ACCESO A DATOS

UNIDAD 3. ACCESO A BASES DE DATOS RELACIONALES

ÍNDICE DE CONTENIDOS

INTRODUCCIÓN 2

1. INTRODUCCIÓN A JDBC 4

1.1. TIPOS DE CONTROLADORES JDBC 4

2. CONSULTAS A LA BASE DE DATOS 10

2.1. CLASE STATEMENT 10

2.2. CLASE PREPAREDSTATEMENT 17

2.3. CLASE RESULTSET 25

3. OPERACIONES CRUD 32

3.1. CREATE 32

3.2. READ 37

3.3. UPDATE 40

3.4. DELETE 43

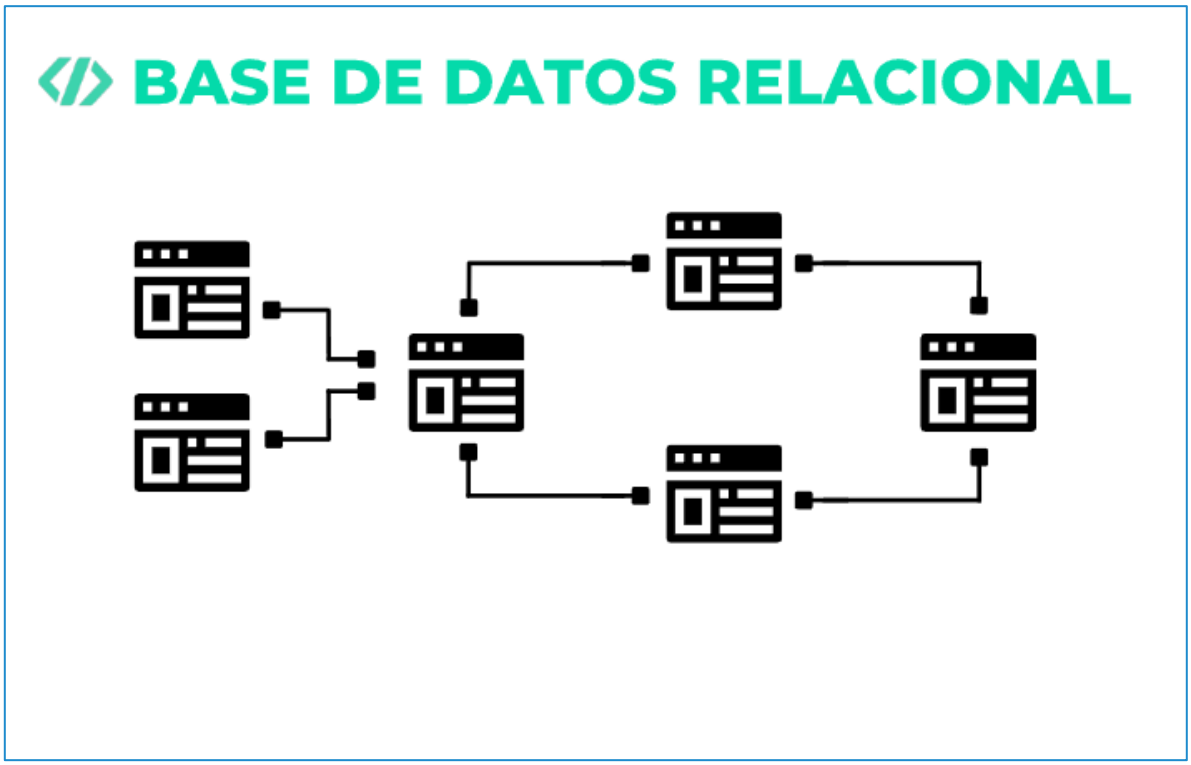
4. INTEGRIDAD DE DATOS 48

4.1. COMMIT 48

4.2. ROLLBACK 50

RESUMEN 53

INTRODUCCIÓN

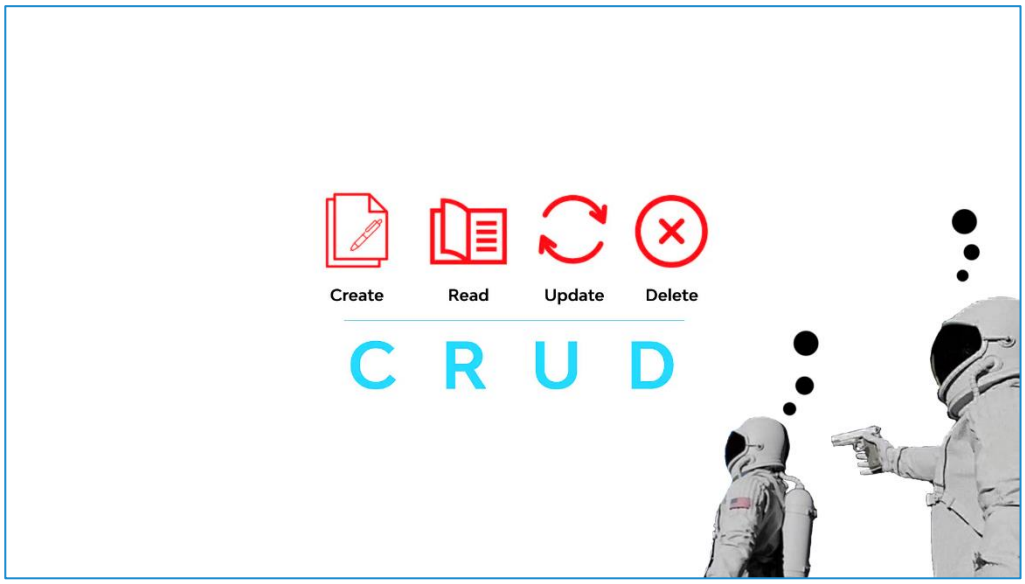


El acceso a bases de datos se realiza a través de JDBC, que es la tecnología de Java diseñada para permitir que las aplicaciones Java se conecten y se comuniquen con bases de datos relacionales. **JDBC** (**J**ava **D**ata**b**ase **C**onnectivity) **proporciona un conjunto de interfaces y clases que facilitan la interacción con cualquier sistema de gestión de bases de datos que utilice SQL**.

El proceso de consulta a una base de datos se realiza a través de diversas clases proporcionadas por JDBC, entre las que se destacan "Statement", "PreparedStatement" y "ResultSet". **La clase "Statement" es utilizada para ejecutar sentencias SQL simples.** Su uso es habitual en situaciones donde no se requiere parametrizar las consultas. Sin embargo, su implementación puede ser menos segura frente a ataques de inyección SQL, donde el atacante intenta manipular la consulta SQL a través de entradas maliciosas. **Para mitigar este riesgo, se recomienda utilizar "PreparedStatement"**. Esta clase permite precompilar las consultas SQL y luego ejecutarlas múltiples veces con diferentes parámetros. Esto no solo mejora el rendimiento de la aplicación al reducir la sobrecarga de compilación de la consulta, sino que también refuerza la seguridad mediante el uso de parámetros que se manejan de manera separada de la consulta SQL. "ResultSet" permite tratar los resultados devueltos por las consultas SQL. Proporciona métodos que permiten recorrer los registros devueltos y extraer los datos de las columnas de cada fila.

Las operaciones CRUD son el conjunto de acciones más básicas que pueden realizarse sobre la información almacenada en una base de datos. Este acrónimo se refiere a las operaciones de Crear, Leer, Actualizar y Eliminar datos. La operación de creación (Create) se refiere a la inserción de

nuevos registros en la base de datos, que puede implicar la definición de una nueva entidad o la adición de información a una tabla existente. La lectura (Read) involucra la recuperación de datos almacenados, lo que permite a la aplicación obtener la información necesaria para su funcionamiento o para su presentación al usuario. La actualización (Update) se utiliza para modificar los registros existentes en la base de datos, permitiendo reflejar cambios en la información. Finalmente, la eliminación (Delete) permite quitar registros de la base de datos, lo que puede ser necesario en un proceso de mantenimiento o cuando la información ya no es relevante.



La **integridad de los datos** es un aspecto clave al trabajar con bases de datos, y se asegura mediante el uso de transacciones. Las transacciones son una serie de operaciones que se ejecutan como una unidad discreta, lo que significa que o se completan todas las acciones o no se realiza ninguna. Para gestionar estas transacciones, se utilizan los comandos 'commit' y 'rollback'. Cuando todos los cambios realizados en el contexto de una transacción se completan con éxito, se utiliza 'commit' para confirmar los cambios y hacerlos permanentes en la base de datos. Este comando garantiza que los datos se mantengan consistentes después de una serie de operaciones. En caso de que ocurra un error en alguno de los pasos de la transacción, el uso de 'rollback' permite revertir todos los cambios realizados hasta ese momento, devolviendo la base de datos a un estado previo. Este manejo de transacciones es un componente primordial para garantizar que la aplicación funcione de manera fiable y segura, evitando inconsistencias en los datos que podrían comprometer la calidad y la efectividad del software.

1. INTRODUCCIÓN A JDBC

Java Database Connectivity (JDBC) es una API que permite a las aplicaciones Java interactuar con bases de datos relacionales. La conexión a bases de datos mediante JDBC implica varios pasos (ordenados), que abarcan:

- La configuración de controladores
- La gestión de conexiones
- La ejecución de consultas SQL
- La manipulación de resultados.

JDBC se basa en un diseño que permite la separación entre la lógica de la aplicación y la gestión de datos. Esto significa que la aplicación no necesita preocuparse por los detalles del almacenamiento, lo que facilita el desarrollo de soluciones que pueden interactuar con diferentes bases de datos.

1.1. TIPOS DE CONTROLADORES JDBC

Los controladores JDBC son elementos necesarios que permiten que las aplicaciones Java accedan a datos en un sistema de gestión de bases de datos específico. Cada tipo de controlador tiene características y usos que reflejan diferentes necesidades de implementación.

- **Controlador JDBC-ODBC puente**: Este controlador permite el acceso a bases de datos que utilizan ODBC como interfaz. Aunque facilita la compatibilidad con diversas fuentes de datos, su rendimiento no es óptimo y no es recomendable para aplicaciones de producción. Se suele utilizar en entornos de desarrollo o pruebas donde se requiere un acceso rápido a múltiples tipos de bases de datos.
- **Controlador nativo de API**: Este tipo de controlador se comunica con el SGBD utilizando su propia API. Proporciona un mejor rendimiento en comparación con el controlador ODBC, aunque introduce la dependencia del software específico del SGBD, lo que puede limitar la capacidad de traslado de la aplicación.
- **Controlador de red puro**: Estos controladores organizan la interacción en una arquitectura cliente-servidor y permiten que las aplicaciones se comuniquen directamente con el servidor de la base de datos a través de la red. Esto proporciona un rendimiento optimizado y facilita la administración de las bases de datos de manera centralizada.
- **Driver JDBC puro Java**: Estos controladores están completamente escritos en Java, lo que implica que pueden ejecutarse en cualquier entorno que soporte Java sin necesidad de componentes adicionales. Esto asegura una distribución y uso más eficientes en aplicaciones que requieren alta portabilidad.

1.1.1. Estableciendo una conexión

Para iniciar la interacción con una base de datos utilizando JDBC, es necesario establecer una conexión. Se ejemplificará un proceso de conexión usando el controlador correspondiente para acceder a una base de datos MySQL.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

public class ConexionBD {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_basedatos";
        String usuario = "mi_usuario";
        String contraseña = "mi_contraseña";

        try {
            Connection conexion = DriverManager.getConnection(url, usuario, contraseña);
            System.out.println(x:"Conexión establecida con éxito.");
            // Operaciones sobre la base de datos.
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Es relevante manejar adecuadamente las excepciones SQL. Esto es necesario para que los desarrolladores puedan identificar problemas con la conexión y abordarlos de manera efectiva.

1.1.2. Consultas y ejecución de comandos SQL

Las consultas en la base de datos se ejecutan a través de diferentes interfaces que proporciona JDBC: `Statement`, `PreparedStatement` y `CallableStatement`. Cada uno de estos objetos tiene su propio uso y particularidades.

- **Statement**: Se utiliza para ejecutar consultas SQL simples. A continuación, se presenta un ejemplo de cómo utilizar un objeto `Statement` para realizar una consulta y obtener datos.

  El uso de `Statement` es adecuado cuando las consultas SQL son conocidas de antemano y no se requiere parametrización.

```java
import java.sql.Connection;
import java.sql.Statement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConsultaBD {
    Run | Debug
    public static void main(String[] args) {
        Connection conexion = null; // Asegúrate de que la conexión esté inicializada previamente.

        try {
            Statement statement = conexion.createStatement();
            String consulta = "SELECT * FROM empleados";
            ResultSet resultado = statement.executeQuery(consulta);

            while (resultado.next()) {
                System.out.println("ID: " + resultado.getInt(columnLabel:"id") + ", Nombre: " + resultado.getString(columnLabel:"nombre"));
            }

            // Cerrar ResultSet y Statement
            resultado.close();
            statement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Asegurarse de cerrar la conexión si no es null
            if (conexion != null) {
                try {
                    conexion.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

- **PreparedStatement**: Este objeto permite ejecutar consultas SQL predefinidas con parámetros. Se recomienda su uso en situaciones donde se necesita prevenir inyecciones SQL. Un ejemplo de inserción de datos usando `PreparedStatement` es el siguiente:

```java
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InsertarEmpleado {
    Run | Debug
    public static void main(String[] args) {
        Connection conexion = null; // Asegúrate de que la conexión esté inicializada previamente.

        try {
            // Consulta de inserción
            String sqlInsertar = "INSERT INTO empleados (nombre, edad) VALUES (?, ?)";

            // Preparar el statement
            PreparedStatement preparedStatement = conexion.prepareStatement(sqlInsertar);
            preparedStatement.setString(parameterIndex:1, x:"Juan Pérez");
            preparedStatement.setInt(parameterIndex:2, x:30);

            // Ejecutar la inserción
            int filasAfectadas = preparedStatement.executeUpdate();
            System.out.println("Filas insertadas: " + filasAfectadas);

            // Cerrar el PreparedStatement
            preparedStatement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Asegurarse de cerrar la conexión si no es null
            if (conexion != null) {
                try {
                    conexion.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

En este caso, los parámetros se establecen antes de la ejecución de la consulta, lo que mejora la seguridad y eficiencia. Utilizar `PreparedStatement` es beneficioso cuando se ejecuta la misma consulta repetidamente, ya que solo se compila una vez.

- **CallableStatement**: **Este objeto se usa para ejecutar procedimientos almacenados en la base de datos**. **Permite pasar parámetros y también recuperar resultados**. Un ejemplo de cómo llamar a un procedimiento almacenado puede ser el siguiente:

```java
import java.sql.Connection;
import java.sql.CallableStatement;
import java.sql.SQLException;

public class ObtenerEmpleado {
    Run | Debug
    public static void main(String[] args) {
        Connection conexion = null; // Asegúrate de que la conexión esté inicializada previamente.

        try {
            // Preparar la llamada al procedimiento almacenado
            CallableStatement callableStatement = conexion.prepareCall(sql:"{call obtenerEmpleado(?, ?)}");

            // Establecer los parámetros de entrada y salida
            callableStatement.setInt(parameterIndex:1, x:1); // Parámetro de entrada (ID del empleado)
            callableStatement.registerOutParameter(parameterIndex:2, java.sql.Types.VARCHAR); // Parámetro de salida (Nombre del empleado)

            // Ejecutar la llamada
            callableStatement.execute();

            // Obtener el valor del parámetro de salida
            String nombreEmpleado = callableStatement.getString(parameterIndex:2);
            System.out.println("Nombre del empleado: " + nombreEmpleado);

            // Cerrar el CallableStatement
            callableStatement.close();
        } catch (SQLException e) {
            e.printStackTrace();
        } finally {
            // Asegurarse de cerrar la conexión si no es null
            if (conexion != null) {
                try {
                    conexion.close();
                } catch (SQLException e) {
                    e.printStackTrace();
                }
            }
        }
    }
}
```

Este caso muestra cómo se pueden utilizar procedimientos almacenados, facilitando la lógica de negocio directamente en la base de datos.

1.1.3. Manejo de transacciones

Las transacciones son aspectos importantes en el trabajo con bases de datos, ya que permiten agrupar un conjunto de operaciones en una única unidad de trabajo. En JDBC, se puede gestionar el control de transacciones de la siguiente manera:

```java
try {
    // Desactivar el auto-commit para iniciar una transacción
    conexion.setAutoCommit(false);

    // Código para realizar operaciones SQL
    // ...

    // Confirmar la transacción
    conexion.commit();
} catch (SQLException e) {
    try {
        // Revertir cambios en caso de error
        if (conexion != null) {
            conexion.rollback();
        }
    } catch (SQLException rollbackEx) {
        rollbackEx.printStackTrace();
    }
    e.printStackTrace();
} finally {
    // Asegurarse de cerrar la conexión
    if (conexion != null) {
        try {
            conexion.setAutoCommit(true); // Volver a activar el auto-commit
            conexion.close();
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

El código anterior establece que la conexión no realice un *commit* automático después de cada operación, lo que permite agrupar múltiples operaciones en una sola transacción. En caso de un error, se lleva a cabo un *rollback* para revertir todos los cambios realizados hasta ese punto.

1.1.4. Cierre de recursos

Es importante cerrar todos los objetos utilizados para acceder a la base de datos. Esto se realiza para liberar recursos y evitar fugas de memoria. El cierre de recursos se ejecuta de manera controlada, utilizando bloques `try-catch-finally` para garantizar que las conexiones se cierren adecuadamente incluso si se produce una excepción.

```java
try {
    // Código de operaciones SQL
    // ...
} catch (SQLException e) {
    e.printStackTrace();
} finally {
    try {
        if (resultado != null && !resultado.isClosed()) {
            resultado.close(); // Cerrar ResultSet si no está cerrado
        }
        if (statement != null && !statement.isClosed()) {
            statement.close(); // Cerrar Statement si no está cerrado
        }
        if (conexion != null && !conexion.isClosed()) {
            conexion.close(); // Cerrar conexión si no está cerrada
        }
    } catch (SQLException e) {
        e.printStackTrace();
    }
}

```

Implementar correctamente el cierre de recursos contribuye a mantener la eficiencia y la estabilidad de la aplicación.

1.1.5. Casos de uso de JDBC

JDBC es ampliamente utilizado en diferentes aplicaciones. Por ejemplo, en entornos empresariales, JDBC es esencial para la construcción de sistemas de gestión, donde se requieren consultas a bases de datos para acceder a datos de clientes, pedidos y productos. En aplicaciones web, JDBC puede utilizarse para gestionar registros de usuarios, autenticar accesos y realizar transacciones en línea.

También se observa en aplicaciones de informes y análisis, donde facilita la extracción de datos de bases de datos para su procesamiento y visualización en tiempo real.

2. CONSULTAS A LA BASE DE DATOS

Las consultas de selección permiten obtener datos de una o varias tablas en la base de datos. Estas acciones pueden incorporar cláusulas como WHERE, que filtran los resultados, ORDER BY, que organizan los datos, y GROUP BY, que agrupan la información según ciertos criterios. Además, es posible aplicar funciones de agregado para facilitar el análisis de grandes volúmenes de datos.

Las consultas de inserción realizan la incorporación de nuevos registros a una tabla. Este proceso implica la especificación de las columnas en las que se insertarán los valores, junto con los datos correspondientes. En cuanto a las actualizaciones, permiten modificar la información de registros existentes, definiendo qué columnas y valores deben cambiarse bajo ciertas condiciones.

Por su parte, las consultas de eliminación se encargan de borrar registros de una tabla. Es importante proceder con precaución al ejecutar estas acciones para evitar la pérdida no intencionada de datos importantes. Las sentencias de consulta deben manejarse con cuidado, y se recomienda realizar copias de seguridad regularmente de la base de datos.

Las consultas a la base de datos pueden ejecutarse de manera sincrónica o asincrónica, dependiendo del entorno de programación utilizado. Existen diferentes métodos para llevar a cabo estas consultas, que incluyen el uso de diversas clases y enfoques dentro de los lenguajes de programación, facilitando así la interacción con el sistema de gestión de la base de datos.

2.1. CLASE STATEMENT

La clase ‘Statement’ en Java es parte de la API JDBC y permite ejecutar instrucciones SQL en bases de datos relacionales. **Se utiliza principalmente para ejecutar consultas estáticas que no requieren parámetros de entrada**. La simplicidad de esta clase facilita su uso, pero también es importante considerar aspectos relacionados con la seguridad y la eficiencia.

La creación de un objeto ‘Statement’ se realiza después de establecer una conexión con la base de datos. **Dicho proceso se realiza mediante el uso de ‘DriverManager,’ el cual gestiona los controladores JDBC**. A continuación, se presenta un ejemplo que muestra este proceso:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class ConexionBaseDatos {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        // Uso de try-with-resources para manejar la conexión y la sentencia de manera automática
        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             Statement sentencia = conexion.createStatement()) {

            // Aquí puedes añadir las instrucciones SQL que quieras ejecutar.
            // Por ejemplo:
            // String sql = "INSERT INTO empleados (nombre, edad) VALUES ('Juan Pérez', 30)";
            // sentencia.executeUpdate(sql);

            System.out.println(x:"Operación realizada con éxito.");
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Una vez que se tiene un objeto 'Statement', se pueden ejecutar varios tipos de instrucciones SQL. **Para las consultas que devuelven datos, se utiliza el método `executeQuery`, que está diseñado para instrucciones SQL que inician con SELECT**.

Aquí se incluye un ejemplo concreto de cómo obtener datos utilizando un 'Statement':

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarClientes {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        // Uso de try-with-resources para manejar la conexión, sentencia y result set
        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             Statement sentencia = conexion.createStatement();
             ResultSet resultado = sentencia.executeQuery(sql:"SELECT nombre, edad FROM clientes")) {

            // Procesar el resultado
            while (resultado.next()) {
                String nombre = resultado.getString(columnLabel:"nombre");
                int edad = resultado.getInt(columnLabel:"edad");
                System.out.println("Nombre: " + nombre + ", Edad: " + edad);
            }
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Este bloque de código realiza una consulta para recuperar los nombres y edades de los clientes y los imprime. La navegación en el 'ResultSet' se lleva a cabo mediante el método `next()`, que mueve el cursor a la siguiente fila de datos. Los métodos `getString` y `getInt` obtienen los valores en los tipos de datos apropiados.

Los objetos Statement también se utilizan para realizar modificaciones en la base de datos mediante el método `executeUpdate`. Este método es adecuado para instrucciones que no devuelven un conjunto de resultados, como INSERT, UPDATE o DELETE. A continuación, se muestra un ejemplo de cómo insertar un nuevo registro en la tabla:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class InsertarCliente {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        // Uso de try-with-resources para manejar la conexión y la sentencia
        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             Statement sentencia = conexion.createStatement()) {

            // Inserción de datos
            String insercion = "INSERT INTO clientes (nombre, edad) VALUES ('María López', 28)";
            int filasAfectadas = sentencia.executeUpdate(insercion);

            // Mostrar el número de filas afectadas
            System.out.println("Filas afectadas: " + filasAfectadas);
        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Este código inserta un nuevo cliente en la tabla y devuelve el número de filas que han sido afectadas, lo cual es útil para verificar si la operación se realizó con éxito.

Un caso de uso práctico para la clase 'Statement' podría ser el manejo de un sistema de seguimiento de pedidos. Si se desea obtener todos los pedidos asociados a un cliente específico, se podría realizar lo siguiente:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarPedidos {
    Run | Debug
    public static void main(String[] args) {
        int idCliente = 123;  // ID del cliente que se busca
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             Statement sentencia = conexion.createStatement()) {

            // Construcción de la consulta
            String consultaPedidos = "SELECT * FROM pedidos WHERE cliente_id = " + idCliente;

            // Ejecución de la consulta
            ResultSet resultadoPedidos = sentencia.executeQuery(consultaPedidos);

            // Procesar los resultados
            while (resultadoPedidos.next()) {
                int idPedido = resultadoPedidos.getInt(columnLabel:"id");
                String producto = resultadoPedidos.getString(columnLabel:"producto");
                int cantidad = resultadoPedidos.getInt(columnLabel:"cantidad");
                System.out.println("ID Pedido: " + idPedido + ", Producto: " + producto + ", Cantidad: " + cantidad);
            }

            // Cerrar el ResultSet
            resultadoPedidos.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Sin embargo, este enfoque tiene una vulnerabilidad a inyecciones SQL. Para evitar este tipo de problemas, se recomienda el uso de 'PreparedStatement' al trabajar con datos variables. A continuación, se presenta un código que demuestra cómo hacerlo:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConsultarPedidosSeguros {
    Run | Debug
    public static void main(String[] args) {
        int idCliente = 123;  // ID del cliente que se busca
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        // Uso de try-with-resources para gestionar los recursos
        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             PreparedStatement preparacionConsulta = conexion.prepareStatement(sql:"SELECT * FROM pedidos WHERE cliente_id = ?")) {

            // Establecer el valor del parámetro
            preparacionConsulta.setInt(parameterIndex:1, idCliente);

            // Ejecutar la consulta
            ResultSet resultadoSegurosPedidos = preparacionConsulta.executeQuery();

            // Procesar los resultados
            while (resultadoSegurosPedidos.next()) {
                int idPedido = resultadoSegurosPedidos.getInt(columnLabel:"id");
                String producto = resultadoSegurosPedidos.getString(columnLabel:"producto");
                int cantidad = resultadoSegurosPedidos.getInt(columnLabel:"cantidad");
                System.out.println("ID Pedido: " + idPedido + ", Producto: " + producto + ", Cantidad: " + cantidad);
            }

            // Cerrar el ResultSet
            resultadoSegurosPedidos.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Este código mejora la seguridad al parametrizar la consulta, lo que reduce la posibilidad de inyección SQL. Es recomendable utilizar PreparedStatement cuando se manejan entradas de usuario.

Hay métodos adicionales en la clase 'Statement' que son útiles en diversas situaciones. Por ejemplo, **el método `addBatch` permite agrupar múltiples sentencias SQL para ser ejecutadas en conjunto**. Esto puede optimizar el rendimiento al disminuir la cantidad de solicitudes a la base de datos. A continuación, se muestra un ejemplo de uso de `addBatch`:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class InsercionPorLotes {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             Statement sentencia = conexion.createStatement()) {

            // Crear las sentencias de inserción
            String insercion1 = "INSERT INTO clientes (nombre, edad) VALUES ('Laura Gómez', 31)";
            String insercion2 = "INSERT INTO clientes (nombre, edad) VALUES ('Carlos López', 22)";

            // Añadir las inserciones al lote (batch)
            sentencia.addBatch(insercion1);
            sentencia.addBatch(insercion2);

            // Ejecutar el lote de inserciones
            int[] resultados = sentencia.executeBatch();

            // Mostrar el número de filas afectadas por el lote
            System.out.println("Filas afectadas: " + resultados.length);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Este fragmento de código muestra cómo agregar múltiples inserciones a un batch y ejecutarlas de una sola vez. La ejecución por lotes no solo mejora el rendimiento, sino que también simplifica la gestión de transacciones al agrupar cambios relacionados en la base de datos.

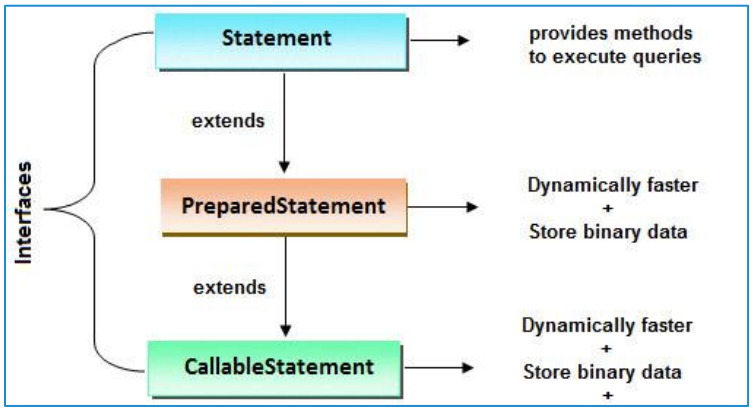
En situaciones que requieren realizar una transacción con múltiples sentencias SQL, se puede manejar 'Autocommit'. **Al desactivar 'Autocommit', las operaciones no se confirmarán hasta que se llame a `commit`**, lo que brinda la opción de revertir cambios si ocurre un error. Un ejemplo de cómo manejar transacciones se presenta a continuación:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.Statement;

public class TransaccionClientesPedidos {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "mi_usuario";
        String contrasena = "mi_contrasena";

        try (Connection conexion = DriverManager.getConnection(url, usuario, contrasena);
             Statement sentencia = conexion.createStatement()) {

            // Desactivar auto-commit para iniciar la transacción
            conexion.setAutoCommit(autoCommit:false);

            try {
                // Actualización de la edad del cliente
                sentencia.executeUpdate(sql:"UPDATE clientes SET edad = 29 WHERE id = 1");

                // Inserción de un nuevo pedido
                sentencia.executeUpdate(sql:"INSERT INTO pedidos (cliente_id, producto) VALUES (1, 'Producto A')");

                // Confirmar la transacción si todo va bien
                conexion.commit();
                System.out.println(x:"Transacción completada con éxito.");

            } catch (SQLException e) {
                // Revertir cambios en caso de error
                conexion.rollback();
                System.err.println(x:"Transacción revertida. Ocurrió un error.");
                e.printStackTrace();
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

Este bloque garantiza que ambas operaciones se realicen en conjunto; si alguna falla, se revierten todas las modificaciones, manteniendo la integridad de los datos.

La clase 'Statement', junto con sus métodos y funcionalidades, permite llevar a cabo operaciones de creación, lectura, actualización y eliminación en bases de datos relacionales de manera efectiva. Si bien sirve como una herramienta básica para el acceso a los datos, es importante considerar la seguridad y el rendimiento. Para aplicaciones más complejas, es recomendable integrar 'PreparedStatement' y utilizar transacciones a medida que se incrementa la complejidad del acceso a los datos.

2.2. CLASE PREPAREDSTATEMENT



La clase 'PreparedStatement' forma parte de la API de Java para acceder a bases de datos y se utiliza para ejecutar consultas SQL precompiladas. Su diseño permite mejorar la seguridad y el rendimiento de las aplicaciones. A continuación, se abordarán las funcionalidades y características relevantes de esta clase, acompañadas de ejemplos prácticos y situaciones de uso.

2.2.1. Creación de un 'PreparedStatement'

Para utilizar 'PreparedStatement', es necesario tener una conexión a la base de datos, que se establece mediante la clase 'Connection', utilizando el método `DriverManager.getConnection()`. Después de establecer la conexión, se puede crear un 'PreparedStatement' a partir de una consulta SQL con el método `prepareStatement()`.

Ejemplo:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConsultarEmpleados {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        // Uso de try-with-resources para manejar recursos automáticamente
        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             PreparedStatement preparedStatement = connection.prepareStatement(sql:"SELECT * FROM empleados WHERE departamento = ?")) {

            // Establecer el valor del parámetro
            preparedStatement.setString(parameterIndex:1, x:"Ventas");

            // Ejecutar la consulta y procesar el resultado
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    int id = resultSet.getInt(columnLabel:"id");
                    String nombre = resultSet.getString(columnLabel:"nombre");
                    String departamento = resultSet.getString(columnLabel:"departamento");
                    System.out.println("ID: " + id + ", Nombre: " + nombre + ", Departamento: " + departamento);
                }
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

En este caso, se configura una consulta para seleccionar todos los registros de la tabla empleados cuyo departamento sea 'Ventas'. ***El signo de interrogación (?) actúa como un marcador de posición que será reemplazado por el valor correspondiente***.

2.2.2. Ejecución de consultas

Para ejecutar una consulta a través de 'PreparedStatement' se utilizan el método `executeQuery()` para consultas SELECT y `executeUpdate()` para las operaciones de inserción, actualización o eliminación.

Para las consultas SELECT, se gestiona el resultado mediante un objeto 'ResultSet', que permite acceder a los resultados devueltos.

Ejemplo de consulta SELECT:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConsultarEmpleados {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             PreparedStatement preparedStatement = connection.prepareStatement(sql:"SELECT * FROM empleados WHERE departamento = ?")) {

            // Establecer el valor del parámetro
            preparedStatement.setString(parameterIndex:1, x:"Ventas");

            // Ejecutar la consulta y procesar el resultado
            try (ResultSet resultSet = preparedStatement.executeQuery()) {
                while (resultSet.next()) {
                    System.out.println("ID: " + resultSet.getInt(columnLabel:"id"));
                    System.out.println("Nombre: " + resultSet.getString(columnLabel:"nombre"));
                    System.out.println("Email: " + resultSet.getString(columnLabel:"email"));
                }
            }

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

El 'ResultSet' permite iterar sobre cada fila devuelta. En este caso, se imprimen el ID, nombre y correo electrónico de cada empleado presente en el departamento de Ventas.

2.2.3. Establecimiento de parámetros

Para configurar los parámetros en una consulta, 'PreparedStatement' ofrece métodos específicos según el tipo de dato. Algunos de los métodos más comunes son `setInt()`, `setString()`, `setDouble()`, entre otros, cada uno asociado a un tipo de dato en particular.

Ejemplo de uso de diferentes tipos de parámetros:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InsertarProducto {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        // Uso de try-with-resources para gestionar recursos automáticamente
        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             PreparedStatement preparedStatement = connection.prepareStatement(sql:"INSERT INTO productos (nombre, precio, cantidad) VALUES (?, ?, ?)")) {

            // Establecer los valores para los parámetros
            preparedStatement.setString(parameterIndex:1, x:"Laptop");
            preparedStatement.setDouble(parameterIndex:2, x:999.99);
            preparedStatement.setInt(parameterIndex:3, x:10);

            // Ejecutar la inserción
            int filasInsertadas = preparedStatement.executeUpdate();

            // Mostrar el número de filas insertadas
            System.out.println("Filas insertadas: " + filasInsertadas);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

En este fragmento, se preparan los valores de nombre, precio y cantidad para insertar un nuevo producto en la base de datos.

2.2.4. Manejo de excepciones y cierre de recursos

El manejo de excepciones es importante al trabajar con bases de datos, ya que pueden producirse errores durante la ejecución de consultas. Normalmente, se utilizan bloques try-catch para capturar excepciones de tipo SQLException. Además, es necesario liberar recursos cerrando objetos 'PreparedStatement' y 'ResultSet' en bloques 'finally' o utilizando la instrucción try-with-resources.

Ejemplo de manejo de excepciones:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class EliminarProducto {
    Run | Debug
    public static void main(String[] args) {
        // Uso de try-with-resources para gestionar los recursos automáticamente
        try (Connection connection = DriverManager.getConnection(url:"jdbc:mysql://localhost:3306/mi_base_de_datos", user:"usuario",
                password:"contraseña");
             PreparedStatement preparedStatement = connection.prepareStatement(sql:"DELETE FROM productos WHERE id = ?")) {

            // Establecer el valor del parámetro (ID del producto a eliminar)
            preparedStatement.setInt(parameterIndex:1, x:5);

            // Ejecutar la eliminación y obtener el número de filas eliminadas
            int filasEliminadas = preparedStatement.executeUpdate();

            // Mostrar el número de filas eliminadas
            System.out.println("Filas eliminadas: " + filasEliminadas);

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

En este caso, tanto la conexión como el 'PreparedStatement' se cerrarán automáticamente al final del bloque try.

2.2.5. Uso de transacciones

'PreparedStatement' es útil en operaciones que requieren asegurar que un conjunto de acciones se ejecute en su totalidad o en ninguna circunstancia. Para trabajar con transacciones, se debe desactivar el modo de 'auto-commit' utilizando `setAutoCommit(false)` antes de ejecutar las acciones y llamar a `commit()` al final de una operación exitosa o `rollback()` en caso de error.

Ejemplo de manejo de transacciones:

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class TransaccionVentas {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        Connection connection = null;
        PreparedStatement insertar = null;
        PreparedStatement actualizar = null;

        try {
            connection = DriverManager.getConnection(url, usuario,
contrasena);
            connection.setAutoCommit(false); // Iniciar la transacción

            // Inserción en ventas
            insertar = connection.prepareStatement("INSERT INTO ventas
(producto_id, cantidad) VALUES (?, ?)");
            insertar.setInt(1, 1);
            insertar.setInt(2, 5);
            insertar.executeUpdate();

            // Actualización de productos
            actualizar = connection.prepareStatement("UPDATE productos SET
cantidad = cantidad - ? WHERE id = ?");
            actualizar.setInt(1, 5);
            actualizar.setInt(2, 1);
```

```
            actualizar.executeUpdate();

            // Confirmar la transacción
            connection.commit();
            System.out.println("Transacción completada con éxito.");

        } catch (SQLException e) {
            if (connection != null) {
                try {
                    // Revertir si ocurre un error
                    connection.rollback();
                    System.err.println("Transacción revertida.");
                } catch (SQLException rollbackEx) {
                    rollbackEx.printStackTrace();
                }
            }
            e.printStackTrace();
        } finally {
            try {
                // Cerrar los recursos de manera adecuada
                if (insertar != null) {
                    insertar.close();
                }
                if (actualizar != null) {
                    actualizar.close();
                }
                if (connection != null) {
                    connection.setAutoCommit(true); // Restablecer el auto-
commit
                    connection.close();
                }
            } catch (SQLException e) {
                e.printStackTrace();
            }
        }
    }
}
```

Aquí, se inserta una venta y se actualiza el stock correspondiente del producto. En caso de que cualquiera de las acciones falle, se lleva a cabo un rollback para mantener la coherencia de la base de datos.

2.2.6. Prevención de inyecciones SQL

Una de las ventajas del uso de 'PreparedStatement' es la reducción del riesgo de inyecciones SQL. A diferencia de la concatenación de cadenas para construir consultas, 'PreparedStatement' se asegura de que los parámetros sean manejados de manera adecuada, evitando que los valores introducidos alteren la ejecución de la consulta.

Ejemplo de comparación de enfoques:

Consulta vulnerable a inyección SQL:

```
String email = "user@example.com"; String sql = "SELECT * FROM usuarios WHERE email = '" + email + "'"; // Inseguro
```

Consulta segura utilizando 'PreparedStatement':

```
String sql = "SELECT * FROM usuarios WHERE email = ?";
PreparedStatement preparedStatement = connection.prepareStatement(sql);
preparedStatement.setString(1, email); // Parámetero seguro, evitando inyecciones SQL

ResultSet resultSet = preparedStatement.executeQuery();

// Procesar el resultado
while (resultSet.next()) {
    String nombre = resultSet.getString("nombre");
    String correo = resultSet.getString("email");
    // Otras operaciones con el resultado
    System.out.println("Nombre: " + nombre + ", Correo: " + correo);
}

// Cerrar el ResultSet y el PreparedStatement cuando termines
resultSet.close();
preparedStatement.close();
```

En el segundo caso, el uso de PreparedStatement previene cualquier intento de manipulación de la consulta SQL mediante la inyección de código.

2.2.7. Consultas de actualización en lote

Las consultas en lote son útiles para optimizar el rendimiento al realizar múltiples acciones de manera eficiente. Preparando una consulta y ejecutándola varias veces con diferentes parámetros, se puede reducir la cantidad de comunicaciones entre la aplicación y la base de datos.

Ejemplo de actualización en lote:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class InsercionEmpleados {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena)) {

            String sql = "INSERT INTO empleados (nombre, departamento) VALUES (?, ?)";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);

            String[][] datos = { {"Carlos", "IT"}, {"Ana", "Marketing"}, {"Luis", "Ventas"} };

            // Añadir las inserciones al lote
            for (String[] empleado : datos) {
                preparedStatement.setString(parameterIndex:1, empleado[0]); // Nombre
                preparedStatement.setString(parameterIndex:2, empleado[1]); // Departamento
                preparedStatement.addBatch(); // Añadir al lote
            }

            // Ejecutar el lote de inserciones
            int[] resultados = preparedStatement.executeBatch();

            // Mostrar el número de filas insertadas
            System.out.println("Filas insertadas: " + resultados.length);

            // Cerrar PreparedStatement
            preparedStatement.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

En este caso, se preparan múltiples inserciones de empleados en una sola operación, lo que mejora la eficiencia del acceso a la base de datos.

2.2.8. Uso avanzado de PreparedStatement

Además de las funcionalidades básicas, 'PreparedStatement' permite realizar consultas más complejas. Por ejemplo, se pueden utilizar operaciones JOIN para recuperar datos de diferentes tablas o emplear subconsultas.

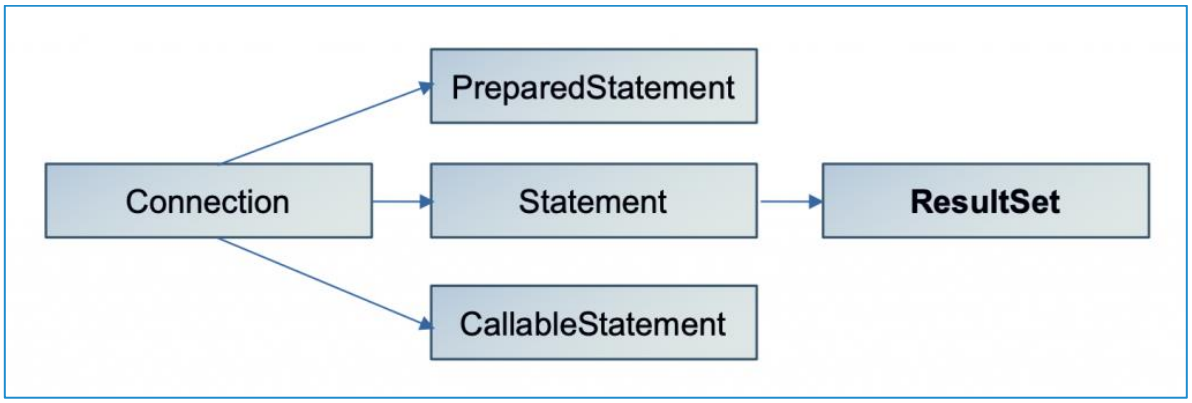
Ejemplo de consulta con JOIN:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;

public class ConsultarEmpleadosPorDepartamento {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena)) {

            // Consulta con JOIN entre empleados y departamentos
            String sql = "SELECT e.nombre, d.nombre AS departamento
                FROM empleados e JOIN departamentos d ON e.departamento_id = d.id WHERE d.nombre = ?";
            PreparedStatement preparedStatement = connection.prepareStatement(sql);

            // Establecer el valor del departamento (en este caso, "IT")
            preparedStatement.setString(1, "IT");

            // Ejecutar la consulta
            ResultSet resultSet = preparedStatement.executeQuery();

            // Procesar el resultado
            while (resultSet.next()) {
                String nombreEmpleado = resultSet.getString("nombre");
                String nombreDepartamento = resultSet.getString("departamento");
                System.out.println("Empleado: " + nombreEmpleado + ", Departamento: " + nombreDepartamento);
            }

            // Cerrar el ResultSet y el PreparedStatement
            resultSet.close();
            preparedStatement.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

En este fragmento, se genera una consulta que une la tabla de empleados con la tabla de departamentos para obtener información sobre los empleados que pertenecen al departamento de IT.

El uso de 'PreparedStatement' es amplio y versátil, lo que permite a los desarrolladores construir aplicaciones robustas que interactúan con bases de datos de manera segura y eficiente. Al aplicar esta clase, se pueden gestionar operaciones complejas con facilidad, manteniendo la integridad y mejorando el rendimiento de las aplicaciones.

2.3. CLASE RESULTSET

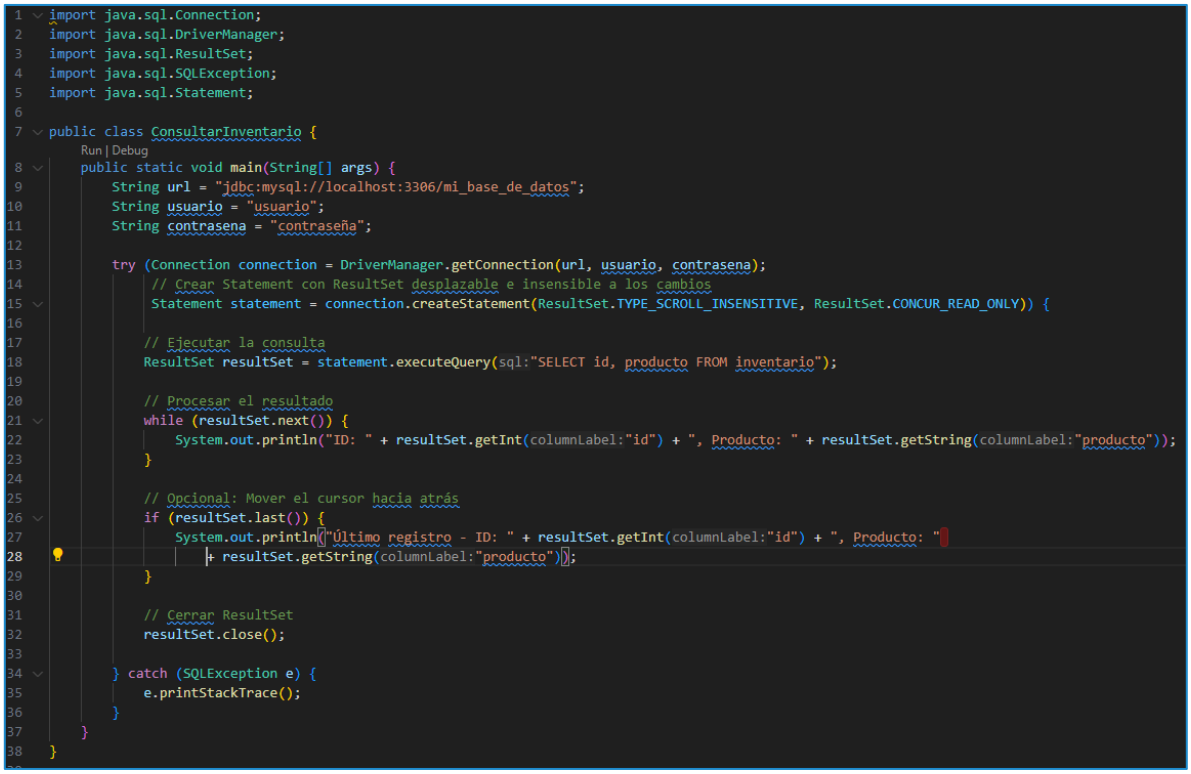


La clase 'ResultSet' en Java es una herramienta que permite a las aplicaciones acceder a los resultados de consultas SQL. Se utiliza con las interfaces de JDBC, facilitando la manipulación y recuperación de datos de bases de datos relacionales. Cada objeto 'ResultSet' está diseñado para contener y gestionar filas de datos devueltos por una sentencia SQL, funcionando como una tabla temporal de información procesada.

2.3.1. Tipos de ResultSet

Existen diferentes tipos de ResultSet, destacando el tipo `TYPE_FORWARD_ONLY`, que permite la navegación únicamente en una dirección, es decir, hacia adelante. En contraste, `TYPE_SCROLL_INSENSITIVE` posibilita moverse tanto hacia adelante como hacia atrás, y `TYPE_SCROLL_SENSITIVE` incluye los cambios realizados por otras transacciones en el ResultSet. Esta característica resulta útil en aplicaciones que requieren distintos niveles de acceso a la información.

Por ejemplo, un sistema de gestión de inventarios podría emplear un ResultSet de tipo `TYPE_SCROLL_INSENSITIVE` para permitir que los administradores naveguen por una lista de productos y realicen modificaciones según sea necesario:



```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarInventario {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             // Crear Statement con ResultSet desplazable e insensible a los cambios
             Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY)) {

            // Ejecutar la consulta
            ResultSet resultSet = statement.executeQuery(sql:"SELECT id, producto FROM inventario");

            // Procesar el resultado
            while (resultSet.next()) {
                System.out.println("ID: " + resultSet.getInt(columnLabel:"id") + ", Producto: " + resultSet.getString(columnLabel:"producto"));
            }

            // Opcional: Mover el cursor hacia atrás
            if (resultSet.last()) {
                System.out.println("Último registro - ID: " + resultSet.getInt(columnLabel:"id") + ", Producto: "
                    + resultSet.getString(columnLabel:"producto"));
            }

            // Cerrar ResultSet
            resultSet.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.3.1.1. Navegación por el ResultSet

La clase ‘ResultSe’t ofrece múltiples métodos de navegación. Además de `next()`, que desplaza el cursor a la siguiente fila, se pueden utilizar `previous()`, `first()`, `last()`, y `absolute(int row)`. *Para un sistema de reservas de hoteles, por ejemplo, sería conveniente mostrar todos los clientes y permitir al usuario desplazarse hacia delante y hacia atrás por los resultados*.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarReservas {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             // Crear un Statement con un ResultSet desplazable e insensible a los cambios
             Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_READ_ONLY)) {

            // Ejecutar la consulta
            ResultSet resultSet = statement.executeQuery(sql:"SELECT cliente, reserva FROM reservas");

            // Mover el cursor a la última fila
            if (resultSet.last()) {
                System.out.println("Total de reservas: " + resultSet.getRow());
            }

            // Mover el cursor a la primera fila
            if (resultSet.absolute(row:1)) {
                System.out.println("Primera reserva - Cliente: " + resultSet.getString(columnLabel:"cliente"));
            }

            // Cerrar ResultSet
            resultSet.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.3.2. Extracción de Datos

Los métodos de la clase 'ResultSet' permiten acceder a los datos de diferentes maneras, dependiendo del tipo de datos. Cada columna del 'ResultSet' se puede recuperar mediante métodos de acceso, como `getInt()`, `getString()`, `getDouble()`, entre otros. Al trabajar en una aplicación de gestión de ventas, la extracción de datos se realizaría con base en los campos seleccionados en la consulta.

Por ejemplo, si se tiene una tabla `ventas`, se puede consultar información de cada venta:

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarVentas {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             Statement statement = connection.createStatement()) {

            // Ejecutar la consulta
            ResultSet resultSet = statement.executeQuery("SELECT id, cliente, total FROM ventas");

            // Procesar el resultado
            while (resultSet.next()) {
                int id = resultSet.getInt("id");
                String cliente = resultSet.getString("cliente");
                double total = resultSet.getDouble("total");

                System.out.println("ID Venta: " + id + ", Cliente: " + cliente + ", Total: " + total);
            }

            // Cerrar ResultSet
            resultSet.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.3.3. Actualización de Datos

Los 'ResultSet' que permiten la actualización se crean usando `ResultSet.CONCUR_UPDATABLE`. Esto facilita la modificación directa de los valores que representa el 'ResultSet'. En un sistema bancario, podría ser común actualizar el saldo de una cuenta tras realizar una transacción.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ActualizarSaldo {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             Statement statement = connection.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE, ResultSet.CONCUR_UPDATABLE)) {

            // Ejecutar la consulta y obtener el ResultSet actualizable
            ResultSet resultSet = statement.executeQuery(sql:"SELECT id, saldo FROM cuentas WHERE cliente_id = 1");

            // Recorrer los resultados y actualizar el saldo
            while (resultSet.next()) {
                double saldoActual = resultSet.getDouble(columnLabel:"saldo");
                double nuevoSaldo = saldoActual - 100; // Retirar 100 unidades

                // Actualizar el saldo
                resultSet.updateDouble(columnLabel:"saldo", nuevoSaldo);
                resultSet.updateRow(); // Aplicar la actualización a la base de datos

                System.out.println("ID Cuenta: " + resultSet.getInt(columnLabel:"id") + ", Nuevo Saldo: " + nuevoSaldo);
            }

            // Cerrar el ResultSet
            resultSet.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.3.4. Metadatos de ResultSet

La clase 'ResultSetMetaData' proporciona información sobre la estructura de las columnas del 'ResultSet'. Esto es ventajoso para conocer la cantidad de columnas y sus nombres de manera **dinámica**, permitiendo que las aplicaciones se adapten a cambios en la estructura de la base de datos.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.ResultSetMetaData;
import java.sql.SQLException;
import java.sql.Statement;

public class MostrarMetaData {
    Run | Debug
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             Statement statement = connection.createStatement()) {

            // Ejecutar la consulta y obtener el ResultSet
            ResultSet resultSet = statement.executeQuery(sql:"SELECT * FROM cuentas");

            // Obtener los metadatos del ResultSet
            ResultSetMetaData metaData = resultSet.getMetaData();
            int columnCount = metaData.getColumnCount();

            // Mostrar la información de cada columna
            for (int i = 1; i <= columnCount; i++) {
                String columnName = metaData.getColumnName(i);
                String columnType = metaData.getColumnTypeName(i);
                System.out.println("Columna: " + columnName + ", Tipo: " + columnType);
            }

            // Cerrar el ResultSet
            resultSet.close();

        } catch (SQLException e) {
            e.printStackTrace();
        }
    }
}
```

2.3.5. Manejo de Errores

Al trabajar con 'ResultSet', es importante implementar un manejo adecuado de errores, utilizando bloques 'try-catch' para interceptar excepciones como `SQLException`, que pueden surgir de operaciones erróneas o problemas en la base de datos.

```java
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;

public class ConsultarProductos {
    public static void main(String[] args) {
        String url = "jdbc:mysql://localhost:3306/mi_base_de_datos";
        String usuario = "usuario";
        String contrasena = "contraseña";

        try (Connection connection = DriverManager.getConnection(url, usuario, contrasena);
             Statement statement = connection.createStatement();
             ResultSet resultSet = statement.executeQuery(sql:"SELECT * FROM productos")) {

            // Procesar los resultados del ResultSet
            while (resultSet.next()) {
                int id = resultSet.getInt(columnLabel:"id");
                String nombre = resultSet.getString(columnLabel:"nombre");
                double precio = resultSet.getDouble(columnLabel:"precio");

                System.out.println("ID: " + id + ", Producto: " + nombre + ", Precio: " + precio);
            }

        } catch (SQLException e) {
            e.printStackTrace(); // Manejar el error de SQL
        }
    }
}
```

2.3.6. Aislamiento de Transacciones

El aislamiento de transacciones es un concepto en la gestión de 'ResultSet' que define cómo se comportan las transacciones concurrentes. La configuración de los niveles de aislamiento permite controlar cómo los cambios realizados por una transacción son visibles para otras.

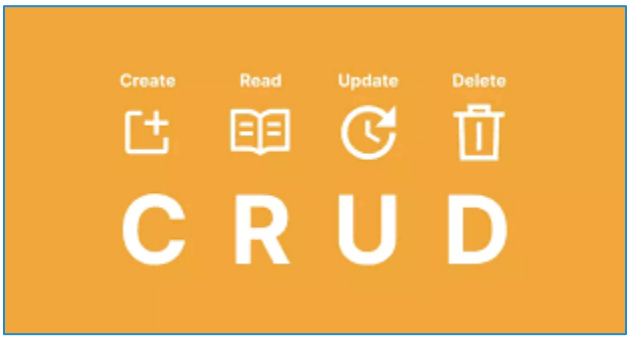
Niveles como `READ_COMMITTED` evitan que transacciones lean datos no confirmados, mientras que `SERIALIZABLE` garantiza un aislamiento más estricto, aunque podría impactar el rendimiento.

connection.setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED);

El uso de 'ResultSet' en aplicaciones complejas, como sistemas CRM o ERP, ayuda a mantener la integridad de los datos y su coherencia, incluso cuando hay interacciones simultáneas.

La clase 'ResultSet' se posiciona como un componente central para las aplicaciones que interactúan con bases de datos relacionales, ofreciendo herramientas para la navegación, modificación y manejo de datos en tiempo real.

3. OPERACIONES CRUD



Las operaciones CRUD son un conjunto de funciones básicas que permiten gestionar datos dentro de una base de datos relacional. Estas acciones son necesarias para cualquier aplicación que requiera interacción con datos almacenados, ya que comprenden todas las tareas para manipular información de manera efectiva.

Estas operaciones suelen implementarse a través de sentencias SQL (Structured Query Language), que proporcionan la sintaxis requerida para llevar a cabo cada acción en la base de datos. La aplicación adecuada de las operaciones CRUD es un principio básico en el diseño de bases de datos y en el desarrollo de aplicaciones que interactúan con ellas. Comprender y utilizar estas operaciones de forma eficaz es importante para construir aplicaciones eficientes y bien organizadas que respondan a las necesidades de los usuarios. Adicionalmente, implementar mecanismos de seguridad y validación relacionados con estas acciones es relevante para proteger la integridad de los datos y prevenir accesos no autorizados.

3.1. CREATE

La operación "Create" en el acceso a bases de datos relacionales se refiere a la acción de insertar nuevos registros en una tabla. **Este proceso es fundamental para la gestión de datos y se lleva a cabo mediante la instrucción SQL `INSERT INTO`**. Esta acción permite a los desarrolladores agregar datos importantes para el funcionamiento de aplicaciones, ya que habilita la creación de entidades en el sistema de gestión de bases de datos.

La estructura básica de un comando `INSERT INTO` es:

INSERT INTO nombre_tabla (columna1, columna2, columna3, ...) VALUES (valor1, valor2, valor3, ...);

Donde `nombre_tabla` denota la tabla específica donde se realizará la inserción. Las columnas especificadas determinan qué datos se agregarán en cada registro. A continuación, se explorará en detalle cada uno de los componentes de esta operación, junto con ejemplos y situaciones prácticas que ilustran su utilización.

3.1.1. Ejemplo de Inserción Simple

Considerando una base de datos de una biblioteca que contiene una tabla llamada `libros` con las columnas: `id`, `titulo`, `autor`, `anio_publicacion` y `genero`. *Para agregar un nuevo libro, se ejecutaría la siguiente consulta*:

INSERT INTO libros (titulo, autor, anio_publicacion, genero) VALUES ('1984', 'George Orwell', 1949, 'Distopía');

Este comando inserta un nuevo registro en la tabla `libros`, añadiendo la obra de George Orwell junto con su año de publicación y género. La comprensión de cómo se estructuran las consultas SQL influye en el desarrollo eficaz, ya que afecta la cantidad y calidad de los datos almacenados.

3.1.2. Inserción de Múltiples Registros

Además de la inserción individual, es posible realizar inserciones de manera masiva utilizando la misma instrucción `INSERT`, lo que mejora la eficiencia en la gestión de bases de datos al reducir el número de transacciones necesarias. Por ejemplo, si se desean agregar varios libros de una sola vez:

INSERT INTO libros (titulo, autor, anio_publicacion, genero) VALUES ('Cien Años de Soledad', 'Gabriel García Márquez', 1967, 'Realismo mágico'), ('El Quijote', 'Miguel de Cervantes', 1605, 'Novela'), ('Fahrenheit 451', 'Ray Bradbury', 1953, 'Ciencia ficción');

Con este comando se están insertando tres registros en una única operación, lo que resulta más eficiente, especialmente cuando se necesitan agregar grandes cantidades de datos.

3.1.3. Validación de Datos

Es importante llevar a cabo la validación de los datos antes de realizar una operación de creación. Las restricciones en las tablas, como claves primarias y restricciones de unicidad, deben ser tomadas en cuenta. Al intentar insertar un registro que infringe estas reglas, se genera un error.

Por ejemplo, si la columna `id` de la tabla `libros` es una clave primaria que no permite valores nulos y se intenta ejecutar lo siguiente:

INSERT INTO libros (id, titulo, autor, anio_publicacion, genero) VALUES (1, 'El Principito', 'Antoine de Saint-Exupéry', 1943, 'Fábula');

Si después se intenta agregar otro libro con el mismo `id`:

INSERT INTO libros (id, titulo, autor, anio_publicacion, genero) VALUES (1, 'Matar a un Ruiseñor', 'Harper Lee', 1960, 'Ficción');

El sistema generará un error indicando que ya existe una entrada con el `id` 1.

3.1.4. Manejo de Errores

Implementar un manejo de errores es importante al realizar operaciones de inserción. Se pueden emplear transacciones para asegurar la integridad de los datos. Si un comando `INSERT` falla y no se estructura correctamente la transacción, la base de datos podría quedar en un estado inconsistente:

```sql
START TRANSACTION;

INSERT INTO libros (titulo, autor, anio_publicacion, genero)
VALUES ('El Hobbit', 'J.R.R. Tolkien', 1937, 'Fantasía');

INSERT INTO libros (titulo, autor, anio_publicacion, genero)
VALUES ('La Odisea', 'Homero', 800, 'Épico');

COMMIT;

```

Si ambos inserts se ejecutan sin problemas, se confirma la transacción. En caso de que un error ocurra durante cualquiera de los inserts, se puede ejecutar un `ROLLBACK` para revertir las inserciones, manteniendo la base de datos en un estado seguro.

3.1.5. Uso de ORM y abstracción de la lógica de datos

En muchas aplicaciones modernas, se utilizan ORM (Object Relational Mapping), lo que permite interactuar con la base de datos a través de un modelo orientado a objetos (como Java). Por ejemplo, en un entorno de aplicación .NET con `Entity Framework`, un desarrollador **podría agregar un nuevo libro utilizando código C# en lugar de escribir SQL directamente**:

```csharp
using (var context = new BibliotecaContext())
{
    // Crear un nuevo objeto 'Libro'
    var nuevoLibro = new Libro
    {
        Titulo = "Cien Años de Soledad",
        Autor = "Gabriel García Márquez",
        AnioPublicacion = 1967,
        Genero = "Realismo mágico"
    };

    // Agregar el libro al contexto
    context.Libros.Add(nuevoLibro);

    // Guardar los cambios en la base de datos
    context.SaveChanges();
}
```

Este enfoque simplifica el manejo de datos, además de encargarse de generar automáticamente las instrucciones SQL necesarias para realizar la inserción en la base de datos. La separación de la lógica de negocio y de acceso a datos facilita el mantenimiento y la escalabilidad de las aplicaciones.

3.1.6. Inserciones disponibles a través de API

Otro enfoque contemporáneo implica el uso de APIs RESTful para realizar acciones de creación. Por ejemplo, en una aplicación web, un cliente puede realizar una solicitud POST para agregar un nuevo libro a la base de datos. La solicitud podría incluir los datos en formato JSON:

```c#
//c#
using Microsoft.AspNetCore.Mvc;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace BibliotecaAPI.Controllers
{
    [ApiController]
    [Route("api/[controller]")]
    public class LibrosController : ControllerBase
    {
        private readonly BibliotecaContext _context;

        public LibrosController(BibliotecaContext context)
        {
            _context = context;
        }

        // POST: api/libros
        [HttpPost]
        public async Task<ActionResult<Libro>> PostLibro(Libro nuevoLibro)
        {
            // Agregar el nuevo libro al contexto
            _context.Libros.Add(nuevoLibro);

            // Guardar cambios en la base de datos
            await _context.SaveChangesAsync();

            // Retornar el libro recién creado y la URI del recurso
            return CreatedAtAction(nameof(GetLibro), new { id = nuevoLibro.Id }, nuevoLibro);
        }

        // Ejemplo de un método para obtener un libro por ID (referencia en CreatedAtAction)
        [HttpGet("{id}")]
        public async Task<ActionResult<Libro>> GetLibro(int id)
        {
            var libro = await _context.Libros.FindAsync(id);
            if (libro == null)
            {
                return NotFound();
            }
            return libro;
        }
    }
}
```

El servidor recibiría esta petición y el código asociado realizaría la inserción llamando a la lógica de acceso a datos correspondiente:

```
[HttpPost]
public IActionResult CrearLibro([FromBody] Libro libro)
{
    // Verificar si el modelo es válido según las reglas de validación
    if (ModelState.IsValid)
    {
        // Agregar el libro al repositorio (o base de datos)
        _repositorio.AgregarLibro(libro);

        // Retornar el recurso creado con su ubicación y el objeto libro
        return CreatedAtAction(nameof(ObtenerLibro), new { id = libro.Id }, libro);
    }

    // Si el modelo no es válido, retornar una respuesta 400 (Bad Request)
    return BadRequest(ModelState); // Devuelve también detalles del error de validación
}
```

Este método asegura que los datos se procesen correctamente y se tomen en cuenta las validaciones requeridas.

3.1.7. Inserción de datos con carga masiva desde archivos

La carga masiva de datos desde archivos es otra práctica común en la operación "Create". En ciertos escenarios, puede ser necesario importar datos de un archivo CSV o Excel. Para las bases de datos que brindan dicha funcionalidad, **se emplean comandos específicos como `LOAD DATA INFILE` o `COPY`**, dependiendo del sistema de gestión de bases de datos utilizado.

En MySQL, una instrucción para cargar datos podría verse así:

```
LOAD DATA INFILE 'ruta/del/archivo.csv'
INTO TABLE libros
FIELDS TERMINATED BY ','
ENCLOSED BY '"'
LINES TERMINATED BY '\n'
IGNORE 1 ROWS;
```

Este comando permite la importación directa de registros de un archivo, lo cual es eficiente para llenar tablas con grandes volúmenes de datos.

3.1.8. Consideraciones de seguridad

Finalmente, al manejar la operación de creación de registros, garantizar la seguridad del acceso a la base de datos es importante. Es recomendable implementar medidas que prevengan ataques como la inyección de SQL. **Utilizar consultas parametrizadas mitigará significativamente este riesgo. En C#, esto se logra a través de `SqlCommand`**:

```
using (var command = new SqlCommand("INSERT INTO libros (titulo, autor, anio_publicacion, genero) VALUES (@titulo, @autor, @anio, @genero)",
    connection))
{
    // Agregar los parámetros de forma segura, evitando la inyección SQL
    command.Parameters.AddWithValue("@titulo", libro.Titulo);
    command.Parameters.AddWithValue("@autor", libro.Autor);
    command.Parameters.AddWithValue("@anio", libro.AnioPublicacion);
    command.Parameters.AddWithValue("@genero", libro.Genero);

    // Ejecutar el comando
    command.ExecuteNonQuery();
}
```

Esto asegura que los datos sean tratados de manera segura, evitando la ejecución de código malicioso que podría comprometer la base de datos.

3.2. READ

La operación "Read" en el acceso a bases de datos relacionales permite realizar consultas y recuperar información desde las tablas. El uso adecuado de SQL (Structured Query Language) es esencial para llevar a cabo estas acciones de manera eficiente. A continuación, se analizan diferentes aspectos de la operación de lectura.

La consulta SELECT es la instrucción central para ejecutar lecturas. Su estructura básica implica definir las columnas que se desean seleccionar, la tabla de origen y las condiciones que filtran los datos. Por ejemplo, si una tabla de empleados contiene las columnas "id", "nombre", "departamento" y "salario", para recuperar el nombre y el salario de todos los empleados en el departamento de ventas, se utilizaría:

SELECT nombre, salario FROM empleados WHERE departamento = 'Ventas';

Esto devolvería una lista con los nombres y salarios de los empleados que trabajan en el departamento especificado.

Además de seleccionar columnas individuales, se puede utilizar el asterisco () para seleccionar todas las columnas de una tabla. La consulta para obtener información completa de la tabla de empleados sería:

SELECT * FROM empleados;

Sin embargo, se recomienda usar esta opción con precaución, ya que extraer todos los datos puede resultar en un uso innecesario de recursos, especialmente en tablas con un gran número de columnas o registros.

La cláusula WHERE permite filtrar los registros según condiciones específicas. Las condiciones pueden incluir operadores de comparación (como =, <>, >, <) y operadores lógicos (AND, OR, NOT). *Por ejemplo, para recuperar empleados cuyo salario esté por encima de 30000 y que pertenezcan al departamento de marketing, la consulta sería*:

SELECT nombre, salario FROM empleados WHERE salario > 30000 AND departamento = 'Marketing';

Este tipo de filtrado es útil en aplicaciones que muestran información relevante según criterios específicos, mejorando la experiencia del usuario.

ORDER BY se emplea para organizar los resultados de una consulta. Por ejemplo, si se desea recuperar los nombres de empleados ordenados por salario de manera descendente, la consulta sería:

SELECT nombre, salario FROM empleados ORDER BY salario DESC;

Esto resulta útil en aplicaciones donde se requiere presentar datos de forma jerárquica, como listas de productos organizados por precio en una tienda en línea.

El uso de JOIN es importante cuando se necesita consultar datos de múltiples tablas relacionadas. Las combinaciones pueden ser de varios tipos: INNER JOIN, LEFT JOIN, RIGHT JOIN y FULL JOIN. Supongamos que se tienen dos tablas: "productos" y "categorias", donde "productos" contiene un "categoria_id" que se relaciona con el "id" de "categorias". Para obtener una lista de productos junto con sus categorías, se podría utilizar INNER JOIN:

SELECT productos.nombre, categorias.nombre FROM productos INNER JOIN categorias ON productos.categoria_id = categorias.id;

Esto permite obtener cada producto junto a su categoría asociada, lo cual es relevante para aplicaciones que requieren presentar información contextual.

Las funciones agregadas en SQL son herramientas que permiten realizar cálculos sobre grupos de registros. Si se desea obtener el salario promedio de los empleados en una empresa, la consulta correspondiente sería:

SELECT AVG(salario) AS salario_promedio FROM empleados;

Este tipo de consulta es útil para generar informes y análisis en aplicaciones de gestión empresarial.

La cláusula GROUP BY se utiliza para agrupar resultados basándose en una o más columnas. Por ejemplo, si se desea contar el número de empleados por departamento, la consulta se estructuraría así:

SELECT departamento, COUNT(*) AS total_empleados FROM empleados GROUP BY departamento;

Este método permite realizar análisis demográficos y detectar tendencias dentro de la organización.

Las transacciones son un concepto importante en las operaciones de lectura. **Una transacción es una serie de operaciones que se ejecutan como una única unidad lógica de trabajo, garantizando la atomicidad**. Por ejemplo, al leer datos en un entorno multiusuario, es necesario que la información que un usuario está visualizando no se vea alterada por modificaciones realizadas por

otros usuarios simultáneamente. Para manejar esta situación, se pueden aplicar niveles de aislamiento, como:

- **'Read Uncommitted'**: Permite leer datos que han sido modificados, pero no confirmados por otras transacciones.
- **'Read Committed'**: Asegura que solo se puedan leer datos que han sido confirmados.
- **'Repeatable Read'**: Previene que los datos leídos cambien durante la misma transacción.
- **'Serializable'**: El nivel más alto que garantiza consistencia en los resultados y evita interferencias de otras transacciones.

En un sistema bancario, por ejemplo, al consultar el saldo de una cuenta, es importante que esta información refleje cualquier cambio que se haya confirmado hasta ese momento.

Las vistas son herramientas que simplifican la recuperación de datos. **Una vista es una consulta almacenada que permite presentar datos de manera cohesiva sin necesidad de realizar la misma consulta repetidamente**. *Por ejemplo, se podría crear una vista que muestre todos los productos junto con sus categorías de la siguiente manera*:

```sql
CREATE VIEW vista_productos_categorias AS
SELECT
    productos.nombre AS nombre_producto,
    categorias.nombre AS nombre_categoria
FROM
    productos
JOIN
    categorias ON productos.categoria_id = categorias.id;

```

Luego, se puede acceder a la vista como si fuera una tabla:

SELECT * FROM vista_productos_categorias;

Esto mejora la organización de las consultas y puede servir para limitar la exposición de datos, ofreciendo un nivel de seguridad adicional.

La seguridad en las operaciones de lectura es un aspecto importante. **'SQL Injection' es una vulnerabilidad en la que un atacante puede inyectar código malicioso a través de entradas de usuario, comprometiendo la integridad de la base de datos. Para protegerse contra estos riesgos, se deben aplicar buenas prácticas como declaraciones preparadas**:

```
-- Preparar la consulta
PREPARE stmt FROM 'SELECT * FROM empleados WHERE nombre = ?';

-- Asignar valor al parámetro
SET @nombre = 'Juan';

-- Ejecutar la consulta con el parámetro
EXECUTE stmt USING @nombre;

```

Separar los datos de la lógica de la consulta ayuda a minimizar el riesgo de inyección.

El uso de índices también influye en el rendimiento de las lecturas. Un índice es una estructura que mejora la velocidad de las operaciones de búsqueda en una tabla. Por ejemplo, si en la tabla de "empleados" se crea un índice en la columna "nombre", las consultas que filtran o buscan por nombre se ejecutarán más rápidamente en comparación con la ausencia de dicho índice.

La selección adecuada de índices y el diseño del esquema de base de datos son aspectos que contribuyen a mantener un rendimiento óptimo a medida que las bases de datos aumentan en tamaño y complejidad.

La operación de lectura en bases de datos relacionales implica no solo la obtención de datos, sino que también abarca una serie de prácticas y técnicas que aseguran el acceso eficiente, seguro y organizado a la información. Implementar adecuadamente estas operaciones es determinante para el funcionamiento eficaz de aplicaciones que dependen de bases de datos.

3.3. UPDATE

La operación "Update" en bases de datos relacionales permite modificar datos que ya existen. Es una acción que se ejecuta sobre registros almacenados en la base de datos. La sintaxis SQL para llevar a cabo una actualización es clara y estructurada. Esta estructura incluye el nombre de la tabla a la que se le hará la modificación, los valores que se asignarán a las columnas específicas y una condición que delimita los registros que se verán afectados por la acción.

La sintaxis básica para un comando "Update" es la siguiente:

UPDATE nombre_tabla SET columna1 = valor1, columna2 = valor2, ... WHERE condicion;

Si no se incluye una cláusula "WHERE", todos los registros de la tabla serán modificados, lo que puede resultar en cambios no deseados.

Un ejemplo práctico puede ser una tabla "Clientes" que contenga datos sobre los clientes en un sistema de gestión. Si un cliente desea cambiar su número de teléfono, la consulta "Update" se vería así:

```
UPDATE Clientes SET telefono = '123456789' WHERE ID = 42;
```

En este caso, solo se actualizará el número de teléfono del cliente cuyo ID es 42. Este tipo de consultas es común en sistemas de gestión de relaciones con los clientes, donde los datos deben ser precisos y actualizados.

En un escenario diferente, se podría necesitar actualizar la información de estado laboral de un empleado que ha sido promovido. La consulta de actualización se formularía de este modo:

```
UPDATE Empleados SET estado_empleo = 'Promovido' WHERE ID = 10;
```

Mediante este comando, el estado de empleo del trabajador en la tabla se modifica a "Promovido" para el registro donde el ID es 10. La instrucción es efectiva gracias a la cláusula "WHERE" que evita modificar registros adicionales.

La operación de actualización tiene la capacidad de influir en varias filas de la tabla. Esto es útil en situaciones como cuando una empresa decide aumentar el salario de todos sus empleados en un porcentaje determinado. La actualización se realizaría así:

```
UPDATE Empleados SET sueldo = sueldo * 1.05;
```

En este caso, **todos** los registros en la tabla "Empleados" se verían afectados por un aumento del 5% en su sueldo, dado que no se incluye una cláusula "WHERE". Este tipo de acciones puede ser común en ciertas épocas del año, como durante revisiones salariales.

El concepto de transacciones es importante en las operaciones de actualización, especialmente en ambientes donde múltiples usuarios pueden interactuar con los mismos datos simultáneamente. Esto puede crear problemas de consistencia. Por eso, se recomienda usar mecanismos de bloqueo para asegurar que una operación de modificación finalice antes que otra inicie.

En un sistema con diferentes usuarios, si uno de ellos intenta actualizar un registro mientras otro también lo está haciendo, puede ocurrir que un cambio sobrescriba al otro, lo que resulta en pérdida de información. Implementar transacciones ayuda a que estas modificaciones se realicen de manera coherente. Por lo general, se puede aplicar un enfoque de "Control de Concurrencia Optimista", que verifica el estado de los datos antes de confirmar cualquier operación.

Además, se pueden utilizar triggers que se activen al realizar un "Update". Por ejemplo, se podría crear un trigger que grabe automáticamente cualquier cambio en una tabla de "Auditoria". Esto es útil para mantener un registro de modificaciones como el siguiente:

```
CREATE TRIGGER ActualizarAuditoria
AFTER UPDATE ON Empleados
FOR EACH ROW
BEGIN
    INSERT INTO Auditoria (ID_Empleado, vieja_informacion, nueva_informacion, fecha_cambio)
    VALUES (OLD.ID, OLD.sueldo, NEW.sueldo, NOW());
END;
```

Este trigger se activará cada vez que un registro en la tabla "Empleados" sea actualizado, almacenando el ID del empleado y la información anterior y actual, junto con la fecha del cambio.

La validación de datos es un aspecto que debe considerarse al realizar una operación de actualización. Antes de ejecutar una consulta "Update", es recomendado que se valide tanto la nueva información como las condiciones de aplicación. Esto ayuda a garantizar que los cambios no infrinjan reglas de negocio o restricciones. Por ejemplo, al actualizar el salario de un empleado, podría ser necesario confirmar que el nuevo salario cumpla con el mínimo establecido.

Un caso que ilustra este procedimiento sería:

```
UPDATE Empleados
SET sueldo = 2800
WHERE ID = 7
AND 2800 > (SELECT sueldo_minimo FROM Politicas WHERE ID = 1);
```

En este ejemplo, la actualización del sueldo del empleado con ID 7 se llevará a cabo solamente si el nuevo sueldo de 2800 es superior al sueldo mínimo establecido por las políticas de la empresa.

La gestión de errores es otro aspecto relevante al implementar actualizaciones. Si se intenta modificar un registro y se presentan restricciones de integridad, como claves foráneas o chequeos de unicidad, es necesario manejar adecuadamente esas excepciones. **Las aplicaciones deben ser programadas para interceptar errores, informar al usuario y ofrecer alternativas para resolver los problemas**.

Un enfoque adicional incluye el registro y la auditoría de cambios como parte de la operación "Update". **Las bases de datos pueden implementar estrategias para rastrear no solo quién realiza el cambio, sino también cuándo y qué información se modifica. Este procedimiento es común en aplicaciones que requieren un seguimiento detallado para cumplir con normativas**.

La operación de actualización en bases de datos abarca más que simplemente modificar valores en una tabla. **Involucra consideraciones sobre la coherencia de los datos, la gestión de concurrencia, la validación y el control de errores**, creando un sistema integral que permite el manejo adecuado de la información. Cada intento de actualización genera varias prácticas que aseguran que los datos

gestionados sean relevantes y precisos. Esto es clave para el funcionamiento efectivo de aplicaciones que interactúan con bases de datos relacionales.

3.4. DELETE

La operación "Delete" en el acceso a bases de datos relacionales **permite eliminar registros de las tablas**, lo que ayuda a mantener la integridad y la relevancia de la información. La instrucción DELETE se basa en la sintaxis SQL, que es el lenguaje estándar utilizado para gestionar bases de datos.

La sintaxis básica de la instrucción DELETE es:

DELETE FROM nombre_tabla WHERE condición;

En esta sintaxis, `nombre_tabla` representa el nombre de la tabla de la que se desea eliminar información, y `condición` especifica cuáles registros deben ser eliminados. **La cláusula WHERE es muy importante, ya que su ausencia resultará en la eliminación de todos los registros en la tabla**.

Ejemplo de eliminación de un registro específico

Considerando una tabla denominada `usuarios` que contiene datos de usuarios registrados:

id_usuario	nombre	correo
1	Ana	ana@example.com
2	Luis	luis@example.com
3	Carla	carla@example.com

Si se desea eliminar al usuario con id 1 (Ana), la instrucción sería:

DELETE FROM usuarios WHERE id_usuario = 1;

Después de ejecutar esta consulta, la tabla se verá así:

id_usuario	nombre	correo
2	Luis	luis@example.com
3	Carla	carla@example.com

Eliminación de múltiples registros

La instrucción DELETE también permite eliminar varios registros a la vez, lo cual es útil en situaciones donde se necesita limpiar los datos de manera eficiente. *Por ejemplo, en una tabla `artículos` que contiene productos de una tienda*:

id_articulo	nombre	categoria
1	Televisor	Electrónica
2	Camiseta	Vestimenta
3	Laptop	Electrónica
4	Pantalones	Vestimenta

Si se desea eliminar todos los artículos que pertenecen a la categoría "Vestimenta", se puede ejecutar la siguiente consulta:

DELETE FROM articulos WHERE categoria = 'Vestimenta';

Tras esta operación, la tabla se mantendrá de la siguiente manera:

id_articulo	nombre	categoria
1	Televisor	Electrónica
3	Laptop	Electrónica

Ejemplos en aplicaciones prácticas

La operación DELETE se usa en muchos escenarios en aplicaciones empresariales. *Por ejemplo, en un sistema de gestión de recursos humanos, puede ser necesario eliminar registros de empleados que han dejado la organización. Por ejemplo, en una tabla `empleados`*:

id_empleado	nombre	departamento
1	Javier	IT
2	Sofia	RRHH
3	Marcos	Diseño

Al dar de baja a Javier, la estructura de la tabla tras ejecutar la consulta:

DELETE FROM empleados WHERE id_empleado = 1;

resultará en:

id_empleado	nombre	departamento
2	Sofia	RRHH
3	Marcos	Diseño

Esto mantiene la aplicación actualizada sin información redundante o irrelevante sobre empleados ya no activos.

3.4.1. Integridad referencial

La integridad referencial es un concepto importante al realizar operaciones DELETE. **Este concepto asegura que las relaciones entre tablas se mantengan de manera coherente**. Por ejemplo, si existe una tabla `pedidos` que registra pedidos realizados por los empleados, eliminar un registro de la tabla `empleados` podría causar inconsistencias en los datos.

Considerando la tabla `pedidos` que tiene la siguiente estructura:

id_pedido	id_empleado	producto
101	1	Laptop
102	2	Smartphone

Si se desea eliminar al empleado con id 1, que tiene pedidos relacionados, la consulta:

DELETE FROM empleados WHERE id_empleado = 1;

debería llevarse a cabo con precaución. **Si se eliminara este registro y se mantuvieran los pedidos asociados, podría resultar en información huérfana**.

Para evitar problemas de integridad, se pueden establecer claves foráneas junto con acciones de eliminación en cascada. Esto permitirá que todos los registros en la tabla de pedidos se eliminen automáticamente cuando el empleado asociado sea eliminado. La declaración para crear una relación con eliminación en cascada sería:

```sql
CREATE TABLE pedidos (
    id_pedido INT PRIMARY KEY,
    id_empleado INT,
    producto VARCHAR(50),
    FOREIGN KEY (id_empleado) REFERENCES empleados(id_empleado) ON DELETE CASCADE
);
```

Con esta configuración, al eliminar un empleado, se eliminarán automáticamente todos los pedidos correspondientes.

3.4.2. Transacciones y seguridad en la eliminación de registros

Las transacciones son importantes para proteger la base de datos ante modificaciones no deseadas. **Permiten agrupar varias operaciones en una sola unidad lógica**. Si una parte de la transacción falla, es posible revertir toda la operación a su estado original.

El uso de transacciones puede expresarse de esta manera (MySQL):

```
DELIMITER $$

CREATE PROCEDURE eliminar_empleado(IN empleado_id INT)
BEGIN
    DECLARE exit HANDLER FOR SQLEXCEPTION
    BEGIN
        -- Si ocurre un error, revertimos la transacción
        ROLLBACK;
    END;

    -- Iniciamos la transacción
    START TRANSACTION;

    -- Intentamos eliminar al empleado
    DELETE FROM empleados WHERE id_empleado = empleado_id;

    -- Si no hay errores, confirmamos la transacción
    COMMIT;
END$$

DELIMITER ;
```

Este método asegura que los datos se mantengan consistentes. En entornos de producción, esto es especialmente relevante, especialmente al realizar múltiples eliminaciones que afectan a diferentes tablas.

3.4.3. Registros de auditoría

Se puede implementar un sistema de auditoría para mantener un historial de las eliminaciones realizadas en la base de datos. **Almacenar información sobre qué registros se eliminaron, quién los eliminó y cuándo, resulta útil para futuras evaluaciones o para la recuperación de datos en caso de error**. Un enfoque simple para registrar las eliminaciones podría implicar tener una tabla llamada `auditoria_eliminaciones`:

id_eliminacion	tabla	id_registro	fecha	usuario
1	empleados	1	2023-10-10 14:00	admin

Cada vez que se realice una eliminación, estos detalles pueden ser insertados en la tabla de auditoría, proporcionando un historial accesible y útil para referencias futuras.

3.4.4. Eliminaciones seguras y prevención de errores

Es recomendable implementar estrategias para realizar eliminaciones seguras. **Una práctica útil consiste en ejecutar una consulta SELECT antes de eliminar registros. Esto permite revisar qué datos se van a eliminar y, si es necesario, ajustar la cláusula WHERE para asegurar que se filtran correctamente los registros**:

```
SELECT * FROM empleados WHERE id_empleado = 1;
```

Otra estrategia puede ser la implementación de una "eliminación suave" o soft delete, donde no se eliminan físicamente los registros, sino que se marcan como inactivos. Esta técnica se puede llevar a cabo añadiendo una columna `activo` a la tabla:

id_empleado	nombre	departamento	activo
1	Javier	IT	0
2	Sofia	RRHH	1

Con esta estructura, en lugar de eliminar el registro de Javier, se actualizaría su estado:

```
UPDATE empleados SET activo = 0 WHERE id_empleado = 1;
```

Esta técnica permite conservar el historial de empleados y sus datos sin afectar la consulta de registros activos. En sistemas que requieren mantener un registro completo de la actividad histórica, la eliminación suave puede ser un enfoque acertado.

4. INTEGRIDAD DE DATOS

La integridad de datos se relaciona con la precisión y consistencia de la información almacenada en una base de datos. Asegurar que los datos sean correctos tiene un impacto significativo en las operaciones y decisiones dentro de un sistema. Existen diferentes métodos para garantizar esta integridad, comenzando por la '**integridad de entidad'**, que valida que cada registro en una tabla sea único y se pueda identificar de manera inequívoca, comúnmente a través de una clave primaria.

La integridad referencial mantiene la consistencia entre tablas relacionadas. **Esto se consigue mediante el uso de claves foráneas, que deben coincidir con los valores de una clave primaria en otra tabla**. Este enfoque previene situaciones de 'registros huérfanos', que son aquellos que hacen referencia a datos ausentes. Mantener esta relación es importante para garantizar que las consultas que involucren varias tablas produzcan resultados precisos.

Otra forma de integridad es la integridad de dominio, que establece limitaciones sobre los valores que pueden ser introducidos en un campo específico. Estas restricciones pueden incluir el tipo de dato, la longitud o los valores permitidos. Así, se evita la inserción de datos no válidos que podrían deteriorar la calidad de la información.

La integridad transaccional se relaciona con la ejecución de operaciones de bases de datos, asegurando que se realicen de manera completa o no se efectúen en absoluto. Este concepto resulta relevante en escenarios donde múltiples operaciones deben llevarse a cabo juntas, lo que está directamente relacionado con los mecanismos de Commit y Rollback, que permiten mantener la coherencia del sistema en situaciones de error o fallo durante una transacción.

Es importante implementar mecanismos de integridad adecuados en cualquier sistema de gestión de bases de datos para proteger la calidad de la información y asegurar la fiabilidad de las operaciones realizadas. Esto incluye no solo la definición de reglas y restricciones adecuadas al momento de diseñar la base de datos, sino también la monitorización continua y el manejo correcto de las transacciones en su uso cotidiano.

4.1. COMMIT

En sistemas de bases de datos relacionales, "commit" describe la operación que se utiliza para hacer permanentes los cambios que se realizan durante una transacción. Para comprenderlo con mayor claridad, es necesario desglosar la función de 'commit' y cómo se relaciona con las operaciones en la base de datos, especialmente en relación con las propiedades ACID.

Una transacción se define como un conjunto de operaciones que deben ejecutarse como una unidad única. Esto implica que o todas las operaciones de la transacción se ejecutan con éxito o, en caso contrario, ninguna se ejecutará. Este mecanismo ayuda a mantener la coherencia de los datos. Las propiedades ACID, que garantizan un manejo seguro de las transacciones, son:

- **Atomicidad**: Esta propiedad garantiza que todas las operaciones involucradas se concluyan satisfactoriamente o, en caso de error, se reviertan. *Por ejemplo, en una transferencia de fondos entre cuentas, si la operación de debitar una cuenta se completa pero la de acreditar otra falla, ambas acciones deben cancelarse para evitar la pérdida de dinero*.
- **Consistencia**: Las transacciones deben llevar la base de datos de un estado válido a otro estado también válido. *Por ejemplo, en un sistema de reservas, si se realiza la reserva de una habitación, el estado posterior debe mostrar que dicha habitación no está disponible. Si hay un error durante el proceso de reserva, el sistema debe regresar al estado anterior*.
- **Aislamiento**: Las transacciones deben llevarse a cabo sin interferencias entre ellas. Es decir, los cambios realizados por una transacción no deben ser visibles para otras transacciones hasta que se complete el proceso. *Por ejemplo, si dos usuarios intentan reservar la misma habitación casi simultáneamente, el sistema debe garantizar que una reserva no influya en la otra hasta que se realicen los commits necesarios*.
- **Durabilidad**: Una vez que se ha ejecutado un 'commit', los cambios realizados se mantienen incluso en el caso de que ocurra un fallo en el sistema. Esta propiedad se obtiene mediante el registro de las transacciones en un log, lo que permite restaurar los datos a un estado consistente tras un fallo.

La operación 'commit' indica el momento en que las modificaciones realizadas durante una transacción se hacen permanentes. Desde este punto, los datos quedan disponibles para consultas y otros procesos. *Por ejemplo, en un sistema de gestión de facturas, después de calcular y actualizar totales, se llevaría a cabo un commit para guardar esos datos. Si se produce un problema antes de esta operación, los cálculos no se guardan, evitando la entrada de información incorrecta*.

El uso de 'commit' también es relevante en la gestión de inventarios. *Por ejemplo: imaginemos que se está registrando la llegada de un nuevo lote de productos. Antes de confirmar el 'commit', el sistema puede validar que el número físico de productos recibido coincide con el registrarlo. En caso de discrepancias, se puede realizar un 'rollback' que elimine las operaciones anteriores, asegurando que no se registren datos erróneos en la base de datos*.

El rendimiento del sistema puede verse afectado por la frecuencia con la que se realizan los commits. Cada operación de commit implica la escritura de cambios en el disco, lo que puede consumir recursos. **Para optimizar este proceso, algunas bases de datos utilizan técnicas como la escritura diferida**. Este enfoque permite agrupar múltiples cambios en la memoria y ejecutar un solo commit, en lugar de varios, lo que mejora la eficiencia.

Un ejemplo adicional se presenta en aplicaciones de comercio electrónico. Al procesar un pedido, pueden requerirse varias actualizaciones en la base de datos, como reducir el inventario, actualizar el historial de compras del cliente y enviar una confirmación de pedido. Todas estas operaciones deben realizarse dentro de una única transacción. Se ejecutaría un commit una vez que todas las

operaciones sean exitosas, garantizando cohesión. Si alguna de las acciones falla, por ejemplo, si un artículo está agotado, se ejecutará un rollback, recuperando el estado anterior.

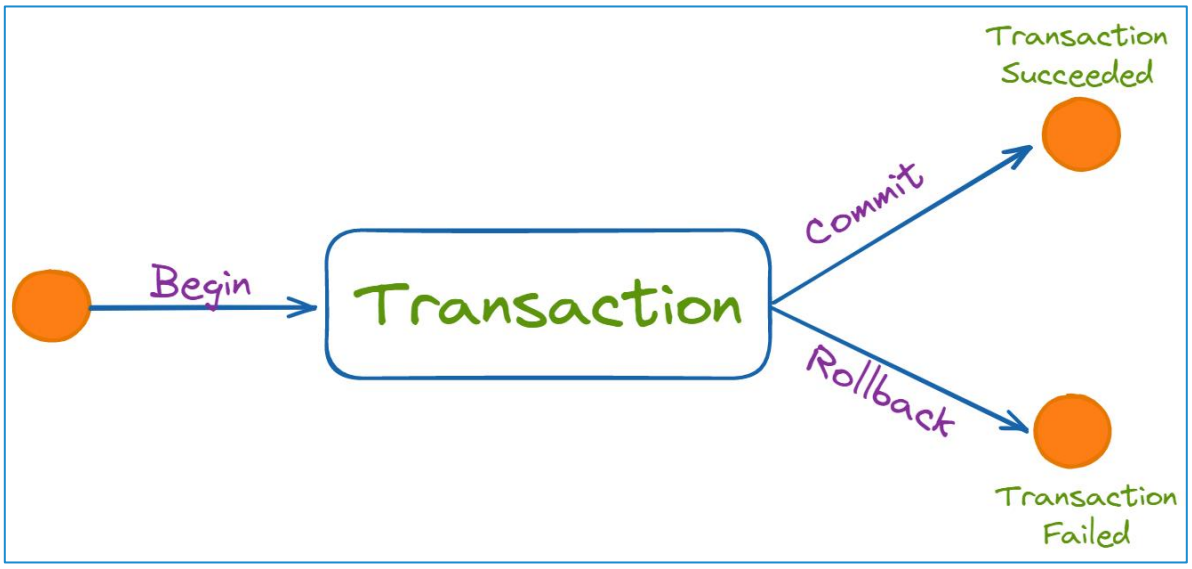
El 'Two-Phase Commit (2PC)' se emplea en entornos distribuidos donde las transacciones incluyen múltiples bases de datos. *Este protocolo consiste en dos fases: en la primera, el coordinador solicita a todos los participantes que indiquen si pueden realizar commit. Si todos los participantes votan a favor, en la segunda fase se lleva a cabo el commit. Si algún participante vota negativamente, la transacción se deshace y se aplican los rollbacks necesarios*.

El control de accesos es un aspecto importante en la gestión de commits. Solo los usuarios autorizados deben tener la capacidad de realizar commits en la base de datos, especialmente en sistemas que manejen información sensible o crítica. **Es recomendable establecer roles y permisos que regulen quién tiene derecho a confirmar cambios**. Por ejemplo, en una plataforma de gestión de datos médicos, solo el personal médico con autorización debería poder realizar commit de cambios en los registros de los pacientes.

Además, **incorporar mecanismos de logging detallados resulta útil para mantener la integridad de los datos y recuperar la base de datos en caso de fallos**. Un registro de transacciones permite a los administradores restaurar el sistema a un estado consistente tras un fallo, examinando todas las operaciones realizadas antes del incidente.

Todo este proceso relacionado con commit es relevante para la gestión de bases de datos relacionales, ya que asegura el manejo adecuado de las transacciones. Esto resulta esencial en diversos escenarios de aplicación, garantizando que los datos se procesen de forma segura y confiable.

4.2. ROLLBACK



El concepto de 'rollback' se centra en la restauración de una base de datos a un estado anterior ante la ocurrencia de un error o la decisión de no realizar una operación. Dentro de la gestión de transacciones en bases de datos relacionales, el rollback actúa como un mecanismo importante para asegurar la consistencia de los datos. Este mecanismo está destinado a garantizar que las operaciones se ejecuten de forma atómica, evitando así que se generen datos inconsistentes o errores en el sistema.

Cuando se inicia una transacción, puede involucrar múltiples acciones que cambian el contenido de la base de datos. *Por ejemplo, en un sistema de gestión de inventarios, una transacción podría incluir las siguientes operaciones: registrar la venta de un producto, disminuir la cantidad disponible en el inventario y generar un registro de la transacción en la base de datos de ventas. Si, durante alguno de estos pasos, hay un fallo, la utilización de rollback permite revertir todos los cambios realizados hasta ese punto. Así, la base de datos regresaría al estado previo al inicio de la transacción*.

Un caso práctico que ilustra el uso de rollback es en el área bancaria. Consideremos una transacción en la que un cliente realiza una transferencia de fondos desde su cuenta a otra. La operación implica varias etapas: primero, se verifica que el cliente tenga suficientes fondos; luego, se deduce el monto de la cuenta del remitente; y, finalmente, se acredita el mismo monto en la cuenta del destinatario. Si hay un problema técnico impide actualizar la cuenta del destinatario, el sistema aplicaría un rollback para revertir la deducción en la cuenta del remitente. De esta manera, se asegura que no haya pérdida de fondos y que las cuentas involucradas permanezcan correctas.

Los registros de transacciones tienen un papel importante en la implementación de rollback. Estos registros, conocidos como logs, contienen información sobre cada operación que ha modificado los datos. Al comenzar una transacción, se registran cada cambio en el log. Si se produce un error y es necesario ejecutar un rollback, el sistema consulta el log para identificar las acciones que deben deshacerse, aplicando los cambios en reversa hasta alcanzar el estado anterior. Esto significa que se deben almacenar detalles específicos sobre cada operación, como el tipo de modificación y los valores previos de los datos antes de que se apliquen cambios actuales.

En aplicaciones de comercio electrónico, el rollback también se pone en práctica. Imaginemos que un cliente añade varios artículos a su carrito de compras y procede a realizar el pago. Si en este proceso hay un error en el procesamiento del pago, el sistema debe revertir cualquier cambio realizado, como la disminución del stock de productos reservados temporalmente para el pedido. Aquí, el rollback aseguraría que todos los artículos regresen a la disponibilidad para otros compradores.

La funcionalidad de rollback también puede ser útil en situaciones donde el resultado de una operación no sea satisfactorio para el usuario. En plataformas de edición de contenido, si un usuario modifica un texto y decide que no desea conservar esos cambios, el sistema debe ofrecer una opción de deshacer, que representa un rollback de las modificaciones realizadas. Este tipo de función es común en programas de procesamiento de texto, donde las correcciones o ediciones pueden revertirse si el usuario determina que son inapropiadas.

El uso de transacciones se extiende a bases de datos distribuidas, donde varios nodos pueden participar en una única operación. En estos sistemas, la coordinación del rollback puede ser más complicada, ya que es necesario asegurar que todas las partes de la transacción sean revertidas si una de ellas falla. **Esto implica técnicas adicionales de sincronización y comunicación entre los nodos para conservar la integridad global de los datos**.

A medida que las aplicaciones aumentan en complejidad y manejan mayores volúmenes de datos, el rollback y el tratamiento de transacciones en bases de datos relacionales se vuelven más significativos. La capacidad de revertir cambios de forma confiable no solo asegura la coherencia de los datos, sino que también influye en la usabilidad y la confianza del usuario en el sistema.

RESUMEN

El acceso a bases de datos se realiza a través de JDBC (Java Database Connectivity), una tecnología en Java diseñada para que las aplicaciones se conecten y comuniquen con bases de datos relacionales. JDBC proporciona un conjunto de interfaces y clases que facilitan la interacción con sistemas de gestión de bases de datos que utilizan SQL, permitiendo ejecutar consultas SQL desde el código Java para acceder, modificar y gestionar datos de forma dinámica y eficiente.

El proceso de consulta a una base de datos se lleva a cabo a través de varias clases proporcionadas por JDBC. La clase "Statement" se utiliza para ejecutar sentencias SQL simples y es adecuada para situaciones que no requieren parametrización de las consultas, aunque es menos segura frente a inyecciones SQL. Para mitigar el riesgo de ataques, se recomienda utilizar "PreparedStatement", que permite precompilar las consultas SQL y ejecutarlas múltiples veces con diferentes parámetros, mejorando el rendimiento y la seguridad. La clase "ResultSet" es utilizada para manejar los resultados devueltos por las consultas SQL.

Las operaciones CRUD representan las funciones básicas necesarias para gestionar datos dentro de una base de datos relacional: Crear (Create), Leer (Read), Actualizar (Update) y Eliminar (Delete).

La integridad de los datos se garantiza mediante el uso de transacciones, que consisten en una serie de operaciones que se ejecutan como una unidad. Las transacciones se gestionan con los comandos 'commit' y 'rollback'.

La clase `PreparedStatement` mejora la seguridad y el rendimiento al permitir consultas SQL precompiladas. Su uso básico implica crear la consulta con marcadores de posición (`?`) y luego establecer los valores en tiempo de ejecución. Estas consultas pueden ejecutar operaciones como `SELECT`, `INSERT`, `UPDATE` y `DELETE`, según sea necesario.

La operación "Update" se basa en la instrucción SQL `UPDATE`, que define el nombre de la tabla, las columnas y sus nuevos valores, y una condición que especifica qué registros deben ser modificados. Las actualizaciones pueden afectar una sola fila o varias filas dependiendo de las condiciones especificadas.

La operación "Delete" permite eliminar registros de una tabla específica usando la instrucción SQL `DELETE`, especificando una condición para determinar qué registros deben ser eliminados. Es crucial usar la cláusula `WHERE` para evitar la eliminación accidental de todos los registros de la tabla.

Finalmente, el manejo de errores es un componente crítico en la gestión de bases de datos. Implementar bloques de captura de excepciones (`try-catch`) permite interceptar y manejar errores como `SQLException`, asegurando que la aplicación pueda responder de manera adecuada a problemas inesperados durante la interacción con la base de datos.