Antes de hablar de la base de datos Berkeley DB Java Edition, se presentan las características generales y la definición de Berkeley DB.

La base de datos Berkeley DB "es una base de datos embebida con APIs para C, C++, Java, Perl, Python, TCL y muchos otros lenguajes. Soporta múltiples datos para una misma clave. Berkeley DB permite miles de hilos de control manipulando bases de datos de hasta 256 terabytes en muchos sistemas, incluidos la mayoría del tipo UNIX y Windows, e incluso sistemas operativos de tiempo real" **(6)**.

Las características generales de este tipo de base de datos son las siguientes:

- Los datos se almacenan en el formato nativo del lenguaje de programación.

- No maneja modo cliente-servidor.

- Caché configurable para modificar el rendimiento.

- Permite crear bloqueos de forma detallada. Esto es especialmente útil para trabajos concurrentes sobre la base de datos de forma que se bloquea una página de registros durante una transacción para evitar que se modifiquen hasta que termine pero permitiendo actuar sobre el resto de páginas.

- Posibilidad de realizar backups y replicación en caliente[2].

Berkeley DB además de las anteriores características presenta una versión llamada "Berkeley DB Java Edition" la cual incorpora menos características a la versión original pero trae la ventaja de estar escrita en un lenguaje multiplataforma como java y de la cual se habla para referirnos a la base de datos embebida. La estructura de Berkeley DB Java Edition **(11)** se puede observar en la Figura 4.

Figura 4. Estructura básica de la base de datos Berkeley DB Java Edition.

En Berkeley DB Java Edition se maneja un motor altamente transaccional el cual permite crear objetos que tendrán la capacidad de persistencia mejorando la escalabilidad, velocidad y flexibilidad que no ofrecen mapeadores objeto-relacionales existentes para los mismos propósitos cuando se habla de realizar operaciones con los datos. Adicionalmente, no requiere de conversión de los objetos a los registros de una base de datos relacional debido al manejo fácil y eficiente que hace por medio de paquetes implementados en su estructura interna.

[2] Capacidad para realizar respaldo de información y transacciones en la base de datos mientras ésta se encuentra en ejecución o está activa.

El concepto de embebido que se presenta en la base de datos Berkeley DB Java Edition consiste en realizar todo el proceso de almacenamiento y consulta de los datos en la propia aplicación con el propósito de minimizar el número de accesos remotos al servidor de la base de datos. Este proceso lo realiza a través de la capa de persistencia en el cual se pueden llevar a cabo muchas de las operaciones que comúnmente se asignan a un motor de base de datos configurado en un servidor remoto.

En el ejemplo de la figura 5 **(11)** se observa una aplicación que está definida en clases java con el propósito de persistir los objetos que realizan operaciones entre sí. Es necesario importar una serie de librerías que permitan realizar las diferentes operaciones con los objetos.

Para definir el dominio de la aplicación Berkeley DB Java Edition utiliza el concepto de clase entidad que permite definir una clase ordinaria en el lenguaje de programación JAVA adicionando en su definición una clave primaria (PrimaryKey) y un número de claves secundarias (SecondaryKeys) que definen las relaciones utilizadas en un modelo entidad-relación como son: Muchos-a-uno, uno-a-muchos, muchos-a-muchos y uno-a-uno. Tanto las claves primarias y las secundarias son accedidas por las aplicaciones por medio de índices primarios y secundarios respectivamente, los cuales van a permitir a los objetos persistir en memoria. Para que dichos objetos sean persistentes, Berkeley DB Java Edition en su capa de persistencia utiliza el objeto raíz llamado **EntityStore**. Este objeto permite manejar cualquier número de objetos de las clases entidad por medio de sus índices primarios y secundarios.

```
// Definición de una clase entidad.
class Person {
String ssn; @PrimaryKey
String name;
Address address;

@SecondaryKey(relate=MANY_TO_ONE, relatedEntity=Person.class)
String parentSsn;
@SecondaryKey(relate=ONE_TO_MANY)
Set<String> emailAddresses = new HashSet<String>();
```

```
@SecondaryKey(relate=MANY_TO_MANY, relatedEntity=Employer.class,
onRelatedEntityDelete=NULLIFY)
Set<Long> employerIds = new HashSet<Long>();

Person(String name, String ssn, String parentSsn) {

this.name = name;
this.ssn = ssn;
this.parentSsn = parentSsn;
}

private Person() {} // Para enlazar con otras clases entidad
}
// Otra clase entidad.
//
@Entity
class Employer {
@PrimaryKey(sequence="ID")
long id;
@SecondaryKey(relate=ONE_TO_ONE)
String name;
Address address;
Employer(String name) {
this.name = name;
}
private Employer() {} // Para enlazar con otras clases entidad
}
//Una clase persistente en otras clases.
//
@Persistent
class Address {
String street;
String city;
String state;
int zipCode;
private Address() {} // Para enlazar con otras clases entidad
}

// Definición de la clase de acceso para índices primaros y
secundarios.

// Índices de Persona
class PersonAccessor {
PrimaryIndex<String,Person> personBySsn;
SecondaryIndex<String,String,Person> personByParentSsn;
SecondaryIndex<String,String,Person> personByEmailAddresses;
SecondaryIndex<Long,String,Person> personByEmployerIds;

// Indices de Empleado
PrimaryIndex<Long,Employer> employerById;
SecondaryIndex<String,Long,Employer> employerByName;
```

```
// Abrir todos los índices primarios y secundarios.
public PersonAccessor(EntityStore store)
throws DatabaseException {
personBySsn = store.getPrimaryIndex(
String.class, Person.class);
personByParentSsn = store.getSecondaryIndex(
personBySsn, String.class, "parentSsn");
personByEmailAddresses = store.getSecondaryIndex(
personBySsn, String.class, "emailAddresses");
personByEmployerIds = store.getSecondaryIndex(
personBySsn, Long.class, "employerIds");
employerById = store.getPrimaryIndex(
Long.class, Employer.class);
employerByName = store.getSecondaryIndex(
employerById, String.class, "name");
  }
}

// Abrir ambiente transaccional de Berkeley DB
EnvironmentConfig envConfig = new EnvironmentConfig();
envConfig.setAllowCreate(true);
envConfig.setTransactional(true);
Environment env = new Environment(new File("/my/data"), envConfig);

// Abrir objeto transaccional entity store.
StoreConfig storeConfig = new StoreConfig();
storeConfig.setAllowCreate(true);
storeConfig.setTransactional(true);

// Inicializar el objeto para accede a los datos
EntityStore store = new EntityStore(env, "PersonStore", storeConfig);
PersonAccessor dao = new PersonAccessor(store);

// Adicionar un padre y dos hijos usando el índice primario de
//Persona y especificando el padre a los dos hijos mediante la clave
//primaria del padre en el sub-índice
dao.personBySsn.put(new Person("Bob Smith", "111-11-1111", null));
dao.personBySsn.put(new Person("Mary Smith", "333-33-3333", "111-11-
1111"));
dao.personBySsn.put(new Person("Jack Smith", "222-22-2222", "111-11-
1111"));

// Imprimir el niño de un padre usando un sub-índice y un cursor
EntityCursor<Person> children =
dao.personByParentSsn.subIndex("111-11-1111").entities();
try {
for (Person child : children) {
System.out.println(child.ssn + ' ' + child.name);
}
} finally {
children.close();
}
```

```
// Obtener al padre Bob pasando como parámetro la clave primaria
//usando el índice primario.
Person bob = dao.personBySsn.get("111-11-1111");
assert bob != null;

// Crear dos empleados. Sus claves primarias se autoincrementan.
Employer gizmoInc = new Employer("Gizmo Inc");
Employer gadgetInc = new Employer("Gadget Inc");
dao.employerById.put(gizmoInc);
dao.employerById.put(gadgetInc);

// Asignar dos trabajos y correos a Bob.
bob.employerIds.add(gizmoInc.id);
bob.employerIds.add(gadgetInc.id);
bob.emailAddresses.add("bob@bob.com");
bob.emailAddresses.add("bob@gmail.com");

// Actualiza registro de Bob.
dao.personBySsn.put(bob);

// Consultar a Bob por sus dos correos.
bob = dao.personByEmailAddresses.get("bob@bob.com");
assert bob != null;
bob = dao.personByEmailAddresses.get("bob@gmail.com");
assert bob != null;

//Consultar a Bob por sus empleos.
EntityIndex<String,Person> employees;
employees = dao.personByEmployerIds.subIndex(gizmoInc.id);
assert employees.contains("111-11-1111");
employees = dao.personByEmployerIds.subIndex(gadgetInc.id);
assert employees.contains("111-11-1111");

// Borrar a Bob de un trabajo.
dao.employerById.delete(gizmoInc.id);
bob = dao.personBySsn.get("111-11-1111");
assert !bob.employerIds.contains(gizmoInc.id);
store.close();
env.close();
```

Figura 5. Ejemplo de Berkeley DB Java Edition.

Para presentar la definición del objeto **EntityStore** se utiliza el ejemplo anteriormente citado como se observa a continuación en la figura 6:

```
EntityStore store = new EntityStore(...);
PrimaryIndex<String,Person> personBySsn =
store.getPrimaryIndex(String.class, Person.class);
```

Figura 6. Definición del objeto EntityStore

Se puede observar que al definir el objeto **EntityStore** se asigna a la clave primaria (**personBySsn**) el índice primario (**PrimaryIndex <String,Person>**) al cual se accede a través de la invocación del método **getPrimaryIndex()**.

Finalmente, cuando se usa la base datos Berkeley DB Java Edition con el propósito de incluirla en las aplicaciones de carácter embebido se debe tener en cuenta algunos factores que son:

- Cuando se utilizan objetos para representar el dominio de una aplicación, y se considera que es relativamente estable, se podrá usar la capa de persistencia que ofrece Berkeley DB Java Edition.

- Cuando se tiene una complejidad baja debido a la poca cantidad de clases que modelan el dominio de la aplicación, Berkeley DB Java Edition es muy buena alternativa.

1.3.3.3. Metakit (BD):

Es una base datos que se utiliza como librería para conectar bases de datos relacionales y diferentes estructuras de datos con el propósito de mostrar transparencia en la gestión de los datos de una determinada aplicación. Sus interfaces para lenguajes de programación orientados a objetos como C++, Python y TCL son las más utilizadas para aprovechar el carácter de base de datos que pueda proporcionar a las aplicaciones permitiéndoles almacenar sus datos de manera eficiente y con poca complejidad de tiempo de ejecución **(7)**.

En términos del modelo de datos se encuentra en el término medio entre Bases de Datos Relacionales (RDBMS), Bases de Datos Orientadas a Objetos (OOBDMS), archivos planos y estructuras como B-Trees; sin embargo, es muy diferente de cada uno de ellas **(7)**.

Para ser una base de datos objeto-relacional embebida, Metakit utiliza objetos para acceder a los datos con el propósito de permitir la ejecución de una serie de operaciones que se realizan en la estructura relacional que almacena dichos datos. Esta estructura se maneja a través de vistas (Views) que están formadas por filas donde a cada una de ellas se le asigna como referencia un índice (Index) que le indica al objeto en que fila se realizan dichas operaciones **(8).**

Para dar una visión más clara de la base de datos Metakit se muestra un ejemplo en la figura 7:

```
import metakit

//permite crear  la base de datos.
db = metakit.storage('gastos.mk',1)

// Crear o redefinir una vista en la Base de Datos.
vista = db.getas("gastos[fecha:S,concepto:S,ingreso:F,gasto:F,
descripcion:S]")

//Realizar commit a la transacción
db.commit()

//Abrir una vista y agregar información a la base de datos
vista.append(fecha='10/10/2003',concepto='Comida-4',
gasto=11.4,concepto='menú del día')
vista.append(['10/10/2003','Comida-5',0,6.6,'menú del día'])
db.commit()

//Muestra la información de la base de datos por medio de una vista
metakit.dump(vista)

La salida del ejemplo anterior es el siguiente:
```

```
fecha       concepto   ingreso     gasto    descripción
---------- -------- ------- ------------- ----------------
20/10/2003    Comida-1   100.0      100.0   Carne de Res
02/10/2003    Comida-2   10.0       5.0     Carne de Pollo
02/10/2003    Comida-3   10.0       0.0     Postre Manzana
10/10/2003    Comida-4   0.0        11.399  Menú del día
10/10/2003    Comida-5   0.0        6.599   Menú del día
---------- -------- ------- ------------- ----------------
Total: 5 rows
```

Figura 7. Ejemplo de base de datos Metakit.

1.4. COMPARATIVO DE LAS BASES DE DATOS EMBEBIDAS

La tabla 1 presenta un comparativo entre las Bases de Datos Embebidas descritas anteriormente. En esta sección se presenta las principales características de algunas Bases de Datos Embebidas que existen actualmente en el mercado con el propósito de mostrar cuales son comunes en este tipo de bases de datos como también las diferencias en sus estructuras y en la forma como manejan el almacenamiento de los datos al ser utilizadas por una determinada aplicación. Las características que se comparan se definen a continuación:

- **Portabilidad:** Capacidad que tiene una aplicación para ejecutarse en diferentes plataformas.

- **Tipo de Base de Datos:** Es la clasificación que se tiene según la tecnología a la que pertenece.

- **Concurrencia:** Es la capacidad que tiene una aplicación para ejecutar múltiples procesos.

- **Rendimiento:** Capacidad que tiene una aplicación para ejecutar un conjunto de tareas.

- **Operaciones SQL:** Operaciones que se pueden realizar mediante un lenguaje de consultas estructurado (SQL).

Tabla 1. Comparativo entre las BD Embebidas.

	Berkeley DB Java Edition	**SQLite**	**Metakit**
Portabilidad	Multiplataforma	Multiplataforma	Multiplataforma
Tipo de Base de Datos	BDOR	BDR	BDOR
Concurrencia	Si	Monousuario	Multiusuario
Rendimiento	No óptima para aplicaciones de alto volumen de datos	No óptima para aplicaciones de alto nivel	No óptima para aplicaciones de alta complejidad
Operaciones SQL	No aplica	Aplica	No aplica

1.5. COMPARATIVO DE TIPOS DE BASES DE DATOS

La Tabla 2 presenta un comparativo entre los tipos de base de datos que se describieron anteriormente. Las características que se tomaron como punto de comparación se describen a continuación:

- **Manejo eficiente de los recursos:** Es la capacidad que tiene una base de datos para usar los recursos que tiene disponibles.

- **Operaciones SQL:** Operaciones que se pueden realizar en una base de datos mediante un lenguaje de consultas estructurado (SQL).

- **Persistencia de objetos:** Es el mecanismo que tiene una aplicación para guardar un objeto de manera permanente con el fin de recuperarlo y/o utilizarlos nuevamente.

- **Desfase de impedancia:** Es la inconsistencia que se presenta entre el modelo de datos y el modelo de objetos.

- **Manejo de objetos complejos:** Es la capacidad de una base de datos para manejar gran cantidad de relaciones entre los objetos que componen la aplicación.

- **Ambientes distribuidos:** Es un conjunto de estaciones separadas físicamente conectadas entre sí a través de una red y que no comparten memoria y sistema operativo.

- **Manejo de conceptos de la POO:** Aplicar "conceptos" tales como herencia, polimorfismo, encapsulamiento, agregación, composición, entre otros, de la POO.

Tabla 2. Comparativo entre tecnologías de Bases de Datos.

Tecnologías	Bases de datos Embebidas	Bases de Datos Orientadas a Objetos	Bases de datos Objeto-Relacional
Manejo eficiente de los recursos.	Aplica	Aplica	No Aplica
Operaciones SQL	Se utilizan si la base de datos es relacional.	No se utilizan.	Se utilizan para realizar consultas más complejas.
Persistencia de Objetos	Aplica si el lenguaje orientado a objetos maneja persistencia de objetos	La maneja directamente.	La maneja a través de motores de persistencia.
Desfasaje de impedancia	Se presenta siempre y cuando la base de datos es relacional	No se presenta	Si se presenta
Manejo de objetos complejos.	Aplica	Aplica	No Aplica
Se utiliza en ambientes distribuidos	Si	Si	Si
Maneja conceptos de la POO	Si es BDOO.	Si	Si el motor utiliza conceptos orientados a objetos.

2. DB4O: UNA BASE DE DATOS ORIENTADA A OBJETOS EMBEBIDA

En cumplimiento de uno de los objetivos específicos definidos para el proyecto, en esta sección se presenta un estudio de la Base de Datos db4o con el fin de mostrar el funcionamiento y la forma de uso con respecto a los conceptos de Base de Datos Orientado a Objetos y Base de Datos Embebida.

2.1. Definición

db4o es una Base de Datos Orientada a Objetos y Embebida. Es una BDOO porque almacena los objetos directamente sin necesidad de ningún tipo de conversión comúnmente llamado mapeo objeto relacional **(9)**. Y es una BDE porque utiliza una librería la cual la hace nativa al entorno de desarrollo, es decir, la base de datos corre dentro del mismo proceso de la aplicación y todas las transacciones y manejo de los objetos es en el lenguaje de desarrollo **(13)**.

Con db4o se puede manejar objetos persistentes ya que trabaja con el modelo de objetos de la aplicación convirtiéndose en la gran ventaja de la eliminación del problema de impedancia que trae inherente el proceso de mapeo objeto relacional. Igualmente se puede tener un manejo automático del esquema de la base de datos y es gracias a que los objetos se almacenan tal y como son, no hay que cambiar las clases para poder almacenarlas (*ver figura 8*). Esto último lleva a concluir que con db4o ya no existe el modelo de datos sino que se trabaja ya directamente con el modelo de objetos y por consiguiente el término de base de datos puede cambiar por el de base de objetos el cual es el que se seguirá manejando en adelante.

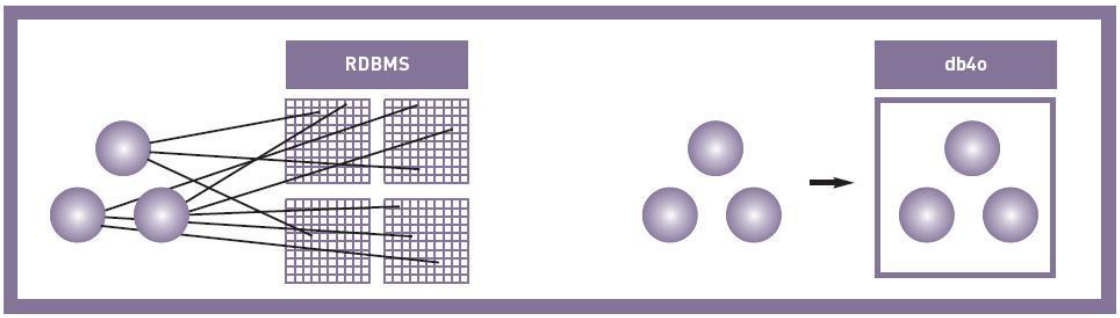

Figura 8. Almacenamiento directo de objetos en db4o.

2.2. Características de db4o

db4o posee características que la ponen en ventaja con respecto a los otros motores de bases de datos, ya que es pequeño, rápido, poderoso y fácil de utilizar. Esto hace que se cumplan muchos requisitos que se esperan a la hora de utilizar un motor de base de datos **(13)**, como:

- **Consumo mínimo de recursos:** Como db4o está diseñado para ser embebido, esto permite un consumo mínimo de recursos ya que el motor se encuentra dentro de la misma aplicación, es decir, si la aplicación está corriendo, la base de objetos también está corriendo dentro del mismo proceso.

- **Alto rendimiento:** db4o ofrece rapidez en los tiempos de acceso de la información dado a su ambiente embebido en una aplicación orientada a objetos ya que permite manipular transparentemente la persistencia de objetos, control de concurrencia, encapsulamiento, polimorfismo, herencia y aprovechar muchos más conceptos de la programación orientada a objetos junto con los conceptos de las bases de datos.

- **Fácil implementación:** db4o ofrece una fácil implementación gracias a que el esquema de la base de datos es el mismo modelo de objetos de la aplicación y que es un motor basado en una librería lo que permite abrir el archivo de la base de objetos mediante una sentencia de código en Java o .NET.

- **Portabilidad:** db4o es un motor multiplataforma porque se trata de una base de objetos embebida que se puede desplegar en varias plataformas móviles como por ejemplo PDA´s así como también, en diversos clientes y servidores bajo plataformas Java y .Net.

- **Confiabilidad:** db4o ofrece confiabilidad ya que las transacciones las termina con los métodos **commit()** y **rollback()**, y en caso de que el sistema se caiga durante una transacción entonces esta será completada cuando se abra de nuevo la aplicación.

Las características mencionadas anteriormente describen claramente a db4o como una base de objetos con una arquitectura de carácter embebido como se puede observar en figura 9.

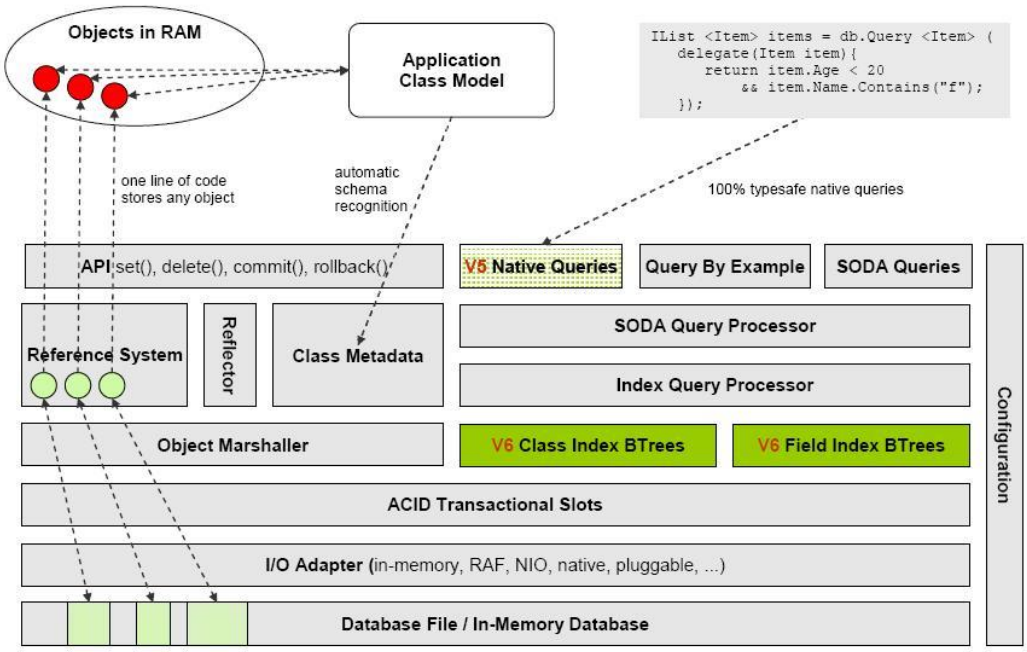

Figura 9. Arquitectura db4o.

Ahora, debido a que una de las características mencionadas anteriormente es el alto rendimiento de db4o, se puede preguntar entonces; ¿cómo se maneja un concepto tan importante de las bases de datos como lo es el lenguaje de definición de datos (DDL) y el lenguaje de manipulación de datos (DML)?. La respuesta es sencilla y lleva a demostrar que db4o es una Base de Datos Orientada a Objetos Embebidas (BDOOE) porque no utiliza un lenguaje de definición de datos (DDL) o un lenguaje de manejo de datos (DML) como SQL, ya que la base de objetos corre dentro de la aplicación, y la definición y manejo de objetos se realizan en el lenguaje de desarrollo (Java o .NET); así por ejemplo, para hacer una actualización (UPDATE) o un eliminar (DELETE) ahora se hace mediante un *get* sobre el objeto y posteriormente llamar un *set* o un *delete*; o bien, para crear una tabla, vista o procedimiento, se define mediante el lenguaje de programación en que se desarrolle la aplicación como se podrá ver en el siguiente caso de estudio.

2.3. Caso de estudio con db4o

A continuación se quiere poner en práctica los conceptos tratados anteriormente y demostrar como db4o se comporta como una BDOOE por medio de un caso de estudio donde se describa como se recupera, inserta, suprime o modifica información almacenada en la base de objetos. Para este caso de estudio se usa como ejemplo el modelo de negocio de un hotel enfocado al mercado corporativo en el que las empresas alojan en un hotel a sus empleados que llegan a una ciudad por asuntos de negocios[3].

En la figura 10 se muestra cómo se asignan las reservas hechas por una empresa en un determinado hotel indicando que empleado trabaja, que huésped se aloja y que facturas de una empresa están asociadas a éste.

[3] Al final del trabajo se anexa diagrama de clases del caso de estudio completo.

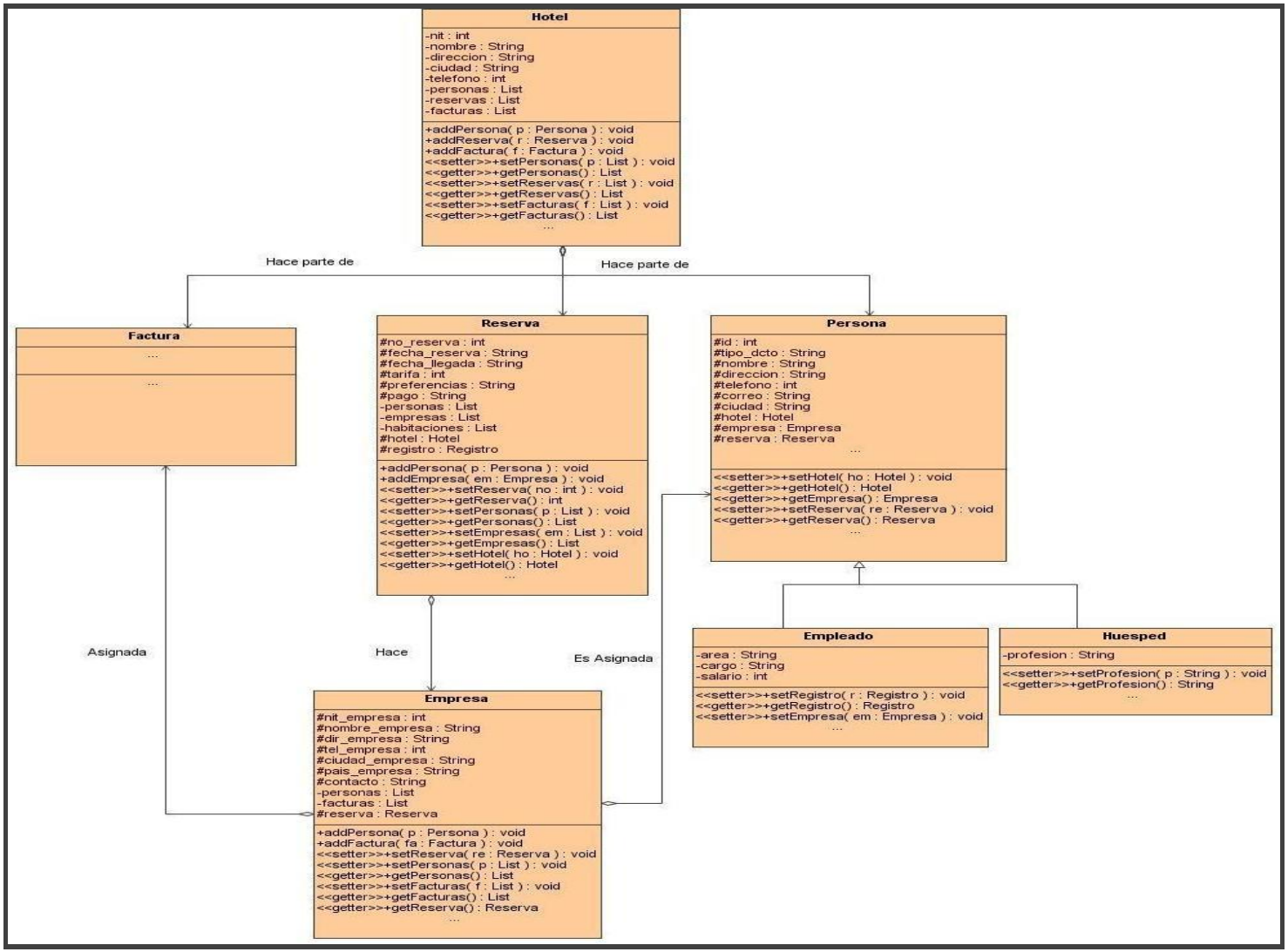

Figura 10. Diagrama de clases para caso de estudio.

2.3.1. Instalación de db4o.

Como se menciona anteriormente en las características, db4o es un motor nativo para Java o .NET y su ejecución es muy sencilla. Primero se descarga la librería db4o o el **jar** que contiene todas las clases de db4o **(15)**. Luego se crea un proyecto en donde se guarda y agrega el archivo como una librería. Cabe recordar que todo esto se puede hacer gracias a un IDE (Integrated Development Environment) en el que se pueda desarrollar la aplicación como Eclipse o JBoss. También es posible descargar una herramienta para el manejo de la base de objetos dentro del IDE como es el ObjectManager el cual sirve para navegar el contenido de la base de objetos, hacer consultas sobre ésta, y conectarse a un servidor con una base de objetos db4o. Durante el desarrollo del caso de estudio se trabajó en el lenguaje Java y con Eclipse como IDE.

2.3.2. Creación de Clases

La creación de las clases en db4o se presenta en forma embebida definiéndolas en el lenguaje nativo de la aplicación (en nuestro caso Java) representando la estructura de cada una de las clases que se tienen en el modelo objetual. Estas clases definen los atributos y los métodos de los objetos que más adelante permitirán manipularlos (**set() y get()**), crear listas de objetos que permitan incluirlos de otras clases cumpliendo con el concepto de agregación; y crear subclases con el fin de añadir atributos sobre ésta y representar la herencia, tal como se puede ver en el modelo de clases. Cabe anotar que no es necesaria la definición de claves únicas sobre las clases ya que en java la base de objetos los almacena automáticamente con un identificador de objeto único conocido como OID. A continuación se muestra un ejemplo con algunas clases del caso de estudio que representan la base de objetos.

→ Clase Hotel:

La razón principal de un hotel son los huéspedes y sus empleados los cuales tienen como fin brindar un buen servicio al huésped. Por esto es necesario crear una clase Persona la cual heredará sus características a dos clases hijas que

son Empleado y Huésped. La operación de un hotel comienza cuando un huésped hace una previa reserva para su estadía y de esta saldrá una factura la cual deberá cancelar el huésped o la empresa a la que pertenece éste. Para la creación de la clase Hotel primero se define cada uno de los atributos y por consiguiente la lista de objetos para las clases Factura, Reserva y Persona ya que éstas son una agregación de Hotel. Luego se hace referencia a cada uno de los objetos. Después de esto, se crean los métodos que permiten agregar un huésped o un empleado en un determinado hotel. Y por último se crean los métodos **set()** y **get()** los cuales permitirán la manipulación de objetos. En la figura 11 se puede ver el ejemplo completo:

```
public class Hotel {
      public class Hotel {
      private int nit;
      private String nombre;
      private String direccion;
      private String ciudad;
      private int telefono;
      private List personas;

//Se hace referencia a cada uno de los objetos.
public Hotel(int nit, String nombre, String direccion, String ciudad,
int telefono){
            this.nit = nit;
            this.nombre = nombre;
            this.direccion = direccion;
            this.ciudad = ciudad;
            this.telefono = telefono;
            this.personas = new ArrayList();
      }

//Se crean métodos para agregar un huésped o un empleado en un
//determinado hotel

      public void addPersona(Persona p) {
            personas.add(p);
      }

//Métodos para obtener y almacenar objetos de un empleado y/o huésped
public void setPersonas(List p){
            personas = p;
      }
      public List getPersonas(){
            return personas;
      }
```

```
//Se crean métodos que permitirán la manipulación de objetos.

      public void setNit(int ni){
            this.nit = ni;
      }
      public int getNit(){
            return this.nit;
      }

      public void setNombre(String n){
            this.nombre = n;
      }
       public String getNombre(){
            return this.nombre;
      }

       public void setDireccion(String d){
            this.direccion = d;
      }
       public String getDireccion(){
            return this.direccion;
      }

      public void setCiudad(String c){
            this.ciudad = c;
      }
      public String getCiudad(){
            return this.ciudad;
      }

      public void setTelefono(int t){
            this.telefono = t;
      }
      public int getTelefono(){
            return this.telefono;
      }
}
```

Figura 11. Creación de la clase Hotel.

→ Clase Persona:

La clase Persona representa una clase padre de la que descienden los empleados y huéspedes pertenecientes a un determinado hotel por lo que se declaran los atributos que van a ser heredables por estas dos clases. También se declara un atributo el cual referencia la clase Hotel con el fin de asignar en que hotel está hospedado un determinado huésped o trabaja un determinado

empleado; se declara otro atributo el cual referencia a la clase Empresa ya que un huésped pertenece a una empresa. Como una reserva tiene asociada el huésped que se va alojar y el empleado quien hace la reserva, se declara un atributo que referencia a la clase Reserva. Por último, debido a que un empleado hace el registro de un huésped, se declara un atributo Registro. Después de declarar los atributos se hace referencia a los objetos que van a ser heredados a las clases hijas. Y al igual que la clase Hotel se termina declarando los métodos **set()** y **get()**, como se puede observar en la figura 12:

```
public class Persona {

            protected int id;
            protected String tipo_dcto;
            protected String nombre;
            protected String direccion;
            protected int telefono;
            protected String correo;
            protected String ciudad;
            protected Hotel hotel;

public Persona(int id, String tipo_dcto, String nombre,  String
direccion, int telefono, String correo, String ciudad){
                  this.id = id;
                  this.tipo_dcto = tipo_dcto;
                  this.nombre = nombre;
                  this.direccion = direccion;
                  this.telefono = telefono;
                  this.correo = correo;
                  this.ciudad = ciudad;
            }


          public void setHotel(Hotel ho){
                  this.hotel = ho;
                }
                public Hotel getHotel(){
                  return this.hotel;
                }

          public void setId(int i){
                  this.id = i;
                }
                public int getId(){
                  return this.id;
                }
```

```
            public void setTipodcto(String td){
              this.tipo_dcto = td;
            }
            public String getTipodcto(){
              return this.tipo_dcto;
            }

            public void setNombre(String n){
              this.nombre = n;
            }
            public String getnombre(){
              return this.nombre;
            }

            public void setDireccion(String d){
              this.direccion = d;
            }
            public String getDireccion(){
              return this.direccion;
            }

            public void setTelefono(int t){
              this.telefono = t;
            }
            public int getTelefono(){
              return this.telefono;
            }

            public void setCorreo(String m){
              this.correo = m;
            }
            public String getCorreo(){
              return this.correo;
            }

            public void setCiudad(String c){
              this.ciudad = c;
            }
            public String getCiudad(){
              return this.ciudad;
            }

            public String toString() {
                return nombre + ":" + empresa;
            }
    }
```

Figura 12. Creación de la clase padre Persona.

→ **Subclase Huésped:**

En la figura 13 se puede observar la subclase Huesped que hereda todas las características de la clase padre Persona con la diferencia que se declara un atributo profesión. A estos atributos se les declara sus métodos que permitan manipular la información del huésped.

```
public class Huesped extends Persona{

	private String profesion;

	public Huesped(int id, String tipo_dcto, String nombre,  String
direccion, int telefono, String correo, String ciudad, String
profesion){
		super(id, tipo_dcto, nombre, direccion, telefono, correo,
ciudad);

		this.profesion = profesion;
	}

	public void setProfesion(String p){
		this.profesion = p;
	}
	public String getProfesion(){
		return this.profesion;
	}

    public String toString() {
        return nombre + ", Telefono: " + telefono;
    }
}
```

Figura 13. Creación de la clase hija Huésped.

→ **Subclase Empleado:**

La subclase Empleado se refiere al personal administrativo y operativo que labora en un determinado hotel, es decir, esta clase describe una especialización de la clase Persona y por ser una clase hija tiene como únicos atributos particulares el área donde trabaja, el cargo y el salario del empleado; así como también, un método **set()** y un método **get()** con el fin de asignar que empleado realiza el registro de un huésped:

```
public class Empleado extends Persona {
      private String area;
      private String cargo;
      private int salario;

      public Empleado(int id, String tipo_dcto, String nombre,
String direccion, int telefono, String correo, String ciudad, String
area, String cargo, int salario){
            super(id, tipo_dcto, nombre, direccion, telefono, correo,
ciudad);
            this.area = area;
            this.cargo = cargo;
            this.salario = salario;
      }

      public void setArea(String a){
            this.area = a;
      }
      public String getArea(){
            return this.area;
      }

      public void setCargo(String ca){
            this.cargo = ca;
      }
    public String getCargo() {
      return this.cargo;
    }

    public void setSalario(final int s){
      this.salario = s;
    }
    public int getSalario(){
      return this.salario;
    }

    public void setRegistro(Registro r){
      this.registro = r;
    }
    public Registro getRegistro(){
      return this.registro;
    }

    public String toString() {
        return nombre + ", " + cargo + ", " + area + ", " + salario;
    }
}
```

Figura 14. Creación de la clase hija Empleado.

2.3.3. Inserción de objetos

Para insertar o almacenar cada uno de los objetos que representan el diagrama de clases, primero se crean los objetos mediante variables ya que en Java es la unidad de almacenamiento. Estas variables son de instancias accesibles por todos los métodos de las clases definidas anteriormente para lo cual se crea uno de los objetos a manera de ejemplo.

Para crear un hotel, una empresa, o una reserva, se instancian los objetos declarados en cada una de las respectivas clases definidas anteriormente en el respectivo orden de definición. Ahora, para crear un empleado y/o un huésped, la instancia se hace hacia la clase Persona pero se define el objeto como un nuevo empleado y/o un nuevo huésped dado a que la clase Persona es una generalización de todos los empleados y huéspedes de un hotel. En la figura 15 se puede ver todo lo mencionado anteriormente:

```
private void crearDatos(){

Hotel ho1 = new Hotel(811045658,"Hotel Plaza Rosa", "Cra 32D # 9-
17","Medellin", 3120005);
Hotel ho2 = new Hotel(801343409,"Hotel Windsor Cali", "Cll 4 N # 67-
7","Cali", 6606390);

Empresa em1 = new Empresa(811321456,"Avianca S.A", "Cra 8 # 7-87
of.304", 2443678,"Bogota","Colombia","Maria Naranjo");
Empresa em2 = new Empresa(810856345,"Exito S.A", "Cll 63 # 43-19",
2400828,"Barranquilla","Colombia","Amparo Gomez");

Reserva(1000, "2008/05/01", "2008/05/04", 154000, "Hab. sencilla con
desayuno", "Directo");
Reserva re2 = new Reserva(1001, "2008/05/04", "2008/05/09", 154000,
"Hab. sencilla, sin desayuno", "CXC");

Persona p1 = new Huesped(1234, "CE", "Jaime Gomez", "Cra 64 #45-8",
2543765, "jaime.gomez@gmail.com", "Medellin", "Abogado");
Persona p2 = new Huesped(1324, "CX", "Maria Giraldo", "Cll 87 # 14 A
45", 2604896, "maria.giraldo@gmail.com", "Miami", "Ing. Procesos");

Persona p7 = new Empleado(3265, "CE", "Juana Maria Velez", "cra 58 #
78 c 54", 2954359, "juana.velez@plazarosa.com", "Cali",
"Alojamiento", "Recepcionista", 700000);
Persona p10 = new Empleado(6926, "CE", "Mauricio Alzate Bolaños",
"Cll 42 # 67-11", 2354900, "mauricio.alzate@plazarosa.com",
```

```
"Medellín", "Comercial", "Cord. Comercial", 900000);

Factura fa1 = new Factura(10, "2008/05/06", 658160);
Factura fa2 = new Factura(11, "2008/05/10", 203940);
```

Figura 15. Creación de objetos.

El segundo paso es asignar los objetos según las referencias creadas en la definición de cada una de las clases mediante sus respectivos métodos que representan el caso de estudio. El carácter embebido que se presenta en esta etapa se aplica cuando se asigna un objeto pasando como parámetro otro objeto. Esto se debe realizar con el propósito de mantener las relaciones de agregación y herencia. A continuación en la figura 16 se presentan varios ejemplos[4]:

```
//Agregar Empleados a un Hotel
ho2.addPersona(p6); p6.setHotel(ho2);
ho1.addPersona(p7); p7.setHotel(ho1);
ho1.addPersona(p8); p8.setHotel(ho1);
ho2.addPersona(p9); p9.setHotel(ho2);
ho1.addPersona(p10); p10.setHotel(ho1);

//Agregar huéspedes a sus empresas
em1.addPersona(p1); p1.setEmpresa(em1);
em1.addPersona(p2); p2.setEmpresa(em1);
em2.addPersona(p3); p3.setEmpresa(em2);

//Agregar Huspedes a Reservas
re1.addPersona(p1); p1.setReserva(re1);
re2.addPersona(p2); p2.setReserva(re2);
```

Figura 16. Asignar objetos.

Por último se abre la base de objetos parar comenzar a almacenar los objetos asignados aplicando el método **set()** sobre la referencia db definida al objeto ObjectContainer de la librería de db4o en el momento de abrir la base de objetos e indicando como parámetro el objeto a almacenar junto con sus datos asociados como se puede observar en la figura 17:

[4] Ver Anexo con la definición de las demás clases del caso de estudio.

```
// Abrir una nueva base de datos
new File(filename).delete();
ObjectContainer db = Db4o.openFile(filename);

// Almacenamos Hoteles y todos los datos asociados
db.set(ho1);
db.set(ho2);

// Close database
db.close();
}
```

Figura 17. Almacenar objetos y sus datos asociados.

2.3.4. Acceder y cerrar una base de objetos

En db4o para crear y abrir una base de objetos es necesario definir un path con la ruta donde se desea almacenar y el nombre de la base de objetos que se quiere llamar. Luego se abre la base de objetos llamando el path declarado anteriormente mediante la interfaz ObjectContainer la cual provee los métodos para almacenar, consultar y borrar los objetos y para efectuar múltiples transacciones de rollback y commit (*véase 2.3.8*) ya que todo trabajo con db4o es transaccional[5].

```
	private String filename;
	filename = "d:/Hotel.yap";
	ObjectContainer db = Db4o.openFile(filename);
```

Figura 18. Crear y abrir una base de objetos.

Si antes de trabajar en una base de objetos se desea borrar todo el contenido con el fin de comenzar de cero, o bien, debido a que se hacen diversas pruebas sobre el modelo, se puede llamar el archivo y aplicar el método **delete()** y luego abrir la base de objetos para comenzar a almacenar de nuevo los objetos mediante el método **set()** de la interfaz ObjectContainer (*véase 2.3.3*) como se puede ver en la figura 19:

[5] La interfaz ObjectContainer puede también representar una base de datos stand-alone o una conexión a un servidor.

```
new File(filename).delete();
ObjectContainer db = Db4o.openFile(filename);
//Acá se comienza de ceros con la Base de objetos
db.close();
```

Figura 19. Borrar todo el contenido de una base de objetos.

Para cerrar una base de objetos, se hace referencia al objeto en el cual se creó a la interfaz ObjectContainer a la hora de abrir la base de datos con la sentencia **db.close()** tal como se muestra en la figura anterior.

2.3.5. Tipos de consultas en db4o

En db4o existen tres métodos diferentes para consultar información de una base de objetos. Inicialmente se tratan las consultas por ejemplo o QBE por sus siglas en inglés (Query By Example) las cuales se basan como su nombre lo dice, en ejemplos; luego se tratan las consultas de Acceso Simple a Base de Datos Objetuales o S.O.D.A por sus siglas en inglés Simple Object Database Access que se basan en recuperar objetos mediante las librería db4o; y por último se explica las consultas en el mismo lenguaje de la aplicación bien sea Java o .Net comúnmente llamadas consultas nativas o NQ´s (Natives Queries)[6].

2.3.5.1. Query By Example (QBE)

Las consultas en QBE a diferencia de las consultas nativas se expresan por ejemplos, es decir, en vez de dar un procedimiento para obtener la respuesta deseada, el usuario da un ejemplo de lo que desea; y así, db4o generaliza este ejemplo para calcular la respuesta a la consulta retornando todos los objetos que concuerden con los campos y sus valores que no sean por defecto. Hacer estas consultas tiene sus desventajas como son **(15)**:

- No se pueden utilizar expresiones de consultas avanzadas como AND, NOT, OR, etc.

[6] Para los ejemplos de los diferentes tipos de consultas se asume que la base de objetos se abre antes de realizar una consulta y se cierra posteriormente. Ver anexo con la definición de las demás clases del caso de estudio.

- Se necesita de un constructor para crear objetos sin inicializar campos.

- No se puede indexar sobre valores como integers (0), strings vacíos (“”) o nulos debido a que ellos no son representados como índices.

Para hacer una consulta QBE en el caso de estudio, es necesario crear un objeto como ejemplo de una clase la cual se quiere consultar. A continuación para consultar un empleado de nombre Raúl Navarro, se crea un objeto empleado como ejemplo y se finaliza imprimiendo el resultado de la consulta como se puede observar en la figura 20:

```
Empleado ejemploEmpleado = new Empleado(0,null,"Raul Navarro", null,
0, null, null, null, null, 0);
        result=db.get(ejemploEmpleado);

        System.out.println(result.size());
        while(result.hasNext()) {
            System.out.println(result.next());
        }
        result.reset();
//Resultado en consola
1
Raul Navarro, Botones, Alojamiento, 600000
```

Figura 20. Ejemplo QBE.

2.3.5.2. Simple Object Database Access (SODA)

SODA es un tipo de consultas más complejas dentro de estructuras profundas a través de la clase Query de la librería db4o ya que provee diversos métodos los cuales permiten descender en la información de las propiedades de los objetos que se quieren consultar; es decir, permiten descender y restringir por nodos que pueden ser una o muchas clases, atributos de una clase, o consultas mismas **(15)**. Las consultas SODA presentan una desventaja la cual consiste en que utilizan cadenas para identificar los campos y esto hace que las consultas estén propensas a errores sin que puedan ser detectados en el momento de la compilación; aunque con la ventaja de que las consultas puedan ser más

optimizadas que las consultas nativas las cuales se tratarán en el siguiente numeral.

En la figura 21 se presenta una consulta que permita seleccionar las facturas cuyo valor total sea mayor a 200.000. Esto se hace primero representando una clase de db4o para crear una consulta y luego se define la clase a la cual se desea aplicar las restricciones. Después mediante el método **descend()** se desciende hasta el atributo "Total" de la clase Factura con el fin de aplicar la restricción que arroje todas las facturas con valor total mayor a 200.000 y luego ejecutar la consulta. Y por último se listan los resultados mostrando cada factura que cumpla con la restricción y la empresa a la que pertenece la factura:

```
Query q = db.query();
q.constrain(Factura.class);

q.descend("total").constrain(new Integer(200000)).greater();
result = q.execute();

System.out.println(result.size());
        while(result.hasNext()) {
            Factura fa = (Factura) result.next();
            System.out.println(fa);
            Empresa em = fa.getEmpresa();
            System.out.println(em);
        }
        result.reset();

//El resultado de la consulta es el siguiente:

2
Factura No: 10 Fecha Factura: 2008/05/06 Total Factura: 658160
Avianca S.A
Factura No: 11 Fecha Factura: 2008/05/10 Total Factura: 203940
Exito S.A
```

Figura 21. Ejemplo SODA.

2.3.5.3. Consultas Nativas (Native Query):

Las consultas nativas son aquellas que son hechas en el lenguaje de programación en la cual se desarrolla la aplicación logrando que éstas sean

seguras en tiempo de compilación **(13)**. A nivel de desempeño este tipo de consultas presenta una desventaja ya que las consultas SODA son la base de las consultas nativas por lo que afectan el desempeño en tiempo de compilación debido a que la consulta pasa por un conversor para que finalmente sea ejecutada en SODA. Para contrarrestar esto, existen herramientas que optimizan las consultas con el fin de mejorar el desempeño.

Las consultas nativas están basadas en cadenas lo cual implica que deben ser analizadas léxicamente y sintácticamente; en otras palabras, debe ser analizado por el parser afectando como se dijo anteriormente el tiempo de compilación.

En la figura 22 se toma el mismo ejemplo de las consultas SODA con el fin de demostrar las diferencias entre éstas y cómo definir la expresión que se quiere consultar. En este caso se quiere obtener todas las facturas cuyo total es mayor a 200.000.

```
       List <Factura> facturas = db.query(new Predicate<Factura>(){
             public boolean match(Factura factura){
                   return factura.getTotal() > 200000;
             }
       });

       for(Factura factura : facturas)
             System.out.println(factura);

//Y el resultado es el siguiente:
Factura No: 10 Fecha Factura: 2008/05/06 Total Factura: 658160
Factura No: 11 Fecha Factura: 2008/05/10 Total Factura: 203940
```

Figura 22. Ejemplo NQ.

2.3.6. Actualizar en db4o

Para actualizar objetos en db4o se recupera primero el objeto que se quiere actualizar de la misma forma como se consultó anteriormente un objeto con QBE con el fin de actualizar el o los atributos deseados mediante los métodos definidos en la creación de la clase del objeto **(16)**. En la figura 23 se va a actualizar el teléfono de un

huésped mediante una consulta QBE que trae el objeto que se quiere actualizar y se almacena mediante el método **setTelefono()** creado en la clase Persona la cual es una generalización de Huésped y Empleado.

```
 Huesped ejemploHuesped = new Huesped(0, null, "Gabriel Jaime Gomez",
null, 0, null, null);
result=db.get(ejemploHuesped);
Huesped h = (Huesped) result.next;
System.out.println("Sin Actualizar: " + h);

h.setTelefono(2650400);
db.set(h);
System.out.println("Actualizado: " + h);

//El resultado en consola es el siguiente:

Sin Actualizar: Gabriel Jaime Gomez, Telefono: 2543765
Actualizado: Gabriel Jaime Gomez, Telefono: 2650400
```

Figura 23. Actualizar en db4o.

2.3.7. Borrar en db4o

En db4o es rápido y fácil borrar un objeto dado que inicialmente se procede de igual forma a la actualización de objetos, es decir, se recupera primero el objeto que se desea eliminar mediante una consulta QBE y por último se borra el objeto mediante el método **delete()** de la interfaz ObjectContainer de la librería db4o **(16)**. En la figura 24 procede a eliminar un huésped de un hotel.

```
 Huesped ejemploHuesped = new Huesped(0, null, "Juan Andres Perez",
null, 0, null, null);
result=db.get(ejemploHuesped);
Huesped h = (Huesped) result.next();
System.out.println("Sin Borrar: " + h);
db.delete(h);
System.out.println("Borrado: " + h);

//El resultado en consola es el siguiente:

Sin Borrar: Juan Andres Perez, Telefono: 2234098
Borrado: Juan Andres Perez, Telefono: 2234098
```

Figura 24. Borrar en db4o.

2.3.8. Concepto de Transacción en la base de datos db4o

Para que una base de datos sea transaccional es necesario que cumpla una serie de propiedades que garanticen que las transacciones sean seguras y que no haya pérdida de información debido a procesos de lectura o escritura a la misma vez en la base de datos, una falla eléctrica, un fallo de disco, un error lógico, o cualquier eventualidad que de paso a una caída del sistema y del motor de base de datos. En otras palabras, para que una base de objetos, especialmente db4o, se considere transaccional se debe identificar los conceptos que se describirán a continuación:

- **Atomicidad:** La atomicidad consiste en que toda transacción se realice completamente o no se realice ninguna parte de la transacción. En otras palabras, la atomicidad asegura que una transacción no quede incompleta.

- **Consistencia:** La consistencia en una transacción es similar a la atomicidad y significa que todas las reglas de la transacción sean ejecutadas como por ejemplo una validación después de terminar una transacción; si la validación no se cumple, toda la transacción se aborta.

- **Aislamiento (Isolation):** El aislamiento garantiza en una transacción, que ésta sea invisible a todos los usuarios finales hasta el momento en que dicha transacción sea ejecutada completamente.

- **Durabilidad:** La durabilidad significa que una transacción que haya sido ejecutada satisfactoriamente, debe permanecer aún si se presentan fallas externas a la base de objetos como de software o hardware.

El mecanismo utilizado por db4o para empezar una transacción inicia cuando se abre la base de objetos en modo stand - alone mediante la sentencia **OpenFile** o cuando se crea un cliente mediante la sentencia **OpenClient**, y termina la transacción cuando el objeto Contenedor (**ObjectContainer**) cierra la base de objetos o el cliente mediante la

sentencia **close()**. Por defecto actualizar cualquier objeto por medio de la sentencia **set()**, no garantiza que la transacción finalice completamente o llegue hasta el fin. Para esto db4o antes de iniciar cualquier transacción se asegura que la base de objetos esté actualizada correctamente.

Cuando el usuario ejecuta una sentencia **commit()**, finaliza todas las actualizaciones durante una transacción explícita, es decir, no se asegura que la transacción llegue a su fin pero sí que todas las actualizaciones lleguen hasta su punto final de tal forma que queden escritas en la base de objetos. De igual forma, se permite hacer proceso de rollback mediante la sentencia **rollback()** de manera que todas las actualizaciones hechas desde el último proceso de commit sean descartadas.

En la figura 25 se toma el ejemplo visto anteriormente donde se muestra como se actualiza un objeto en db4o, para aplicar un procedimiento de rollback y demostrar de esta forma el concepto de transacción en db4o:

```
private void actualizarObjetos(){
        ObjectSet result = null;
        ObjectContainer db = Db4o.openFile(filename);

        Huesped ejemploHuesped = new Huesped(0, null, "Gabriel Jaime
Gomez", null, 0, null, null);
        result=db.get(ejemploHuesped);
        Huesped h = (Huesped) result.next();
        System.out.println("Sin Actualizar: " + h);
        h.setTelefono(2650400);
        db.set(h);
        System.out.println("Actualizado: " + h);

        db.rollback();

        System.out.println("Rollback: " + db.get(h));

        db.close();
    }
El resultado en consola es el siguiente:

Sin Actualizar: Gabriel Jaime Gomez, Telefono: 2543765
Actualizado: Gabriel Jaime Gomez, Telefono: 2650400
Rollback: Gabriel Jaime Gomez, Telefono: 2543765
```

Figura 25. Ejemplo transaccional en db4o.

2.4. Cuadro comparativo de los tipos de consultas en db4o

En la tabla 3 se presenta las ventajas y desventajas de los 3 tipos de consultas que se pueden realizar con db4o considerando el contexto en que se quieren utilizar con el propósito de evitar posibles deficiencias en el desarrollo de las aplicaciones:

Tabla 3. Comparativo de tipos de consultas en db4o.

Tipo de Consulta	Ventajas	Desventajas
Query By Example (QBE)	Es un tipo de consultas básicas recomendada para principiantes.	Funcionalidades limitadas: Carece de consultas con conectores (AND,NOT,OR,etc). Se necesita agregar un constructor para crear objetos sin campos inicializados. No se puede consultar por valores nulos.
Simple Object Database Access (S.O.D.A)	Independencia del lenguaje de programación utilizado. API simple. Minimiza el uso de cadenas para consultas. Introduce conceptos relacionados con grafos de nodos (natural a las bases de objetos) Son más optimizadas con respecto a los otros tipos de consultas.	Consultas basadas en cadenas susceptibles a errores en compilación.
Native Querys	Verificación de errores semánticos y sintácticos en tiempo de compilación. 100% nativas. Permite refactorizar el código.	Consultas complejas significan mayor costo en rendimiento.