VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

INTRODUCTION TO DATABASE SYSTEMS

Collection of tasks including solutions

Petr Lukáš, Peter Chovanec, Radim Bača

November 10, 2020

Contents

Introduction

This document is created for practicing of SQL language. This document is categorized into the 5 categories, where each of them represents the topic of one practice from the subject Introduction to Database Systems. Each category contains approximately 30 tasks to solve. The first practice is dedicated to base usage of command SELECT, the second practice is focused on the joins of the tables, the third practice is focused on aggregation functions, the fourth practice is focused on set operations and the last practice is about complex queries containing subqueries. This document is published in two versions: version without solutions and version with solutions. Students work on the practice with the version without solutions. The version with solutions will be published after the practice.

Sincerely ask students to report any mistakes (unclear tasks, mistakes in solutions, unclear description of solution and others) to one of the following email addresses : `petr.lukas@vsb.cz`, `peter.chovanec@vsb.cz` or `radim.baca@vsb.cz`. Your help can improve the practices in next academic years.

Sakila Database

We use database of artificial movie rental called Sakila for the practices of subject Introduction to Database Systems. The database is originally designed for demonstration of SQL queries in database system MySQL[1]. In the last years, the versions for another database systems[2] have been published, e.g. Microsoft SQL Server. In our case, the scripts for the Microsoft SQL Server will be used. These scripts are published on the website of the subject `dbedu.cs.vsb.cz`. The data in the database have been slightly modified for the better demonstration of some SQL possibilities, i.e. some data have to be added or modified to get satisfying results of some SQL queries.

Relational Data Model

The structure of relation database is visualised by so-called E-R (Entity-Relationship) diagram. The E-R diagram of database Sakila is presented in Figure 1. We recommend students to print out the Figure, because we will work with it very often.

In Figure, we can see table `film` containing a list of all movies and table `actor` containing a list of all actors. These tables are joined by association table `film_actor`, therefore we have information which actor acts in each movie. There is a relationship M:N between film and actor, that means one actor can act in many movie and one movie can be acted by many actors. Similar situation is presented in the case of table `category` containing a list of all movie categories; it is joined with the table `film` by association table `film_category`. Therefore, one movie can be marked by more categories (horror, comedy, etc.) and vice versa. Moreover, there are two relationships N:1 between tables `film` and table `language` containing a list of all languages. The first relationship describes the real language of the movie, the second relationship describes the original language of the movie (in the case that movie has been dubbed).

We can continue with the description of table `inventory` containing a list of all movie copies. There is a relationship 1:N between tables `film` and `inventory`, it means, the movie rental can own one movie in many copies. Table `rental` contains list of all movie rents. Each rent is associated to some specified movie copy, to some specified customer in table `customer` and it is processed by some specified employee in table `staff`. Therefore, there are relationships N:1 between table `rental` and tables `inventory`, `customer` and `staff`. Table `payment` contains a list of all payments for the rents. Each payment is done by some customer in table `customer` and processed by some employee in table `staff`. Let us note, that not all payments represent payments for the movie rents. Some of them represents e.g. payments for subscription.

The database contains also tables `country`, `city` and `address` which are joined by relationships 1:N, i.e. one country has many cities and one city has many addresses. Table `address` has relationship 1:1 to tables `customer`, `store` and `staff`, i.e. each customer/store/employee can have only one address.

[1] `https://dev.mysql.com/doc/sakila/en/`

[2] `https://github.com/jOOQ/jOOQ/tree/master/jOOQ-examples/Sakila`

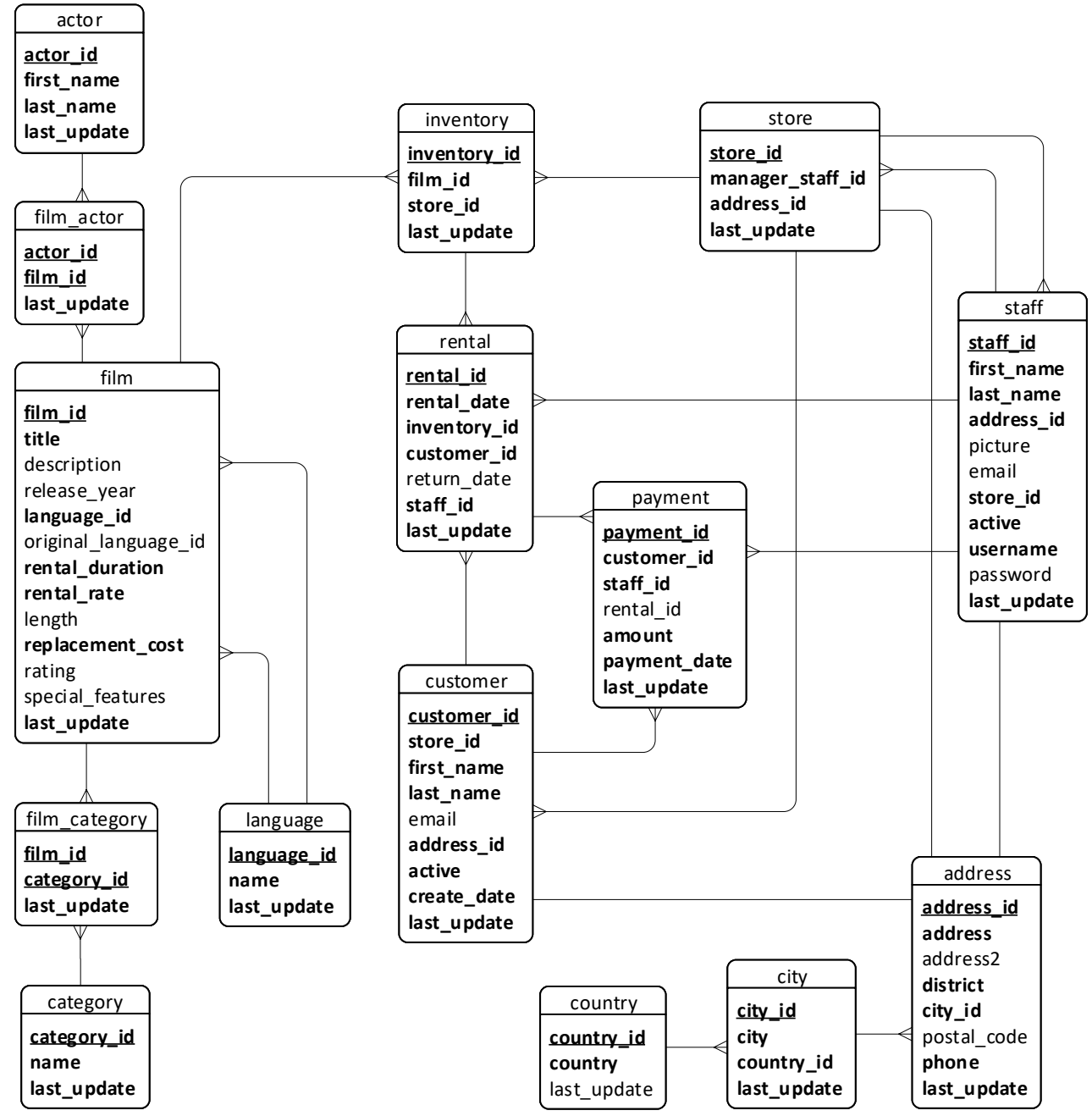


Figure 1: E-R diagram of database Sakila

Data Dictionary

Although, the name of the tables and attributes in database Sakila are mostly self-describing, we present their detail description in the form of data dictionary.

NULL an information whether the column is optional or not `NULL`
PK an information whether the column is a primary key
FK an information whether the column is a foreign key

RENTAL

the rental table contains one row for each rental of each inventory item with information about who rented what item, when it was rented, and when it was returned

column	data type	NULL	PK	FK	description
rental_id	integer number	no	yes	no	a surrogate primary key
rental_date	date and time	no	no	no	the date and time that the item was rented
inventory_id	integer number	no	no	yes	the item being rented
customer_id	integer number	no	no	yes	the customer renting the item
return_date	date and time	yes	no	no	the date and time the item was returned
staff_id	integer number	no	no	yes	the staff member who processed the rental
last_update	date and time	no	no	no	the time that the row was created or most recently updated

ACTOR

the actor table lists information for all actors

column	data type	NULL	PK	FK	description
actor_id	integer number	no	yes	no	a surrogate primary key
first_name	string, max. 45 chars.	no	no	no	the actor's first name
last_name	string, max. 45 chars.	no	no	no	the actor's last name
last_update	date and time	no	no	no	the time that the row was created or most recently updated

COUNTRY

the country table contains a list of countries

column	data type	NULL	PK	FK	description
country_id	integer number	no	yes	no	a surrogate primary key
country	string, max. 50 chars.	no	no	no	the name of the country
last_update	date and time	yes	no	no	the time that the row was created or most recently updated

CITY

the city table contains a list of cities

column	data type	NULL	PK	FK	description
city_id	integer number	no	yes	no	a surrogate primary key
city	string, max. 50 chars.	no	no	no	the name of the city
country_id	integer number	no	no	yes	a foreign key identifying the country that the city belongs to
last_update	date and time	no	no	no	the time that the row was created or most recently updated

ADDRESS

the address table contains address information for customers, staff, and stores

column	data type	NULL	PK	FK	description
address_id	integer number	no	yes	no	a surrogate primary key
address	string, max. 50 chars.	no	no	no	the first line of an address
address2	string, max. 50 chars.	yes	no	no	an optional second line of an address
district	string, max. 20 chars.	no	no	no	the region of an address, this may be a state, province, prefecture, etc.
city_id	integer number	no	no	yes	a foreign key pointing to the city table
postal_code	string, max. 10 chars.	yes	no	no	the postal code or ZIP code of the address (where applicable)
phone	string, max. 20 chars.	no	no	no	the telephone number for the address
last_update	date and time	no	no	no	the time that the row was created or most recently updated

LANGUAGE

the language table is a lookup table listing the possible languages that films can have for their language and original language values

column	data type	NULL	PK	FK	description
language_id	integer number	no	yes	no	a surrogate primary key
name	string, max. 20 chars.	no	no	no	the English name of the language
last_update	date and time	no	no	no	the time that the row was created or most recently updated

CATEGORY

the category table lists the categories that can be assigned to a film

column	data type	NULL	PK	FK	description
category_id	integer number	no	yes	no	a surrogate primary key
name	string, max. 25 chars.	no	no	no	the name of the category
last_update	date and time	no	no	no	the time that the row was created or most recently updated

CUSTOMER

the customer table contains a list of all customers

column	data type	NULL	PK	FK	description
customer_id	integer number	no	yes	no	a surrogate primary key
store_id	integer number	no	no	yes	a foreign key identifying the customer's home store
first_name	string, max. 45 chars.	no	no	no	the customer's first name
last_name	string, max. 45 chars.	no	no	no	the customer's last name
email	string, max. 50 chars.	yes	no	no	the customer's email address
address_id	integer number	no	no	yes	a foreign key identifying the customer's address in the address table
active	string, max. 1 chars.	no	no	no	whether the customer is an active customer
create_date	date and time	no	no	no	the date the customer was added to the system
last_update	date and time	no	no	no	the time that the row was created or most recently updated

FILM

the film table is a list of all films potentially in stock in the stores

column	data type	NULL	PK	FK	description
film_id	integer number	no	yes	no	a surrogate primary key
title	string, max. 255 chars.	no	no	no	the title of the film
description	text	yes	no	no	a short description or plot summary of the film
release_year	string, max. 4 chars.	yes	no	no	the year in which the movie was released
language_id	integer number	no	no	yes	a foreign key pointing at the language table; identifies the language of the film
original_language_id	integer number	yes	no	yes	a foreign key pointing at the language table; identifies the original language of the film
rental_duration	integer number	no	no	no	the length of the rental period, in days
rental_rate	decimal number	no	no	no	the cost to rent the film for the period specified in the rental_duration column
length	integer number	yes	no	no	the duration of the film, in minutes
replacement_cost	decimal number	no	no	no	the amount charged to the customer if the film is not returned or is returned in a damaged state
rating	string, max. 10 chars.	yes	no	no	the MPAA rating assigned to the film
special_features	string, max. 255 chars.	yes	no	no	lists which common special features are included on the DVD
last_update	date and time	no	no	no	the time that the row was created or most recently updated

FILM_ACTOR

the film_actor table is used to support a many-to-many relationship between films and actors

column	data type	NULL	PK	FK	description
actor_id	integer number	no	yes	yes	the film_actor table is used to support a many-to-many relationship between films and actors
film_id	integer number	no	yes	yes	the film_actor table is used to support a many-to-many relationship between films and actors
last_update	date and time	no	no	no	the film_actor table is used to support a many-to-many relationship between films and actors

FILM_CATEGORY

the film_category table is used to support a many-to-many relationship between films and categories

column	data type	NULL	PK	FK	description
film_id	integer number	no	yes	yes	the film_category table is used to support a many-to-many relationship between films and categories
category_id	integer number	no	yes	yes	the film_category table is used to support a many-to-many relationship between films and categories
last_update	date and time	no	no	no	the film_category table is used to support a many-to-many relationship between films and categories

INVENTORY

the inventory table contains one row for each copy of a given film in a given store

column	data type	NULL	PK	FK	description
inventory_id	integer number	no	yes	no	a surrogate primary key
film_id	integer number	no	no	yes	a foreign key pointing to the film this item represents
store_id	integer number	no	no	yes	a foreign key pointing to the store stocking this item
last_update	date and time	no	no	no	the time that the row was created or most recently updated

STAFF

the staff table lists all staff members, including information on email address, login information, and picture

column	data type	NULL	PK	FK	description
staff_id	integer number	no	yes	no	a surrogate primary key
first_name	string, max. 45 chars.	no	no	no	the first name of the staff member
last_name	string, max. 45 chars.	no	no	no	the last name of the staff member
address_id	integer number	no	no	yes	a foreign key to the staff member's address in the address table
picture	image	yes	no	no	a BLOB containing a photograph of the employee
email	string, max. 50 chars.	yes	no	no	the staff member's email address
store_id	integer number	no	no	yes	the staff member's home store
active	bit	no	no	no	whether this is an active employee
username	string, max. 16 chars.	no	no	no	the user name used by the staff member to access the rental system
password	string, max. 40 chars.	yes	no	no	the SHA1 hashed password used by the staff member to access the rental system
last_update	date and time	no	no	no	the time that the row was created or most recently updated

STORE

the store table lists all stores in the system

column	data type	NULL	PK	FK	description
store_id	integer number	no	yes	no	a surrogate primary key
manager_staff_id	integer number	no	no	yes	a foreign key identifying the manager of this store
address_id	integer number	no	no	yes	a foreign key identifying the address of this store
last_update	date and time	no	no	no	the time that the row was created or most recently updated

PAYMENT

the payment table records each payment made by a customer, with information such as the amount and the rental being paid for (when applicable)

column	data type	NULL	PK	FK	description
payment_id	integer number	no	yes	no	a surrogate primary key
customer_id	integer number	no	no	yes	the customer whose balance the payment is being applied to
staff_id	integer number	no	no	yes	the staff member who processed the payment
rental_id	integer number	yes	no	yes	the rental that the payment is being applied to
amount	decimal number	no	no	no	the amount of the payment
payment_date	date and time	no	no	no	the date the payment was processed
last_update	date and time	no	no	no	the time that the row was created or most recently updated

1 SQL Basics, command SELECT

This practice will be about base syntax of the command SELECT. All queries will be processed over one table. Queries will be oriented on simple selection, projection, conditions, base date/-time/text functions and so-called aggregation functions.

1. Select email adresses of all inactive customers.

```
SELECT email
FROM customer
WHERE active = 0
```

2. Select names and description of all movies with classification G (attribute `rating`). The result has to be ordered by the name of movie.

```
SELECT title, description
FROM film
WHERE rating = 'G'
ORDER BY title DESC
```

3. Select all information about payments since the year 2006 and payments with amount lower than 2.

```
SELECT *
FROM payment
WHERE payment_date >= '2006-01-01' AND amount < 2
```

4. Select all movies classified as G or PG.

```
SELECT description
FROM film
WHERE rating = 'G' OR rating = 'PG'
```

5. Select all movies classified as G, PG or PG-13.

```
SELECT description
FROM film
WHERE rating IN ('G', 'PG', 'PG-13')
```

6. Select description of all movies not classified as G, PG and PG-13.

```
SELECT description
FROM film
WHERE rating NOT IN ('G', 'PG', 'PG-13')
```

7. Select all information about movies longer that 50 minutes that have rental duration 3 or 5 days.

```
SELECT *
FROM film
WHERE length > 50 AND (rental_duration = 3 OR rental_duration = 5)
```

8. Select names of all movies longer than 70 minutes and names containing word 'RAINBOW' or beginning on word 'TEXAS'.

```
SELECT title
FROM film
WHERE
  (title LIKE '%RAINBOW%' OR title LIKE 'TEXAS%')
  AND length > 70
```

9. Select names of all movies which description contains word, their length is between 80 and 90 minutes and standard rental duration is odd number.

```
SELECT title
FROM film
WHERE
  description LIKE '%And%' AND length BETWEEN 80 AND 90
  AND rental_duration % 2 = 1
```

10. Select features (attribute `special_features`) of all movies where cost of replacement is between 14 and 16. Ensure that each feature occurs only once in the result and order the features alphabetically. Why is the result automatically ordered even if the ORDER BY is not used?

```
SELECT DISTINCT special_features
FROM film
WHERE replacement_cost BETWEEN 14 AND 16
ORDER BY special_features
```

11. Select all information about movies with standard rental duration lower than 4 days or classified as PG. The result can not contain movies satisfying both condition.

```
SELECT title
FROM film
WHERE
  rental_duration < 4 AND rating != 'PG' OR
  rental_duration >= 4 AND rating = 'PG'
```

12. Select all information about addresses with filled postal code.

```
SELECT *
FROM address
WHERE postal_code IS NOT NULL
```

13. Select IDs of all customers with some currently rented movie. Do you know how to count those customers?

```
SELECT DISTINCT customer_id
FROM rental
WHERE return_date IS NULL
```

14. Select year, month and day in separate columns of each payment in the database. Name the columns as pay_year, pay_month and pay_day.

```
SELECT payment_id, YEAR(payment_date) AS pay_year, MONTH(payment_date) AS
    pay_month, DAY(payment_date) AS pay_day
FROM payment
```

15. Select movies with the length of their name not equal to 20 characters.

```
SELECT *
FROM film
WHERE LEN(title) != 20
```

16. Select duration (in minutes) of each rent in the database. Name this column as duration [min.].

```
SELECT rental_id, DATEDIFF(minute, rental_date, return_date) AS duration[
    min.]
FROM rental
```

17. Select full name in one column for each active customer. Result has to contain two columns – `customer_id` and `full_name`.

```
SELECT customer_id, first_name + ' ' + last_name AS full_name
FROM customer
WHERE active = 1
```

18. Select zip code for each address in the database. In the case of null zip code print out text '(empty)'.

```
SELECT address, COALESCE(postal_code, '(empty)') AS psc
FROM address
```

19. Select interval from – to (it means both dates in one column) for all closed rents (closed rent has filled return date).

```
SELECT rental_id, CAST(rental_date AS VARCHAR) + ' - ' + CAST(return_date
    AS VARCHAR) AS interval
FROM rental
WHERE return_date IS NOT NULL
```

20. Select interval from – to (it means both dates in one column) for all rents. If the rent is not closed yet, print only date of rent.

```
SELECT rental_id, CAST(rental_date AS VARCHAR) + COALESCE(' - ' + CAST(
    return_date AS VARCHAR), '') AS interval
FROM rental
```

21. Select number of all movies in the database.

```
SELECT COUNT(*) AS pocet_filmu
FROM film
```

22. Select number of various movie classification (attribute `rating`).

```
SELECT COUNT(DISTINCT rating) AS pocet_kategorii
FROM film
```

23. Select number of all addresses, number of addresses with filled zip code and number of various zip codes using one query.

```
SELECT
  COUNT(*) AS pocet_celkem,
  COUNT(postal_code) AS pocet_s_psc,
  COUNT(DISTINCT postal_code) AS pocet_psc
FROM address
```

24. Select minimal, maximal and average length of all movies. Check if the average length is equal to ratio of summary length of all movies and total number of movies in the database.

```
SELECT MIN(length) AS nejmensi, MAX(length) AS nejvetsi, AVG(CAST(length
    AS FLOAT)) AS prumerna
FROM film
```

25. Select number and sum of all payments of the year 2005.

```
SELECT COUNT(*) AS pocet, SUM(amount) AS soucet
FROM payment
WHERE YEAR(payment_date) = 2005
```

26. Select total number of characters in names of all movies.

```
SELECT SUM(LEN(title))
FROM film
```

2 Table Joins

The first practice has been focused on queries over one table. However, more tables are usually needed in query to get a required result. In this practice, we show how to join tables in queries. We focus on inner joins and left outer joins. All tasks of this practice have to be solved without aggregation function, subqueries and constructions `IN/EXISTS`. All tasks have to be solved only by adequate join of several tables and restriction of redundant data in a result by code word `DISTINCT`.

1. Select all information about cities including information about the countries, where are the cities located.

```
SELECT *
FROM city JOIN country ON city.country_id = country.country_id
```

2. Select names of all movies including the names of their language.

```
SELECT film.title, language.name
FROM film JOIN language ON film.language_id = language.language_id
```

3. Select IDs of all rents of customer with surname SIMPSON.

```
SELECT rental_id
FROM rental JOIN customer ON
  rental.customer_id = customer.customer_id
WHERE customer.last_name = 'SIMPSON'
```

4. Select address (attribute `address` in table `address`) of customer with surname SIMPSON. Compare the number of records in the result with the previous task.

```
SELECT address
FROM customer JOIN address ON
  customer.address_id = address.address_id
WHERE customer.last_name = 'SIMPSON'
```

5. Select name and surname of all customers including their addresses, zip codes and cities.

```
SELECT first_name, last_name, address, postal_code, city
FROM
  customer
  JOIN address ON customer.address_id = address.address_id
  JOIN city ON address.city_id = city.city_id
```

6. Select name and surname of all customers including their cities.

```
SELECT first_name, last_name, city
FROM
  customer
  JOIN address ON customer.address_id = address.address_id
  JOIN city ON address.city_id = city.city_id
```

7. Select IDs of all rents including name of the staff, name of the customer and title of the movie.

```
SELECT rental_id, staff.first_name AS staff_first_name,
  staff.last_name AS staff_last_name,
  customer.first_name AS customer_first_name,
  customer.last_name AS customer_last_name,
  film.title
FROM
  rental
  JOIN staff ON rental.staff_id = staff.staff_id
  JOIN customer ON rental.customer_id = customer.customer_id
  JOIN inventory ON rental.inventory_id = inventory.inventory_id
  JOIN film ON inventory.film_id = film.film_id
```

8. Select all movies (their titles) together with the actors playing in them (their names and surnames). How many records will be in the result of this query?

```
SELECT film.title, actor.first_name, actor.last_name
FROM
  film
  JOIN film_actor ON film.film_id = film_actor.film_id
  JOIN actor ON film_actor.actor_id = actor.actor_id
ORDER BY film.title
```

9. Select all actors (their names and surnames) together with their movies. What is the difference in comparison with previous query? What we can say about inner joins?

```
SELECT actor.first_name, actor.last_name, film.title
FROM
  film
  JOIN film_actor ON film.film_id = film_actor.film_id
  JOIN actor ON film_actor.actor_id = actor.actor_id
ORDER BY actor.last_name, actor.first_name
```

10. Select titles of all movies in the category 'Horror'.

```
SELECT film.title
FROM
  category
  JOIN film_category ON
    category.category_id = film_category.category_id
  JOIN film ON film_category.film_id = film.film_id
WHERE name = 'Horror'
```

11. Select all stores (their IDs) together with their managers (their names and surnames). Moreover, select addresses of stores and addresses of managers (attribute `address` in table `address`). As a last step, append the cities and countries of stores and managers to the result.

```
SELECT store.store_id, store_address.address AS store_address, store_city.
    city AS store_city, store_country.country AS store_country, staff.
    first_name, staff.last_name, staff_address.address AS staff_address,
    staff_city.city AS staff_city, staff_country.country AS staff_country
FROM
  store
  JOIN staff ON
```

```
    store.manager_staff_id = staff.staff_id
  JOIN address store_address ON
    store.address_id = store_address.address_id
  JOIN city store_city ON
    store_address.city_id = store_city.city_id
  JOIN country store_country ON
    store_city.country_id = store_country.country_id
  JOIN address staff_address ON
    staff.address_id = staff_address.address_id
  JOIN city staff_city ON
    staff_address.city_id = staff_city.city_id
  JOIN country staff_country ON
    staff_city.country_id = staff_country.country_id
```

12. Select all movies (their IDs and titles) together with IDs of actors playing in them and IDs of categories belonging to. It means, a result of the query has to contain attributes `film_id`, `actor_id` and `category_id` and it has to be order by `film_id`.

```
SELECT film.film_id, film.title, actor_id, category_id
FROM
  film
  JOIN film_actor ON film_actor.film_id = film.film_id
  JOIN film_category ON film_category.film_id = film.film_id
ORDER BY film.film_id
```

13. Select all combinations of actors and categories (their IDs) where specified actors played in a movie of specified category. Result order by ID of actor. Consequently, extend result by the names and surnames of actors and by the names of categories.

```
SELECT DISTINCT actor.actor_id, actor.first_name, actor.last_name,
   category.category_id, category.name
FROM
  film
  JOIN film_actor ON film_actor.film_id = film.film_id
  JOIN film_category ON film_category.film_id = film.film_id
  JOIN actor ON film_actor.actor_id = actor.actor_id
  JOIN category ON film_category.category_id = category.category_id
ORDER BY actor.actor_id
```

14. Select names of movies that rental owns in at least one copy.

```
SELECT DISTINCT film.title
FROM film JOIN inventory ON film.film_id = inventory.film_id
```

15. Select the actors playing in at least one comedy (category 'Comedy').

```
SELECT DISTINCT actor.actor_id, actor.first_name, actor.last_name
FROM
  film
  JOIN film_actor ON film_actor.film_id = film.film_id
  JOIN actor ON film_actor.actor_id = actor.actor_id
  JOIN film_category ON film_category.film_id = film.film_id
  JOIN category ON film_category.category_id = category.category_id
WHERE category.name = 'Comedy'
```

16. Select names of customers from Italy that borrowed movie with title MOTIONS DETAILS.

```
SELECT DISTINCT customer.first_name, customer.last_name
FROM
  customer
  JOIN address ON customer.address_id = address.address_id
  JOIN city ON address.city_id = city.city_id
  JOIN country ON city.country_id = country.country_id
  JOIN rental ON customer.customer_id = rental.customer_id
  JOIN inventory ON rental.inventory_id = inventory.inventory_id
  JOIN film ON inventory.film_id = film.film_id
WHERE country.country = 'Italy' AND film.title = 'MOTIONS DETAILS'
```

17. Select names of customers with the currently borrowed movie with actor SEAN GUINESS.

```
SELECT DISTINCT customer.first_name, customer.last_name
FROM
  actor
  JOIN film_actor ON actor.actor_id = film_actor.actor_id
  JOIN film ON film_actor.film_id = film.film_id
  JOIN inventory ON film.film_id = inventory.film_id
  JOIN rental ON inventory.inventory_id = rental.inventory_id
  JOIN customer ON rental.customer_id = customer.customer_id
WHERE actor.first_name = 'SEAN' AND actor.last_name = 'GUINESS' AND rental
   .return_date IS NULL
```

18. Select IDs and amounts of all payments together with the date of rental (attribute `rental_date` in table `rental`). In the case of payments not linked to any rent, print empty date of rental (`NULL`).

```
SELECT payment.payment_id, payment.amount, rental.rental_date
FROM
  payment
  LEFT JOIN rental ON payment.rental_id = rental.rental_id
```

19. Select all languages together with the list of all movies catched in the specified language for each of them. Ensure that all languages are in the result.

```
SELECT language.name, film.title
FROM
  language
  LEFT JOIN film ON language.language_id = film.language_id
```

20. Select all movies (their IDs and titles) together with their languages and original languages.

```
SELECT film.film_id, film.title, language.name AS language,
   original_language.name AS original_language
FROM
  film
  JOIN language ON film.language_id = language.language_id
  LEFT JOIN language original_language ON film.original_language_id =
     original_language.language_id
```

21. Select names of movies borrowed by customer TIM CARY and names of movies 48 minutes long.

```
SELECT DISTINCT film.title
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  LEFT JOIN customer ON customer.customer_id = rental.customer_id
WHERE (customer.first_name = 'TIM' AND customer.last_name = 'CARY') OR
   film.length = 48
```

22. Select names of movies that rental does not own (it means they are not in table `inventory`).

```
SELECT film.title, length
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
WHERE inventory.inventory_id IS NULL
```

23. Select name of customer that did not pay for some rent.

```
SELECT DISTINCT first_name, last_name
FROM
  customer
  JOIN rental ON customer.customer_id = rental.customer_id
  LEFT JOIN payment ON rental.rental_id = payment.rental_id
WHERE payment.payment_id IS NULL
```

24. Select all movies together with the name of language. The language has to be in result only, if it starts with letter 'I', otherwise print out value `NULL`.

```
SELECT film.title, language.name
FROM
  film
  LEFT JOIN language ON film.language_id = language.language_id AND
     language.name LIKE 'I%'
```

25. Select all customers together with IDs of their payments higher than 9. In the case of customers without such payment, print out value `NULL`.

```
SELECT first_name, last_name, payment.payment_id
FROM
  customer
  LEFT JOIN payment ON customer.customer_id = payment.customer_id AND
     payment.amount > 9
```

26. Select all rents (their IDs) together with the titles of movies (but only if they contain letter 'U') and with cities and countries of customer (but only if customer address contains letter 'A'). If the value does not satisfy the condition, print out value `NULL`.

```
SELECT rental_id, film.title, city.city, country.country
FROM
  rental
  LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
```

```
  LEFT JOIN film ON inventory.film_id = film.film_id AND film.title LIKE '
     %U%'
  LEFT JOIN customer ON rental.customer_id = customer.customer_id
  LEFT JOIN address ON customer.address_id = address.address_id AND
     address.address LIKE '%A%'
  LEFT JOIN city ON address.city_id = city.city_id
  LEFT JOIN country ON city.country_id = country.country_id
```

27. Select all pairs movie title - customer surname where specified customer borrowed specified movie. In the case of rents after 01.01.2006, the customer surnamen has to be empty (it means `NULL`). Ensure that result do not contain redundant data.

```
SELECT DISTINCT film.title, customer.last_name
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id AND
     rental_date > '2006-01-01'
  LEFT JOIN customer ON rental.customer_id = customer.customer_id
ORDER BY film.title
```

3 Aggregate Functions and Group By

We already met with aggregate functions on the first practice, where we used them to get one row containing one or more calculated values. This practice will show use that aggregate functions can be used not only for complete data table but also for some groups of records in them. Consequently, the result will not be only one row, but more rows grouping records on the basis of some conditions. At the beginning, we will start with aggregate functions over one table, and then we will use your experiences from previous practice and we will use query over more tables.

1. Select the number of movies of particular classifications (attribute `rating`).

```
SELECT rating, COUNT(*) AS count
FROM film
GROUP BY rating
```

2. Select the number of surnames for particular customers (their IDs).

```
SELECT customer_id, COUNT(last_name) AS count
FROM customer
GROUP BY customer_id
```

3. Select customer IDs ordered by the total amount of their payments. Customers without any payment will not be in the result.

```
SELECT customer_id
FROM payment
GROUP BY customer_id
ORDER BY SUM(amount)
```

4. Select number of actors with the specified name and surname of each actors name and surname. The result must be ordered by the number descendingly.

```
SELECT first_name, last_name, COUNT(*) AS count
FROM actor
GROUP BY first_name, last_name
ORDER BY pocet DESC
```

5. Select total amount of all payments for particular years and months. The result must be ordered by years and months.

```
SELECT YEAR(payment_date) AS payment_year, MONTH(payment_date) AS
    payment_month, SUM(amount) AS count
FROM payment
GROUP BY YEAR(payment_date), MONTH(payment_date)
ORDER BY payment_year, payment_month
```

6. Select stores (their IDs) with more than 2 300 movie copies.

```
SELECT store_id, COUNT(*)
FROM inventory
GROUP BY store_id
HAVING COUNT(*) > 2300
```

7. Select the shortest movie per language ID and select only those language IDs where the shortest movie is longer than 46 minutes.

```
SELECT language_id
FROM film
GROUP BY language_id
HAVING MIN(length) > 46
```

8. Select years and months when total amount of payments was higher than 20 000.

```
SELECT
  YEAR(payment_date) AS payment_year, MONTH(payment_date) AS payment_month
     ,
  SUM(amount) AS summary
FROM payment
GROUP BY YEAR(payment_date), MONTH(payment_date)
HAVING SUM(amount) > 20000
```

9. Let us consider just movies shorter than 50 minutes. We are interested in the total length per the rating, and we want only those ratings where the total length is higher than 250 minutes. The result must be ordered alphabetically.

```
SELECT rating
FROM film
WHERE length < 50
GROUP BY rating
HAVING SUM(length) > 250
ORDER BY rating DESC
```

10. Select the number of movies per language ID. The result *will not* contain languages without a movie.

```
SELECT language_id, COUNT(*) AS movies_count
FROM film
GROUP BY language_id
```

11. Select the number of movies per language name. The result *will not* contain languages without any movie.

```
SELECT
  language.language_id, language.name, COUNT(*) AS movies_count
FROM
  language
  JOIN film ON language.language_id = film.language_id
GROUP BY language.language_id, language.name
```

12. Select the number of movies per language name. The result *will* contain languages without any movie.

```
SELECT language.language_id, language.name, COUNT(film.film_id) AS
    movies_count
FROM
  language
  LEFT JOIN film ON language.language_id = film.language_id
GROUP BY language.language_id, language.name
```

13. Select number of rentals per customer (print out his ID, first name and surname).

```
SELECT
  customer.customer_id, first_name, last_name,
  COUNT(rental.rental_id) AS rentals_count
FROM
  customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
GROUP BY customer.customer_id, first_name, last_name
```

14. Select all customers (their IDs, first names and surnames) and how many *different* movies they rented.

```
SELECT customer.customer_id, first_name, last_name, COUNT(DISTINCT
   inventory.film_id) AS pocet_filmu
FROM
  customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
  LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
GROUP BY customer.customer_id, first_name, last_name
```

15. Select names and surnames of actors acting in more than 20 movies.

```
SELECT actor.first_name, actor.last_name
FROM
  actor
  JOIN film_actor ON actor.actor_id = film_actor.actor_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name
HAVING COUNT(film_actor.film_id) > 20
```

16. Select all customers together with the informations: how much money they paid for rentals in total, how much money they paid for one rental maximally, minimally and in average.

```
SELECT
  customer.customer_id, first_name, last_name,
  SUM(payment.amount) AS total, MIN(payment.amount) AS minimal,
  MAX(payment.amount) AS maximal, AVG(payment.amount) AS average
FROM
  customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
  LEFT JOIN payment ON rental.rental_id = payment.rental_id
GROUP BY customer.customer_id, first_name, last_name
```

17. Select average length of movie per movie category. Include *all* categories!

```
SELECT category.category_id, category.name,
  AVG(CAST(film.length AS FLOAT)) AS average
FROM
  category
  LEFT JOIN film_category ON category.category_id = film_category.
     category_id
  LEFT JOIN film ON film_category.film_id = film.film_id
GROUP BY category.category_id, category.name
```

18. Select how much customers spent for rentals of particular movies. Select only movies with the total rental amount higher than 100.

```
SELECT film.film_id, film.title, SUM(payment.amount) AS total
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  LEFT JOIN payment ON rental.rental_id = payment.rental_id
GROUP BY film.film_id, film.title
HAVING SUM(payment.amount) > 100
```

19. Select the number of *different* movie categories per actor. Select the actor ID, first name and last name.

```
SELECT
  actor.actor_id, actor.first_name, actor.last_name,
  COUNT(DISTINCT film_category.category_id) AS categories_count
FROM
  actor
  LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
  LEFT JOIN film_category ON film_actor.film_id = film_category.film_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name
```

20. Select addresses, cities and countries of customers which borrowed movies with together at least 40 different actors.

```
SELECT address.address, city.city, country.country
FROM
  customer
  JOIN address ON customer.address_id = address.address_id
  JOIN city ON address.city_id = city.city_id
  JOIN country ON city.country_id = country.country_id
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
  LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
  LEFT JOIN film_actor ON inventory.film_id = film_actor.film_id
GROUP BY address.address, city.city, country.country
HAVING COUNT(DISTINCT film_actor.actor_id) >= 40
```

21. Select ID and title of all movies with category 'Horror' together with the number of different cities of customers that borrowed them.

```
SELECT
  film.film_id, film.title, COUNT(DISTINCT address.city_id) AS
     cities_count
FROM
  film
  JOIN film_category ON film.film_id = film_category.film_id
  JOIN category ON film_category.category_id = category.category_id
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  LEFT JOIN customer ON rental.customer_id = customer.customer_id
  LEFT JOIN address ON customer.address_id = address.address_id
WHERE category.name = 'Horror'
GROUP BY film.film_id, film.title
```

22. Select all customers from Poland together with the number of different categories of the movies that they borrowed.

```
SELECT customer.customer_id, customer.first_name, customer.last_name,
    COUNT(DISTINCT film_category.category_id) AS pocet_kategorii
FROM
  country
  JOIN city ON country.country_id = city.country_id
  JOIN address ON city.city_id = address.city_id
  JOIN customer ON address.address_id = customer.address_id
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
  LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
  LEFT JOIN film ON inventory.film_id = film.film_id
  LEFT JOIN film_category ON film.film_id = film_category.film_id
WHERE country.country = 'Poland'
GROUP BY customer.customer_id, customer.first_name, customer.last_name
```

23. Select names of all languages together with the number of movies longer than 350 minutes catched in those languages.

```
SELECT language.name, COUNT(film.film_id) AS pocet
FROM
  language
  LEFT JOIN film ON language.language_id = film.language_id
    AND film.length > 350
GROUP BY language.name
```

24. Select all customers together with information how much they paid for rentals started in june.

```
SELECT
  customer.customer_id, first_name, last_name,
  COALESCE(SUM(payment.amount), 0) AS celkem
FROM
  customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
    AND MONTH(rental.rental_date) = 6
  LEFT JOIN payment ON rental.rental_id = payment.rental_id
GROUP BY customer.customer_id, first_name, last_name
```

25. Select names of all categories ordered by the number of movies catched in language starting with letter 'E'.

```
SELECT
  category.name
FROM
  category
  LEFT JOIN film_category ON category.category_id = film_category.
     category_id
  LEFT JOIN film ON film_category.film_id = film.film_id
  LEFT JOIN language ON film.language_id = language.language_id AND
     language.name LIKE 'E%'
GROUP BY category.name
ORDER BY COUNT(language.language_id)
```

26. Select titles of movies shorter than 50 minutes which customers with surname BELL borrowed exactly 1x.

```
SELECT film.film_id, film.title, customer.last_name, COUNT(customer.
    customer_id)
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  LEFT JOIN customer ON rental.customer_id = customer.customer_id AND
     customer.last_name = 'BELL'
WHERE film.length < 50
GROUP BY film.film_id, film.title, customer.last_name
HAVING COUNT(customer.customer_id) = 1
```

```
SELECT film.film_id, film.title, customer.last_name, COUNT(customer.
    customer_id)
FROM
  film
  JOIN inventory ON film.film_id = inventory.film_id
  JOIN rental ON inventory.inventory_id = rental.inventory_id
  JOIN customer ON rental.customer_id = customer.customer_id
WHERE film.length < 50 AND customer.last_name = 'BELL'
GROUP BY film.film_id, film.title, customer.last_name
HAVING COUNT(customer.customer_id) = 1
```

4 Set Operations and Quantifiers

Many tasks is possible to solve without so-called subqueries; it means clause `SELECT` is included in the query exactly once. This practice is focused on the constructions `IN`, `EXISTS`, `ANY` and `ALL` that require an application of subqueries. Although many of the following tasks is possible to solve also by aggregate functions, use mentioned constructions instead. All tasks in the practice is possible to solve without aggregate functions and data grouping. In the real world (and also on the SQL test) it will be up to you, if you will choose aggregate functions or subqueries to solve the tasks.

1. Select IDs and titles of the movies of actor with ID = 1. The query has to be solved without `JOIN`.

```
SELECT film_id, title
FROM film
WHERE film_id IN (SELECT film_id FROM film_actor WHERE actor_id = 1)
```

   OR

```
SELECT film_id, title
FROM film
WHERE EXISTS (SELECT * FROM film_actor WHERE film.film_id = film_actor.
    film_id AND actor_id = 1)
```

2. Select IDs of the movies of actor with ID = 1.

```
SELECT film_id
FROM film
WHERE film_id IN (SELECT film_id FROM film_actor WHERE actor_id = 1)
```

   OR SIMPLER SOLUTION:

```
SELECT film_id
FROM film_actor
WHERE actor_id = 1
```

3. Select IDs and titles of the movies in which plays actor with ID = 1 as well as actor with ID = 10.

```
SELECT film_id, title
FROM film
WHERE
  film_id IN (SELECT film_id FROM film_actor WHERE actor_id = 1) AND
  film_id IN (SELECT film_id FROM film_actor WHERE actor_id = 10)
```

   OR

```
SELECT film_id, title
FROM film
WHERE
  EXISTS (SELECT * FROM film_actor WHERE film.film_id = film_actor.film_id
      AND actor_id = 1) AND
  EXISTS (SELECT * FROM film_actor WHERE film.film_id = film_actor.film_id
      AND actor_id = 10)
```

4. Select IDs and titles of the movies in which plays actor with ID = 1 or actor with ID = 10.

```
SELECT film_id, title
FROM film
WHERE
   film_id IN (SELECT film_id FROM film_actor WHERE actor_id = 1) OR
   film_id IN (SELECT film_id FROM film_actor WHERE actor_id = 10)
```

OR SIMPLER:

```
SELECT film.film_id, title
FROM film
WHERE film_id IN (
   SELECT film_id
   FROM film_actor
   WHERE actor_id = 1 OR actor_id = 10
)
```

5. Select IDs of the movies in which did not play actor with ID = 1.

```
SELECT film_id
FROM film
WHERE film_id NOT IN (
   SELECT film_id
   FROM film_actor
   WHERE actor_id = 1
)
```

OR

```
SELECT film_id
FROM film
WHERE NOT EXISTS (
   SELECT film_id
   FROM film_actor
   WHERE film.film_id = film_actor.film_id AND actor_id = 1
)
```

6. Select IDs and titles of the movies in which plays actor with ID = 1 or actor with ID = 10, but not both together.

```
SELECT film_id, title
FROM film
WHERE
   film_id IN (
      SELECT film_id FROM film_actor
      WHERE actor_id = 1 OR actor_id = 10
   )
   AND NOT
   (
      film_id IN (
         SELECT film_id FROM film_actor WHERE actor_id = 1
      )
      AND
      film_id IN (
         SELECT film_id FROM film_actor WHERE actor_id = 10
      )
   )
```

7. Select IDs and titles of the movies in which plays actor PENELOPE GUINESS as well as actor CHRISTIAN GABLE.

```
SELECT film_id, title
FROM film
WHERE
  film_id IN (
    SELECT film_id
    FROM actor JOIN film_actor ON
      actor.actor_id = film_actor.actor_id
    WHERE
      actor.first_name = 'PENELOPE' AND
      actor.last_name = 'GUINESS'
  )
  AND film_id IN (
    SELECT film_id
    FROM actor JOIN film_actor
      ON actor.actor_id = film_actor.actor_id
    WHERE
      actor.first_name = 'CHRISTIAN' AND
      actor.last_name = 'GABLE'
  )
```

8. Select IDs and titles of the movies in which did not play actor PENELOPE GUINESS.

```
SELECT film_id, title
FROM film
WHERE
  film_id NOT IN (
    SELECT film_id
    FROM actor JOIN film_actor ON
      actor.actor_id = film_actor.actor_id
    WHERE actor.first_name = 'PENELOPE' AND actor.last_name = 'GUINESS'
  )
```

9. Select customers (their IDs and names) which borrowed all movies from the following list: ENEMY ODDS, POLLOCK DELIVERANCE a FALCON VOLUME.

```
SELECT customer.customer_id, customer.first_name, customer.last_name
FROM customer
WHERE
  customer_id IN
  (
    SELECT customer_id
    FROM
      rental
      JOIN inventory ON
        rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
    WHERE film.title = 'ENEMY ODDS'
  ) AND customer_id IN
  (
    SELECT customer_id
    FROM
      rental
      JOIN inventory ON
        rental.inventory_id = inventory.inventory_id
```

```
      JOIN film ON inventory.film_id = film.film_id
    WHERE film.title = 'POLLOCK DELIVERANCE'
  ) AND customer_id IN
  (
    SELECT customer_id
    FROM
      rental
      JOIN inventory ON
        rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
    WHERE film.title = 'FALCON VOLUME'
  )
```

10. Select customers (their IDs and names) which borrowed movie GRIT CLOCKWORK in May as well as in June (of arbitrary year).

```
SELECT first_name, last_name
FROM customer
WHERE
  customer_id IN (
    SELECT customer_id
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
    WHERE
      film.title = 'GRIT CLOCKWORK' AND
      MONTH(rental.rental_date) = 5
  ) AND customer_id IN (
    SELECT customer_id
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
    WHERE
      film.title = 'GRIT CLOCKWORK' AND
      MONTH(rental.rental_date) = 6
  );
```

11. Select names and surnames of the customers which have the same surname as some actor.

```
SELECT first_name, last_name
FROM customer
WHERE last_name IN (SELECT last_name FROM actor)
```

OR

```
SELECT first_name, last_name
FROM customer
WHERE EXISTS (
  SELECT *
  FROM actor
  WHERE actor.last_name = customer.last_name
)
```

12. Select titles of the movies with same length as another movies.

```
SELECT title
FROM film f1
WHERE EXISTS (
  SELECT *
  FROM film f2
  WHERE f1.length = f2.length AND f1.film_id != f2.film_id
)
```

OR

```
SELECT title
FROM film f1
WHERE length IN (
  SELECT length
  FROM film f2
  WHERE f1.film_id != f2.film_id
)
```

13. Select titles of the movies shorter than any movie of actor BURT POSEY.

```
SELECT title
FROM film
WHERE length < ANY (
  SELECT film.length
  FROM
    actor
    JOIN film_actor ON actor.actor_id = film_actor.actor_id
    JOIN film ON film_actor.film_id = film.film_id
  WHERE actor.first_name = 'BURT' AND actor.last_name = 'POSEY'
)
```

OR

```
SELECT title
FROM film f1
WHERE EXISTS
(
  SELECT *
  FROM
    actor
    JOIN film_actor ON actor.actor_id = film_actor.actor_id
    JOIN film f2 ON film_actor.film_id = f2.film_id
  WHERE actor.first_name = 'BURT' AND actor.last_name = 'POSEY' AND f1.
     length < f2.length
)
```

14. Select names of the actors playing in any movie shorter than 50 minutes.

```
SELECT actor.first_name, actor.last_name
FROM actor
WHERE 50 > ANY (
  SELECT length
  FROM film JOIN film_actor ON film.film_id = film_actor.film_id
  WHERE film_actor.actor_id = actor.actor_id
)
```

OR

```
SELECT actor.first_name, actor.last_name
FROM actor
WHERE EXISTS (
  SELECT *
  FROM film JOIN film_actor ON film.film_id = film_actor.film_id
  WHERE film_actor.actor_id = actor.actor_id AND film.length < 50
)
```

OR

```
SELECT DISTINCT first_name, last_name
FROM
  actor
  JOIN film_actor ON actor.actor_id = film_actor.actor_id
  JOIN film ON film_actor.film_id = film.film_id
WHERE film.length < 50
```

15. Select the movies rented at least twice.

```
SELECT DISTINCT f1.title
FROM
  rental r1
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN film f1 ON i1.film_id = f1.film_id
WHERE
  EXISTS (
    SELECT *
      FROM
        rental r2
        JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
      WHERE i2.film_id = i1.film_id AND r1.rental_id != r2.rental_id
  )
```

OR

```
SELECT film.title
FROM
  film
  JOIN inventory ON film.film_id = inventory.film_id
  JOIN rental ON inventory.inventory_id = rental.inventory_id
GROUP BY film.film_id, film.title
HAVING COUNT(rental.customer_id) > 1
```

16. Select the movies rented by at least two different customers.

```
SELECT DISTINCT f1.title
FROM
  rental r1
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN film f1 ON i1.film_id = f1.film_id
WHERE
  AND  EXISTS (
    SELECT *
    FROM
      rental r2
      JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
```

```
    WHERE i2.film_id = i1.film_id AND r1.customer_id != r2.customer_id
  )
```

OR

```
SELECT film.title
FROM
  film
  JOIN inventory ON film.film_id = inventory.film_id
  JOIN rental ON inventory.inventory_id = rental.inventory_id
GROUP BY film.film_id, film.title
HAVING COUNT(DISTINCT rental.customer_id) > 1
```

17. Select the customers which borrowed at least two different movies at once (at the same moment).

```
SELECT DISTINCT customer.customer_id, customer.first_name, customer.
    last_name
FROM
  rental r1
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN customer ON r1.customer_id = customer.customer_id
WHERE EXISTS (
  SELECT *
  FROM
    rental r2
    JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
  WHERE
    r1.customer_id = r2.customer_id AND
    i1.film_id != i2.film_id AND
    r1.return_date >= r2.rental_date AND
    r1.rental_date <= r2.return_date
  )
```

18. Select customers (their names) which borrowed movie GRIT CLOCKWORK in May as well as in June of the same year.

```
SELECT first_name, last_name
FROM
  customer
  JOIN rental r1 ON customer.customer_id = r1.customer_id
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN film f1 ON i1.film_id = f1.film_id
WHERE
  f1.title = 'GRIT␣CLOCKWORK'
  AND MONTH(r1.rental_date) = 5
  AND EXISTS (
    SELECT *
    FROM
      rental r2
      JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
      JOIN film f2 ON i2.film_id = f2.film_id
    WHERE
      r1.customer_id = r2.customer_id
      AND f2.title = 'GRIT␣CLOCKWORK'
      AND MONTH(r2.rental_date) = 6
      AND YEAR(r1.rental_date) = YEAR(r2.rental_date)
```

```
)
```

19. Select the movies (their titles) shorter than all movies of actor BURT POSEY.

```
SELECT title
FROM film
WHERE length < ALL (
  SELECT film.length
  FROM
    actor
    JOIN film_actor ON actor.actor_id = film_actor.actor_id
    JOIN film ON film_actor.film_id = film.film_id
  WHERE actor.first_name = 'BURT' AND actor.last_name = 'POSEY'
)
```

OR

```
SELECT title
FROM film f1
WHERE NOT EXISTS
(
  SELECT *
  FROM
    actor
    JOIN film_actor ON actor.actor_id = film_actor.actor_id
    JOIN film f2 ON film_actor.film_id = f2.film_id
  WHERE actor.first_name = 'BURT' AND actor.last_name = 'POSEY' AND f2.
      length <= f1.length
)
```

20. Select name of the actors which play only in movies shorter than 180 minutes.

```
SELECT actor.first_name, actor.last_name
FROM actor
WHERE
  180 > ALL (
    SELECT length
    FROM film JOIN film_actor ON film.film_id = film_actor.film_id
    WHERE film_actor.actor_id = actor.actor_id
  )
  AND actor_id IN (SELECT actor_id FROM film_actor)
```

OR

```
SELECT actor.first_name, actor.last_name
FROM actor
WHERE
  NOT EXISTS (
    SELECT *
    FROM film JOIN film_actor ON film.film_id = film_actor.film_id
    WHERE film_actor.actor_id = actor.actor_id AND film.length >= 180
  )
  AND actor_id IN (SELECT actor_id FROM film_actor)
```

21. Select the customers (their names) which never borrowed more than 3 movies in the same month. Use aggregate functions and group by to get number of rents.

```sql
SELECT first_name, last_name
FROM customer
WHERE customer_id NOT IN
(
  SELECT customer_id
  FROM rental
  GROUP BY customer_id, MONTH(rental_date)
  HAVING COUNT(*) > 3
)
```

22. Select the customers (their names) which borrowed movies only during summer (it means in the July and August).

```sql
SELECT first_name, last_name
FROM customer
WHERE NOT EXISTS (
  SELECT *
  FROM rental
  WHERE
    customer.customer_id = rental.customer_id AND
    MONTH(rental.rental_date) NOT BETWEEN 6 AND 8
) AND customer_id IN (SELECT customer_id FROM rental)
```

OR

```sql
SELECT first_name, last_name
FROM customer
WHERE customer_id NOT IN
(
  SELECT customer_id
  FROM rental
  WHERE MONTH(rental.rental_date) NOT BETWEEN 6 AND 8
) AND customer_id IN (SELECT customer_id FROM rental)
```

23. Select the customers which have always returned the borrowed movies within 8 days. Ignore rentals that the customer has not returned yet.

```sql
SELECT *
FROM customer
WHERE NOT EXISTS (
  SELECT *
  FROM rental
  WHERE
    rental.customer_id = customer.customer_id
    AND DATEDIFF(day, rental.rental_date, rental.return_date) > 8
) AND customer_id IN (SELECT customer_id FROM rental)
```

24. Select the customers whose all rentals were longer than one day and they borrowed a movie starring DEBBIE AKROYD.

```sql
SELECT first_name, last_name
FROM customer
WHERE
  customer_id NOT IN (
    SELECT customer_id
```

```
    FROM rental
    WHERE DATEDIFF(DAY, rental_date, return_date) <= 1
  )
  AND customer_id IN (
    SELECT customer_id
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
      JOIN film_actor ON film.film_id = film_actor.film_id
      JOIN actor ON film_actor.actor_id = actor.actor_id
    WHERE
      actor.first_name = 'DEBBIE' AND actor.last_name = 'AKROYD'
  )
```

25. Select the names and surnames of customers who have made exactly one rent.

```
SELECT customer.first_name, customer.last_name
FROM
  rental r1
  JOIN customer ON r1.customer_id = customer.customer_id
WHERE NOT EXISTS (
  SELECT *
  FROM rental r2
  WHERE r1.customer_id = r2.customer_id AND
    r1.rental_id != r2.rental_id
)
```

OR

```
SELECT customer.first_name, customer.last_name
FROM
  rental
  JOIN customer ON rental.customer_id = customer.customer_id
GROUP BY customer.customer_id, customer.first_name, customer.last_name
HAVING COUNT(*) = 1
```

26. Select titles of the movies where only one actor plays.

```
SELECT film.film_id, film.title
FROM
  film
  JOIN film_actor fa1 ON film.film_id = fa1.film_id
WHERE NOT EXISTS (
  SELECT *
  FROM film_actor fa2
  WHERE fa1.film_id = fa2.film_id AND fa1.actor_id != fa2.actor_id
)
```

OR

```
SELECT film.film_id, film.title
FROM
  film
  JOIN film_actor ON film.film_id = film_actor.film_id
GROUP BY film.film_id, film.title
HAVING COUNT(*) = 1
```

27. Select customers who have always borrowed only the same movie.

```
SELECT DISTINCT customer.first_name, customer.last_name
FROM
  rental r1
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN customer ON r1.customer_id = customer.customer_id
WHERE NOT EXISTS (
  SELECT *
  FROM
    rental r2
    JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
  WHERE r1.customer_id = r2.customer_id AND i1.film_id != i2.film_id
)
```

OR

```
SELECT customer.first_name, customer.last_name
FROM
  rental
  JOIN customer ON rental.customer_id = customer.customer_id
  JOIN inventory ON inventory.inventory_id = rental.inventory_id
GROUP BY customer.customer_id, customer.first_name, customer.last_name
HAVING COUNT(DISTINCT inventory.film_id) = 1
```

`COUNT(DISTINCT film_id)` bude pro jednotlivé zákazníky počítat unikátní hodnoty atributu `film_id`. Klauzulí `HAVING` zajistíme, aby byl počet unikátních výskytů roven 1.

28. Select the titles of movies that have ever been rented by customers which have never rented another movie.

```
SELECT DISTINCT film.title
FROM
  rental r1
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN film ON i1.film_id = film.film_id
  JOIN customer ON r1.customer_id = customer.customer_id
WHERE NOT EXISTS (
  SELECT *
  FROM
    rental r2
    JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
  WHERE r1.customer_id = r2.customer_id AND i1.film_id != i2.film_id
)
```

29. Select all customers (names and surnames) and languages if the customer only rented movies in that language.

```
SELECT DISTINCT customer.first_name, customer.last_name, language.name
FROM
  customer
  JOIN rental r1 ON customer.customer_id = r1.customer_id
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN film f1 ON i1.film_id = f1.film_id
  JOIN language ON f1.language_id = language.language_id
WHERE NOT EXISTS (
```

```
    SELECT *
    FROM
      rental r2
      JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
      JOIN film f2 ON i2.film_id = f2.film_id
    WHERE r2.customer_id = r1.customer_id AND f2.language_id != f1.
        language_id
  )
```

OR

```
SELECT customer.first_name, customer.last_name, MIN(language.name) AS name
FROM
customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
  LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
  LEFT JOIN film ON inventory.film_id = film.film_id
  LEFT JOIN language ON film.language_id = language.language_id
GROUP BY customer.customer_id, customer.first_name, customer.last_name
HAVING COUNT(DISTINCT language.language_id) = 1
```

30. Select titles of the movies that have only been rented by customers who have never rented another movie.

```
SELECT title
FROM film
WHERE
  film_id NOT IN
  (
    SELECT i1.film_id
    FROM
      rental r1
      JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
    WHERE EXISTS (
      SELECT *
      FROM rental r2 JOIN inventory i2 ON r2.inventory_id = i2.
          inventory_id
      WHERE r1.customer_id = r2.customer_id AND i1.film_id != i2.film_id
    )
  )
  AND film_id IN (
    SELECT film_id
    FROM
      inventory
      JOIN rental ON inventory.inventory_id = rental.inventory_id
  )
```

31. Select names and surnames of the customers which have always borrowed only movies where the actor CHRISTIAN GABLE starred.

```
SELECT first_name, last_name
FROM customer
WHERE customer_id NOT IN
(
  SELECT DISTINCT customer_id
  FROM
    rental
```

```
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
    WHERE film_id NOT IN (
      SELECT film_id
      FROM
        film_actor
        JOIN actor ON film_actor.actor_id = actor.actor_id
      WHERE first_name = 'CHRISTIAN' AND last_name = 'GABLE'
    )
  ) AND customer_id IN (SELECT customer_id FROM rental)
```

32. Select the actors which have always played only in a movie owned by rental in at least three copies. Use aggregate function to get the number of copies in the inventory. ční funkci.

```
SELECT first_name, last_name
FROM actor
WHERE actor_id NOT IN
(
  SELECT actor_id
  FROM film_actor
  WHERE film_id NOT IN
  (
    SELECT film.film_id
    FROM
      film
      LEFT JOIN inventory ON film.film_id = inventory.film_id
    GROUP BY film.film_id
    HAVING COUNT(*) >= 3
  )
) AND actor_id IN (SELECT actor_id FROM film_actor)
```

OR

```
SELECT first_name, last_name
FROM actor
WHERE actor_id NOT IN
(
  SELECT actor_id
  FROM film_actor
  WHERE film_id IN
  (
    SELECT film.film_id
    FROM
      film
      LEFT JOIN inventory ON film.film_id = inventory.film_id
    GROUP BY film.film_id
    HAVING COUNT(*) < 3
  )
) AND actor_id IN (SELECT actor_id FROM film_actor)
```

OR

```
SELECT first_name, last_name
FROM actor
WHERE actor_id NOT IN
(
  SELECT film_actor.actor_id
```

```
  FROM
    film
    JOIN film_actor ON film.film_id = film_actor.film_id
    LEFT JOIN inventory ON film.film_id = inventory.film_id
  GROUP BY film.film_id, film_actor.actor_id
  HAVING COUNT(*) < 3
) AND actor_id IN (SELECT actor_id FROM film_actor)
```

33. Select the movies whose all copies have been rented at least 4x. Use aggregate function to get the number of copies in the inventory.

```
SELECT title
FROM film
WHERE film_id NOT IN
(
  SELECT inventory.film_id
  FROM
    inventory
    LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  GROUP BY inventory.inventory_id, inventory.film_id
  HAVING COUNT(rental.rental_id) < 4
)
AND film_id IN (SELECT film_id FROM inventory)
```

34. Select the actors (their names) whose all movies are longer than the movies where the actor CHRISTIAN GABLE starred.

```
SELECT first_name, last_name
FROM actor
WHERE actor_id NOT IN (
  SELECT film_actor.actor_id
  FROM
    film_actor
    JOIN film ON film_actor.film_id = film.film_id
  WHERE film.length < SOME (
    SELECT film.length
    FROM
      actor
      JOIN film_actor ON actor.actor_id = film_actor.actor_id
      JOIN film ON film_actor.film_id = film.film_id
    WHERE actor.first_name = 'CHRISTIAN' AND
      actor.last_name = 'GABLE'
  )
) AND actor_id IN (SELECT actor_id FROM film_actor)
```

35. Select the actors whose movies, longer than 180 minutes, have been borrowed by customers from the same country.

```
SELECT actor.actor_id, first_name, last_name
FROM actor
WHERE NOT EXISTS (
  SELECT film_actor.actor_id
  FROM
    film_actor
    JOIN film ON film_actor.film_id = film.film_id
    JOIN inventory i1 ON film.film_id = i1.film_id
```

```
  JOIN rental r1 ON i1.inventory_id = r1.inventory_id
  JOIN customer c1 ON r1.customer_id = c1.customer_id
  JOIN address a1 ON c1.address_id = a1.address_id
  JOIN city ct1 ON a1.city_id = ct1.city_id
WHERE film_actor.actor_id = actor.actor_id AND film.length > 180 AND
   EXISTS (
    SELECT *
    FROM
      inventory i2
      JOIN rental r2 ON i2.inventory_id = r2.inventory_id
      JOIN customer c2 ON r2.customer_id = c2.customer_id
      JOIN address a2 ON c2.address_id = a2.address_id
      JOIN city ct2 ON a2.city_id = ct2.city_id
    WHERE i2.film_id = i1.film_id AND ct2.country_id != ct1.country_id
  )
)
```

5 Subqueries

The last practice from SQL language is focused on the subqueries in general. The subqueries can solve many complex task in very elegant way.

1. Select the number of actors in a movie and the number of categories of a movie for each movie in the database.

```
SELECT
  film.film_id, film.title, COUNT(DISTINCT actor_id) AS actors,
  COUNT(DISTINCT category_id) AS categories
FROM
  film
  LEFT JOIN film_category ON film.film_id = film_category.film_id
  LEFT JOIN film_actor ON film.film_id = film_actor.film_id
GROUP BY film.film_id, film.title;
```

OR BY SUBQUERIES:

```
SELECT
  film.film_id, film.title,
  (
    SELECT COUNT(*)
    FROM film_actor
    WHERE film_actor.film_id = film.film_id
  ) AS actors,
  (
    SELECT COUNT(*)
    FROM film_category
    WHERE film_category.film_id = film.film_id
  ) AS categories
FROM film
```

OR BY CTE (Common Table Expressions):

```
WITH
  actors_cte AS (
    SELECT film.film_id, film.title, COUNT(film_actor.film_id) AS actors
    FROM film LEFT JOIN film_actor ON film.film_id = film_actor.film_id
    GROUP BY film.film_id, film.title
  ),
  categories_cte AS (
    SELECT film.film_id, COUNT(film_category.film_id) AS categories
    FROM film LEFT JOIN film_category ON film.film_id = film_category.
        film_id
    GROUP BY film.film_id, film.title
  )
SELECT actors_cte.film_id, actors_cte.title, actors, categories
FROM
  actors_cte
  JOIN categories_cte ON actors_cte.film_id = categories_cte.film_id
```

2. Select the number of borrowings lasting less than 5 days and the number of borrowings lasting less than 7 days for for each customer.

```
SELECT
  first_name, last_name,
  (
    SELECT COUNT(*)
    FROM rental
    WHERE
      rental.customer_id = customer.customer_id
      AND DATEDIFF(day, rental_date, return_date) < 5
  ) AS less_5,
  (
    SELECT COUNT(*)
    FROM rental
    WHERE
      rental.customer_id = customer.customer_id
      AND DATEDIFF(day, rental_date, return_date) < 7
  ) AS less_7
FROM customer
```

OR BY CTE (Common Table Expressions):

```
WITH
  k5 AS
  (
    SELECT customer.customer_id, customer.first_name, customer.last_name,
       COUNT(rental.rental_id) AS less_5
    FROM
      customer
      LEFT JOIN rental ON customer.customer_id = rental.customer_id
        AND DATEDIFF(day, rental_date, return_date) < 5
    GROUP BY customer.customer_id, customer.first_name, customer.last_name
  ),
  k7 AS
  (
    SELECT customer.customer_id, customer.first_name, customer.last_name,
       COUNT(rental.rental_id) AS less_7
    FROM
      customer
      LEFT JOIN rental ON customer.customer_id = rental.customer_id
        AND DATEDIFF(day, rental_date, return_date) < 7
    GROUP BY customer.customer_id, customer.first_name, customer.last_name
  )
SELECT k5.first_name, k5.last_name, less_5, less_7
FROM k5 JOIN k7 ON k5.customer_id = k7.customer_id;
```

3. Select the number of copies (it means items in the store) of the English and French films for each store.

```
SELECT
  store.store_id,
  (
    SELECT COUNT(*)
    FROM
      inventory
      JOIN film ON inventory.film_id = film.film_id
      JOIN language ON film.language_id = language.language_id
    WHERE inventory.store_id = store.store_id AND language.name = 'English
       '
```

```
  ) AS english,
  (
    SELECT COUNT(*)
    FROM
      inventory
      JOIN film ON inventory.film_id = film.film_id
      JOIN language ON film.language_id = language.language_id
    WHERE inventory.store_id = store.store_id AND language.name = 'French'
  ) AS french
FROM store
```

OR BY CTE (Common Table Expressions):

```
WITH t AS (
  SELECT inventory.store_id, language.name
  FROM
    inventory
    JOIN film ON inventory.film_id = film.film_id
    JOIN language ON film.language_id = language.language_id
)
SELECT
  store_id,
  (
    SELECT COUNT(*)
    FROM t
    WHERE t.name = 'English' AND t.store_id = store.store_id
  ) AS english,
  (
    SELECT COUNT(*)
    FROM t
    WHERE t.name = 'French' AND t.store_id = store.store_id
  ) AS czech
FROM store
```

4. Select following information for each movie:

   (a) the number of actors in the movie,

   (b) the number of different customers who rented the movie in August,

   (c) the average amount of payment for your movie rental.

```
SELECT
  film.film_id,
  film.title,
  (
    SELECT COUNT(*)
    FROM film_actor
    WHERE film_actor.film_id = film.film_id
  ) AS actors,
  (
    SELECT COUNT(DISTINCT customer_id)
    FROM
      inventory
      JOIN rental ON inventory.inventory_id = rental.inventory_id
    WHERE
      inventory.film_id = film.film_id
```

```
      AND MONTH(rental.rental_date) = 8
  ) AS customers,
  (
    SELECT AVG(amount)
    FROM
      payment
      JOIN rental ON payment.rental_id = rental.rental_id
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
    WHERE inventory.film_id = film.film_id
  ) AS avg_payment
FROM film
```

5. Select customers with more than 5 payments in June and the longest movie they rented has at least 185 minutes.

```
SELECT first_name, last_name
FROM customer
WHERE
  (
    SELECT COUNT(*)
    FROM payment
    WHERE payment.customer_id = customer.customer_id AND MONTH(
       payment_date) = 6
  ) > 5 AND
  (
    SELECT MAX(length)
    FROM
      film
      JOIN inventory ON film.film_id = inventory.film_id
      JOIN rental ON inventory.inventory_id = rental.inventory_id
    WHERE rental.customer_id = customer.customer_id
  ) >= 185
```

OR

```
SELECT first_name, last_name
FROM
(
  SELECT first_name, last_name,
    (
      SELECT COUNT(*)
      FROM payment
      WHERE payment.customer_id = customer.customer_id AND MONTH(
         payment_date) = 6
    ) AS payments,
    (
      SELECT MAX(length)
      FROM
        film
        JOIN inventory ON film.film_id = inventory.film_id
        JOIN rental ON inventory.inventory_id = rental.inventory_id
      WHERE rental.customer_id = customer.customer_id
    ) AS max_length
  FROM customer
) t
WHERE payments > 5 AND max_length >= 185
```

6. Select customers whose payments are with amount higher than 4 in the most cases.

```
SELECT first_name, last_name
FROM customer
WHERE
(
  SELECT COUNT(*)
  FROM payment
  WHERE payment.customer_id = customer.customer_id AND amount > 4
) >
(
  SELECT COUNT(*)
  FROM payment
  WHERE payment.customer_id = customer.customer_id AND amount <= 4
)
```

OR

```
SELECT first_name, last_name
FROM
(
  SELECT first_name, last_name,
  (
    SELECT COUNT(*)
    FROM payment
    WHERE payment.customer_id = customer.customer_id AND amount > 4
  ) AS higher_4,
  (
    SELECT COUNT(*)
    FROM payment
    WHERE payment.customer_id = customer.customer_id AND amount <= 4
  ) AS lower_4
  FROM customer
) pocty
WHERE higher_4 > lower_4
```

7. Select actors playing in comedies two times more often than in horror movies.

```
SELECT first_name, last_name
FROM actor
WHERE
(
  SELECT COUNT(*)
  FROM film_actor
  WHERE film_actor.actor_id = actor.actor_id AND film_id IN (
    SELECT film_id
    FROM
      film_category
      JOIN category ON film_category.category_id = category.category_id
    WHERE category.name = 'comedy'
  )
)
>
(
  SELECT COUNT(*)
  FROM film_actor
  WHERE film_actor.actor_id = actor.actor_id AND film_id IN (
    SELECT film_id
```

```
    FROM
      film_category
      JOIN category ON film_category.category_id = category.category_id
    WHERE category.name = 'horror'
  )
) * 2
```

8. Select the actors playing most often in movies longer than 150 minutes,it means they play more often in movies longer than 150 minutes than in other movies.

```
SELECT actor_id, first_name, last_name
FROM actor
WHERE
(
  SELECT COUNT(*)
  FROM film_actor JOIN film ON film_actor.film_id = film.film_id
  WHERE film_actor.actor_id = actor.actor_id AND length > 150
)
>
(
  SELECT COUNT(*)
  FROM film_actor JOIN film ON film_actor.film_id = film.film_id
  WHERE film_actor.actor_id = actor.actor_id AND length <= 150
)
```

9. Select customers whose total payments are less than they should pay according to attributes the `film.rental_duration`, `film.rental_rate` and difference between attributes `rental_date` and `return_date`. You can ignore non-returned rents.

```
SELECT first_name, last_name
FROM customer
WHERE
(
  SELECT SUM(amount)
  FROM
    rental
    JOIN payment ON rental.rental_id = payment.rental_id
  WHERE rental.customer_id = customer.customer_id
)
<
(
  SELECT SUM(film.rental_rate * DATEDIFF(day, rental.rental_date, rental.
     return_date) / film.rental_duration)
  FROM
    rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
    JOIN film ON inventory.film_id = film.film_id
  WHERE rental.customer_id = customer.customer_id
)
```

10. Select customers borrowing movies with actor TOM MCKELLEN more often than movies with actor GROUCHO SINATRA.

```
SELECT first_name, last_name
FROM customer
WHERE
```

```
(
  SELECT COUNT(*)
  FROM
    rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
  WHERE rental.customer_id = customer.customer_id AND film_id IN
  (
    SELECT film_id
    FROM
      actor
      JOIN film_actor ON actor.actor_id = film_actor.actor_id
    WHERE actor.first_name = 'TOM' AND actor.last_name = 'MCKELLEN'
  )
)
>
(
  SELECT COUNT(*)
  FROM
    rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
  WHERE rental.customer_id = customer.customer_id AND film_id IN
  (
    SELECT film_id
    FROM
      actor
      JOIN film_actor ON actor.actor_id = film_actor.actor_id
    WHERE actor.first_name = 'GROUCHO' AND actor.last_name = 'SINATRA'
  )
)
```

OR

```
SELECT first_name, last_name
FROM customer
WHERE
  (
    SELECT COUNT(*)
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film_actor ON inventory.film_id = film_actor.film_id
      JOIN actor ON film_actor.actor_id = actor.actor_id
    WHERE rental.customer_id = customer.customer_id AND actor.first_name =
        'TOM' AND actor.last_name = 'MCKELLEN'
  )
  >
  (
    SELECT COUNT(*)
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film_actor ON inventory.film_id = film_actor.film_id
      JOIN actor ON film_actor.actor_id = actor.actor_id
    WHERE rental.customer_id = customer.customer_id AND actor.first_name =
        'GROUCHO' AND actor.last_name = 'SINATRA'
  )
```

11. Select customers renting only movies in english language together with information how many rents they have.

```
SELECT first_name, last_name,
(
  SELECT COUNT(*)
  FROM rental
  WHERE rental.customer_id = customer.customer_id
) AS rents_count
FROM customer
WHERE NOT EXISTS (
  SELECT *
  FROM
    rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
    JOIN film ON inventory.film_id = film.film_id
    JOIN language ON film.language_id = language.language_id
  WHERE rental.customer_id = customer.customer_id AND language.name != '
     English'
) AND customer_id IN (SELECT customer_id FROM rental)
```

OR

```
SELECT first_name, last_name, COUNT(rental.rental_id) AS rents_count
FROM
  customer
  JOIN rental ON customer.customer_id = rental.customer_id
WHERE NOT EXISTS (
  SELECT *
  FROM
    rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
    JOIN film ON inventory.film_id = film.film_id
    JOIN language ON film.language_id = language.language_id
  WHERE rental.customer_id = customer.customer_id AND language.name != '
     English'
)
GROUP BY customer.customer_id, first_name, last_name
```

12. Select customers who rented a movie with at least 15 actors together with the total amount of the payments they made.

```
SELECT
  first_name, last_name,
  (
    SELECT SUM(amount)
    FROM payment
    WHERE payment.customer_id = customer.customer_id
  ) AS total_amount
FROM customer
WHERE customer_id IN
(
  SELECT customer_id
  FROM
    rental
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
  WHERE inventory.film_id IN
```

```
    (
      SELECT film_id
      FROM film_actor
      GROUP BY film_id
      HAVING COUNT(*) >= 15
    )
)
```

13. Select the name of the longest movie(s).

    NOT CORRECT SOLUTION:

```
SELECT TOP 1 title
FROM film
ORDER BY length DESC
```

    CORRECT SOLUTIONS:

```
SELECT title
FROM film
WHERE length = (
  SELECT MAX(length)
  FROM film
)
```

    OR

```
SELECT title
FROM film
WHERE length >= ALL (
  SELECT length
  FROM film
)
```

    OR

```
SELECT title
FROM film f1
WHERE NOT EXISTS (
  SELECT *
  FROM film f2
  WHERE f2.length > f1.length
)
```

14. Select the name of the longest movie(s) for each rating (attribute `film.rating`).

```
SELECT rating, title
FROM film f1
WHERE length = (
  SELECT MAX(length)
  FROM film f2
  WHERE f1.rating = f2.rating
)
ORDER BY rating
```

    OR

```
SELECT rating, title
FROM film f1
WHERE length >= ALL(
  SELECT length
  FROM film f2
  WHERE f1.rating = f2.rating
)
ORDER BY rating
```

15. For each customer, find the last movie he rented. Sort the result alphabetically by last name and first name of the customers.

```
SELECT customer.customer_id, first_name, last_name, film.title
FROM
  customer
  JOIN rental r1 ON customer.customer_id = r1.customer_id
  JOIN inventory ON r1.inventory_id = inventory.inventory_id
  JOIN film ON inventory.film_id = film.film_id
WHERE r1.rental_date = (
  SELECT MAX(rental_date)
  FROM rental r2
  WHERE r1.customer_id = r2.customer_id
)
ORDER BY last_name, first_name
```

OR

```
SELECT customer.customer_id, first_name, last_name, film.title
FROM
  customer
  JOIN rental r1 ON customer.customer_id = r1.customer_id
  JOIN inventory ON r1.inventory_id = inventory.inventory_id
  JOIN film ON inventory.film_id = film.film_id
  WHERE r1.rental_date >= ALL (
    SELECT rental_date
    FROM rental r2
    WHERE r1.customer_id = r2.customer_id
  )
ORDER BY last_name, first_name
```

16. Select all actors (their name and surname) together with their longest movie.

```
SELECT actor.first_name, actor.last_name, film.title
FROM
  actor
  JOIN film_actor fa1 ON actor.actor_id = fa1.actor_id
  JOIN film ON fa1.film_id = film.film_id
WHERE film.length >= ALL (
  SELECT film.length
  FROM
    film
    JOIN film_actor fa2 ON film.film_id = fa2.film_id
  WHERE fa2.actor_id = fa1.actor_id
)
```

OR

```
SELECT actor.first_name, actor.last_name, f1.title
FROM
  actor
  JOIN film_actor fa1 ON actor.actor_id = fa1.actor_id
  JOIN film f1 ON fa1.film_id = f1.film_id
WHERE NOT EXISTS
(
  SELECT *
  FROM
    film_actor fa2
    JOIN film f2 ON fa2.film_id = f2.film_id
  WHERE fa2.actor_id = actor.actor_id AND f2.length > f1.length
)
```

17. Select all movies together with the customers who have borrowed them for the longest time (within one rent).

```
SELECT title, first_name, last_name
FROM
  film
  JOIN inventory i1 ON film.film_id = i1.film_id
  JOIN rental ON i1.inventory_id = rental.inventory_id
  JOIN customer ON rental.customer_id = customer.customer_id
WHERE
  DATEDIFF(day, rental.rental_date, rental.return_date) >= ALL (
    SELECT DATEDIFF(day, rental.rental_date, rental.return_date)
    FROM
      inventory i2
      JOIN rental ON i2.inventory_id = rental.inventory_id
    WHERE i2.film_id = i1.film_id
  )
```

18. For each customer, select the last borrowed movie starring actor PENELOPE GUINESS. If the customer has never rented a movie with PENELOPE GUINESS, the customer will not be selected. Sort the result by customer ID.

```
SELECT customer.customer_id, first_name, last_name, film.title
FROM
  customer
  JOIN rental r1 ON customer.customer_id = r1.customer_id
  JOIN inventory i1 ON r1.inventory_id = i1.inventory_id
  JOIN film ON i1.film_id = film.film_id
WHERE
  film.film_id IN (
    SELECT film_id
    FROM film_actor JOIN actor ON film_actor.actor_id = actor.actor_id
    WHERE actor.first_name = 'PENELOPE' AND actor.last_name = 'GUINESS'
  ) AND r1.rental_date = (
    SELECT MAX(rental_date)
    FROM rental r2 JOIN inventory i2 ON r2.inventory_id = i2.inventory_id
    WHERE r1.customer_id = r2.customer_id AND i2.film_id IN (
      SELECT film_id
      FROM film_actor JOIN actor ON film_actor.actor_id = actor.actor_id
      WHERE actor.first_name = 'PENELOPE' AND actor.last_name = 'GUINESS'
    )
  )
```

```
ORDER BY customer.customer_id
```

OR

```
WITH film_pg AS
(
  SELECT film_id, title
  FROM film
  WHERE film_id IN
  (
    SELECT film_id
    FROM film_actor JOIN actor ON film_actor.actor_id = actor.actor_id
    WHERE actor.first_name = 'PENELOPE' AND actor.last_name = 'GUINESS'
  )
)
SELECT customer.customer_id, first_name, last_name, film_pg.title
FROM
  customer
  JOIN rental r1 ON customer.customer_id = r1.customer_id
  JOIN inventory ON r1.inventory_id = inventory.inventory_id
  JOIN film_pg ON inventory.film_id = film_pg.film_id
WHERE r1.rental_date = (
  SELECT MAX(rental_date)
  FROM
    rental r2
    JOIN inventory ON r2.inventory_id = inventory.inventory_id
    JOIN film_pg ON inventory.film_id = film_pg.film_id
  WHERE r1.customer_id = r2.customer_id
)
ORDER BY customer.customer_id
```

19. List customers who have borrowed both the shortest and the longest movie.

```
SELECT first_name, last_name
FROM customer
WHERE
  customer_id IN
  (
    SELECT rental.customer_id
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
    WHERE film.length = (SELECT MIN(length) FROM film)
  )
  AND customer_id IN
  (
    SELECT rental.customer_id
    FROM
      rental
      JOIN inventory ON rental.inventory_id = inventory.inventory_id
      JOIN film ON inventory.film_id = film.film_id
    WHERE film.length = (SELECT MIN(length) FROM film)
  )
```

20. Select the actors who played at least 2 times in the longest film.

```
SELECT actor.actor_id, first_name, last_name
FROM
  actor
  JOIN film_actor ON actor.actor_id = film_actor.actor_id
  JOIN film ON film_actor.film_id = film.film_id
WHERE length = (SELECT MAX(length) FROM film)
GROUP BY actor.actor_id, first_name, last_name
HAVING COUNT(film.film_id) >= 2
```

OR

```
WITH t AS
(
  SELECT film_id
  FROM film
  WHERE length = (SELECT MAX(length) FROM film)
)
SELECT actor.actor_id, actor.first_name, actor.last_name
FROM
  actor
  JOIN film_actor ON actor.actor_id = film_actor.actor_id
  JOIN t ON film_actor.film_id = t.film_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name
HAVING COUNT(film_actor.film_id) >= 2
```

OR

```
WITH t AS
(
  SELECT film_id
  FROM film
  WHERE length >= ALL(SELECT length FROM film)
)
SELECT DISTINCT actor.actor_id, first_name, last_name
FROM
  actor
  JOIN film_actor fa1 ON actor.actor_id = fa1.actor_id
  JOIN t t1 ON fa1.film_id = t1.film_id
WHERE EXISTS
(
  SELECT *
  FROM
    film_actor fa2
    JOIN t t2 ON fa2.film_id = t2.film_id
  WHERE fa2.actor_id = fa1.actor_id AND fa2.film_id != fa1.film_id
)
```

21. Select movies that at least two customers rented for the last time.

```
SELECT film_id, title
FROM
(
  SELECT film.film_id, film.title, customer_id
  FROM
    rental r1
    JOIN inventory ON r1.inventory_id = inventory.inventory_id
    JOIN film ON inventory.film_id = film.film_id
```

```
  WHERE r1.rental_date = (
    SELECT MAX(rental_date)
    FROM rental r2
    WHERE r1.customer_id = r2.customer_id
  )
) t
GROUP BY film_id, title
HAVING COUNT(*) >= 2
```

22. Select all actors together with the average number of rents for the movies in which they play.

```
SELECT actor_id, first_name, last_name, AVG(CAST(rent_count AS FLOAT)) AS
    average
FROM
(
  SELECT actor.actor_id, first_name, last_name, film.film_id, COUNT(rental
      .rental_id) AS rent_count
  FROM
    actor
    LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
    LEFT JOIN film ON film_actor.film_id = film.film_id
    LEFT JOIN inventory ON film.film_id = inventory.film_id
    LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  GROUP BY actor.actor_id, first_name, last_name, film.film_id
) t
GROUP BY actor_id, first_name, last_name
```

OR

```
WITH t AS
(
  SELECT actor.actor_id, first_name, last_name, film.film_id, COUNT(rental
      .rental_id) AS rent_count
  FROM
    actor
    LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
    LEFT JOIN film ON film_actor.film_id = film.film_id
    LEFT JOIN inventory ON film.film_id = inventory.inventory_id
    LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  GROUP BY actor.actor_id, first_name, last_name, film.film_id
)
SELECT actor_id, first_name, last_name, AVG(CAST(rent_count AS FLOAT)) AS
    average
FROM t
GROUP BY actor_id, first_name, last_name
```

23. For each movie classification (attribute texttt film.rating), select the largest number of actors playing in the movie of that classification.

```
WITH
  rating AS (
    SELECT DISTINCT rating
    FROM film
  ),
```

```
  actors_count AS (
    SELECT film.rating, film.film_id, COUNT(film_actor.film_id) AS countA
    FROM
      film
      LEFT JOIN film_actor ON film.film_id = film_actor.film_id
    GROUP BY film.rating, film.film_id
  )
SELECT rating.rating, MAX(countA) AS max_actors
FROM
  rating
  LEFT JOIN pocty_hercu ON rating.rating = pocty_hercu.rating
GROUP BY rating.rating;
```

OR

```
WITH actors_count AS (
  SELECT film.rating, film.film_id, COUNT(film_actor.film_id) AS countA
  FROM
    film
    LEFT JOIN film_actor ON film.film_id = film_actor.film_id
  GROUP BY film.rating, film.film_id
)
SELECT rating, MAX(countA) AS max_actors
FROM actors_count
GROUP BY rating
```

24. Select the most frequently cast actors, it means the actors who play in the largest number of movies. The number of movies the actor plays will be included in the result.

```
SELECT actor.first_name, actor.last_name, COUNT(film_actor.actor_id) AS
    actors
FROM
  actor
  LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name
HAVING COUNT(film_actor.actor_id) =
(
  SELECT MAX(pocet)
  FROM
  (
    SELECT COUNT(film_actor.actor_id) as actors
    FROM
      actor
      LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
    GROUP BY actor.actor_id
  ) t
)
```

OR

```
SELECT actor.first_name, actor.last_name, COUNT(film_actor.actor_id) AS
    actors
FROM
  actor
  LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
GROUP BY actor.actor_id, actor.first_name, actor.last_name
HAVING COUNT(film_actor.actor_id) >= ALL
```

```
(
  SELECT COUNT(film_actor.actor_id) as actors
  FROM
    actor
    LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
  GROUP BY actor.actor_id, actor.first_name, actor.last_name
);
```

OR

```
WITH t AS
(
  SELECT actor.actor_id, actor.first_name, actor.last_name, COUNT(
      film_actor.actor_id) as actors
  FROM
    actor
    LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
  GROUP BY actor.actor_id, actor.first_name, actor.last_name
)
SELECT first_name, last_name, pocet
FROM t
WHERE actors >= ALL(SELECT actors FROM t)
```

25. Select customers with the most rents.

```
SELECT customer.first_name, customer.last_name, COUNT(rental.rental_id) as
    rents
FROM
  customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
GROUP BY customer.customer_id, customer.first_name, customer.last_name
HAVING COUNT(rental.rental_id) = (
  SELECT MAX(rents)
  FROM
  (
    SELECT COUNT(rental.rental_id) as rents
    FROM
      customer
      LEFT JOIN rental ON customer.customer_id = rental.customer_id
    GROUP BY customer.customer_id
  ) t
)
```

OR

```
SELECT customer.first_name, customer.last_name, COUNT(rental.rental_id) as
    rents
FROM
  customer
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
GROUP BY customer.customer_id, customer.first_name, customer.last_name
HAVING COUNT(rental.rental_id) >= ALL (
  SELECT COUNT(rental.rental_id) as rents
  FROM
    customer
    LEFT JOIN rental ON customer.customer_id = rental.customer_id
```

```
  GROUP BY customer.customer_id
)
```

OR

```
WITH t AS
(
  SELECT customer.customer_id, customer.first_name, customer.last_name,
     COUNT(rental.rental_id) as rents
  FROM
    customer
    LEFT JOIN rental ON customer.customer_id = rental.customer_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name
)
SELECT first_name, last_name, rents
FROM t
WHERE rents = (SELECT MAX(rents) FROM t)
```

26. Select the titles of movies that have been rented the most times. The number of rents will be included in the result.

```
SELECT film.film_id, film.title, COUNT(rental.rental_id) AS rents
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
GROUP BY film.film_id, film.title
HAVING COUNT(rental.rental_id) = (
  SELECT MAX(rents)
  FROM
  (
    SELECT COUNT(rental.rental_id) AS rents
    FROM
      film
      LEFT JOIN inventory ON film.film_id = inventory.film_id
      LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
    GROUP BY film.film_id
  ) t
)
```

27. Select the customers who made the most payments. The highest number of payments should be included in the result.

```
SELECT customer.first_name, customer.last_name, COUNT(payment.payment_id)
   AS payments
FROM
  customer
  LEFT JOIN payment ON customer.customer_id = payment.customer_id
GROUP BY customer.customer_id, customer.first_name, customer.last_name
HAVING COUNT(payment.payment_id) >= ALL
(
  SELECT COUNT(payment.payment_id)
  FROM
    customer
    LEFT JOIN payment ON customer.customer_id = payment.customer_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name
)
```

28. Select titles of the movies with number of rents higher than average number of rents.

```
SELECT film.film_id, film.title, COUNT(rental.rental_id) AS rents
FROM
  film
  LEFT JOIN inventory ON film.film_id = inventory.film_id
  LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
GROUP BY film.film_id, film.title
HAVING COUNT(rental.rental_id) > (
  SELECT AVG(rents)
  FROM
  (
    SELECT COUNT(rental.rental_id) AS rents
    FROM
      film
      LEFT JOIN inventory ON film.film_id = inventory.film_id
      LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
    GROUP BY film.film_id
  ) t
)
```

OR

```
SELECT film.film_id, film.title
FROM film
WHERE
(
  SELECT COUNT(*)
  FROM inventory JOIN rental ON inventory.inventory_id = rental.
     inventory_id
  WHERE inventory.film_id = film.film_id
)
>
(
  SELECT AVG(rents)
  FROM
  (
    SELECT film.film_id, film.title, COUNT(rental.rental_id) AS rents
    FROM
      film
      LEFT JOIN inventory ON film.film_id = inventory.film_id
      LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
    GROUP BY film.film_id, film.title
  ) rentals
)
```

OR

```
WITH t AS
(
  SELECT film.film_id, film.title, COUNT(rental.rental_id) AS rents
  FROM
    film
    LEFT JOIN inventory ON film.film_id = inventory.film_id
    LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  GROUP BY film.film_id, film.title
)
SELECT film_id, title
```

```
FROM t
WHERE rents > (SELECT AVG(rents) FROM t)
```

29. Select the actors playing most often in moviess longer than 150 minutes, it means in movies with a length over 150 minutes, they are the most frequently cast actors.

```
WITH t AS (
  SELECT actor.actor_id, actor.first_name, actor.last_name, COUNT(film.
     film_id) AS films
  FROM
    actor
    LEFT JOIN film_actor ON actor.actor_id = film_actor.actor_id
    LEFT JOIN film ON film_actor.film_id = film.film_id AND film.length >
       150
  GROUP BY actor.actor_id, actor.first_name, actor.last_name
)
SELECT *
FROM t
WHERE films = (SELECT MAX(films) FROM t)
```

OR

```
WITH t AS (
  SELECT actor.actor_id, actor.first_name, actor.last_name, COUNT(film.
     film_id) AS films
  FROM
    actor
    JOIN film_actor ON actor.actor_id = film_actor.actor_id
    JOIN film ON film_actor.film_id = film.film_id
  WHERE film.length > 150
  GROUP BY actor.actor_id, actor.first_name, actor.last_name
)
SELECT *
FROM t
WHERE films = (SELECT MAX(films) FROM t)
```

30. Select the customers with the biggest difference between the minimum and maximum payment for a movie rent in June. The difference will be included in the result.

```
WITH t AS
(
  SELECT customer.customer_id, customer.first_name, customer.last_name,
     MAX(amount) - MIN(amount) AS diff
  FROM
    customer
    LEFT JOIN rental ON customer.customer_id = rental.customer_id AND
       MONTH(rental.rental_date) = 6
    LEFT JOIN payment ON rental.rental_id = payment.rental_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name
)
SELECT first_name, last_name
FROM t
WHERE diff = (SELECT MAX(diff) FROM t)
```

31. Select movies that have been rented by one customer the most times.

```
WITH t AS
(
  SELECT customer.customer_id, customer.first_name, customer.last_name,
     film.film_id, film.title, COUNT(rental.rental_id) AS rents
  FROM
    customer
    LEFT JOIN rental ON customer.customer_id = rental.customer_id
    LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
    LEFT JOIN film ON inventory.film_id = film.film_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name,
     film.film_id, film.title
)
SELECT DISTINCT title
FROM t
WHERE rents = (SELECT MAX(rents) FROM t)
```

32. List customers borrowing the same movie the most times.

```
WITH t AS
(
  SELECT customer.customer_id, customer.first_name, customer.last_name,
     film.film_id, film.title, COUNT(rental.rental_id) AS rents
  FROM
    customer
    LEFT JOIN rental ON customer.customer_id = rental.customer_id
    LEFT JOIN inventory ON rental.inventory_id = inventory.inventory_id
    LEFT JOIN film ON inventory.film_id = film.film_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name,
     film.film_id, film.title
)
SELECT DISTINCT first_name, last_name
FROM t
WHERE rents = (SELECT MAX(rents) FROM t)
```

33. For each city, select the customer with the most rents.

```
SELECT c1.city_id, city, customer.customer_id, first_name, last_name,
   COUNT(rental.rental_id) AS rents
FROM
  city c1
  LEFT JOIN address ON c1.city_id = address.city_id
  LEFT JOIN customer ON address.address_id = customer.customer_id
  LEFT JOIN rental ON customer.customer_id = rental.customer_id
GROUP BY c1.city_id, city, customer.customer_id, first_name, last_name
HAVING COUNT(rental.rental_id) =
(
  SELECT MAX(rents)
  FROM
  (
    SELECT COUNT(rental.rental_id) AS rents
    FROM
      city c2
      LEFT JOIN address ON c2.city_id = address.city_id
      LEFT JOIN customer ON address.address_id = customer.customer_id
      LEFT JOIN rental ON customer.customer_id = rental.customer_id
    WHERE c2.city_id = c1.city_id
    GROUP BY customer.customer_id
```

```
  ) t
)
```

OR

```
WITH t AS
(
  SELECT c1.city_id, city, customer.customer_id, first_name, last_name,
     COUNT(rental.rental_id) AS rents
  FROM
    city c1
    LEFT JOIN address ON c1.city_id = address.city_id
    LEFT JOIN customer ON address.address_id = customer.customer_id
    LEFT JOIN rental ON customer.customer_id = rental.customer_id
  GROUP BY c1.city_id, city, customer.customer_id, first_name, last_name
)
SELECT *
FROM t t1
WHERE rents >= ALL(SELECT rents FROM t t2 WHERE t1.city_id = t2.city_id)
```

34. Select all customers together with the title of the movie most often borrowed by him and the number of rents of this movie. Ignore customers without rents.

```
WITH t AS
(
  SELECT customer.customer_id, customer.first_name, customer.last_name,
     film.film_id, film.title, COUNT(rental.rental_id) AS rents
  FROM
    customer
    JOIN rental ON customer.customer_id = rental.customer_id
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
    JOIN film ON inventory.film_id = film.film_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name,
     film.film_id, film.title
)
SELECT DISTINCT first_name, last_name, title, pocet
FROM t t1
WHERE rents = (SELECT MAX(rents) FROM t t2 WHERE t1.customer_id = t2.
   customer_id)
```

35. Select all categories together with their movies with the lowest number of rents.

```
WITH t AS
(
  SELECT category.category_id, category.name, film.film_id, film.title,
     COUNT(rental.rental_id) AS rents
  FROM
    category
    JOIN film_category ON category.category_id = film_category.category_id
    JOIN film ON film_category.film_id = film.film_id
    LEFT JOIN inventory ON film.film_id = inventory.film_id
    LEFT JOIN rental ON inventory.inventory_id = rental.inventory_id
  GROUP BY category.category_id, category.name, film.film_id, film.title
)
SELECT *
FROM t t1
```

```
WHERE rents = (SELECT MIN(rents) FROM t t2 WHERE t1.category_id = t2.
    category_id)
ORDER BY category_id
```

36. Select all categories together with the most frequently cast actors in the movies of those categories.

```
WITH t AS
(
  SELECT category.category_id, category.name, actor.actor_id, actor.
     first_name, actor.last_name, COUNT(film.film_id) AS movies
  FROM
    category
    JOIN film_category ON category.category_id = film_category.category_id
    JOIN film ON film_category.film_id = film.film_id
    JOIN film_actor ON film.film_id = film_actor.film_id
    JOIN actor ON film_actor.actor_id = actor.actor_id
  GROUP BY category.category_id, category.name, actor.actor_id, actor.
     first_name, actor.last_name
)
SELECT *
FROM t t1
WHERE movies = (SELECT MAX(movies) FROM t t2 WHERE t1.category_id = t2.
    category_id)
ORDER BY category_id
```

37. Select all customers together with their favorite actor, it means the actor who played in the most different films the customer has borrowed. Ignore customers without rents.

```
WITH t AS (
  SELECT
    customer.customer_id, customer.first_name AS c_first_name, customer.
       last_name AS c_last_name,
    actor.actor_id, actor.first_name AS a_first_name, actor.last_name AS
       a_last_name,
  COUNT(DISTINCT film.film_id) AS movies
  FROM
    customer
    JOIN rental ON customer.customer_id = rental.customer_id
    JOIN inventory ON rental.inventory_id = inventory.inventory_id
    JOIN film ON inventory.film_id = film.film_id
    JOIN film_actor ON film.film_id = film_actor.film_id
    JOIN actor ON film_actor.actor_id = actor.actor_id
  GROUP BY customer.customer_id, customer.first_name, customer.last_name,
     actor.actor_id, actor.first_name, actor.last_name
)
SELECT *
FROM t t1
WHERE movies = (SELECT MAX(movies) FROM t t2 WHERE t1.customer_id = t2.
    customer_id)
ORDER BY customer_id
```

6 Commands for data modification and definition

So far, we have not made any modifications to our database. We used only SELECT statements, whose possibilities are huge, but the data itself and the structure of the database remain intact. Today, on the contrary, we will show commands belonging to the category of DML (Data Manipulation Language) for editing the content of tables and commands belonging to the category of DDL (Data Definition Language) for editing the structure of tables.

This practise will differ slightly in structure from the previous ones for several reasons. Some tasks will consist of several points that need to be solved in the exact order. Furthermore, if it is not specified directly, you can solve tasks with multiple SQL statements, which you will run sequentially.

Before we start solving the tasks, note that while the syntax of the SELECT statement is almost the same across different DBMS (the ANSI SQL standard is followed), there are often slight differences between the commands in the DML and DDL categories. In this collection we will show the syntax for Microsoft SQL Server. Generally, the solution of problems for other relational DBMS will not differ.

1. (a) Insert a new actor named Arnold Schwarzenegger into the database. Leave the default value for the last record update (`last_update`) (i.e. do not set this value).

```
INSERT INTO actor (first_name, last_name)
VALUES ('Arnold', 'Schwarzenegger');
```

Surely you understand that the INSERT statement inserts a new row (a record) into the `actor` table. Although the list of attributes after a table name is optional, we should always explicitly specify it for several reasons:

- Not all attributes are mandatory in the table, some attributes can be set to the default value (as in this case the `last_update` attribute) and we must not explicitly enter a value for automatically generated IDs (in this case `actor_id`).
- By omitting the parentheses, you rely on a specific order of columns (attributes) in the table. However, it can very easily happen that in another database the order of the columns will be different. If, for example, you rely on the order of the columns in your local test database, then after deploying to the production database, a big mess can occur - the data will be written to the wrong columns or error will occur!
- The last reason is a bit psychological. By explicitly specifying attributes, you will better remember the structure of the tables. In other words, laziness doesn't pay off here.

(b) Insert the movie Terminator into the database. Find out the description and length of the film, for example, in the IMDB database[3]. Set the language of the movie to English, the standard rental period to 3 days and the price to 1.99. Other attributes will be left blank or set to the default value.

[3] `https://www.imdb.com/title/tt0088247/`

```
INSERT INTO film (title, description, language_id, rental_duration,
    rental_rate, length)
VALUES ('Terminator', 'A human soldier is sent from 2029 to 1984 to
   stop an almost indestructible cyborg killing machine, sent from
   the same year, which has been programmed to execute a young
   woman whose unborn son is the key to humanity future salvation.'
   , 1, 3, 1.99, 107);
```

In this task we will try the ordinary INSERT once again. Note that explicitly listing the attributes will make our work much easier - there are 14 attributes in the `film` table, while we have only filled in 6.

(c) Update the database so that actor Arnold Schwarzenegger plays in the movie Terminator. Find out the actor ID and film ID in advance by suitable queries.

Firstly, we need to find out the IDs assigned to the existing records of the film and actor. For example, we can use these two simple queries:

```
SELECT film_id
FROM film
WHERE title = 'Terminator';

SELECT actor_id
FROM actor
WHERE last_name = 'Schwarzenegger';
```

Suppose queries return the values `$x` and `$y`, respectively. We will use this notation in the following tasks to denote 'fictitious' variables. It is not part of the SQL syntax – it will only be an auxiliary notation for our purposes, in order to avoid specific constants that everyone in the database may have a little differently.

Note for more curious students: In databases that provide automatically generated ID usually contain also special functions for finding the last generated ID. Finding this last ID using queries like `SELECT MAX(id) FROM table` may not lead to the correct result. For example in Microsoft SQL Server, we can use the query `SELECT @@IDENTITY`, which returns the last generated ID for any table, or `SELECT IDENT_CURRENT('table name')`, which returns the last generated ID for a specific table. We present these functions here for completeness - they may be useful when you you will be developing a real information system.

It should be obvious that we will assign the actor to the film by inserting an entry into the `film_actor` table using the following command:

```
INSERT INTO film_actor (film_id, actor_id) VALUES ($x, $y);
```

(d) Set the Terminator movie in the 'Action' and 'Sci-Fi' categories. Find out the IDs of the relevant records in advance with suitable queries.

The solution of this task is similar to the previous task. The ID of the film, which we will mark as `$x`, we will find out, for example, using query:

```
SELECT film_id
FROM film
WHERE title = 'Terminator';
```

The easiest way to find IDs of the 'Action' and 'Sci-Fi' categories is to select the complete `category` table. Suppose the IDs of the required categories are `$y` and `$z`.

```
SELECT *
FROM category;
```

Consquently, we assign the film to the given categories using the commands:

```
INSERT INTO film_category (film_id, category_id) VALUES ($x, $y);
INSERT INTO film_category (film_id, category_id) VALUES ($x, $z);
```

(e) Put the Terminator movie in the 'Comedy' category. Solve the task using one command with subqueries. You have to avoid manually writing constants for IDs of movie and category. Find the required IDs using (`film.title` and `category.name`).

```
INSERT INTO film_category (film_id, category_id) VALUES
(
  (SELECT film_id FROM film WHERE title = 'Terminator'),
  (SELECT category_id FROM category WHERE name = 'Comedy')
);
```

While for previous tasks could by solved by several commands (find out the individual IDs in advance), here it is required to solve the task with one command (i.e. with one press of F5). It should not be surprise that DML statements (e.g. INSERT) are very often combined with SELECT queries. In this case, we will use subqueries instead of constants for the movie ID and category ID. Evidently, it is necessary that both of these subqueries return exactly one value – e.g., we can not have two movies named 'Terminator' in the database.

(f) Set the rental price of the Terminator movie to 2.99. At the same time update the `last_update` attribute to the current timestamp.

```
UPDATE film
SET rental_rate = 2.99, last_update = CURRENT_TIMESTAMP
WHERE title = 'Terminator';
```

The UPDATE statement is another DML command. This command changes the value of one or more attributes for the selected rows. The `WHERE` clause works similarly to the `SELECT` command. To get the current date and time, we can use the standard built-in function `CURRENT_TIMESTAMP`.

Another way is to build a condition based on the movie ID and subquery:

```
UPDATE film
SET rental_rate = 2.99, last_update = CURRENT_TIMESTAMP
WHERE film_id = (SELECT film_id FROM film WHERE title = 'Termintor');
```

This solution would of course be correct, but the result would be the same as in the previous case. But there will be a small difference between the commands - you know what?

Finally, one big warning! The `WHERE` clause is optional for the UPDATE statement. If we will not specify it or forget it, the values in the whole table will be updated. And this can be a big problem in a real production database!

2. (a) Create employees with your name and address (address information can of course be fictional). The username will be your login and you will be included in the warehouse with ID = 2. Find out the necessary constants for foreign keys in advance with suitable queries.

   We include this task here mainly for practicing the INSERT statement. Before an employee will be created, we must firstly create an address for him. Obviously, the address must related to a city and it must be located in a country. As a result, we will insert records into the tables `staff`, `address`, most likely into `city` and possibly into `country`.

   The task should therefore be solved in the following order:

   i. Firstly, we will find out whether there is a record of our country in the database (the Czech Republic and Slovakia are in the database). We can do it using the query:

   ```
   SELECT *
   FROM country
   ORDER BY country;
   ```

   Let's mark the detected country ID as `$x`.

   ii. Consequently, we will find out if our city is located in the given country:

   ```
   SELECT *
   FROM city
   WHERE country_id = $x
   ORDER BY city;
   ```

   iii. If not (i.e. if you are not from Olomouc), you need to insert a record:

   ```
   INSERT INTO city (city, country_id)
   VALUES ('Ostrava', $x);
   ```

   iv. If we inserted the record, it is necessary to remember the ID of the city – variable `$y`.

   ```
   SELECT *
   FROM city
   WHERE city = 'Ostrava';
   ```

   v. Only now can we insert the address:

   ```
   INSERT INTO address (address, district, city_id, phone)
   VALUES ('Testova␣123', 'Okres␣Ostrava', $y, '+420␣601␣001␣001');
   ```

   vi. We find the ID of the inserted address (`$z`):

   ```
   SELECT *
   FROM address
   WHERE address = 'Testova␣123'
   ```

   vii. Finally, we can insert the employee himself:

   ```
   INSERT INTO staff (first_name, last_name, address_id, store_id,
      username)
   VALUES ('Jan', 'Novak', $z, 2, 'nov001');
   ```

   (b) Create the address of our university in the database.

   Since you probably already have a record for the city of Ostrava in the database (after solving the previous task), let's just remember the ID of our city in the variable `$x`.

```
SELECT *
FROM city
WHERE city = 'Ostrava';
```

Using this ID insert the address:

```
INSERT INTO address (address, district, city_id, phone)
VALUES ('17. listopadu 2172/15', 'Okres_Ostrava', $x, '+420 597 326
    001');
```

(c) Create a new store at our university address. You will be the manager in the new store. Using simple queries, we firstly find out our ID and the address ID of our university (variables `$x` and `$y`).

```
SELECT *
FROM staff;

SELECT *
FROM address
WHERE address = '17. listopadu 2172/15';
```

The following INSERT should not be a problem for us:

```
INSERT INTO store (manager_staff_id, address_id)
VALUES ($x, $);
```

(d) For each movie that the rental company owns in at least one copy, move its copy with the highest ID to the new store (see previous task).

Firstly, as usual, we find out the ID of the store. Let's call it `$s`. Since we know that there are only few stores, we can use following trivial query to get it:

```
SELECT *
FROM store
```

Select the appropriate copies of the movies will be more complex query. The most important is know how to put together a query that returns the IDs of the last copies of movies (`inventory_id`):

```
SELECT i1.inventory_id
FROM inventory i1
WHERE i1.inventory_id >= ALL(
  SELECT i2.inventory_id
  FROM inventory i2
  WHERE i1.film_id = i2.film_id
)
```

If the query structure is not clear, return to the task 14 on the page 50.

We can then very easily include the query in the `WHERE` condition of the UPDATE statement. The solution of the problem could therefore look like this:

```
UPDATE inventory
SET store_id = $s
WHERE inventory_id IN (
  SELECT i1.inventory_id
  FROM inventory i1
  WHERE i1.inventory_id >= ALL(
    SELECT i2.inventory_id
    FROM inventory i2
```

```
      WHERE i1.film_id = i2.film_id
    )
  )
```

It means we update all records whose IDs fall (construction `IN`) into the set returned by the subquery.

The previous solution is correct, however, we can more simplify the query by integrating the query logic directly into the UPDATE itself:

```
UPDATE inventory
SET store_id = $s
WHERE inventory_id >= ALL(
  SELECT i2.inventory_id
  FROM inventory i2
  WHERE inventory.film_id = i2.film_id
)
```

Finally, let's look at the UPDATE syntax which can contain a `FROM` clause. The syntax is quite useful, but unfortunately specific to Microsoft SQL Server:

```
UPDATE i1
SET store_id =$s
FROM inventory i1
WHERE i1.inventory_id >= ALL(
  SELECT i2.inventory_id
  FROM inventory i2
  WHERE i1.film_id = i2.film_id
)
```

The `UPDATE` and `SET` clauses come from the UPDATE command. The rest (from the `FROM` clause) you already know from a classic SELECT query. Note that if we assign an alias to a table after `FROM`, we must use that alias after the `UPDATE` keyword instead of the original table name.

3. Increase the rental price of all films with the actor ZERO CAGE by 10 %. Solve the task with one command without writing the constant for the actor ID (the actor will be identified by his name).

```
UPDATE film
SET rental_rate = rental_rate * 1.1
WHERE film_id IN (
  SELECT film_id
  FROM
    film_actor
    JOIN actor ON film_actor.actor_id = actor.actor_id
  WHERE first_name = 'ZERO' AND last_name = 'CAGE'
);
```

Firstly, we have to construct the query, which returns the ID of the films with the actor ZERO CAGE. Above this query, we then build the UPDATE command, which multiplies the rental price for the selected movies by a constant 1.1. It should be clear that this multiplication represents an increase of 10 %.

Note that in the SET clause we can easily refer to the original values of the record (i.e. the original rental_rate). To find out the original rental_rate, which we want to multiply by 1.1, it is definitely not necessary to write a subquery.

4. Set the original language to NULL to all movies whose original language (original_language) is Mandarin. Avoid select the ID for the language in separate query.

```
UPDATE film
SET original_language_id = NULL
WHERE original_language_id =
  (SELECT language_id FROM language WHERE name = 'Mandarin');
```

We choose the task to practice the UPDATE statement using a subquery. We can find out the ID of the Mandarin language by subquery. For movies in this language, we will set original_language_id to NULL.

5. For each film with GROUCHO SINATRA, insert one new copy into the inventory table. All these new copies will be placed in the store with ID = 2. Leave the date of the last update of the record at the default value. Solve the task again with one command without writing the constant for the actor ID (the actor will be identified by his name).

```
INSERT INTO inventory (film_id, store_id)
SELECT film_id, 2
FROM
  actor
  JOIN film_actor ON actor.actor_id = film_actor.actor_id
WHERE first_name = 'GROUCHO' AND last_name = 'SINATRA';
```

While in previous tasks we wrote the INSERT statement with the VALUES clause (i.e. we listed specific values), this task shows that the INSERT command can also be written using the SELECT query. In this way, we can very easily insert a large number of records into the table at once. This task also shows that INSERT always does not insert only a single records.

6. Delete the Mandarin language from the database. Solve the task only after solving the task 4.

```
DELETE FROM language
WHERE name = 'Mandarin';
```

The DELETE commands is the last of the DML statements. Its syntax is similar to the UPDATE command - the basis is again to set the condition WHERE correctly. Remember, WHERE can be omitted, but it results in deleting the complete content of the table (it will delete the content not the table itself).

In summary - there are 3 standard DML commands for editing data in the database:

- INSERT – inserts new rows into the table,
- UPDATE – updates existing records, i.e. changes the values of their attributes,

- `DELETE` – deletes rows from the table.

Note: Microsoft SQL Server provides also the command `MERGE`, but we will not show it here.

7. Delete the Terminator movie from the database (solve the task after solving the example 1). Is it possible to solve this task only by deleting the appropriate record from the `film` table?

   The command to delete the movie 'Terminator' looks very simple:

```
DELETE
FROM film
WHERE title = 'Terminator';
```

   However, if you solved the task 1, the command will not run (you will receive an error message). It should be clear that the problem is that there are records that refer to the movie by a foreign key. In this case, these are records in the `film_actor` and `film_category` tables. Therefore, you must delete those records in advance using the following two commands:

```
DELETE
FROM film_actor
WHERE film_id = (SELECT film_id FROM film WHERE title = 'Terminator');


DELETE
FROM film_category
WHERE film_id = (SELECT film_id FROM film WHERE title = 'Terminator');
```

   Only then it will be possible to run DELETE over the `film` table. However, in the task 20 we will show that there is a so-called *cascade deletion*, where the child records will be deleted automatically.

8. Delete all inactive customers from the database.

   In this task we solve a similar problem as in the previous task. We cannot delete a customer as long as some rentals or payments are linked to him. We cannot delete loans as long as they are referenced by payments (the payment always refers to the customer and usually also to the loan – see database model on page 4).

   So we need to start by deleting payments that refer to inactive customers either through the loan or directly (condition `OR` in the following order):

```
DELETE
FROM payment
WHERE
  rental_id IN
  (
    SELECT rental.rental_id
    FROM customer JOIN rental ON customer.customer_id = rental.customer_id
    WHERE customer.active = 0
  )
  OR customer_id IN
  (
    SELECT customer_id
    FROM customer
```

```
    WHERE active = 0
);
```

Consequently, we can delete the loans:

```
DELETE
FROM rental
WHERE customer_id IN (SELECT customer_id FROM customer WHERE active = 0);
```

At this point, there is no payment linked to inactive customers, so we will execute DELETE record over the`customer` table:

```
DELETE
FROM customer
WHERE active = 0;
```

9. Add the optional integer attribute `inventory_count` to the `movie` table. Set this attribute for all movies to the number of copies of that movie (the number of matching records in the `inventory` table).

   So far, we have not changed the structure of the database. Although we changed the content of the tables, the structure itself (tables, their columns, relationships, etc.) remained the same. The commands in the DDL category are used to modify the structure itself.

   ```
   ALTER TABLE film
   ADD inventory_count INT;
   ```

   The ALTER TABLE statement is a typical representative of DDL statements. In this case, we use the `ADD` clause to say that we want to add a new column with a certain name and data type – in this case `INT`, which represents an integer in a certain range. We will not list data types here - you can find it, for example, in the documentation of the used DBMS [4].

   The second part of the task is here mainly to practice the UPDATE statement and also to remind the aggregation functions:

   ```
   UPDATE film
   SET inventory_count = (
     SELECT COUNT(*)
     FROM inventory
     WHERE inventory.film_id = film.film_id
   )
   ```

10. Edit the `name` attribute in the `category` table to string of variable length 50 characters.

    ```
    ALTER TABLE category
    ALTER COLUMN name VARCHAR(50);
    ```

    This is a quite common modification that needs to be made when a customer complains that the required text does not fit in the field. The `ALTER COLUMN` clause is used to modify the column definition (data type, data type range, mandatory, etc.). Let us note, for example, in Oracle DBMS, `MODIFY` is written instead of `ALTER COLUMN`.

[4] https://docs.microsoft.com/en-us/sql/t-sql/data-types/data-types-transact-sql

11. Add the mandatory text attribute `phone` with a maximum length of 20 characters to the `customer` table. Set the phone according to the `phone` attribute, which is part of the customer's address.

```
ALTER TABLE customer
ADD phone VARCHAR(20) NOT NULL;
```

We specify the mandatory attribute by writing `NOT NULL` ater the data type (in this case a string with a variable length and specification of the maximum number of characters). However, if we try to run the command, we find that the DBMS reports an error. The problem is that the attribute should be mandatory but at the same time it is not clear how its value should be set for records that already exist in the table.

There are two possible solutions: (1) set the attribute to the default value, which we will show later (see task 12), or (2) create the attribute as optional, set its value for all records with the UPDATE command and finally change the attribute to mandatory. So let's try the second option.

Firstly, we add the optional `phone` attribute. We specify an optional attribute by specifying `NULL` or nothing after the data type instead of `NOT NULL`:

```
ALTER TABLE customer
ADD phone VARCHAR(20);
```

Consequently, we update the value of this attribute for all records:

```
UPDATE customer
SET phone = (
  SELECT phone
  FROM address
  WHERE address.address_id = customer.address_id
)
```

Finally, we can modify the attribute to make it mandatory:

```
ALTER TABLE customer
ALTER COLUMN phone VARCHAR(20) NOT NULL;
```

12. Add the mandatory attribute `create_date` to the `rental` table, whose default value will be the current timestamp.

So here we will show the second way to add a mandatory attribute to a non-empty table. We will add an attribute with a default value.

You can specify the default value simply by adding the keyword `DEFAULT` after the data type and `NOT NULL`. To get the current date and time, use the `CURRENT_TIMESTAMP` function:

```
ALTER TABLE rental
ADD create_date DATETIME NOT NULL DEFAULT CURRENT_TIMESTAMP;
```

This solution is fine, but for a reason, which we will show in the next task, it could be improved a bit. The specification of the default value is one of the so-called integrity constraints and each integrity constraint should have a name. If we do not specify a name, the DBMS will choose a name itself, and it will be a partially random combination of

characters and numbers. Let's learn to name integrity constraints explicitly by specifying the keyword CONSTRAINT followed by the name (e.g. DF_rental_create_date):

```
ALTER TABLE rental
ADD create_date DATETIME NOT NULL
  CONSTRAINT DF_rental_create_date DEFAULT CURRENT_TIMESTAMP;
```

13. Drop the attribute `rental.create_date` created in the previous task.

    We also work with the ALTER TABLE statement in this task, this time using the DROP COLUMN clause. The following command is syntactically correct, but we receive an error after running it.

    ```
    ALTER TABLE rental
    DROP COLUMN create_date;
    ```

    The problem is that the column is bound by an integrity constraint, which we named DF_rental_create_date. Before droping a column, you must delete this integrity constraint using the following command:

    ```
    ALTER TABLE rental
    DROP CONSTRAINT DF_rental_create_date;
    ```

    And this is an example why it is appropriate to name integrity constraints. Otherwise, we would have to find out what name the system generated for the integrity constraint in advance.

14. Add the optional attribute `creator_staff_id` to the table `film`, which will be a foreign key to the table `staff`. Name the foreign key `fk_film_staff`.

    Generally, we can solve this problem in two ways. Either we add a column first and then make it a foreign key, or we add a column with a foreign key setting with just one command.

    So let's try the first option and add the `creator_staff_id` column:

    ```
    ALTER TABLE film
    ADD creator_staff_id TINYINT NULL;
    ```

    Let's note that we used TINYINT as the data type, which is an integer with a smaller range than INT. This is because the foreign key must always have exactly the same data type (including the range if we consider e.g. VARCHAR) as the corresponding primary key. For example, by looking at the column list of the table staff in Microsoft SQL Server Management Studio (Object Explorer panel), we can make sure that the primary key staff_id is actually of the data type TINYINT (see Image 2).

    Now we create a foreign key from the new column with command:

    ```
    ALTER TABLE film
    ADD FOREIGN KEY (creator_staff_id) REFERENCES staff (staff_id);
    ```

    We will use ALTER TABLE again because we are modifying the table structure. After the ADD FOREIGN KEY clause, we write into parentheses which attributes represent the foreign key, and after REFERENCES, which table and which attributes this foreign key

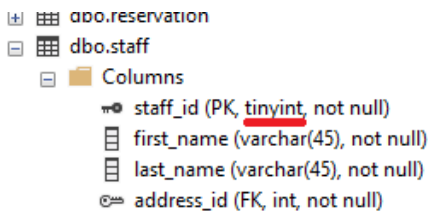


Figure 2: Finding the data type of the primary key `staff_id` in the table `staff`

refers to. The number of attributes in both parentheses must be the same. If we refer to a simple primary key (i.e. represented by one attribute), the foreign key is also simple. But there are situations where we have to refer to a composite primary key - then the foreign key will also be composite.

The foreign key is another of the integrity constraints that we should name. This means that we should rather remember the following notation, where we also set the name of the foreign key (`FK_film_staff`):

```
ALTER TABLE film
ADD CONSTRAINT FK_film_staff FOREIGN KEY (creator_staff_id) REFERENCES
    staff (staff_id);
```

As we mentioned earlier, we can also add a column directly by making it a foreign key:

```
ALTER TABLE film
ADD creator_staff_id TINYINT NULL CONSTRAINT FK_film_staff FOREIGN KEY
    REFERENCES staff (staff_id);
```

Finally, let's see probably the most effective solution of this task:

```
ALTER TABLE film
ADD creator_staff_id TINYINT NULL REFERENCES staff;
```

15. Set check of the attribute `staff.email` so that the email value always will contain the character '@' followed by the character '.'.

```
ALTER TABLE staff
ADD CONSTRAINT check_email CHECK (email LIKE '%@%.%');
```

We come to another of the integrity constraints – `CHECK`. With this integrity constraint, we can specify a logical condition that each record in a particular table must satisfy. If we try to add or modify any record that violate the given condition (e.g. the e-mail will not contain '@'), the given command will end with an error. Likewise, it should be clear that it is not possible to create an integrity constraint `CHECK` over a table with non-empty content, where some records do not satisfy the specified condition.

16. Drop the check constaint cread in the previous task.

```
ALTER TABLE staff
DROP CONSTRAINT check_email;
```

If we have properly named the integrity constraint, there will be no problem with removing it. We proceed the same way as in the case of deleting a foreign key or default value.

17. Set the loan check so that the return date is always greater than the loan date.

```
ALTER TABLE rental
ADD CONSTRAINT check_dates CHECK (return_date > rental_date)
```

The solution of the task is similar to the task 15. We only show here that the condition can work simultaneously with more attributes from the given table. We might wonder if a condition can contain a subquery. In this case, unfortunately not. Database triggers can be used for more complex checks but we will not practise them in this subject.

18. Create a new table `reservation`, i.e. a table of reservations, with the automatically generated primary key `reservation_id` of the data type integer. The table will also contain the following attributes: mandatory reservation date `reservation_date` with a the current date as a default value, mandatory reservation end date `end_date`, mandatory customer ID `customer_id` as a foreign key to the table `customer`, mandatory movie ID `movie_id` as foreign key to table `film` and optional employee ID `staff_id` as foreign key to table `staff`.

```
CREATE TABLE reservation
(
  reservatoin_id TINYINT IDENTITY PRIMARY KEY NOT NULL,
  reservation_date DATE NOT NULL,
  end_date DATE NOT NULL,
  customer_id INT CONSTRAINT fk_reservation_customer FOREIGN KEY
    REFERENCES customer (customer_id),
  film_id INT CONSTRAINT fk_reservation_film FOREIGN KEY
    REFERENCES film (film_id),
  staff_id TINYINT CONSTRAINT fk_reservation_staff FOREIGN KEY
    REFERENCES staff (staff_id)
);
```

In this task, we finally get to the very important CREATE TABLE statement, which we use to create a completely new table. In parentheses after `CREATE TABLE` we list the columns, including their data type and other integrity restrictions. The syntax for individual columns is similar to adding columns with the `ALTER TABLE ... ADD` command. The only novelty here is the specification of the primary key using `PRIMARY KEY`, preceded by the keyword `IDENTITY`. This means that the primary key will be generated automatically.

Note that this method of defining the automatically generated key is specific to Microsoft SQL Server. For example, other systems use other keywords (`AUTO_INCREMENT` for MySQL or `COUNTER` for Microsoft Access) or so-called sequences (Oracle, PostgreSQL or, more recently, Microsoft SQL Server).

19. Insert some two records in the table created in the previous task. Then delete the second record. What ID will be assigned to the next inserted record?

So let's insert some two records:

```
INSERT INTO reservation (reservation_date, end_date, customer_id, film_id,
    staff_id)
VALUES ('2020-10-02', '2020-10-05', 25, 10, 1);

INSERT INTO reservation (reservation_date, end_date, customer_id, film_id,
    staff_id)
VALUES ('2020-10-02', '2020-10-15', 56, 78, 2);
```

Of course, we just need to make sure that the foreign key constants used actually refer to existing records, otherwise the commands will raise an error.

Now let's look at the IDs of the inserted records:

```
SELECT *
FROM reservation
```

Let the ID of the second record be `$x` (most likely `$x` = 2). So let's try to delete this record:

```
DELETE FROM reservation
WHERE reservatoin_id = $x
```

Finally, we insert another new record and find out what ID was assigned to it:

```
INSERT INTO reservation (reservation_date, end_date, customer_id, film_id,
    staff_id)
VALUES ('2020-10-20', '2020-10-21', 5, 64, 2);

SELECT *
FROM reservation
```

Somebody may be surprised that the ID of the last inserted record is not `$x`, but it is increased by 1. This is because the automatic generator will never assign a new record an ID that has been used in the past. Can you guess the reason of this behavior?

20. (a) Create a table `review` with the attributes `film_id` and `customer_id` representing foreign keys in the tables `film` and `customer`. Both of these attributes will together represent a composite primary key. The table will also contain mandatory attribute `stars`, which will take integers in the interval $\langle 1, 5 \rangle$, and optional attribute `actor_id`, which will be a foreign key to the table `actor`. Ensure that in the case you delete a customer or movie, all related records in the table `review` will be also automatically deleted. Also, make sure that when you delete an actor, for related records in table `review` will be `actor_id` set to `NULL`.

```
CREATE TABLE review
(
  film_id INT NOT NULL
    CONSTRAINT fk_review_film
    FOREIGN KEY REFERENCES film (film_id) ON DELETE CASCADE,
  customer_id INT NOT NULL
    CONSTRAINT fk_review_customer
    FOREIGN KEY REFERENCES customer (customer_id) ON DELETE CASCADE,
  stars TINYINT NOT NULL
    CONSTRAINT ch_review_stars
    CHECK (stars BETWEEN 1 AND 5),
```

```
    actor_id INT NULL
      CONSTRAINT fk_review_actor
      FOREIGN KEY REFERENCES actor (actor_id) ON DELETE SET NULL,
    PRIMARY KEY (film_id, customer_id)
)
```

In this task, we will show you how to create a new table again. In comparison to the task 18, there are several differences. The primary key is not composed of one, but of several attributes - specifically, the attributes `film_id` and `customer_id`. In this case, we can not write the keyword `PRIMARY KEY` directly after the attribute (or someone might think to write `PRIMARY KEY` after both attributes - that's syntactically wrong), but we have to write it separately below the column list.

Furthermore, we are tasked with the specific behavior of individual relationships. The following three modifiers can be part of a relationships and they determine what happens if a record is deleted from the referenced table:

i. `ON DELETE NO ACTION` is the default option, i.e. if the deleting record is referenced by another record, deleting will not be allowed (see task 7).

ii. `ON DELETE CASCADE` says that records that refer to a deleted record will be automatically deleted as well – so-called cascading deletion.

iii. `ON DELETE SET NULL` says that the foreign key referring to the deleted record will be set to `NULL`. Of course, this option only makes sense if the given foreign key is not a mandatory attribute.

In our case, we set the modifier `ON DELETE CASCADE` for the foreign keys `film_id` and `customer_id`. That means if a movie or customer is deleted, then all reviews (records in the table `review`) that refer to the movie or customer will be automatically deleted as well. Using the texttt ON DELETE SET NULL modifier, we specified that when we delete an actor, all in related reviews the attribute `actor_id` will be set to `NULL`.

Finally, let's see a more general and universal notation for the CREATE TABLE statement. It looks like we first specify the individual columns and consequently we write the individual integrity constraints separately. The following more general notation is therefore equivalent to the previous notation:

```
CREATE TABLE review
(
  film_id INT NOT NULL,
  customer_id INT NOT NULL,
  stars TINYINT NOT NULL,
  actor_id INT NULL,
  PRIMARY KEY (film_id, customer_id),
  CONSTRAINT fk_review_film
    FOREIGN KEY (film_id) REFERENCES film (film_id) ON DELETE CASCADE
      ,
  CONSTRAINT fk_review_customer
    FOREIGN KEY (customer_id) REFERENCES customer (customer_id) ON
      DELETE CASCADE,
  CONSTRAINT fk_review_actor
    FOREIGN KEY (actor_id) REFERENCES actor (actor_id) ON DELETE SET
      NULL,
  CONSTRAINT ch_review_stars CHECK (stars BETWEEN 1 AND 5)
```

```
)
```

(b) Insert two records in the table `review`:

- Review of the movie ARMY FLINTSTONES by the customer BRIAN WYMAN - 4 stars, without mentioning the actor.
- Review of the movie ARSENIC INDEPENDENCE by the customer CHERYL MURPHY – 5 stars mentioning actor EMILY DEE.

At the beginning, let's solve the 'more annoying' part, i.e. find out the ID for the mentioned films, actors and customers. We will not write specific questions here for their simplicity, the constants should be as follows:

film	ARMY FLINTSTONES	`film_id`	= 40
film	ARSENIC INDEPENDENCE	`film_id`	= 41
customer	BRIAN WYMAN	`customer_id`	= 318
customer	CHERYL MURPHY	`customer_id`	= 59
actor	EMILY DEE	`actor_id`	= 148

INSERT statements should then look like this:

```
INSERT INTO review (film_id, customer_id, stars, actor_id)
VALUES (40, 318, 4, NULL);

INSERT INTO review (film_id, customer_id, stars, actor_id)
VALUES (41, 59, 5, 148);
```

Note: Customer BRIAN WYMAN and actor EMILY DEE are chosen in this role on purpose since they are not referenced in any other records (except our records in `review`).

(c) Delete customer BRIAN WYMAN and actor EMILY DEE from the database. Then look at the content of the table `review`.

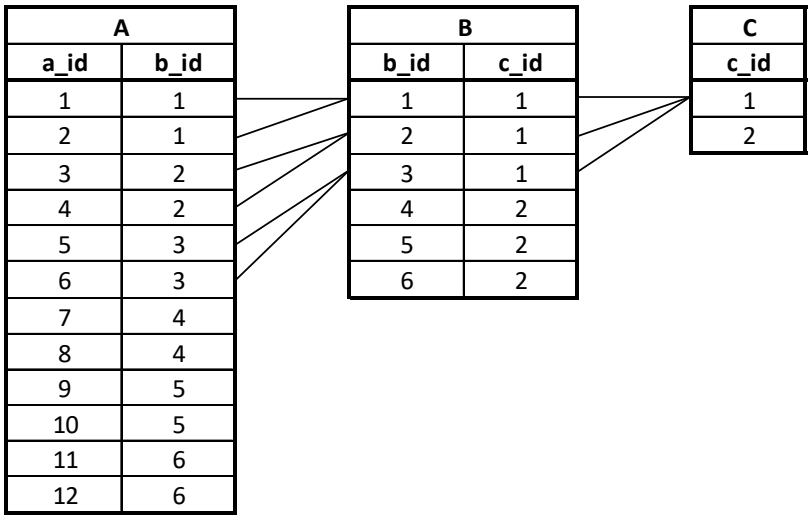
```
DELETE FROM customer
WHERE customer_id = 318

DELETE FROM actor
WHERE actor_id = 148;
```

The commands for deleting the relevant records themselves will not surprise anyone. However, we should be aware of the difference in comparsion with the task 7 task, where deletion was not possible until the record was referenced by other records. However, using modifiers `ON DELETE CASCADE` (between the evaluation and the customer) and `ON DELETE SET NULL` (between the evaluation and the actor) in this case, it is not a problem to run the above commands. The review made by the customer with ID 318 (BRIAN WYMAN) will be automatically deleted, and in the review of the actor with ID 148 (EMILY DEE) will be `actor_id` set to `NULL`. Make sure of that.

Let's try to demonstrate cascade delete with one more example and explain where the name 'cascade' actually came from. Let's have tables `A(a_id, b_id),B(b_id,`

c_id) and C(c_id), where A.a_id, B.b_id and C.c_id are the primary keys in the individual tables and A.b_id, B.c_id are the foreign keys. Cascade delete will be set for foreign keys A.b_id and B.c_id (i.e. ON DELETE CASCADE). Content of the sample table is shown in Figure 3. If we delete the first record from the table C (i.e. c_id = 1), we will automatically delete the related records from the table B (b_id $\in \{1, 2, 3\}$) and related records from the table A (a_id $in\{1, 2, 3, 4, 5, 6\}$) . We see that the records will be deleted 'in cascade order'.



A	
a_id	b_id
1	1
2	1
3	2
4	2
5	3
6	3
7	4
8	4
9	5
10	5
11	6
12	6

B	
b_id	c_id
1	1
2	1
3	1
4	2
5	2
6	2

C
c_id
1
2

Figure 3: Example of cascade delete

There is one warning concluding from the example in Figure 3. Cascade delete is 'a good servant but a bad lord'. We can make our work easier by not having to think about manually deleting related records. On the other hand, as we can see in the picture, by carelessly setting up cascade delete, we can very easily inadvertently delete the content of a large part of the database.

21. Back up the content of the table film to the new table film_backup. The new table will be identical in structure with the table film but it will not contain primary or foreign key settings. In other words, attributes like film_id, language_id will be common integer (non-key) attributes.

    We will show two solutions for this task. The first one you should be able to put together now, and the second one that is surprisingly very simple. So let's start with the first one with the CREATE TABLE statement to create a table film_backup with the same structure as the table film:

```
CREATE TABLE film_backup
(
  film_id INT,
  title VARCHAR(255),
  description TEXT,
  release_year VARCHAR(4),
  language_id TINYINT,
  original_language_id TINYINT,
  rental_duration TINYINT,
```

```
  rental_rate DECIMAL(4, 2),
  length SMALLINT,
  replacement_cost DECIMAL(5, 2),
  rating VARCHAR(10),
  special_features VARCHAR(255),
  last_update DATETIME
);
```

Consequently, we use the INSERT INTO command, where we use `SELECT` instead of `VALUES`, copy all the movies to a new table:

```
INSERT INTO film_backup (
  film_id, title, description, release_year, language_id,
  original_language_id, rental_duration, rental_rate,
  length, replacement_cost, rating, special_features, last_update)
SELECT
  film_id, title, description, release_year, language_id,
  original_language_id, rental_duration, rental_rate,
  length, replacement_cost, rating, special_features, last_update
FROM film
```

And now the second, more economical, solution:

```
SELECT * INTO film_backup
FROM film
```

Using this trivial and very useful command, we can directly create a new table from the result of any query. Unfortunately, the syntax of the statement is specific to Microsoft SQL Server, however, there are other similar constructs in other databases.

22. Drop tables the `review` and `film_backup` created in the previous two tasks.

    In the case of dropping the entire table from the database, use the DROP TABLE command. Therefore, dropping the tables `film_backup` and `review` will look like this:

```
DROP TABLE film_backup;
DROP TABLE review;
```

    We may notice that in order to delete the table, it is not necessary to first delete the included integrity constraints (e.g. default values, foreign keys, etc.). On the other hand, we will not be able to delete a table if it is referenced by a foreign key from another table, as we will show in the last task 24.

    To summarize, at this point we already know all DDL operations with the table:

    - `CREATE TABLE table` – creates a new table,
    - `ALTER TABLE table` – modifies the structure of a table, where
      - `ADD column` – adds a column,
      - `ALTER COLUMN column` – modifies a column,
      - `DROP COLUMN column` – drops a column,
      - `ADD CONSTRAINT` – adds integrity restriction (`DEFAULT`, `CHECK`, `FOREIGN KEY`, `PRIMARY KEY`),
      - `DROP CONSTRAINT` – drops integrity restriction,

- `DROP TABLE table` – drops an existing table.

We should be aware of the differences between the DML commands `INSERT`, `UPDATE`, `DELETE` and the DDL commands `CREATE`, `ALTER`, `DROP`. While the first group of statements modifies the content of the tables, the second group modifies the structure of the tables.

23. Create a table `rating` with attributes `rating_id` - integer automatically generated primary key, `name` - mandatory string with a maximum of 10 characters and `description` - an optional string with an unlimited number of characters. Select unique values from the attribute `rating` located in the table `film` and insert them into the new table. Create a mandatory attribute `rating_id` in the table `movie`, which will be a foreign key to the newly created table `rating`. The values in this attribute will be set according to the attribute `rating`. Finally, delete the original attribute `rating`.

    This task demonstrates the solution to a relatively common problem. The attribute `rating` in the table `film` contains a few unique values – movie categories. In order to add a description to each category, we decided to create a separate category codebook into the database and instead of the original attribute `film.rating` register a foreign key `film.rating_id` referring to this codebook.

    So let's start by creating a table representing a new codebook:

```
CREATE TABLE rating
(
  rating_id TINYINT NOT NULL IDENTITY PRIMARY KEY,
  name VARCHAR(10) NOT NULL,
  description TEXT NULL
);
```

    Now, using the `INSERT` statement with the `SELECT` clause, we insert all categories into the codebook with one command. We must not forget to mention `DISTINCT` after `SELECT`, otherwise the new table `rating` would contain duplicate categories.

```
INSERT INTO rating (name)
SELECT DISTINCT rating
FROM film;
```

    Consequently, we add a foreign key to the table `movie` referencing to the new table `rating`. The foreign key will be optional for now, we will fill it later.

```
ALTER TABLE film
ADD rating_id TINYINT NULL CONSTRAINT fk_film_rating FOREIGN KEY
   REFERENCES rating (rating_id);
```

    So now we set the value of the foreign key by searching for the corresponding category for each movie – i.e. where `rating.name = film.rating`:

```
UPDATE film
SET rating_id = (
  SELECT rating_id
  FROM rating
  WHERE rating.name = film.rating
);
```

At this point, the foreign key is set for all movies and we can change the attribute `rating_id` to mandatory:

```
ALTER TABLE film
ALTER COLUMN rating_id TINYINT NOT NULL;
```

Finally, we remove the original attribute `film.rating`:

```
ALTER TABLE film
DROP COLUMN rating;
```

Execution of this command most likely results in an error because there are two integrity constraints associated with this attribute. How do we find out what these integrity constraints are? Probably the fastest way is to read the error message, which should look like Figure 4.

```
Msg 5074, Level 16, State 1, Line 1
The object 'DF__film__rating__59063A47' is dependent on column 'rating'.
Msg 5074, Level 16, State 1, Line 1
The object 'CHECK_special_rating' is dependent on column 'rating'.
Msg 4922, Level 16, State 9, Line 1
ALTER TABLE DROP COLUMN rating failed because one or more objects access this column.
```

Figure 4: Error message in attempt to delete column `film.rating`

Let's try to look at error messages as useful advice. In our case, we have the integrity constraints `DF__film__rating__59063A47` (default value) and `CHECK_special_rating` (check valid values). Therefore, we delete these integrity constraint with the following commands:

```
ALTER TABLE film
DROP CONSTRAINT DF__film__rating__59063A47;

ALTER TABLE film
DROP CONSTRAINT CHECK_special_rating;
```

We can now go back and delete the column `film.rating` and the task is done.

24. Drop all tables from the database.

    In this very last task we will learn to clean up after ourselves, we will practice the commands `DROP TABLE` and `DROP CONSTRAINT` and we will mention the so-called system catalog. There are generally two ways to solve a problem: (1) drop the tables in order so that we never drop the table referenced by the foreign key, or (2) drop all foreign keys from the database first, and then all the tables.

    Let's start with the first option and follow the E-R database diagram in Figure 1 (page 5). Tables that will certainly not be referenced by foreign keys are e.g. `film_actor` and `film_category`. So let's start with:

    ```
    DROP TABLE film_actor;
    DROP TABLE film_category;
    ```

    Consequently, we can delete the actors and categories, because they are no longer referenced by any foreign key at this time:

    ```
    DROP TABLE actor;
    DROP TABLE category;
    ```

We will not show the complete solution here - with the help of the E-R diagram, everyone certainly understands how to continue. Unfortunately, over time we will get into a situation where simply choosing the right order will not be enough. For example, the tables `store` and `staff`, where one refers to the other and vice versa (`store.manager_staff_id` and `staff.store_id`). For similar cases, we have to remove the foreign keys first. In this case, the following commands are used:

```
ALTER TABLE staff
DROP CONSTRAINT fk_staff_store;


ALTER TABLE store
DROP CONSTRAINT fk_store_staff;
```

To clarify, removing a foreign key does not delete the attribute. We will only remove a certain special property of the attribute - i.e. the attribute will continue to exist, but it will not be a foreign key (its value will not be checked or restricted in any way).

Let's return to the second option - i.e. remove all foreign keys first and then all tables. Of course, there is a practical problem here – where can we find a list of all foreign keys? If we do not want to manually 'click' the tree with tables (Object Explorer in Microsoft SQL Server Management Studio), we can use so-called system catalog. The possibility of using this catalog is presented here for the sake of interest – it is not a part of the subject. The system catalog is a collection of some system tables that contains metadata about the tables themselves.

The following query over the `INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS` and `INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE` tables that are part of the system catalog returns table names and foreign key names. Please note that the catalog looks a little different in each DBMS – the solution will be specific for Microsoft SQL Server:

```
SELECT ctu.TABLE_NAME, rc.CONSTRAINT_NAME
FROM
  INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS rc
  JOIN INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE ctu ON
    rc.CONSTRAINT_NAME = ctu.CONSTRAINT_NAME
```

By appropriate modification of the `SELECT` clause, we can select the `ALTER TABLE ... DROP CONSTRAINT` statements directly, which we can then simply copy from the result, paste as a script and execute (see Figure 5).

```
SELECT 'ALTER␣TABLE␣' + ctu.TABLE_NAME + '␣␣DROP␣CONSTRAINT␣' + rc.
   CONSTRAINT_NAME
FROM
  INFORMATION_SCHEMA.REFERENTIAL_CONSTRAINTS rc
  JOIN INFORMATION_SCHEMA.CONSTRAINT_TABLE_USAGE ctu ON
    rc.CONSTRAINT_NAME = ctu.CONSTRAINT_NAME
```

Finally, we can list the commands for deleting all tables that are no longer referenced by any foreign keys. We will use another of the system catalog tables – `INFORMATION_SCHEMA.TABLES`

```
SELECT 'DROP␣TABLE␣' + TABLE_NAME
FROM INFORMATION_SCHEMA.TABLES
WHERE TABLE_TYPE = 'BASE␣TABLE'
```

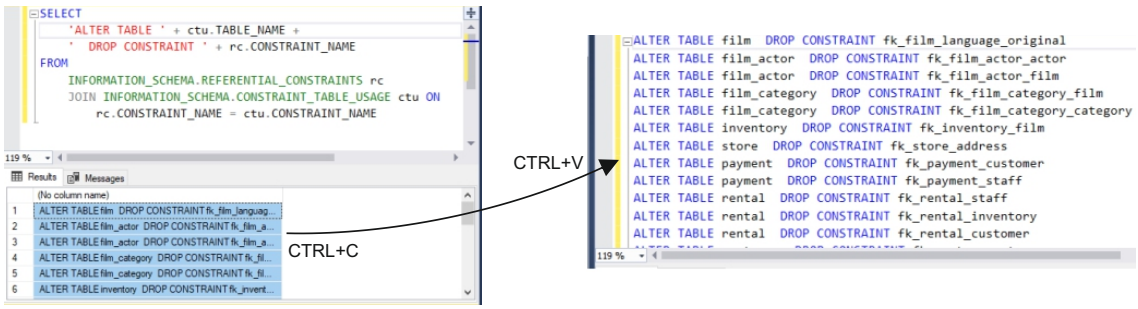


Figure 5: Example of using the system catalog