

Name: Stefan Kasperzack

Table of Contents

Project Direction Overview	2
Use Cases and Fields	3
Structural Database Rules	8
Conceptual Entity-Relationship Diagram	11
Full DBMS Physical ERD	12
Stored Procedure Execution and Explanations	16
Question Identification and Explanations	20
Query Executions and Explanations	21
Index Identification and Creations	23
History Table Demonstration	26
Data Visualizations	27
Summary and Reflection	29

Project Direction Overview

I would like to develop a mobile app named ScooterRent, through which a startup rents electric scooters to individuals in three branches in Stuttgart, Germany. To rent a scooter, the individual must download and install the ScooterRent mobile app on her smartphone and create a user account in ScooterRent. Note, the individual must be at least 14 years old, because according to the law, these rented scooters can only be driven by people who have reached the age of 14. To rent a scooter, the registered user must first log in and enter her credit card information. Then the desired branch and scooter must be selected, and the rental start date and time specified. When the user confirms that the selected scooter should be rented, ScooterRent displays a confirmation and the user can pick up the scooter from the branch. To return the scooter, the user needs to drive the scooter to one of the three branches, park it, switch it off, and connect it to the charging station. The charging station then automatically sends a signal to the backend of the ScooterRent app that the rented scooter has been returned. Whereupon the rental duration is calculated. The duration in hours is multiplied by the hourly price for the rented scooter and invoiced to the user.

Since this course is a course on database design and implementation, I will only deal with the database part of the ScooterRent app. The database must contain the following:

- Branches (table Branch): branch ID, street name, house number, ZIP code, city, telephone number, email address. Note: the database will not contain any field for state; the ZIP code uniquely identifies each part of Germany independent of the state.
- Scooters (table Scooter): scooter ID, scooter name, scooter vendor, maximum range, maximum speed, year of construction, hourly rental price, branch where scooter is located (if not rented out), scooter status (operational or nonfunctional).
- Person (table Person (as supertype)): person ID, first name, last name, date of birth, contact information (street name, house number, ZIP code, city, telephone number, email address). IsSubtype1, IsSubtype2, ... as subtype discriminators, so that the subtype tables of Person do not have to be joined first to know which overlapping subtypes Person is.
- Users (table User (as subtype)): user ID (as a subtype of person, user ID should be called person ID), password, registration date; (and sets contact information in supertype Person).
- Employees (table Employee (as subtype)): employee ID (as a subtype of person, employee ID should be called person ID), name of the position (e.g., accountant, sales manager, etc.), hire date; (and sets contact information in supertype Person).
- Rental agreements (table RentalAgreement): rental agreement ID, rental start date and time, rental end date and time, (and the rental agreement will also contain the rented scooter ID (thus the hourly rental price), branch ID, and the renter's user ID (PersonID)).
- Invoices (table Invoice): invoice ID, invoice date, total amount scooter rental (calculated field), date on which user paid the invoice (and the invoice should also contain the rented scooter ID, branch ID, and the renter's user ID (PersonID)). An invoice could be linked to the corresponding rental agreement that contains this information).
- Credit cards (table CreditCard): credit card ID, card number (note: the name of the credit card provider is encoded in the credit card number), cardholder first name, cardholder last name, card expiry date.
- Advertisements (table Advertisement): advertisement ID, title, message, start date, end date, budget.
- AdvertisingAgencies (table AdvertisingAgency): advertising agency ID, advertising agency name, telephone number, email address.

- RentalPriceChanges (history table RentalPriceChange): rental price change ID, old hourly rate (old rental price), new hourly rate (old hourly rate is changed to this hourly rate), change date, (and the rental price change will also contain the rented scooter ID – so that it is clear for which scooter the hourly rate has changed).

Furthermore, it can be assumed that bridging tables are necessary to solve many-to-many relationships and foreign keys to enforce relationship constraints.

Many years ago, colleagues and I had the idea to create just such an app. However, we did not do it. I think this course offers a good opportunity to learn how the database for the ScooterRent app could be designed and implemented to avoid redundant data, inconsistent data and an incomprehensibility database design.

Use Cases and Fields

- I. Individual installs the ScooterRent mobile app on her smartphone via the app store and creates a user account (registers).
 1. User downloads the ScooterRent app from the app store and installs ScooterRent on her smartphone.
 2. ScooterRent shows user two options: "Create account" and "Log in".
 3. User clicks on "Create account".
 4. ScooterRent asks user to set a password.
 5. User sets a password.
 6. ScooterRent asks user for her date of birth.
 7. User enters date of birth.
 8. ScooterRent: If the user is younger than 14 years, the account creation will be canceled and the user will be informed that ScooterRent can only be used from the age of 14.
Else if the user is at least 14 years old, the user is asked to enter her personal data.
 9. User enters personal data.
 10. ScooterRent creates a user in the database and stores the values entered by the user in the database.

Field	What it Stores	Why it is Needed
UserID (PersonID)	A unique user ID.	To uniquely identify users. There is theoretically the possibility that users have the same name and place of residence and the ID ensures that each user can be uniquely identified.
EncryptedPassword	Encrypted password.	A password that allows users to log into their accounts with user ID and password.
FirstName	First name of user.	To address users by their first name (for example when sending an e-mail).

LastName	Last name of user.	To address users by their last name (for example when sending an e-mail).
StreetName	Street name of street where user lives.	To be able to contact users by post (for example, to deliver advertising material; or contract documents by post, if this should become necessary).
HouseNumber	House number of building where user lives.	
ZipCode	ZIP code of area where user lives.	
City	City where user lives.	
DateOfBirth	Date of birth of user.	To determine whether users are at least 14 years old; younger users are not allowed to use ScooterRent / are not allowed to rent and drive the scooters.
TelephoneNumber	Phone number of user.	To contact users by phone.
EmailAddress	E-mail address of user.	To contact users by e-mail.
RegistrationDate	Registration date of user in ScooterRent mobile app.	Helps sales managers to give appropriate loyalty discounts based on age of the account.

- II. Registered user adds credit card information (assumptions: user has created an account in ScooterRent and is logged in).
1. User clicks on “Add credit card information” in ScooterRent.
 2. ScooterRent displays the page for entering the credit card information.
 3. User enters the credit card information.
 4. ScooterRent checks the credit card information. If entries are incorrect, user is asked again to provide credit card information.
 5. ScooterRent stores the user's credit card information in the database.

Field	What it Stores	Why it is Needed
CreditCardID	A unique credit card ID.	To uniquely identify a credit card that can be used for payment.
CardNumber	Credit card number.	To charge credit card of user (card number, cardholder name, card expiry date, and card security number are used to uniquely identify credit card accounts (however, according to the law, the security code must not be stored in databases by companies)).
CardholderFirstName	First name of cardholder.	
CardholderLastName	Last name of cardholder.	

ExpiryDate	Expiration date of credit card.	To charge credit card of user. And to determine that the credit card is still valid.
------------	---------------------------------	---

III. The user rents a scooter from one of the branches (assumptions: user has created an account in ScooterRent, is logged in, and has entered valid credit card information).

1. User selects a branch in ScooterRent.
2. ScooterRent asks for the rental start date and time.
3. User enters the rental start date and time (no rental end time is needed; users can rent scooters for as long as they want).
4. ScooterRent shows the available scooters with a description of the scooters (scooter name, maximum range, maximum speed) and rental prices per hour.
5. User selects a scooter.
6. ScooterRent creates the rental agreement and displays it to the user.
7. User confirms rental agreement.
8. User visits the branch and picks up the scooter.

Field	What it Stores	Why it is Needed
BranchID	A unique branch ID.	To uniquely identify branches. In this use case: to not show the user the same branch for selection several times. And to know branch where scooter is located (if not rented out).
StreetName	Street name of branch.	So that user knows where the branch is located (to pick up the rented scooter and return it).
HouseNumber	House number of branch.	
ZipCode	ZIP code of area where branch is located.	
City	City where branch is located.	
TelephoneNumber	Phone number of branch.	In case user has any questions and would like to contact an employee by phone.
EmailAddress	E-mail address of branch.	In case user has any questions and would like to contact an employee by email.
ScooterID	A unique scooter ID.	To uniquely identify the scooters that can be rented.
ScooterName	Name of scooter.	So that the user knows what kind of scooter it is (e.g., the E-Scooter GTX-135).
MaximumSpeed	Top speed of scooter.	Different users have different needs: some want a fast scooter (lower range), others a slow scooter
MaximumRange	Maximum range of scooter.	

		(higher range); top speed and maximum range informs users about the speed and range.
HourlyRate	Scooter's hourly rental rate.	So that user can be informed about hourly rental rate (price) to rent a scooter. And to calculate total price of rental (for this, rental duration must be known).
RentalAgreementID	A unique rental agreement ID.	To create and uniquely identify a rental agreement.
StartDateTime	Start date and time of scooter rental.	To calculate rental duration of scooter (when EndDateTime is known).
UserID (PersonID)	A unique user ID.	In this case, to connect the rental agreement with the user the rental agreement belongs to.

- IV. The user returns the rented scooter to a branch (assumption: the user has rented a scooter).
1. User parks the rented scooter at a branch, switches the scooter off, and connects the scooter to a charger.
 2. ScooterRent changes the rental agreement from "not ended" to "ended" and informs the user that the rented scooter has been successfully returned and the rental agreement is thus ended.
 3. User confirms that the rental agreement has ended.
 4. ScooterRent creates an invoice for the ended rental agreement and displays the invoice to the user.

Field	What it Stores	Why it is Needed
EndDateTime	End date and time of scooter rental.	To calculate rental duration of scooter (with StartDateTime).
RentalAgreementID	A unique rental agreement ID.	To close/end and uniquely identify a rental agreement.
BranchID	A unique branch ID.	To know which branch the scooter has been returned to.
UserID (PersonID)	A unique user ID.	To know which user returned the scooter.
ScooterID	A unique scooter ID.	To know which scooter was returned.
HourlyRate	Scooter's hourly rental rate.	To calculate total price of rental (hourly rate * rental duration in hours).

InvoiceID	A unique invoice ID.	To uniquely identify invoices.
InvoiceDate	Date on which invoice was created.	To know when the invoice was created (the exact date must be known for accounting).
PaymentDate	Date when user has paid the invoice amount.	So that the startup knows when the user has paid the invoice (no recorded date means that the user has not yet paid the invoice).

Note: There is a calculated field that is not stored in the database: InvoiceTotalAmount (total rental amount to be paid by the user – to invoice user total rental amount).

V. An employee of a branch opens the employee application to the database to commission an advertisement.

1. Employee clicks on "Commission advertisement".
2. Employee adds advertisement title.
3. Employee adds advertisement message.
4. Employee sets advertisement start date.
5. Employee sets advertisement end date.
6. Employee sets advertisement budget
7. Employee selects advertising agency.

Field	What it Stores	Why it is Needed
EmployeeID (PersonID)	A unique employee ID	To uniquely identify employees.
AdvertisementID	A unique advertisement ID.	To uniquely identify advertisements.
Title	Title of the advertisement.	This is the title of the advertising message so that the recipient of the advertising message knows immediately what it is about.
Message	Message of advertisement.	More detailed advertising text to convince recipients.
StartDate	Start date of advertisement.	The start and end dates are important to see what impact the advertisement had on marketing KPIs during that period.
EndDate	End date of advertisement.	
Budget	Budget for advertisement.	To know how much money is available for this advertisement and to calculate profitability ratios

		in combination with other information.
AdvertisingAgencyName	Name of advertising agency.	Name of advertising agency to advertise through them.
AdvertisingAgencyID	A unique advertising agency ID.	To uniquely identify advertising agency.
TelephoneNumber	Phone number of advertising agency.	To tell advertising agency advertising orders by phone.
EmailAddress	E-mail address of advertising agency.	To send advertising agency advertising orders by email.

- VI. The price changes (hourly rates) of the scooter rentals are recorded in a history table so that ScooterRent can track how the rental prices have changed over time.

Field	What it Stores	Why it is Needed
RentalPriceChangeID	A unique rental price change ID.	To uniquely identify the rental price change.
ScooterID	A unique scooter ID.	This is a foreign key to the Scooter table, a reference to the scooter that had the change in the price (hourly rental rate).
OldHourlyRate	The old hourly rate.	To record the old and new hourly rate respectively so that the price change can be tracked and calculated.
NewHourlyRate	The new hourly rate.	
ChangeDate	The change date.	To record the change date when the old hourly rate was changed to the new hourly rate.

Structural Database Rules

- **For use case (I):**

- Each user has one user account; each user account has one user.

Explanation:

Each user (entity 1) has (relationship) one (plurality constraint) user account (entity 2); each user account (entity 2) has (relationship) one (plurality constraint) user (entity 1).

Each user must have a ScooterRent account to rent scooters. Likewise, each account must have a user. But why not 'each user account has many users'? This is theoretically possible, for example via a corporate account that has multiple users with different roles; but in the project directions it was defined that the startup only rents to individuals (and not to companies), and each individual user can only have one account.

And since the user and user account can be viewed as one and the same entity from the perspective of the startup database, I will merge the entities user and user account into one entity user (see entity relationship diagram).

- **For use case (II):**

- Each user may have many credit cards. Each credit card applies to one user.

Explanation:

Each user (entity 1) may (participation constraint) have (relationship) many (plurality constraint) credit cards (entity 2). Each credit card (entity 2) applies (relationship) to one (plurality constraint) user (entity 1). Note: In the following, entity, participation constraint, relationship, and plurality constraint are not labeled again because it is always the same principle. Each ScooterRent user can be a customer with many credit card providers and can therefore add many credit cards in ScooterRent, but the user does not have to add any, she may add them. And the same credit card applies to one user; for the sake of simplification, it is assumed that no more than one user uses the same credit card, which would be theoretically possible. and it is also possible that a payment provider is not added by any user.

- **For use case (III):**

- Each user may enter many rental agreements; each rental agreement is entered by one user.

Explanation:

Each user can enter into many rental agreements, for example scooter 1 is rented tomorrow (rental agreement 1) and the day after scooter 2 (rental agreement 2); but a user does not have to rent scooters, she may. And since the startup's rental agreements should not lead to complex legal cases and disputes, the rental agreements are limited to two parties: one branch and one user. However, it is imperative that both parties are present to enter into a rental agreement; thus, each rental agreement is entered by one user.

- Each rental agreement is made with one branch; each branch may make many rental agreements.

Explanation:

Each branch is responsible for its own rental agreements, so each rental agreement is made with one branch; at least one branch must be part of the rental agreement for a rental agreement to be made. And a branch can make several rental agreements, but it can also be the case that things are going badly economically and no rental agreements are made.

- Each user may rent many scooters; each scooter may be rented by many users.

Explanation:

The users cannot ride several scooters at the same time, but they can still rent (enter rental agreements that include) one or many scooters. And a scooter may not be rented at all or it is rented out many times (is included in many rental agreements).

Since the structural database rule describes a many-to-many relationship, a bridge table with the entity name RentalAgreement will be used in the conceptual (enhanced) entity-relationship diagram to resolve the many-to-many relationship in one-to-many relationships.

- Each scooter belongs to one branch; each branch may have many scooters.

Explanation:

Each scooter belongs to one branch (is part of the assets of one branch from an accounting perspective), and each branch can have several scooters on site – or if business is bad, none.

- **For use case (IV):**

- Each user may end many rental agreements; each rental agreement is ended by one user.
- Each rental agreement is ended with one branch; each branch may end many rental agreements.

- Each user may rent many scooters; each scooter may be rented by many users.

Explanation:

The three structural database rules above are exactly the same as in use case (III), only that the rental agreement is made / entered in use case (III) and the rental agreement ends in use case (IV). See the explanations for use case (III).

- Each rental agreement results in one invoice; each invoice resulted from one rental agreement.

Explanation:

When the rental agreement is completed (has ended), an invoice will be created, so that the user pays for the rental. And an invoice refers exactly to one rental agreement. A one-to-one relationship was chosen to avoid complex accounting issues that could arise when rental agreements are split across multiple invoices.

- **For use case (V):**

- Each employee works for one branch; each branch has one or many employees.

Explanation:

To have a clear separation between the branches, each employee only works for one branch. And 'each employee may work for one branch' is not possible, because then it would not be an employee. The same applies to a branch, every branch must have at least one (or many) employee(s), because otherwise it is not a branch, but an empty building.

- Each branch may commission many advertising agencies; each advertising agency may be commissioned by many branches.

Explanation:

Each branch is responsible for its own advertisement and therefore every branch commissions an advertising agency (which creates advertisement according to the wishes of the branch). However, it can be that a branch does not commission any advertising agency at all if there is no marketing budget, or many advertising agencies if there is enough marketing budget. Since the structural database rule describes a many-to-many relationship, a bridge table with the entity name Advertisement will be used in the conceptual (enhanced) entity-relationship diagram to resolve the many-to-many relationship in one-to-many relationships.

- **For use case (I and V):**

- A person is a user, an employee, several of these, or none of these.

Explanation:

A person (supertype) is a user (subtype), an employee (subtype), several of these (disjointness constraint), or none of these (completeness constraint).

This structural database rule concerning specialization-generalization consists of the supertype person and two overlapping (disjointness constraint) subtypes user and employee; overlapping because a person can be an employee who is also a user. Since not every subtype of person has been enumerated, it is a partially complete enumeration (completeness constraint).

- **History table:**

- Each scooter may have many rental price changes; each rental price change is for one scooter.

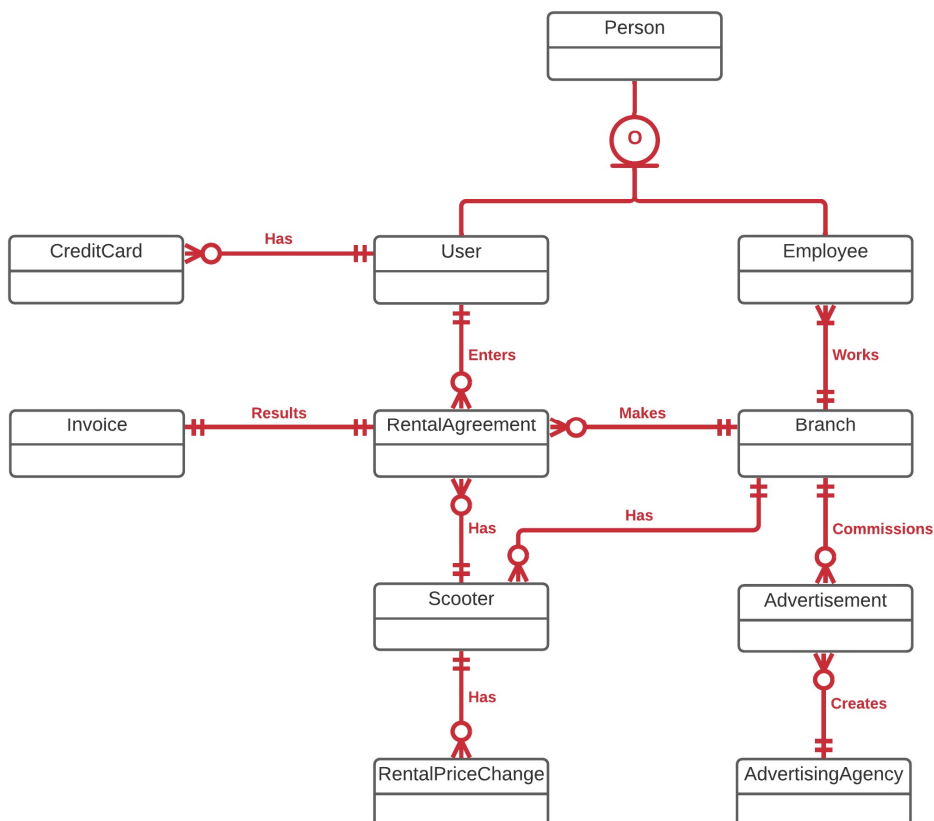
Explanation:

Each scooter can have a constant, unchanged rental price, but the rental price may also change. For example, the scooter and employee costs increase and thus the scooter rental prices have to be increased so that ScooterRent continues to be profitable. And every scooter has a rental price, so every rental price change is for a scooter.

Conceptual Entity-Relationship Diagram

The following diagram contains specialization-generalization, making it an enhanced entity-relationship diagram. A supertype (generalization) person and two subtypes (specialization) user and employee were identified. The constraints for the specialization-generalization relationships are overlapping and partially complete. Using specialization-generalization allows the attributes that user and employee have in common, such as address, phone number, and e-mail address, to be defined in the supertype person without having to include the attributes in the user and employee tables respectively.

No further specialization-generalization were added to the enhanced entity-relationship diagram. But if the project directions and use cases were different, then further specialization-generalization could be added to the enhanced entity-relationship diagram. For example, user could not only be a subtype, but also a supertype with subtypes VIP user and simple user; employee could be a supertype with salary employee and hourly employee, branch could be a supertype with subtypes premium branch and simple branch, etc. However, according to the project directions and uses cases, there are no VIP and simple users, only (simple) users; only salary employees are employed; and the branches do not differ. Further, the enhanced entity-relationship diagram contains two bridge tables: RentalAgreement and Advertisement. The bridge tables are necessary to resolve many-to-many relationships into one-to-many relationships.



Full DBMS Physical ERD

Table	Attribute	Datatype	Reasoning
Person	FirstName	VARCHAR(35)	A person's first name, 35 characters is recommended in the UK's Government Data Standards Catalogue (GDSC) (see source below table).
Person	LastName	VARCHAR(35)	A person's last name, 35 characters is recommended in the GDSC.
Person	DateOfBirth	DATE	This is the data type designed to store a date (year, month, day).
Person	StreetName	VARCHAR(35)	35 characters is recommended in the GDSC.
Person	HouseNumber	VARCHAR(35)	35 characters is recommended in the GDSC; and should be varchar, because the house 'number' can contain letters or entire building names.
Person	ZipCode	VARCHAR(10)	10 characters is recommended in the GDSC.
Person	City	VARCHAR(35)	35 characters is recommended in the GDSC.
Person	TelephoneNumber	VARCHAR(15)	15 characters is recommended in the GDSC.
Person	EmailAddress	VARCHAR(255)	255 characters is recommended in the GDSC; and 255 represents the maximum email address length allowed by email providers.
Person	IsUser	BIT	To identify in the supertype Person which overlapping subtypes (User or/and Employee) the Person is, without joining the supertype and subtype tables; BIT can be 1 (to represent True) or 0 (to represent False).
Person	IsEmployee	BIT	
User	RegistrationDate	DATE	This is the data type designed to store a date (year, month, day).
User	EncryptedPassword	VARCHAR(255)	255 characters allows storing of very secure passwords.

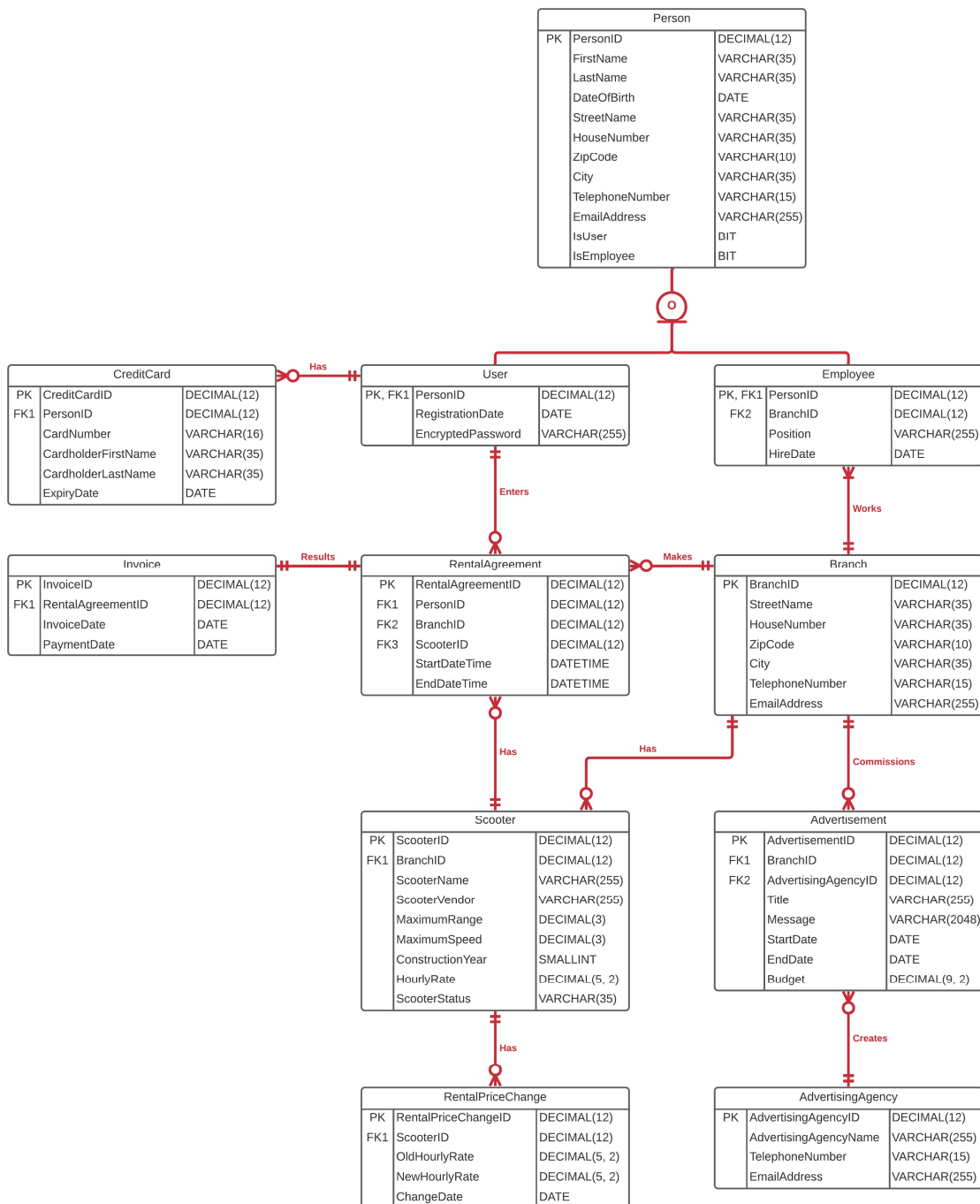
Employee	Position	VARCHAR(255)	Company positions can consist of a large number of words, so 255 character was chosen as safe upper limit.
Employee	HireDate	DATE	This is the data type designed to store a date (year, month, day).
CreditCard	CardNumber	VARCHAR(16)	All major credit card providers have a maximum credit card number length of 16.
CreditCard	CardholderFirstName	VARCHAR(35)	Since the recommendation for first and last name is 35 characters, the same number of characters is used for the first and last name on the credit card.
CreditCard	CardholderLastName	VARCHAR(35)	
CreditCard	ExpiryDate	DATE	This is the data type designed to store a date (year, month, day).
Invoice	InvoiceDate	DATE	
Invoice	PaymentDate	DATE	
RentalAgreement	StartDateTime	DATETIME	This is the data type designed to store a datetime (year, month, day; hours, minutes, seconds). The time must be included so that the total rental time can be calculated in hours.
RentalAgreement	EndDateTime	DATETIME	
Branch	StreetName	VARCHAR(35)	AS recommended in the GDSC.
Branch	HouseNumber	VARCHAR(35)	
Branch	ZipCode	VARCHAR(10)	
Branch	City	VARCHAR(35)	
Branch	TelephoneNumber	VARCHAR(15)	
Branch	EmailAddress	VARCHAR(255)	
Scooter	ScooterName	VARCHAR(255)	A scooter name can contain information about the scooter (because the manufacturer named the scooter that way), so 255 characters were chosen.
Scooter	ScooterVendor	VARCHAR(255)	Company names can be long, so 255 characters were chosen.
Scooter	MaximumRange	DECIMAL(3)	No scooter has a maximum range of more than 999 kilometers, but often more than 100 kilometers; so a number with up to 3 digits was chosen.
Scooter	MaximumSpeed	DECIMAL(3)	No scooter has a maximum speed of more than 999

			kilometers per hour, however, some scooters have a maximum speed of 120 kilometers per hour, so a number with up to 3 digits was chosen.
Scooter	ConstructionYear	SMALLINT	Only the year, e.g. 2022, should be saved. However, since SQL Server does not have a data type YEAR and since DATE also expects a value for day and month, the data type SMALLINT was chosen, which can be used up to the year 32,767 and is 2 bytes smaller than the alternative INT.
Scooter	HourlyRate	DECIMAL(5, 2)	No scooter will cost more than 999.99 euros per hour.
Scooter	ScooterStatus	VARCHAR(35)	35 characters are sufficient to set a scooter status such as operational, nonfunctional or a longer status.
Advertisement	Title	VARCHAR(255)	Advertising titles can be long, therefore 255 characters.
Advertisement	Message	VARCHAR(2048)	An average advertising message consists of less than 80 words. The average length of a word in German is about 8 characters, giving 640 characters + 79 spaces between words, giving 719 characters. So 2048 characters is enough for advertising messages (even for advertising messages that are more than three times the average length).
Advertisement	StartDate	DATE	This is the data type designed to store a date (year, month, day).
Advertisement	EndDate	DATE	
Advertisement	Budget	DECIMAL(9, 2)	Advertising can be expensive, and 9,999,999.99 euros is a safe upper limit for a small company.
AdvertisingAgency	AdvertisingAgencyName	VARCHAR(255)	Company names can be long, so 255 characters were chosen.
AdvertisingAgency	TelephoneNumber	VARCHAR(15)	AS recommended in the GDSC.

AdvertisingAgency	EmailAddress	VARCHAR(255)	No scooter will cost more than 999.99 euros per hour.
RentalPriceChange	OldHourlyRate	DECIMAL(5, 2)	
RentalPriceChange	NewHourlyRate	DECIMAL(5, 2)	This is the data type designed to store a date (year, month, day).
RentalPriceChange	ChangeDate	DATE	

UK's Government Data Standards Catalogue (GDSC):

<https://webarchive.nationalarchives.gov.uk/ukgwa/+http://www.cabinetoffice.gov.uk/media/254290/GDS%20Catalogue%20Vol%202.pdf>



Note on normalization to BCNF:

- 1NF has been achieved: Each table has a primary key and each cell has a single value (e.g., ID columns have one number per cell and not two, three or more).
- 2NF has been achieved: Achieved 1NF and no partial dependencies where an attribute is dependent on only a portion of a composite primary key. No table uses a composite primary key.
- 3NF has been achieved: Achieved 2NF and no transitive dependencies where dependency between non-key attributes exist. In the Person table, the attributes Zip and City do not represent a transitive dependency. In Germany, small villages that are close to each other in rural areas can have the same ZIP code. CardNumber in table CreditCard could create a transitive dependency (with attributes CardholderFirstName, CardholderLastName, ExpiryDate), but only if credit card numbers are globally unique. However, since it could not be ruled out with absolute certainty that the same credit card number exists twice, it is assumed that two identical credit card numbers can exist and then there is no transitive dependency (therefore, CardNumber was not used as the primary key in the CreditCard table, but a CreditCardID was added as the primary key).
- BCNF has been achieved: Achieved 3NF and there are no non-primary key attributes that determine primary key attributes. In the Advertisement table, Title, Message, StartDate, EndDate, Budget may be used together to determine the primary key; however, it is possible for an advertisement to have the same Title, Message, StartDate, EndDate, and Budget, but a different layout or a different advertising format (print or digital), which means it is not the same advertisement (different primary key values) and therefore the Advertisement table is in BCNF. The two tables Branch and AdvertisingAgency have two attributes TelephoneNumber and EmailAddress, which could make it possible to determine the primary keys of both tables. However, the bigger ScooterRent becomes and the bigger the advertising agency is, the more likely it is that a central telephone number and email address are used to route requests (emails and calls) to the appropriate branch or agency. In such a case, a combination of the two attributes TelephoneNumber and EmailAddress cannot determine the primary key to the table Branch and AdvertisingAgency respectively.

Note: it is conceivable to add more tables, such as an Address table, but the term project should solve the use cases and remain reasonable in scope. Therefore, not all conceivable tables have been added.

Stored Procedure Execution and Explanations

A stored procedure named AddBranch was created with parameter validation to add branches. All values passed when executing the stored procedure AddBranch are checked that they are not empty strings, otherwise the error is raised that none of the parameters can be an empty string. The value to the parameter @HouseNumber is checked that it contains at least one number, because house numbers in Germany must always contain at least one number. The value of the parameter @TelephoneNumber is checked to ensure that the value only contains numbers. And the value to the @EmailAddress parameter is checked to ensure that an @ character is included in the value, because every email address must contain an @. If any of the validations is not successful, an error is raised, which informs that the input is incorrect. If all validations are successful, the values are inserted into the database.


```

133 CREATE OR ALTER PROCEDURE AddBranch
134     @StreetName VARCHAR(35),
135     @HouseNumber VARCHAR(35),
136     @ZipCode VARCHAR(10),
137     @City VARCHAR(35),
138     @TelephoneNumber VARCHAR(15),
139     @EmailAddress VARCHAR(255)
140 AS
141 BEGIN
142     -- Parameter validation.
143     IF (@StreetName = '' OR @HouseNumber = '' OR @ZipCode = '' OR @City = '' OR
144         @TelephoneNumber = '' OR @EmailAddress = '')
145     BEGIN
146         RAISERROR('Invalid parameter: none of the parameters can be an empty string.', 18, 0)
147         RETURN
148     END
149
150     IF (@HouseNumber NOT LIKE '%[0-9]%')
151     BEGIN
152         RAISERROR('Invalid parameter: @HouseNumber must contain at least one number.', 18, 0)
153         RETURN
154     END
155
156     IF (@TelephoneNumber LIKE '%[^0-9]%')
157     BEGIN
158         RAISERROR('Invalid parameter: @TelephoneNumber must only contain numbers.', 18, 0)
159         RETURN
160     END
161
162     IF (@EmailAddress NOT LIKE '%@%')
163     BEGIN
164         RAISERROR('Invalid parameter: @EmailAddress must contain an @ sign.', 18, 0)
165         RETURN
166     END
167     INSERT INTO Branch(BranchID, StreetName, HouseNumber, ZipCode, City, TelephoneNumber, EmailAddress)
168     VALUES(NEXT VALUE FOR BranchSeq, @StreetName, @HouseNumber, @ZipCode, @City, @TelephoneNumber, @EmailAddress);
169 END;

```

121 %
 Messages
 Commands completed successfully.

```

172 BEGIN TRANSACTION AddBranch;
173 EXECUTE AddBranch 'Dorotheenplatz', '1A', '70173', 'Stuttgart', '071120702009', 'hub@scooterrent.com';
174 COMMIT TRANSACTION AddBranch;

```

121 %
 Messages
 (1 row affected)

A second stored procedure named AddPersonAsUserOrEmployee was created with parameter validation to add a person (supertype) as user (subtype) or a person as employee (subtype). All varchar values passed when executing the stored procedure AddPersonAsUserOrEmployee are checked that they are not empty strings, otherwise the error is raised that none of the parameters can be an empty string. The value to the parameter @HouseNumber is checked that it contains at least one number, because house numbers in Germany must always contain at least one number. The value of the parameter @TelephoneNumber is checked to ensure that the value only contains numbers. The value to the @EmailAddress parameter is checked to ensure that an @ character is included in the value, because every email address must contain an @. And all DATE values are checked: @DateOfBirth if the person is older than 120 years; @RegistrationDate if the user registered before the creation of the ScooterRent app on 01/01/2019 (this is not possible); and @HireDate if the employee was hired before ScooterRent was founded on 01/01/2018 (this is not possible). If any of the validations is not successful, an error is

raised, which informs that the input is incorrect. If all validations are successful, the values are inserted into the database. To insert the subtypes, if statements are used to check whether the person is a user or an employee to insert the corresponding values in the User or employee table.

Note: The screenshot for the second stored procedure had to be split up because the entire code did not fit on the screen.

```

211 CREATE OR ALTER PROCEDURE AddPersonAsUserOrEmployee
212 -- Person.
213 @FirstName VARCHAR(35),
214 @LastName VARCHAR(35),
215 @DateOfBirth DATE,
216 @StreetName VARCHAR(35),
217 @HouseNumber VARCHAR(35),
218 @ZipCode VARCHAR(10),
219 @City VARCHAR(35),
220 @TelephoneNumber VARCHAR(15),
221 @EmailAddress VARCHAR(255),
222 @IsUser BIT,
223 @IsEmployee BIT,
224 -- User.
225 @RegistrationDate DATE,
226 @EncryptedPassword VARCHAR(255),
227 -- Employee.
228 @BranchID DECIMAL(12),
229 @Position VARCHAR(255),
230 @HireDate DATE
231 AS
232 BEGIN
233 -- Parameter validation.
234 IF (@FirstName = '' OR @LastName = '' OR @StreetName = '' OR @HouseNumber = '' OR
235 @ZipCode = '' OR @City = '' OR @TelephoneNumber = '' OR @EmailAddress = '')
236 BEGIN
237 RAISERROR('Invalid parameter: none of the parameters can be an empty string.', 18, 0)
238 RETURN
239 END
240
241 IF (@HouseNumber NOT LIKE '%[0-9]%')
242 BEGIN
243 RAISERROR('Invalid parameter: @HouseNumber must contain at least one number.', 18, 0)
244 RETURN
245 END
246
247 IF (@TelephoneNumber LIKE '%[^0-9]%')
248 BEGIN
249 RAISERROR('Invalid parameter: @TelephoneNumber must only contain numbers.', 18, 0)
250 RETURN
251 END
252
253 IF (@EmailAddress NOT LIKE '%@%')
254 BEGIN
255 RAISERROR('Invalid parameter: @EmailAddress must contain an @ sign.', 18, 0)
256 RETURN
257 END
258
259 IF (DATEDIFF(YEAR, @DateOfBirth, GETDATE()) > 120)
260 BEGIN
261 RAISERROR('Invalid parameter: @DateOfBirth; person cannot be older than 120 years.', 18, 0)
262 RETURN
263 END
264
265 DECLARE @PersonID DECIMAL(12);
266 SET @PersonID = NEXT VALUE FOR PersonSeq;
267 -- Inserts Person.
268 INSERT INTO Person(PersonID, FirstName, LastName, DateOfBirth, StreetName, HouseNumber, ZipCode,
269 City, TelephoneNumber, EmailAddress, IsUser, IsEmployee)
270 VALUES(@PersonID, @FirstName, @LastName, @DateOfBirth, @StreetName, @HouseNumber, @ZipCode,
271 @City, @TelephoneNumber, @EmailAddress, @IsUser, @IsEmployee);
272 -- Inserts User.
273 IF (@IsUser = 1)
274 BEGIN
275 IF (@RegistrationDate < '01/01/2019')
276 BEGIN
277 RAISERROR('Invalid parameter: @RegistrationDate must be after ScooterRent was created on 01/01/2019.', 18, 0)
278 RETURN
279 END
280 INSERT INTO [User](PersonID, RegistrationDate, EncryptedPassword) VALUES(@PersonID, @RegistrationDate, @EncryptedPassword);
281 END

```

```

282 -- Inserts Employee.
283 IF (@IsEmployee = 1)
284 BEGIN
285     IF (@HireDate < '01/01/2018')
286     BEGIN
287         RAISERROR('Invalid parameter: @HireDate must be after ScooterRent was founded on 01/01/2018.', 18, 0)
288         RETURN
289     END
290     INSERT INTO Employee(PersonID, BranchID, Position, HireDate) VALUES(@PersonID, @BranchID, @Position, @HireDate);
291 END
292 END;

```

121 %
Messages
Commands completed successfully.

Inserts Person as User:

```

295 BEGIN TRANSACTION AddPersonAsUserOrEmployee;
296 EXECUTE AddPersonAsUserOrEmployee 'Abby', 'Aoe', '06/22/1990', 'Sonnenallee', '94', '86074', 'Augsburg', '0821794234', 'AA@gmail.com', 1,
297 '05/24/2020', 'qQuQ8VI60eZzJQt1sJhm', NULL, NULL, NULL;
298 COMMIT TRANSACTION AddPersonAsUserOrEmployee;

```

121 %
Messages
(1 row affected)
(1 row affected)

Inserts Person as Employee:

```

346 BEGIN TRANSACTION AddPersonAsUserOrEmployee;
347 EXECUTE AddPersonAsUserOrEmployee 'Katie', 'Koe', '02/28/1987', 'Kronprinzstraße', '11', '70173', 'Stuttgart', '071165626187', 'KK@gmail.com', 0, 1,
348 NULL, NULL, 1, 'Operations manager', '07/27/2019';
349 COMMIT TRANSACTION AddPersonAsUserOrEmployee;

```

121 %
Messages
(1 row affected)
(1 row affected)

A third stored procedure named AddScooter was created with parameter validation for adding scooters. All values of type varchar passed when executing AddScooter are checked that they are not empty strings, otherwise the error is raised that none of the parameters can be an empty string. The values to the parameters @MaximumRange and @MaximumSpeed are checked to ensure that they are at least greater than zero, because a scooter cannot have a negative (maximum) range and negative speed. The value of the @ConstructionYear parameter is checked to ensure that the value for the construction year is greater than 1999, because according to the project direction it can be ruled out that scooters with a construction year before 2000 are rented. And the value for the @HourlyRate parameter is checked to ensure that it is not less than or equal to 0, because there can be no negative rental prices and because the scooters are not rented out for free, so the hourly rate (price) must always be larger than zero euros. If the validations are not successful, an error is raised, which informs that the input is incorrect. If all validations are successful, the values are inserted into the database.

```

377 CREATE OR ALTER PROCEDURE AddScooter
378     @BranchID DECIMAL(12),
379     @ScooterName VARCHAR(255),
380     @ScooterVendor VARCHAR(255),
381     @MaximumRange DECIMAL(3),
382     @MaximumSpeed DECIMAL(3),
383     @ConstructionYear SMALLINT,
384     @HourlyRate DECIMAL(5, 2),
385     @ScooterStatus VARCHAR(35)
386 AS
387 BEGIN
388     -- Parameter validation.
389     IF (@ScooterName = '' OR @ScooterVendor = '' OR @ScooterStatus = '')
390     BEGIN
391         RAISERROR('Invalid parameter: @ScooterName, @ScooterVendor or @ScooterStatus cannot be an empty string.', 18, 0)
392         RETURN
393     END
394
395     IF (@MaximumRange <= 0 OR @MaximumSpeed <= 0)
396     BEGIN
397         RAISERROR('Invalid parameter: @MaximumRange or @MaximumSpeed must be greater than zero.', 18, 0)
398         RETURN
399     END
400
401     IF (@ConstructionYear <= 1999)
402     BEGIN
403         RAISERROR('Invalid parameter: @ConstructionYear must be after 1999.', 18, 0)
404         RETURN
405     END
406
407     IF (@HourlyRate <= 0)
408     BEGIN
409         RAISERROR('Invalid parameter: @HourlyRate must be greater than zero.', 18, 0)
410         RETURN
411     END
412     INSERT INTO Scooter(ScooterID, BranchID, ScooterName, ScooterVendor, MaximumRange, MaximumSpeed, ConstructionYear, HourlyRate, ScooterStatus)
413     VALUES(NEXT VALUE FOR ScooterSeq, @BranchID, @ScooterName, @ScooterVendor, @MaximumRange, @MaximumSpeed, @ConstructionYear, @HourlyRate, @ScooterStatus);
414 END;

```

Messages
Commands completed successfully.

```

417 BEGIN TRANSACTION AddScooter;
418 EXECUTE AddScooter 1, 'TVS iQube Electric', 'TVS', 75, 78, 2022, 19.99, 'Operational';
419 COMMIT TRANSACTION AddScooter;

```

Messages
(1 row affected)

Question Identification and Explanations

- With which 10 users did scooter rent achieve the highest annual revenue in 2022?
Explanation:
The marketing manager would like to know this to specifically contact the most valuable customers to bind them to ScooterRent in the long term. Customers who are particularly valuable to ScooterRent should be given more attention. Note: In reality the marketing manager would not ask for 10 users, but for 10% of the users who generate the highest sales; however, the term project database does not have thousands of users, so 10% would only show one or two entries. So the question asked about 10 users and not 10% of the users.
- Which invoices have been overdue for how many days and for what amount? So that the corresponding users can be asked to pay.
Explanation:
The accounting manager would like to know this to contact these customers to ask them to pay the invoice amount. Above all, a startup must ensure that the cash does not run out, which could

happen if scooters are constantly being rented, which incur costs and cash outflows, but the invoices are not paid by the users, so there are no or few cash inflows.

- How many scooters were rented this and last year per branch?

Explanation:

That is what the general manager wants to know. Every company has unit sales figures (the number of scooters that were rented) and ScooterRent has them for each branch. The general manager checks whether the unit sales goals are being achieved; and these goals are compared to previous years to see how the current year compares to the past year. In the case of a startup, how much credit and venture capital are granted and under what conditions depend on unit sales figures. Also, if the quantities sold are too high or too low, the general manager must take action and buy or sell assets, hire or fire employees, etc.

Query Executions and Explanations

Since the marketing manager wants to know the 10 users with the highest revenue, the RentalAgreement (ScooterID, PersonID, StartDateTime, and EndDateTime of the rental), Scooter (HourlyRate), User (PersonID; to access personal data), and Person (FirstName, LastName, EmailAddress) tables need to be joined. Since only the year 2022 is of interest, the year 2022 is filtered in line 530. In line 524, the revenue per user is calculated by calculating the difference between the EndDateTime and StartDateTime of the rental in minutes, then this value is divided by 60 minutes and then multiplied by the HourlyRate (rental price per hour). However, since the sum of the revenues per user is required, the revenues per user are grouped per user based on the PersonID and summed up. Then the revenue amounts are formatted: no decimal place is displayed, a thousand separator is inserted, and the euro sign is appended. However, since the revenue amounts are now formatted as a varchar, the varchar must first be sorted in line 532 in descending order of length and then in descending order of numeric values, so that the values are sorted from highest to lowest (in descending order). If this were not done then 840 € would be listed first as sorting by varchar does not take into account the length of the varchar. Since the question was asked by the marketing manager, who wants to retain the users with the highest revenue in the long term and will contact them for this purpose, the first and last names and the email addresses of the users were included in the results table. The PersonID (User ID column) has also been included so that users can be uniquely identified and, if necessary, the marketing manager can look up further personal data in the database.

```

523 -- With which 10 users did scooter rent achieve the highest annual revenue in 2022?
524 SELECT TOP 10 FORMAT(SUM(DATEDIFF(MINUTE, StartDateTime, EndDateTime) / 60.0 * HourlyRate), '#,0. €') AS 'Total annual revenue in 2022 per user',
525     Person.PersonID AS 'User ID', FirstName AS 'First name', LastName AS 'Last name', EmailAddress AS 'Email address'
526 FROM RentalAgreement
527 JOIN Scooter ON Scooter.ScooterID = RentalAgreement.ScooterID
528 JOIN [User] ON [User].PersonID = RentalAgreement.PersonID
529 JOIN Person ON Person.PersonID = RentalAgreement.PersonID
530 WHERE YEAR(EndDateTime) = 2022
531 GROUP BY Person.PersonID, FirstName, LastName, EmailAddress
532 ORDER BY LEN(SUM(DATEDIFF(MINUTE, StartDateTime, EndDateTime) / 60.0 * HourlyRate)) DESC, 'Total annual revenue in 2022 per user' DESC;

```

	Total annual revenue in 2022 per user	User ID	First name	Last name	Email address
1	3,671 €	1	Abby	Aoe	AA@gmail.com
2	1,707 €	2	Brian	Boe	BB@gmail.com
3	1,483 €	3	Charlie	Coe	CC@gmail.com
4	1,399 €	4	Daisy	Doe	DD@gmail.com
5	1,287 €	5	Elliott	Eoe	EE@gmail.com
6	840 €	7	Graham	Goe	GG@gmail.com
7	672 €	9	Isabella	Ioe	II@gmail.com
8	644 €	10	Jessica	Joe	JJ@gmail.com
9	644 €	8	Hudson	Hoe	HH@gmail.com
10	560 €	6	Finley	Foe	FF@gmail.com

To answer the accounting manager's question, the tables Invoice, RentalAgreement, Scooter, User, and Person had to be joined. Since an invoice can only be overdue if it has not yet been paid and if it has not

been paid within 10 days, line 543 filters the rows accordingly. Line 535 calculates how many days the invoice is overdue. To do this, the current date (GETDATE()) is subtracted from the InvoiceDate and 10 days are subtracted from this difference (because a user can only be overdue after 10 days). In line 536, the invoice amount is determined to show the accounting manager how important it is that the invoice is paid quickly (an overdue 20 € invoice is less of a problem than an overdue 20,000 € invoice). To uniquely identify the overdue invoices, the InvoiceIDs are part of the results table; also, the invoice date was included in the results table, so that the accounting manager can be sure that the days overdue are correct. And since the accounting manager has to contact users to clarify why the payment has not yet been made, the first and last names and email addresses of the corresponding users have been included in the results table.

```

534 -- Which invoices have been overdue for how many days and for what amount?
535 SELECT InvoiceID, InvoiceDate AS 'Invoice date', DATEDIFF(DAY, InvoiceDate, GETDATE()) - 10 AS 'Days overdue',
536        FORMAT(DATEDIFF(MINUTE, StartDateTime, EndDateTime) / 60.0 * HourlyRate, '.00 €') AS 'Amount overdue',
537        FirstName AS 'First name', LastName AS 'Last name', EmailAddress AS 'Email address'
538 FROM Invoice
539 JOIN RentalAgreement ON RentalAgreement.RentalAgreementID = Invoice.RentalAgreementID
540 JOIN Scooter ON Scooter.ScooterID = RentalAgreement.ScooterID
541 JOIN [User] ON [User].PersonID = RentalAgreement.PersonID
542 JOIN Person ON Person.PersonID = [User].PersonID
543 WHERE PaymentDate IS NULL AND DATEDIFF(DAY, InvoiceDate, GETDATE()) > 10;

```

	InvoiceID	Invoice date	Days overdue	Amount overdue	First name	Last name	Email address
1	16	2022-01-23	7	671.72 €	Isabella	Ioe	II@gmail.com
2	17	2022-01-25	5	643.72 €	Jessica	Joe	JJ@gmail.com
3	18	2022-01-27	3	699.72 €	Abby	Aoe	AA@gmail.com
4	19	2022-01-29	1	727.72 €	Brian	Boe	BB@gmail.com

Note: The query in the screenshot was executed on 2/9/2022. And an invoice is overdue if it has not been paid after 10 days. That means the number of overdue days (column 'Days overdue') is today's date minus the invoice date minus 10 days.

The general manager wants to know how many scooters were rented per branch this year and last year. A view is useful for this because this query is sent to the database over and over again throughout the years. The view was defined in such a general way that it can be used regardless of the year without adjustment. The view is defined in lines 547 to 556. The view consists of two SELECT statements that are combined in line 552 via UNION. In line 550, the first select statement filters for all rental agreements with StartDateTime values in the current year and groups the values by BranchID. In line 548, the count of RentalAgreementID is determined, which corresponds to the number of scooters rented. Exactly the same is done in the second select statement after UNION starting at line 553, except that in line 555 one is subtracted from the current year to filter for all rental agreements with StartDateTime values in the previous year. And on line 560, the view is used in a select statement that sorts the results table by BranchID ascending and then by the Year column descending.

```

545 -- How many scooters were rented this and last year per branch?
546 -- Defines view.
547 CREATE OR ALTER VIEW View_scooters_rented_per_branch_this_and_last_year AS
548 SELECT BranchID, COUNT(RentalAgreementID) AS 'Number of rented scooters per branch', YEAR(GETDATE()) AS 'Year'
549 FROM RentalAgreement
550 WHERE YEAR(StartDateTime) = YEAR(GETDATE())
551 GROUP BY BranchID
552 UNION
553 SELECT BranchID, COUNT(RentalAgreementID) AS 'Number of rented scooters per branch', YEAR(GETDATE()) - 1 AS 'Year'
554 FROM RentalAgreement
555 WHERE YEAR(StartDateTime) = YEAR(GETDATE()) - 1
556 GROUP BY BranchID
557
558 -- Uses view.
559 SELECT *
560 FROM View_scooters_rented_per_branch_this_and_last_year
561 ORDER BY BranchID, [Year] DESC;

```

BranchID	Number of rented scooters per branch	Year
1	7	2022
1	2	2021
2	3	2022
2	4	2021
3	5	2022
3	1	2021

Index Identification and Creations

With SQL Server, the primary key columns have unique indexes by default:

Primary key column	Description
Person.PersonID	This is the primary key of the Person table.
User.PersonID	This is the primary key of the User table.
Employee.PersonID	This is the primary key of the Employee table.
Branch.BranchID	This is the primary key of the Branch table.
Scooter.ScooterID	This is the primary key of the Scooter table.
RentalAgreement.RentalAgreementID	This is the primary key of the RentalAgreement table.
Invoice.InvoiceID	This is the primary key of the Invoice table.
CreditCard.CreditCardID	This is the primary key of the CreditCard table.
Advertisement.AdvertisementID	This is the primary key of the Advertisement table.
AdvertisingAgency.AdvertisingAgencyID	This is the primary key of the AdvertisingAgency table.
RentalPriceChange.RentalPriceChangeID	This is the primary key of the RentalPriceChange table.

Unlike primary keys, foreign keys have no indexes by default – and are not always unique; therefore, the indexes must be set manually and it must be specified whether a unique or non-unique index is required:

Foreign key column	Description
User.PersonID	The foreign key in the User table references the Person table. User is a subtype of Person and uses the foreign key as the primary key, so the foreign key already has a unique index.
Employee.PersonID	The foreign key in the Employee table references the Person table. Employee is a subtype of Person and uses the foreign key as the primary key, so the foreign key already has a unique index.

Employee.BranchID	The foreign key in the Employee table references the Branch table. Each branch has one to many employees, a non-unique index must be used.
CreditCard.PersonID	The foreign key in the CreditCard table references the User table. Each user may have many credit cards, a non-unique index must be used.
Invoice.RentalAgreementID	The foreign key in the Invoice table references the RentalAgreement table. Each rental agreement results in one invoice, a unique index must be used.
RentalAgreement.PersonID	The foreign key in the RentalAgreement table references the User table. Each user may have many rental agreements, a non-unique index must be used.
RentalAgreement.BranchID	The foreign key in the RentalAgreement table references the Branch table. Each branch may have many rental agreements, a non-unique index must be used.
RentalAgreement.ScooterID	The foreign key in the RentalAgreement table references the Scooter table. Each scooter may be part of many rental agreements, a non-unique index must be used.
Scooter.BranchID	The foreign key in the Scooter table references the Branch table. Each branch may have many scooters, a non-unique index must be used.
Advertisement.BranchID	The foreign key in the Advertisement table references the Branch table. Each branch may have many advertisements, a non-unique index must be used.
Advertisement.AdvertisingAgencyID	The foreign key in the Advertisement table references the AdvertisingAgency table. Each advertising agency may create many advertisements, a non-unique index must be used.
RentalPriceChange.ScooterID	The foreign key in the RentalPriceChange table references the Scooter table. Each scooter may have many rental price changes, a non-unique index must be used.

As for the three query driven indexes; it is good practice to put indexes on the columns in JOINS and the columns in the WHERE clauses (assuming that the tables do not only consist of a few rows and contain sufficiently different values). Since all columns in JOIN statements are primary and foreign keys in this implementation, the indexes have already been explained above.

For the first query that answers the question ‘with which 10 users did scooter rent achieve the highest annual revenue in 2022’, YEAR(RentalAgreement.EndDateTime) is to be indexed. However, a standard index cannot be used here, since the YEAR() function changes the value of column RentalAgreement.EndDateTime. A **non-unique** function-based index must be applied to YEAR(RentalAgreement.EndDateTime). As the database is used for many years, a new year will be added each year, making the function-based index useful. However, since this is not within the scope of the term project and course, no function-based index was included in the SQL code.

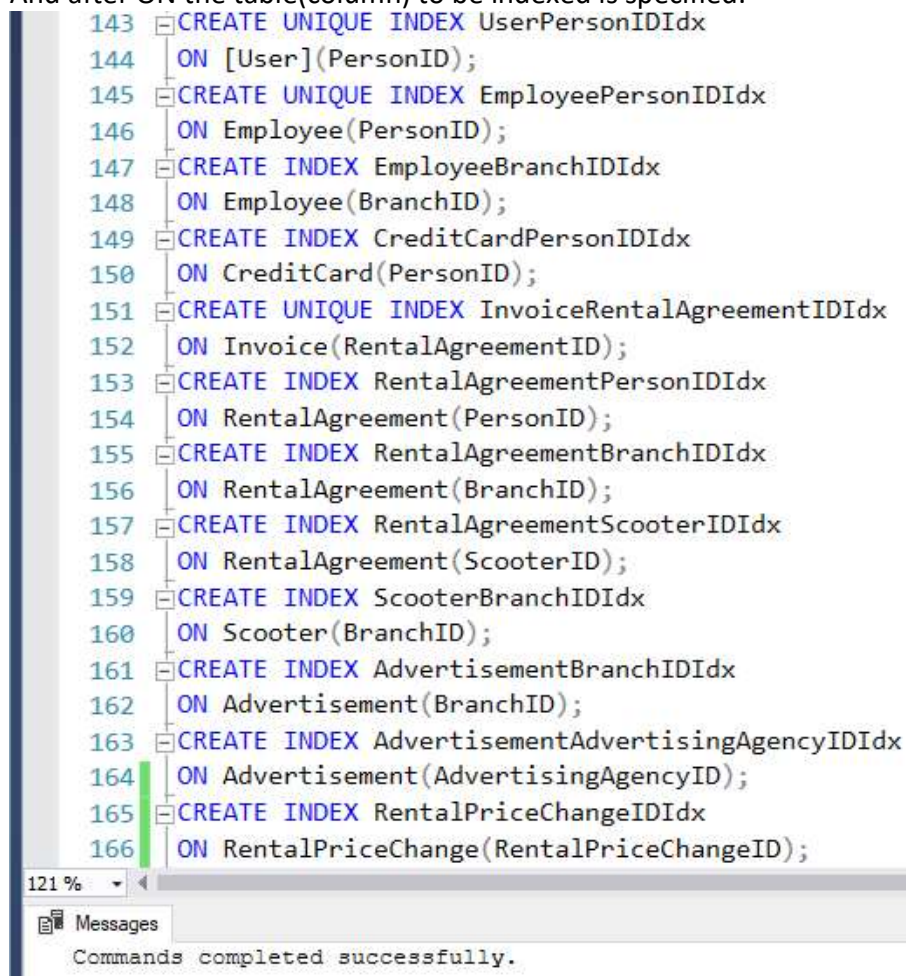
For the second query that answers the question ‘which invoices have been overdue for how many days and for what amount’, the Invoice.PaymentDate in the WHERE clause should not be indexed, since it

only checks if the values for Invoice.PaymentDate are NULL. On the other side, DATEDIFF(DAY, Invoice.InvoiceDate, GETDATE()) is to be indexed. However, a standard index cannot be used, since the function changes the value of column Invoice.InvoiceDate. A **non-unique** function-based index must be applied. However, since this is not within the scope of the term project and course, no function-based index was included in the SQL code.

For the third query that answers the question 'how many scooters were rented this and last year per branch', YEAR(RentalAgreement.StartDateTime) is to be indexed. A **non-unique** function-based index must be applied to YEAR(RentalAgreement.StartDateTime). However, since this is not within the scope of the term project and course, no function-based index was included in the SQL code.

This means that in all three queries, standard indexes are only applied to all foreign keys in the JOIN statements.

Screenshots showing the creation of the indexes; CREATE INDEX creates a non-unique index and CREATE UNIQUE INDEX creates a unique index. This is followed by the index name, for example UserPersonIDIdx, which consists of the table name User, column name PersonID, and Idx, but can be named differently. And after ON the table(column) to be indexed is specified:



```
143 CREATE UNIQUE INDEX UserPersonIDIdx
144 ON [User](PersonID);
145 CREATE UNIQUE INDEX EmployeePersonIDIdx
146 ON Employee(PersonID);
147 CREATE INDEX EmployeeBranchIDIdx
148 ON Employee(BranchID);
149 CREATE INDEX CreditCardPersonIDIdx
150 ON CreditCard(PersonID);
151 CREATE UNIQUE INDEX InvoiceRentalAgreementIDIdx
152 ON Invoice(RentalAgreementID);
153 CREATE INDEX RentalAgreementPersonIDIdx
154 ON RentalAgreement(PersonID);
155 CREATE INDEX RentalAgreementBranchIDIdx
156 ON RentalAgreement(BranchID);
157 CREATE INDEX RentalAgreementScooterIDIdx
158 ON RentalAgreement(ScooterID);
159 CREATE INDEX ScooterBranchIDIdx
160 ON Scooter(BranchID);
161 CREATE INDEX AdvertisementBranchIDIdx
162 ON Advertisement(BranchID);
163 CREATE INDEX AdvertisementAdvertisingAgencyIDIdx
164 ON Advertisement(AdvertisingAgencyID);
165 CREATE INDEX RentalPriceChangeIDIdx
166 ON RentalPriceChange(RentalPriceChangeID);
```

121 %

Messages

Commands completed successfully.

History Table Demonstration

The history table RentalPriceChange records the hourly rate changes (scooter rental price changes) via a trigger, which triggers on the table Scooter after an update of the hourly rate and records the old and new hourly rate respectively in the history table with a reference to the scooter, and the change date, so that the price changes can be tracked.

Trigger RentalPriceChangeTrigger below works in detail as follows: In lines 700 to 702 the trigger RentalPriceChangeTrigger is defined to trigger AFTER an UPDATE statement on the Scooter table. In lines 705 through 706, @OldHourlyRate and @NewHourlyRate are assigned the HourlyRate before (... FROM DELETED) and after (... FROM INSERTED) the update. In line 708 it is checked whether the value of HourlyRate has changed, if not then nothing happens. Otherwise, the INSERT statement is executed in lines 709 to 710, which inserts the unique primary key value, uses the SELECT statement to determine the ScooterID as a foreign key for the scooter that has an updated HourlyRate, inserts the old and new hourly rate, and inserts the change date into the history table RentalPriceChange.

```
700 CREATE OR ALTER TRIGGER RentalPriceChangeTrigger
701 ON Scooter
702 AFTER UPDATE
703 AS
704 BEGIN
705     DECLARE @OldHourlyRate DECIMAL(5, 2) = (SELECT HourlyRate FROM DELETED);
706     DECLARE @NewHourlyRate DECIMAL(5, 2) = (SELECT HourlyRate FROM INSERTED);
707
708     IF (@OldHourlyRate <> @NewHourlyRate)
709     INSERT INTO RentalPriceChange(RentalPriceChangeID, ScooterID, OldHourlyRate, NewHourlyRate, ChangeDate)
710     VALUES(NEXT VALUE FOR RentalPriceChangeSeq, (SELECT ScooterID FROM INSERTED), @OldHourlyRate, @NewHourlyRate, GETDATE());
711 END;
```

121 %

Messages

Commands completed successfully.

The UPDATE queries and the history table, which captured the changes after the updates.

```
714 UPDATE Scooter SET HourlyRate = 24.99 WHERE ScooterID = 1;
715 UPDATE Scooter SET HourlyRate = 26.99 WHERE ScooterID = 2;
716 UPDATE Scooter SET HourlyRate = 34.99 WHERE ScooterID = 3;
717 UPDATE Scooter SET HourlyRate = 14.99 WHERE ScooterID = 4;
718 UPDATE Scooter SET HourlyRate = 54.99 WHERE ScooterID = 5;
719 UPDATE Scooter SET HourlyRate = 64.99 WHERE ScooterID = 5;
720 UPDATE Scooter SET HourlyRate = 74.99 WHERE ScooterID = 5;
721
722 SELECT * FROM RentalPriceChange;
```

121 %

Results Messages

	RentalPriceChangeID	ScooterID	OldHourlyRate	NewHourlyRate	ChangeDate
1	1	1	19.99	24.99	2022-02-13
2	2	2	34.99	26.99	2022-02-13
3	3	3	22.99	34.99	2022-02-13
4	4	4	24.99	14.99	2022-02-13
5	5	5	19.99	54.99	2022-02-13
6	6	5	54.99	64.99	2022-02-13
7	7	5	64.99	74.99	2022-02-13

Data Visualizations

Further UPDATE statements need to be executed for the first data visualization using the history table RentalPriceChange. Below is the resulting history table.

723 `SELECT * FROM RentalPriceChange;`

	RentalPriceChangeID	ScooterID	OldHourlyRate	NewHourlyRate	ChangeDate
1	1	1	19.99	24.99	2022-02-14
2	2	2	34.99	26.99	2022-02-14
3	3	3	22.99	34.99	2022-02-14
4	4	4	24.99	14.99	2022-02-14
5	5	5	19.99	54.99	2022-02-14
6	6	5	54.99	64.99	2022-02-14
7	7	5	64.99	74.99	2022-02-14
8	8	1	24.99	14.99	2022-02-13
9	9	2	26.99	16.99	2022-02-13
10	10	3	34.99	14.99	2022-02-13
11	11	5	74.99	14.99	2022-02-13
12	12	1	14.99	84.99	2022-02-12
13	13	2	16.99	96.99	2022-02-12
14	14	3	14.99	64.99	2022-02-12
15	15	4	14.99	54.99	2022-02-12
16	16	5	14.99	94.99	2022-02-12

With the following query, the history table is grouped by the change date and for each change date the average (AVG) hourly rate change is calculated as AverageHourlyRateChange, formatted as a euro amount with two decimal places, and the number of changes (COUNT) is determined for each date as NumberChanges.

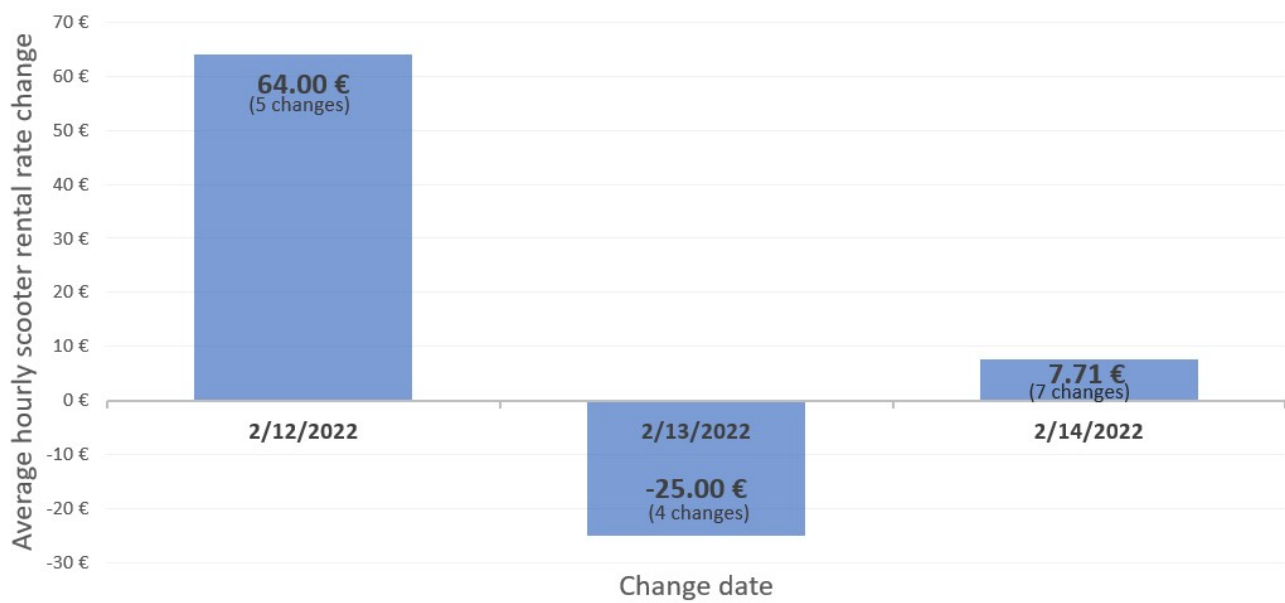
727 `-- 1) Query history table RentalPriceChange.`
728 `SELECT ChangeDate, FORMAT(AVG(NewHourlyRate - OldHourlyRate), '0.00 €') AS AverageHourlyRateChange,`
729 `COUNT(RentalPriceChangeID) AS NumberChanges`
730 `FROM RentalPriceChange`
731 `GROUP BY ChangeDate;`

	ChangeDate	AverageHourlyRateChange	NumberChanges
1	2022-02-12	64.00 €	5
2	2022-02-13	-25.00 €	4
3	2022-02-14	7.71 €	7

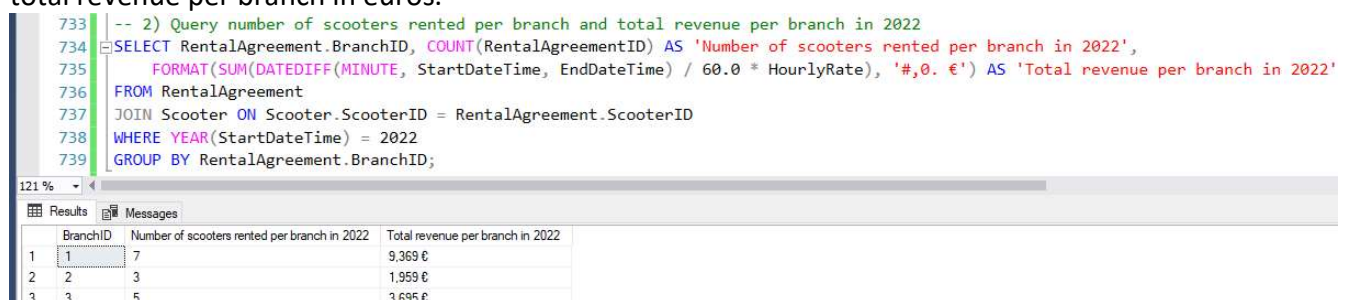
The following visualization of the query 'history table RentalPriceChange' shows a bar chart that shows the 'average hourly scooter rental rate change' on the y-axis and the 'change date' on the x-axis. On 2/12/2022 the average hourly rate over five scooter hourly rate changes increased by 64 €. A day later on 2/13/2022 the average hourly rate across four scooters has fallen by 25 €, and on 2/14/2022 the average hourly rate across seven scooters has increased by 7.71 €. This means that the net average change over the three days from 2/12/2022 to 2/14/2022 was $(64 € * 5 \text{ changes} - 24 € * 4 \text{ changes} + 7.71 € * 7 \text{ changes}) / (5 + 4 + 7 \text{ changes}) = 17.12 €$, which corresponds to a strong average hourly scooter rental rate increase. Now the question has to be asked, why did the hourly rate change so much within 3

days and why by such a high amount of 17.12 €. That can only be answered by speaking to ScooterRent's employees. Furthermore, knowing the hourly rate changes provides an excellent basis for comparing the development of profits with the hourly rate changes to determine what effect the hourly rate changes have on ScooterRent's profit – an hourly rate increase may increase profits or decrease profits because fewer scooters are rented out.

Average hourly scooter rental rate change from 2/12/2022 to 2/14/2022



For the second query 'number of scooters rented per branch and total revenue per branch in 2022' no further data had to be inserted into the database. In line 738, the query filters for all rows that have a StartDateTime with the year 2022 in the RentalAgreement table. In line 739, the rows are grouped by BranchID. And in lines 734 to 735, the number of rental agreements per branch is determined and the total revenue per branch in euros.

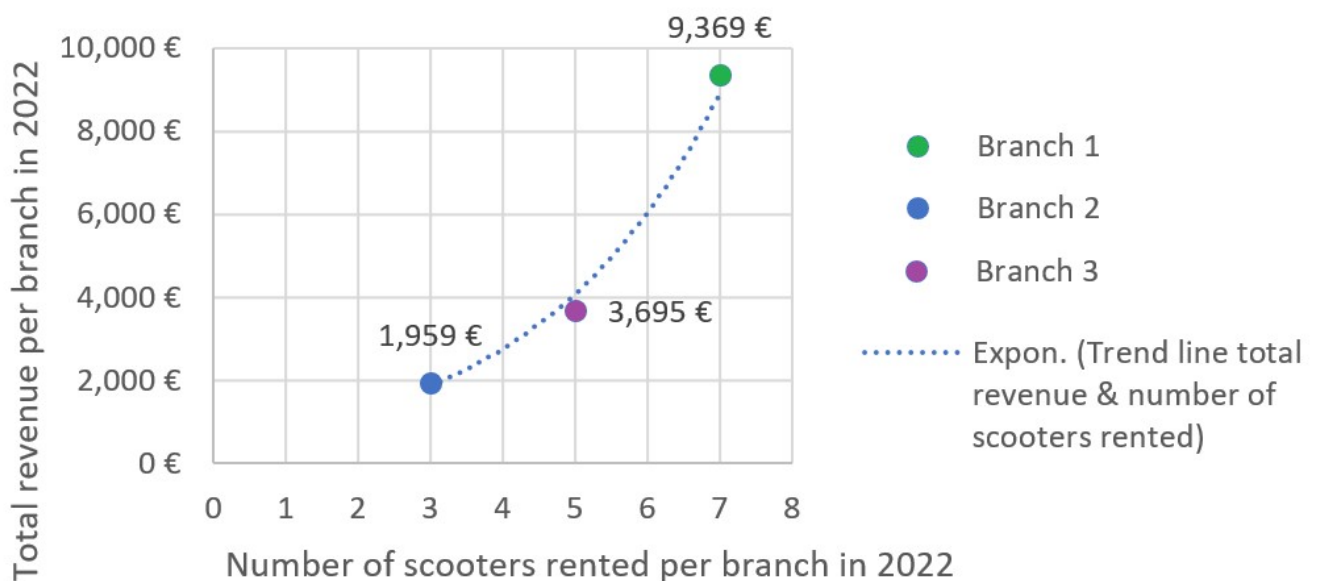


The diagram for query 'number of scooters rented per branch and total revenue per branch in 2022' shows a scatter plot that plots the total revenue per branch in 2022 on the y-axis and the number of scooters rented per branch in 2022 on the x-axis. A blue dot with the coordinates (3, 1959 €) was plotted, which represents the annual revenue of branch 2 for the year 2022 in the amount of €1,959 (3 rented scooters). A violet dot with the coordinates (5, 3695 €) was plotted, which represents the annual revenue of branch 3 for the year 2022 in the amount of 3,695 € (5 rented scooters). And a green dot with the coordinates (7, 9369 €) was plotted, which represents the annual revenue of branch 1 for the year 2022 in the amount of 9,369 € (7 rented scooters). Thus, in 2022, branch 2 rented out the fewest

scooters and generated the lowest revenue, while branch 1 generated the highest revenue and rented out the most scooters. It can also be seen that branch 3 rented out 2 more scooters than branch 2 and generated 1,736 € more in sales than branch 2. And branch 1 has rented out 2 more scooters than branch 3 and generates 5,674 € more in sales than branch 3. The increase in sales per scooter was therefore more than twice as high from branch 3 to branch 1 as from branch 2 to branch 3. This is also shown by the dashed exponential trend line. Based on the data, it can be assumed that scooters in branch 1 are rented out longer and/or scooters that have a higher hourly rate are rented out. Likewise, more scooters are rented out in branch 1. This may be because branch 1 is more centrally located, has advertised more, or has more loyal customers who keep renting scooters. In order to prove or disprove these assumptions, further database queries must be created and executed and the employees and customers in the branches must be spoken to. This will determine why branch 1 has so much higher sales per scooter – and these learnings can be applied to the other branches to increase sales.

Note: The query and the diagram contain few data points, so it is not sensible to make such assumptions; however, if this was not a database for a term project, there would be thousands of rented scooters in the database and exactly the same query and visualization could be used to make the assumptions and conclusions as above.

Total revenue per branch & number of scooters rented per branch in 2022



Summary and Reflection

My database is for a mobile app named ScooterRent, which enables registered user who must be at least 14 years old to rent and return electric scooters in three branches in Stuttgart, Germany – and to pay the rental amount by credit card. The following entities were identified: Person (supertype), User (subtype of Person), Employee (subtype of Person), Branch, Scooter, RentalAgreement, Invoice, CreditCard, Advertisement, AdvertisingAgency, and RentalPriceChange (history table); where

RentalAgreement has the most relationships to other entities. The following most useful relationships have been identified: User is a Person, Employee is a Person, User has a CreditCard, Employee works for Branch, Branch makes RentalAgreement, User enters RentalAgreement, RentalAgreement includes Scooter, RentalAgreement results in an Invoice, Branch commissions Advertisement, AdvertisingAgency creates Advertisement, and Scooter has RentalPriceChanges. To model the many-to-many relationships, the bridge tables RentalAgreement and Advertisement are used. All corresponding tables contain a field that uniquely identifies each row in each table (primary keys): PersonID, BranchID, ScooterID, RentalAgreementID, InvoiceID, CreditCardID, AdvertisementID, AdvertisingAgencyID, and RentalPriceChangeID; no composite primary keys were used. Likewise, foreign keys were added to bridge tables, subtypes, and entities that are related to at most one other entity in a one-to-many relationship to enforce relationship constraints. And all tables have been normalized to Boyce-Codd Normal Form (BCNF) to avoid data redundancy. To speed up the search in the database, indexes were applied to all foreign keys. Likewise, all primary keys have indexes – which did not have to be defined, but are created automatically in SQL Server. Furthermore, three non-unique function-based indexes that indexes the result of a function on the column in the WHERE clauses were identified. And a history table RentalPriceChange was added, which records the hourly scooter rental price changes via a trigger, which triggers on an hourly rate change. The history table RentalPriceChange was visualized with a column chart presenting the daily average hourly rental price changes. And the revenues per branch and the number of rented scooters per branch was visualized with a scatter plot, and a trend line was plotted. Both visualizations serve as a basis for ScooterRent to understand what drives sales and the number of scooters rented, and what impact do hourly rental price changes have on profitability.