Ampliación de Bases de Datos
Año 2014/2015
Grado en Ing. Informática
Fdi - UCM

Examen

Final Junio 2015

▷ 1. Patrones de acceso a datos

[2.5 pt] Supongamos una base de datos que almacena información sobre productos y proveedores mediante dos relaciones:

producto(*Codigo*, *Nombre*, *Descripcion*) **proveedor**(*CIF*, *Nombre*, *TelefonoContacto*, *Poblacion*)

Queremos acceder a esta base de datos mediante una aplicación escrita en Java. Para ello disponemos de las clases `Producto` y `Proveedor` que conforman el modelo de la aplicación:

```
public class Producto {
    private Integer codigo;
    private String nombre;
    private String descripcion;

    public Producto(int codigo,
                    String nombre
                    String descripcion)
    ... // métodos get y set
}
```

```
public class Proveedor {
   private String cif;
   private String nombre;
   private String telefonoContacto;
   private String poblacion;

   public Proveedor(String cif,
                    String nombre,
                    String telefonoContacto,
                    String poblacion)
   ... // métodos get y set
}
```

Partimos de una clase `AbstractMapper<T,K>` que contiene una referencia a un `DataSource` y las declaraciones de algunos métodos abstractos vistos en clase:

```
public class AbstractMapper<T,K> {
    protected DataSource ds;

    // Obtiene el nombre de la tabla
    protected abstract String getTableName();

    // Obtiene los nombres de las columnas de la tabla
    protected abstract String[] getColumnNames();

    // Construye un objeto a partir de un ResultSet
    protected abstract T buildObject(ResultSet rs) throws SQLException;

    ...
}
```

El método de consulta *Query by Example (QBE)* se basa en la creación de un objeto *prototipo* cuyos atributos determinan los criterios de búsqueda. La consulta devuelve aquellos objetos cuyos atributos encajan con los atributos no nulos del prototipo. Por ejemplo:

- Si el prototipo es `new Producto(1023, null, null)`, la consulta devuelve el producto con el código 1023.
- Si el prototipo es `new Proveedor(null, null, null, "Madrid")`, la consulta devuelve los proveedores cuya población es Madrid.
- Si el prototipo es `new Producto(null, "Aceite", "Botella de 400ml")`, la consulta devuelve los productos cuyo nombre es `Aceite` y su descripción es `Botella de 400ml` (es decir, los que cumplen ambas condiciones simultáneamente).
- Si el prototipo es `new Proveedor(null, null, null, null)`, la consulta devuelve todos los proveedores de la base de datos.

Implementa un método genérico `List<T> queryByExample(T prototype)` dentro de `AbstractMapper` que reciba un prototipo y devuelva los objetos de la tabla correspondiente que encajen con el prototipo. La consulta SQL que se construya ha de ser paramétrica. Puedes definir los métodos abstractos adicionales que necesites en `AbstractMapper`, pero tendrás que implementarlos en los mappers concretos (`ProductoMapper` y `ProveedorMapper`).

Solución

Necesitamos un método abstracto adicional en `AbstractMapper`:

```
/**
 * Descompone un objeto en un array con sus componentes. Las componentes
 * han de aparecer en el mismo orden que los nombres de columnas devueltos
 * por getColumnNames()
 */
protected abstract Object[] decomposeObject(T object);
```

Este método recibe un objeto y construye un array con sus componentes. Por ejemplo, dado el objeto `new Producto(1023, null, null)`, su descomposición sería el array `{1023, null, null}`. Esta función se implementa en las subclases del siguiente modo:

- En `ProveedorMapper`:

```
@Override
protected Object[] decomposeObject(Proveedor object) {
  return new Object[] { object.getCif(),
                        object.getNombre(),
                        object.getTelefonoContacto(),
                        object.getPoblacion() };
}
```

- En `ProductoMapper`:

```
    @Override
    protected Object[] decomposeObject(Producto object) {
      return new Object[] { object.getCodigo(),
                            object.getNombre(),
                            object.getDescripcion() };
    }
```

El método `decomposeObject` ya ha aparecido antes en los ejercicios de clase. Era el método que necesitábamos para construir una operación `update` genérica. En el caso de este ejercicio, se pedía implementar una consulta mediante ejemplos. A partir del array proveniente del objeto descompuesto tenemos que añadir una condición WHERE por cada uno de los elementos de este array que sea distinto de `null`.

```
  public List<T> queryByExample(T prototype) {
    List<T> result = new LinkedList<T>();
    Object[] components = decomposeObject(prototype);
    String[] columns = getColumnNames();
    List<String> conditions = new LinkedList<String>();
    List<Object> values = new LinkedList<Object>();

    for (int i = 0; i < components.length; i++) {
      if (components[i] != null) {
        conditions.add(columns[i] + " = ?");
        values.add(components[i]);
      }
    }

    String sql = "SELECT " + StringUtils.join(columns, ", ") +
        " FROM " + getTableName() +
        (!conditions.isEmpty() ?
                " WHERE " + StringUtils.join(conditions, " AND ") : "");

    try(Connection con = ds.getConnection();
        PreparedStatement pst = con.prepareStatement(sql)) {
      for (int i = 0; i < values.size(); i++) {
        pst.setObject(i + 1, values.get(i));
      }

      ResultSet rs = pst.executeQuery();
      while (rs.next()) {
        result.add(buildObject(rs));
      }
    } catch (SQLException e) {
      e.printStackTrace();
    }
    return result;
  }
```

▷ 2. Correspondencia objeto-relacional. Sistemas ORM

Partimos del diagrama ER de la Figura 1, en el que se modela una base de datos sobre clientes, productos y proveedores. Cada cliente dispone de una lista de productos favoritos. Recíprocamente, un producto puede estar en la lista de favoritos de varios clientes. Por otro lado, la relación `Compra` refleja el hecho de que un cliente puede adquirir un determinado producto de un determinado proveedor. Es posible que un mismo cliente compre varias veces el mismo producto de distintos proveedores. Para cada compra se almacena la fecha en la que ésta se lleva a cabo.

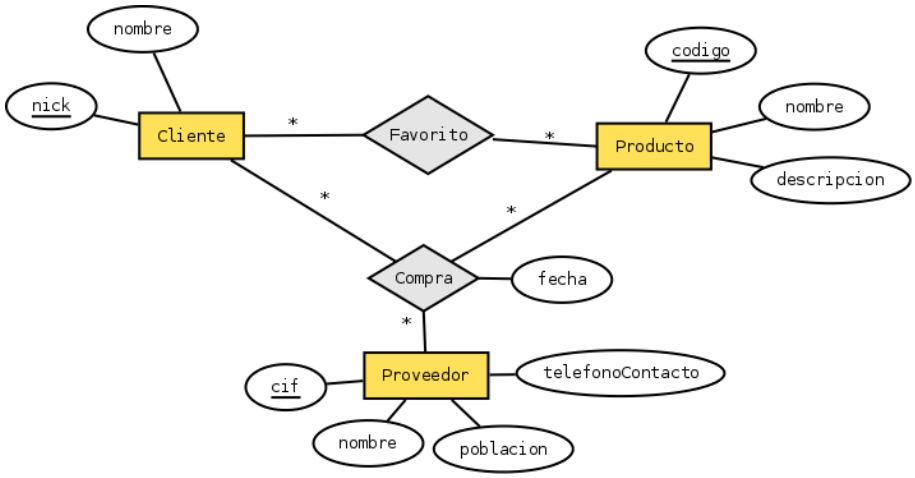


Figura 1: Diagrama ER de portal de compra de productos

(a). *[1 pt]* Suponiendo ya implementadas las clases `Cliente`, `Producto` y `Proveedor`, cada una conteniendo los correspondientes atributos mostrados en el diagrama ER, añade las anotaciones *Hibernate* necesarias para que se genere una BD que modele adecuadamente la información especificada en el diagrama ER. Añade las clases adicionales que necesites.

(b). *[0.5 pt]* Implementa un método

```
private void comprar(SessionFactory sf, Cliente cliente, Producto producto,
            Proveedor proveedor, Date fecha)
```

que añada una entrada nueva a la relación *Compra* de la base de datos con la información pasada como parámetro.

(c). *[1 pt]* Implementa, utilizando una consulta HQL, un método

```
private List<Object[]> numeroFavoritos(SessionFactory sf);
```

que devuelva, para cada producto de la base de datos, el número de usuarios que lo tienen en su lista de favoritos. La información se debe devolver como una lista de arrays. Cada array contiene dos elementos: el código del producto y el número de usuarios que han marcado dicho producto como favorito.

Solución

(a). *[1 pt]*

Las clases `Cliente`, `Producto` y `Compra` quedan del siguiente modo:

```
@Entity
public class Cliente {
  @Id
  private String nick;
  private String nombre;
  @ManyToMany
  private List<Producto> favoritos;
...
}


@Entity
public class Producto {
  @Id
  private Integer codigo;
  private String nombre;
  private String descripcion;

  // Clientes que han marcado este producto como favorito
  // El atributo mappedBy puede ir en la anotación @ManyToMany de favoritos
  // (clase Cliente) en lugar de aquí.
  @ManyToMany(mappedBy = "favoritos")
  private List<Cliente> clientes;
  ...
}


@Entity
public class Proveedor {
  @Id
  private String cif;

  private String nombre;
  private String telefonoContacto;
  private String poblacion;
  ...
}
```

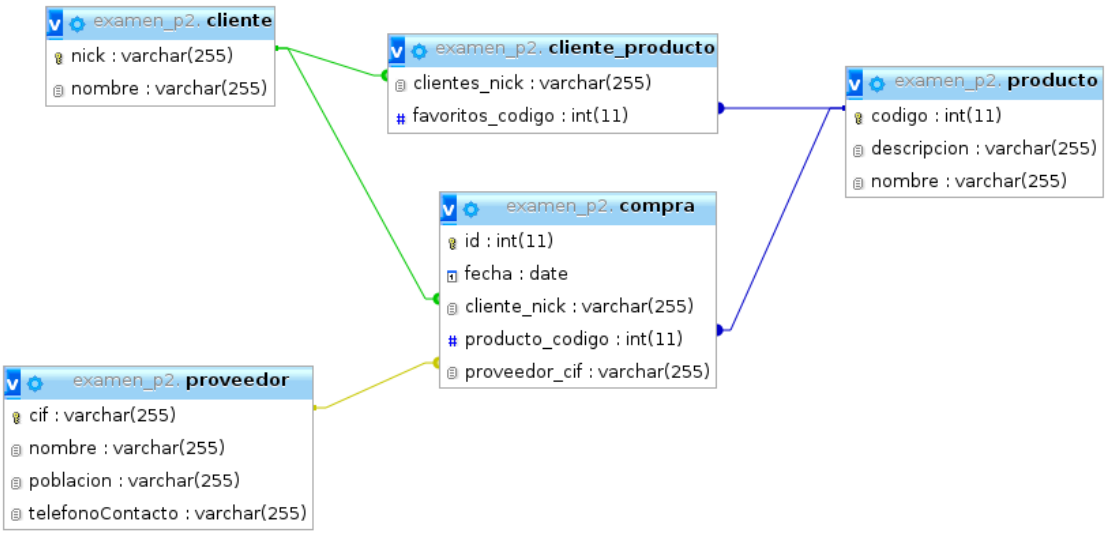
Para modelar la relación ternaria *Compra* es necesaria una clase nueva (`Compra`) que contenga las entidades que participan en esta relación, más un nuevo campo para el atributo `fecha`. Además, hemos de añadir un campo identificador (`id`) gestionado como un `AUTO_INCREMENT`.

```
@Entity
public class Compra {
  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Integer id;

  @ManyToOne
  private Cliente cliente;
  @ManyToOne
  private Producto producto;
  @ManyToOne
  private Proveedor proveedor;

  @Temporal(TemporalType.DATE)
  private Date fecha;
  ...
}
```

Con esto se generaría el siguiente esquema relacional:



(b). *[0.5 pt]*

```
public void comprar(SessionFactory sf, Cliente cliente,
            Producto producto, Proveedor proveedor, Date fecha) {
  Session s = sf.openSession();
  Transaction tr = s.beginTransaction();

  s.save(new Compra(cliente, producto, proveedor, fecha));
```

```
    tr.commit();
    s.close();
  }
```

(c). *[1 pt]*

```
  public static List<Object[]> numeroFavoritos(SessionFactory sf) {
    String hql = "SELECT p.codigo, COUNT(c.nick) FROM Producto p " +
                 "LEFT JOIN p.clientes c GROUP BY p.codigo";

    // Ejecutamos consulta HQL y devolvemos el resultado.
    Session s = sf.openSession();
    Query q = s.createQuery(hql);

    @SuppressWarnings("unchecked")
    List<Object[]> result = (List<Object[]>) q.list();

    s.close();
    return result;
  }
```

▷ 3. Modelo semiestructurado. Bases de datos XML

Consideramos un documento `Mensajeria.xml` como el mostrado en la Figura 2. En este documento se almacenan los usuarios y mensajes gestionados por un servicio de mensajería instantánea. La base de datos contiene un conjunto de cero, uno, o más usuarios. Cada usuario tiene un identificador, un nombre y, opcionalmente, una edad. Además, para cada usuario se almacenan los mensajes *enviados* por éste. Cada mensaje contiene un identificador, una fecha y el identificador del usuario al que va dirigido. Para ahorrar espacio de almacenamiento se tratan los mensajes duplicados mediante la distinción entre dos tipos de mensajes:

- Los mensajes originales (`<mensaje>`), que contienen un texto. Cada uno de ellos está identificado unívocamente mediante su `id`.
- Los reenvíos de un mensaje original (`<reenvio>`). En su atributo `id` contienen una referencia a un mensaje original (`<mensaje>`) ya existente.

Puedes suponer que los mensajes originales y los reenvíos no se "entremezclan" (es decir, dentro de un usuario aparecen primero todos los `<mensaje>` y luego los `<reenvio>`.

(a). *[1 pt]* Especifica la DTD asociada a este documento.

(b). *[0.5 pt]* Decimos que un usuario es un *forever alone* si se envía un mensaje *original* a sí mismo. Escribe una consulta *XQuery* que devuelva los nombres de los usuarios *forever alone*. En este apartado no tengas en cuenta los reenvíos de mensajes.

```
<mensajeria>
  <usuario id="u1">
    <nombre>Benito Benítez</nombre>
    <edad>31</edad>
    <mensaje id="m1" fecha="2015-03-27" destino="u2">
      Estoy en el metro. Llego en 10 minutos.
    </mensaje>
    <mensaje id="m3" fecha="2015-03-14" destino="u3">
      Reenvia este mensaje a diez personas y se te aparecerá
      la virgen para darte las gracias.
    </mensaje>
    ...
    <reenvio id="m3" fecha="2015-03-14" destino="u2"/>
    ...
  </usuario>
  <usuario id="u2">
    <nombre>Marta Mártez</nombre>
    <edad>27</edad>
    <mensaje id="m4" fecha="2015-03-27" destino="u1">
      No mientas. Sé que todavía estás en casa
    </mensaje>
    ...
    <reenvio id="m3" fecha="2015-03-27" destino="u3"/>
    <reenvio id="m3" fecha="2015-03-27" destino="u5"/>
    ...
  </usuario>
  ...
</mensajeria>
```

Figura 2: Base de datos XML de sistema de mensajería instantanea

(c). *[1 pt]* Escribe una consulta *XQuery* que devuelva los nombres de todos los usuarios del sistema, cada ellos con el número de mensajes (tanto originales como reenvíos) que han enviado el día `2015-03-27`. El resultado debe de estar ordenado de forma decreciente según el número de mensajes enviados. Cada elemento del resultado ha de tener el siguiente formato:

```
<usuario nombre="nombre del usuario" num-envios="número de envíos"/>
```

Solución

(a). *[1 pt]*

```
<!ELEMENT mensajeria (usuario*)>
<!ELEMENT usuario (nombre, edad?, mensaje*, reenvio*)>
<!ELEMENT nombre (#PCDATA)>
<!ELEMENT edad (#PCDATA)>
<!ELEMENT mensaje (#PCDATA)>
<!ELEMENT reenvio EMPTY>
```

```
<!ATTLIST usuario id ID #REQUIRED>
<!ATTLIST mensaje id ID #REQUIRED
                  fecha CDATA #REQUIRED
                  destino IDREF #REQUIRED>
<!ATTLIST reenvio id IDREF #REQUIRED
                  fecha CDATA #REQUIRED
                  destino IDREF #REQUIRED>
```

(b). *[0.5 pt]*

```
for $u in doc("Mensajeria.xml")//usuario
where count($u/mensaje[@destino = $u/@id]) > 0
return $u/nombre
```

(c). *[1 pt]*

```
for $u in doc("Mensajeria.xml")//usuario
let $mensajesEnviados := count($u/mensaje) + count($u/reenvio)
order by $mensajesEnviados descending
return <usuario nombre="{$u/nombre/text()}" num-envios="{$mensajesEnviados}"/>
```

▷ 4. Transacciones y control de la concurrencia

Supongamos que los elementos de una base de datos contienen información binaria (0 o 1). El gestor de transacciones controla el acceso concurrente a estos elementos mediante cuatro tipos de candados:

- `RLOCK X`: Adquiere un candado sobre `X` para leer su valor.
- `ZEROLOCK X`: Adquiere un candado sobre `X` para establecer su valor a 0.
- `ONELOCK X`: Adquiere un candado sobre `X` para establecer su valor a 1.
- `INVLOCK X`: Adquiere un candado sobre `X` para invertir su valor de manera atómica. Es decir, si el valor actual de `X` es 0 se establece a 1 y viceversa.

[T1]	[T2]	[T3]
RLOCK A		
UNLOCK A		
	ZEROLOCK A	
		RLOCK B
		ZEROLOCK C
		UNLOCK B
		UNLOCK C
	ONELOCK B	
	UNLOCK A	
INVLOCK A		
ZEROLOCK C		
UNLOCK C		
UNLOCK A		
	UNLOCK B	

Figura 3

(a). *[0.7 pt]* Escribe la matriz de compatibilidad de estas cuatro operaciones.

(b). *[0.8 pt]* Dado el plan de transacciones de la Figura 3, dibuja todas las aristas del grafo de dependencias entre las tres transacciones e indica si el plan es serializable. En caso afirmativo, indica un plan en serie equivalente.

Solución

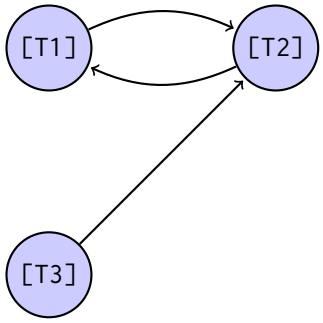
(a). *[0.7 pt]* Todas las instrucciones conmutan consigo mismas, pero no conmutan entre ellas, por lo que la matriz de compatibilidades tiene el siguiente aspecto:

	RLOCK	ZEROLOCK	ONELOCK	INVLOCK
RLOCK	SÍ	NO	NO	NO
ZEROLOCK	NO	SÍ	NO	NO
ONELOCK	NO	NO	SÍ	NO
INVLOCK	NO	NO	NO	SÍ

(b). *[0.8 pt]* El grafo de dependencias se construye del siguiente modo:

- Arista de `[T1]` a `[T2]` por las instrucciones `T1: RLOCK A` $\rightarrow$ `T2: ZEROLOCK A`.
- Arista de `[T2]` a `[T1]` por las instrucciones `T2: ZEROLOCK A` $\rightarrow$ `T1: INVLOCK A`.
- Arista de `[T3]` a `[T2]` por las instrucciones `T3: RLOCK B` $\rightarrow$ `T2: ONELOCK B`.

No se dibuja una arista de `[T3]` a `[T1]` debido al `ZEROLOCK C`, ya que la matriz contiene un SÍ en la celda de ZEROLOCK consigo misma. Tenemos el siguiente grafo de dependencias:



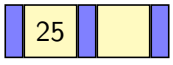
Existe un ciclo entre `[T1]` y `[T2]`, por lo que el plan de transacciones no es serializable.

▷ 5. Almacenamiento e índices

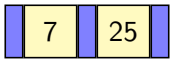
[1 pt] Dado un árbol B+ en el que cada nodo puede tener hasta dos claves y tres hijos (es decir, $n = 3$), partimos de un único nodo con el número 25. Indica el árbol que se obtiene tras insertar sucesivamente las claves 7, 14, 29, 17 y 3 (en este orden) en el árbol inicial. Dibuja también los árboles intermedios que resultan tras insertar cada uno de estos elementos.

Solución

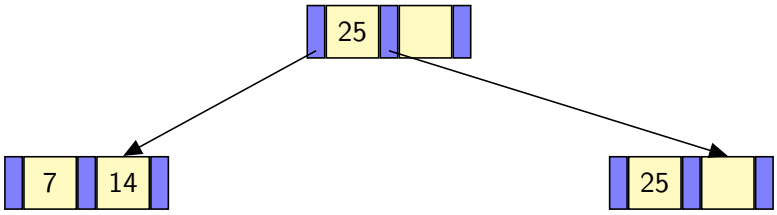
Omitimos los punteros entre hojas, por simplicidad. Árbol inicial:



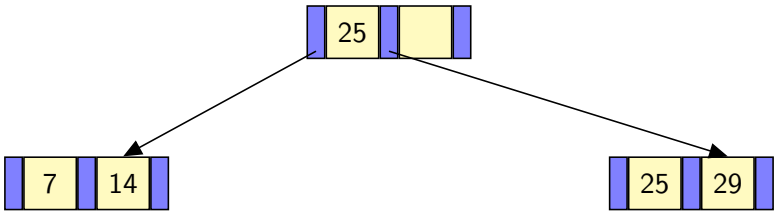
Tras insertar el 7:



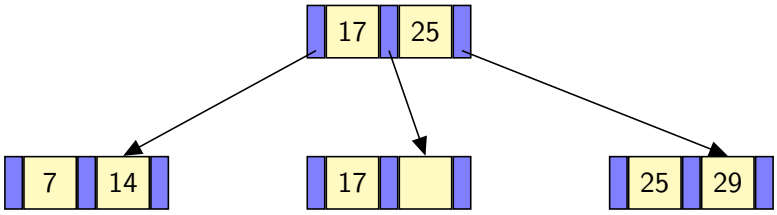
Tras insertar el 14:



Tras insertar el 29:



Tras insertar el 17:



Tras insertar el 3: