

Autora: Giorgia Calvagna

Fecha de envío:

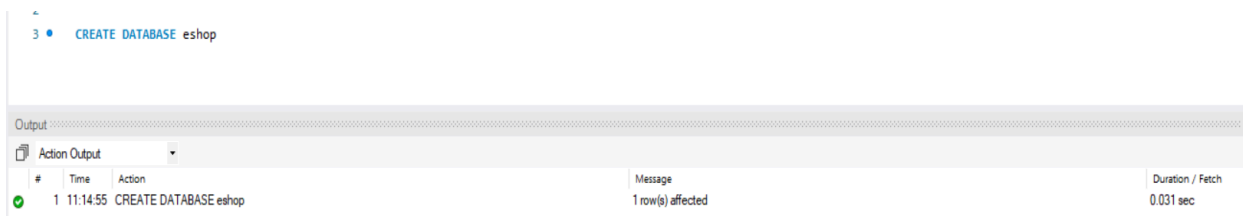
Revisado por:

Sprint 4: Modelat SQL

Nivel 1:

- Partiendo de algunos archivos CSV diseñarán y crearás tu base de datos:

Empezaremos creando una nueva base de datos, que llamaremos *eshop*:



Una vez creada la base de datos, crearemos las respectivas tablas que la formarán:

Las tablas que crearemos son: la tabla de hechos *transaction* y las tablas de dimensiones *companies*, *credit_cards*, *products*, *users*.

Los ficheros de que disponemos con respecto a la tabla de *users* son tres: *user_ca*, *user_uk*, *user_usa*. La estructura de estas tablas es la misma, por lo que crearemos una única tabla de usuarios que las contenga a todas.

La tabla *transactions* recolecta datos cuantitativos que la caracterizan como tabla de hechos además de contener las Foreign Keys relacionadas con las Primary Keys de las tablas de dimensiones.

Después de haber creado las cinco tablas, vamos añadiendo los valores que le pertenecen.

Tabla *companies*:

```
6 • CREATE TABLE IF NOT EXISTS companies (  
7     company_id VARCHAR(15) PRIMARY KEY,  
8     company_name VARCHAR(255) NOT NULL,  
9     phone VARCHAR(15) NULL,  
10    email VARCHAR(100) NOT NULL,  
11    country VARCHAR(100) NULL,  
12    website VARCHAR(255) NULL  
13 );  
14  
15 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/companies.csv'  
16 INTO TABLE companies  
17 FIELDS TERMINATED BY ','  
18 IGNORE 1 LINES;
```

Output

#	Time	Action	Message	Duration / Fetch
1	18:03:50	CREATE TABLE IF NOT EXISTS companies (company_id VARCHAR(15) PRIMARY KEY, -- Chiave primar...	0 row(s) affected	0.047 sec
2	18:04:31	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/companies.csv' INTO TABLE com...	100 row(s) affected Records: 100 Deleted: 0 Skipped: 0 Warnings: 0	0.032 sec

Este código SQL tiene como objetivo gestionar datos relacionados con empresas en una base de datos de manera estructurada y eficiente. El primer bloque define la estructura de la tabla llamada "companies", asegurándose de que esta sea creada solo si no existe previamente. Dentro de esta tabla, se establecen varias columnas: "company_id" como clave primaria, que sirve para identificar de manera única a cada empresa; "company_name" para almacenar el nombre de la empresa, obligatorio; y otras columnas como "phone", "email", "country" y "website" para registrar información adicional, algunas de las cuales son opcionales.

El segundo bloque de código carga datos en esta tabla desde un archivo externo llamado "companies.csv". Este archivo debe estar ubicado en la ruta especificada y contiene la información que se desea importar. Los datos se separan por comas, y la primera línea del archivo, que normalmente contiene los encabezados, es ignorada durante la importación para evitar conflictos con el contenido real. Este proceso permite poblar rápidamente la tabla con datos preexistentes de forma automatizada.

Tabla *credit_cards*:

```
20 CREATE TABLE IF NOT EXISTS credit_cards (  
21     id VARCHAR(15) PRIMARY KEY,  
22     user_id VARCHAR(100) NOT NULL,  
23     iban VARCHAR(40) UNIQUE NOT NULL,  
24     pan VARCHAR(45) NOT NULL,  
25     pin VARCHAR(5) NULL,  
26     cvv VARCHAR(4) NULL,  
27     track1 VARCHAR(100) NULL,  
28     track2 VARCHAR(100) NULL,  
29     expiring_date VARCHAR(10) NULL  
30 );  
31  
32 LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/credit_cards.csv'  
33 INTO TABLE credit_cards  
34 FIELDS TERMINATED BY ','  
35 IGNORE 1 LINES;
```

Output

#	Time	Action	Message	Duration / Fets
1	18:12:23	CREATE TABLE IF NOT EXISTS credit_cards (id VARCHAR(15) PRIMARY KEY, user_id VARCHAR(100) NOT NULL, iban VARCHAR(40) UNIQUE NOT NULL, pan VARCHAR(45) NOT NULL, pin VARCHAR(5) NULL, cvv VARCHAR(4) NULL, track1 VARCHAR(100) NULL, track2 VARCHAR(100) NULL, expiring_date VARCHAR(10) NULL);	0 row(s) affected	0.062 sec
2	18:12:26	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/credit_cards.csv' INTO TABLE credit_cards FIELDS TERMINATED BY ',' IGNORE 1 LINES;	275 row(s) affected Records: 275 Deleted: 0 Skipped: 0 Warnings: 0	0.031 sec

Este código SQL también está diseñado para gestionar datos en una base de datos, pero en este caso se centra en información relacionada con tarjetas de crédito. El primer bloque define la estructura de una tabla llamada "credit_cards" y se asegura de que sea creada solo si no existe previamente. Dentro de esta tabla, se establecen varias columnas: "id" como clave primaria para identificar de manera única cada tarjeta; "user_id" para asociar la tarjeta a un usuario específico, obligatorio; "iban" como un identificador único y obligatorio para la cuenta bancaria asociada; "pan" para almacenar el número principal de la tarjeta, también obligatorio; y otras columnas como "pin", "cvv", "track1", "track2" y "expiring_date" para registrar información adicional, todas ellas opcionales.

El segundo bloque de código carga datos en esta tabla desde un archivo externo llamado "credit_cards.csv". Este archivo debe estar ubicado en la ruta especificada y contiene la información que se desea importar. Los datos se separan por comas, y la primera línea del archivo, que normalmente contiene los encabezados, es ignorada durante la importación para evitar conflictos con el contenido real. Este proceso permite poblar la tabla con información preexistente de manera rápida y automatizada.

Tabla *products*:

```
37 CREATE TABLE IF NOT EXISTS products (  
38     id VARCHAR(255) PRIMARY KEY,  
39     product_name VARCHAR(255) NOT NULL,  
40     price DECIMAL(10, 2) NOT NULL DEFAULT 0.0,  
41     colour VARCHAR(100) NULL,  
42     weight DECIMAL(10, 2) DEFAULT 0.0,  
43     warehouse_id VARCHAR(40) NOT NULL  
44 );  
45  
46 LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/products.csv'  
47 INTO TABLE products  
48 FIELDS TERMINATED BY ','  
49 ENCLOSED BY '"'  
50 LINES TERMINATED BY '\n'  
51 IGNORE 1 LINES  
52 (id, product_name, @price, colour, @weight, warehouse_id)  
53 SET  
54     price = REPLACE(@price, '$', ''),  
55     weight = CAST(@weight AS DECIMAL(10,2));
```

Output

#	Time	Action	Message	Duration / Fets
1	18:26:13	CREATE TABLE IF NOT EXISTS products (id VARCHAR(255) PRIMARY KEY, product_name VARCHAR...	0 row(s) affected	0.047 sec
2	18:26:58	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/products.csv' INTO TABLE produ...	100 row(s) affected Records: 100 Deleted: 0 Skipped: 0 Warnings: 0	0.016 sec

Este código SQL presenta una particularidad en la forma en que maneja los datos durante su importación en la tabla "products". Aunque sigue la lógica de creación y carga de datos descrita anteriormente, se diferencia en que incluye instrucciones adicionales para transformar ciertos datos antes de almacenarlos. Durante la importación, se procesan las columnas "price" y "weight" a través del comando "SET". En el caso de "price", se reemplaza (usando el comando "REPLACE") el símbolo "\$" de los valores importados con una cadena vacía (") para asegurar que sean tratados como datos numéricos. Para "weight", con el comando "CAST" cualquier valor numérico (ya sea entero o decimal) es convertido al formato DECIMAL(10,2), garantizando consistencia en los datos almacenados. Además, se especifica que los campos en el archivo de origen están delimitados por comas y encerrados entre comillas dobles, y que cada línea termina con un salto de línea, lo que asegura una interpretación precisa del archivo durante la carga.

Tabla *users*:

```

58 CREATE TABLE IF NOT EXISTS users (
59     id INT PRIMARY KEY,
60     name VARCHAR(100) NOT NULL,
61     surname VARCHAR(100) NOT NULL,
62     phone VARCHAR(20),
63     email VARCHAR(100),
64     birth_date DATE,
65     country VARCHAR(100),
66     city VARCHAR(100),
67     postal_code VARCHAR(20),
68     address VARCHAR(255),
69     INDEX idx_email (email),
70     INDEX idx_phone (phone)
71 );
72
73 LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/users_ca.csv'
74 INTO TABLE users
75 FIELDS TERMINATED BY ','
76 OPTIONALLY ENCLOSED BY '"'
77 LINES TERMINATED BY '\r\n'
78
79

```

#	Time	Action	Message	Duration / Fetch
1	18:41:17	CREATE TABLE IF NOT EXISTS users (id INT PRIMARY KEY, name VARCHAR(100) NOT NULL, su...	0 row(s) affected	0.063 sec
2	18:41:57	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/users_ca.csv' INTO TABLE users ...	75 row(s) affected Records: 75 Deleted: 0 Skipped: 0 Warnings: 0	0.016 sec
3	18:42:14	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/users_uk.csv' INTO TABLE users ...	50 row(s) affected Records: 50 Deleted: 0 Skipped: 0 Warnings: 0	0.031 sec
4	18:42:32	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/users_usa.csv' INTO TABLE user...	150 row(s) affected Records: 150 Deleted: 0 Skipped: 0 Warnings: 0	0.031 sec

Este código SQL se centra en la gestión de los datos relacionados con usuarios, consolidando información de distintas regiones en una sola tabla llamada "users". La estructura de la tabla incluye campos clave como "id", "name" y "surname" para identificar a los usuarios, junto con otros campos adicionales como "phone", "email", "birth_date", "country", "city", "postal_code" y "address" para registrar información personal y de contacto. Además, se crean índices en las columnas "email" y "phone" para optimizar las consultas que involucren estos campos, mejorando el rendimiento en la búsqueda y filtrado de datos.

El código también incluye instrucciones para cargar datos desde tres archivos de origen: "users_ca.csv", "users_uk.csv" y "users_usa.csv". Aunque estos archivos contienen datos de diferentes regiones (Canadá, Reino Unido y Estados Unidos), comparten la misma estructura, lo que facilita su integración en una única tabla (users). Durante el proceso de carga, los datos se separan por comas, se manejan cadenas opcionalmente encerradas entre comillas dobles y cada línea finaliza con un salto de línea. Además, se utiliza la función "STR_TO_DATE" para convertir el formato de la fecha de nacimiento ("birth_date") en un formato estándar (DATE) compatible con la base de datos, eliminando previamente las comillas con el comando "TRIM".

201,lola,Powers,018-139-4717,ante.blandit@outlook.eu,"Mar 20, 2000",Canada,Rigolet,V6T 6M7,154-5415 Auctor St.
--

Esto asegura la consistencia y la integridad de los datos importados.

Tabla *transactions*:

```
100 • CREATE TABLE IF NOT EXISTS transactions (  
101     id VARCHAR(255) PRIMARY KEY,  
102     card_id VARCHAR(40) NOT NULL,  
103     business_id VARCHAR(40) NOT NULL,  
104     timestamp TIMESTAMP NOT NULL,  
105     amount DECIMAL(10, 2) NOT NULL,  
106     declined BOOLEAN NOT NULL,  
107     product_ids VARCHAR(40) NULL,  
108     user_id INT NOT NULL,  
109     lat FLOAT,  
110     longitude FLOAT,  
111     CONSTRAINT fk_transactions_credit_cards FOREIGN KEY (card_id) REFERENCES credit_cards(id),  
112     CONSTRAINT fk_transactions_companies FOREIGN KEY (business_id) REFERENCES companies(company_id),  
113     CONSTRAINT fk_transactions_users FOREIGN KEY (user_id) REFERENCES users(id),  
114     INDEX idx_card_id (card_id),  
115     INDEX idx_business_id (business_id),  
116     INDEX idx_user_id (user_id)  
117 );  
118  
119 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/transactions.csv'  
120 INTO TABLE transactions  
121 FIELDS TERMINATED BY ','  
122 ENCLOSED BY ''''
```

Output

#	Time	Action	Message	Duration / Fetch
1	19:16:07	CREATE TABLE IF NOT EXISTS transactions (id VARCHAR(255) PRIMARY KEY, card_id VARCHAR(40) NOT NULL, business_id VARCHAR(40) NOT NULL, timestamp TIMESTAMP NOT NULL, amount DECIMAL(10, 2) NOT NULL, declined BOOLEAN NOT NULL, product_ids VARCHAR(40) NULL, user_id INT NOT NULL, lat FLOAT, longitude FLOAT, CONSTRAINT fk_transactions_credit_cards FOREIGN KEY (card_id) REFERENCES credit_cards(id), CONSTRAINT fk_transactions_companies FOREIGN KEY (business_id) REFERENCES companies(company_id), CONSTRAINT fk_transactions_users FOREIGN KEY (user_id) REFERENCES users(id), INDEX idx_card_id (card_id), INDEX idx_business_id (business_id), INDEX idx_user_id (user_id));	0 row(s) affected	0.109 sec
2	19:16:53	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.4/Uploads/transactions.csv' INTO TABLE transactions	587 row(s) affected Records: 587 Deleted: 0 Skipped: 0 Warnings: 0	0.109 sec

La tabla "transactions" se define como la *fact table* del sistema, lo que significa que sirve como el punto central para consolidar y analizar información relacionada con transacciones financieras. Contiene detalles clave de cada transacción, como su identificador único ("id"), el monto ("amount"), la fecha y hora ("timestamp"), así como el estado de la transacción ("declined", que indica si fue aprobada o rechazada).

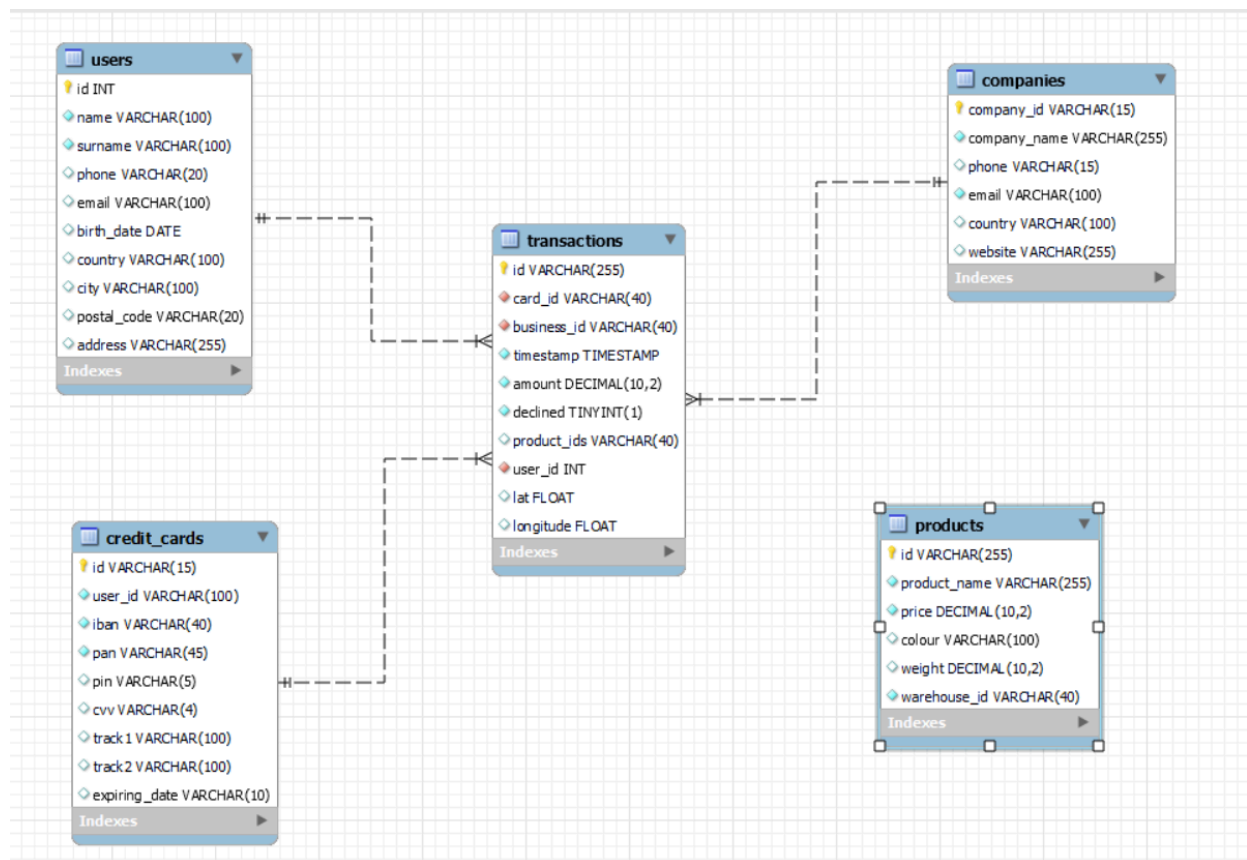
Un aspecto fundamental de esta tabla es la declaración de las claves externas ("foreign keys"), que establecen relaciones con otras tablas del sistema:

- La columna "card_id" está vinculada con la tabla "credit_cards" a través de la clave externa "fk_transactions_credit_cards", lo que permite rastrear qué tarjeta de crédito se utilizó en cada transacción.
- La columna "business_id" referencia la tabla "companies" mediante la clave externa "fk_transactions_companies", conectando cada transacción con la empresa o comercio involucrado.
- La columna "user_id" está relacionada con la tabla "users" mediante la clave externa "fk_transactions_users", lo que permite asociar cada transacción con el usuario que la realizó.

Estas claves externas no sólo garantizan la integridad referencial entre las tablas, sino que también permiten realizar consultas y análisis más complejos al vincular datos de diferentes dimensiones (usuarios, empresas y tarjetas de crédito) de manera estructurada.

Adicionalmente, se incluyen índices en las columnas "card_id", "business_id" y "user_id" para optimizar la velocidad de las consultas relacionadas con estas claves. Finalmente, los campos "lat" y "longitude" permiten almacenar la ubicación geográfica donde se realizó cada transacción, añadiendo un contexto espacial a los datos registrados.

El diagrama del schema *eshop* de momento se presenta así:



Como puede verse, la tabla *products* no tiene conexión con la tabla *transactions* porque las informaciones de la clave primaria de la tabla *products* (*id*) están presente en la tabla *transactions*, pero necesitan una tabla puente para crear una conexión ya que la columna *products_ids* de la tabla *transactions* contiene

más de una información de id separada por una coma y esto impide establecer una conexión:

product_ids
59
71, 41
97, 41, 3
11, 13, 61, 29
47, 37, 11, 1
23, 19, 71
59, 13, 23
47, 67, 31, 5
17, 13, 73

Nivel 1, Ejercicio 1

- Realiza una subconsulta que muestre todos los usuarios con más de 30 transacciones utilizando al menos 2 tablas:

```
128 #Nivel 1, Ejercicio 1
129 #Realiza una subconsulta que muestre todos los usuarios con más de 30 transacciones utilizando al menos 2 tablas:
130
131 • SELECT u.id as users_id, u.name, u.surname
132 FROM users u
133 WHERE u.id IN (
134     SELECT t.user_id
135     FROM transactions t
136     GROUP BY t.user_id
137     HAVING COUNT(t.id) > 30
138 );
139
```

users_id	name	surname
92	Lynn	Riddle
267	Ocean	Nelson
272	Hedwig	Gilbert
275	Kenyon	Hartman

users 1 x Read Only

Output

#	Time	Action	Message	Duration / Fetch
1	19:30:27	SELECT u.id as users_id, u.name, u.surname FROM users u WHERE u.id IN (SELECT t.user_id FROM...	4 row(s) returned	0.016 sec / 0.000 sec

Este código realiza una consulta que muestra todos los usuarios que han realizado más de 30 transacciones. Se utilizan las tablas *users* y *transactions*. La consulta principal selecciona los datos de los usuarios, como su ID, nombre y apellido, de la tabla *users*.

Dentro de esta consulta, hay una subconsulta que trabaja con la tabla *transactions*. Esta subconsulta agrupa las transacciones por el ID del usuario (*t.user_id*) y cuenta cuántas transacciones ha realizado cada usuario. Luego, utiliza la cláusula HAVING para filtrar aquellos usuarios que tienen más de 30 transacciones.

Finalmente, la consulta principal utiliza la condición IN para seleccionar solo los usuarios cuyo ID aparece en el resultado de la subconsulta, es decir, aquellos que han realizado más de 30 transacciones. De esta manera, se obtiene la lista de usuarios que cumplen con este criterio.

Nivel 1, Ejercicio 2

- Muestra la media de amount por IBAN de las tarjetas de crédito a la compañía Donec Ltd, utiliza al menos 2 tablas.

```
143 • SELECT cc.id as credit_card_id, c.company_name, cc.iban, ROUND(AVG(t.amount), 2) AS avg_amount
144 FROM credit_cards cc
145 JOIN transactions t
146 ON cc.id = t.card_id
147 JOIN companies c
148 ON c.company_id = t.business_id
149 WHERE c.company_name = 'Donec Ltd'
150 GROUP BY cc.iban, cc.id;
```

credit_card_id	company_name	iban	avg_amount
CCU-2973	Donec Ltd	PT87806228135092429456346	203.72

Result 4 x

Output

#	Time	Action	Message	Duration / Fetch
1	19:38:59	SELECT cc.id as credit_card_id, c.company_name, cc.iban, ROUND(AVG(t.amount), 2) AS avg_amount FR...	1 row(s) returned	0.000 sec / 0.000 sec

Este código realiza una consulta para mostrar la media del monto (amount) de las transacciones por IBAN de las tarjetas de crédito utilizadas en la compañía llamada "Donec Ltd". La consulta utiliza tres tablas: *credit_cards*, *transactions* y *companies*.

Primero, la consulta selecciona el ID de la tarjeta de crédito (cc.id), el nombre de la compañía (c.company_name), el IBAN de la tarjeta de crédito (cc.iban), y la media de los montos de las transacciones (ROUND(AVG(t.amount), 2) AS avg_amount). La función AVG calcula el promedio de los montos, y ROUND redondea el resultado a dos decimales.

Luego, se realiza una combinación (JOIN) entre la tabla *credit_cards* y *transactions* usando el ID de la tarjeta de crédito (`cc.id = t.card_id`). Después, se hace otro JOIN con la tabla *companies* usando el ID de la empresa (`c.company_id = t.business_id`).

La condición `WHERE c.company_name = 'Donec Ltd'` filtra los resultados para mostrar solo las transacciones asociadas a la compañía "Donec Ltd".

Finalmente, la consulta agrupa los resultados por el IBAN de la tarjeta de crédito y su ID, para que el promedio del monto se calcule por cada tarjeta. De esta manera, se obtiene la media de los montos de las transacciones realizadas con las tarjetas de crédito asociadas a la compañía "Donec Ltd".

Nivel 2, Ejercicio 1

- Crea una nueva tabla que refleje el estado de las tarjetas de crédito basado en si las últimas tres transacciones fueron declinadas y genera la siguiente consulta: ¿Cuántas tarjetas están activas?

```
155 CREATE TABLE card_activity (  
156     card_id VARCHAR(15) PRIMARY KEY,  
157     declined ENUM('YES', 'NO') NOT NULL,  
158     FOREIGN KEY (card_id) REFERENCES credit_cards(id)  
159 );  
160
```

Output

#	Time	Action	Message	Duration / Fetch
1	19:59:20	CREATE TABLE card_activity (card_id VARCHAR(15) PRIMARY KEY, declined ENUM('YES', 'NO') NO...	0 row(s) affected	0.062 sec

El código proporcionado crea una nueva tabla llamada *card_activity* que tiene como objetivo reflejar el estado de las tarjetas de crédito en función de si las últimas tres transacciones fueron declinadas o no. A continuación, explico cada parte del código y cómo cumple con la solicitud del ejercicio.

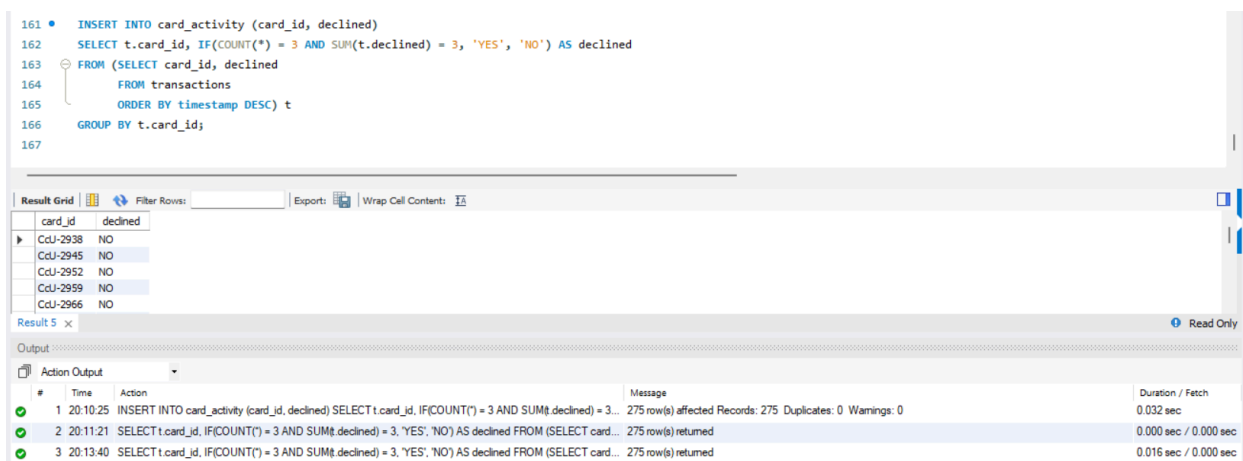
`CREATE TABLE card_activity`: este comando crea una nueva tabla llamada *card_activity*. El nombre refleja que se trata de la actividad o estado de las tarjetas.

`card_id VARCHAR(15) PRIMARY KEY`: la tabla tiene una columna llamada *card_id* que almacena el identificador de la tarjeta de crédito. Esta columna es

de tipo VARCHAR(15), lo que significa que puede almacenar cadenas de texto de hasta 15 caracteres (suficiente para un número de tarjeta de crédito). Esta columna se establece como clave primaria (PRIMARY KEY), lo que significa que no puede haber registros duplicados de tarjetas en la tabla.

declined ENUM('YES', 'NO') NOT NULL: esta columna llamada *declined* almacena el estado de la tarjeta, que se determina en función de si las últimas tres transacciones fueron declinadas o no. El tipo de dato ENUM('YES', 'NO') permite que la columna solo contenga uno de estos dos valores, lo que representa si las transacciones fueron rechazadas (YES) o aceptadas (NO). La restricción NOT NULL asegura que siempre haya un valor en esta columna.

FOREIGN KEY (card_id) REFERENCES credit_cards(id): Se establece una clave externa (FOREIGN KEY) en la columna *card_id*, que referencia el campo *id* de la tabla *credit_cards*. Esto asegura que solo se puedan insertar en la tabla *card_activity* tarjetas que existan previamente en la tabla *credit_cards*, lo que mantiene la integridad referencial entre ambas tablas.



The screenshot displays a SQL query in a text editor, its execution results in a grid, and the database's action log.

```
161 INSERT INTO card_activity (card_id, declined)
162 SELECT t.card_id, IF(COUNT(*) = 3 AND SUM(t.declined) = 3, 'YES', 'NO') AS declined
163 FROM (SELECT card_id, declined
164       FROM transactions
165       ORDER BY timestamp DESC) t
166 GROUP BY t.card_id;
167
```

The **Result Grid** shows the following data:

card_id	declined
CdJ-2938	NO
CdJ-2945	NO
CdJ-2952	NO
CdJ-2959	NO
CdJ-2966	NO

The **Action Output** pane shows the execution of three statements:

#	Time	Action	Message	Duration / Fetch
1	20:10:25	INSERT INTO card_activity (card_id, declined) SELECT t.card_id, IF(COUNT(*) = 3 AND SUM(t.declined) = 3...	275 row(s) affected Records: 275 Duplicates: 0 Warnings: 0	0.032 sec
2	20:11:21	SELECT t.card_id, IF(COUNT(*) = 3 AND SUM(t.declined) = 3, 'YES', 'NO') AS declined FROM (SELECT card...	275 row(s) returned	0.000 sec / 0.000 sec
3	20:13:40	SELECT t.card_id, IF(COUNT(*) = 3 AND SUM(t.declined) = 3, 'YES', 'NO') AS declined FROM (SELECT card...	275 row(s) returned	0.016 sec / 0.000 sec

La consulta SQL tiene como objetivo analizar el comportamiento de las tarjetas de crédito registradas en la tabla "transactions" y determinar si, para cada tarjeta, las últimas tres transacciones fueron rechazadas.

En la subconsulta interna, se seleccionan las columnas *card_id* y *declined* de la tabla *transactions* y se ordenan las transacciones por la columna *timestamp* en orden descendente ("ORDER BY timestamp DESC"). Esto asegura que las transacciones más recientes sean evaluadas primero.

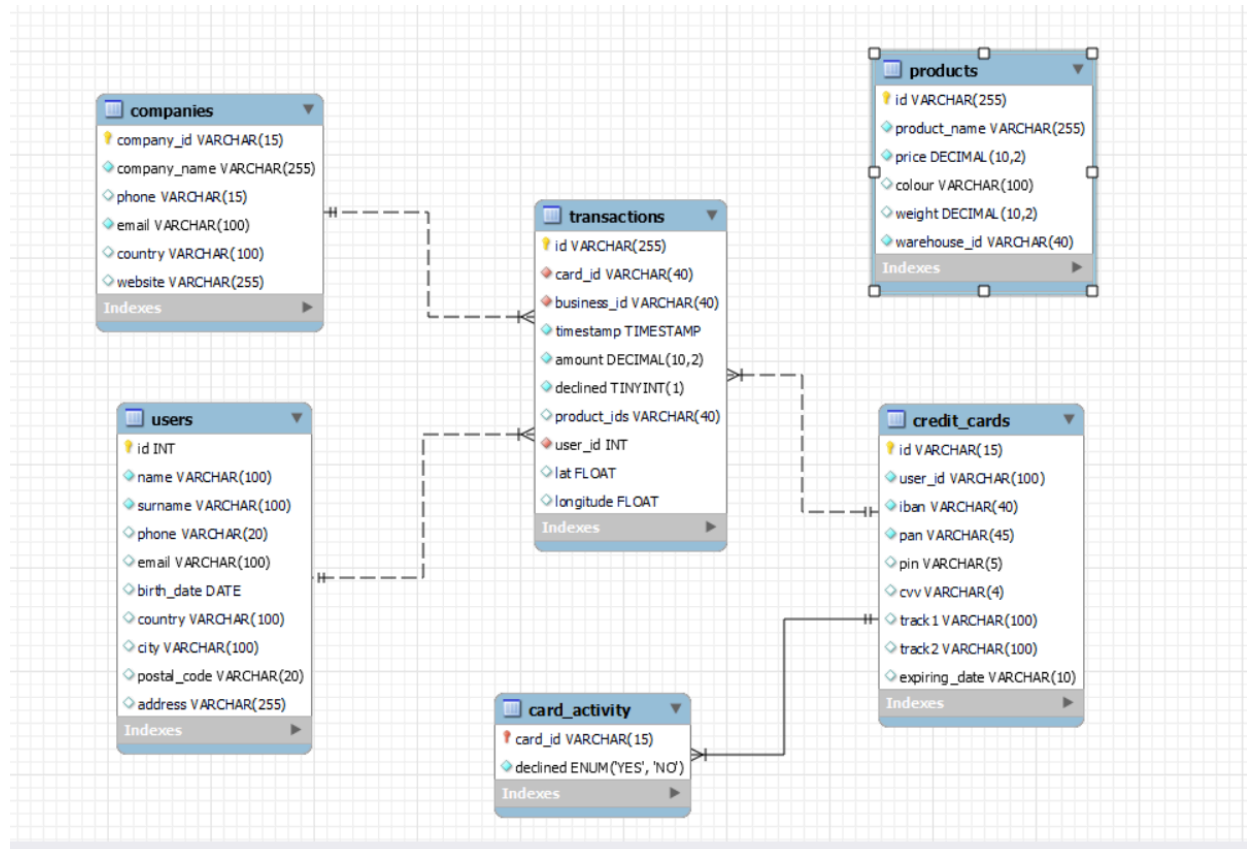
La consulta externa agrupa los datos por *card_id* ("GROUP BY t.card_id"), analizando cada tarjeta de manera independiente. Para cada tarjeta, se calculan dos valores:

- La función "COUNT(*)" cuenta el número total de transacciones evaluadas.
- La función "SUM(t.declined)" suma los valores de la columna "declined", que representa el estado de rechazo de las transacciones (0 para aprobada, 1 para rechazada).

La condición "IF(COUNT() = 3 AND SUM(t.declined) = 3, 'YES', 'NO')" evalúa si existen exactamente tres transacciones recientes ("COUNT() = 3") y si todas ellas fueron rechazadas ("SUM(t.declined) = 3"). Si ambas condiciones se cumplen, se devuelve "YES"; de lo contrario, "NO".

El resultado de esta consulta evidencia que no existen tarjetas con tres transacciones consecutivas rechazadas aplicando los criterios indicados, así que las 275 están activas (no "declined").

Esto es como se muestra el diagrama después de introducir la tabla *card_activity*:



Nivel 3, Ejercicio 1

- Crea una tabla con la cual podamos unir los datos del nuevo archivo products.csv con la base de datos creada, teniendo en cuenta que desde transactions tenemos product_ids. Genera la siguiente consulta: necesitamos conocer el número de veces que se ha vendido cada producto.

Actualmente, la columna *product_ids* en la tabla *transactions* parece violar la regla de normalización (ya que contiene múltiples valores separados por comas). Una solución podría ser crear una tabla de asociación.

Empezamos creando la tabla *sold_products*:

```
171 CREATE TABLE sold_products (  
172     transaction_id VARCHAR(255),  
173     product_id VARCHAR(255),  
174     PRIMARY KEY (transaction_id, product_id),  
175     FOREIGN KEY (transaction_id) REFERENCES transactions(id),  
176     FOREIGN KEY (product_id) REFERENCES products(id)  
177 );  
178
```

Output

#	Time	Action	Message	Duration / Fetch
1	16:57:11	CREATE TABLE sold_products (transaction_id VARCHAR(255), product_id VARCHAR(255), PRIMAR	0 row(s) affected	0.203 sec

Esta nueva tabla almacena la relación entre cada transacción y los productos comprados en esa transacción. Esto permite manejar de manera eficiente los datos originalmente almacenados en la columna *product_ids* de la tabla *transactions*, donde se encontraban como una lista separada por comas.

```
179 INSERT INTO sold_products (transaction_id, product_id)  
180 SELECT t.id AS transaction_id, p.id AS product_id  
181 FROM transactions t  
182 CROSS JOIN products p  
183 WHERE FIND_IN_SET(p.id, t.product_ids) > 0;  
184
```

Output

#	Time	Action	Message	Duration / Fetch
1	17:11:12	INSERT INTO sold_products (transaction_id, product_id) SELECT t.id AS transaction_id, p.id AS product_id F...	587 row(s) affected Records: 587 Duplicates: 0 Warnings: 0	0.063 sec

Este fragmento de código tiene como objetivo poblar la tabla *sold_products* con los datos existentes en las tablas *transactions* y *products*. Para lograrlo, se utiliza una combinación de las dos tablas mediante un CROSS JOIN. A través del CROSS JOIN, cada transacción se evalúa contra cada producto, creando todas las posibles combinaciones entre ambas tablas.

La función FIND_IN_SET se utiliza para determinar si un determinado producto está presente en la lista de identificadores de productos (*product_ids*) asociada a una transacción. Dado que en la tabla *transactions* los productos vendidos en cada transacción están almacenados como una lista separada por comas, FIND_IN_SET verifica si el identificador del producto (*p.id*) aparece en esa lista. Si la función devuelve un valor mayor que cero, significa que el producto está efectivamente asociado a la transacción. Esto permite filtrar las combinaciones generadas por el CROSS JOIN y conservar únicamente aquellas que representan relaciones reales entre productos y transacciones.

Finalmente, el resultado de esta operación es una lista de pares *transaction_id* y *product_id* que se insertan en la tabla *sold_products*. Cada par indica que un

producto específico fue vendido como parte de una transacción particular. Este enfoque asegura que la relación muchos-a-muchos entre transacciones y productos quede completamente representada en la nueva tabla, normalizando los datos y eliminando la necesidad de almacenar listas de valores en una única columna.

185 • `SELECT * FROM sold_products;`

transaction_id	product_id
7D2C6E68-5951-C198-A567-75D46E7CDD0	13
88A92E6E-863C-A9AA-D832-846BF23597CE	13
8F54691C-221E-8B4E-4954-5373EE2C959E	13
B9D1E653-C396-DBE8-09E6-D754A877104E	13
BF0BA288-355D-412E-089C-8152516DC965	13

Output:

#	Time	Action	Message	Duration / Fetch
1	17:11:12	INSERT INTO sold_products (transaction_id, product_id) SELECT t.id AS transaction_id, p.id AS product_id ...	587 row(s) affected Records: 587 Duplicates: 0 Warnings: 0	0.063 sec
2	17:21:54	SELECT * FROM sold_products LIMIT 0, 1000	587 row(s) returned	0.000 sec / 0.000 sec

Después haber averiguado que los valores se cargaron correctamente en la tabla *sold_products*, eliminamos la columna no normalizada *products_ids* desde la tabla *transactions* para que no quede redundancia:

187 • `ALTER TABLE transactions DROP COLUMN product_ids;`

Output:

#	Time	Action	Message	Duration / Fetch
1	17:28:22	ALTER TABLE transactions DROP COLUMN product_ids	0 row(s) affected Records: 0 Duplicates: 0 Warnings: 0	0.047 sec

Ahora solo queda contestar a la pregunta del ejercicio sobre el número de veces que se han vendido los productos. Para hacerlo utilizaremos este código:

189 • `SELECT p.id AS product_id, p.product_name, COUNT(sp.transaction_id) AS num_sales`
 190 `FROM products p`
 191 `LEFT JOIN sold_products sp`
 192 `ON p.id = sp.product_id`
 193 `LEFT JOIN transactions t`
 194 `ON sp.transaction_id = t.id AND t.declined = 0`
 195 `GROUP BY p.id, p.product_name`
 196 `ORDER BY num_sales DESC;`

product_id	product_name	num_sales
23	riverlands north	32
79	Direwolf riverlands the	31
7	north of Casterly	30
43	duel	29
61	Winterfell Lannister	28
53	kingsblood Littlefinger the	26
47	Tully	26
17	Winterfell Stark the	26

Output:

#	Time	Action	Message	Duration / Fetch
1	18:21:18	SELECT p.id AS product_id, p.product_name, COUNT(sp.transaction_id) AS num_sales FROM products p LE...	100 row(s) returned	0.016 sec / 0.000 sec

El código realiza una consulta que tiene como objetivo obtener información sobre el número de veces que se ha vendido cada producto, incluyendo aquellos que no han sido vendidos. Para lograrlo, se utilizan dos LEFT JOIN que conectan la tabla *products* con las tablas *sold_products* y *transactions*.

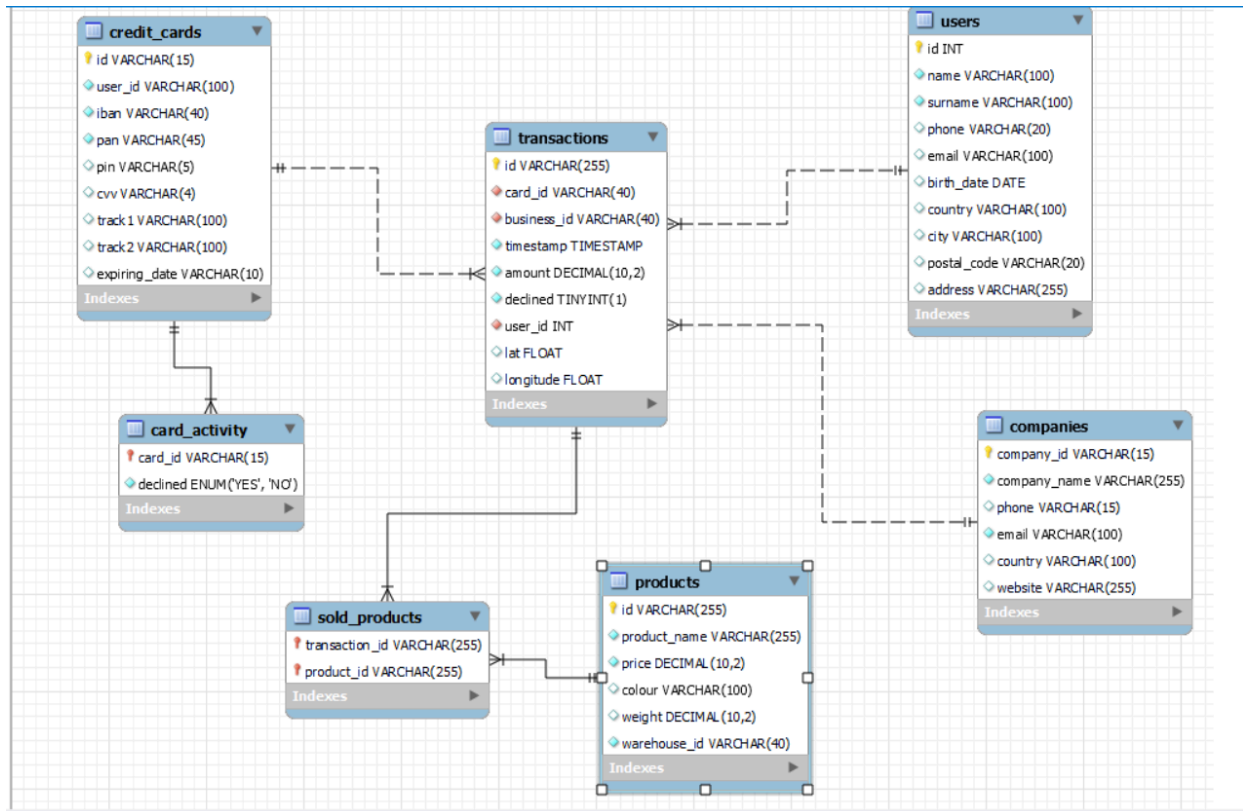
En primer lugar, se toma la tabla *products* como base, lo que asegura que todos los productos estarán presentes en el resultado, independientemente de si han sido vendidos o no. Luego, se utiliza un LEFT JOIN con la tabla *sold_products*, que es la tabla que establece la relación muchos-a-muchos entre los productos y las transacciones. Este paso permite identificar qué transacciones están asociadas a cada producto.

A continuación, se realiza otro LEFT JOIN con la tabla *transactions*, aplicando una condición adicional: solo se incluyen las transacciones que no hayan sido rechazadas, lo que se especifica con *t.declined = 0*. Esto garantiza que las transacciones rechazadas no se consideren en el conteo de las ventas.

El uso de GROUP BY *p.id, p.product_name* permite agrupar los resultados por cada producto, lo que es esencial para calcular el número total de ventas por producto. Finalmente, ORDER BY *num_sales* DESC organiza los productos en orden descendente según el número de ventas, mostrando primero aquellos que tienen el mayor número de transacciones exitosas asociadas.

En resumen, esta consulta combina la información de varias tablas para generar un resultado completo, que incluye tanto los productos vendidos como aquellos que no tienen transacciones, asegurando que las transacciones rechazadas no afecten los cálculos.

Ahora que todas las tablas han sido creadas, este es como aparece el diagrama completo:



En este esquema, la tabla *transactions* actúa como la tabla de hechos (fact table) y se conecta con varias tablas de dimensiones. Cada transacción está relacionada con un usuario a través de la columna *user_id* y con una tarjeta de crédito mediante *card_id*. La tabla *sold_products* establece una relación muchos-a-muchos entre las transacciones y los productos vendidos, permitiendo rastrear qué productos están asociados a cada transacción. Las empresas (companies) se relacionan indirectamente con las transacciones mediante el identificador *business_id*. Finalmente, la actividad de las tarjetas (*card_activity*) está vinculada a las tarjetas de crédito (*credit_cards*) y proporciona información adicional sobre el estado de cada tarjeta.

