Memoria Técnica Práctica 4
Bases de Datos 2
Curso 2024/2025
Universidad de Zaragoza

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

Parte 1: Generación automática del esquema de BD

A continuación se describe el mapeo objeto-relacional de las entidades del sistema bancario cuando se genera automáticamente el esquema de base de datos a partir de las clases Java anotadas con JPA. Se detalla cómo se modelan las claves, atributos y relaciones, incluyendo herencia y claves compuestas.

Entidad Cliente

La clase Cliente está anotada con @Entity y se asocia a la tabla Cliente mediante @Table(name = "Cliente"). La clave primaria natural se representa con el campo DNI, usando @Id y @Column(name = "DNI"). No se emplea generación automática, por lo que el DNI debe proporcionarse manualmente.

Los atributos simples (nombre, apellidos, edad, email, teléfono) se anotan con @Column, marcando como nullable = false aquellos obligatorios.

Relación muchos a muchos con Cuenta

Se modela mediante @ManyToMany y una tabla intermedia clientes_cuentas definida con @JoinTable, que incluye las columnas cliente_dni (FK a Cliente) y cuenta_id (FK a Cuenta).

Para gestionar la relación, se utiliza un Set<Cuenta> con métodos auxiliares addCuenta() y removeCuenta() que sincronizan ambas direcciones de la asociación.

Relación muchos a uno con Direccion

La clase incluye una relación @ManyToOne(cascade = CascadeType.PERSIST) hacia Direccion, indicando que varios clientes pueden compartir la misma dirección. Se especifica la columna de la FK con @JoinColumn(name = "Direccion_ID_Direccion"). El uso de CascadeType.PERSIST asegura que, al crear un cliente, su dirección se persiste automáticamente, pero no se elimina si es compartida.

Entidad Cuenta

La clase Cuenta se anota con @Entity y @Table(name = "Cuenta"), y se define como clase base para una jerarquía de herencia mediante @Inheritance(strategy = InheritanceType.JOINED).

El campo IBAN actúa como clave primaria natural, declarado con @Id y @Column(nullable = false). Los atributos simples incluyen:

- numerocuenta y saldo → @Column(nullable = false)
- fechaCreacion → @Temporal(TemporalType.DATE)

Relaciones

- @ManyToMany(mappedBy = "cuentas"): relación inversa con Cliente.
- @OneToMany(mappedBy = "cuentaOrigen"): lista de operaciones emitidas.
- @OneToMany(mappedBy = "cuentaDestino"): lista de transferencias recibidas.

Estas asociaciones no tienen cascade definido y son LAZY por defecto.

Subclase CuentaAhorro

La clase CuentaAhorro extiende Cuenta, se declara con @Entity y @Table(name = "CuentaAhorro"), y participa en la herencia JOINED.

El campo interes es específico de esta subclase, anotado como @Column(name = "Interes", nullable = false). Se utiliza un constructor que delega en super(...) para inicializar los campos heredados.

Subclase CuentaCorriente

La clase CuentaCorriente extiende Cuenta, se declara con @Entity(name = "CuentaCorriente"), y hereda la estrategia JOINED.

Relación con Oficina

Incluye una relación @ManyToOne hacia Oficina, sin especificar @JoinColumn, por lo que se utiliza el nombre por defecto (oficina_id). Es una relación EAGER sin cascade, por lo que no se propagan operaciones desde la cuenta a la oficina.

Entidad Direccion

La clase Direccion está anotada con @Entity y @Table(name = "Direccion"). Su clave primaria se genera automáticamente con una secuencia:

@Id

@GeneratedValue(strategy = SEQUENCE, generator = "direccion_seq")

@SequenceGenerator(sequenceName = "DIRECCION_SEQ", allocationSize = 1)

Atributos obligatorios:

- calle, codigoPostal, ciudad → todos anotados con @Column(nullable = false)

No tiene relaciones salientes directas; es referenciada desde Cliente.

Entidad Oficina

La clase Oficina se mapea con @Entity y @Table(name = "Oficina"), y utiliza un código de 4 dígitos como clave primaria natural (@Id y @Column(name = "CODIGO_OFICINA")).

Atributos simples:

- direccion, telefono → @Column(...)

Relaciones

- @OneToMany(mappedBy = "oficina") con OperacionEfectivo
- @OneToMany(mappedBy = "oficina") con CuentaCorriente

Estas relaciones no incluyen cascade y son LAZY.

Entidad Operacion (abstracta)

La clase Operacion es abstracta, anotada con @Entity y @Table(name = "Operacion"), y sirve como superclase para operaciones concretas. Utiliza herencia JOINED.

Clave primaria compuesta

Se utiliza @IdClass(OperacionPK.class). La PK se compone de:

- codigoOperacion: @Id @Column(...)
- cuentaOrigen: @Id @ManyToOne @JoinColumn(...)

Atributos:

- fechaHora → @Temporal(TemporalType.TIMESTAMP)
- cantidad (double, obligatorio)

- descripcion (texto, opcional)

Clase auxiliar OperacionPK

Implementa Serializable y define:

- codigoOperacion (int)
- cuentaOrigen (String IBAN)

Esta clase se usa en la entidad Operacion para gestionar la clave compuesta con @IdClass.

Subclase OperacionEfectivo

La clase OperacionEfectivo extiende Operacion, y se mapea con @Entity y @Table(name = "OPERACION_EFECTIVO").

Atributo específico:

- tipoOperacionEfectivo → @Column(nullable = false) + @Enumerated(EnumType.STRING)

Relación:

- @ManyToOne(optional = false) con Oficina. No incluye cascade; el fetch es EAGER.

Subclase Transferencia

La clase Transferencia extiende Operacion y se anota con @Entity y @Table(name = "Transferencia").

Relación específica:

- @ManyToOne(optional = true) con Cuenta cuentaDestino. La FK se crea por convención (cuentaDestino_IBAN).

No se propagan operaciones (cascade), y la relación es EAGER por defecto.

Pruebas de inserción

Para verificar el correcto funcionamiento del modelo de datos, se ha implementado una rutina de inserción completa que permite poblar la base de datos con una estructura representativa y suficientemente compleja como para validar relaciones, claves primarias y foráneas, asociaciones bidireccionales, herencia y operaciones bancarias.

En primer lugar, se han creado dos oficinas (Oficina o1 y Oficina o2) con sus respectivos códigos, direcciones y teléfonos. A continuación, se han insertado cuatro clientes distintos, cada uno con una dirección única, y se han persistido en la base de datos junto con dichas direcciones. Cada cliente ha sido vinculado a una cuenta bancaria individual, generando así una relación many-to-many entre Cliente y Cuenta, gestionada por medio de la tabla intermedia clientes_cuentas.

Adicionalmente, se ha creado un cliente de prueba con una cuenta convencional (Cuenta) y una cuenta corriente (CuentaCorriente) asociada a una oficina específica, distinta de las anteriores, para validar la correcta gestión de la herencia entre cuentas y la relación many-to-one con la entidad Oficina.

Una vez insertadas estas entidades principales, se ha procedido al registro de operaciones bancarias, incluyendo diferentes tipos de transacciones para cubrir los principales casos del modelo:

- Una transferencia de prueba de 750 €, asociada al cliente de prueba, cuya cantidad supera los 500 € y se realiza dentro del periodo actual. Esta operación se utiliza más adelante para comprobar la lógica de las consultas basadas en transferencias recientes y de alto importe.

- Dos operaciones en efectivo, una de tipo INGRESO (500 €) y otra de tipo RETIRADA (200 €), ambas realizadas a través de instancias de la subclase OperacionEfectivo, lo cual permite verificar la jerarquía de herencia en la clase Operacion y su relación con Oficina.

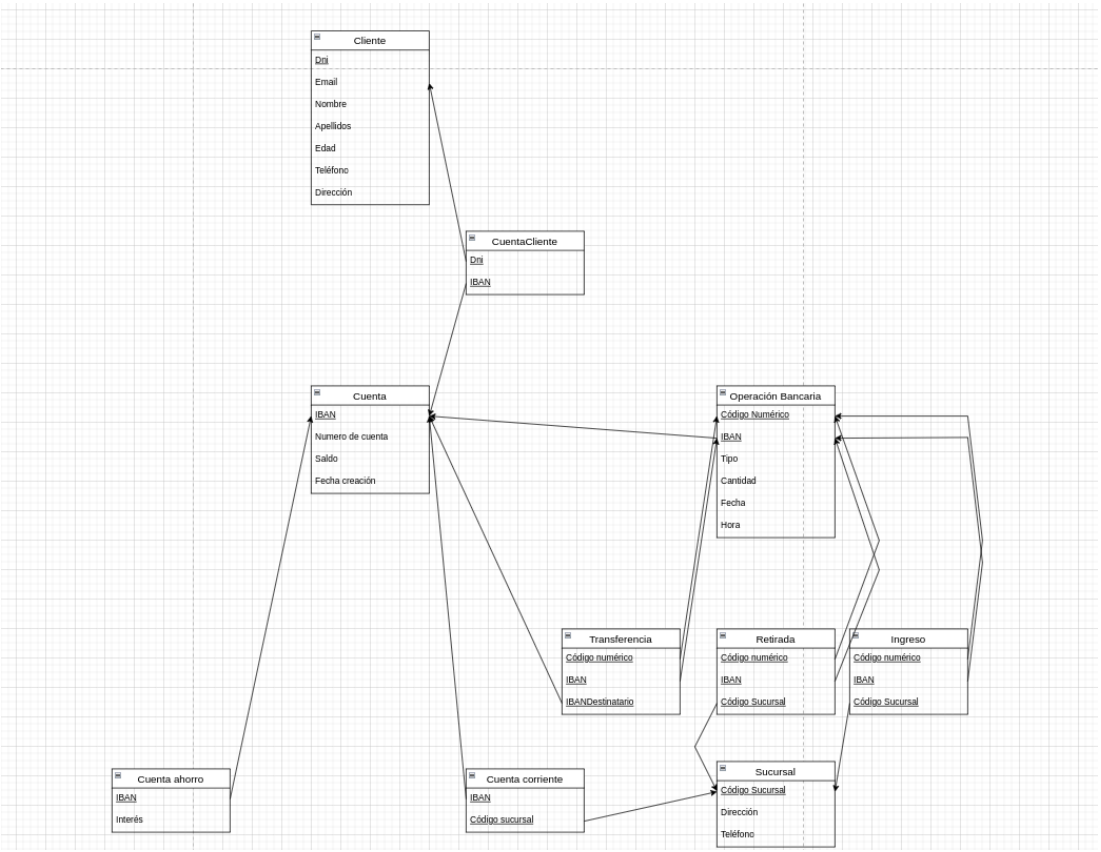
- Una transferencia estándar de 300 € entre dos cuentas existentes, que permite evaluar el correcto funcionamiento de las relaciones entre operaciones y cuentas de origen/destino.

Estas inserciones han sido ejecutadas dentro de bloques transaccionales controlados mediante EntityTransaction, con manejo de excepciones para garantizar la integridad del modelo y una trazabilidad clara de los errores en consola en caso de fallo.

Las operaciones descritas se han realizado exclusivamente con el fin de poblar el sistema y preparar el entorno para la ejecución de consultas, las cuales serán explicadas en detalle en el siguiente apartado.

Parte 2 : Esquema de BD preexistente

Para la segunda parte, se ha basado en el esquema de la BD de banquito de la P2, a continuación se recuerdan las claves de ese esquema relacional



Entidad Cliente

La tabla Cliente almacena información personal (como DNI, nombre, email, edad, etc.). Cada cliente puede tener múltiples cuentas asociadas, lo que se refleja mediante una tabla intermedia CuentaCliente, que modela una relación de muchos a muchos entre clientes y cuentas.

Entidad Cuenta

La entidad Cuenta actúa como tabla base para los dos subtipos: Cuenta_ahorro (con interés asociado) y Cuenta_corriente (que incluye una clave foránea a Sucursal). Se utiliza el campo IBAN como clave primaria, y además como clave foránea en las tablas hijas.

Operaciones bancarias

La entidad OperacionBancaria representa operaciones realizadas sobre una cuenta (IBAN), e incluye tipo, cantidad, fecha y hora.

Esta tabla es extendida por tres subtipos: Ingreso, Retirada y Transferencia. Cada subtipo representa un tipo específico de operación. Tanto Ingreso como Retirada están también relacionadas con la entidad Sucursal, permitiendo trazar en qué oficina se realizó la operación, mientras que Transferencias tiene un campo específico IBAN Destinatario que es una clave ajena que apunta a la entidad cuenta.

Entidad Sucursal

Sucursal incluye los campos codigoSucursal, dirección y teléfono, y se relaciona con operaciones como Ingreso y Retirada, así como con las cuentas corrientes.

Correspondencia entre el esquema relacional y JPA

A continuación se detallan los aspectos clave del mapeo objeto-relacional realizado para establecer la correspondencia entre el esquema relacional preexistente del sistema *banquito* y las clases Java anotadas con JPA.

---

1. Mapeo de entidades base y atributos

Cada una de las tablas principales (Cliente, Cuenta, Sucursal, Operacion) ha sido modelada como una clase Java con la anotación @Entity, y se ha indicado el nombre de la tabla correspondiente con @Table.
Los atributos de cada clase han sido mapeados mediante @Column, manteniendo los nombres y restricciones del esquema relacional (por ejemplo, nullable = false o el tipo de dato DATE usando @Temporal).

---

2. Relaciones muchos a muchos

La relación muchos a muchos entre Cliente y Cuenta ha sido modelada utilizando @ManyToMany en ambas entidades.
Se ha definido una tabla intermedia clientes_cuentas mediante @JoinTable, que refleja fielmente la tabla relacional CuentaCliente.

Además, se han añadido los métodos addCuenta() y removeCuenta() en la clase Cliente para mantener sincronizadas ambas direcciones de la relación.

Se ha especificado el atributo cascade en la relación para facilitar la propagación automática de operaciones entre entidades relacionadas. Concretamente:

@ManyToMany(cascade = {CascadeType.PERSIST, CascadeType.MERGE})

Esto implica que al persistir o actualizar un Cliente, también se persistirán o actualizarán automáticamente las cuentas asociadas.

Es útil para mantener la consistencia del modelo sin necesidad de manejar manualmente las operaciones sobre las entidades dependientes.

---

3. Herencia: Cuenta → Ahorro / Corriente

El esquema relacional implementa herencia entre cuentas (Cuenta, Cuenta_ahorro, Cuenta_corriente).

Para reflejar esta jerarquía en JPA, se ha utilizado: : @Inheritance(strategy = InheritanceType.JOINED)en la clase base Cuenta.

Las subclases Ahorro y Corriente extienden esta clase, y se definen como entidades independientes (@Entity).

Esto permite almacenar los atributos comunes en la tabla Cuenta, y los específicos en las tablas hijas.

---

4. Relaciones uno a muchos / muchos a uno

Las relaciones entre cuentas, operaciones y sucursales se han modelado usando @OneToMany y @ManyToOne, según corresponda:

- Una Cuenta puede tener muchas OperacionBancaria.
- Una Sucursal puede estar asociada a múltiples Retirada, Ingreso y CuentaCorriente.

Estas relaciones se han implementado respetando la semántica del modelo relacional, especificando las claves foráneas con @JoinColumn en el lado poseedor.

---

5. Herencia en operaciones bancarias

La jerarquía de operaciones bancarias (OperacionBancaria, Ingreso, Retirada, Transferencia) ha sido implementada con herencia JOINED, de forma similar al caso de cuentas.

Se ha creado una clase abstracta OperacionBancaria con la anotación @Inheritance, que actúa como superclase de las operaciones concretas.

La clave primaria compuesta (codigo_numerico, iban) se ha gestionado usando la anotación @IdClass con una clase auxiliar OperacionPK.

---

6. Claves compuestas

El uso de claves compuestas en la tabla OperacionBancaria se ha resuelto correctamente mediante @IdClass, utilizando una clase OperacionPK que implementa Serializable.

Los campos identificadores se anotan con @Id y se gestiona la clave foránea a Cuenta mediante @ManyToOne y @JoinColumn.

---

7. Pruebas de Inserción

Para verificar el funcionamiento correcto del modelo objeto-relacional implementado, se han realizado una serie de inserciones representativas que permiten validar la integridad de las relaciones, la estructura de herencia y las operaciones bancarias más comunes.

En primer lugar, se han creado dos sucursales (Sucursal o1 y Sucursal o2) con sus respectivos códigos, direcciones y teléfonos. A continuación, se han insertado cuatro clientes con direcciones en diferentes ciudades, edades y datos de contacto. Cada cliente ha sido vinculado a una cuenta bancaria individual, estableciendo así la relación many-to-many entre Cliente y Cuenta mediante la tabla intermedia correspondiente.

Además de estos datos básicos, se ha introducido un cliente de prueba con una cuenta de tipo corriente (Corriente) asociada a una sucursal adicional, creada específicamente para esta prueba. También se ha añadido un cliente adicional (Lucía) con una cuenta de alto saldo para simular escenarios de transferencias relevantes. Ambas cuentas han sido correctamente relacionadas con los clientes y las entidades correspondientes.

A nivel de operaciones, se han registrado varias transacciones para validar el modelo de herencia y las asociaciones entre operaciones y cuentas:

- Una transferencia de 750 €, emitida por la cuenta de Lucía y recibida por una cuenta creada específicamente para tal fin. Esta operación tiene un importe superior a 500 € y se realiza en una fecha actual, por lo que resulta útil para pruebas relacionadas con consultas temporales y de umbral.

- Un ingreso en efectivo de 500 € a través de la subclase Ingreso, realizado en la sucursal o1 y vinculado a una de las cuentas existentes.

- Una retirada de 200 € mediante la subclase Retirada, realizada en la sucursal o2.

- Una transferencia estándar de 300 € entre dos cuentas, que permite comprobar tanto la relación entre cuentas como la correcta actualización de saldos.

Todas las operaciones se han ejecutado dentro de transacciones gestionadas explícitamente mediante EntityTransaction, con control de errores para garantizar la consistencia de los datos. Adicionalmente, se han actualizado los saldos de las cuentas involucradas en cada operación mediante operaciones merge().

Estas pruebas de inserción permiten poblar el sistema con datos completos y coherentes, estableciendo así un escenario adecuado para la ejecución de consultas más avanzadas, que serán explicadas en apartados posteriores.

Parte 3: Consultas

En esta parte, el objetivo principal es la **prueba de la base de datos** mediante la formulación de consultas no triviales utilizando las distintas posibilidades que ofrece JPA:

- JPQL (Java Persistence Query Language)
- Criteria API
- SQL Nativo

Las consultas que se proponen tienen como finalidad resolver distintos problemas comunes en sistemas bancarios, tales como obtener estadísticas agregadas, identificar clientes según sus operaciones, analizar el uso de servicios, etc.

En este apartado se ha estructurado la presentación de las consultas en bloques, uno por cada consulta planteada explicando los 3 tipos de consulta utilizados para cada una. Dichas consultas son las siguientes:

- El saldo medio de las cuentas de los clientes por ciudad
- Las sucursales cuyo número de clientes tiene un saldo mayor que el saldo medio de los clientes.
- Los clientes que en los últimos 3 meses hayan realizado transferencias mayores a 500 euros

Consulta 1

JPQL

Esta consulta se realiza con **JPQL (Java Persistence Query Language)**, que permite escribir consultas orientadas a objetos, basadas en las entidades mapeadas.

Parte 1

```
        List<Object[]> mediasPorCiudad = em.createQuery(
                "SELECT d.Ciudad, AVG(cu.Saldo) " +
                        "  FROM Cliente cli " +
                        "  JOIN cli.Direccion d " +
                        "  JOIN cli.Cuentas cu " +
                        " GROUP BY d.Ciudad" +
                        " ORDER BY AVG(cu.Saldo) DESC",
                Object[].class)
                .getResultList();

        System.out.println("Resultado JPQL:");
        // Mostramos el resultado de la consulta
```

```
            for (Object[] fila : mediasPorCiudad) {
                String ciudad = (String) fila[0];
                Double saldoMedio = (Double) fila[1];
                System.out.printf("Ciudad: %s → Saldo medio: %.2f%n",
ciudad, saldoMedio);
            }
```

Parte 2

Dado que la ciudad no está normalizada en una entidad propia, se extrae directamente del campo **direccion** de **Cliente** utilizando funciones JPQL como **SUBSTRING**, **LOCATE**, **CASE** y **TRIM**..

Este uso avanzado de JPQL permite realizar transformaciones sobre cadenas dentro de la propia consulta, lo que evita la necesidad de normalizar direcciones si no se dispone de una entidad **Direccion**.

```
List<Object[]> mediasPorCiudad = em.createQuery(
   // Seleccionamos dos columnas: la ciudad extraída y el promedio de
saldo
   "SELECT " +
   // ======= COLUMNA 1: ciudad extraída =======
            "  TRIM( " + // Quita espacios al inicio/fin
            "    SUBSTRING( cli.direccion, " + // Toma subcadena de
'direccion' a partir de…
            "      (CASE " +
            "         WHEN LOCATE(',', cli.direccion, LOCATE(',',
cli.direccion) + 1) > 0 " +
            "           THEN LOCATE(',', cli.direccion, LOCATE(',',
cli.direccion) + 1)" +
            "         WHEN LOCATE(',', cli.direccion) > 0 " +
            "           THEN LOCATE(',', cli.direccion)" +
            "         ELSE 0 " +
            "       END) + 1 " + // …la posición resultante +1
            "    )" +
            "  ), " +
            // ======= COLUMNA 2: promedio de saldo =======
            "  AVG(ct.saldo) " +
            "FROM Cliente cli " + // Entidad Cliente
            "  JOIN cli.cuentas ct " + // Relación a Cuenta
            // Agrupamos por la misma expresión de ciudad
            "GROUP BY " +
            "  TRIM( " +
            "    SUBSTRING( cli.direccion, " +
```

```
            "       (CASE " +
            "          WHEN LOCATE(',', cli.direccion, LOCATE(',',
cli.direccion) + 1) > 0 " +
            "            THEN LOCATE(',', cli.direccion, LOCATE(',',
cli.direccion) + 1)" +
            "          WHEN LOCATE(',', cli.direccion) > 0 " +
            "            THEN LOCATE(',', cli.direccion)" +
            "          ELSE 0 " +
            "        END) + 1 " +
            "    )" +
            "  ) " +
            // Orden descendente por el promedio calculado
            "ORDER BY AVG(ct.saldo) DESC",
   Object[].class).getResultList();

System.out.println("Resultados JPQL:");
for (Object[] fila : mediasPorCiudad) {
System.out.printf("Ciudad: %s → Saldo medio: %.2f%n",
       fila[0], fila[1]);
}
```

Criteria API

Con **Criteria API**, la misma consulta se construye paso a paso mediante objetos tipados que permiten construir dinámicamente consultas de forma segura. Estas consultas se construyen paso a paso, creando variables intermedias hasta llegar a la solución final. Cada paso define una parte de la consulta: **multiselect, groupBy, orderBy,** etc.

Parte 1

Se define la raíz sobre la entidad **Cliente**, y se realizan dos **join**: uno hacia **Direccion** para obtener la ciudad (**joinDir.get("Ciudad")**), y otro hacia **Cuentas** para acceder al saldo.

Se construye una expresión de promedio (**cb.avg(...)**) y se seleccionan ambos valores con **multiselect**. A continuación, se agrupa por ciudad y se ordena de forma descendente por saldo medio. Finalmente, se ejecuta la consulta con **em.createQuery(cq).getResultList().**

```
            CriteriaBuilder cb = em.getCriteriaBuilder();

            CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);



            // 2. Definimos la raíz de la consulta sobre Cliente
```

```
            Root<Cliente> cliente = cq.from(Cliente.class);


            // 3. Hacemos los joins a las asociaciones de Cliente
            Join<Cliente, Direccion> joinDir =
cliente.join("Direccion");
            Join<Cliente, Cuenta> joinCta = cliente.join("Cuentas");


            // Expresión de promedio
            Expression<Double> avgSaldo = cb.avg(joinCta.get("Saldo"));
            // 4. Seleccionamos dos cosas:
            // Construcción de la consulta
            cq.multiselect(
                    joinDir.get("Ciudad"),
                    avgSaldo);


            // 5. Definimos la cláusula GROUP BY y ORDER BY
            cq.groupBy(joinDir.get("Ciudad"));
            cq.orderBy(cb.desc(avgSaldo));


            // 6. Ejecutamos la consulta
            List<Object[]> resultados =
em.createQuery(cq).getResultList();


            System.out.println("Resultado Criteria API:");
            // 7. Procesamos resultados
            for (Object[] fila : resultados) {
```

```
                String ciudad = (String) fila[0];

                Double saldMedio = (Double) fila[1];

                System.out.printf("Ciudad=%s → Saldo medio=%.2f%n",
ciudad, saldMedio);

            }
```

Parte 2

```
            CriteriaBuilder cb = em.getCriteriaBuilder();
            CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);

            // 2. Definimos la raíz de la consulta sobre Cliente
            Root<Cliente> cliente = cq.from(Cliente.class);

            // 3. Hacemos el join con Cuenta
            Join<Cliente, Cuenta> cuentas = cliente.join("cuentas");

            // 4. Expresiones de selección:
            // a) ciudad = campo direccion tal cual
            Expression<String> ciudadExpr = cliente.get("direccion");
            // b) promedio de saldo
            Expression<Double> avgSaldo = cb.avg(cuentas.get("saldo"));

            // 5. Construimos SELECT, GROUP BY y ORDER BY
            cq.multiselect(ciudadExpr, avgSaldo)
                    .groupBy(ciudadExpr)
                    .orderBy(cb.desc(avgSaldo));

            // 6. Ejecutamos y procesamos resultados
            List<Object[]> resultados =
em.createQuery(cq).getResultList();
            System.out.println("Resultados Criteria API (sin parseo de
comas):");
            for (Object[] fila : resultados) {
                String ciudad = (String) fila[0];
                Double saldoMedio = (Double) fila[1];
                System.out.printf("Ciudad: %s → Saldo medio: %.2f%n",
ciudad, saldoMedio);
            }
```

SQL Nativo

El **SQL Nativo** permite escribir consultas directamente en SQL, tal como se harían en el SGBD, sin restricciones del modelo de objetos. Se usa **createNativeQuery()** para ejecutarlas, lo cual ofrece máxima flexibilidad y control sobre la consulta, especialmente útil cuando JPQL o Criteria API no permiten expresar algo de forma sencilla.

Parte 1

```
            // SQL Nativo
            @SuppressWarnings("unchecked")
            List<Object[]> resultadosNativo = em.createNativeQuery(
                    "SELECT d.CIUDAD, AVG(c.SALDO) AS SaldoMedio " + //
Seleccionamos la ciudad y el saldo medio
                            "  FROM CLIENTE cli " + // Desde la entidad
Cliente
                            "  JOIN DIRECCION d  " + // Join con la
entidad Direccion
                            "    ON cli.DIRECCION_ID_DIRECCION =
d.ID_DIRECCION " + // Sobre el id de Direccion
                            "  JOIN CLIENTES_CUENTAS cc  " + // Join con
la tabla intermedia CLIENTES_CUENTAS
                            "    ON cli.DNI = cc.CLIENTE_DNI " + //
Sobre el dni de Cliente
                            "  JOIN CUENTA c  " + // Join con la entidad
Cuenta
                            "    ON cc.CUENTA_ID = c.IBAN " + // Sobre
el id de Cuenta
                            " GROUP BY d.CIUDAD" + // //Agrupamos
resultados por ciudad
                            " ORDER BY SaldoMedio DESC" // añadimos
orden descendente
            ).getResultList();

            System.out.println("Resultado SQL Nativo:");
            for (Object[] fila : resultadosNativo) {
                String ciudad = (String) fila[0];
                Number saldMedio = (Number) fila[1];
                System.out.printf("Ciudad=%s → Saldo medio=%.2f%n",
                        ciudad, saldMedio.doubleValue());
            }
```

Parte 2

```
            @SuppressWarnings("unchecked")
            // Ejecutar consulta SQL nativa para calcular el saldo medio
por ciudad,
            // extrayendo la ciudad de 'direccion' tras la última coma
(o toda la cadena si
            // no hay comas)
            List<Object[]> resultadosNativo = em.createNativeQuery(
                    // Iniciamos el SELECT con un CASE para extraer la
ciudad
                    "SELECT " +
                            "  CASE " +
                            "    WHEN INSTR(cli.direccion, ',') > 0 " +
                            // Si existe al menos una coma, usamos INSTR
con parámetro -1 para la última
                            // coma,
                            // y SUBSTR + LTRIM para obtener y recortar
lo que va después
                            "    THEN LTRIM(SUBSTR(cli.direccion,
INSTR(cli.direccion, ',', -1) + 1)) " +
                            // Si no hay coma, tomamos la dirección
completa y aplicamos TRIM para recortar
                            // espacios
                            "    ELSE TRIM(cli.direccion) " +
                            "  END AS ciudad, " + // Alias para la
columna ciudad
                            // Calculamos el promedio de saldo de las
cuentas
                            "  AVG(c.saldo) AS SaldoMedio " + // Alias
para la columna de promedio
                            "FROM Cliente cli " +
                            // Join con la tabla intermedia que
relaciona clientes y cuentas
                            "JOIN clientes_cuentas cc ON cli.dni =
cc.cliente_dni " +
                            // Join con la tabla Cuenta para acceder al
campo 'saldo'
                            "JOIN Cuenta c           ON cc.cuenta_id =
c.iban " +
                            // Agrupamos por la misma expresión CASE
para ciudad
                            "GROUP BY " +
                            "  CASE " +
```

```
                            "    WHEN INSTR(cli.direccion, ',') > 0 " +
                            "    THEN LTRIM(SUBSTR(cli.direccion,
INSTR(cli.direccion, ',', -1) + 1)) " +
                            "    ELSE TRIM(cli.direccion) " +
                            "  END " +
                            // Ordenamos los resultados de mayor a menor
saldo medio
                            "ORDER BY SaldoMedio DESC")
                    .getResultList();

            System.out.println("Resultados SQL nativo:");
            for (Object[] fila : resultadosNativo) {
                String ciudad = (String) fila[0];
                Number saldoMedio = (Number) fila[1];
                System.out.printf("Ciudad=%s → Saldo medio=%.2f%n",
                        ciudad, saldoMedio.doubleValue());
            }
```

Consulta 2

JPQL

Parte 1

Se calcula el saldo medio de todas las cuentas con **SELECT AVG(c.Saldo) FROM Cuenta c**, y se usa como parámetro en una consulta que obtiene la oficina con más clientes cuyas cuentas superan dicha media. Se hace **JOIN** entre **Oficina**, sus **cuentas** y los **clientes** asociados, se agrupan los resultados por oficina y se ordenan por el número de clientes únicos en orden descendente, devolviendo solo la primera.

```
// First calculate average account balance
            Double avgBalance = (Double) em.createQuery(
                "SELECT AVG(c.Saldo) FROM Cuenta c")
                .getSingleResult();

            System.out.println("Saldo medio de todas las cuentas: " +
avgBalance);

            // JPQL query for branches with most clients with
above-average balance
            List<Object[]> sucursalConMasClientes = em.createQuery(
                "SELECT o.codigoOficina, o.direccion, o.telefono,
COUNT(DISTINCT cli) " +
```

```
                "FROM Oficina o " +
                "JOIN o.cuentas c " +
                "JOIN c.Clientes cli " +
                "WHERE c.Saldo > :avgBalance " +
                "GROUP BY o.codigoOficina, o.direccion, o.telefono " +
                "ORDER BY COUNT(DISTINCT cli) DESC",
                Object[].class)
                .setParameter("avgBalance", avgBalance.longValue())
                .setMaxResults(1)
                .getResultList();

            if (!sucursalConMasClientes.isEmpty()) {
                Object[] resultado = sucursalConMasClientes.get(0);
                String codigoOficina = (String) resultado[0];
                String direccion = (String) resultado[1];
                String telefono = (String) resultado[2];
                Long numeroClientes = (Long) resultado[3];

                System.out.printf("\n" + "Sucursal con más clientes con
saldo superior a la media (%.2f): " +
                                  "Código: %s, Dirección: %s, Teléfono:
%s, Clientes: %d%n",
                                  avgBalance, codigoOficina, direccion,
telefono, numeroClientes);
            } else {
                System.out.println("No se encontraron sucursales con
clientes que tengan saldo superior a la media");
            }

            Double avgBalance1 = (Double) em.createQuery(
                "SELECT AVG(c.Saldo) FROM Cuenta c")
                .getSingleResult();
            System.out.println("Saldo medio de todas las cuentas: " +
avgBalance1);
```

Parte 2

```
Double avgBalance = em.createQuery(
                "SELECT AVG(c.saldo) FROM Cuenta c", Double.class)
              .getSingleResult();
            System.out.println("Saldo medio de todas las cuentas: " +
avgBalance);
            // 2) JPQL: branch with most clients above-average
            List<Object[]> sucursalJPQL = em.createQuery(
```

```
                "SELECT s.codigo_sucursal, s.direccion, s.telefono,
COUNT(DISTINCT cli) " +
                "FROM Sucursal s " +
                " JOIN s.cuentas c " +
                " JOIN c.clientes cli " +
                "WHERE c.saldo > :avgBalance " +
                "GROUP BY s.codigo_sucursal, s.direccion, s.telefono " +
                "ORDER BY COUNT(DISTINCT cli) DESC",
                Object[].class)
              .setParameter("avgBalance", avgBalance.longValue())
              .setMaxResults(1)
              .getResultList();

            // print sucursalJPQL
            if (!sucursalJPQL.isEmpty()) {
                Object[] r = sucursalJPQL.get(0);
                System.out.printf(
                  " \n   JPQL - CódigoSucursal=%d, Dirección=%s,
Teléfono=%s, Clientes=%d%n",
                    ((Number)r[0]).intValue(), r[1], r[2],
((Number)r[3]).longValue()
                );
            }
            else {
                System.out.println("\n No hay sucursales con saldo medio
superior a " + avgBalance + "\n");
            }
```

Criteria API

Parte 1

Se crea una consulta con **CriteriaBuilder** sobre la entidad **Sucursal**, haciendo **join** con sus **cuentas** y los **clientes** de cada cuenta. Se filtran solo las cuentas cuyo saldo supera la media (**avgBalance**) y se agrupan por oficina. Se seleccionan el código, dirección, teléfono y el número de clientes distintos. Se ordena por cantidad de clientes en orden descendente y se limita el resultado a una única fila con **setMaxResults(1).**

```
            CriteriaBuilder cb = em.getCriteriaBuilder();

            // First get average balance (already calculated above)

            // Now build the query for offices with clients having
above-average balances
            CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
```

```
            Root<Oficina> oficinaRoot = cq.from(Oficina.class);

            // Join to associated accounts and then to clients
            Join<Oficina, Cuenta> joinCuentas =
oficinaRoot.join("cuentas");
            Join<Cuenta, Cliente> joinClientes =
joinCuentas.join("Clientes");

            // COUNT distinct clients
            Expression<Long> countClients =
cb.countDistinct(joinClientes);

            // Select office attributes and client count
            cq.multiselect(
                    oficinaRoot.get("codigoOficina"),
                    oficinaRoot.get("direccion"),
                    oficinaRoot.get("telefono"),
                    countClients);

            // Add WHERE clause for above-average balance
            cq.where(cb.gt(joinCuentas.get("Saldo"),
avgBalance.longValue()));

            // Group by office fields
            cq.groupBy(
                    oficinaRoot.get("codigoOficina"),
                    oficinaRoot.get("direccion"),
                    oficinaRoot.get("telefono"));

            // Order by client count descending
            cq.orderBy(cb.desc(countClients));

            // Execute and get results
            List<Object[]> sucursalConMasClientesCriteria =
em.createQuery(cq)
                    .setMaxResults(1)
                    .getResultList();

            if (!sucursalConMasClientesCriteria.isEmpty()) {
                Object[] resultado =
sucursalConMasClientesCriteria.get(0);
                String codigoOficina = (String) resultado[0];
                String direccion = (String) resultado[1];
```

```
                String telefono = (String) resultado[2];
                Long numeroClientes = (Long) resultado[3];

                System.out.printf(
                        "\n " + "Criteria API - Sucursal con más
clientes con saldo superior a la media (%.2f): " +
                                "Código: %s, Dirección: %s, Teléfono:
%s, Clientes: %d%n",
                        avgBalance, codigoOficina, direccion, telefono,
numeroClientes);
            }
```

Parte 2

```
            CriteriaBuilder cb = em.getCriteriaBuilder();
            CriteriaQuery<Object[]> cq = cb.createQuery(Object[].class);
            Root<Sucursal> suRoot = cq.from(Sucursal.class);
            Join<Sucursal, Cuenta> joinC  = suRoot.join("cuentas");
            Join<Cuenta, Cliente> joinCl  = joinC.join("clientes");
            Expression<Long> cnt           = cb.countDistinct(joinCl);

            cq.multiselect(
                suRoot.get("codigo_sucursal"),
                suRoot.get("direccion"),
                suRoot.get("telefono"),
                cnt
            )
            .where(cb.gt(joinC.get("saldo"), avgBalance))
            .groupBy(
                suRoot.get("codigo_sucursal"),
                suRoot.get("direccion"),
                suRoot.get("telefono")
            )
            .orderBy(cb.desc(cnt));

            List<Object[]> sucursalCriteria = em.createQuery(cq)
                .setMaxResults(1)
                .getResultList();

            // print sucursalCriteria
            if (!sucursalCriteria.isEmpty()) {
                Object[] r = sucursalCriteria.get(0);
                System.out.printf(
```

```
                "\n  Criteria API - CódigoSucursal=%d, Dirección=%s,
Teléfono=%s, Clientes=%d%n",
                ((Number)r[0]).intValue(), r[1], r[2],
((Number)r[3]).longValue()
            );
        }
```

SQL Nativo

Parte 1

```
        @SuppressWarnings("unchecked")
        List<Object[]> sucursalConMasClientesNativo =
em.createNativeQuery(
                "SELECT o.CODIGO_OFICINA, o.DIRECCION, o.TELEFONO,
COUNT(DISTINCT cli.DNI) as NUM_CLIENTES " +
                        "FROM OFICINA o " +
                        "LEFT JOIN CUENTACORRIENTE cc ON
o.CODIGO_OFICINA = cc.OFICINA_CODIGO_OFICINA " +
                        "LEFT JOIN CUENTA c ON cc.IBAN = c.IBAN " +
                        "JOIN CLIENTES_CUENTAS cc_junction ON c.IBAN
= cc_junction.CUENTA_ID " +
                        "JOIN CLIENTE cli ON cc_junction.CLIENTE_DNI
= cli.DNI " +
                        "WHERE c.SALDO > ? " +
                        "GROUP BY o.CODIGO_OFICINA, o.DIRECCION,
o.TELEFONO " +
                        "ORDER BY NUM_CLIENTES DESC")
                .setParameter(1, avgBalance1.longValue())
                .setMaxResults(1)
                .getResultList();

        if (!sucursalConMasClientesNativo.isEmpty()) {
            Object[] resultado =
sucursalConMasClientesNativo.get(0);
            String codigoOficina = (String) resultado[0];
            String direccion = (String) resultado[1];
            String telefono = (String) resultado[2];
            Number numeroClientes = (Number) resultado[3];

            System.out.printf(
                    "\n" + "SQL Nativo - Sucursal con más clientes
con saldo superior a la media (%.2f): " +
```

```
                                "Código: %s, Dirección: %s, Teléfono:
%s, Clientes: %d%n",
                        avgBalance, codigoOficina, direccion, telefono,
numeroClientes.longValue());
            }
```

Parte 2

```
            @SuppressWarnings("unchecked")
            List<Object[]> sucursalNativo = (List<Object[]>)
em.createNativeQuery(
                "SELECT s.CODIGO_SUCURSAL, s.DIRECCION, s.TELEFONO,
COUNT(DISTINCT cli.dni) " +
                "FROM OFICINA s " +  // ← Use correct table name
matching your entity
                "  JOIN Corriente co           ON s.CODIGO_SUCURSAL =
co.codigo_sucursal_codigo_sucursal " +
                "  JOIN Cuenta c               ON co.iban            =
c.iban " +
                "  JOIN clientes_cuentas cc    ON c.iban             =
cc.cuenta_id " +
                "  JOIN Cliente cli            ON cc.cliente_dni     =
cli.dni " +
                "WHERE c.saldo > ? " +
                "GROUP BY s.CODIGO_SUCURSAL, s.DIRECCION, s.TELEFONO " +
                "ORDER BY COUNT(DISTINCT cli.dni) DESC"
            )
            .setParameter(1, avgBalance.longValue())
            .setMaxResults(1)
            .getResultList();

            // print sucursalNativo
            if (!sucursalNativo.isEmpty()) {
                Object[] r = sucursalNativo.get(0);
                System.out.printf(
                  "\n  Native SQL - CódigoSucursal=%d, Dirección=%s,
Teléfono=%s, Clientes=%d%n",
                    ((Number)r[0]).intValue(), r[1], r[2],
((Number)r[3]).longValue()
                );
            }
            else {
                System.out.println("\n No hay sucursales con saldo medio
superior a " + avgBalance + "\n");
```

```
            }
```

Consulta 3

JPQL

Parte 1

```
            Date fechaLimite = new Date(System.currentTimeMillis() - 90L
* 24 * 60 * 60 * 1000); // 3 meses

            List<Cliente> clientesTransferencias = em.createQuery(
                    "SELECT DISTINCT c " +
                            "FROM Cliente c " +
                            "JOIN c.Cuentas cuenta " +
                            "JOIN cuenta.operaciones op " +
                            "WHERE TYPE(op) = Transferencia " +
                            "AND op.cantidad > 500 " +
                            "AND op.fechaHora >= :fechaLimite",
                    Cliente.class)
                    .setParameter("fechaLimite", fechaLimite)
                    .getResultList();

            System.out.println("Clientes con transferencias > 500€ en
los últimos 3 meses (JPQL):");
            for (Cliente c : clientesTransferencias) {
                System.out.println(c);
            }
```

Parte 2

```
            Date fechaLimite = new Date(System.currentTimeMillis() - 90L
* 24 * 60 * 60 * 1000); // hace 3 meses
            // JPQL
            List<Cliente> clientesTransferencias = em.createQuery(
                    "SELECT DISTINCT cli " +
                            "FROM Cliente cli " +
                            "JOIN cli.cuentas c " +
                            "JOIN c.operaciones op " +
                            "WHERE TYPE(op) = Transferencia " +
                            "  AND op.cantidad > 500 " +
                            "  AND op.fecha >= :fechaLimite",
```

```
                    Cliente.class).setParameter("fechaLimite",
fechaLimite)
                    .getResultList();

            System.out.println("Clientes con transferencias > 500 € en
los últimos 3 meses (JPQL):");
            for (Cliente c : clientesTransferencias) {
                System.out.println(c);
            }
```

Criteria API

Parte 1

```
            CriteriaBuilder cb = em.getCriteriaBuilder();
            CriteriaQuery<Cliente> cq = cb.createQuery(Cliente.class);

            Root<Cliente> clienteRoot = cq.from(Cliente.class);
            Join<Cliente, Cuenta> joinCuenta =
clienteRoot.join("Cuentas");
            Join<Cuenta, Operacion> joinOperacion =
joinCuenta.join("operaciones");

            // ¡Aquí filtramos por clase Transferencia!
            cq.select(clienteRoot).distinct(true)
                    .where(
                            cb.and(
                                    cb.equal(joinOperacion.type(),
Transferencia.class),

cb.greaterThan(joinOperacion.<Double>get("cantidad"), 500.0),

cb.greaterThanOrEqualTo(joinOperacion.<Date>get("fechaHora"),
fechaLimite)));

            List<Cliente> resultados =
em.createQuery(cq).getResultList();

            System.out.println("Resultado Criteria API (transferencias
>500€ en últimos 3 meses):");
            for (Cliente c : resultados) {
                System.out.println(c);
            }
```

Parte 2

```
            CriteriaBuilder cb = em.getCriteriaBuilder();
            CriteriaQuery<Cliente> cq = cb.createQuery(Cliente.class);
            Root<Cliente> rootCliente = cq.from(Cliente.class);
            Join<Cliente, Cuenta> joinCuenta =
rootCliente.join("cuentas");
            Join<Cuenta, OperacionBancaria> joinOperacion =
joinCuenta.join("operaciones");

            // Filtro por tipo Transferencia (usando discriminador
"tipo")
            cq.select(rootCliente).distinct(true)
                    .where(
                            cb.and(
                                    cb.equal(joinOperacion.get("tipo"),
"transferencia"),

cb.greaterThan(joinOperacion.get("cantidad"), 500.0),

cb.greaterThanOrEqualTo(joinOperacion.get("fecha"), fechaLimite)));

            List<Cliente> resultado =
em.createQuery(cq).getResultList();

            System.out.println("Resultado Criteria API (transferencias
>500 € en últimos 3 meses):");
            for (Cliente c : resultado) {
                System.out.println(c);
            }
```

SQL Nativo

Parte 1

```
            @SuppressWarnings("unchecked")
            List<Object[]> resultadosSQLNative = em.createNativeQuery(
                    "SELECT DISTINCT c.* " +
                            "FROM CLIENTE c " +
                            "JOIN CLIENTES_CUENTAS cc ON c.DNI =
cc.CLIENTE_DNI " +
                            "JOIN CUENTA cu ON cc.CUENTA_ID = cu.IBAN "
+
                            "JOIN OPERACION op ON cu.IBAN =
op.CUENTA_ORIGEN_IBAN " +
```

```
                            "JOIN TRANSFERENCIA t ON t.CODIGO_OPERACION
= op.CODIGO_OPERACION AND t.CUENTA_ORIGEN_IBAN = op.CUENTA_ORIGEN_IBAN
"
                            +
                            "WHERE op.CANTIDAD > 500 AND op.FECHA_HORA
>= ?")
                    .setParameter(1, fechaLimite)
                    .getResultList();

            // Mostrar resultados
            System.out.println("Resultado SQL Nativo (transferencias
>500€ en últimos 3 meses):");
            for (Object[] fila : resultadosSQLNative) {
                Cliente cli = new Cliente();
                cli.setDni((String) fila[0]);
                cli.setApellidos((String) fila[1]);
                cli.setEdad(((Number) fila[2]).intValue());
                cli.setEmail((String) fila[3]);
                cli.setNombre((String) fila[4]);
                cli.setTelefono((String) fila[5]);
                System.out.println(cli);
            }
```

Parte 2

```
            @SuppressWarnings("unchecked")
            List<Cliente> clientesSQLNativo = em.createNativeQuery(
                    "SELECT DISTINCT cli.* " +
                            "FROM Cliente cli " +
                            "JOIN clientes_cuentas cc ON cli.dni =
cc.cliente_dni " +
                            "JOIN Cuenta c ON cc.cuenta_id = c.iban " +
                            "JOIN OPERACION o ON o.iban_iban = c.iban "
+
                            "JOIN Transferencia t ON t.codigo_numerico =
o.codigo_numerico AND t.iban_iban = o.iban_iban "
                            +
                            "WHERE o.cantidad > 500 AND o.fecha >= ?",
                    Cliente.class // <-- Esto es lo que permite que te
devuelva objetos Cliente directamente
            ).setParameter(1, fechaLimite)
                    .getResultList();
```

```
            System.out.println("Resultado SQL nativo (clientes con
transferencia >500 €):");
            for (Cliente c : clientesSQLNativo) {
                System.out.println(c);
            }
```

Esfuerzos invertidos

Horas invertidas	Diego	Carlos	Daniel	Total
Parte 1	6	11	3	20
Parte 2	7	0	7	14
Parte 3	2	3	3	8