SPRINT 4: CREACIÓN DE BASES DE DATOS

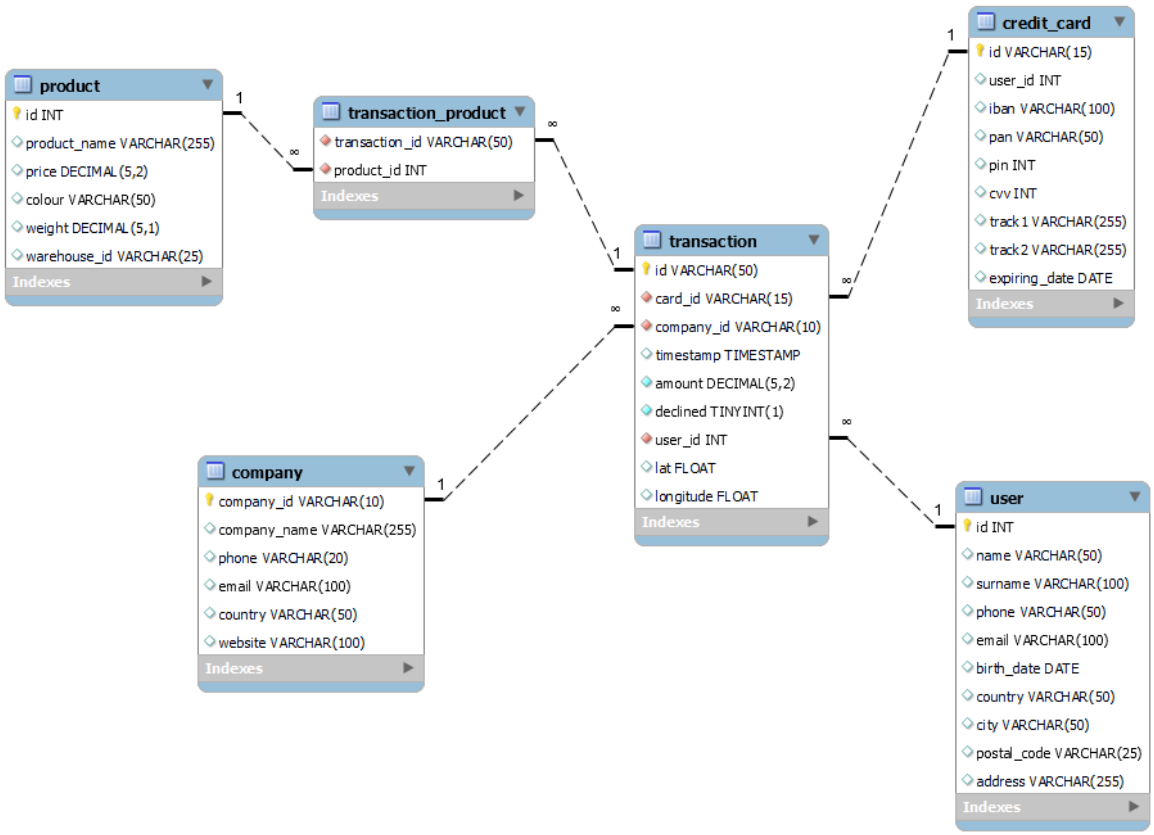
Descarga los archivos CSV, estúdialos y diseña una base de datos con un esquema de estrella que contenga, al menos, 4 tablas.

Los archivos CSV contenían registros sobre las transacciones, las empresas que participan en dichas transacciones, las tarjetas de crédito utilizadas, los usuarios que las realizaron (de tres países: Reino Unido, Estados Unidos y Canadá) y los productos vendidos. Se ha creado una base de datos llamada *operations* y las tablas correspondientes para cargar los datos de los archivos CSV (véase script estructura_operationsdb.sql para más detalles).

Los registros de transacciones contenían varios identificativos de productos por transacción en una única columna, cosa que complicaba filtrar los datos de un único producto de una lista string. Para solucionarlo, he creado una tabla nueva (transaction_product) para los ids de transacciones y los ids de productos. A continuación, he transformado la columna de product_ids de transactions.csv para separar cada producto de una transacción en una fila. *

Los tipos de datos de las variables se escogieron en función de la información que contenían. Por ejemplo, las columnas que contenían fechas las he definido como DATE (a pesar de que no tenían el formato aceptado YYYY-MM-DD), las que contenían números decimales (amount, price y weight) las he definido como DECIMAL y el resto como VARCHAR. **

En este modelo tenemos la tabla de hechos transaction, que conecta con las tablas dimensión credit_card ***, user y company con una relación N:1 (transaction N:1 dimensión). Para relacionar transaction con dimensión product tenemos la tabla intermedia transaction_product, que conecta N:1 con transaction y N:1 con product (véase diagrama a continuación).



* NOTA 1: He realizado la transformación de productos por fila en MySQL para que el modelo funcione cada vez que se reciba otro archivo CSV con nuevas transacciones. Para conseguir esta modificación, he necesitado los siguientes pasos:

- Crear una tabla temporal para cargar las columnas transaction_id y product_ids (en formato VARCHAR, al ser múltiples valores separados por coma; no se detectaría como INT)
- Importar las columnas mencionadas a esta tabla temporal

```
CREATE TEMPORARY TABLE transaction_product_varchar_temp (
    transaction_id VARCHAR(50),
    product_ids VARCHAR(100)
);
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/transactions.csv'
INTO TABLE transaction_product_varchar_temp
FIELDS TERMINATED BY ';'
LINES TERMINATED BY '\r\n'
IGNORE 1 ROWS
(transaction_id, @dummy, @dummy, @dummy, @dummy, @dummy, product_ids, @dummy, @dummy, @dummy);
SELECT * FROM transaction_product_varchar_temp;
```

- Crear otra tabla temporal en la que insertaremos los product_ids (en INT) separados por fila (se podrían insertar los datos transformados directamente a la tabla transaction_product, pero para asegurarme de que la transformación funciona bien, opté por una tabla temporal)
- Separar los productos en filas separadas:
    - Creamos una columna, llamada n, para establecer un índice interno que nos ayudará a separar cada registro de la columna product_ids; la relacionamos con la tabla temporal con los datos de product_ids (en VARCHAR) teniendo en cuenta que la longitud de product_ids – longitud product_ids (sin comas) es superior o igual a n-1. Por ejemplo:
        - Product_ids 71, 41 ⟶ char_length 6 – char_length 5 = 1 ≥ n-1
            - n=2 porque tenemos dos registros de products_ids
        - Product_ids 47, 37, 11, 1 ⟶ char_length 13 – char_length 10 = 3 ≥ n-1
            - n=4 porque tenemos cuatro registros de product_ids
    - Separamos los registros de product_ids con SUBSTRING_INDEX. Por ejemplo:
        - product_ids 71, 41
            - Para n=1 ⟶ 71
            - Para n=2 ⟶ 71, 41 ⟶ SUBSTRING_INDEX ((71, 41), ‘,’ -1) ⟶ 41
            - Para n=3: nada porque no se cumpliría que la longitud de caracteres es superior que n-1 ⟶ 1 no es superior que 2

      Si el número generado (n) es menor o igual al número de elementos en la lista, se extrae el elemento correspondiente de la cadena product_ids. Si no, esa fila de la subconsulta numbers no contribuirá a la extracción de elementos para esa fila de la tabla.

- Insertar los registros en la tabla transaction_product (una vez comprobado que la transformación es correcta)
- Eliminar las tablas temporales creadas

```sql
CREATE TEMPORARY TABLE transaction_product_int_temp (
    transaction_id VARCHAR(50),
    product_id INT
);
INSERT INTO transaction_product_int_temp (transaction_id, product_id)
SELECT  transaction_id,
        SUBSTRING_INDEX(SUBSTRING_INDEX(product_ids, ',', numbers.n), ',', -1) AS product_id
FROM transaction_product_varchar_temp
JOIN (  SELECT 1 AS n
        UNION ALL SELECT 2
        UNION ALL SELECT 3
        UNION ALL SELECT 4  ) AS numbers
ON CHAR_LENGTH(product_ids) - CHAR_LENGTH(REPLACE(product_ids, ',', '')) >= n - 1;
SELECT * FROM transaction_product_int_temp;

INSERT INTO transaction_product (transaction_id, product_id)
SELECT * FROM transaction_product_int_temp;

DROP TEMPORARY TABLE transaction_product_varchar_temp;
DROP TEMPORARY TABLE transaction_product_int_temp;
```

** NOTA 2: Para garantizar una carga precisa de los datos, he tenido que agregar cláusulas para ajustar las columnas del archivo CSV de origen. Esto funcionará siempre y cuando el archivo de origen mantenga el formato de los datos; de lo contrario, habría que modificar código.

- Para modificar el formato de fechas de origen a YYYY-MM-DD (DATE); modificación similar para birth_date (tabla user)

```sql
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/credit_cards.csv'
INTO TABLE credit_card
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(id, user_id, iban, pan, pin, cvv, track1, track2, @expiring_date)
SET expiring_date=STR_TO_DATE(@expiring_date,'%c/%d/%y');
```

- Para quitar el símbolo de $ de la columna price y, así, se pueda detectar como DECIMAL

```sql
LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/products.csv'
INTO TABLE product
FIELDS TERMINATED BY ','
LINES TERMINATED BY '\n'
IGNORE 1 ROWS
(id, product_name, @price, colour, weight, warehouse_id)
SET price=REPLACE(@price,'$','');
```

*** NOTA 3: La tabla credit_card contiene el identificador de los usuarios que son propietarios de las tarjetas de crédito. No he establecido una relación entre las tablas credit_card y user ya que se formaría una relación cíclica en mi base de datos.

NIVEL 1

<u>Ejercicio 1.</u> Realiza una subconsulta que muestre a todos los usuarios con más de 30 transacciones utilizando al menos 2 tablas.

Primero de todo debemos conocer qué usuarios realizaron más de 30 transacciones, mediante una subconsulta que devuelva los id del usuario y la cantidad de transacciones (filtradas por > 30).

Una vez sabemos los identificadores de estos usuarios, los relacionamos con la tabla user para conocer sus datos personales.

```
SELECT u.*, user_trans.cant_trans
FROM user u
JOIN (SELECT user_id, COUNT(id) AS cant_trans
        FROM transaction
        GROUP BY user_id
        HAVING cant_trans > 30) user_trans
ON u.id=user_trans.user_id;
```

Obtenemos un total de cuatro usuarios que cumplen el criterio establecido: Lynn Riddle (id 92), Ocean Nelson (id 267), Hedwig Gilbert (id 272) y Kenyon Hartman (id 275). Sus datos personales y la cantidad de transacciones realizadas por cada uno de ellos se muestran en la tabla a continuación.

id	name	surname	phone	email	birth_date	country	city	postal_code	address	cant_trans
92	Lynn	Riddle	1-387-885-4...	vitae.aliquet@outlo...	1984-09-21	United States	Bozeman	61871	P.O. Box 712, 790...	39
267	Ocean	Nelson	079-481-2745	aenean@yahoo.com	1991-12-26	Canada	Charlottetown	85X 3P4	Ap #732-8357 Pe...	52
272	Hedwig	Gilbert	064-204-8788	sem.eget@icloud.edu	1991-04-16	Canada	Tuktoyaktuk	Q4C 3G7	P.O. Box 496, 514...	76
275	Kenyon	Hartman	082-871-7248	convallis.ante.lectu...	1982-08-03	Canada	Richmond	R8H 2K2	8564 Facilisi. St.	48

<u>Ejercicio 2.</u> Muestra el promedio de las transacciones por IBAN de las tarjetas de crédito en la compañía Donec Ltd. utilizando al menos 2 tablas.

En la tabla de transacciones tenemos id_company pero no su nombre, por lo que tenemos que recurrir a la tabla company mediante una subconsulta para conocer el identificador de la compañía Donec Ltd.

Una vez sabemos el id, podemos calcular el promedio de las transacciones de esta compañía en función de las tarjetas de crédito utilizadas (GROUP BY card_id).

Al tener ya el promedio por tarjeta, solo falta relacionarlo con la tabla credit_card para tener el número de IBAN asociado a la tarjeta.

```
WITH avg_trans_card AS
(
SELECT card_id, ROUND(AVG(amount),2) AS avg_amount
FROM transaction
WHERE company_id = (SELECT company_id
                    FROM company
                    WHERE company_name='Donec Ltd')
GROUP BY card_id
)
SELECT iban, atc.avg_amount AS promedio_transaccion
FROM credit_card cc
JOIN avg_trans_card atc
ON cc.id=atc.card_id;
```

En este caso, la compañía Donec Ltd. tenía tan solo dos transacciones y las dos se realizaron con la misma tarjeta de crédito. Por tanto, como resultado obtenemos un número iban (mostrado en la tabla) con un promedio de 203.72 euros.

iban	promedio_transaccion
PT87806228135092429456346	203.72

NIVEL 2

Crea una nueva tabla que refleje el estado de las tarjetas de crédito basado en si las últimas tres transacciones fueron declinadas.

Para crear una tabla que muestre el estado de las tarjetas, debemos conseguir las últimas tres transacciones realizadas con las tarjetas. Para ello, seleccionamos información relevante, como es el id de la tarjeta, timestamp (fecha y hora de la transacción) y declined (0: aceptada, 1: rechazada).

Dado que el número de transacciones por tarjeta varían, debemos aplicarle un "índice" interno a cada fila de transacción de una determinada tarjeta con la función ROW_NUMBER ().

Al usar PARTITION BY card_id, estamos indicando a la función que separe las asignaciones de números secuenciales para cada tarjeta individualmente. Es decir, cuando terminan las transacciones de una tarjeta y comienzan las de otra, el índice se reinicia a 1 para la nueva tarjeta. Esto asegura que cada tarjeta tenga su propio conjunto de índices únicos para sus transacciones.

Además, al especificar ORDER BY timestamp DESC, estamos ordenando las transacciones dentro de cada grupo de tarjeta en orden descendente según su fecha de realización. Esto significa que la asignación de índices se realiza de las transacciones más recientes a las más antiguas.

Así, obtenemos 587 transacciones, cada una con un índice asignado en función de la tarjeta. Este índice comienza desde 1 para la transacción más reciente de cada tarjeta y aumenta secuencialmente hacia las transacciones más antiguas.

Para considerar si una tarjeta está activa o no, hay que considerar las 3 transacciones más recientes de cada tarjeta (row_transaction <= 3); por tanto, en la tabla card_status únicamente incluiremos aquellas tarjetas que tengan al menos 3 transacciones registradas (COUNT card_id = 3). La razón detrás de este requisito es asegurarnos de tener suficiente información para tomar una decisión precisa sobre el estado de la tarjeta.

Esto quiere decir que aquellas tarjetas que tengan un total de 1 o 2 transacciones no cumplen con el criterio por el siguiente motivo:

- Dos transacciones: aunque ambas transacciones sean rechazadas, no podemos concluir que la tarjeta está inactiva. Faltaría saber si la tercera transacción que se realizará con esa tarjeta sería aceptada o no.
- Una transacción: aunque la única transacción esté rechazada, desconocemos si es debido a que la tarjeta está inactiva u otro motivo. Faltarían dos transacciones más para poder evaluar con mayor precisión el estado de la tarjeta.

En la tabla creada tendremos dos columnas, la primera con el identificador de la tarjeta y la segunda con el estado de la tarjeta: 'operativa' o 'inoperativa' (asignamos el estado con CASE).

- Si las tres transacciones son rechazadas, la suma de declined sería 3, por tanto 'inoperativa'

- Si hay una o dos transacciones rechazadas de las tres a considerar, la suma de declined sería <= 2, por tanto 'operativa'

Todas aquellas tarjetas que actualmente tengan una o dos transacciones se podrán añadir más adelante a la tabla card_status si obtenemos registros de dos o una transacción más, respectivamente.

```
CREATE TABLE IF NOT EXISTS card_status AS
(WITH trans_card AS
(
SELECT  card_id,
        timestamp,
        declined,
        ROW_NUMBER() OVER (PARTITION BY card_id ORDER BY timestamp DESC) AS row_transaction
FROM transaction
)
SELECT  card_id AS id_tarjeta,
        CASE
            WHEN SUM(declined) <= 2 THEN 'operativa'
            ELSE 'inoperativa'
        END AS estado_tarjeta
FROM trans_card
WHERE row_transaction <= 3
GROUP BY id_tarjeta
HAVING COUNT(card_id) = 3
);
```

<u>Ejercicio 1.</u> ¿Cuántas tarjetas están activas?

Una vez tenemos creada la tabla con el estado de las tarjetas, contamos cuántas están operativas.

```
SELECT COUNT(id_tarjeta) AS 'cantidad tarjetas activas'
FROM card_status
WHERE estado_tarjeta='operativa';
```

Como resultado, obtenemos 9 tarjeta activas. Cabe destacar que en la tabla card_status tenemos un total de nueve tarjetas, que son las únicas que cumplen con los criterios establecidos. Por tanto, todas las tarjetas presentes en card_status están activas.

cantidad tarjetas activas
9

NIVEL 3

<u>Ejercicio 1.</u> Necesitamos conocer el número de veces que se ha vendido cada producto.

Como queremos saber la cantidad de veces que se vendió cada producto, las transacciones rechazadas no deberían considerarse para el conteo. Es decir, considero que una transacción rechazada no representa una venta exitosa del producto.

Sin embargo, tener información sobre la cantidad de veces que fue rechazada la venta de un producto puede ser útil para otros análisis. Por ejemplo, podría servir para hacer un seguimiento de rechazos, para optimizar los procesos de venta, para estudiar las tendencias de venta, entre otros. Por tanto, en

mi informe he querido incluir tanto la cantidad de transacciones aceptadas como la cantidad de rechazadas para cada producto.

Con la tabla transaction_product podemos saber la cantidad total de transacciones que se han registrado para un determinado producto; pero, tal como he diseñado la tabla, no podemos distinguir si esas transacciones fueron aceptadas o rechazadas. Para ello, tenemos que recurrir a la tabla transaction mediante una JOIN.

Para calcular la cantidad de transacciones, utilizo la función SUM con la condición descrita en CASE.

- En la columna ventas.cant_acept aplico la condición de que todas las transacciones aceptadas (declined 0) tengan valor 1 y las rechazadas 0, por lo que sumando las veces que aparece cada producto obtenemos el total de ventas exitosas.
- En la columna ventas.cant_rech aplico la condición de que todas las transacciones rechazadas (declined 1) tengan valor 1 y las aceptadas 0, por lo que la suma daría el total de ventas rechazadas para cada producto.

En la columna ventas.cant_tot aplico la función COUNT sin ninguna condición para calcular la cantidad total de transacciones registradas para cada producto.

Una vez tenemos esta información, podemos relacionarla con la tabla product para conocer los detalles de cada producto mediante LEFT JOIN, ya que hay productos sin ninguna transacción registrada. Para evitar tener valores NULL para los productos sin ventas, aplico la función IFNULL ().

```sql
WITH ventas AS
(
SELECT  product_id,
        SUM(CASE WHEN declined = 0 THEN 1 ELSE 0 END) AS cant_acept,
        SUM(CASE WHEN declined = 1 THEN 1 ELSE 0 END) AS cant_rech,
        COUNT(product_id) AS cant_tot
FROM transaction_product tp
JOIN transaction t
ON tp.transaction_id = t.id
GROUP BY product_id
ORDER BY product_id
)
SELECT  p.*,
        IFNULL(v.cant_acept,0) AS cantidad_vendida,
        IFNULL(v.cant_rech,0) AS cantidad_rechazada,
        IFNULL(cant_tot,0) AS cantidad_total
FROM product p
LEFT JOIN ventas v
ON v.product_id=p.id;
```

Con esta consulta, obtenemos una tabla con 100 registros, lo que corresponde a la cantidad total de productos registrados en la tabla product. Al examinar la tabla, vemos que algunos productos tienen cero transacciones registradas (por ejemplo, el producto con el ID 4). Esto significa que no se ha realizado ninguna transacción relacionada con este producto hasta el momento.

id	product_name	price	colour	weight	warehouse_id	cantidad_vendida	cantidad_rechazada	cantidad_total
1	Direwolf Stannis	161.11	#7c7c7c	1.0	WH-4	51	10	61
2	Tarly Stark	9.24	#919191	2.0	WH-3	56	9	65
3	duel tourney Lannister	171.13	#d8d8d8	1.5	WH-2	43	8	51
4	warden south duel	71.89	#111111	3.0	WH-1	0	0	0
5	skywalker ewok	171.22	#dbdbdb	3.2	WH-0	42	7	49

Otro enfoque a este ejercicio sería agrupar los identificadores de los productos por el nombre, asumiendo que son el mismo producto con ligeras diferencias de peso y/o color. De este modo, relacionamos información de tres tablas (transaction, transaction_product y product) y hacemos los cálculos de los productos vendidos, rechazados y el total (véase query a continuación). Cabe destacar que, al hacer la agrupación por el nombre del producto, hay que omitir los detalles de peso, color, precio, etc.

```
WITH transacciones AS
(
SELECT  tp.*, t.declined
FROM transaction_product tp
JOIN transaction t
ON tp.transaction_id=t.id
)
SELECT  product_name AS producto,
        SUM(CASE WHEN declined=0 THEN 1 ELSE 0 END) AS cantidad_vendida,
        SUM(CASE when declined=1 THEN 1 ELSE 0 END) AS cantidad_rechazada,
        COUNT(product_id) AS cantidad_total
FROM product p
LEFT JOIN transacciones t
    ON t.product_id=p.id
GROUP BY producto
ORDER BY cantidad_total DESC, producto;
```

Como resultado, obtenemos 70 productos distintos, 24 con transacciones registradas y los 46 restantes sin ventas.

El producto con más registros es Direwolf Stannis con un total de 106 transacciones. Los cinco primeros resultados (ordenados por la cantidad total) de la consulta se muestran en la tabla a continuación:

producto	cantidad_vendida	cantidad_rechazada	cantidad_total
Direwolf Stannis	86	20	106
skywalker ewok	88	12	100
riverlands north	60	8	68
Winterfell	59	9	68
Direwolf riverlands the	52	14	66