Calidad de los Sistemas Informáticos, 2020-2021
Práctica 7 – Optimización

Objetivo

El objetivo de esta práctica es revisar y optimizar aspectos del código poco eficientes o reductibles. Para su puesta en práctica, deberá resolverse cada problema según se indica en la solución antes de proceder al siguiente problema. En la práctica se trabajarán índices, desencadenadores, constructores privados, herencia y tipos de datos.

Requisitos previos

Para la realización de esta práctica se requieren los recursos y resultados de las prácticas 1 a 6.

Criterios sintácticos de este documento

- `Courier`: código fuente, nombres de archivos, paquetes, entidades, atributos, tablas y campos.
- *`Cursiva`*: términos en otro idioma.
- Negrita: contenido resaltado.
- **$**Entre dólar**$**: contenido no literal, generalmente a decidir por el desarrollador.

Pablo de la Torre Moreno
Departamento de Ingeniería Informática

Procedimiento

Base de datos

1. Revisión de índices y campos únicos

Problema

Es posible que no estén creados todos los posibles índices para campos de texto candidatos a ser buscados. Si bien hasta ahora la aplicación no aprovecharía dichos índices (ya que las búsquedas utilizan comodines delante y detrás del campo de texto), en esta práctica se solventará dicha eventualidad.

Solución

Crear índices para los campos que se prevean más buscados, y no para los demás, dado que los índices toman recursos. Crear índices únicos para atributos `Nombre` de las tablas de tipo y campos donde no se permitan valores duplicados. No debe olvidarse exportar el código de la base de datos e incluirlo en el SQL del proyecto.

2. Impedir valores no permitidos en la base de datos

Problema

A pesar de que se pueden controlar los valores de entrada mediante precondiciones en el código de la aplicación, aún es posible insertar en la base de datos valores de dominios no permitidos –por ejemplo, una cadena vacía en un campo `Nombre`– directamente desde `INSERT` o `UPDATE`. Esto abre una posibilidad de error tanto en la aplicación como en la administración directa del servicio: ¿Y si el administrador abre la tabla y, sin querer, borra un campo y lo deja vacío? ¿Y si las precondiciones no están bien escritas en la aplicación? Esta estrategia deberá seguirse siempre que sea posible, para garantizar las reglas de negocio.

Solución

Utilizar disparadores (*triggers*) PREVIOS a los cambios en los datos, que no permitan la inserción o modificación si las precondiciones no se cumplen. Se incluye un ejemplo[1], en un contexto de personajes fantásticos, con una tabla `Race` que tiene los campos `Id` y `Name`:

```
delimiter $$
create trigger `Race_bi` before insert on `Race`
for each row
begin
  if NEW.Name='' then
    signal sqlstate '45000' set
        message_text = 'El nombre de la raza no puede estar vacío.';
  end if;
end$$
delimiter ;
```

[1] Deberá hacerse otro para la actualización –idéntico, pero cambiando `insert` por `update` en la definición, bu en vez de `bi` en el nombre–, y todos los cambios en uno deberán replicarse en el otro. Se pueden ver todos los desencadenadores con `show triggers`, y se eliminan con `drop trigger [tabla].[desdencadenador]`. Si se hace a través de phpMyAdmin, debe escribirse exclusivamente lo que se encuentra entre `begin` y `end$$` del código anterior. Aunque el entorno indique errores con los dobles dólares, debe de funcionar al ejecutar.

Deberán cambiarse los nombres del código para adaptarlo al proyecto actual, así como ampliarse el código para todos los campos de las dos tablas implicadas en la práctica. Asimismo, se eliminarán o comentarán las precondiciones en código y se probará la aplicación con los campos vacíos y/o valores no permitidos, observándose la respuesta.

A los disparadores puede accederse desde el menú de la estructura de la tabla, *Más > Disparadores*.

Capa de acceso a datos

3. Reducción del número de conexiones en la búsqueda

Problema

El método `Select` de las clases de entidades invoca tantas veces al constructor de la clase como número de elementos encontrados. Cada constructor abre una conexión y la cierra. Las conexiones son recursos caros.

Solución

Modificar la búsqueda de `Select` para incluir todos los campos. Crear un segundo constructor privado que reciba como parámetros todos estos valores (incluido `Id`) y los asigne a las variables privadas. Invocar a dicho constructor desde `Select` al montar la lista, en lugar de al constructor original.

Solución alternativa

Crear un nuevo constructor que reciba el identificador (como el actual) y un parámetro de tipo `Connection`, usando éste para acceder a la base de datos. Para evitar código duplicado entre ambos constructores, crear un método privado `void Initialize(int iId, Connection con)` que haga dicho trabajo, siendo invocado desde ambos constructores.

4. Reducción del número de conexiones en las construcciones anidadas

Problema

En menor medida, los constructores anidados adolecen del mismo problema que el referido en el punto anterior: el constructor público de **$**`Entidad`**$** llama al constructor público de **$**`TipoEntidad`**$**, que crea una nueva conexión.

Solución

Siguiendo la solución alternativa del punto anterior, crear un segundo constructor público que, además del identificador, reciba un parámetro de tipo conexión. Pasar el código del constructor original a un método `void Initialize(int iId, Connection con)`, invocado por ambos constructores (el constructor original deberá ahora crear una conexión antes de llamar a dicho método, y liberarla al final). Modificar la llamada al constructor de **$**`TipoEntidad`**$** por parte de **$**`Entidad`**$** para que le pase también la conexión. Implementar la misma estructura de constructores en **$**`Entidad`**$**.

5. Clase base

Problema

Empieza a observarse código que se repite en las clases de la lógica desarrolladas hasta el momento. La solución a este problema se empleará, asimismo, en el siguiente.

Solución

Crear en el paquete `data` una clase abstracta llamada `Entidad` (o `Entity`) y que contenga los siguientes métodos (con las variables necesarias correspondientes).

- `public int getId()`
- `public bool getIsDeleted()`
- `protected void Update(String sQuery)`
- `public void Delete()`

Hacer que el resto de las clases del paquete data hereden de dicha clase, eliminando los métodos y variables ya innecesarios, y modificando el método `Update()` para que invoque al del padre con la consulta construida (`super.Update(sQuery)`).

6. Método *Where*

Problema

El método `Where` debe ser construido manualmente para cada entidad. Aunque no es especialmente problemático –ya que siempre se hace igual–, ilustrativamente se podría aprovechar la herencia y que el modo de crear condiciones coincide para mismos tipos de datos.

Solución

Pasar el método como protegido a la clase `Entidad` (o `Entity`), y que recibe los siguientes parámetros: una lista de los campos, una lista de valores de la clase `java.sql.Types` y una lista de valores. Para cada campo y si el valor respectivo no es `null`, se añadirá a la lista resultante (`StringBuilder`, como antes) una condición en función del tipo respectivo del segundo parámetro.

Una vez realizado, se modificará el `Select` de **`$`**`Entidad`**`$`** tal que invoque al método anterior (al de la superclase) como se indica en el ejemplo:

```
Entity.Where(
      new String[] { "StoryCharacter.Name",
                     "StoryCharacter.Age", "Race.Name"},
      new int[] { Types.VARCHAR,
                  Types.INTEGER, Types.VARCHAR },
      new Object[] { sName, iAge, sRacer })
```

Siendo los valores de la última lista los recibidos por los parámetros de entrada de `Select`. El método `Where` deberá iterar entre los valores de la primera lista, revisar si no es

nulo el respectivo de la tercera e incluir la condición en función del tipo del respectivo de la segunda[2].

Capa de presentación

7. Evitación de instancias duplicadas

Problema

Hacer clic en el mismo elemento que aparezca en dos resultados de dos formularios de búsqueda diferentes genera dos objetos distintos[3].

Solución

Tras pulsar doble clic, el algoritmo deberá buscar en todos los formularios de detalle abiertos de ese tipo si representa un registro con el mismo identificador que el escogido. De ser así, no se abrirá un nuevo formulario detalle, sino que se mostrará el que lo contiene. Esto requiere un acceso a todos los formularios de detalle abiertos, lo que implica tener una lista con todos ellos en `FrmMain` (por ejemplo), accesible por todos los formularios de búsqueda.

[2] Para ello se requiere el método `Where` suficientemente bien optimizado y adaptable fácilmente a cualquier número de parámetros.
[3] No es infrecuente la construcción de *pools* de información de manera similar a ésta.