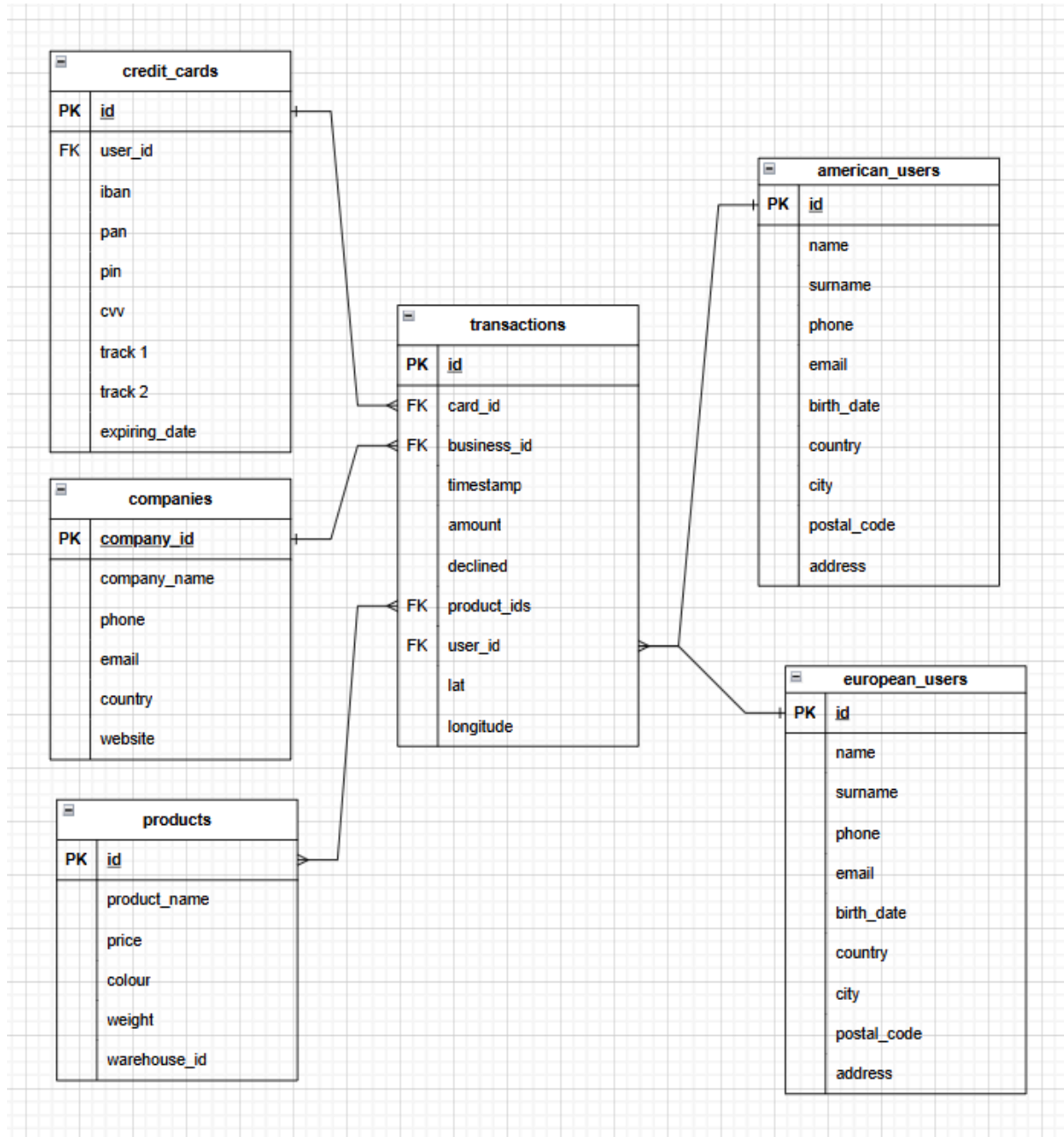


Nivel 1

Descarga los archivos CSV, estudiales y diseña una base de datos con un esquema de estrella que contenga, al menos 4 tablas de las que puedas realizar las siguientes consultas:



La base de datos está compuesta por seis tablas que almacenan información sobre las transacciones realizadas por clientes de América y Europa, incluyendo las empresas, los productos comprados y las tarjetas de crédito utilizadas.

La información de los usuarios estaba inicialmente dividida en dos tablas según la región: american_users y european_users. Ambas compartían la misma estructura y contenían

datos personales (nombre, apellido, fecha de nacimiento) y datos de contacto (teléfono, email, país, ciudad, código postal y dirección). Para simplificar el esquema y facilitar las consultas generales, se unificaron en una sola tabla users, añadiendo una columna region con los valores “american” o “european”.

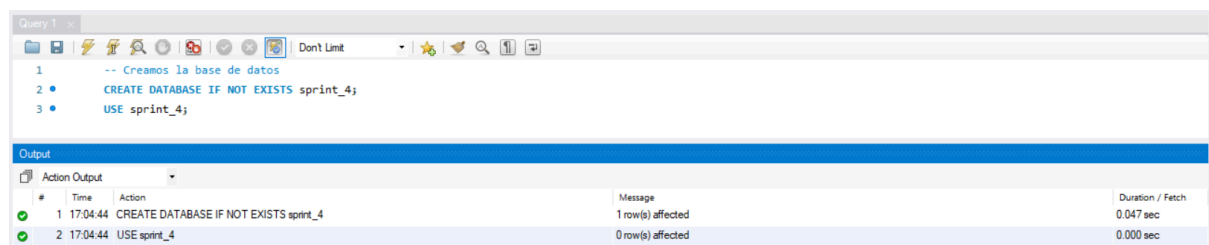
La tabla de dimensión products incluye: id, nombre, precio, color, peso e id del almacén.

La tabla de dimensión companies contiene: id, nombre, teléfono, email, país y página web.

La tabla de dimensión credit_cards contiene: id, id de usuario, IBAN, PAN, PIN, CVV, track1, track2 y fecha de caducidad.

La tabla de hechos transactions conecta todas las dimensiones y almacena la información de cada transacción: id, timestamp, amount, indicador de si fue declinada, coordenadas (latitud y longitud), y claves foráneas: card_id, business_id, product_ids y user_id.

Primero, se crea la **base de datos**:



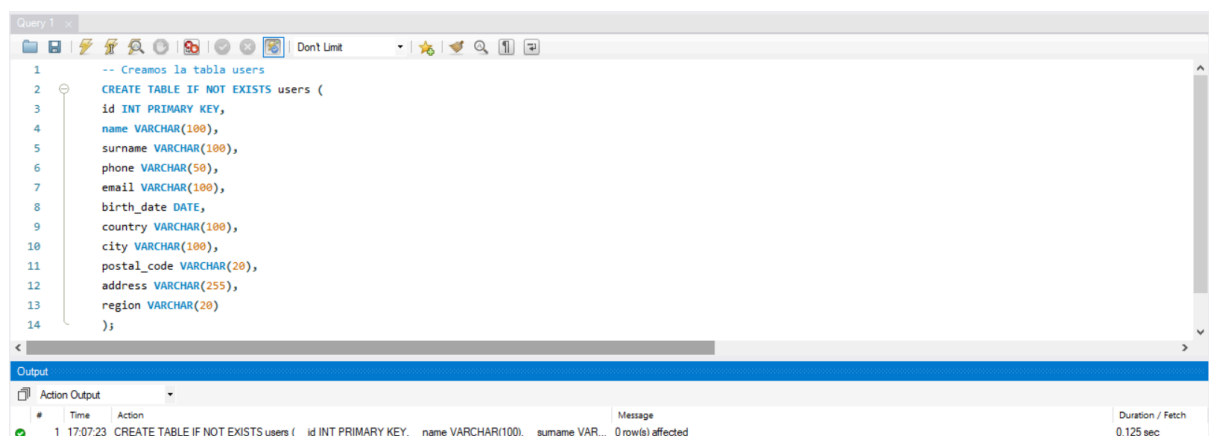
```
Query 1
-- Creamos la base de datos
1 CREATE DATABASE IF NOT EXISTS sprint_4;
2 USE sprint_4;
```

#	Time	Action	Message	Duration / Fetch
1	17:04:44	CREATE DATABASE IF NOT EXISTS sprint_4	1 row(s) affected	0.047 sec
2	17:04:44	USE sprint_4	0 row(s) affected	0.000 sec

A continuación, se crean las tablas correspondientes. Las tablas american_users y european_users comparten la misma estructura y sus IDs no se repiten, lo que sugiere que originalmente pertenecían a una sola tabla. Mantenerlas separadas complicaría las consultas generales sobre transactions, ya que habría que unir información de ambas tablas según la región.

Por coherencia del modelo y para simplificar el esquema estrella, decidí unificar estas tablas en una sola tabla users e incorporar la columna region, con valores “european” y “american”.

Se procede a crear la tabla **users** con esta nueva columna para mantener la información de origen de cada usuario:



```
Query 1
-- Creamos la tabla users
1 CREATE TABLE IF NOT EXISTS users (
2   id INT PRIMARY KEY,
3   name VARCHAR(100),
4   surname VARCHAR(100),
5   phone VARCHAR(50),
6   email VARCHAR(100),
7   birth_date DATE,
8   country VARCHAR(100),
9   city VARCHAR(100),
10  postal_code VARCHAR(20),
11  address VARCHAR(255),
12  region VARCHAR(20)
13 );
```

#	Time	Action	Message	Duration / Fetch
1	17:07:23	CREATE TABLE IF NOT EXISTS users (id INT PRIMARY KEY, name VARCHAR(100), surname VAR...	0 row(s) affected	0.125 sec

Seguimos con las demás tablas.

companies:

The screenshot shows a SQL editor window titled "SQL File 3*" with a toolbar and a "Don't Limit" button. The SQL code is as follows:

```
1 -- Creamos la tabla companies
2 CREATE TABLE IF NOT EXISTS companies (
3     id VARCHAR(100) PRIMARY KEY,
4     company_name VARCHAR(255),
5     phone VARCHAR(15),
6     email VARCHAR(100),
7     country VARCHAR(100),
8     website VARCHAR(255)
9 );
```

Below the code editor is an "Output" section with a dropdown menu set to "Action Output". It displays a single execution result:

#	Time	Action	Message	Duration / Fetch
1	14:03:46	CREATE TABLE IF NOT EXISTS companies (id VARCHAR(100) PRIMARY KEY, company_name VA...	0 row(s) affected	0.094 sec

products:

The screenshot shows a SQL editor window titled "Query 1" with a toolbar and a "Don't Limit" button. The SQL code is as follows:

```
1 -- Creamos la tabla products
2 CREATE TABLE IF NOT EXISTS products (
3     id INT PRIMARY KEY,
4     product_name VARCHAR(100),
5     price DECIMAL(10, 2),
6     colour VARCHAR(20),
7     weight DECIMAL(10, 2),
8     warehouse_id VARCHAR(15)
9 );
```

Below the code editor is an "Output" section with a dropdown menu set to "Action Output". It displays a single execution result:

#	Time	Action	Message	Duration / Fetch
1	17:11:15	CREATE TABLE IF NOT EXISTS products (id INT PRIMARY KEY, product_name VARCHAR(100), price DE...	0 row(s) affected	0.093 sec

transactions:

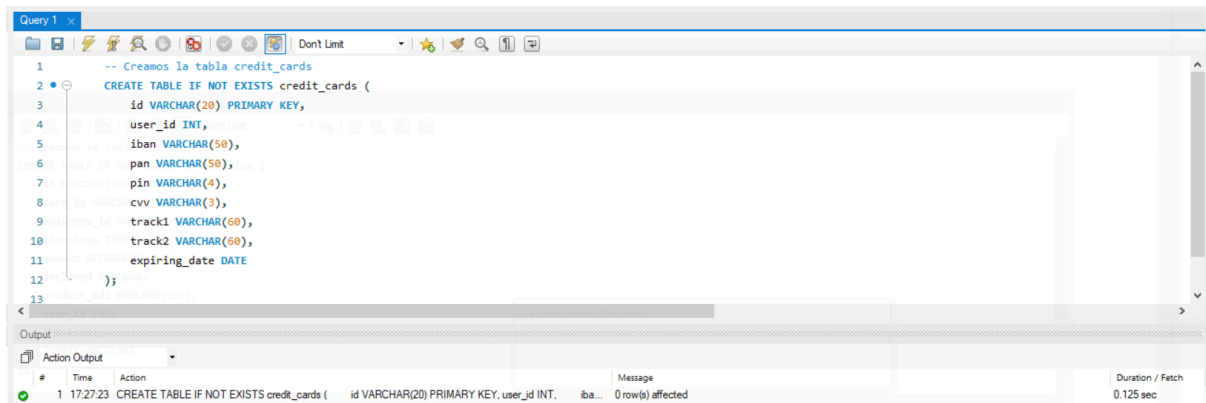
The screenshot shows a SQL editor window titled "SQL File 3*" with a toolbar and a "Don't Limit" button. The SQL code is as follows:

```
1 -- Creamos la tabla transaction
2 CREATE TABLE IF NOT EXISTS transaction (
3     id VARCHAR(100) PRIMARY KEY,
4     card_id VARCHAR(20),
5     business_id VARCHAR(100),
6     timestamp TIMESTAMP,
7     amount DECIMAL(10, 2),
8     declined BOOLEAN,
9     product_ids VARCHAR(255),
10    user_id INT,
11    lat FLOAT,
12    longitude FLOAT
13 );
```

Below the code editor is an "Output" section with a dropdown menu set to "Action Output". It displays a single execution result:

#	Time	Action	Message	Duration / Fetch
1	14:01:14	CREATE TABLE IF NOT EXISTS transaction (id VARCHAR(100) PRIMARY KEY, card_id VARCHAR(...	0 row(s) affected	0.110 sec

credit_cards:

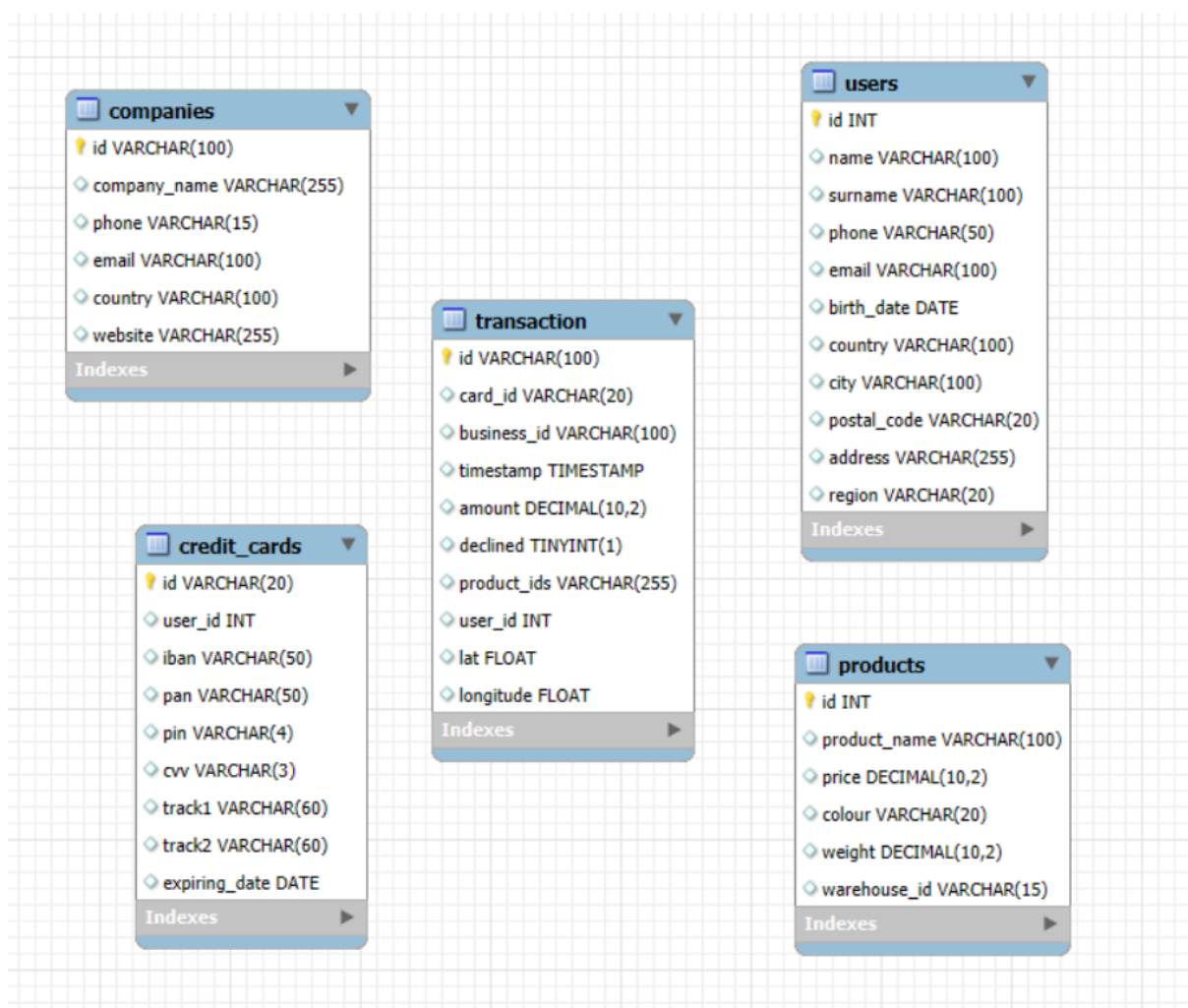


```
1 -- Creamos la tabla credit_cards
2 CREATE TABLE IF NOT EXISTS credit_cards (
3     id VARCHAR(20) PRIMARY KEY,
4     user_id INT,
5     iban VARCHAR(50),
6     pan VARCHAR(50),
7     pin VARCHAR(4),
8     cvv VARCHAR(3),
9     track1 VARCHAR(60),
10    track2 VARCHAR(60),
11    expiring_date DATE
12 );
```

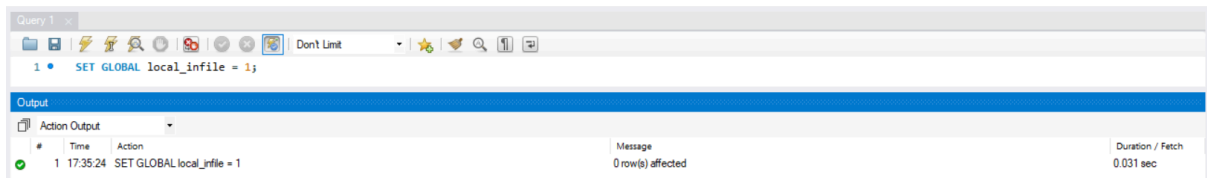
Output

#	Time	Action	Message	Duration / Fetch
1	17:27:23	CREATE TABLE IF NOT EXISTS credit_cards (id VARCHAR(20) PRIMARY KEY, user_id INT, iba...	0 row(s) affected	0.125 sec

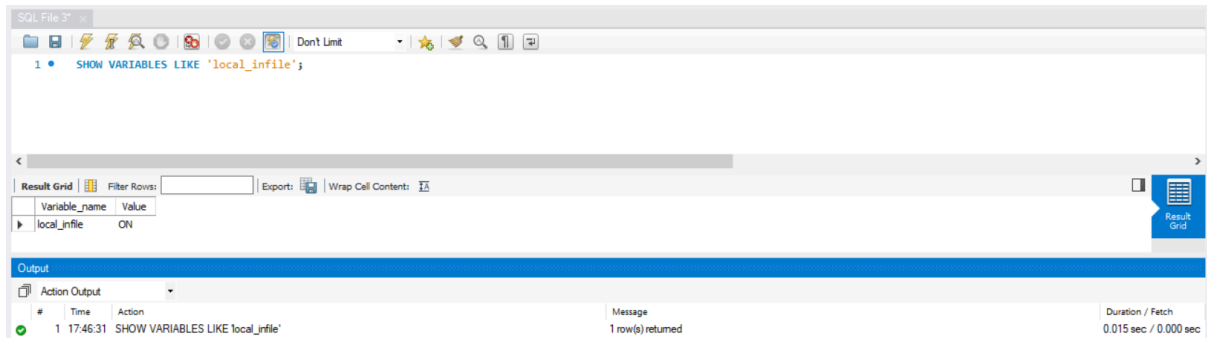
Tras crear todas las tablas revisamos el esquema general de la base de datos. De momento no hemos definido las relaciones mediante claves foráneas, ya que primero importaremos los datos desde los archivos CSV para evitar errores de integridad referencial.



Para poder cargar los datos de los archivos CSV hace falta activar los permisos:

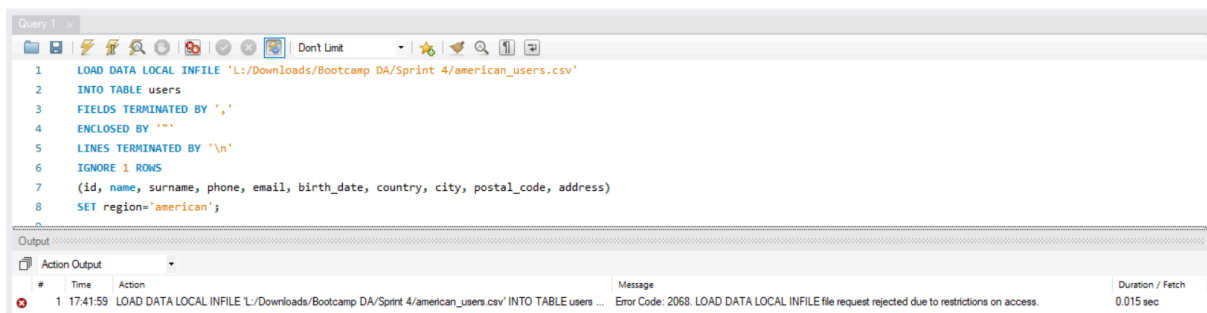


Lo comprobamos:

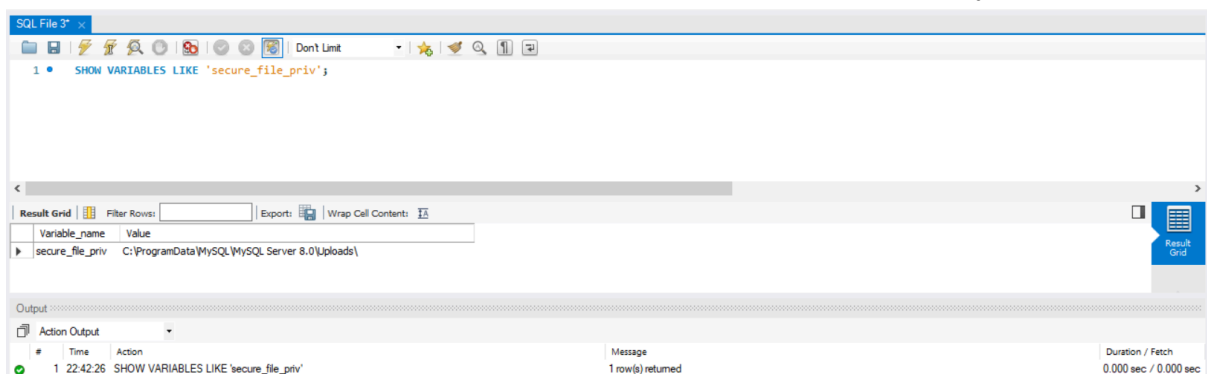


Empezamos cargando los datos de los dos archivos a la tabla users.
american:

Apareció el error 2068: Aunque local_infile = ON, MySQL Workbench bloqueó el uso de archivo local.



Revisé la variable secure_file_priv para conocer la carpeta permitida por MySQL:



Moví todos los CSV a esa carpeta (C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/).
Reemplacé LOAD DATA LOCAL INFILE por LOAD DATA INFILE apuntando a la carpeta segura.

```
1 LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/american_users.csv'
2 INTO TABLE users
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY ''''
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, name, surname, phone, email, birth_date, country, city, postal_code, address)
8 SET region='american';
```

#	Time	Action	Message	Duration / Fetch
1	23:00:42	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/american_users.csv' INTO TABL...	Error Code: 1292. Incorrect date value: 'Nov 17, 1985' for column 'birth_date' at row 1	0.140 sec

Formato original de la fecha: Nov 17, 1985 mientras que el tipo de columna en la tabla es DATE (MySQL requiere YYYY-MM-DD).

```
1 LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/american_users.csv'
2 INTO TABLE users
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY ''''
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, name, surname, phone, email, @birth_date, country, city, postal_code, address)
8 SET birth_date = STR_TO_DATE(@birth_date, '%b %d, %Y'),
9 region='american';
```

#	Time	Action	Message	Duration / Fetch
1	23:06:34	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/american_users.csv' INTO TABL...	1010 row(s) affected. Records: 1010 Deleted: 0 Skipped: 0 Warnings: 0	0.141 sec

STR_TO_DATE() convierte automáticamente el texto al formato correcto durante la importación.

@birth_date es una variable temporal que almacena la cadena del CSV.

('%b %d, %Y') indica a MySQL cómo interpretar ese texto:

- %b → nombre abreviado del mes (Jan, Feb, Mar...)
- %d → día con dos dígitos
- %Y → año con cuatro dígitos

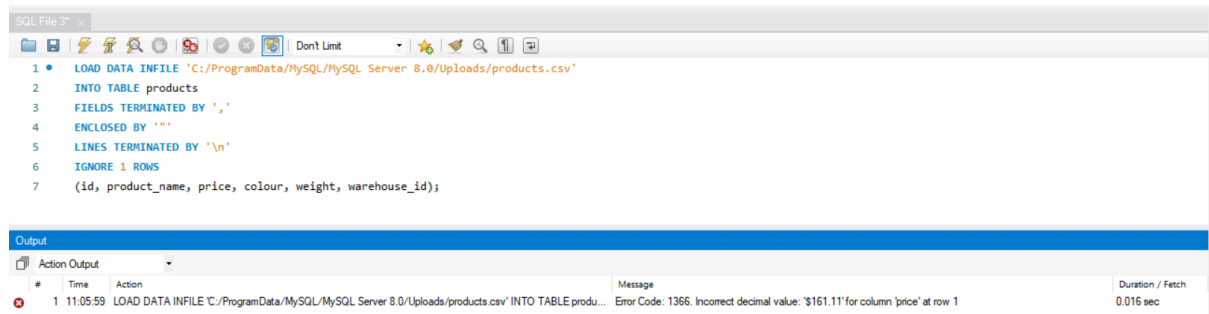
Esto permite cargar los datos sin modificar el archivo original.

european:

```
1 LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/european_users.csv'
2 INTO TABLE users
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY ''''
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, name, surname, phone, email, @birth_date, country, city, postal_code, address)
8 SET birth_date = STR_TO_DATE(@birth_date, '%b %d, %Y'),
9 region='european';
```

#	Time	Action	Message	Duration / Fetch
1	23:08:18	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/european_users.csv' INTO TABL...	3990 row(s) affected. Records: 3990 Deleted: 0 Skipped: 0 Warnings: 0	0.625 sec

products:

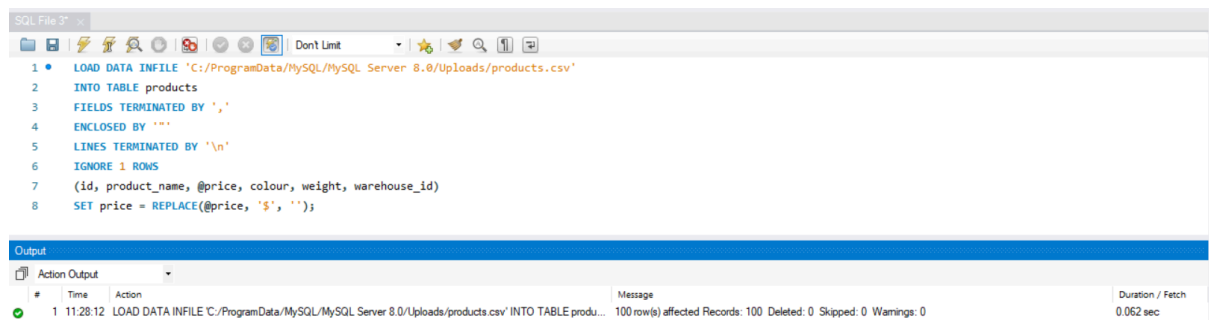


```
1 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/products.csv'
2 INTO TABLE products
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY '"'
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, product_name, price, colour, weight, warehouse_id);
```

Output

#	Time	Action	Message	Duration / Fetch
1	11:05:59	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/products.csv' INTO TABLE products	Error Code: 1366. Incorrect decimal value: '\$161.11' for column 'price' at row 1	0.016 sec

El error 1366 se debe a que MySQL no puede convertir directamente "\$161.11" a un DECIMAL, es el tipo más apropiado porque permite realizar las operaciones aritméticas. Todos los productos están en la misma moneda, por eso podemos ignorar el símbolo y quedarnos solo con el número, no hay necesidad de almacenar la moneda como columna separada.



```
1 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/products.csv'
2 INTO TABLE products
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY '"'
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, product_name, @price, colour, weight, warehouse_id)
8 SET price = REPLACE(@price, '$', '');
```

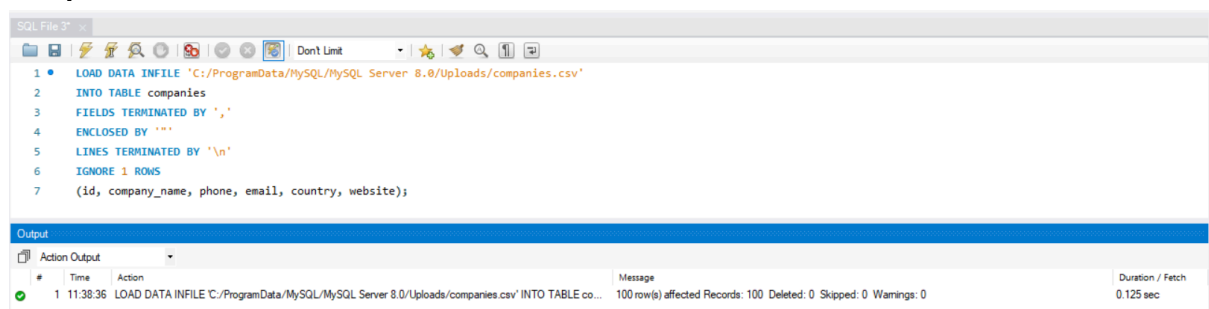
Output

#	Time	Action	Message	Duration / Fetch
1	11:28:12	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/products.csv' INTO TABLE products	100 row(s) affected Records: 100 Deleted: 0 Skipped: 0 Warnings: 0	0.062 sec

La solución es limpiar durante la importación con SET y REPLACE():

- Usamos una variable temporal (@price) que captura el valor del CSV y luego eliminamos el signo \$ antes de insertarlo.
- REPLACE(@price, '\$', '') elimina el \$ antes de insertarlo en la columna price.

companies:

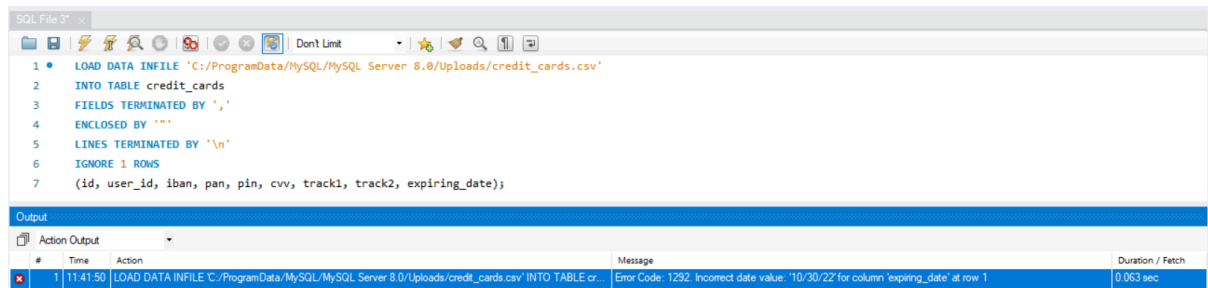


```
1 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/companies.csv'
2 INTO TABLE companies
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY '"'
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, company_name, phone, email, country, website);
```

Output

#	Time	Action	Message	Duration / Fetch
1	11:38:36	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/companies.csv' INTO TABLE companies	100 row(s) affected Records: 100 Deleted: 0 Skipped: 0 Warnings: 0	0.125 sec

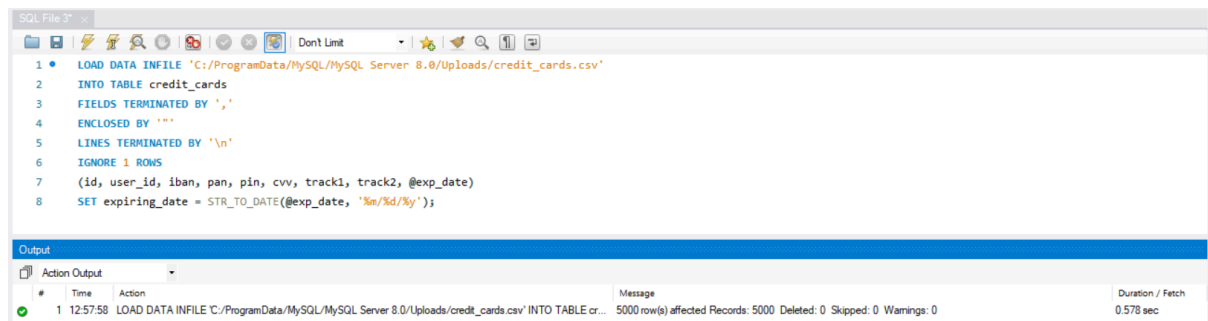
credit_cards:



```
1 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/credit_cards.csv'
2 INTO TABLE credit_cards
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY ''''
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, user_id, iban, pan, pin, cvv, track1, track2, expiring_date);
```

#	Time	Action	Message	Duration / Fetch
1	11:41:50	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/credit_cards.csv' INTO TABLE cr...	Error Code: 1292. Incorrect date value: '10/30/22' for column 'expiring_date' at row 1	0.063 sec

El campo `expiring_date` de la tabla `credit_cards` se definió como `DATE`, ya que conceptualmente todas las fechas deberían almacenarse en este tipo de datos. Sin embargo, los datos del CSV vienen en formato `MM/DD/YY`, que no coincide con el formato que MySQL espera (`YYYY-MM-DD`). Por esta razón, al intentar importar directamente se produjo un error de tipo 1292 (Incorrect date value).



```
1 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/credit_cards.csv'
2 INTO TABLE credit_cards
3 FIELDS TERMINATED BY ','
4 ENCLOSED BY ''''
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, user_id, iban, pan, pin, cvv, track1, track2, @exp_date)
8 SET expiring_date = STR_TO_DATE(@exp_date, '%m/%d/%y');
```

#	Time	Action	Message	Duration / Fetch
1	12:57:58	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/credit_cards.csv' INTO TABLE cr...	5000 row(s) affected. Records: 5000 Deleted: 0 Skipped: 0 Warnings: 0	0.578 sec

Usamos `SET` para convertir la fecha del CSV al formato que MySQL entiende (`YYYY-MM-DD`).

@exp_date:

- Se usa una variable temporal para capturar el valor tal cual viene del CSV.
- Esto evita que MySQL intente insertarlo directamente en la columna `DATE` y falle.

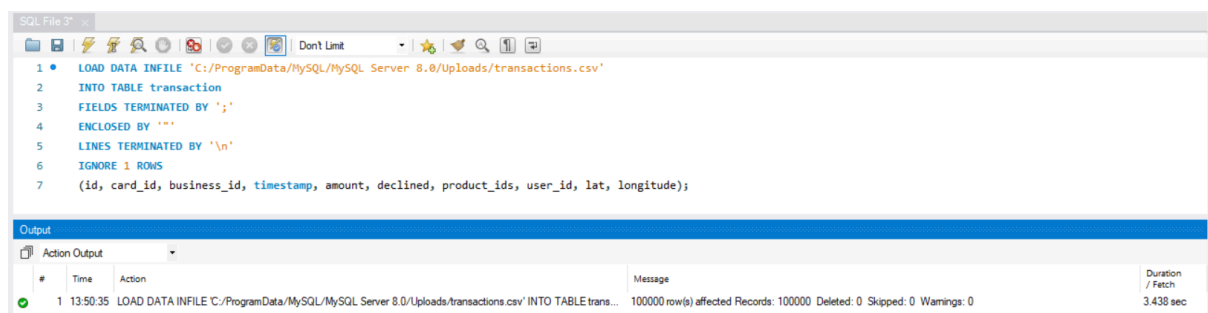
`STR_TO_DATE(@exp_date, '%m/%d/%y')`:

- Convierte la cadena `'10/30/22'` a un valor `DATE '2022-10-30'`.
- `%m` → mes, `%d` → día, `%y` → año de 2 dígitos.

`SET expiring_date = ...`:

- Inserta el valor convertido en la columna `expiring_date` de tipo `DATE`.

transactions:



```
1 • LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/transactions.csv'
2 INTO TABLE transaction
3 FIELDS TERMINATED BY ';'
4 ENCLOSED BY ''''
5 LINES TERMINATED BY '\n'
6 IGNORE 1 ROWS
7 (id, card_id, business_id, timestamp, amount, declined, product_ids, user_id, lat, longitude);
```

#	Time	Action	Message	Duration / Fetch
1	13:50:35	LOAD DATA INFILE 'C:/ProgramData/MySQL/MySQL Server 8.0/Uploads/transactions.csv' INTO TABLE trans...	100000 row(s) affected. Records: 100000 Deleted: 0 Skipped: 0 Warnings: 0	3.438 sec

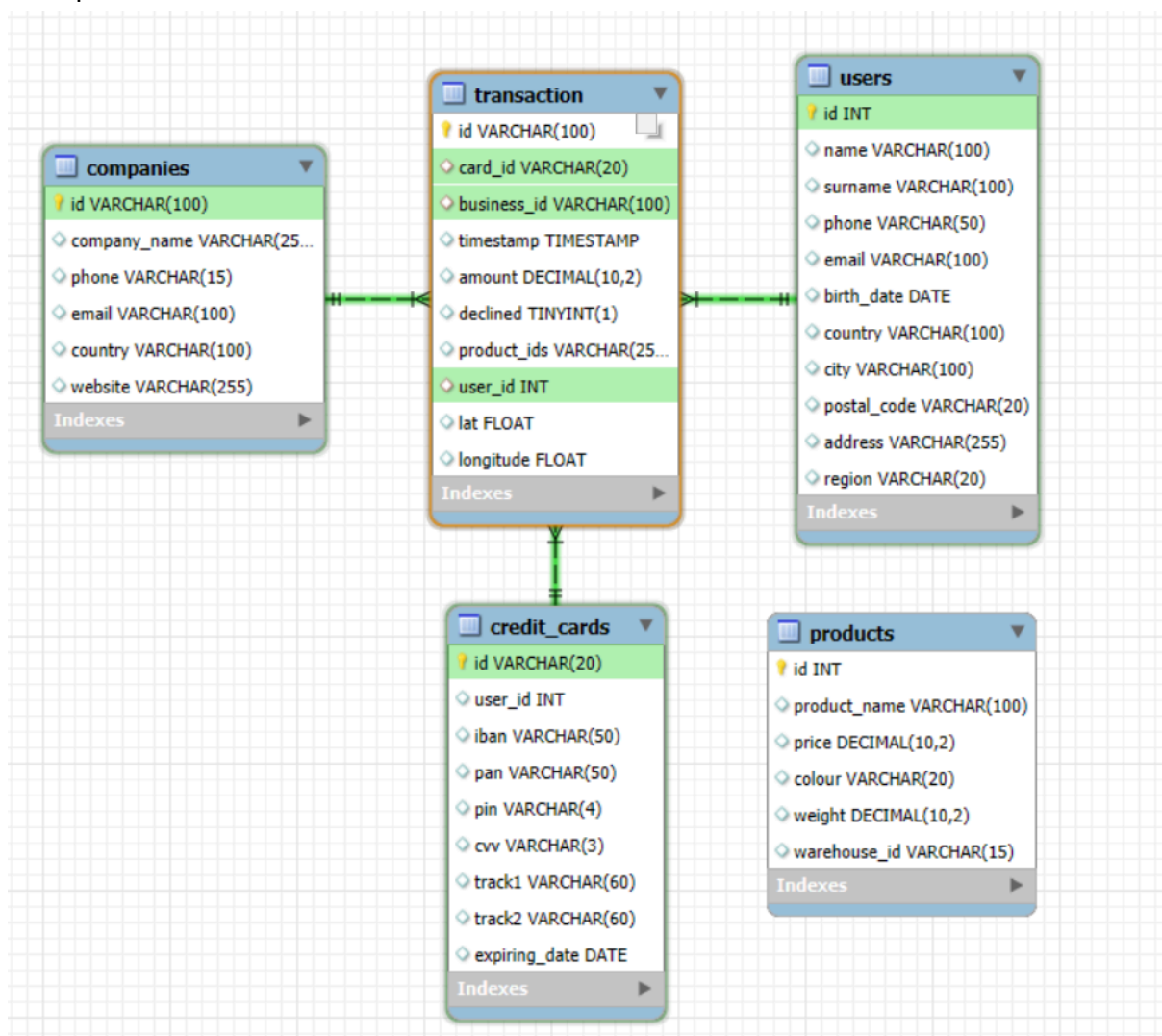
A diferencia de las otras tablas la tabla transactions tiene como delimitador “;” y no “,”. Añadimos las FK a la tabla transaction para conectarla con las demás tablas:

```
SQL File 3"
1 ALTER TABLE transaction
2 ADD CONSTRAINT fk_transaction_card
3 FOREIGN KEY (card_id) REFERENCES credit_cards(id),
4 ADD CONSTRAINT fk_transaction_company
5 FOREIGN KEY (business_id) REFERENCES companies(id),
6 ADD CONSTRAINT fk_transaction_user
7 FOREIGN KEY (user_id) REFERENCES users(id);
```

Output

#	Time	Action	Message	Duration / Fetch
1	14:20:54	ALTER TABLE transaction ADD CONSTRAINT fk_transaction_card FOREIGN KEY (card_id) REFERENCES ...	100000 row(s) affected Records: 100000 Duplicates: 0 Warnings: 0	4.531 sec

El esquema actualizado:



Ejercicio 1

Realiza una subconsulta que muestre a todos los usuarios con más de 80 transacciones utilizando al menos 2 tablas.

The screenshot shows a SQL query in a file named 'SQL File 3*'. The query is as follows:

```
1 SELECT u.id, u.name, u.surname, u.region, u.country
2 FROM users u
3 WHERE u.id IN
4     (SELECT t.user_id
5      FROM transaction t
6      GROUP BY t.user_id
7      HAVING COUNT(t.id)>80);
```

The results are displayed in a table with the following columns: id, name, surname, region, country. The results are:

id	name	surname	region	country
185	Molly	Gillam	european	United Kingdom
289	Dxwgi	Hwicu	european	Germany
318	Bnyr	Astuw	european	Italy
454	Sfzzoh	Xgyfrdxs	european	Poland

The output section shows the execution time and message:

#	Time	Action	Message	Duration / Fetch
1	15:08:29	SELECT u.id, u.name, u.surname, u.region, u.country FROM users u WHERE u.id IN (SELECT t.user_id FROM transaction t GROUP BY t.user_id HAVING COUNT(t.id)>80);	4 row(s) returned	0.047 sec / 0.000 sec

Para realizar esta consulta necesitamos la tabla users para obtener la información relevante como, por ejemplo, el nombre, apellido, región y país. El enunciado nos pide sólo los usuarios que tienen más de 80 transacciones, por tanto en la consulta interna hacemos recuento de las transacciones y filtramos aquellos usuarios que cumplen con la condición.

Ejercicio 2

Muestra la media de amount por IBAN de las tarjetas de crédito en la compañía Donec Ltd., utiliza por lo menos 2 tablas.

The screenshot shows a SQL query in a file named 'SQL File 3*'. The query is as follows:

```
1 SELECT c.iban, ROUND(AVG(t.amount),2) AS transaccion_media
2 FROM credit_cards c
3 JOIN transaction t ON c.id=t.card_id
4 JOIN companies co ON co.id=t.business_id
5 WHERE co.company_name= 'Donec Ltd'
6 GROUP BY c.iban;
```

The results are displayed in a table with the following columns: iban, transaccion_media. The results are:

iban	transaccion_media
XX911406401125586307586805	356.25
SK9446370242474562577506	142.96
XX776752917845952975555640	257.37
XX413827362289719304908990	139.59
XX347787246070769610780308	240.41
XX688768436543090894854602	188.58
MC28368851538688349	439.39
PL76249283566852676343404576	541.56
XX804008331134918341803913	189.21
LB6465553777363327873049938	155.50

The output section shows the execution time and message:

#	Time	Action	Message	Duration / Fetch
1	15:39:39	SELECT c.iban, ROUND(AVG(t.amount),2) AS transaccion_media FROM credit_cards c JOIN transaction t ON c.id=t.card_id JOIN companies co ON co.id=t.business_id WHERE co.company_name= 'Donec Ltd' GROUP BY c.iban;	371 row(s) returned	0.000 sec / 0.000 sec

Para este ejercicio usamos tres tablas: companies para filtrar la empresa Donec Ltd, credit_cards para obtener el IBAN de cada tarjeta, y transaction para calcular la media de los importes.

Unimos las tablas con JOIN, filtramos por la compañía y agrupamos por IBAN para obtener la media del amount.

Nivel 2

Crea una nueva tabla que refleje el estado de las tarjetas de crédito basado en *si las tres últimas transacciones han sido declinadas entonces es inactivo, si al menos una no es rechazada entonces es activo* . Partiendo de esta tabla responde:

Ejercicio 1

¿Cuántas tarjetas están activas?

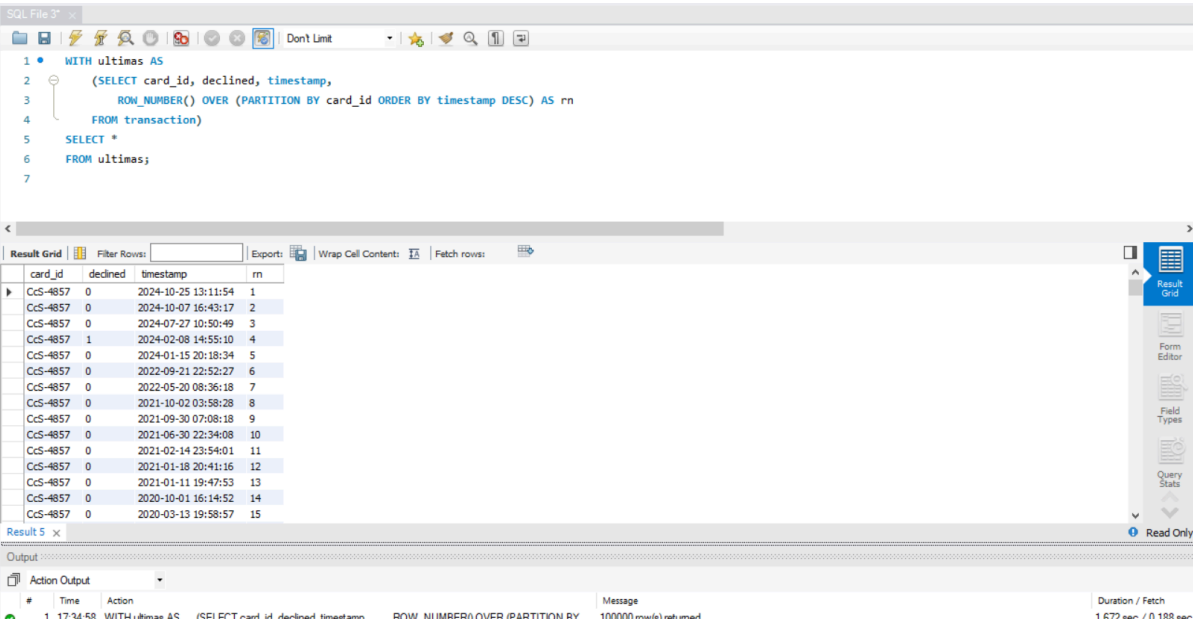
Dividimos el problema en varios pasos.

1. Identificación de las transacciones más recientes

Primero creamos la CTE ultimas, donde organizamos todas las transacciones por tarjeta, ordenándolas de la más reciente a la más antigua.

Para ello utilizamos la función de ventana ROW_NUMBER(), particionada por card_id y ordenada por timestamp DESC.

Esto nos permite enumerar cada transacción dentro de su tarjeta y saber cuál es la 1ª, 2ª, 3ª, etc.



The screenshot shows a SQL query in a file named 'SQL File 3*'. The query is as follows:

```
1 WITH ultimas AS
2   (SELECT card_id, declined, timestamp,
3    ROW_NUMBER() OVER (PARTITION BY card_id ORDER BY timestamp DESC) AS rn
4    FROM transaction)
5 SELECT *
6 FROM ultimas;
```

The results are displayed in a table with the following columns: card_id, declined, timestamp, and rn. The table contains 15 rows of data, showing transactions for card_id 'Cc5-4857'.

card_id	declined	timestamp	rn
Cc5-4857	0	2024-10-25 13:11:54	1
Cc5-4857	0	2024-10-07 16:43:17	2
Cc5-4857	0	2024-07-27 10:50:49	3
Cc5-4857	1	2024-02-08 14:55:10	4
Cc5-4857	0	2024-01-15 20:18:34	5
Cc5-4857	0	2022-09-21 22:52:27	6
Cc5-4857	0	2022-05-20 08:36:18	7
Cc5-4857	0	2021-10-02 03:58:28	8
Cc5-4857	0	2021-09-30 07:08:18	9
Cc5-4857	0	2021-06-30 22:34:08	10
Cc5-4857	0	2021-02-14 23:54:01	11
Cc5-4857	0	2021-01-18 20:41:16	12
Cc5-4857	0	2021-01-11 19:47:53	13
Cc5-4857	0	2020-10-01 16:14:52	14
Cc5-4857	0	2020-03-13 19:58:57	15

The bottom of the screenshot shows the 'Output' section with the following message:

```
1 17:34:58 WITH ultimas AS (SELECT card_id, declined, timestamp, ROW_NUMBER() OVER (PARTITION BY ... 100000 row(s) returned
```

The duration of the query execution is 1.672 sec / 0.188 sec.

2. Selección de las tres últimas transacciones

Como el ejercicio pide evaluar el estado de la tarjeta basándose solo en las tres últimas transacciones, creamos la CTE tres, filtrando aquellas transacciones donde $rn \leq 3$. De este modo obtenemos exactamente las tres transacciones más recientes por tarjeta.

```
1 WITH ultimas AS
2   (SELECT card_id, declined, timestamp,
3    ROW_NUMBER() OVER (PARTITION BY card_id ORDER BY timestamp DESC) AS rn
4   FROM transaction),
5   tres AS
6   (SELECT *
7    FROM ultimas
8   WHERE rn <= 3)
9   SELECT *
10  FROM tres;
```

Result Grid

card_id	declined	timestamp	rn
CcS-4857	0	2024-10-25 13:11:54	1
CcS-4857	0	2024-10-07 16:43:17	2
CcS-4857	0	2024-07-27 10:50:49	3
CcS-4858	0	2024-08-28 13:48:22	1
CcS-4858	0	2023-12-10 10:56:40	2
CcS-4858	0	2023-12-10 09:21:55	3
CcS-4859	0	2024-06-15 21:04:11	1
CcS-4859	0	2023-11-11 15:18:56	2
CcS-4859	0	2023-03-11 06:20:59	3
CcS-4860	0	2024-10-02 08:31:13	1
CcS-4860	0	2024-06-06 18:23:47	2
CcS-4860	0	2023-07-14 06:59:27	3
CcS-4861	0	2024-07-30 10:30:48	1

Output

#	Time	Action	Message	Duration / Fetch
1	17:40:27	WITH ultimas AS (SELECT card_id, declined, timestamp, ROW_NUMBER() OVER (PARTITION BY ...	15000 row(s) returned	1.062 sec / 0.063 sec

3. Determinación del estado (“activo” vs “inactivo”)

A partir de estas tres transacciones, calculamos el estado de cada tarjeta:

- si las 3 últimas transacciones fueron declinadas → inactivo
- si al menos una fue aceptada → activo

```
1 WITH ultimas AS
2   (SELECT card_id, declined, timestamp,
3    ROW_NUMBER() OVER (PARTITION BY card_id ORDER BY timestamp DESC) AS rn
4   FROM transaction),
5   tres AS
6   (SELECT *
7    FROM ultimas
8   WHERE rn <= 3)
9   SELECT card_id,
10  CASE
11    WHEN SUM(declined) = 3 THEN 'inactivo'
12    ELSE 'activo'
13  END AS estado
14 FROM tres
15 GROUP BY card_id;
```

Result Grid

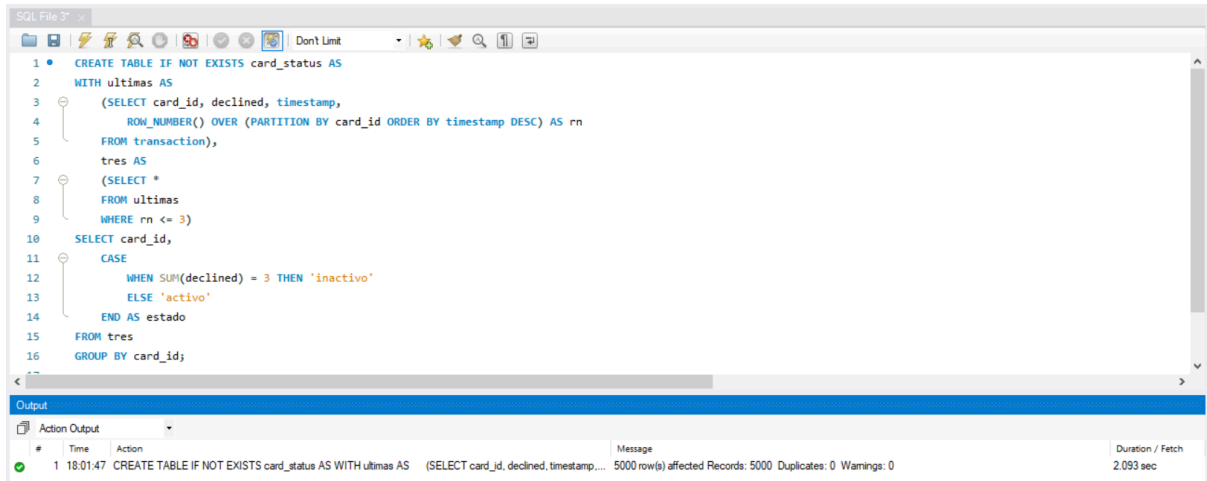
card_id	estado
CcS-4857	activo
CcS-4858	activo
CcS-4859	activo
CcS-4860	activo
CcS-4861	activo
CcS-4862	activo
CcS-4863	activo
CcS-4864	activo
CcS-4865	activo

Output

#	Time	Action	Message	Duration / Fetch
1	18:00:22	WITH ultimas AS (SELECT card_id, declined, timestamp, ROW_NUMBER() OVER (PARTITION BY ...	5000 row(s) returned	0.953 sec / 0.016 sec

4. Creación de la nueva tabla solicitada (card_status)

Como el ejercicio indica explícitamente que debe crearse una nueva tabla, generamos la tabla card_status a partir de CTE para reflejar el estado final de cada tarjeta según sus transacciones más recientes.



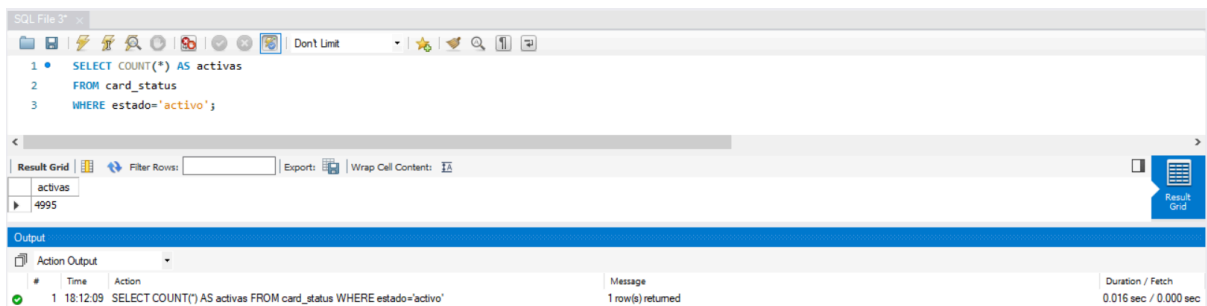
```
1 CREATE TABLE IF NOT EXISTS card_status AS
2 WITH ultimas AS
3     (SELECT card_id, declined, timestamp,
4      ROW_NUMBER() OVER (PARTITION BY card_id ORDER BY timestamp DESC) AS rn
5      FROM transaction),
6 tres AS
7     (SELECT *
8      FROM ultimas
9      WHERE rn <= 3)
10 SELECT card_id,
11        CASE
12            WHEN SUM(declined) = 3 THEN 'inactivo'
13            ELSE 'activo'
14        END AS estado
15 FROM tres
16 GROUP BY card_id;
```

Output

#	Time	Action	Message	Duration / Fetch
1	18:01:47	CREATE TABLE IF NOT EXISTS card_status AS WITH ultimas AS (SELECT card_id, declined, timestamp...	5000 row(s) affected Records: 5000 Duplicates: 0 Warnings: 0	2.093 sec

5. Consulta final

Para averiguar cuántas tarjetas están activas, consultamos la tabla creada “card_status”:



```
1 SELECT COUNT(*) AS activas
2 FROM card_status
3 WHERE estado='activo';
```

Result Grid

activas
4995

Output

#	Time	Action	Message	Duration / Fetch
1	18:12:09	SELECT COUNT(*) AS activas FROM card_status WHERE estado='activo'	1 row(s) returned	0.016 sec / 0.000 sec

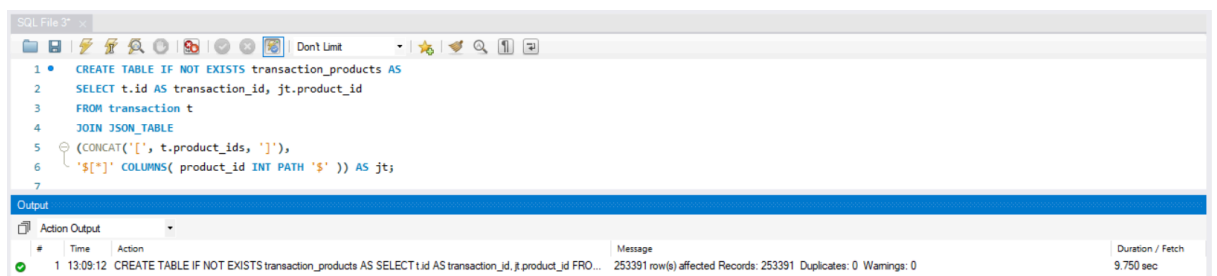
Nivel 3

Crea una tabla con la que podamos unir los datos del nuevo archivo **products.csv** con la base de datos creada, teniendo en cuenta que desde **transaction** tienes **product_ids**. Genera la siguiente consulta:

Ejercicio 1

Necesitamos conocer el número de veces que se ha vendido cada producto.

1. Creamos una tabla intermedia **transaction_products** para poder descomponer los **product_ids** de la tabla **transaction**.
 - a. Convertimos la columna **product_ids** a formato JSON válido, pasando de valores como 75, 73, 98 a [75, 73, 98] usando **CONCAT**, para que **JSON_TABLE** pueda interpretarlos como un array.
 - b. **JSON_TABLE** genera una fila por cada producto dentro de cada transacción:
 - o '\$[*]' recorre todos los elementos del array.
 - o **product_id INT PATH '\$'** crea una columna con cada ID.
 - c. Hacemos el **JOIN** correspondiente para crear la tabla que relaciona cada transacción con sus productos.



```
1 CREATE TABLE IF NOT EXISTS transaction_products AS
2 SELECT t.id AS transaction_id, jt.product_id
3 FROM transaction t
4 JOIN JSON_TABLE
5 (CONCAT('[', t.product_ids, ']'),
6 '$[*]' COLUMNS( product_id INT PATH '$' )) AS jt;
```

Output

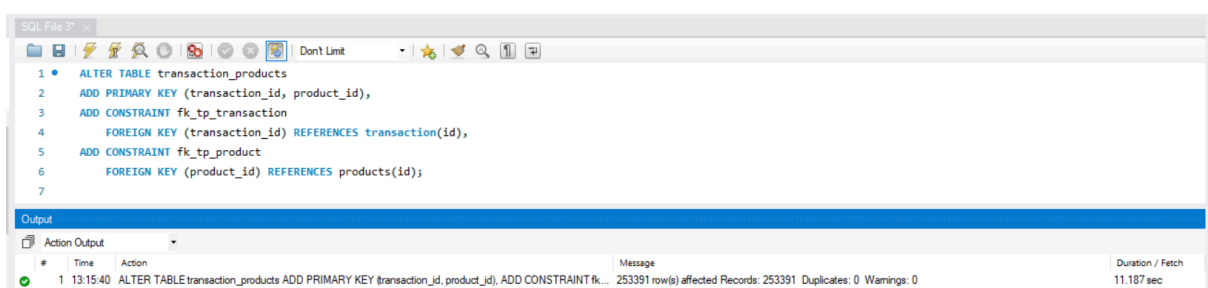
#	Time	Action	Message	Duration / Fetch
1	13:09:12	CREATE TABLE IF NOT EXISTS transaction_products AS SELECT t.id AS transaction_id, jt.product_id FROM...	253391 row(s) affected Records: 253391 Duplicates: 0 Warnings: 0	9.750 sec

2. Definimos una llave primaria compuesta (**transaction_id**, **product_id**), ya que ni **transaction_id** ni **product_id** por sí solos son únicos. Cada fila representa un producto dentro de una transacción.

Añadimos FK hacia las tablas originales:

- **transaction_id** → **transaction(id)**
- **product_id** → **products(id)**

Lo que asegura integridad referencial: cada fila de la tabla intermedia solo contenga transacciones y productos existentes.



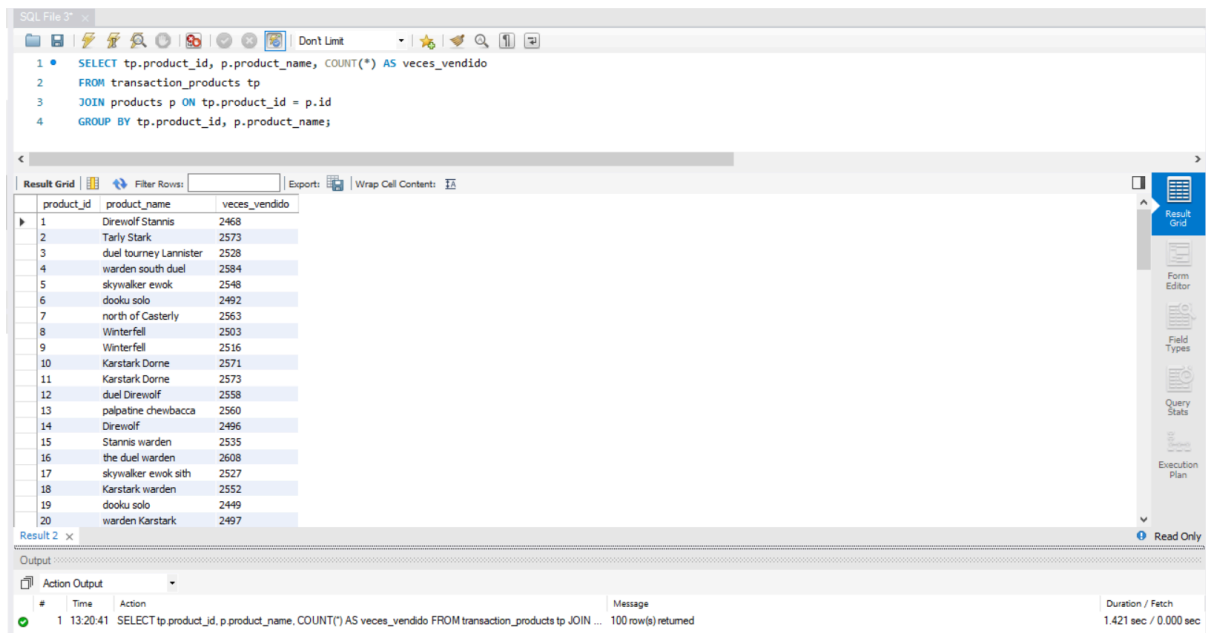
```
1 ALTER TABLE transaction_products
2 ADD PRIMARY KEY (transaction_id, product_id),
3 ADD CONSTRAINT fk_tp_transaction
4 FOREIGN KEY (transaction_id) REFERENCES transaction(id),
5 ADD CONSTRAINT fk_tp_product
6 FOREIGN KEY (product_id) REFERENCES products(id);
7
```

Output

#	Time	Action	Message	Duration / Fetch
1	13:15:40	ALTER TABLE transaction_products ADD PRIMARY KEY (transaction_id, product_id), ADD CONSTRAINT fk...	253391 row(s) affected Records: 253391 Duplicates: 0 Warnings: 0	11.187 sec

3. Consulta final

Utilizamos la tabla products junto con la tabla intermedia, agrupamos por product_id y contamos cuántas veces aparece cada uno para obtener el número de ventas por producto.



The screenshot shows a SQL query in a file named 'SQL File 3'. The query is as follows:

```
1 • SELECT tp.product_id, p.product_name, COUNT(*) AS veces_vendido
2 FROM transaction_products tp
3 JOIN products p ON tp.product_id = p.id
4 GROUP BY tp.product_id, p.product_name;
```

Below the query, the 'Result Grid' displays 20 rows of data. The columns are 'product_id', 'product_name', and 'veces_vendido'.

product_id	product_name	veces_vendido
1	Direwolf Stannis	2468
2	Tarly Stark	2573
3	duel tourney Lannister	2528
4	warden south duel	2584
5	skywalker evok	2548
6	dooku solo	2492
7	north of Casterly	2563
8	Winterfell	2503
9	Winterfell	2516
10	Karstark Dorne	2571
11	Karstark Dorne	2573
12	duel Direwolf	2558
13	palpatine chewbacca	2560
14	Direwolf	2496
15	Stannis warden	2535
16	the duel warden	2608
17	skywalker evok sith	2527
18	Karstark warden	2552
19	dooku solo	2449
20	warden Karstark	2497

At the bottom, the 'Output' section shows the execution details: 1 action at 13:20:41, returning 100 rows in 1.421 seconds.

La tabla de hechos central sigue siendo transaction.

transaction_products actúa como tabla intermedia para descomponer los productos por transacción.

