EXAMPLE USE OF AI TOOLS TO CREATE SQL STATEMENTS

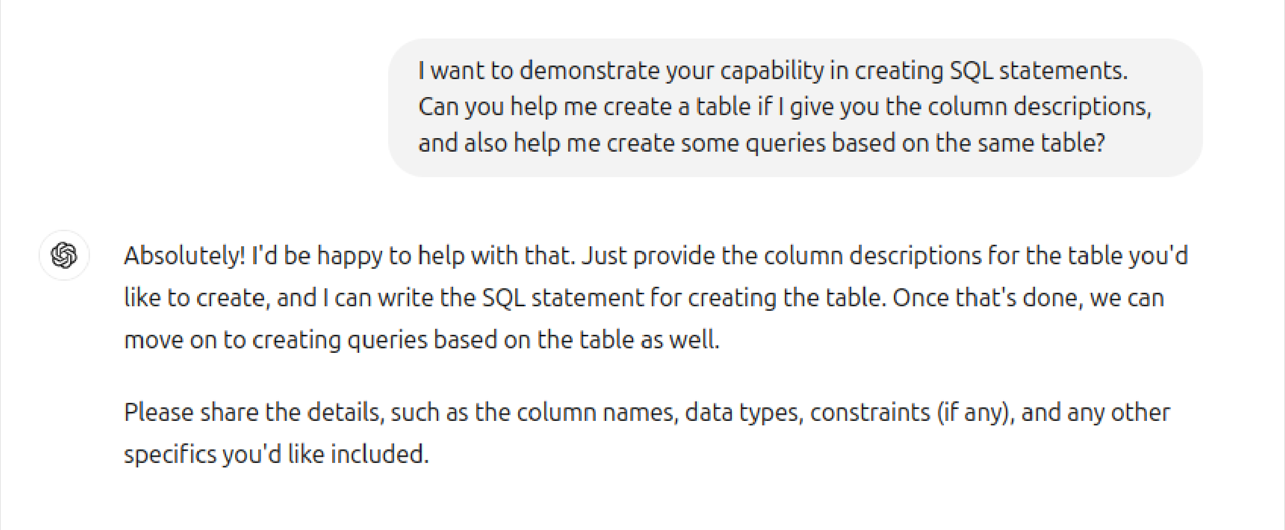
When using AI tools to generate SQL statements, you should be careful about a number of issues.

- Computers can not assume things easily. Even the best AI is very bad at assumptions. So when you want to create code-like output such as SQL, you should be very specific.
- Creating output step by step and asking for corrections is good practice.
- You should test the generated SQL as well.

The sample use will be about billing bus rides. The initial work will be simple, but AI will struggle with complex work.

TABLE CREATION

Always start by limiting the AI, so that unnecessary output is not created.



I want to demonstrate your capability in creating SQL statements. Can you help me create a table if I give you the column descriptions, and also help me create some queries based on the same table?

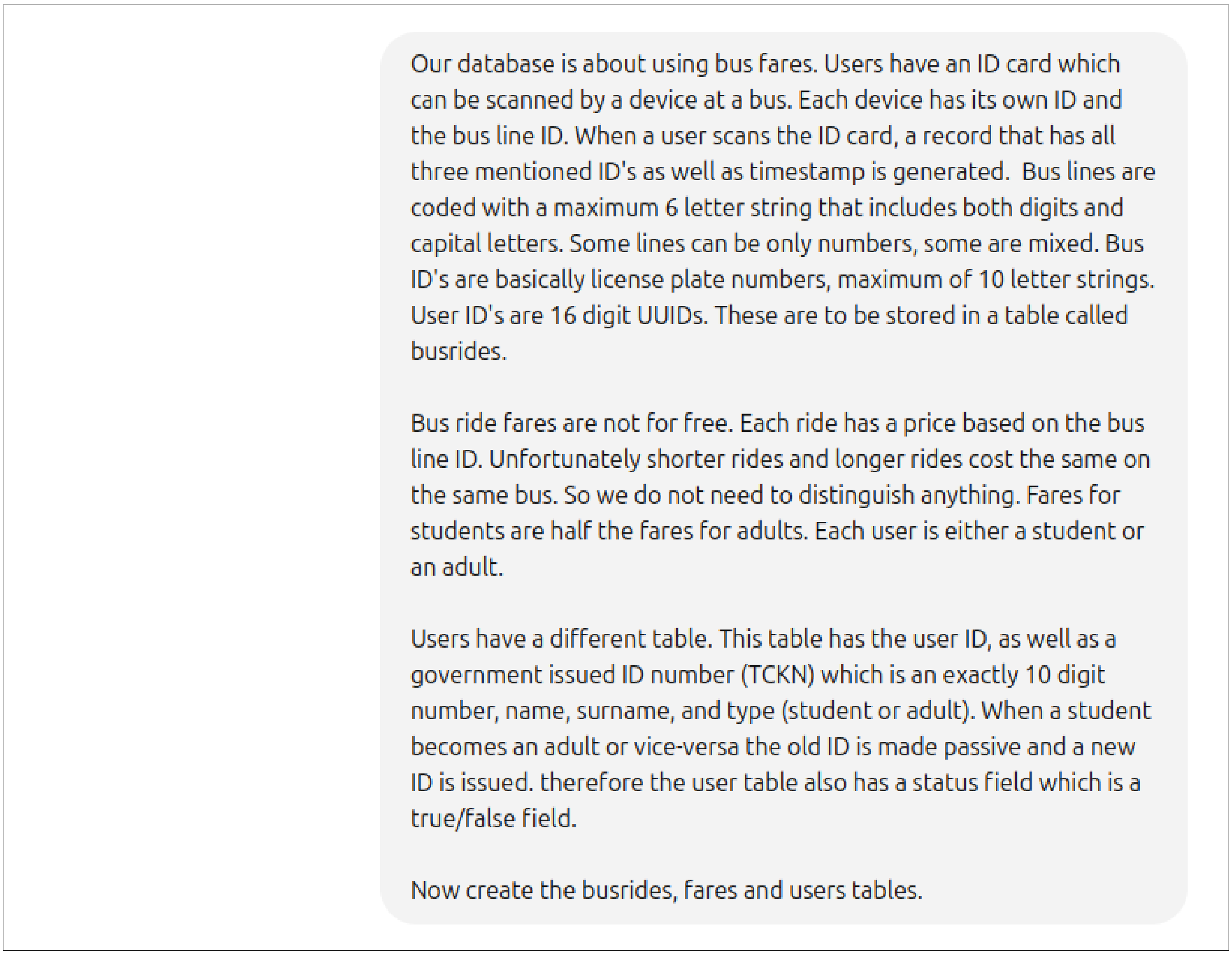
Absolutely! I'd be happy to help with that. Just provide the column descriptions for the table you'd like to create, and I can write the SQL statement for creating the table. Once that's done, we can move on to creating queries based on the table as well.

Please share the details, such as the column names, data types, constraints (if any), and any other specifics you'd like included.

Then state the need.

- Here we first create a context so that AI assumes less.
- When a data field is described, we try to explain its content and any formats.
- Simple fields such as names and surnames can be left for the AI.

Also note that at this stage we do not describe much about how data is created, only its structure.



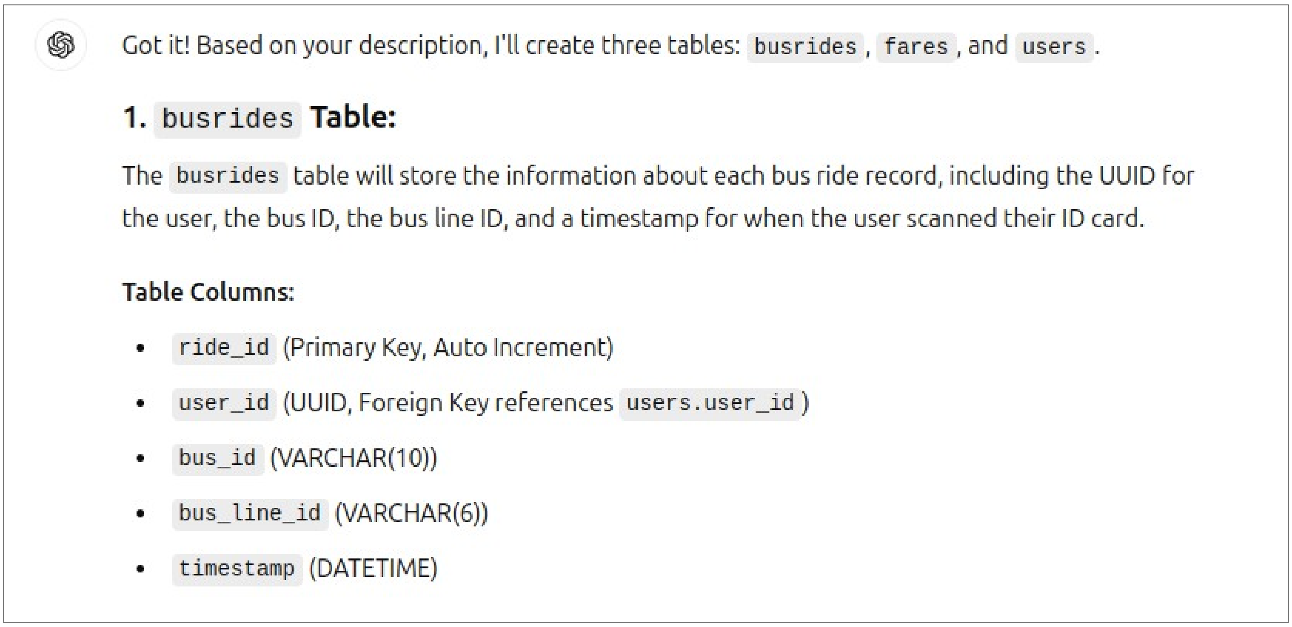
Our database is about using bus fares. Users have an ID card which can be scanned by a device at a bus. Each device has its own ID and the bus line ID. When a user scans the ID card, a record that has all three mentioned ID's as well as timestamp is generated. Bus lines are coded with a maximum 6 letter string that includes both digits and capital letters. Some lines can be only numbers, some are mixed. Bus ID's are basically license plate numbers, maximum of 10 letter strings. User ID's are 16 digit UUIDs. These are to be stored in a table called busrides.

Bus ride fares are not for free. Each ride has a price based on the bus line ID. Unfortunately shorter rides and longer rides cost the same on the same bus. So we do not need to distinguish anything. Fares for students are half the fares for adults. Each user is either a student or an adult.

Users have a different table. This table has the user ID, as well as a government issued ID number (TCKN) which is an exactly 10 digit number, name, surname, and type (student or adult). When a student becomes an adult or vice-versa the old ID is made passive and a new ID is issued. therefore the user table also has a status field which is a true/false field.

Now create the busrides, fares and users tables.

The output in stages:



Got it! Based on your description, I'll create three tables: `busrides`, `fares`, and `users`.

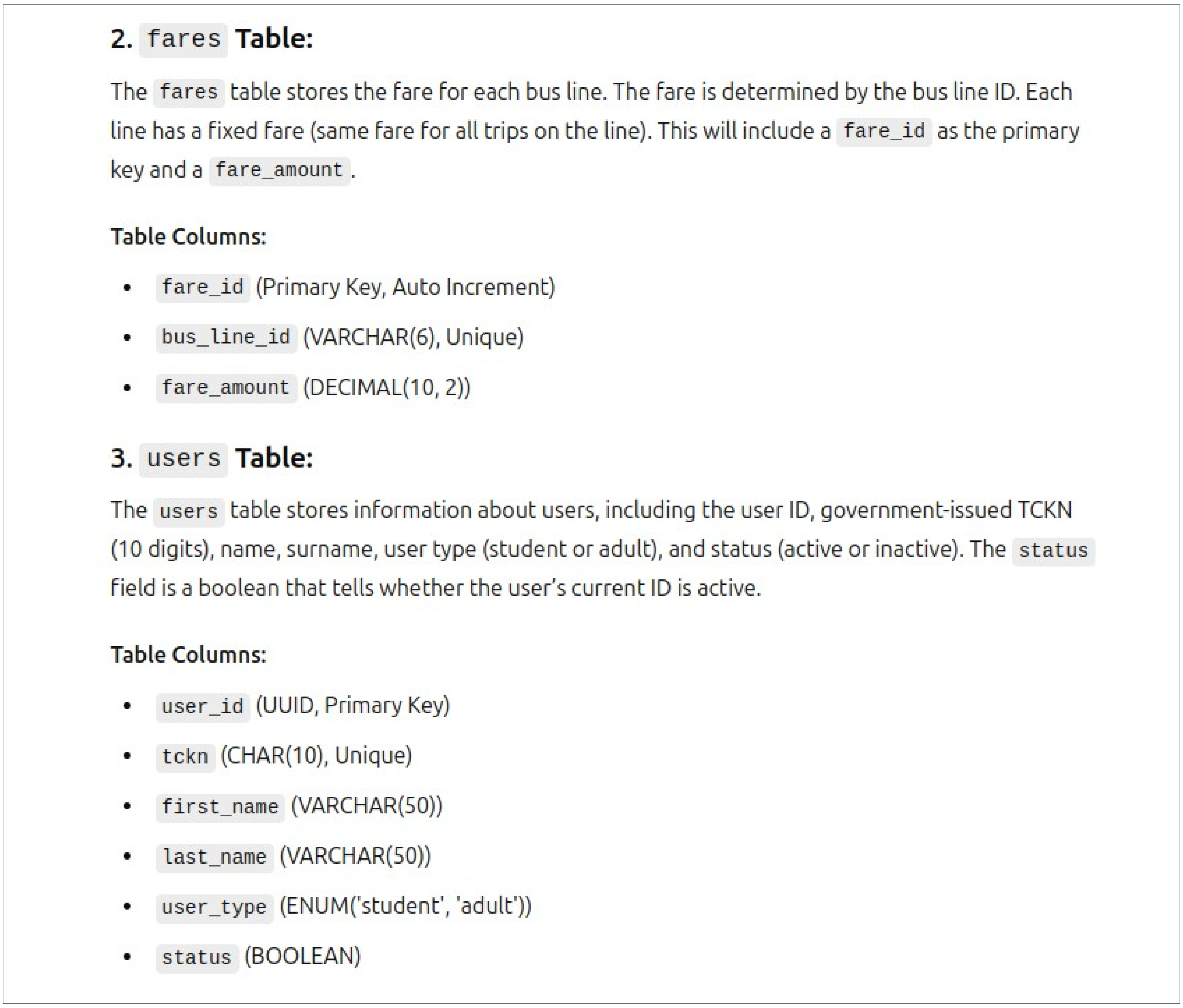
1. `busrides` Table:

The `busrides` table will store the information about each bus ride record, including the UUID for the user, the bus ID, the bus line ID, and a timestamp for when the user scanned their ID card.

Table Columns:

- `ride_id` (Primary Key, Auto Increment)
- `user_id` (UUID, Foreign Key references `users.user_id`)
- `bus_id` (VARCHAR(10))
- `bus_line_id` (VARCHAR(6))
- `timestamp` (DATETIME)

Note that AI can add an ID field to be used as primary key. The type selections are generic. No DBMS specific types are selected.



2. `fares` Table:

The `fares` table stores the fare for each bus line. The fare is determined by the bus line ID. Each line has a fixed fare (same fare for all trips on the line). This will include a `fare_id` as the primary key and a `fare_amount`.

Table Columns:

- `fare_id` (Primary Key, Auto Increment)
- `bus_line_id` (VARCHAR(6), Unique)
- `fare_amount` (DECIMAL(10, 2))

3. `users` Table:

The `users` table stores information about users, including the user ID, government-issued TCKN (10 digits), name, surname, user type (student or adult), and status (active or inactive). The `status` field is a boolean that tells whether the user's current ID is active.

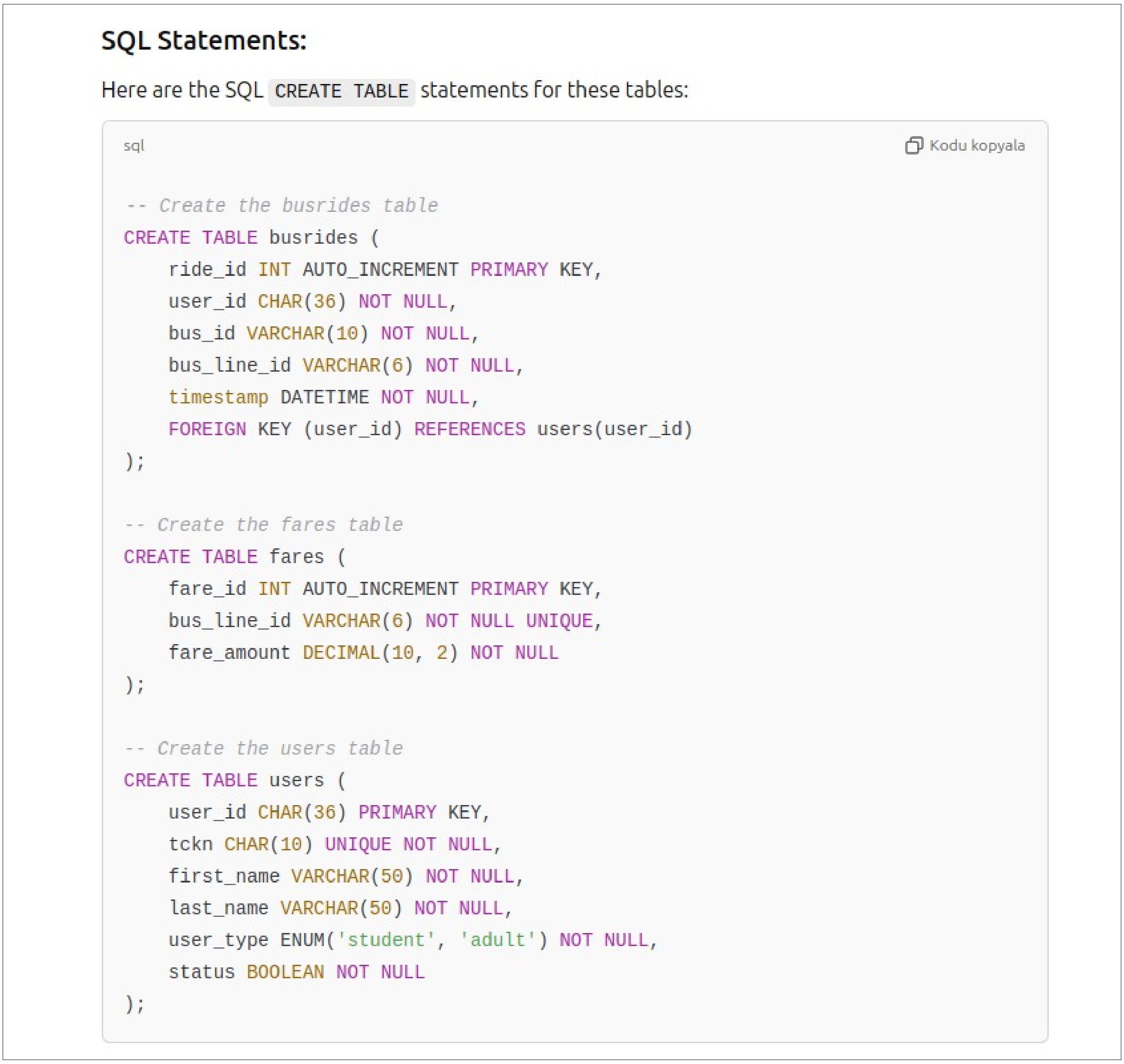
Table Columns:

- `user_id` (UUID, Primary Key)
- `tckn` (CHAR(10), Unique)
- `first_name` (VARCHAR(50))
- `last_name` (VARCHAR(50))
- `user_type` (ENUM('student', 'adult'))
- `status` (BOOLEAN)

The SQL statements to create these tables are also generic and these we need to review.

- Check the primary and foreign key decisions.
- NULL and NOT NULL selections

If you want to change things, state the AI to change them so that it knows about the change. This is important because later you will ask the same AI to create queries.



SQL Statements:

Here are the SQL `CREATE TABLE` statements for these tables:

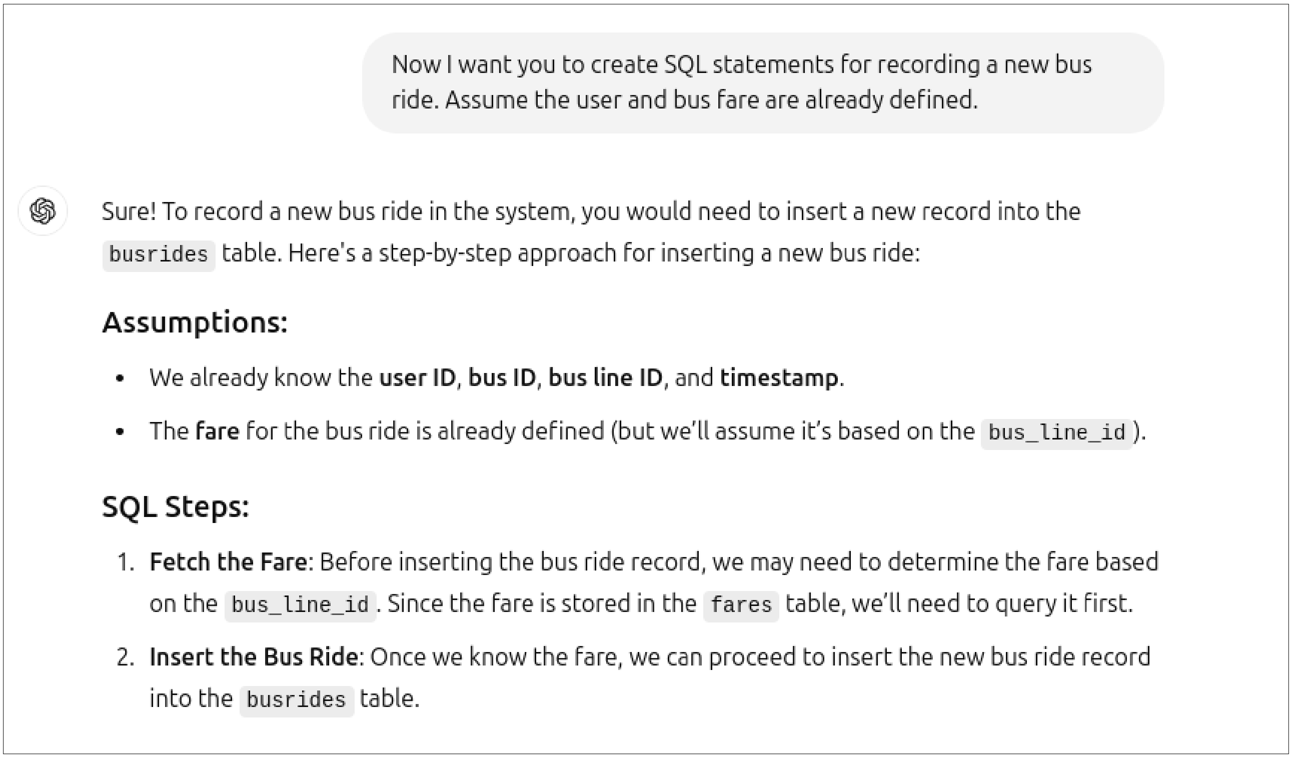
```sql
-- Create the busrides table
CREATE TABLE busrides (
    ride_id INT AUTO_INCREMENT PRIMARY KEY,
    user_id CHAR(36) NOT NULL,
    bus_id VARCHAR(10) NOT NULL,
    bus_line_id VARCHAR(6) NOT NULL,
    timestamp DATETIME NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);

-- Create the fares table
CREATE TABLE fares (
    fare_id INT AUTO_INCREMENT PRIMARY KEY,
    bus_line_id VARCHAR(6) NOT NULL UNIQUE,
    fare_amount DECIMAL(10, 2) NOT NULL
);

-- Create the users table
CREATE TABLE users (
    user_id CHAR(36) PRIMARY KEY,
    tckn CHAR(10) UNIQUE NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    user_type ENUM('student', 'adult') NOT NULL,
    status BOOLEAN NOT NULL
);
```

SQL STATEMENTS

When you create a record. There will be related records and AI may make assumptions on the bus ride and the way your application works. See now



Now I want you to create SQL statements for recording a new bus ride. Assume the user and bus fare are already defined.

Sure! To record a new bus ride in the system, you would need to insert a new record into the `busrides` table. Here's a step-by-step approach for inserting a new bus ride:

Assumptions:

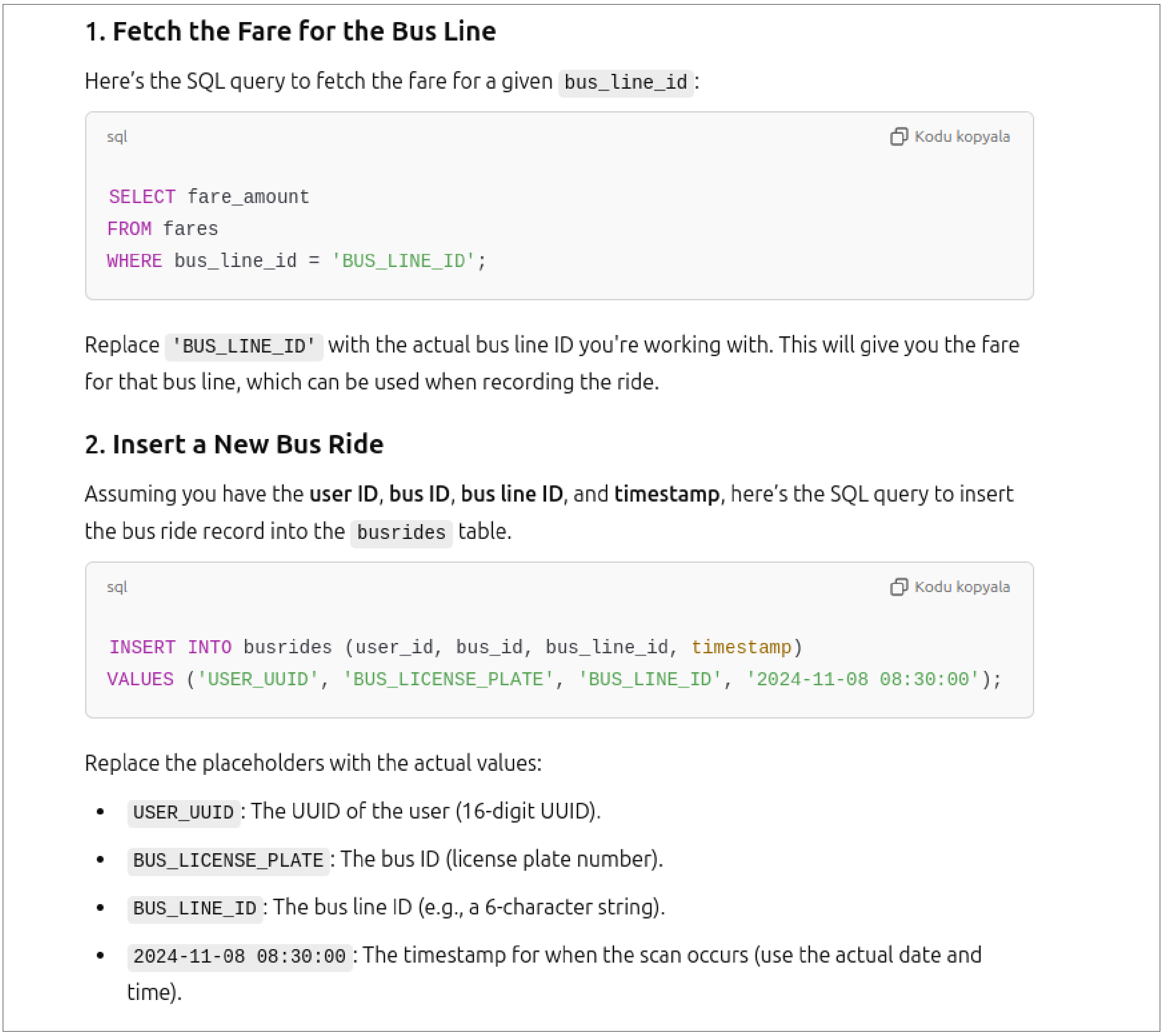
- We already know the **user ID**, **bus ID**, **bus line ID**, and **timestamp**.
- The **fare** for the bus ride is already defined (but we'll assume it's based on the `bus_line_id`).

SQL Steps:

1. **Fetch the Fare**: Before inserting the bus ride record, we may need to determine the fare based on the `bus_line_id`. Since the fare is stored in the `fares` table, we'll need to query it first.
2. **Insert the Bus Ride**: Once we know the fare, we can proceed to insert the new bus ride record into the `busrides` table.

And the SQL statements are reflecting those assumptions. You should validate these. It may also help you understand your case better.

- If the bus ride is recorded by a device that know the bus ride number, does it also know the fare as well?
- When exactly do you execure these statements?
- How often do you execute these statements?
- Is the fare stored in memory so that we do not execute every SQL statement?



1. Fetch the Fare for the Bus Line

Here's the SQL query to fetch the fare for a given `bus_line_id`:

```sql
SELECT fare_amount
FROM fares
WHERE bus_line_id = 'BUS_LINE_ID';
```

Replace `'BUS_LINE_ID'` with the actual bus line ID you're working with. This will give you the fare for that bus line, which can be used when recording the ride.

2. Insert a New Bus Ride

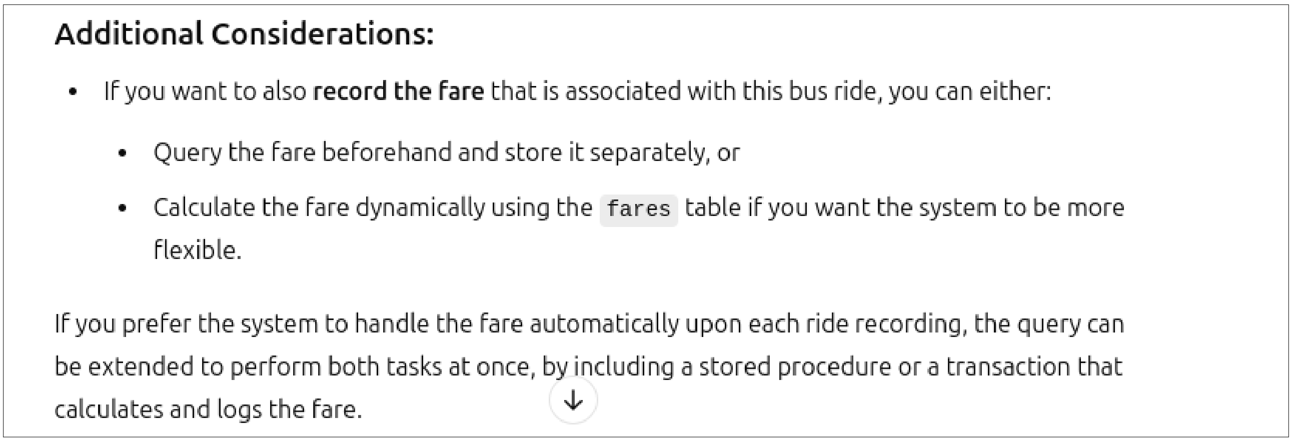
Assuming you have the **user ID**, **bus ID**, **bus line ID**, and **timestamp**, here's the SQL query to insert the bus ride record into the `busrides` table.

```sql
INSERT INTO busrides (user_id, bus_id, bus_line_id, timestamp)
VALUES ('USER_UUID', 'BUS_LICENSE_PLATE', 'BUS_LINE_ID', '2024-11-08 08:30:00');
```

Replace the placeholders with the actual values:

- `USER_UUID`: The UUID of the user (16-digit UUID).
- `BUS_LICENSE_PLATE`: The bus ID (license plate number).
- `BUS_LINE_ID`: The bus line ID (e.g., a 6-character string).
- `2024-11-08 08:30:00`: The timestamp for when the scan occurs (use the actual date and time).

Also always read and evaluate any comments made by theAI.

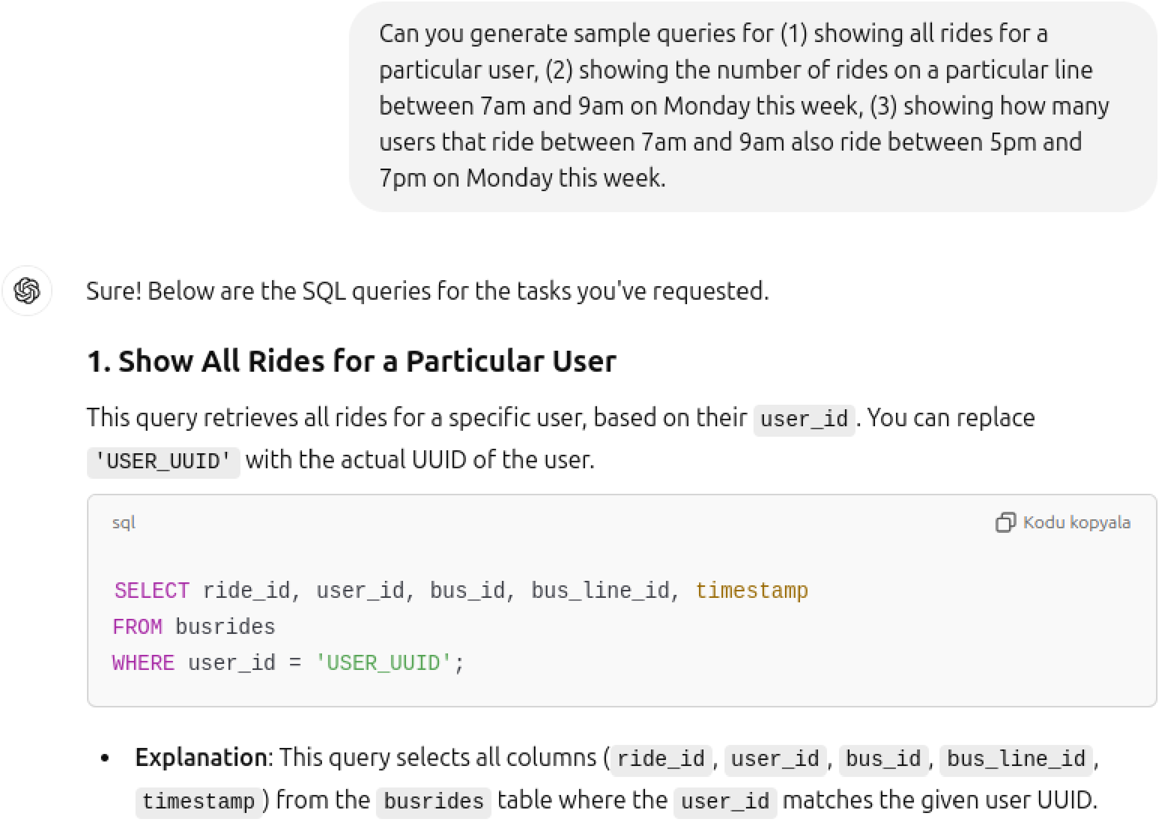


Additional Considerations:

- If you want to also **record the fare** that is associated with this bus ride, you can either:
  - Query the fare beforehand and store it separately, or
  - Calculate the fare dynamically using the `fares` table if you want the system to be more flexible.

If you prefer the system to handle the fare automatically upon each ride recording, the query can be extended to perform both tasks at once, by including a stored procedure or a transaction that calculates and logs the fare.

SIMPLE CASES



Can you generate sample queries for (1) showing all rides for a particular user, (2) showing the number of rides on a particular line between 7am and 9am on Monday this week, (3) showing how many users that ride between 7am and 9am also ride between 5pm and 7pm on Monday this week.

Sure! Below are the SQL queries for the tasks you've requested.

1. Show All Rides for a Particular User

This query retrieves all rides for a specific user, based on their `user_id`. You can replace `'USER_UUID'` with the actual UUID of the user.

```sql
SELECT ride_id, user_id, bus_id, bus_line_id, timestamp
FROM busrides
WHERE user_id = 'USER_UUID';
```

- **Explanation**: This query selects all columns (`ride_id`, `user_id`, `bus_id`, `bus_line_id`, `timestamp`) from the `busrides` table where the `user_id` matches the given user UUID.

2. Show the Number of Rides on a Particular Bus Line Between 7am and 9am on Monday This Week

This query counts how many rides occurred on a particular bus line between 7:00 AM and 9:00 AM on the most recent Monday. We will use the `TIMESTAMPDIFF` function to filter based on the time and date.

First, we need to get the date for the Monday of the current week, which you can do using SQL's date functions.

sql Kodu kopyala

```sql
SELECT COUNT(*) AS number_of_rides
FROM busrides
WHERE bus_line_id = 'BUS_LINE_ID'
  AND timestamp >= TIMESTAMP(DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT_DAT
  AND timestamp <= TIMESTAMP(DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT_DAT
```

- **Explanation**:
  - `CURRENT_DATE` gives today's date.
  - `DAYOFWEEK(CURRENT_DATE)` returns the current day of the week (1 for Sunday, 2 for Monday, etc.).
  - `DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT_DATE) - 2) DAY)` calculates the date for the most recent Monday.
  - `TIMESTAMP(DATE_SUB(...), '07:00:00')` and `TIMESTAMP(..., '09:00:00')` set the time range for 7:00 AM to 9:00 AM on that Monday.
  - The `WHERE` clause ensures we only count rides on the specific bus line (`bus_line_id = 'BUS_LINE_ID'`) within that time range.

You can replace `'BUS_LINE_ID'` with the actual bus line ID you're interested in.

3. Show How Many Users Who Ride Between 7am and 9am Also Ride Between 5pm and 7pm on Monday This Week

This query will first identify users who have rides between 7:00 AM and 9:00 AM on Monday and then check if those same users also have rides between 5:00 PM and 7:00 PM on the same day.

We can achieve this by using a subquery to find users who meet the first time range condition and then join it with the same table to check for the second time range.

sql Kodu kopyala

```sql
SELECT COUNT(DISTINCT br1.user_id) AS number_of_users
FROM busrides br1
JOIN busrides br2 ON br1.user_id = br2.user_id
WHERE br1.timestamp >= TIMESTAMP(DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT
  AND br1.timestamp <= TIMESTAMP(DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT
  AND br2.timestamp >= TIMESTAMP(DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT
  AND br2.timestamp <= TIMESTAMP(DATE_SUB(CURRENT_DATE, INTERVAL (DAYOFWEEK(CURRENT
```

- **Explanation**:
  - We join the `busrides` table to itself (`br1` and `br2`) based on matching `user_id`. This allows us to compare two different time ranges for the same user.
  - The first `WHERE` condition checks for rides between 7:00 AM and 9:00 AM (`br1.timestamp`).
  - The second `WHERE` condition checks for rides between 5:00 PM and 7:00 PM (`br2.timestamp`).
  - The `COUNT(DISTINCT br1.user_id)` ensures that only distinct users are counted, preventing duplicates if the same user has multiple rides within the specified time ranges.

- **Explanation**:
  - We join the `busrides` table to itself (`br1` and `br2`) based on matching `user_id`. This allows us to compare two different time ranges for the same user.
  - The first `WHERE` condition checks for rides between 7:00 AM and 9:00 AM (`br1.timestamp`).
  - The second `WHERE` condition checks for rides between 5:00 PM and 7:00 PM (`br2.timestamp`).
  - The `COUNT(DISTINCT br1.user_id)` ensures that only distinct users are counted, preventing duplicates if the same user has multiple rides within the specified time ranges.

COMPLEX CASE