Memoria Técnica Práctica 2
Bases de Datos 2
Curso 2024/2025
Universidad de Zaragoza

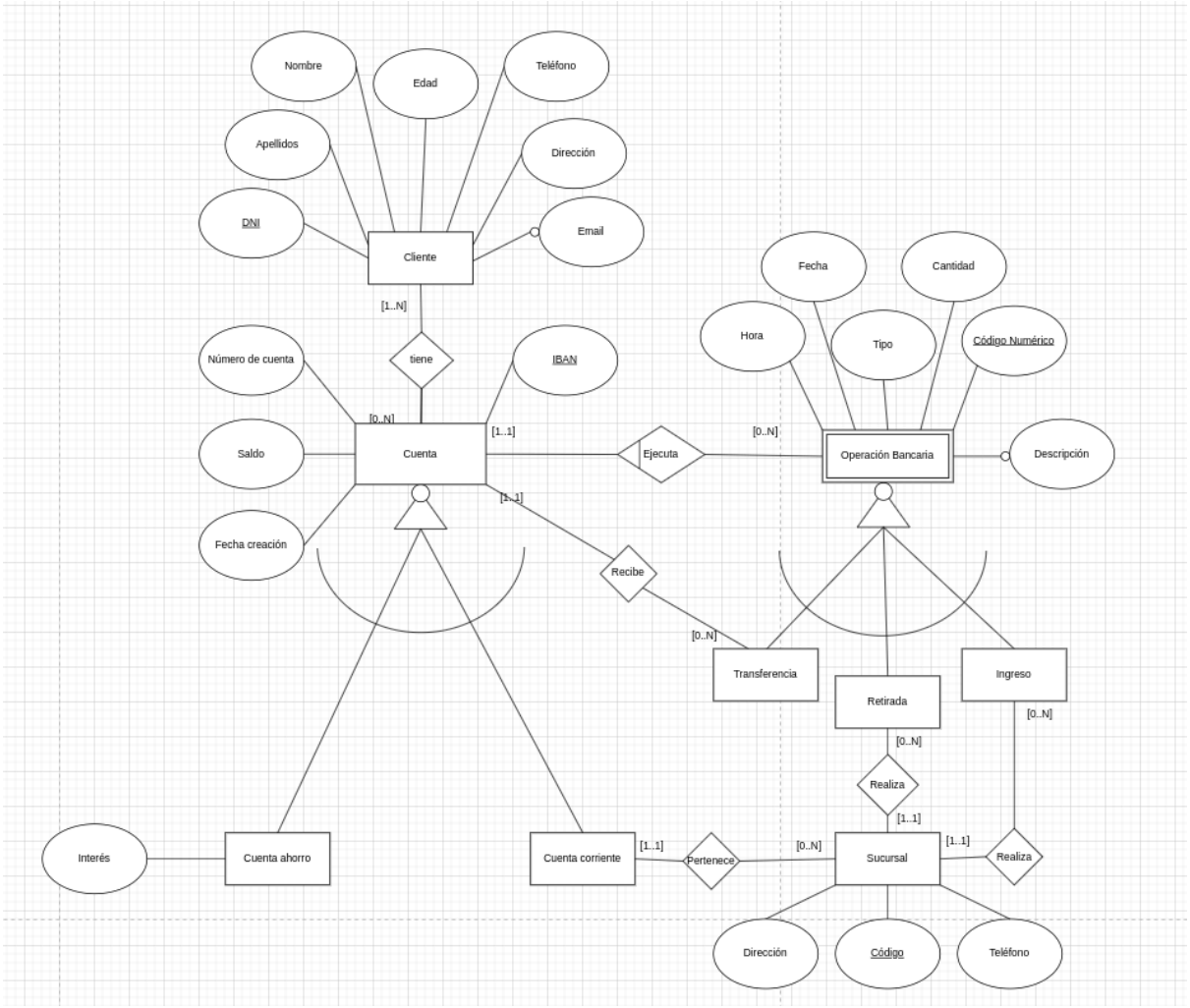
Diego Mateo Lorente - 873338
Daniel Simón Gayán - 870984
Carlos Solana Melero - 872815

Esfuerzos invertidos

Horas de cada uno

	Diego	Carlos	Daniel	Total
P2	13h	10h	8h 30m	31h 30m

Diseño Conceptual de la Base de Datos



El modelo Entidad-Relación presentado describe un sistema bancario mediante una estructura que refleja las relaciones entre clientes, cuentas y operaciones bancarias, destacando especialmente por sus dos jerarquías principales: una para la entidad cuenta, y otra para la entidad operación bancaria, ambas especializaciones son exclusivas (cada instancia de la superentidad solo puede pertenecer a una única subentidad.) y totales (cada instancia de la superentidad necesariamente pertenece a una de las subentidades definidas) La entidad Cuenta constituye una jerarquía esencial del modelo. Se plantea inicialmente como una superentidad general que tiene los atributos comunes, tales como

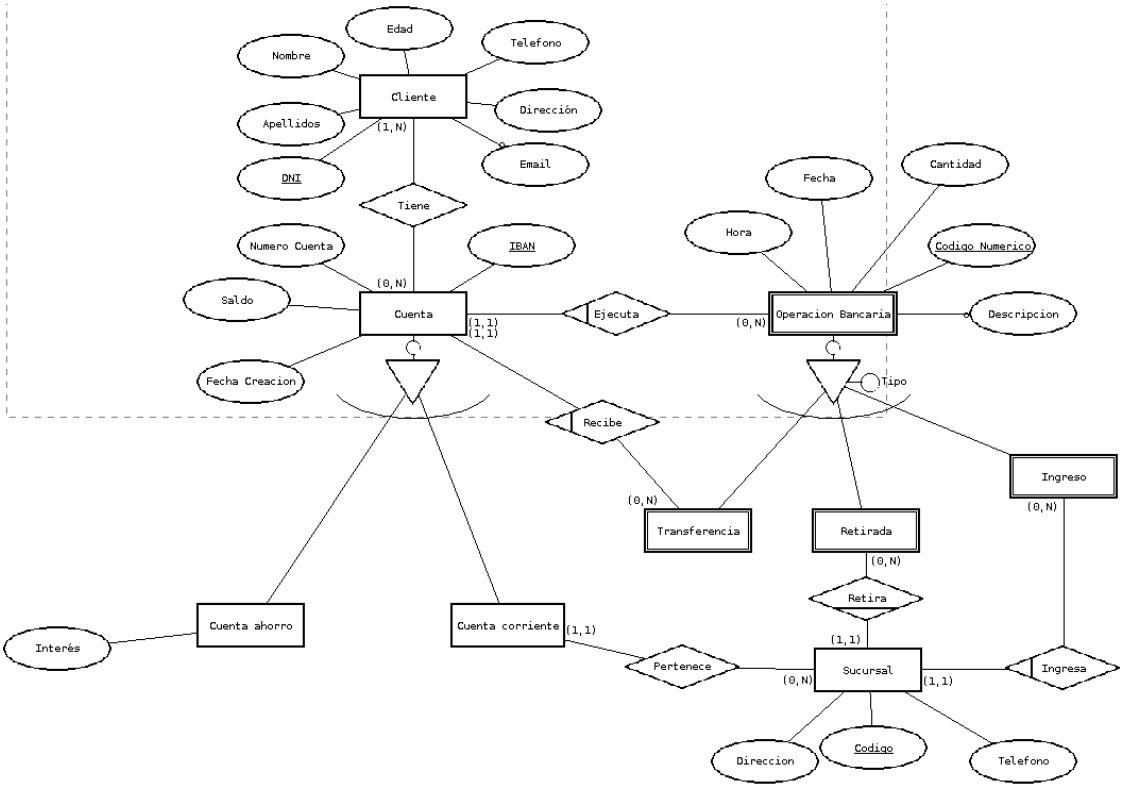
Número de cuenta, IBAN que es la clave primaria de dicha superentidada, Saldo y Fecha de creación. Desde esta superentidad, se especializan en dos subentidades específicas: Cuenta Ahorro y Cuenta Corriente. Ambas subentidades heredan todos los atributos generales de la cuenta principal y añaden características particulares que las diferencian, en el caso de la subentidad Cuenta Ahorro, esta incorpora un atributo adicional denominado Interés, por su parte, la Cuenta Corriente no añade atributos propios, pero tiene una relación específica con la entidad Sucursal, indicando de forma obligatoria la pertenencia a una única sucursal. Esta jerarquía permite representar diferentes tipos de cuentas bancarias, facilitando una gestión organizada de los datos y diferenciando los aspectos comunes y particulares de cada tipo de cuenta manteniendo sus similitudes derivadas de la superentidad.

Por otro lado, la entidad Operación Bancaria también actúa como una superentidad que representa cualquier transacción relacionada con las cuentas bancarias, es una entidad débil respecto a Cuenta ya que sin una cuenta, las operaciones bancarias no tendrían sentido su existencia y además sería necesaria la cuenta para identificar a las operaciones ya que por sí solas con el código numérico son incapaces . Sus atributos principales son Fecha, Hora, Cantidad, Tipo, Código numérico y Descripción. Esta entidad, a su vez, da lugar a una segunda jerarquía por tres subentidades especializadas: Transferencia, Retirada e Ingreso. Cada una de estas subentidades hereda los atributos generales mencionados anteriormente y, además, participa en relaciones específicas según el tipo de operación. En particular, la subentidad Transferencia establece relaciones con dos cuentas distintas: una emisora y otra receptora, Las subentidades Retirada e Ingreso están relacionadas obligatoriamente a una sucursal específica mediante la relación Realiza.

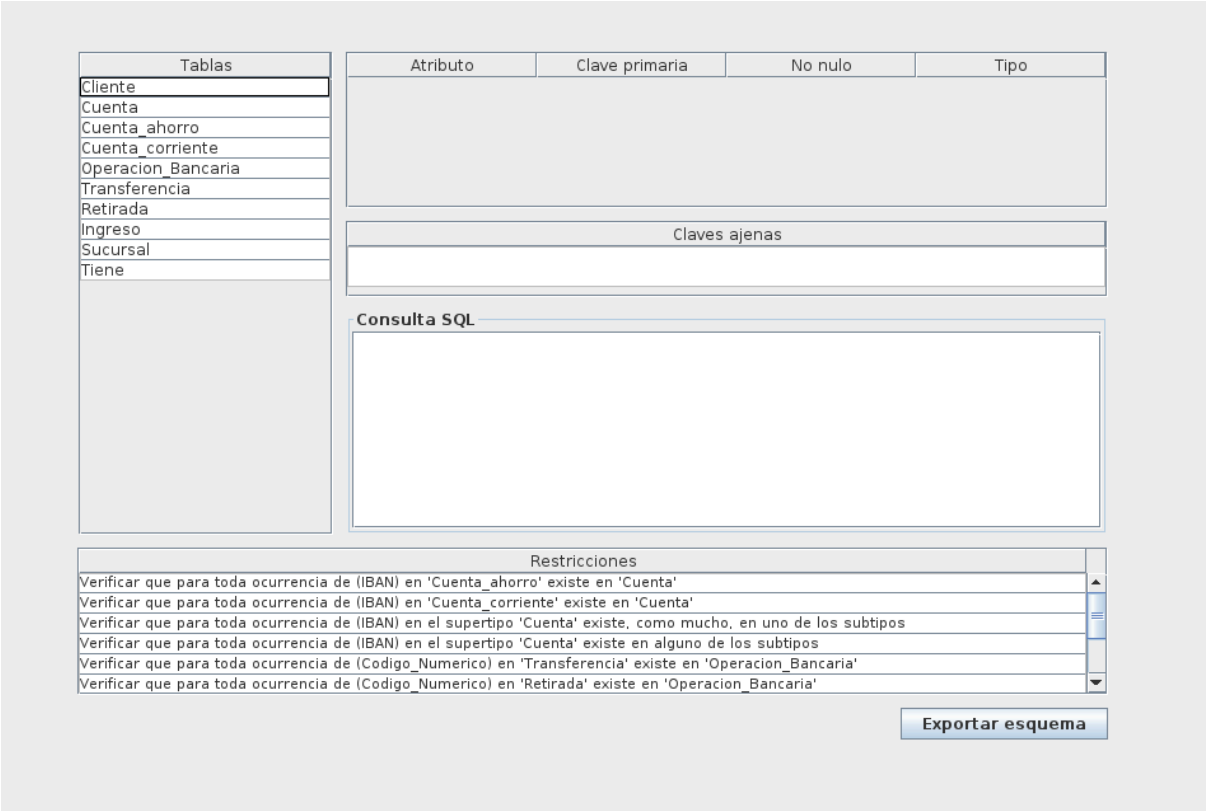
Finalmente se han definido 2 entidades, La entidad Cliente representa a las personas que poseen una relación con el banco, siendo esencial dentro del modelo planteado. Sus atributos son Nombre, Apellidos, Edad, Teléfono, Dirección, Email y DNI, donde el DNI actúa como clave primaria única. Esta entidad participa en una relación múltiple con la entidad Cuenta, indicando que un cliente puede tener varias cuentas bancarias y, al mismo tiempo, una cuenta puede ser compartida por varios clientes. Por otro lado, la entidad Sucursal describe las oficinas físicas del banco donde se gestionan y realizan operaciones financieras. Sus atributos clave son Código (que es clave primaria) , dirección y Teléfono. Esta entidad está vinculada directamente con la jerarquía de Operación Bancaria mediante la relación denominada Realiza, indicando que operaciones específicas, tales como Ingresos y Retiradas, se deben ejecutar obligatoriamente en una única sucursal. Además, la entidad Sucursal se relaciona con las cuentas corrientes, especificando la sucursal a la que pertenece cada cuenta corriente.

Diseño Lógico de una Base de Datos Relacional

Para facilitar la tarea de la transformación del Modelo ER al Modelo Relacional, se ha realizado una versión del diagrama ER en la aplicación DBDap, un programa software de alto nivel que permite realizar diagramas de forma sencilla.



Una gran ventaja de este programa es que, habiendo realizado el diagrama ER, es posible realizar una transformación automática al diagrama relacional, lo cual acelera considerablemente el proceso. Cuando se realiza la transformación, el programa pregunta la forma en la cual se quieren transformar las jerarquías. Se ha decidido mantener las tablas padre para evitar la redundancia de atributos comunes, facilitar el mantenimiento y reflejar fielmente la jerarquía del modelo conceptual. Esta opción permite una estructura más normalizada y coherente, aprovechando la funcionalidad de herencia que ofrece DB2. Así, las tabla generadas quedan de esta manera generadas por DBDap:



Finalmente, se han definido una serie de restricciones textuales para mejorar la robustez de la base de datos diseñada:

- Se restringe la realización de transferencias si la cuenta emisora no tiene saldo suficiente para cubrir el importe.

- El saldo de la cuenta se actualiza automáticamente al realizar un ingreso, sumando el importe al saldo actual.

- El saldo de la cuenta se actualiza automáticamente al realizar una retirada, restando el importe del saldo actual.

- No es posible eliminar un cliente sin eliminar previamente sus relaciones con las cuentas asociadas.

- Se impide realizar transferencias cuando la cuenta emisora y la receptora son la misma.

Implementación con el Modelo Relacional

Este es el script utlizado para crear las tablas que implementan el modelo relacional

```
#!/usr/bin/env bash
```

```
# Script para dropear y crear las tablas en Oracle XE dentro de un
contenedor Docker.
# Asegúrate de que tu contenedor se llame "p2-oracle-1" (según tu
docker-compose.yml),
# y que las credenciales y el servicio (XEPDB1) sean correctos.

docker exec -i p2-oracle-1 sqlplus admin/admin@//localhost:1521/XEPDB1
<<'EOF'
-- Bloques para dropear tablas si existen. Se usa SQLCODE -942 (tabla
no existe).

BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Retirada CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Ingreso CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Transferencia CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE OperacionBancaria CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Sucursal CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Corriente CASCADE CONSTRAINTS';
```

```
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Ahorro CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE CuentaCliente CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Cuenta CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE Cliente CASCADE CONSTRAINTS';
EXCEPTION WHEN OTHERS THEN
  IF SQLCODE != -942 THEN RAISE; END IF;
END;
/

-- Creación de tablas en el orden adecuado

-- Tabla CLIENTE
CREATE TABLE Cliente (
   dni         NUMBER(10)       NOT NULL,
   nombre      VARCHAR2(50)     NOT NULL,
   apellidos   VARCHAR2(50)     NOT NULL,
   edad        NUMBER(3)        NOT NULL,
   telefono    VARCHAR2(20)     NOT NULL,
   email       VARCHAR2(50),     -- Opcional
   direccion   VARCHAR2(100)    NOT NULL,
   CONSTRAINT pk_cliente PRIMARY KEY (dni)
);
```

```
-- Tabla CUENTA
CREATE TABLE Cuenta (
   iban            VARCHAR2(34)      NOT NULL,
   numero_cuenta   VARCHAR2(20)      NOT NULL,
   saldo           NUMBER(15,2)      NOT NULL,
   fecha_creacion  DATE,
   CONSTRAINT pk_cuenta PRIMARY KEY (iban)
);

-- Tabla intermedia CUENTACLIENTE (relaciona Cliente y Cuenta)
CREATE TABLE CuentaCliente (
   dni  NUMBER(10)       NOT NULL,
   iban VARCHAR2(34)     NOT NULL,
   CONSTRAINT pk_cuentacliente PRIMARY KEY (dni, iban),
   CONSTRAINT fk_cuentacliente_cliente
       FOREIGN KEY (dni)
       REFERENCES Cliente (dni),
   CONSTRAINT fk_cuentacliente_cuenta
       FOREIGN KEY (iban)
       REFERENCES Cuenta (iban)
);

-- Tabla AHORRO (para cuentas de ahorro)
CREATE TABLE Ahorro (
   iban    VARCHAR2(34)  NOT NULL,
   interes NUMBER(5,2)   NOT NULL,
   CONSTRAINT pk_ahorro PRIMARY KEY (iban),
   CONSTRAINT fk_ahorro_cuenta
       FOREIGN KEY (iban)
       REFERENCES Cuenta (iban)
);

-- Tabla CORRIENTE (para cuentas corrientes)
CREATE TABLE Corriente (
   iban            VARCHAR2(34)  NOT NULL,
   codigo_sucursal NUMBER(10)    NOT NULL,
   CONSTRAINT pk_corriente PRIMARY KEY (iban, codigo_sucursal),
   CONSTRAINT fk_corriente_cuenta
       FOREIGN KEY (iban)
       REFERENCES Cuenta (iban)
);

-- Tabla SUCURSAL
```

```
CREATE TABLE Sucursal (
   codigo_sucursal NUMBER(10)     NOT NULL,
   direccion       VARCHAR2(100)  NOT NULL,
   telefono        VARCHAR2(20)   NOT NULL,
   CONSTRAINT pk_sucursal PRIMARY KEY (codigo_sucursal)
);

-- Tabla OPERACIONBANCARIA (superentidad)
CREATE TABLE OperacionBancaria (
   codigo_numerico NUMBER(10)    NOT NULL,
   iban            VARCHAR2(34)  NOT NULL,  -- Cuenta sobre la que se
hace la operación
   cantidad        NUMBER(15,2),
   tipo            VARCHAR2(20),
   fecha           DATE,
   hora            TIMESTAMP,
   CONSTRAINT pk_operacionbancaria PRIMARY KEY (codigo_numerico, iban),
   CONSTRAINT fk_operacionbancaria_cuenta
       FOREIGN KEY (iban)
       REFERENCES Cuenta (iban),
   CONSTRAINT tipo_operacion CHECK (tipo IN ('Retirada', 'Ingreso',
'Transferencia'))
);

-- Subentidad RETIRADA
CREATE TABLE Retirada (
   codigo_numerico NUMBER(10)    NOT NULL,
   codigo_sucursal NUMBER(10)    NOT NULL,
   iban            VARCHAR2(34)  NOT NULL,
   CONSTRAINT pk_retirada PRIMARY KEY (codigo_numerico,
codigo_sucursal, iban),
   CONSTRAINT fk_retirada_operacion
       FOREIGN KEY (codigo_numerico,iban)
       REFERENCES OperacionBancaria (codigo_numerico,iban),
   CONSTRAINT fk_operacionbancaria_sucursal
       FOREIGN KEY (codigo_sucursal)
       REFERENCES Sucursal (codigo_sucursal)
);

-- Subentidad INGRESO
CREATE TABLE Ingreso (
   codigo_numerico NUMBER(10)    NOT NULL,
   codigo_sucursal NUMBER(10)    NOT NULL,
```

```
   iban            VARCHAR2(34)  NOT NULL,
   CONSTRAINT pk_ingreso PRIMARY KEY (codigo_numerico, codigo_sucursal,
iban),
   CONSTRAINT fk_ingreso_operacion
       FOREIGN KEY (codigo_numerico, iban)
       REFERENCES OperacionBancaria (codigo_numerico, iban),
   CONSTRAINT fk_operacionbancaria_sucursal_ingreso
       FOREIGN KEY (codigo_sucursal)
       REFERENCES Sucursal (codigo_sucursal)
);

-- Tabla TRANSFERENCIA
CREATE TABLE Transferencia (
   codigo_numerico NUMBER(10)    NOT NULL,
   iban_emisor     VARCHAR2(34)  NOT NULL,
   iban_receptor   VARCHAR2(34)  NOT NULL,
   CONSTRAINT pk_transferencia PRIMARY KEY (codigo_numerico,
iban_emisor, iban_receptor),
   CONSTRAINT fk_transferencia_emisor
       FOREIGN KEY (codigo_numerico, iban_emisor)
       REFERENCES OperacionBancaria (codigo_numerico, iban),
   CONSTRAINT fk_transferencia_receptor
       FOREIGN KEY (iban_receptor)
       REFERENCES Cuenta (iban)
);

EXIT;
EOF
```

Las tablas creadas son las del modelo relacional
Cliente:
Contiene los datos personales de los clientes (DNI, nombre, apellidos, etc.), donde el DNI actúa como clave primaria.

Cuenta:
Almacena la información común de todas las cuentas bancarias (IBAN, número de cuenta, saldo, fecha de creación), siendo el IBAN la clave primaria.

CuentaCliente:
Es la tabla intermedia que gestiona la relación muchos a muchos entre Cliente y Cuenta, permitiendo que un cliente tenga varias cuentas y una cuenta pueda estar asociada a varios

clientes.

Ahorro:
Especializa la tabla Cuenta para representar las cuentas de ahorro. Se utiliza el mismo IBAN como clave primaria y se añade el atributo “interés” para diferenciar este tipo de cuenta.

Corriente:
Representa las cuentas corrientes, extendiendo la tabla Cuenta. Además de los atributos comunes, se relaciona obligatoriamente con la tabla Sucursal mediante el atributo “código_sucursal”, que forma parte de su clave primaria compuesta.

Sucursal:
Registra los datos de las oficinas físicas del banco (código de sucursal, dirección, teléfono), con el código de sucursal como clave primaria.

OperacionBancaria:
Actúa como superentidad para todas las transacciones realizadas sobre las cuentas. Incluye atributos generales (código numérico, IBAN, cantidad, tipo, fecha y hora) y se vincula con la cuenta a través del IBAN. Se ha incluido una restricción para que el tipo de operación sea retirada, ingreso o transferencia.

Retirada e Ingreso:
Son subentidades de OperacionBancaria que representan, respectivamente, las operaciones de retirada e ingreso de efectivo. Cada una utiliza claves compuestas (incluyendo código numérico, IBAN y código de sucursal) y se asocia obligatoriamente con una sucursal.

Transferencia:
Especializa OperacionBancaria para las transacciones de transferencia. Se definen claves compuestas que incluyen el código numérico, el IBAN del emisor y el IBAN del receptor, estableciendo la relación entre ambas cuentas.

Triggers:

```
docker exec -i p2-oracle-1 sqlplus admin/admin@//localhost:1521/XEPDB1
<<'EOF'
-- ############################################################
-- Script: docker_oracle_triggers.sql
-- Objetivo:
--   - Eliminar triggers existentes (si existen).
--   - Crear triggers adaptados a Oracle para:
--       1) Verificar saldo suficiente antes de transferencia.
--       2) Actualizar saldo después de ingreso.
--       3) Actualizar saldo después de retirada.
--       4) Eliminar relaciones Cliente-Cuenta al borrar un Cliente.
```

```
--       5) Evitar transferencia a la misma cuenta.
--       6) Validar el tipo de operación (INGRESO, TRANSFERENCIA,
RETIRO).
--
--   - Insertar datos de prueba que verifiquen el funcionamiento de los
triggers.
-- ############################################################

-- ------------------------------------------------------------
-- 1. Eliminación de TRIGGERS previos (si existieran)
--    Usamos bloques anónimos para evitar errores si no existen.
-- ------------------------------------------------------------

BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER check_balance_before_transfer';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER update_balance_after_ingreso';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER update_balance_after_retiro';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER delete_cliente_cuentas';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER check_transferencia_same_account';
```

```
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER check_tipo_operacion_ingreso';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER check_tipo_operacion_transferencia';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
BEGIN
  EXECUTE IMMEDIATE 'DROP TRIGGER check_tipo_operacion_retiro';
EXCEPTION
  WHEN OTHERS THEN
     IF SQLCODE != -4080 THEN RAISE; END IF;
END;
/
PROMPT "Triggers anteriores (si existían) han sido eliminados."

-- ------------------------------------------------------------
-- 2. Creación de funciones (PL/SQL) y TRIGGERS en ORACLE
--    Se utilizan triggers directos (sin crear funciones aparte)
--    dado que en Oracle es común escribir la lógica en el bloque
--    del propio trigger.
-- ------------------------------------------------------------

-- 2.1. Verificar saldo suficiente antes de una transferencia
CREATE OR REPLACE TRIGGER check_balance_before_transfer
BEFORE INSERT ON Transferencia
FOR EACH ROW
DECLARE
   v_saldo    NUMBER(15,2);
   v_cantidad NUMBER(15,2);
BEGIN
```

```
   -- Obtenemos la cantidad y el saldo de la cuenta emisora
   SELECT cantidad
     INTO v_cantidad
     FROM OperacionBancaria
    WHERE codigo_numerico = :NEW.codigo_numerico
      AND iban = :NEW.iban_emisor;

   SELECT saldo
     INTO v_saldo
     FROM Cuenta
    WHERE iban = :NEW.iban_emisor;

   IF v_saldo < v_cantidad THEN
      RAISE_APPLICATION_ERROR(-20001, 'Saldo insuficiente para realizar
la transferencia.');
   END IF;
END;
/
PROMPT "Trigger check_balance_before_transfer creado."

-- 2.2. Actualizar saldo después de un ingreso
CREATE OR REPLACE TRIGGER update_balance_after_ingreso
AFTER INSERT ON Ingreso
FOR EACH ROW
DECLARE
   v_cantidad NUMBER(15,2);
BEGIN
   -- Obtenemos la cantidad de la operación
   SELECT cantidad
     INTO v_cantidad
     FROM OperacionBancaria
    WHERE codigo_numerico = :NEW.codigo_numerico
      AND iban = :NEW.iban;

   -- Actualizamos el saldo
   UPDATE Cuenta
      SET saldo = saldo + v_cantidad
    WHERE iban = :NEW.iban;
END;
/
PROMPT "Trigger update_balance_after_ingreso creado."

-- 2.3. Actualizar saldo después de una retirada
```

```
CREATE OR REPLACE TRIGGER update_balance_after_retiro
AFTER INSERT ON Retirada
FOR EACH ROW
DECLARE
   v_cantidad NUMBER(15,2);
BEGIN
   -- Obtenemos la cantidad de la operación
   SELECT cantidad
     INTO v_cantidad
     FROM OperacionBancaria
    WHERE codigo_numerico = :NEW.codigo_numerico
      AND iban = :NEW.iban;

   -- Actualizamos el saldo
   UPDATE Cuenta
      SET saldo = saldo - v_cantidad
    WHERE iban = :NEW.iban;
END;
/
PROMPT "Trigger update_balance_after_retiro creado."

-- 2.4. Eliminar relaciones en CUENTACLIENTE al borrar un CLIENTE
CREATE OR REPLACE TRIGGER delete_cliente_cuentas
AFTER DELETE ON Cliente
FOR EACH ROW
BEGIN
   DELETE FROM CuentaCliente
    WHERE dni = :OLD.dni;
END;
/
PROMPT "Trigger delete_cliente_cuentas creado."

-- 2.5. Evitar transferencia a la misma cuenta (mismo IBAN
emisor/receptor)
CREATE OR REPLACE TRIGGER check_transferencia_same_account
BEFORE INSERT ON Transferencia
FOR EACH ROW
BEGIN
   IF :NEW.iban_emisor = :NEW.iban_receptor THEN
      RAISE_APPLICATION_ERROR(-20002, 'No se puede realizar una
transferencia a la misma cuenta.');
   END IF;
END;
```

```
/
PROMPT "Trigger check_transferencia_same_account creado."

-- 2.6. Validar que el tipo de operación sea 'INGRESO' para la tabla
INGRESO
CREATE OR REPLACE TRIGGER check_tipo_operacion_ingreso
BEFORE INSERT ON Ingreso
FOR EACH ROW
DECLARE
   v_tipo VARCHAR2(20);
BEGIN
   SELECT tipo
     INTO v_tipo
     FROM OperacionBancaria
    WHERE codigo_numerico = :NEW.codigo_numerico
      AND iban = :NEW.iban;

   IF UPPER(v_tipo) <> 'INGRESO' THEN
      RAISE_APPLICATION_ERROR(-20003, 'El tipo de operación no es
"INGRESO".');
   END IF;
END;
/
PROMPT "Trigger check_tipo_operacion_ingreso creado."

-- 2.7. Validar que el tipo de operación sea 'TRANSFERENCIA' para la
tabla TRANSFERENCIA
CREATE OR REPLACE TRIGGER check_tipo_operacion_transferencia
BEFORE INSERT ON Transferencia
FOR EACH ROW
DECLARE
   v_tipo VARCHAR2(20);
BEGIN
   SELECT tipo
     INTO v_tipo
     FROM OperacionBancaria
    WHERE codigo_numerico = :NEW.codigo_numerico
      AND iban = :NEW.iban_emisor;

   IF UPPER(v_tipo) <> 'TRANSFERENCIA' THEN
      RAISE_APPLICATION_ERROR(-20004, 'El tipo de operación no es
"TRANSFERENCIA".');
   END IF;
```

```
END;
/
PROMPT "Trigger check_tipo_operacion_transferencia creado."

-- 2.8. Validar que el tipo de operación sea 'RETIRO' para la tabla
RETIRADA
CREATE OR REPLACE TRIGGER check_tipo_operacion_retiro
BEFORE INSERT ON Retirada
FOR EACH ROW
DECLARE
   v_tipo VARCHAR2(20);
BEGIN
   SELECT tipo
     INTO v_tipo
     FROM OperacionBancaria
    WHERE codigo_numerico = :NEW.codigo_numerico
      AND iban = :NEW.iban;

   IF UPPER(v_tipo) <> 'RETIRO' THEN
      RAISE_APPLICATION_ERROR(-20005, 'El tipo de operación no es
"RETIRO".');
   END IF;
END;
/
PROMPT "Trigger check_tipo_operacion_retiro creado."

PROMPT "Todos los triggers han sido creados correctamente."
--

-- ######################################################
-- 3. Inserts de PRUEBA
--    Asumimos que ya existen varias cuentas y un cliente
--    creados según tu script de creación de tablas y datos
--    iniciales. En caso necesario, descomenta o ajusta
--    según tu data real (IBAN, saldo, etc.).
-- ######################################################

PROMPT "Insertando datos de prueba..."

-- Ejemplo de creación rápida de cuentas y clientes
-- (Descomenta si necesitas probar desde cero)
-- INSERT INTO Cliente(dni, nombre, apellidos, edad, telefono, email,
direccion)
```

```
-- VALUES (1001, 'Juan', 'Pérez', 30, '123456789',
'ejemplo@cliente.com', 'Calle Falsa 123');
-- INSERT INTO Cuenta(iban, numero_cuenta, saldo, fecha_creacion)
-- VALUES ('ES1234567890123456789012', '1234567890', 1000, SYSDATE);
-- INSERT INTO Cuenta(iban, numero_cuenta, saldo, fecha_creacion)
-- VALUES ('ESCORRIENTE1234567890123', '5432109876', 1500, SYSDATE);
-- INSERT INTO Cuenta(iban, numero_cuenta, saldo, fecha_creacion)
-- VALUES ('ESAHORRO1234567890123456', '0987654321', 2000, SYSDATE);
-- INSERT INTO CuentaCliente(dni, iban)
-- VALUES (1001, 'ES1234567890123456789012');

-- ----------------------------
-- Test A1: Transferencia válida
--  1) Creamos la operación general en OperacionBancaria
--  2) Insertamos en Transferencia
-- ----------------------------
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (101, 'ES1234567890123456789012', 500, 'TRANSFERENCIA', SYSDATE,
SYSTIMESTAMP);

INSERT INTO Transferencia (codigo_numerico, iban_emisor, iban_receptor)
VALUES (101, 'ES1234567890123456789012', 'ESCORRIENTE1234567890123');

COMMIT;

-- ----------------------------
-- Test A2: Transferencia con saldo insuficiente (debe fallar)
-- ----------------------------
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (102, 'ES1234567890123456789012', 600, 'TRANSFERENCIA', SYSDATE,
SYSTIMESTAMP);

BEGIN
   INSERT INTO Transferencia (codigo_numerico, iban_emisor,
iban_receptor)
   VALUES (102, 'ES1234567890123456789012',
'ESCORRIENTE1234567890123');
EXCEPTION
   WHEN OTHERS THEN
       DBMS_OUTPUT.put_line('Error detectado (saldo insuficiente): ' ||
SQLERRM);
```

```
END;
/

ROLLBACK;

-- ----------------------------
-- Test E: Transferencia a la misma cuenta (debe fallar)
-- ----------------------------
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (103, 'ESCORRIENTE1234567890123', 100, 'TRANSFERENCIA', SYSDATE,
SYSTIMESTAMP);

BEGIN
   INSERT INTO Transferencia (codigo_numerico, iban_emisor,
iban_receptor)
   VALUES (103, 'ESCORRIENTE1234567890123',
'ESCORRIENTE1234567890123');
EXCEPTION
   WHEN OTHERS THEN
       DBMS_OUTPUT.put_line('Error detectado (transferencia a la misma
cuenta): ' || SQLERRM);
END;
/

ROLLBACK;

-- ----------------------------
-- Test B: Ingreso (actualiza saldo +300)
-- ----------------------------
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (201, 'ES1234567890123456789012', 300, 'INGRESO', SYSDATE,
SYSTIMESTAMP);

INSERT INTO Ingreso (codigo_numerico, codigo_sucursal, iban)
VALUES (201, 1, 'ES1234567890123456789012');

COMMIT;

-- ----------------------------
-- Test C: Retirada (actualiza saldo -500)
-- ----------------------------
```

```
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (301, 'ESAHORRO1234567890123456', 500, 'RETIRO', SYSDATE,
SYSTIMESTAMP);

INSERT INTO Retirada (codigo_numerico, codigo_sucursal, iban)
VALUES (301, 1, 'ESAHORRO1234567890123456');

COMMIT;

-- ----------------------------
-- Test D: Eliminar cliente (borrado en CuentaCliente)
-- ----------------------------
BEGIN
   DELETE FROM Cliente WHERE dni = 1001;
   COMMIT;
EXCEPTION
   WHEN OTHERS THEN
       DBMS_OUTPUT.put_line('Error al eliminar el cliente: ' ||
SQLERRM);
END;
/

PROMPT "Inserciones de prueba completadas. Revisa los
resultados/errores en la salida."

EXIT;
EOF
```

Se ha usado este script para crear y probar los triggers descritos al final del documento.

Diseño Lógico de una Base de Datos Objeto/Relacional

PostgreSQL

```
#!/bin/bash
# Script: docker_postgres_create_tables.sh
# Objetivo: Borrar todas las tablas existentes, crear el esquema
bancario en PostgreSQL usando herencia,
```

```
#           e insertar datos de ejemplo de forma idempotente.
# Se asume que el contenedor tiene psql disponible.

CONTAINER_NAME="p2_postgres_1"  # Contenedor de PostgreSQL
DATABASE="p2"                   # Nombre de la base de datos
ADMIN_USER="postgres"           # Usuario administrador de PostgreSQL

echo "Ejecutando script dentro del contenedor '$CONTAINER_NAME' para
borrar y crear tablas en PostgreSQL..."

# Crear la base de datos si no existe
echo "Creando base de datos '$DATABASE' (si no existe)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -c "SELECT 'CREATE
DATABASE $DATABASE' WHERE NOT EXISTS (SELECT FROM pg_database WHERE
datname = '$DATABASE')\gexec;"
echo "Base de datos '$DATABASE' creada (o ya existía)."

# Borrar todas las tablas existentes (ordenado para evitar
dependencias)
echo "Borrando todas las tablas existentes..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
DROP TABLE IF EXISTS transferencia CASCADE;
DROP TABLE IF EXISTS ingreso CASCADE;
DROP TABLE IF EXISTS retirada CASCADE;
DROP TABLE IF EXISTS operacion_bancaria CASCADE;
DROP TABLE IF EXISTS cuenta_corriente CASCADE;
DROP TABLE IF EXISTS cuenta_ahorro CASCADE;
DROP TABLE IF EXISTS cuenta_cliente CASCADE;
DROP TABLE IF EXISTS cuenta CASCADE;
DROP TABLE IF EXISTS cliente CASCADE;
DROP TABLE IF EXISTS sucursal CASCADE;
"
echo "Todas las tablas borradas."

# Crear las tablas con herencia
echo "Creando tablas bancarias..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
-- Tabla sucursal
CREATE TABLE IF NOT EXISTS sucursal (
 codigo_sucursal INT PRIMARY KEY,
 direccion       VARCHAR(100),
 telefono        VARCHAR(20)
);
```

```
-- Tabla cliente
CREATE TABLE IF NOT EXISTS cliente (
 email       VARCHAR(50) PRIMARY KEY,
 nombre      VARCHAR(50),
 apellidos   VARCHAR(50),
 edad        INT,
 telefono    VARCHAR(15),
 direccion   VARCHAR(100)
);

-- Tabla base para cuenta
CREATE TABLE IF NOT EXISTS cuenta (
 iban           VARCHAR(34) PRIMARY KEY,
 numero_cuenta  VARCHAR(50),
 saldo          NUMERIC(15,2),
 fecha_creacion DATE
);

-- Tablas derivadas de cuenta usando herencia
CREATE TABLE IF NOT EXISTS cuenta_ahorro (
 interes       NUMERIC(5,2)
) INHERITS (cuenta);

CREATE TABLE IF NOT EXISTS cuenta_corriente (
 codigo_sucursal INT,
 FOREIGN KEY (codigo_sucursal) REFERENCES sucursal(codigo_sucursal)
) INHERITS (cuenta);

-- Tabla intermedia para relacionar cliente y cuenta
CREATE TABLE IF NOT EXISTS cuenta_cliente (
 email VARCHAR(50),
 iban  VARCHAR(34),
 PRIMARY KEY (email, iban),
 FOREIGN KEY (email) REFERENCES cliente(email),
 FOREIGN KEY (iban) REFERENCES cuenta(iban)
);

-- Tabla base para operaciones bancarias
CREATE TABLE IF NOT EXISTS operacion_bancaria (
 codigo_numerico INT PRIMARY KEY,
 iban            VARCHAR(34),
 tipo            VARCHAR(20),
```

```
 cantidad        NUMERIC(15,2),
 fecha           DATE,
 hora            TIME,
 FOREIGN KEY (iban) REFERENCES cuenta(iban)
);

-- Tablas derivadas de operacion_bancaria usando herencia
CREATE TABLE IF NOT EXISTS transferencia (
 iban_destinatario VARCHAR(34),
 PRIMARY KEY (codigo_numerico, iban, iban_destinatario),
 FOREIGN KEY (iban_destinatario) REFERENCES cuenta(iban)
) INHERITS (operacion_bancaria);

CREATE TABLE IF NOT EXISTS retirada (
 codigo_sucursal INT,
 PRIMARY KEY (codigo_numerico, iban, codigo_sucursal),
 FOREIGN KEY (codigo_sucursal) REFERENCES sucursal(codigo_sucursal)
) INHERITS (operacion_bancaria);

CREATE TABLE IF NOT EXISTS ingreso (
 codigo_sucursal INT,
 PRIMARY KEY (codigo_numerico, iban, codigo_sucursal),
 FOREIGN KEY (codigo_sucursal) REFERENCES sucursal(codigo_sucursal)
) INHERITS (operacion_bancaria);
"
echo "Tablas creadas (o ya existían)."

# Insertar datos de ejemplo de forma idempotente
echo "Insertando datos de ejemplo en las tablas (si están vacías)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
-- Insertar un cliente de ejemplo
INSERT INTO cliente (email, nombre, apellidos, edad, telefono,
direccion)
SELECT 'ejemplo@cliente.com', 'Juan', 'Pérez', 30, '123456789', 'Calle
Falsa 123'
WHERE NOT EXISTS (SELECT 1 FROM cliente WHERE email =
'ejemplo@cliente.com');

-- Insertar una sucursal de ejemplo
INSERT INTO sucursal (codigo_sucursal, direccion, telefono)
SELECT 1, 'Sucursal Central', '987654321'
WHERE NOT EXISTS (SELECT 1 FROM sucursal WHERE codigo_sucursal = 1);
```

```
-- Insertar una cuenta base de ejemplo
INSERT INTO cuenta (iban, numero_cuenta, saldo, fecha_creacion)
SELECT 'ES1234567890123456789012', '1234567890', 1000.00, CURRENT_DATE
WHERE NOT EXISTS (SELECT 1 FROM cuenta WHERE iban =
'ES1234567890123456789012');

-- Insertar una cuenta de ahorro derivada
INSERT INTO cuenta_ahorro (iban, numero_cuenta, saldo, fecha_creacion,
interes)
SELECT 'ESAHORRO1234567890123456', 'AHO123456', 2000.00, CURRENT_DATE,
1.50
WHERE NOT EXISTS (SELECT 1 FROM cuenta WHERE iban =
'ESAHORRO1234567890123456');

-- Insertar una cuenta corriente derivada
INSERT INTO cuenta_corriente (iban, numero_cuenta, saldo,
fecha_creacion, codigo_sucursal)
SELECT 'ESCORRIENTE1234567890123', 'COR123456', 1500.00, CURRENT_DATE,
1
WHERE NOT EXISTS (SELECT 1 FROM cuenta WHERE iban =
'ESCORRIENTE1234567890123');

-- Insertar relación cliente-cuenta
INSERT INTO cuenta_cliente (email, iban)
SELECT 'ejemplo@cliente.com', 'ES1234567890123456789012'
WHERE NOT EXISTS (SELECT 1 FROM cuenta_cliente WHERE email =
'ejemplo@cliente.com' AND iban = 'ES1234567890123456789012');
"
echo "Datos insertados (o ya estaban presentes)."

echo "Script docker_postgres_create_tables.sh completado."
```

Para transformar el modelo relacional original en un esquema objeto-relacional en PostgreSQL se ha adoptado una estrategia basada en la herencia de tablas, aprovechando las capacidades que ofrece este sistema gestor para simular la orientación a objetos. En primer lugar, se define una tabla base o superentidad, como la tabla cuenta, que agrupa los atributos comunes a todas las cuentas (IBAN, número de cuenta, saldo y fecha de creación), y de forma análoga se establece la tabla operacion_bancaria para representar las transacciones, en la cual se incluyen atributos generales como código numérico, IBAN, tipo, cantidad, fecha y hora. A partir de estas tablas base se crean tablas derivadas mediante la

cláusula INHERITS, lo que permite que las subclases hereden la estructura de la superclase y, al mismo tiempo, puedan incorporar atributos específicos. Así, la tabla cuenta_ahorro se crea heredando de cuenta y se le añade el atributo “interés”, mientras que la tabla cuenta_corriente, también derivada de cuenta, incorpora el campo “codigo_sucursal” para establecer la vinculación obligatoria con una sucursal. De forma similar, en la jerarquía de operaciones se han definido las tablas transferencia, retirada e ingreso, que heredan de operacion_bancaria y agregan atributos particulares (por ejemplo, en la transferencia se añade el IBAN del destinatario, y en retirada e ingreso se incorpora el código de sucursal) para diferenciar los distintos tipos de transacciones. Adicionalmente, se ha implementado una tabla intermedia, cuenta_cliente, para gestionar la relación muchos a muchos entre cliente y cuenta, permitiendo que un cliente posea varias cuentas y que una cuenta pueda estar asociada a múltiples clientes. Una de las limitaciones propias de PostgreSQL es que las restricciones definidas en una tabla padre, como las claves primarias o restricciones únicas, no se heredan de forma automática en las tablas hijas; por ello, fue necesario declarar estas restricciones explícitamente en cada tabla derivada para asegurar la integridad referencial. Asimismo, se debe prestar especial atención en las operaciones de consulta, inserción y actualización, ya que las acciones realizadas sobre la tabla base pueden no propagarse de forma automática a las tablas hijas.

En PostgreSQL se prefiere no usar tablas tipadas en combinación con la herencia debido a que el mecanismo de herencia de tablas ya permite extender estructuras y reutilizar definiciones de columnas sin recurrir a tipos compuestos. Además, la herencia en PostgreSQL presenta limitaciones importantes: las restricciones y claves definidas en la tabla padre no se heredan automáticamente en las tablas hijas, lo que implica que, si se emplean tablas tipadas, habría que replicar manualmente dichas restricciones en cada subtabla, incrementando la complejidad y el riesgo de inconsistencias. Por lo tanto hemos decidido usar solo la herencia para implementar nuestro modelo.

```
#!/bin/bash
# Script: docker_postgres_create_triggers.sh
# Objetivo:
#  - Eliminar triggers y funciones existentes (si existen).
#  - Crear triggers adaptados a PostgreSQL para validar:
#       • Que haya saldo suficiente antes de una transferencia.
#       • Que se actualice el saldo después de un ingreso o retirada.
#       • Que se eliminen las relaciones de cliente con cuenta al
borrar un cliente.
#       • Que no se pueda realizar una transferencia a la misma cuenta.
#       • Que el campo 'tipo' (extraído de operacion_bancaria) coincida
con el tipo de operación
#         esperado al insertar en las tablas derivadas (ingreso,
transferencia, retirada).
#
#  - Ejecutar inserts de prueba para comprobar el funcionamiento de los
triggers.
```

```
#
# Se asume que:
#   • El contenedor de Postgres se llama "eb099d90c2ae".
#   • La base de datos se llama "p2".
#   • El usuario administrador es "postgres".

CONTAINER_NAME="p2_postgres_1"  # Nombre del contenedor de PostgreSQL
DATABASE="p2"                   # Nombre de la base de datos
ADMIN_USER="postgres"           # Usuario administrador

echo "Eliminando triggers y funciones existentes (si existen)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
DROP TRIGGER IF EXISTS check_balance_before_transfer ON transferencia;
DROP FUNCTION IF EXISTS check_balance_before_transfer_func();

DROP TRIGGER IF EXISTS update_balance_after_ingreso ON ingreso;
DROP FUNCTION IF EXISTS update_balance_after_ingreso_func();

DROP TRIGGER IF EXISTS update_balance_after_retiro ON retirada;
DROP FUNCTION IF EXISTS update_balance_after_retiro_func();

DROP TRIGGER IF EXISTS delete_cliente_cuentas ON cliente;
DROP FUNCTION IF EXISTS delete_cliente_cuentas_func();

DROP TRIGGER IF EXISTS check_transferencia_same_account ON
transferencia;
DROP FUNCTION IF EXISTS check_transferencia_same_account_func();

DROP TRIGGER IF EXISTS check_tipo_operacion_ingreso ON ingreso;
DROP FUNCTION IF EXISTS check_tipo_operacion_ingreso_func();

DROP TRIGGER IF EXISTS check_tipo_operacion_transferencia ON
transferencia;
DROP FUNCTION IF EXISTS check_tipo_operacion_transferencia_func();

DROP TRIGGER IF EXISTS check_tipo_operacion_retiro ON retirada;
DROP FUNCTION IF EXISTS check_tipo_operacion_retiro_func();
"

echo "Creando funciones y triggers..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
-- Función y trigger para validar el saldo antes de una transferencia
```

```
CREATE OR REPLACE FUNCTION check_balance_before_transfer_func() RETURNS
trigger AS \$\$
DECLARE
   v_saldo NUMERIC;
BEGIN
   SELECT saldo INTO v_saldo FROM cuenta WHERE iban = NEW.iban;
   IF v_saldo < NEW.cantidad THEN
       RAISE EXCEPTION 'Saldo insuficiente para realizar la
transferencia.';
   END IF;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER check_balance_before_transfer
BEFORE INSERT ON transferencia
FOR EACH ROW
EXECUTE FUNCTION check_balance_before_transfer_func();

-- Función y trigger para actualizar el saldo después de un ingreso
CREATE OR REPLACE FUNCTION update_balance_after_ingreso_func() RETURNS
trigger AS \$\$
BEGIN
   UPDATE cuenta SET saldo = saldo + NEW.cantidad WHERE iban =
NEW.iban;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER update_balance_after_ingreso
AFTER INSERT ON ingreso
FOR EACH ROW
EXECUTE FUNCTION update_balance_after_ingreso_func();

-- Función y trigger para actualizar el saldo después de una retirada
CREATE OR REPLACE FUNCTION update_balance_after_retiro_func() RETURNS
trigger AS \$\$
BEGIN
   UPDATE cuenta SET saldo = saldo - NEW.cantidad WHERE iban =
NEW.iban;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER update_balance_after_retiro
AFTER INSERT ON retirada
FOR EACH ROW
EXECUTE FUNCTION update_balance_after_retiro_func();

-- Función y trigger para eliminar las relaciones de cliente con cuenta
al borrar un cliente
CREATE OR REPLACE FUNCTION delete_cliente_cuentas_func() RETURNS
trigger AS \$\$
BEGIN
   DELETE FROM cuenta_cliente WHERE email = OLD.email;
   RETURN OLD;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER delete_cliente_cuentas
AFTER DELETE ON cliente
FOR EACH ROW
EXECUTE FUNCTION delete_cliente_cuentas_func();

-- Función y trigger para evitar transferencias a la misma cuenta
CREATE OR REPLACE FUNCTION check_transferencia_same_account_func()
RETURNS trigger AS \$\$
BEGIN
   IF NEW.iban = NEW.iban_destinatario THEN
       RAISE EXCEPTION 'No se puede realizar una transferencia a la
misma cuenta.';
   END IF;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER check_transferencia_same_account
BEFORE INSERT ON transferencia
FOR EACH ROW
EXECUTE FUNCTION check_transferencia_same_account_func();

-- Función y trigger para validar que el codigo_numerico y el iban
correspondan a una operación de INGRESO
CREATE OR REPLACE FUNCTION check_tipo_operacion_ingreso_func() RETURNS
trigger AS \$\$
DECLARE
```

```
   v_tipo TEXT;
BEGIN
   SELECT tipo INTO v_tipo FROM operacion_bancaria
    WHERE codigo_numerico = NEW.codigo_numerico AND iban = NEW.iban;
   IF v_tipo <> 'INGRESO' THEN
       RAISE EXCEPTION 'El codigo_numerico y el iban no están asociados
a un tipo de operación \"INGRESO\".';
   END IF;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER check_tipo_operacion_ingreso
BEFORE INSERT ON ingreso
FOR EACH ROW
EXECUTE FUNCTION check_tipo_operacion_ingreso_func();

-- Función y trigger para validar que el codigo_numerico y el iban
correspondan a una operación de TRANSFERENCIA
CREATE OR REPLACE FUNCTION check_tipo_operacion_transferencia_func()
RETURNS trigger AS \$\$
DECLARE
   v_tipo TEXT;
BEGIN
   SELECT tipo INTO v_tipo FROM operacion_bancaria
    WHERE codigo_numerico = NEW.codigo_numerico AND iban = NEW.iban;
   IF v_tipo <> 'TRANSFERENCIA' THEN
       RAISE EXCEPTION 'El codigo_numerico y el iban no están asociados
a un tipo de operación \"TRANSFERENCIA\".';
   END IF;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER check_tipo_operacion_transferencia
BEFORE INSERT ON transferencia
FOR EACH ROW
EXECUTE FUNCTION check_tipo_operacion_transferencia_func();

-- Función y trigger para validar que el codigo_numerico y el iban
correspondan a una operación de RETIRO
CREATE OR REPLACE FUNCTION check_tipo_operacion_retiro_func() RETURNS
trigger AS \$\$
```

```
DECLARE
   v_tipo TEXT;
BEGIN
   SELECT tipo INTO v_tipo FROM operacion_bancaria
    WHERE codigo_numerico = NEW.codigo_numerico AND iban = NEW.iban;
   IF v_tipo <> 'RETIRO' THEN
       RAISE EXCEPTION 'El codigo_numerico y el iban no están asociados
a un tipo de operación \"RETIRO\".';
   END IF;
   RETURN NEW;
END;
\$\$ LANGUAGE plpgsql;

CREATE TRIGGER check_tipo_operacion_retiro
BEFORE INSERT ON retirada
FOR EACH ROW
EXECUTE FUNCTION check_tipo_operacion_retiro_func();
"

echo "Triggers y funciones creados exitosamente en la base de datos
'$DATABASE'."

echo "Insertando datos de prueba para verificar el funcionamiento de
los triggers..."

#--------------------------------------------------------------------
--------
# Nota:
# Se asume que el script de creación de tablas e inserción de datos
base ya
# creó:
#   • La cuenta 'ES1234567890123456789012' con saldo 1000.00.
#   • La cuenta de ahorro 'ESAHORRO1234567890123456' con saldo 2000.00.
#   • La cuenta corriente 'ESCORRIENTE1234567890123' con saldo 1500.00.
#   • Un cliente con email 'ejemplo@cliente.com' y la relación en
cuenta_cliente.
#--------------------------------------------------------------------
--------

# Test A1: Insertar una transferencia válida (suficiente saldo y
cuentas distintas)
echo "Test A1: Insertar transferencia válida (saldo suficiente)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
```

```
INSERT INTO transferencia (codigo_numerico, iban, tipo, cantidad,
fecha, hora, iban_destinatario)
VALUES (101, 'ES1234567890123456789012', 'TRANSFERENCIA', 500.00,
CURRENT_DATE, CURRENT_TIME, 'ESCORRIENTE1234567890123');
"

# Test A2: Intentar insertar una transferencia con saldo insuficiente
(debe fallar)
echo "Test A2: Insertar transferencia con saldo insuficiente (debe
fallar)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
INSERT INTO transferencia (codigo_numerico, iban, tipo, cantidad,
fecha, hora, iban_destinatario)
VALUES (102, 'ES1234567890123456789012', 'TRANSFERENCIA', 600.00,
CURRENT_DATE, CURRENT_TIME, 'ESCORRIENTE1234567890123');
"

# Test E: Intentar insertar una transferencia a la misma cuenta (debe
fallar)
echo "Test E: Insertar transferencia a la misma cuenta (debe
fallar)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
INSERT INTO transferencia (codigo_numerico, iban, tipo, cantidad,
fecha, hora, iban_destinatario)
VALUES (103, 'ESCORRIENTE1234567890123', 'TRANSFERENCIA', 100.00,
CURRENT_DATE, CURRENT_TIME, 'ESCORRIENTE1234567890123');
"

# Test B: Insertar un ingreso para actualizar el saldo (cuenta
'ES1234567890123456789012')
echo "Test B: Insertar ingreso para actualizar saldo..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
INSERT INTO ingreso (codigo_numerico, iban, tipo, cantidad, fecha,
hora, codigo_sucursal)
VALUES (201, 'ES1234567890123456789012', 'INGRESO', 300.00,
CURRENT_DATE, CURRENT_TIME, 1);
"

# Test C: Insertar una retirada para actualizar el saldo (cuenta
'ESAHORRO1234567890123456')
echo "Test C: Insertar retirada para actualizar saldo..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
```

```
INSERT INTO retirada (codigo_numerico, iban, tipo, cantidad, fecha,
hora, codigo_sucursal)
VALUES (301, 'ESAHORRO1234567890123456', 'RETIRO', 500.00,
CURRENT_DATE, CURRENT_TIME, 1);
"

# Test D: Eliminar el cliente y comprobar que se borran las relaciones
en cuenta_cliente
echo "Test D: Eliminar cliente 'ejemplo@cliente.com' (se deben borrar
sus relaciones en cuenta_cliente)..."
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c "
DELETE FROM cliente WHERE email = 'ejemplo@cliente.com';
"

echo "Inserciones de prueba completadas. Verifica los resultados
(errores y actualizaciones de saldo) en los logs del contenedor."
```

Estos han sido los triggers implementados para mantener la integridad. Por ejemplo, el trigger que valida el saldo antes de una transferencia (check_balance_before_transfer) se ejecuta antes de insertar un registro en la tabla transferencia; la función asociada consulta el saldo actual de la cuenta emisora en la tabla cuenta y, si éste es insuficiente para cubrir la cantidad de la transferencia, lanza una excepción que impide la operación. En paralelo, se han implementado triggers para actualizar el saldo de las cuentas: el trigger update_balance_after_ingreso se dispara después de insertar un ingreso, sumando el monto correspondiente al saldo de la cuenta, mientras que el trigger update_balance_after_retiro se activa tras una retirada, restando el monto respectivo. Además, se ha definido un trigger (delete_cliente_cuentas) que se ejecuta después de eliminar un cliente, garantizando que las relaciones existentes en la tabla intermedia cuenta_cliente se eliminen, evitando así registros huérfanos. Otro trigger importante, check_transferencia_same_account, previene que se realice una transferencia a la misma cuenta, comprobando que el IBAN de la cuenta emisora no coincida con el IBAN del destinatario y lanzando una excepción en caso contrario. Finalmente, se han creado triggers adicionales (check_tipo_operacion_ingreso, check_tipo_operacion_transferencia y check_tipo_operacion_retiro) que se ejecutan antes de insertar registros en las tablas ingreso, transferencia y retirada, respectivamente. Estas funciones verifican que el código numérico e IBAN del registro coincidan con un registro existente en la tabla operacion_bancaria y que el campo 'tipo' de dicha tabla se corresponda con el tipo de operación esperado (ya sea 'INGRESO', 'TRANSFERENCIA' o 'RETIRO'), rechazando la inserción si la validación falla.

Oracle

```
#!/usr/bin/env bash

docker exec -i p2-oracle2-1 sqlplus admin/admin@//localhost:1521/XEPDB1
<<SQL

-- 1️⃣ Crear Tipos de Objetos

-- Crear el tipo de objeto Cliente
CREATE OR REPLACE TYPE ClienteUDT AS OBJECT (
   nombre          VARCHAR2(50),
   apellidos       VARCHAR2(50),
   edad            NUMBER(3),
   telefono        VARCHAR2(20),
   email           VARCHAR2(100),
   direccion       VARCHAR2(100),
   dni             VARCHAR2(20)
);
/

-- Crear el tipo de objeto Cuenta (superentidad, NOT FINAL para
permitir subtipos)
CREATE OR REPLACE TYPE CuentaUDT AS OBJECT (
   numero_cuenta   VARCHAR2(20),
   iban            VARCHAR2(34),
   saldo           NUMBER(15,2),
   fecha_creacion  DATE
) NOT FINAL;
/

-- Crear el tipo de objeto OperacionBancaria (superentidad, NOT FINAL)
CREATE OR REPLACE TYPE OperacionBancariaUDT AS OBJECT (
   codigo_numerico    NUMBER(10),
   fecha           DATE,
   cantidad           NUMBER(15,2),
   tipo_operacion  VARCHAR2(20),  -- Para diferenciar el tipo de
operación (Transferencia, Retirada, Ingreso)
   descripcion     CLOB,
   iban            VARCHAR2(34)  -- Cuenta sobre la que se hace la
operación
) NOT FINAL;
/
```

```
-- Crear el tipo de objeto Sucursal
CREATE OR REPLACE TYPE SucursalUDT AS OBJECT (
   codigo_sucursal NUMBER(10),
   direccion       VARCHAR2(100),
   telefono        VARCHAR2(20)
);
/

-- Crear el tipo de objeto Transferencia (subentidad de
OperacionBancaria)
CREATE OR REPLACE TYPE TransferenciaUDT UNDER OperacionBancariaUDT (
   iban_receptor   VARCHAR2(34)   -- Cuenta receptora
);
/

-- Crear el tipo de objeto Retirada (subentidad de OperacionBancaria)
CREATE OR REPLACE TYPE RetiradaUDT UNDER OperacionBancariaUDT (
   codigo_sucursal NUMBER(10)
);
/

-- Crear el tipo de objeto Ingreso (subentidad de OperacionBancaria)
CREATE OR REPLACE TYPE IngresoUDT UNDER OperacionBancariaUDT (
   codigo_sucursal NUMBER(10)
);
/

-- Crear el tipo de objeto CuentaAhorro (subentidad de Cuenta)
CREATE OR REPLACE TYPE CuentaAhorroUDT UNDER CuentaUDT (
   interes         NUMBER(10,2)  -- Atributo específico para
CuentaAhorro
);
/

-- Crear el tipo de objeto CuentaCorriente (subentidad de Cuenta)
CREATE OR REPLACE TYPE CuentaCorrienteUDT UNDER CuentaUDT (
   codigo_sucursal NUMBER(10)
);
/

-- 2️⃣ Crear las Tablas Basadas en los Tipos de Objetos
```

```
-- Tabla para Clientes
CREATE TABLE Clientes OF ClienteUDT (
   CONSTRAINT pk_cliente PRIMARY KEY (dni)
);
/

-- Tabla para Sucursales (debe crearse antes de Operaciones Bancarias)
CREATE TABLE Sucursales OF SucursalUDT (
   CONSTRAINT pk_sucursal PRIMARY KEY (codigo_sucursal)
);
/

-- Tabla para Cuentas
CREATE TABLE Cuentas OF CuentaUDT (
   CONSTRAINT pk_cuenta PRIMARY KEY (iban)
);
/

-- Tabla para Operaciones Bancarias (superentidad)
CREATE TABLE OperacionesBancarias OF OperacionBancariaUDT (
   CONSTRAINT pk_operacionbancaria PRIMARY KEY (codigo_numerico,iban),
   CONSTRAINT fk_operacionbancaria_cuenta
       FOREIGN KEY (iban) REFERENCES Cuentas (iban)
);
/

-- Tabla para Transferencias (subentidad de OperacionBancaria)
CREATE TABLE Transferencias OF TransferenciaUDT (
   CONSTRAINT pk_transferencia PRIMARY KEY (codigo_numerico, iban,
iban_receptor),  -- Clave primaria compuesta
   CONSTRAINT fk_transferencia_operacion FOREIGN KEY (codigo_numerico,
iban)
       REFERENCES OperacionesBancarias (codigo_numerico, iban),  --
Relación correcta
   CONSTRAINT fk_transferencia_receptor FOREIGN KEY (iban_receptor)
REFERENCES Cuentas (iban)
);
/



-- Tabla para Retiradas (subentidad de OperacionBancaria)
CREATE TABLE Retiradas OF RetiradaUDT (
```

```
   CONSTRAINT pk_retirada PRIMARY KEY
(codigo_numerico,iban,codigo_sucursal),
   CONSTRAINT fk_retirada_operacion FOREIGN KEY (codigo_numerico,iban)
REFERENCES OperacionesBancarias (codigo_numerico,iban),
   CONSTRAINT fk_retirada_sucursal FOREIGN KEY (codigo_sucursal)
REFERENCES Sucursales (codigo_sucursal)
);
/

-- Tabla para Ingresos (subentidad de OperacionBancaria)
CREATE TABLE Ingresos OF IngresoUDT (
   CONSTRAINT pk_ingreso PRIMARY KEY
(codigo_numerico,iban,codigo_sucursal),
   CONSTRAINT fk_ingreso_operacion FOREIGN KEY (codigo_numerico,iban)
REFERENCES OperacionesBancarias (codigo_numerico,iban),
   CONSTRAINT fk_ingreso_sucursal FOREIGN KEY (codigo_sucursal)
REFERENCES Sucursales (codigo_sucursal)
);
/

-- Tabla para CuentaAhorro (subentidad de Cuenta)
CREATE TABLE CuentasAhorro OF CuentaAhorroUDT (
   CONSTRAINT pk_cuenta_ahorro PRIMARY KEY (iban),
   CONSTRAINT fk_cuenta_ahorro FOREIGN KEY (iban) REFERENCES Cuentas
(iban)
);
/

-- Tabla para CuentaCorriente (subentidad de Cuenta)
CREATE TABLE CuentasCorriente OF CuentaCorrienteUDT (
   CONSTRAINT pk_cuenta_corriente PRIMARY KEY (iban),
   CONSTRAINT fk_cuenta_corriente FOREIGN KEY (iban) REFERENCES Cuentas
(iban),
   CONSTRAINT fk_cuenta_corriente_sucursal FOREIGN KEY
(codigo_sucursal) REFERENCES Sucursales (codigo_sucursal)
);
/

-- Tabla intermedia entre Cliente y Cuenta (relación N:M)
CREATE TABLE CuentaCliente (
   dni_cliente   VARCHAR2(20),  -- Referencia al DNI de Cliente
   iban          VARCHAR2(34),  -- Referencia al IBAN de Cuenta
   CONSTRAINT pk_cuentacliente PRIMARY KEY (dni_cliente, iban),
```

```
   CONSTRAINT fk_cuentacliente_cliente FOREIGN KEY (dni_cliente)
REFERENCES Clientes (dni),
   CONSTRAINT fk_cuentacliente_cuenta FOREIGN KEY (iban) REFERENCES
Cuentas (iban)
);
/


-- 4️⃣ Verificación de la Creación de Tipos y Tablas

-- Verifica los tipos creados
SELECT type_name FROM user_types;
/

SQL
```

Para transformar el modelo entidad/relación original al modelo objeto-relacional en Oracle, se ha seguido una estrategia orientada a objetos aprovechando las ventajas que ofrece Oracle para definir tipos abstractos de datos y jerarquías mediante herencia.

En primer lugar, se han creado tipos objeto para representar cada una de las entidades principales del modelo original, como Cliente, Cuenta, Sucursal y Operación Bancaria. Cada tipo objeto encapsula los atributos específicos de su respectiva entidad. Para aquellas entidades que forman parte de una jerarquía, como Cuenta y Operación Bancaria, se han definido como tipos objeto "NOT FINAL" para permitir la creación posterior de subtipos derivados mediante herencia.

Para las jerarquías presentes en el modelo, se han creado subtipos especializados bajo cada supertipo definido previamente con la palabra clave “UNDER” indicando así que hereda de otro tipo previamente creado. Por ejemplo, para la entidad "Cuenta", se han derivado los subtipos "CuentaCorriente" y "CuentaAhorro". Ambos heredan los atributos generales del supertipo "Cuenta" y añaden atributos propios: "interés" en el caso de "CuentaAhorro" y "codigo_sucursal" para "CuentaCorriente", que permite especificar en qué sucursal se gestionan estas cuentas corrientes. Se sigue la misma lógica para la entidad "Operación Bancaria", de la cual se derivan "Transferencia", "Ingreso" y "Retirada". Cada uno de estos subtipos añade atributos específicos ya que en oracle no se permite la especialización de tipos si no añaden ningún atributo extra, así que se ha optado por añadir algún campo que aunque no sea mencionado en el enunciado podría ser útil almacenarlo en la base de datos.

Una vez definidos los tipos objeto y sus respectivas jerarquías, se crean tablas basadas en dichos tipos objeto. Cada tabla hereda la estructura del tipo objeto correspondiente, simplificando así la definición del esquema y asegurando coherencia en los datos. Las tablas relacionadas mediante herencia se vinculan a través de claves primarias compuestas

que coinciden con los atributos heredados, asegurando así integridad referencial y consistencia lógica entre supertipos y subtipos.

Para gestionar la relación muchos a muchos existente entre Cliente y Cuenta, se ha implementado una tabla intermedia llamada "CuentaCliente". Esta tabla almacena la información de las cuentas que posee cada cliente y permite vincular múltiples clientes a una misma cuenta bancaria, así como múltiples cuentas bancarias a un mismo cliente, garantizando flexibilidad en la representación de las relaciones entre ambos tipos objeto.

Finalmente, para garantizar integridad adicional, se han aplicado restricciones de clave primaria y claves foráneas entre las tablas, reforzando la coherencia del modelo objeto-relacional obtenido. Esto garantiza que cada instancia de subtipo se corresponda correctamente con la instancia de su respectivo supertipo, y que todas las relaciones entre objetos estén adecuadamente representadas y controladas por el sistema gestor de bases de datos Oracle.

```
#!/usr/bin/env bash
docker exec -i p2-oracle2-1 sqlplus admin/admin@//localhost:1521/XEPDB1
<<SQL

-- Trigger para validar el saldo en la cuenta antes de realizar una
transferencia
CREATE OR REPLACE TRIGGER check_balance_before_transfer
BEFORE INSERT ON Transferencias
FOR EACH ROW
DECLARE
   v_saldo NUMBER;
BEGIN
   -- Obtener el saldo actual de la cuenta emisora
   SELECT saldo INTO v_saldo
   FROM Cuentas
   WHERE iban = :NEW.iban;

   -- Verificar que haya suficiente saldo
   IF v_saldo < :NEW.cantidad THEN
       RAISE_APPLICATION_ERROR(-20001, 'Saldo insuficiente para
realizar la transferencia.');
   END IF;
END;
/


-- Trigger para actualizar el saldo de la cuenta después de un ingreso
CREATE OR REPLACE TRIGGER update_balance_after_ingreso
AFTER INSERT ON Ingresos
```

```
FOR EACH ROW
BEGIN
   UPDATE Cuentas
   SET saldo = saldo + :NEW.cantidad
   WHERE iban = :NEW.iban;
END;
/

-- Trigger para actualizar el saldo de la cuenta después de un retiro
CREATE OR REPLACE TRIGGER update_balance_after_retiro
AFTER INSERT ON Retiradas
FOR EACH ROW
BEGIN
   UPDATE Cuentas
   SET saldo = saldo - :NEW.cantidad
   WHERE iban = :NEW.iban;
END;
/

-- Trigger para eliminar las relaciones de cliente con cuentas cuando
el cliente se elimina
CREATE OR REPLACE TRIGGER delete_cliente_cuentas
AFTER DELETE ON Clientes
FOR EACH ROW
BEGIN
   DELETE FROM CuentaCliente
   WHERE dni_cliente = :OLD.dni;
END;
/

-- Trigger para evitar transferencias a la misma cuenta
CREATE OR REPLACE TRIGGER check_transferencia_same_account
BEFORE INSERT ON Transferencias
FOR EACH ROW
BEGIN
   -- Verificar que el IBAN de la cuenta emisora no sea igual al IBAN
de la cuenta receptora
   IF :NEW.iban = :NEW.iban_receptor THEN
       RAISE_APPLICATION_ERROR(-20001, 'No se puede realizar una
transferencia a la misma cuenta.');
   END IF;
END;
/
```

```
-- Trigger para asegurar que `codigo_numerico` y `iban` coincidan en
`OperacionesBancarias` antes de insertar en `Ingresos`
CREATE OR REPLACE TRIGGER check_tipo_operacion_ingreso
BEFORE INSERT ON Ingresos
FOR EACH ROW
DECLARE
   v_tipo_operacion VARCHAR2(20);
BEGIN
   -- Verificar que el `codigo_numerico` y `iban` en
`OperacionesBancarias` coincidan con el ingreso
   SELECT tipo_operacion
   INTO v_tipo_operacion
   FROM OperacionesBancarias
   WHERE codigo_numerico = :NEW.codigo_numerico
   AND iban = :NEW.iban;

   -- Si el tipo no es 'INGRESO', lanzar un error
   IF v_tipo_operacion != 'INGRESO' THEN
       RAISE_APPLICATION_ERROR(-20003, 'El codigo_numerico y el iban no
están asociados a un tipo de operación "INGRESO".');
   END IF;
END;
/


-- Trigger para asegurar que `codigo_numerico` y `iban` coincidan en
`OperacionesBancarias` antes de insertar en `Transferencias`
CREATE OR REPLACE TRIGGER check_tipo_operacion_transferencia
BEFORE INSERT ON Transferencias
FOR EACH ROW
DECLARE
   v_tipo_operacion VARCHAR2(20);
BEGIN
   -- Verificar que el `codigo_numerico` y `iban` en
`OperacionesBancarias` coincidan con la transferencia
   SELECT tipo_operacion
   INTO v_tipo_operacion
   FROM OperacionesBancarias
   WHERE codigo_numerico = :NEW.codigo_numerico
   AND iban = :NEW.iban;

   -- Si el tipo no es 'TRANSFERENCIA', lanzar un error
```

```
   IF v_tipo_operacion != 'TRANSFERENCIA' THEN
       RAISE_APPLICATION_ERROR(-20001, 'El codigo_numerico y el iban no
están asociados a un tipo de operación "TRANSFERENCIA".');
   END IF;
END;
/


-- Trigger para asegurar que `codigo_numerico` y `iban` coincidan en
`OperacionesBancarias` antes de insertar en `Retiradas`
CREATE OR REPLACE TRIGGER check_tipo_operacion_retiro
BEFORE INSERT ON Retiradas
FOR EACH ROW
DECLARE
   v_tipo_operacion VARCHAR2(20);
BEGIN
   -- Verificar que el `codigo_numerico` y `iban` en
`OperacionesBancarias` coincidan con la retirada
   SELECT tipo_operacion
   INTO v_tipo_operacion
   FROM OperacionesBancarias
   WHERE codigo_numerico = :NEW.codigo_numerico
   AND iban = :NEW.iban;

   -- Si el tipo no es 'RETIRO', lanzar un error
   IF v_tipo_operacion != 'RETIRO' THEN
       RAISE_APPLICATION_ERROR(-20002, 'El codigo_numerico y el iban no
están asociados a un tipo de operación "RETIRO".');
   END IF;
END;
/




SQL
```

Se han implementado estos triggers para asegurarse de cumplir las restricciones textuales En concreto, el trigger **check_balance_before_transfer** se ejecuta antes de insertar una transferencia y se encarga de validar que el saldo de la cuenta emisora, obtenido desde la

tabla Cuentas, sea suficiente para cubrir la cantidad a transferir, impidiendo así que se realicen transferencias que puedan dejar a la cuenta con saldo negativo. Posteriormente, los triggers **update_balance_after_ingreso** y **update_balance_after_retiro** se activan después de insertar registros en las tablas Ingresos y Retiradas, respectivamente; el primero incrementa el saldo de la cuenta correspondiente al sumar el importe ingresado, mientras que el segundo lo decrementa en función del monto retirado, asegurando que el saldo refleje fielmente las operaciones realizadas. Además, el trigger **delete_cliente_cuentas** se ejecuta tras la eliminación de un cliente en la tabla Clientes y se encarga de borrar todas las relaciones asociadas en la tabla intermedia CuentaCliente, lo que previene la existencia de registros huérfanos y mantiene la integridad referencial. Complementariamente, el trigger **check_transferencia_same_account** evita que se realice una transferencia a la misma cuenta, comparando el IBAN de la cuenta emisora con el IBAN de la cuenta receptora y lanzando un error si estos son iguales. Por último, se han definido tres triggers de validación –**check_tipo_operacion_ingreso**, **check_tipo_operacion_transferencia** y **check_tipo_operacion_retiro**– que se ejecutan antes de insertar registros en las tablas Ingresos, Transferencias y Retiradas, respectivamente. Cada uno de estos triggers consulta la tabla OperacionesBancarias para comprobar que el código numérico e IBAN especificados correspondan al tipo de operación adecuado (ya sea 'INGRESO', 'TRANSFERENCIA' o 'RETIRO') y, de no ser así, se lanza un error que impide la inserción del registro.

IBM DB2

```
#!/bin/bash

# Nombre del contenedor de DB2
CONTAINER_NAME="db2-server"

# Usuario, contraseña y base de datos (según tu docker-compose.yml)
DB_USER="db2inst1"
DB_PASSWORD="admin"
DB_NAME="testdb"

# Ruta dentro del contenedor donde se copiará el script SQL
SQL_SCRIPT_PATH="/tmp/crear_tipos.sql"

echo "Verificando si el contenedor $CONTAINER_NAME está corriendo..."
if ! docker ps | grep -q "$CONTAINER_NAME"; then
   echo "Error: El contenedor $CONTAINER_NAME no está corriendo.
Inícialo con:"
   echo "docker-compose up -d"
   exit 1
fi
```

```
# Crear el script SQL
SQL_SCRIPT=$(cat <<EOF
CONNECT TO $DB_NAME USER $DB_USER USING '$DB_PASSWORD';

CREATE TYPE ClienteUdt AS (
   DNI        VARCHAR(255),
   Edad       INT,
   Nombre     VARCHAR(255),
   Apellidos  VARCHAR(255),
   Email      VARCHAR(255),
   Direccion  VARCHAR(255),
   Telefono   VARCHAR(255)
) INSTANTIABLE NOT FINAL REF USING INTEGER mode db2sql;

CREATE TYPE SucursalUdt AS (
   Codigo                INT,
   Direccion             VARCHAR(255),
   Telefono              INT,
   Cuenta_corriente_IBAN VARCHAR(255)
) INSTANTIABLE NOT FINAL REF USING INTEGER mode db2sql;

-- Tipo padre
CREATE TYPE CuentaUdt AS (
   IBAN VARCHAR(255),
   Numero_Cuenta INT,
   Saldo FLOAT,
   Fecha_Creacion DATE
) INSTANTIABLE NOT FINAL REF USING INTEGER mode db2sql;

-- Hijos con herencia
CREATE TYPE CuentaAhorroUdt UNDER CuentaUdt AS (
   Interes FLOAT
) INSTANTIABLE NOT FINAL mode db2sql;

CREATE TYPE CuentaCorrienteUdt UNDER CuentaUdt AS (
   Sucursal        REF(sucursalUdt)
) INSTANTIABLE NOT FINAL mode db2sql;

-- Tipo padre
CREATE TYPE OperacionBancariaUdt AS (
   Descripcion     VARCHAR(255),
   Codigo_Numerico INT,
   Hora            TIME,
```

```
   Fecha           DATE,
   Cantidad        FLOAT,
   Tipo            VARCHAR(255),
   Cuenta_Emisora  REF(CuentaUdt)
) INSTANTIABLE NOT FINAL REF USING INTEGER mode db2sql;

-- Subtipo Transferencia
CREATE TYPE TransferenciaUdt UNDER OperacionBancariaUdt AS (
   Cuenta_Receptor REF(CuentaUdt)
) INSTANTIABLE NOT FINAL mode db2sql;

-- Subtipo Retirada
CREATE TYPE RetiradaUdt UNDER OperacionBancariaUdt AS (
   Sucursal        REF(sucursalUdt)
) INSTANTIABLE NOT FINAL mode db2sql;

-- Subtipo Ingreso
CREATE TYPE IngresoUdt UNDER OperacionBancariaUdt AS (
   Sucursal        REF(sucursalUdt)
) INSTANTIABLE NOT FINAL mode db2sql;

CREATE TYPE TieneUdt AS (
   DNI     REF(ClienteUdt),
   IBAN    REF(CuentaUdt)
) INSTANTIABLE NOT FINAL REF USING INTEGER mode db2sql;

EOF
)

# Guardar el script en un archivo dentro del host
echo "$SQL_SCRIPT" > crear_tipos.sql

# Copiar el archivo al contenedor DB2
docker cp crear_tipos.sql $CONTAINER_NAME:$SQL_SCRIPT_PATH

# Ejecutar el script dentro del contenedor
docker exec -i $CONTAINER_NAME bash -c "
   su - $DB_USER -c \"
       db2 connect to $DB_NAME user $DB_USER using '$DB_PASSWORD';
       db2 -tvf $SQL_SCRIPT_PATH;
       db2 terminate;
   \"
"
```

```
# Limpiar archivos temporales
rm -f crear_tipos.sql

echo "Creación de tipos y tablas completada en DB2 dentro del
contenedor $CONTAINER_NAME."
```

```
#!/bin/bash

# Nombre del contenedor de DB2
CONTAINER_NAME="db2-server"

# Usuario, contraseña y base de datos (según tu docker-compose.yml)
DB_USER="db2inst1"
DB_PASSWORD="admin"
DB_NAME="testdb"

# Ruta dentro del contenedor donde se copiará el script SQL
SQL_SCRIPT_PATH="/tmp/crear_tablas.sql"

echo "Verificando si el contenedor $CONTAINER_NAME está corriendo..."
if ! docker ps | grep -q "$CONTAINER_NAME"; then
   echo "Error: El contenedor $CONTAINER_NAME no está corriendo.
Inícialo con:"
   echo "docker-compose up -d"
   exit 1
fi

# Crear el script SQL
SQL_SCRIPT=$(cat <<EOF
CONNECT TO $DB_NAME USER $DB_USER USING '$DB_PASSWORD';

CREATE TABLE Cliente OF ClienteUdt
   (REF IS id_cliente USER GENERATED,
    DNI WITH OPTIONS NOT NULL
   );

CREATE TABLE Sucursal OF SucursalUdt
   (REF IS id_sucursal USER GENERATED,
    Codigo WITH OPTIONS NOT NULL
   );

CREATE TABLE Cuenta OF CuentaUdt
```

```
   (REF IS id_cuenta USER GENERATED,
    IBAN WITH OPTIONS NOT NULL
   );

CREATE TABLE CuentaAhorro OF CuentaAhorroUdt
   UNDER Cuenta
   INHERIT SELECT PRIVILEGES;

CREATE TABLE CuentaCorriente OF CuentaCorrienteUdt
   UNDER Cuenta
   INHERIT SELECT PRIVILEGES (
       Sucursal WITH OPTIONS SCOPE Sucursal
   );


CREATE TABLE Operacion_Bancaria OF OperacionBancariaUdt
   (REF IS id_operacion USER GENERATED,
    Codigo_Numerico WITH OPTIONS NOT NULL,
    Cuenta_Emisora WITH OPTIONS SCOPE Cuenta
   );

CREATE TABLE Transferencia OF TransferenciaUdt
   UNDER Operacion_Bancaria
   INHERIT SELECT PRIVILEGES(
       Cuenta_Receptor WITH OPTIONS SCOPE Cuenta
   );

CREATE TABLE Retirada OF RetiradaUdt
   UNDER Operacion_Bancaria
   INHERIT SELECT PRIVILEGES(
       Sucursal WITH OPTIONS SCOPE Sucursal
   );

CREATE TABLE Ingreso OF IngresoUdt
   UNDER Operacion_Bancaria
   INHERIT SELECT PRIVILEGES(
       Sucursal WITH OPTIONS SCOPE Sucursal
   );

CREATE TABLE Tiene OF TieneUdt
   (REF IS id_tiene USER GENERATED,
    DNI WITH OPTIONS SCOPE Cliente,
    IBAN WITH OPTIONS SCOPE Cuenta
```

```
    );

EOF
)

# Guardar el script en un archivo dentro del host
echo "$SQL_SCRIPT" > crear_tablas.sql

# Copiar el archivo al contenedor DB2
docker cp crear_tablas.sql $CONTAINER_NAME:$SQL_SCRIPT_PATH

# Ejecutar el script dentro del contenedor
docker exec -i $CONTAINER_NAME bash -c "
    su - $DB_USER -c \"
        db2 connect to $DB_NAME user $DB_USER using '$DB_PASSWORD';
        db2 -tvf $SQL_SCRIPT_PATH;
        db2 terminate;
    \"
"

# Limpiar archivos temporales
rm -f crear_tablas.sql

echo "Creación de tipos y tablas completada en DB2 dentro del
contenedor $CONTAINER_NAME."
```

```
#!/bin/bash

# Nombre del contenedor de DB2
CONTAINER_NAME="db2-server"

# Usuario, contraseña y base de datos
DB_USER="db2inst1"
DB_PASSWORD="admin"
DB_NAME="testdb"

# Ruta dentro del contenedor donde se copiará el script SQL
SQL_SCRIPT_PATH="/tmp/borrar_tipos.sql"

echo "Verificando si el contenedor $CONTAINER_NAME está corriendo..."
if ! docker ps | grep -q "$CONTAINER_NAME"; then
```

```
    echo "Error: El contenedor $CONTAINER_NAME no está corriendo.
Inícialo con:"
    echo "docker-compose up -d"
    exit 1
fi

echo "Generando el script SQL para eliminar tablas y tipos..."

# Crear el script SQL para borrar las tablas y tipos
SQL_SCRIPT=$(cat <<EOF
CONNECT TO $DB_NAME USER $DB_USER USING '$DB_PASSWORD';


-- Tipos "hoja"
DROP TYPE TieneUdt;
DROP TYPE IngresoUdt;
DROP TYPE RetiradaUdt;
DROP TYPE TransferenciaUdt;
DROP TYPE CuentaCorrienteUdt;
DROP TYPE CuentaAhorroUdt;

-- Tipos "padre"
DROP TYPE SucursalUdt;
DROP TYPE OperacionBancariaUdt;
DROP TYPE CuentaUdt;
DROP TYPE ClienteUdt;


COMMIT;
EOF
)

# Guardar el script en un archivo dentro del host
echo "$SQL_SCRIPT" > borrar_tipos.sql

# Copiar el archivo al contenedor DB2
docker cp borrar_tipos.sql $CONTAINER_NAME:$SQL_SCRIPT_PATH

# Ejecutar el script dentro del contenedor
docker exec -i $CONTAINER_NAME bash -c "
    su - $DB_USER -c \"
        db2 connect to $DB_NAME user $DB_USER using '$DB_PASSWORD';
        db2 -tvf $SQL_SCRIPT_PATH;
```

```
        db2 terminate;
    \"
"

# Limpiar archivos temporales
rm -f borrar_tipos.sql

echo "✅ Eliminación de tipos y tablas completada en DB2 dentro del
contenedor $CONTAINER_NAME."
```

```
#!/bin/bash

# Nombre del contenedor de DB2
CONTAINER_NAME="db2-server"

# Usuario, contraseña y base de datos
DB_USER="db2inst1"
DB_PASSWORD="admin"
DB_NAME="testdb"

# Ruta dentro del contenedor donde se copiará el script SQL
SQL_SCRIPT_PATH="/tmp/borrar_tablas.sql"

echo "Verificando si el contenedor $CONTAINER_NAME está corriendo..."
if ! docker ps | grep -q "$CONTAINER_NAME"; then
    echo "Error: El contenedor $CONTAINER_NAME no está corriendo.
Inícialo con:"
    echo "docker-compose up -d"
    exit 1
fi

echo "Generando el script SQL para eliminar tablas y tipos..."

# Crear el script SQL para borrar las tablas y tipos
SQL_SCRIPT=$(cat <<EOF
CONNECT TO $DB_NAME USER $DB_USER USING '$DB_PASSWORD';

-- Ahora eliminamos en orden correcto
DROP TABLE IF EXISTS Tiene;
DROP TABLE IF EXISTS Ingreso;
DROP TABLE IF EXISTS Retirada;
DROP TABLE IF EXISTS Transferencia;
DROP TABLE IF EXISTS CuentaAhorro;
```

```
DROP TABLE IF EXISTS CuentaCorriente;

DROP TABLE IF EXISTS Sucursal;
DROP TABLE IF EXISTS Operacion_Bancaria;
DROP TABLE IF EXISTS Cuenta;
DROP TABLE IF EXISTS Cliente;

COMMIT;
EOF
)

# Guardar el script en un archivo dentro del host
echo "$SQL_SCRIPT" > borrar_tablas.sql

# Copiar el archivo al contenedor DB2
docker cp borrar_tablas.sql $CONTAINER_NAME:$SQL_SCRIPT_PATH

# Ejecutar el script dentro del contenedor
docker exec -i $CONTAINER_NAME bash -c "
   su - $DB_USER -c \"
       db2 connect to $DB_NAME user $DB_USER using '$DB_PASSWORD';
       db2 -tvf $SQL_SCRIPT_PATH;
       db2 terminate;
   \"
"

# Limpiar archivos temporales
rm -f borrar_tablas.sql

echo "✅ Eliminación de tipos y tablas completada en DB2 dentro del
contenedor $CONTAINER_NAME."
```

Para transformar el modelo entidad/relación original al modelo objeto-relacional en IBM DB2, se ha seguido una estrategia basada en el uso de tipos definidos por el usuario (UDTs) y tablas tipadas, aprovechando las capacidades de DB2 para modelar jerarquías y herencia. Inicialmente, se han creado tipos objeto mediante `CREATE TYPE` para representar las entidades principales. Estos tipos encapsulan los atributos propios de cada entidad y se han definido con `REF USING INT` y `MODE DB2SQL` para permitir la gestión de referencias.

En el caso de entidades jerárquicas, como Cuenta y Operación Bancaria, se ha utilizado la herencia mediante `UNDER`, creando subtipos como CuentaCorriente y CuentaAhorro, o Transferencia, Ingreso y Retirada. Estos subtipos heredan los atributos del supertipo y definen nuevos atributos específicos, como `interes` o `limite_credito`, ya que DB2

requiere que los subtipos contengan al menos un atributo adicional. En el caso de `limite_credito`, este atributo tuvo que ser creado de forma forzada ya que IBM no permite la creación de tipos heredados que no añadan ningún atributo.

Una vez definidos los tipos, se han creado tablas tipadas con `CREATE TABLE OF` basadas en los UDTs. Las tablas hijas se han vinculado con las tablas padre mediante la cláusula `UNDER`, lo que permite mantener la jerarquía y heredar privilegios con `INHERIT SELECT PRIVILEGES`. Además, se han definido claves primarias y foráneas para garantizar la integridad referencial, aplicando restricciones a través de la sintaxis propia de DB2.

Por último, se ha implementado una tabla intermedia “Tiene” para modelar la relación muchos a muchos entre Cliente y Cuenta, asegurando flexibilidad en la gestión de relaciones. El modelo resultante aprovecha las características objeto-relacionales de DB2 para reflejar de forma precisa la estructura y lógica del modelo ER original.

```
#!/bin/bash



# Nombre del contenedor de DB2

CONTAINER_NAME="db2-server"



# Usuario, contraseña y base de datos (según tu docker-compose.yml)

DB_USER="db2inst1"

DB_PASSWORD="admin"

DB_NAME="testdb"



# Ruta dentro del contenedor donde se copiará el script SQL

SQL_SCRIPT_PATH="/tmp/crear_triggers.sql"



echo "Verificando si el contenedor $CONTAINER_NAME está corriendo..."

if ! docker ps | grep -q "$CONTAINER_NAME"; then

   echo "Error: El contenedor $CONTAINER_NAME no está corriendo.
Inícialo con:"

   echo "docker-compose up -d"
```

```
    exit 1
fi


# Crear el script SQL
SQL_SCRIPT=$(cat <<EOF
CONNECT TO $DB_NAME USER $DB_USER USING '$DB_PASSWORD';


--#SET TERMINATOR @


-- Verifica saldo antes de una transferencia
CREATE TRIGGER check_balance_before_transfer
NO CASCADE BEFORE INSERT ON Transferencia
REFERENCING NEW AS N
FOR EACH ROW
BEGIN ATOMIC
  DECLARE v_saldo DECIMAL(15,2);
  SELECT Saldo INTO v_saldo FROM FINAL TABLE (SELECT * FROM Cuenta
WHERE id_cuenta = N.Cuenta_Emisora);
  IF v_saldo < N.Cantidad THEN
     SIGNAL SQLSTATE '75001' SET MESSAGE_TEXT = 'Saldo insuficiente
para realizar la transferencia.';
  END IF;
END@


-- Suma cantidad al saldo tras un ingreso
CREATE TRIGGER update_balance_after_ingreso
```

```
AFTER INSERT ON Ingreso

REFERENCING NEW AS N

FOR EACH ROW

BEGIN ATOMIC

  UPDATE Cuenta SET Saldo = Saldo + N.Cantidad WHERE id_cuenta =
N.Cuenta_Emisora;

END@



-- Resta cantidad del saldo tras un retiro

CREATE TRIGGER update_balance_after_retiro

AFTER INSERT ON Retirada

REFERENCING NEW AS N

FOR EACH ROW

BEGIN ATOMIC

  UPDATE Cuenta SET Saldo = Saldo - N.Cantidad WHERE id_cuenta =
N.Cuenta_Emisora;

END@



-- Elimina entradas en 'Tiene' cuando se elimina un cliente

CREATE TRIGGER delete_cliente_tiene

AFTER DELETE ON Cliente

REFERENCING OLD AS O

FOR EACH ROW

BEGIN ATOMIC

  DELETE FROM Tiene WHERE DNI = O.id_cliente;

END@
```

```
-- Evita transferencia a la misma cuenta

CREATE TRIGGER check_transferencia_same_account

NO CASCADE BEFORE INSERT ON Transferencia

REFERENCING NEW AS N

FOR EACH ROW

BEGIN ATOMIC

  IF N.Cuenta_Emisora = N.Cuenta_Receptor THEN

     SIGNAL SQLSTATE '75002' SET MESSAGE_TEXT = 'No se puede transferir
a la misma cuenta.';

  END IF;

END@



-- Verifica que el tipo de operación sea INGRESO para tabla Ingreso

CREATE TRIGGER check_tipo_operacion_ingreso

NO CASCADE BEFORE INSERT ON Ingreso

REFERENCING NEW AS N

FOR EACH ROW

BEGIN ATOMIC

  DECLARE v_tipo VARCHAR(255);

  SELECT Tipo INTO v_tipo FROM Operacion_Bancaria WHERE id_operacion =
N.id_operacion;

  IF v_tipo != 'INGRESO' THEN

     SIGNAL SQLSTATE '75003' SET MESSAGE_TEXT = 'La operación no es un
INGRESO.';

  END IF;
```

```
END@


-- Verifica que el tipo sea TRANSFERENCIA para la tabla Transferencia

CREATE TRIGGER check_tipo_operacion_transferencia

NO CASCADE BEFORE INSERT ON Transferencia

REFERENCING NEW AS N

FOR EACH ROW

BEGIN ATOMIC

  DECLARE v_tipo VARCHAR(255);

  SELECT Tipo INTO v_tipo FROM Operacion_Bancaria WHERE id_operacion =
N.id_operacion;

  IF v_tipo != 'TRANSFERENCIA' THEN

     SIGNAL SQLSTATE '75004' SET MESSAGE_TEXT = 'La operación no es una
TRANSFERENCIA.';

  END IF;

END@


-- Verifica que el tipo sea RETIRO para tabla Retirada

CREATE TRIGGER check_tipo_operacion_retiro

NO CASCADE BEFORE INSERT ON Retirada

REFERENCING NEW AS N

FOR EACH ROW

BEGIN ATOMIC

  DECLARE v_tipo VARCHAR(255);

  SELECT Tipo INTO v_tipo FROM Operacion_Bancaria WHERE id_operacion =
N.id_operacion;
```

```
  IF v_tipo != 'RETIRO' THEN

     SIGNAL SQLSTATE '75005' SET MESSAGE_TEXT = 'La operación no es un
RETIRO.';

  END IF;

END@



EOF

)



# Guardar el script en un archivo dentro del host

echo "$SQL_SCRIPT" > crear_triggers.sql



# Copiar el archivo al contenedor DB2

docker cp crear_triggers.sql $CONTAINER_NAME:$SQL_SCRIPT_PATH



# Ejecutar el script dentro del contenedor

docker exec -i $CONTAINER_NAME bash -c "

   su - $DB_USER -c \"

       db2 connect to $DB_NAME user $DB_USER using '$DB_PASSWORD';

       db2 -tvf $SQL_SCRIPT_PATH;

       db2 terminate;

   \"

"



# Limpiar archivos temporales
```

```
rm -f crear_triggers.sql



echo "Creación de triggers completada en DB2 dentro del contenedor
$CONTAINER_NAME."
```

Se han implementado los siguientes triggers para asegurar el cumplimiento las restricciones textuales que sugiere el enunciado de la práctica.
En concreto, el trigger **check_balance_before_transfer** se ejecuta antes de insertar una transferencia y se encarga de validar que el saldo de la cuenta emisora, obtenido desde la tabla Cuenta, sea suficiente para cubrir la cantidad a transferir, impidiendo así que se realicen transferencias que puedan dejar a la cuenta con saldo negativo.
Posteriormente, los triggers **update_balance_after_ingreso** y **update_balance_after_retiro** se activan después de insertar registros en las tablas Ingreso y Retirada, respectivamente; el primero incrementa el saldo de la cuenta correspondiente al sumar el importe ingresado, mientras que el segundo lo decrementa en función del monto retirado, asegurando que el saldo refleje fielmente las operaciones realizadas.
Además, el trigger **delete_cliente_tiene** se ejecuta tras la eliminación de un cliente en la tabla Cliente y se encarga de borrar todas las relaciones asociadas en la tabla intermedia Tiene, lo que previene la existencia de registros huérfanos y mantiene la integridad referencial.
Complementariamente, el trigger **check_transferencia_same_account** evita que se realice una transferencia a la misma cuenta, comparando el IBAN de la cuenta emisora con el IBAN de la cuenta receptora y lanzando un error si estos son iguales.
Por último, se han definido tres triggers de validación: **check_tipo_operacion_ingreso**, **check_tipo_operacion_transferencia** y **check_tipo_operacion_retiro**; que se ejecutan antes de insertar registros en las tablas Ingreso, Transferencia y Retirada, respectivamente. Cada uno de estos triggers consulta la tabla OperacionBancaria para comprobar que el código numérico e IBAN especificados correspondan al tipo de operación adecuado (ya sea 'INGRESO', 'TRANSFERENCIA' o 'RETIRO') y, de no ser así, se lanza un error que impide la inserción del registro.

Problemas encontrados

El principal problema ha ocurrido al crear los trigger necesarios para cumplir las restricciones. Inicialmente, se logró la creación de triggers correctamente, pero fue imposible realizar el poblado. La principal dificultad era la mala gestión de las Oid de las tablas y las referencias SCOPE entre tablas. Una vez corregidos los errores, se consiguió poblar la base de datos, pero por numerosos cambios en la creación de tipos y tablas los triggers dejaron de funcionar debido a numerosos errores de sintaxis (Tokens inesperados). Se han estado realizando numerosas modificaciones con el objetivo de lograr que funcionen correctamente, pero finalmente no ha sido posible corregir los errores, por lo cual no funcionan.

Generación de Datos y Pruebas

Modelo relacional en oracle:

```
#!/usr/bin/env bash

docker exec -i p2-oracle-1 sqlplus admin/admin@//localhost:1521/XEPDB1
<<'EOF'
-- Inserciones para CLIENTE. Se usa la columna dni en lugar de
id_cliente.
INSERT INTO Cliente (dni, nombre, apellidos, edad, telefono, direccion)
VALUES (1, 'Juan', 'Pérez', 35, '555-1234', 'Calle Falsa 123');
INSERT INTO Cliente (dni, nombre, apellidos, edad, telefono, direccion)
VALUES (2, 'María', 'González', 42, '555-5678', 'Avenida Siempre Viva
456');
INSERT INTO Cliente (dni, nombre, apellidos, edad, telefono, direccion)
VALUES (3, 'Pedro', 'López', 29, '555-8765', 'Calle Luna 789');
INSERT INTO Cliente (dni, nombre, apellidos, edad, telefono, direccion)
VALUES (4, 'Ana', 'Martínez', 50, '555-4321', 'Plaza Sol 101');
INSERT INTO Cliente (dni, nombre, apellidos, edad, telefono, direccion)
VALUES (5, 'Luis', 'Ramírez', 33, '555-9876', 'Calle Río 202');

-- Inserciones para CUENTA. Se eliminó el campo tipo_cuenta.
INSERT INTO Cuenta (iban, numero_cuenta, saldo, fecha_creacion)
VALUES ('ES0001', '0001', 1000.00, DATE '2023-01-10');
INSERT INTO Cuenta (iban, numero_cuenta, saldo, fecha_creacion)
VALUES ('ES0002', '0002', 2500.00, DATE '2023-02-15');
INSERT INTO Cuenta (iban, numero_cuenta, saldo, fecha_creacion)
VALUES ('ES0003', '0003', 15000.50, DATE '2022-12-01');
INSERT INTO Cuenta (iban, numero_cuenta, saldo, fecha_creacion)
VALUES ('ES0004', '0004', 200.00, DATE '2023-03-05');
INSERT INTO Cuenta (iban, numero_cuenta, saldo, fecha_creacion)
VALUES ('ES0005', '0005', 3200.75, DATE '2023-03-10');

-- Según el tipo de cuenta, se insertan datos en Ahorro o Corriente.
-- Las cuentas ES0001, ES0003 y ES0005 son de Ahorro con interés del
1.50%
INSERT INTO Ahorro (iban, interes)
VALUES ('ES0001', 1.50);
INSERT INTO Ahorro (iban, interes)
VALUES ('ES0003', 1.50);
INSERT INTO Ahorro (iban, interes)
VALUES ('ES0005', 1.50);
```

```
-- Las cuentas ES0002 y ES0004 son Corriente, se asignan códigos de
sucursal.
INSERT INTO Corriente (iban, codigo_sucursal)
VALUES ('ES0002', 102);
INSERT INTO Corriente (iban, codigo_sucursal)
VALUES ('ES0004', 103);

-- Inserciones para la relación CUENTACLIENTE
-- Se insertan en líneas separadas sin comentarios al final.
INSERT INTO CuentaCliente (dni, iban)
VALUES (1, 'ES0001');
INSERT INTO CuentaCliente (dni, iban)
VALUES (1, 'ES0002');
INSERT INTO CuentaCliente (dni, iban)
VALUES (2, 'ES0002');
INSERT INTO CuentaCliente (dni, iban)
VALUES (3, 'ES0003');
INSERT INTO CuentaCliente (dni, iban)
VALUES (4, 'ES0004');
INSERT INTO CuentaCliente (dni, iban)
VALUES (5, 'ES0005');

-- Inserciones para SUCURSAL
INSERT INTO Sucursal (codigo_sucursal, direccion, telefono)
VALUES (101, 'Calle Principal 123', '555-1000');
INSERT INTO Sucursal (codigo_sucursal, direccion, telefono)
VALUES (102, 'Avenida Central 456', '555-2000');
INSERT INTO Sucursal (codigo_sucursal, direccion, telefono)
VALUES (103, 'Plaza Mayor 789', '555-3000');

-- Inserciones para OPERACIONBANCARIA (operaciones de Retirada e
Ingreso)
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (1001, 'ES0001', 500.00, 'Retirada', DATE '2023-03-15',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (1002, 'ES0002', 1000.00, 'Ingreso', DATE '2023-03-16',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
```

```
VALUES (1003, 'ES0003', 250.00, 'Retirada', DATE '2023-03-17',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (1004, 'ES0002', 100.00, 'Ingreso', DATE '2023-03-18',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (1005, 'ES0004', 75.50, 'Retirada', DATE '2023-03-19',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (1006, 'ES0005', 600.00, 'Ingreso', DATE '2023-03-20',
SYSTIMESTAMP);

-- Inserciones para las operaciones de TRANSFERENCIA.
-- Primero se crean los registros parentales en OperacionBancaria para
las transferencias.
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (2001, 'ES0001', 300.00, 'Transferencia', DATE '2023-03-20',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (2002, 'ES0002', 400.00, 'Transferencia', DATE '2023-03-21',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (2003, 'ES0003', 150.00, 'Transferencia', DATE '2023-03-22',
SYSTIMESTAMP);
INSERT INTO OperacionBancaria (codigo_numerico, iban, cantidad, tipo,
fecha, hora)
VALUES (2004, 'ES0005', 250.00, 'Transferencia', DATE '2023-03-23',
SYSTIMESTAMP);

-- Inserciones para las subentidades.
-- Para RETIRADA: se insertan las operaciones de retirada.
INSERT INTO Retirada (codigo_numerico, codigo_sucursal, iban)
VALUES (1001, 101, 'ES0001');
INSERT INTO Retirada (codigo_numerico, codigo_sucursal, iban)
VALUES (1003, 101, 'ES0003');
INSERT INTO Retirada (codigo_numerico, codigo_sucursal, iban)
VALUES (1005, 101, 'ES0004');
```

```
-- Para INGRESO: se insertan las operaciones de ingreso.
INSERT INTO Ingreso (codigo_numerico, codigo_sucursal, iban)
VALUES (1002, 102, 'ES0002');
INSERT INTO Ingreso (codigo_numerico, codigo_sucursal, iban)
VALUES (1004, 103, 'ES0002');
INSERT INTO Ingreso (codigo_numerico, codigo_sucursal, iban)
VALUES (1006, 102, 'ES0005');

-- Inserciones para TRANSFERENCIA.
-- Ahora se pueden insertar, ya que existen los registros parentales.
INSERT INTO Transferencia (codigo_numerico, iban_emisor, iban_receptor)
VALUES (2001, 'ES0001', 'ES0003');
INSERT INTO Transferencia (codigo_numerico, iban_emisor, iban_receptor)
VALUES (2002, 'ES0002', 'ES0005');
INSERT INTO Transferencia (codigo_numerico, iban_emisor, iban_receptor)
VALUES (2003, 'ES0003', 'ES0004');
INSERT INTO Transferencia (codigo_numerico, iban_emisor, iban_receptor)
VALUES (2004, 'ES0005', 'ES0001');

COMMIT;
EXIT;
EOF
```

Se han probado a insertar estos datos para verificar el correcto funcionamiento del modelo

Modelo objeto relacional en postgres:

```
#!/bin/bash
# Script: docker_postgres_insert_and_read.sh
# Objetivo: Insertar muchas filas en las tablas del esquema bancario
(clientes, sucursales, cuentas,
#           cuentas de ahorro, cuentas corrientes, relaciones
cliente-cuenta y operaciones bancarias)
#           y luego ejecutar SELECTs para comprobar la inserción.
# Se asume que el contenedor tiene psql disponible.
```

```
CONTAINER_NAME="eb099d90c2ae"  # Contenedor de PostgreSQL
DATABASE="p2"                  # Nombre de la base de datos
ADMIN_USER="postgres"          # Usuario administrador de PostgreSQL

echo "Ejecutando script dentro del contenedor '$CONTAINER_NAME' para
insertar y leer datos en PostgreSQL..."

SQL_FILE="/tmp/insert_data.sql"
rm -f $SQL_FILE

# Inserciones masivas para clientes
echo "-- Inserciones para cliente" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
for i in $(seq 1 1000); do
 AGE=$((RANDOM % 60 + 18))
 echo "INSERT INTO cliente (email, nombre, apellidos, edad, telefono,
direccion) VALUES ('cliente${i}@ejemplo.com', 'Nombre${i}',
'Apellido${i}', ${AGE}, '123456789', 'Calle Falsa ${i}');" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE

# Inserciones masivas para sucursales
echo "-- Inserciones para sucursal" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
for i in $(seq 1 1000); do
 echo "INSERT INTO sucursal (codigo_sucursal, direccion, telefono)
VALUES (${i}, 'Sucursal ${i}', '987654321');" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE

# Inserciones masivas para cuentas base (excluyendo las de las tablas
derivadas)
echo "-- Inserciones para cuenta (base)" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
# Usamos el rango 1001-2000 para evitar colisión con registros manuales
for i in $(seq 1001 2000); do
 echo "INSERT INTO cuenta (iban, numero_cuenta, saldo, fecha_creacion)
VALUES ('ES1234567890123456789${i}', '123456789${i}', $((RANDOM %
10000 + 100)), CURRENT_DATE);" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE
```

```
# Inserciones masivas para cuentas de ahorro (tabla derivada:
cuenta_ahorro)
echo "-- Inserciones para cuenta_ahorro" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
# Rango de IBANs diferente (por ejemplo, 2001-2300)
for i in $(seq 2001 2300); do
 echo "INSERT INTO cuenta_ahorro (iban, numero_cuenta, saldo,
fecha_creacion, interes) VALUES ('ESAHORRO${i}', 'AHO${i}', $((RANDOM %
10000 + 100)), CURRENT_DATE, $(echo "scale=2; $((RANDOM % 100)) / 100"
| bc));" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE

# Inserciones masivas para cuentas corrientes (tabla derivada:
cuenta_corriente)
echo "-- Inserciones para cuenta_corriente" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
# Rango de IBANs distinto (por ejemplo, 2301-2600)
for i in $(seq 2301 2600); do
 # Asignamos un codigo_sucursal aleatorio entre 1 y 1000 (sucursales
creadas anteriormente)
 codigo_sucursal=$((RANDOM % 1000 + 1))
 echo "INSERT INTO cuenta_corriente (iban, numero_cuenta, saldo,
fecha_creacion, codigo_sucursal) VALUES ('ESCORR${i}', 'COR${i}',
$((RANDOM % 10000 + 100)), CURRENT_DATE, ${codigo_sucursal});" >>
$SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE

# Inserciones masivas para la tabla intermedia cuenta_cliente
(relacionando clientes con cuentas)
echo "-- Inserciones para cuenta_cliente" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
# Relacionamos de forma aleatoria clientes con cuentas (se asume que
las cuentas base tienen IBANs con prefijo 'ES123...')
for i in $(seq 1 1000); do
 cuenta_num=$((RANDOM % 1000 + 1001))
 echo "INSERT INTO cuenta_cliente (email, iban) VALUES
('cliente${i}@ejemplo.com', 'ES1234567890123456789${cuenta_num}');" >>
$SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE
```

```
# Inserciones masivas para operaciones bancarias y sus subentidades
# Primero, se insertan registros en operacion_bancaria y luego se
derivan en cada tipo

# Inserciones para TRANSFERENCIA
echo "-- Inserciones para transferencia" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
for i in $(seq 1 300); do
 cod=$((2000 + i))
 # Seleccionar aleatoriamente dos números para las cuentas (emisor y
destinatario) asegurando que sean distintos
 issuer_num=$((RANDOM % 1000 + 1001))
 recipient_num=$((RANDOM % 1000 + 1001))
 while [ $recipient_num -eq $issuer_num ]; do
   recipient_num=$((RANDOM % 1000 + 1001))
 done
 iban_emisor="ES1234567890123456789${issuer_num}"
 iban_destinatario="ES1234567890123456789${recipient_num}"
 cantidad=$((RANDOM % 500 + 50))
 echo "INSERT INTO transferencia (codigo_numerico, iban, tipo,
cantidad, fecha, hora, iban_destinatario) VALUES (${cod},
'${iban_emisor}', 'TRANSFERENCIA', ${cantidad}, CURRENT_DATE,
CURRENT_TIME, '${iban_destinatario}');" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE

# Inserciones para INGRESO
echo "-- Inserciones para ingreso" >> $SQL_FILE
echo "BEGIN;" >> $SQL_FILE
for i in $(seq 1 300); do
 cod=$((3000 + i))
 iban="ES1234567890123456789$((1001 + (i % 1000)))"
 cantidad=$((RANDOM % 500 + 50))
 echo "INSERT INTO ingreso (codigo_numerico, iban, tipo, cantidad,
fecha, hora, codigo_sucursal) VALUES (${cod}, '${iban}', 'INGRESO',
${cantidad}, CURRENT_DATE, CURRENT_TIME, (SELECT codigo_sucursal FROM
sucursal ORDER BY random() LIMIT 1));" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE


# Inserciones para RETIRADA
echo "-- Inserciones para retirada" >> $SQL_FILE
```

```
echo "BEGIN;" >> $SQL_FILE
for i in $(seq 1 300); do
 cod=$((4000 + i))
 iban="ESAHORRO$((2001 + (i % 300)))"
 cantidad=$((RANDOM % 500 + 50))
 echo "INSERT INTO retirada (codigo_numerico, iban, tipo, cantidad,
fecha, hora, codigo_sucursal) VALUES (${cod}, '${iban}', 'RETIRO',
${cantidad}, CURRENT_DATE, CURRENT_TIME, (SELECT codigo_sucursal FROM
sucursal ORDER BY random() LIMIT 1));" >> $SQL_FILE
done
echo "COMMIT;" >> $SQL_FILE


echo "Insertando datos masivos..."
cat $SQL_FILE | docker exec -i $CONTAINER_NAME psql -U $ADMIN_USER -d
$DATABASE

echo "Datos insertados exitosamente."

# Ejecución de SELECTs para verificar las inserciones en cada tabla
echo "Mostrando 10 registros de la tabla cliente:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT email, nombre, apellidos, edad FROM cliente LIMIT 10;"

echo "Mostrando 10 registros de la tabla sucursal:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT codigo_sucursal, direccion, telefono FROM sucursal LIMIT 10;"

echo "Mostrando 10 registros de la tabla cuenta:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT iban, numero_cuenta, saldo FROM cuenta LIMIT 10;"

echo "Mostrando 10 registros de la tabla cuenta_ahorro:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT iban, numero_cuenta, saldo, interes FROM cuenta_ahorro LIMIT
10;"

echo "Mostrando 10 registros de la tabla cuenta_corriente:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT iban, numero_cuenta, saldo, codigo_sucursal FROM
cuenta_corriente LIMIT 10;"

echo "Mostrando 10 registros de la tabla cuenta_cliente:"
```

```
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT email, iban FROM cuenta_cliente LIMIT 10;"

echo "Mostrando 10 registros de la tabla transferencia:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT codigo_numerico, iban, tipo, cantidad, iban_destinatario FROM
transferencia LIMIT 10;"

echo "Mostrando 10 registros de la tabla ingreso:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT codigo_numerico, iban, tipo, cantidad, codigo_sucursal FROM
ingreso LIMIT 10;"

echo "Mostrando 10 registros de la tabla retirada:"
docker exec -it $CONTAINER_NAME psql -U $ADMIN_USER -d $DATABASE -c
"SELECT codigo_numerico, iban, tipo, cantidad, codigo_sucursal FROM
retirada LIMIT 10;"

echo "Script docker_postgres_insert_and_read.sh completado."
```

En primer lugar, se insertan 1000 registros en la tabla de clientes, generando datos aleatorios para campos como edad, nombre y dirección, seguido de 1000 inserciones en la tabla de sucursales, donde cada sucursal se identifica mediante un código único. Posteriormente, se insertan 1000 registros en la tabla base de cuentas (con un rango de identificadores que evita colisiones con registros manuales), y se procede a insertar datos específicos en las tablas derivadas: para la tabla de cuentas de ahorro se generan 300 registros con un campo adicional de interés, y para la tabla de cuentas corrientes se insertan 300 registros asignando un código de sucursal aleatorio. Además, se llena la tabla intermedia que relaciona clientes con cuentas, asignando de forma aleatoria cada cliente a una cuenta base existente. En cuanto a las operaciones bancarias, el script inserta 300 registros en la tabla de transferencias, asegurando que el IBAN del emisor y el del destinatario (ambos seleccionados aleatoriamente de la tabla base de cuentas) sean distintos para cumplir con la integridad referencial. Para los ingresos y retiradas se realizan 300 inserciones cada una; en estos casos, se utiliza un subquery que selecciona aleatoriamente un código de sucursal existente en la tabla de sucursales, garantizando así que se respeten las restricciones de clave foránea. Finalmente, el script ejecuta una serie de comandos SELECT que muestran 10 registros de cada tabla, permitiendo la verificación visual de que los datos se han insertado correctamente en el esquema.

Inserción de datos en IBM DB2:

```
#!/bin/bash

# Nombre del contenedor de DB2
CONTAINER_NAME="db2-server"
```

```
# Usuario, contraseña y base de datos (según tu docker-compose.yml)
DB_USER="db2inst1"
DB_PASSWORD="admin"
DB_NAME="testdb"

# Ruta dentro del contenedor donde se copiará el script SQL
SQL_SCRIPT_PATH="/tmp/poblado.sql"

echo "Verificando si el contenedor $CONTAINER_NAME está corriendo..."
if ! docker ps | grep -q "$CONTAINER_NAME"; then
   echo "Error: El contenedor $CONTAINER_NAME no está corriendo.
Inícialo con:"
   echo "docker-compose up -d"
   exit 1
fi

# Crear el script SQL
SQL_SCRIPT=$(cat <<EOF
CONNECT TO $DB_NAME USER $DB_USER USING '$DB_PASSWORD';

-- CLIENTES
INSERT INTO Cliente (id_cliente, DNI, Edad, Nombre, Apellidos, Email,
Direccion, Telefono) VALUES
(ClienteUdt(1), '11111111A', 30, 'Juan', 'Pérez',
'juan.perez@email.com', 'Calle Falsa 123, Ciudad X', '555-1234'),
(ClienteUdt(2), '22222222A', 28, 'Ana', 'Gómez', 'ana.gomez@email.com',
'Avenida Siempre Viva 456, Ciudad Y', '555-5678'),
(ClienteUdt(3), '33333333A', 35, 'Carlos', 'Martínez',
'carlos.martinez@email.com', 'Calle Principal 789, Ciudad Z',
'555-9876'),
(ClienteUdt(4), '44444444A', 40, 'Laura', 'Hernández',
'laura.hernandez@email.com', 'Plaza Mayor 12, Ciudad W', '555-1111'),
(ClienteUdt(5), '55555555A', 25, 'Pedro', 'López',
'pedro.lopez@email.com', 'Calle del Sol 45, Ciudad V', '555-2222');

-- SUCURSALES
INSERT INTO Sucursal (id_sucursal, Codigo, Direccion, Telefono) VALUES
(SucursalUdt(1), 101, 'Sucursal Central, Ciudad X', 5551111),
(SucursalUdt(2), 102, 'Sucursal Norte, Ciudad Y', 5552222),
(SucursalUdt(3), 103, 'Sucursal Este, Ciudad Z', 5553333);

-- CUENTAS
```

```
INSERT INTO Cuenta (id_cuenta, IBAN, Numero_Cuenta, Saldo,
Fecha_Creacion) VALUES
(CuentaUdt(1), 'ES1234567890123456789012', 1234567890, 1000.00,
'2022-01-15'),
(CuentaUdt(2), 'ES9876543210987654321098', 987654321, 2500.00,
'2022-02-20'),
(CuentaUdt(3), 'ES5432167890123456789012', 543216789, 1500.00,
'2022-03-01');

-- CUENTAS AHORRO (subtipo)
INSERT INTO CuentaAhorro VALUES
(CuentaAhorroUdt(4), 'ES1234567890123456789012', 1234567890, 1000.00,
'2022-01-15', 0.2),
(CuentaAhorroUdt(5), 'ES9876543210987654321098', 987654321, 2500.00,
'2022-02-20', 0.5);

-- CUENTAS CORRIENTE (subtipo)
INSERT INTO CuentaCorriente VALUES
(CuentaCorrienteUdt(6), 'ES5432167890123456789012', 543216789, 1500.00,
'2022-03-01', SucursalUdt(1));


-- OPERACIONES BANCARIAS (padre)
INSERT INTO Operacion_Bancaria VALUES
(OperacionBancariaUdt(1), 'Transferencia de dinero', 1,
TIME('12:00:00'), DATE('2023-03-01'), 200.00, 'TRANSFERENCIA',
CuentaUdt(1)),
(OperacionBancariaUdt(2), 'Retiro en cajero automático', 2,
TIME('12:00:00'), DATE('2023-03-02'), 500.00, 'RETIRO', CuentaUdt(2)),
(OperacionBancariaUdt(3), 'Ingreso en cuenta', 3, TIME('12:00:00'),
DATE('2023-03-03'), 300.00, 'INGRESO', CuentaUdt(3));


-- TRANSFERENCIAS
INSERT INTO Transferencia VALUES
(TransferenciaUdt(4), 'Transferencia de dinero', 1, TIME('12:00:00'),
DATE('2023-03-01'), 200.00, 'TRANSFERENCIA', CuentaUdt(1),
CuentaUdt(2));

-- RETIRADAS
INSERT INTO Retirada VALUES
(RetiradaUdt(5), 'Retiro en cajero automático', 2, TIME('12:00:00'),
DATE('2023-03-02'), 500.00, 'RETIRO', CuentaUdt(2), SucursalUdt(1));
```

```
-- INGRESOS
INSERT INTO Ingreso VALUES
(IngresoUdt(6), 'Ingreso en cuenta', 3, TIME('12:00:00'),
DATE('2023-03-03'), 300.00, 'INGRESO', CuentaUdt(3),  SucursalUdt(2));

-- TIENE
INSERT INTO Tiene (id_tiene, DNI, IBAN) VALUES
(TieneUdt(1), ClienteUdt(1), CuentaUdt(1)),
(TieneUdt(2), ClienteUdt(2), CuentaUdt(2)),
(TieneUdt(3), ClienteUdt(3), CuentaUdt(3));

SELECT * FROM Cliente;
SELECT * FROM Sucursal;
SELECT * FROM Cuenta;
SELECT * FROM Operacion_Bancaria;
SELECT * FROM Transferencia;
SELECT * FROM Retirada;
SELECT * FROM Ingreso;
SELECT * FROM CuentaAhorro;
SELECT * FROM CuentaCorriente;
SELECT * FROM Tiene;

EOF
)

# Guardar el script en un archivo dentro del host
echo "$SQL_SCRIPT" > poblado.sql

# Copiar el archivo al contenedor DB2
docker cp poblado.sql $CONTAINER_NAME:$SQL_SCRIPT_PATH

# Ejecutar el script dentro del contenedor
docker exec -i $CONTAINER_NAME bash -c "
   su - $DB_USER -c \"
       db2 connect to $DB_NAME user $DB_USER using '$DB_PASSWORD';
       db2 -tvf $SQL_SCRIPT_PATH;
       db2 terminate;
   \"
"

# Limpiar archivos temporales
rm -f poblado.sql
```

```
echo "Inserción de datos completada en DB2 dentro del contenedor
$CONTAINER_NAME."
```

Implementación con db4o

Código Java junto con los comentarios y explicaciones que resulten relevantes. Se deben incluir scripts de compilación y ejecución.

Comparación de los SGBD

Decir diferencias entre los distintos SGBD en cuanto al soporte del relacional, objeto relacional, y otras cosas relevantes.

Descripción de los Triggers:

1. `check_balance_before_transfer`:
    - **Valida el saldo de la cuenta emisora** antes de realizar una transferencia, asegurando que haya suficiente saldo para completar la operación.
2. `update_balance_after_ingreso`:
    - **Actualiza el saldo de la cuenta** después de insertar un **ingreso**, sumando el monto ingresado al saldo actual de la cuenta.
3. `update_balance_after_retiro`:
    - **Actualiza el saldo de la cuenta** después de insertar un **retiro**, restando el monto retirado del saldo actual de la cuenta.
4. `delete_cliente_cuentas`:
    - **Elimina las relaciones entre el cliente y sus cuentas** en la tabla intermedia `CuentaCliente` cuando se elimina un cliente.
5. `check_transferencia_same_account`:
    - **Evita que se realicen transferencias a la misma cuenta** (cuando `iban_emisor` y `iban_receptor` son iguales), lanzando un error si se intenta hacerlo.
6.

-- Trigger para asegurar que `codigo_numerico` y `iban` coincidan en `OperacionesBancarias` antes de insertar en `Transferencias`

CREATE OR REPLACE TRIGGER check_tipo_operacion_transferencia

BEFORE INSERT ON Transferencias

FOR EACH ROW

DECLARE

v_tipo_operacion VARCHAR2(20);

BEGIN

-- Verificar que el `codigo_numerico` y `iban` en `OperacionesBancarias` coincidan con la transferencia

SELECT tipo_operacion

```
    INTO v_tipo_operacion

    FROM OperacionesBancarias

    WHERE codigo_numerico = :NEW.codigo_numerico

    AND iban = :NEW.iban_emisor;



    -- Si el tipo no es 'TRANSFERENCIA', lanzar un error

    IF v_tipo_operacion != 'TRANSFERENCIA' THEN

        RAISE_APPLICATION_ERROR(-20001, 'El codigo_numerico y el iban no están
asociados a un tipo de operación "TRANSFERENCIA".');

    END IF;

END;

/
```

antes de insertar en transferencia, miras si esta ese iban y codigo numerico en la tabla operaciones bancarias, si esta hay que mirar que el tipo sea “transferencia” si no no tendria sentido insertarlo en la tabla transferencias si el tipo != transferencia

teneis que hacer el mismo trigger para ingresos y retiradas