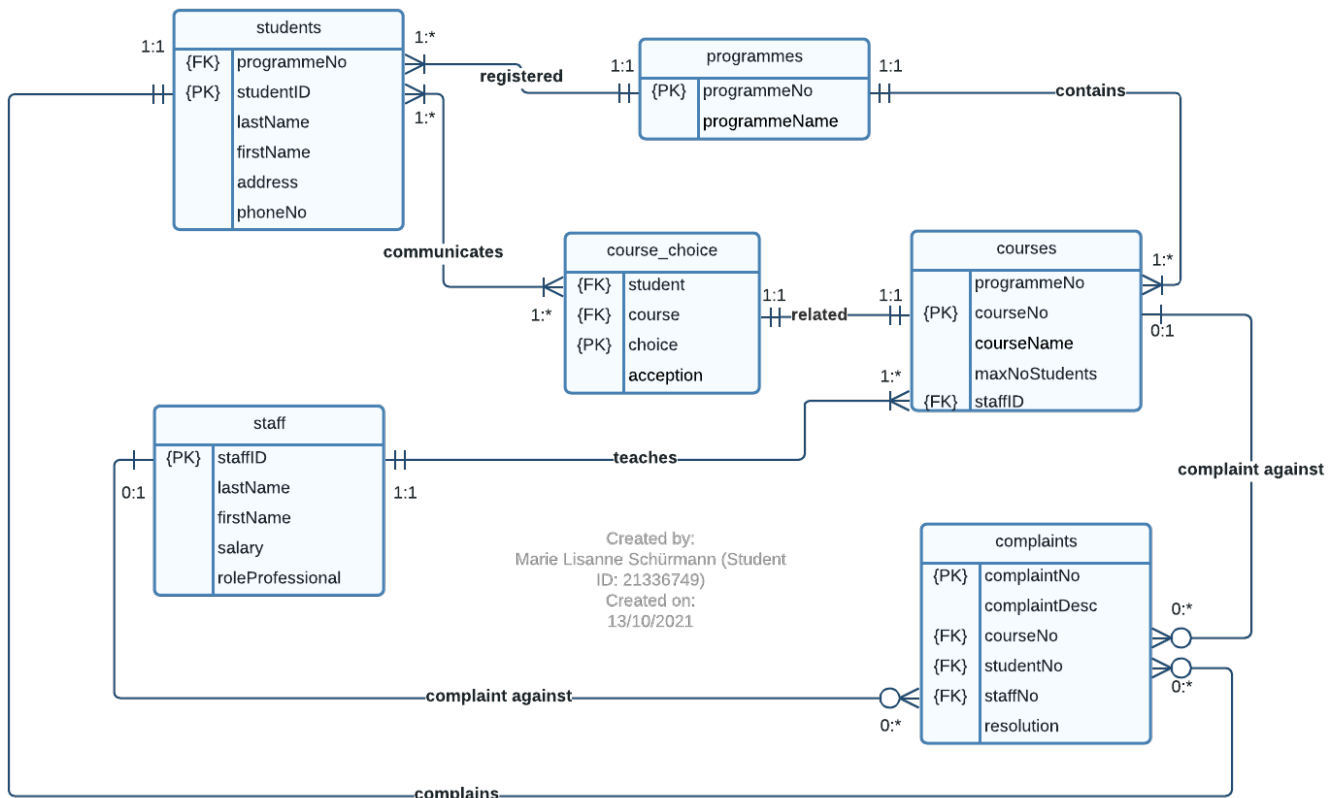


Individual Assignment

Part 1 – Question 1

The following image presents the Entity-Relationship-Model visualisation of the database design of an university as described in the exercise.



I created a table named “**students**” to store all student records. It will be mainly used and data will be filled in by the registration office. It has a primary key “studentID”. Each student must register for a programme. Each student will also have a first and a last name. Also, the registration office will store their address and phone number for communication purposes. There could be added more values as birth date but I tried to keep it straight-forward as those attributes would not affect other tables of the database. The programmeNo is chosen as a foreign key to avoid redundant data; the programme name must not be stored in two tables, so it is simpler to handle changes within the database. Through the foreign key “programmeNo” the programme name can be identified easily. A student must choose only one programme (One-to-One-relationship) but one programme can have many students (One-to-Many-relationship).

I created a table named “**programmes**” to store the programmes available in the university. The programmeNo is the primary key and will be used for identification in other tables such as “students”. The programmes table stores the programmeName as the most necessary attribute. The table will be used by the programs office (to connect courses to programmes) and the registration office (to register students in programmes).

I created a table named **"courses"** to store the courses available in the university. The primary key in this table is courseNo. It will be a foreign key in other tables (e.g. course_choice and complaints) to avoid redundant data and create unique identification. Each course is connected through the foreign key "programmeNo" to the programme table. Each course can be dependent to one programme (One-to-One-relationship) but one programme can have six courses (One-to-Many-relationship). I did not mention a One-to-Six-relationship precisely on purpose, in case that this might be changing in the future and more modules as electives could be added to a programme. Each course will have a course name and needs to store the maximum number of students. Also there will be a staff assigned through the foreign key staffID to each course. Each course can only have one lecturer (One-to-One-relationship) but one lecturer can teach in more than one course (One-to-Many-relationship). Through the staffID the lecturer's name can be identified.

I created a table named **"course_choice"** to solve the Many-to-Many-relationship between students and courses. Each student can have many courses and at least one. Each course can have many students. As in many universities, students can communicate their preferred course choice. Based on the maximum number of seats available, the university's program office has to decide which student will get access to the course in case there are more students applying than seats available. In this case, students enrolled in the connected programme will be given prioritised access before students visiting the course just by interest. Therefore this table will be used by the Registration office. As a primary key I identified the "choice" which is an auto-incremented number for each student's choice communicated to the office. This will always include studentID and the courseNo of choice. One student can have many course choices (One-to-Many-relationship) but one course choice can only store one studentID (One-to-One-relationship). The same applies for the foreign key courseNo: one course choice can only refer to one course (One-to-One-relationship) but all available courses can be included in course choices (One-to-Many-relationship). Also the table will store the decision if the student's choice got already approved or not.

For the HR purposes, I created a table named **"staff"** to store all data of staff, professional and academic staff. It does not make sense to split the data into two tables because it would unnecessarily complicate the database. Most of attributes are identically for teaching and non-teaching staff. I decided to identify the non-teaching staff by adding the professional_role attribute which is NULL when it is a teaching staff. Through a view, this can be made visible more easily for the HR office. The primary key is "staffID" which will be used in other tables as courses or complaints as a foreign key. Each staff will have a first and a last name. Each staff will have a salary.

I created a table named **"complaints"** to store all complaints made by students at the complaints office. Each complaint will get a complaintNo as a primary key. The complaint will be described in "ComplaintDesc". Also there is an attribute compulsory to get the status of the complaint's resolution which is by default "open". There are three foreign keys included in this table:

- StudentID: One student can make many complaints (One-to-Many-relationship) but one complaint is being connected to only one student (One-to-One-relationship).

- StaffID: A complaint can be generally made against many staff members (One-to-Many-relationship) but in one complaint, there will be only one staff member responsible (One-to-One-relationship).
- CourseNo: A complaint can be generally made against many courses (One-to-Many-relationship) but in one complaint, there will be only one course responsible (One-to-One-relationship).

All tables comply with the three normal forms:

- First normal form (1NF): Each data field represents only one value. Address is represented as one data field for simplicity reasons, it may be preferable to divide it into street, house number, city, ZIP code, country, etc. depending on the database usage. In the university case, address is only necessary to contact the students; therefore I used one field to hold all information for simplicity and I do not expect further analytics based on student's residence.
- Second normal form (2NF): The tables avoid redundant data by not having partial dependencies on each other.
- Third normal form (3NF): The primary key as the unique identifier is the only key that other information are dependent on, e.g. courses does hold the ID No. as foreign keys for staff but not the staff's last name and first name as it would be redundant data.

Overall, the solution is workable and elegant because it does not have any redundant data in any table within the database. Also, it solves problems for the Many-to-Many-relationship between student's and courses and simplifies the conversion into MySQL. Through the usage of foreign keys, tables are not affected by changes in other tables which reduces the bureaucracy workload for staff as well as the error likelihood through double entries. All information needs presented in the exercise are fulfilled but it is not too beyond of a university's requirements.

Part 1 – Question 2

Please find the code for the MySQL database creation in the following pages along with the explanation of my decisions. The outcomes including the first five entries per database will be included in the end after entering data.

Please note that I will include explanation for the usage of each command only once if the reasoning is the same in other tables.

First, I created a database if it not already exists named "university". Through the "USE" command, I start the editing of the database:

```
# MySQL Script created by Marie Schürmann, Student ID No.
21336749
# Trinity Business School -- M.Sc. Business Analytics 21/22
# Individual Assignment in BU7141: Business Data Management
and Visualisation
# Date of SQL Script Creation: 15/10/2021
-----
```

```
# creation of a database for the university case
CREATE DATABASE IF NOT EXISTS university;
USE university;
```

Secondly, I started to create the table **programmes**. The primary key programmeNo is used in more than one other table as a foreign key; it is easier to program the tables in an order that does not prevent to create foreign keys because the related table does not exist yet. So I decided to start with the table that does not have any foreign keys and is one of the tables most referred to.

The programmeNo will be an integer value (`int`) that cannot be null (`NOT NULL`) because it is declared as the primary key (`PRIMARY KEY (programmeNo)`) and is generated automatically by the last value + 1 through `AUTO_INCREMENT`. This simplifies their job for the Programme Office by searching a not used programme number and avoids doubled or missing entries.

The programmeName is in text format (`varchar`) and can have maximum 255 characters. This is the saved storage by MySQL in the database for each value. The maximum for `varchar` would be 65,535 characters. Probably in the programmes name's example, it could be less by around 40 characters as most programmes names are not too long but 255 is kind of a default value for shorter texts and names, so I tried to comply with the standard. I used `varchar(255)` in other tables for attributes with character format for the same reason. The name cannot be null (`NOT NULL`) because every programme should have a name for clarification.

```
# creation of a table containing all the programmes
CREATE TABLE programmes (
    programmeNo int NOT NULL AUTO_INCREMENT,
    programmeName varchar(255) NOT NULL,
    PRIMARY KEY (programmeNo)
);
```

In the third step, I created the table for **students**. The primary key's attributes are the same for every table. It is an integer value (`int`) that cannot be null (`NOT NULL`) because it is declared as the primary key (`PRIMARY KEY`) and is generated automatically by the last value + 1 through `AUTO_INCREMENT`.

The student's first and last name will be stored in two separate attributes as it should comply with the first normal form. The address is not split up for simplicity as already explained in Q1's answer. I'm aware of the fact that it might be best practice to split it up but I decided to represent the SQL database in the same way of the ER diagram.

The phone number is an integer value because country's dial code should be typed in as numbers only (Irish numbers) or as "00" instead of "+" to comply with ITU-recommendation that is followed by the EU members (international numbers). The programmeNo that is included in the student records is a foreign key related to the programmes table created before.

All attributes have the `NOT NULL` statement because student records should be complete.

```
# creation of a table containing all the students
immatriculated in the university
CREATE TABLE students (
    studentID int NOT NULL AUTO_INCREMENT,
    programmeNo int NOT NULL,
    lastName varchar(255) NOT NULL,
    firstName varchar(255) NOT NULL,
    address varchar(255) NOT NULL,
    phoneNo int NOT NULL,
    PRIMARY KEY (studentID),
    FOREIGN KEY (programmeNo)
        REFERENCES programmes(programmeNo)
);
```

In the fourth step, I created the table for **staff**. The primary key's attributes are the same as before in the students' and programme's ones.

I introduced the `DEFAULT NULL` command in this table to assign staff usually to be a academic staff if not stated differently. This enables the HR department to not enter any value if it is an academic staff member and reduces the data input in the table.

Also, like in the tables before, I decided to include `NOT NULL` to attributes that should be filled out, as every staff should have a salary and a name.

```
# creation of a table containing all staff (teaching and non-
teaching)
CREATE TABLE staff (
    staffID int NOT NULL AUTO_INCREMENT,
    lastName varchar(255) NOT NULL,
    firstName varchar(255) NOT NULL,
    salary int NOT NULL,
    roleProfessional varchar(255) DEFAULT NULL,
    PRIMARY KEY (staffID)
);
```

In the fifth step, I created the table for **courses**.

I did not introduce any new commands in this table but I would like to get into more details for the foreign key commands. The foreign key connects the attribute included before to an attribute (primary key) of a different table in the same database:

```
FOREIGN KEY ([column name in this table]) REFERENCES [other's
table name]([primary key in other table] )
```

It is a useful command to include because it is used to maintain referential integrity within the database, e.g. you cannot delete tables that are necessary for other's table as a foreign key because it would result in errors in the database.

```
# creation of a table containing all the courses
CREATE TABLE courses (
    courseNo int NOT NULL AUTO_INCREMENT,
    courseName varchar(255) NOT NULL,
    programmeNo int NOT NULL,
```

```
        staffID int NOT NULL,  
        maxNoStudents int NOT NULL,  
        PRIMARY KEY (courseNo),  
FOREIGN KEY (programmeNo)  
            REFERENCES programmes(programmeNo),  
FOREIGN KEY (staffID)  
            REFERENCES staff(staffID)  
    );
```

In the sixth step, I created the table for **course_choice**. This table is very helpful to simplify the Many-to-Many relationship because it can be evaluated which student is in which course if his choice is accepted. In a view, the field for acceptance should only allow the values "accepted" or "decision open" or "not accepted". The configuration through the CHECK command in My SQL that would allow to control the values entered, can be defined but does not operate correctly. It would need a bigger workaround. Therefore, a solution over a view might be the more straight-forward solution for this problem.

```
# creation of a table containing the course choices students  
made  
CREATE TABLE course_choice (  
    choice int NOT NULL AUTO_INCREMENT,  
    student int NOT NULL,  
    course int NOT NULL,  
    acceptance varchar(255) NOT NULL,  
    PRIMARY KEY (choice),  
    FOREIGN KEY (student)  
        REFERENCES students(studentID),  
    FOREIGN KEY (course)  
        REFERENCES courses(courseNo)  
);
```

In the seventh step, I created the table for **complaints**. This table is based on the same commands as before explained.

```
# creation a table containing all complaints and their  
statuses  
CREATE TABLE complaints (  
    complaintID int NOT NULL AUTO_INCREMENT,  
    complaintDesc varchar(255) NOT NULL,  
    studentNo int NOT NULL,  
    courseNo int DEFAULT NULL,  
    staffNo int DEFAULT NULL,  
    resolution varchar(255) DEFAULT "open",  
    PRIMARY KEY (complaintID),  
    FOREIGN KEY (studentNo)  
        REFERENCES students(studentID),
```

```
FOREIGN KEY (courseNo)
REFERENCES courses(courseNo),
FOREIGN KEY (staffNo)
REFERENCES staff(staffID)
);
```

In the eighth step, I started to fill in the first five data entries for each table. I used the command `INSERT INTO [table name] ([table column(s)] VALUES ([value])`. Also I used `SELECT * FROM [database].[table]` to view my entries. In include the screenshots resulting from the SELECT command below.

```
# filling values into the tables
# university programmes:
INSERT INTO programmes (programmeName)
VALUES ("Business Analytics");
INSERT INTO programmes (programmeName)
VALUES ("Finance");
INSERT INTO programmes (programmeName)
VALUES ("Management");
INSERT INTO programmes (programmeName)
VALUES ("Marketing");
INSERT INTO programmes (programmeName)
VALUES ("Entrepreneurship");

# see information of programmes in MySQL workbench:
SELECT * FROM university.programmes;
```

97 # see information of programmes in MySQL workbench:
98 • **SELECT * FROM university.programmes;**

100% 37:98

Result Grid Filter Rows: Search Edit: Export/In

	programmeNo	programmeName
▶	1	Business Analytics
▶	2	Finance
▶	3	Management
▶	4	Marketing
▶	5	Entrepreneurship
	NULL	NULL

```
# students:
INSERT INTO students (programmeNo, lastName, firstName,
address, phoneNo)
VALUES (2, "Granger", "Hermione Jane", "6 Willifield Way,
London, UK", 12345);
INSERT INTO students (programmeNo, lastName, firstName,
address, phoneNo)
```

```
VALUES (1, "Potter", "Harry James", "4 Privet Drive,
Little Whinging, UK", 447923);
INSERT INTO students (programmeNo, lastName, firstName,
address, phoneNo)
VALUES (4, "Weasley", "Ronald Bilius", "The Burrow,
Ottery St. Catchpole, UK", 42938);
INSERT INTO students (programmeNo, lastName, firstName,
address, phoneNo)
VALUES (5, "Lovegood", "Luna", "Lovegood House, Ottery
St. Catchpole, UK", 54878);
INSERT INTO students (programmeNo, lastName, firstName,
address, phoneNo)
VALUES (3, "Malfoy", "Draco", "Malfoy Manor, Wiltshire,
UK", 345776);

# see information of students in MySQL workbench:
SELECT * FROM university.students;
```

112 # see information of students in MySQL workbench:
113 • **SELECT * FROM university.students;**

100% 35:113

Result Grid Filter Rows: Search Edit: Export/In

	studentID	programmeNo	lastName	firstName	address
▶	1	2	Granger	Hermione Jane	6 Willifield Way, London, UK
	2	1	Potter	Harry James	4 Privet Drive, Little Whinging, UK
	3	4	Weasley	Ronald Bilius	The Burrow, Ottery St. Catchpole, UK
	4	5	Lovegood	Luna	Lovegood House, Ottery St. Catchpole, UK
	5	3	Malfoy	Draco	Malfoy Manor, Wiltshire, UK
	NULL	NULL	NULL	NULL	NULL

```
# staff:
INSERT INTO staff (lastName, firstName, salary)
VALUES ("McGonagall", "Minerva", 35000);
INSERT INTO staff (lastName, firstName, salary)
VALUES ("Snape", "Severus", 35000);
INSERT INTO staff (lastName, firstName, salary,
roleProfessional)
VALUES ("Filch", "Argus", 13000, "Caretaker");
INSERT INTO staff (lastName, firstName, salary)
VALUES ("Lupin", "Remus", 25000);
INSERT INTO staff (lastName, firstName, salary,
roleProfessional)
VALUES ("Pomfrey", "Poppy", 25000, "Nurse");

# see information of staff in MySQL workbench:
SELECT * FROM university.staff;
```



```
127      # see information of staff in MySQL workbench:
128 •    SELECT * FROM university.staff;
```

100% 32:128

Result Grid Filter Rows: Search Edit:

	staffID	lastName	firstName	salary	roleProfessional
▶	1	McGonagall	Minerva	35000	NULL
	2	Snape	Severus	35000	NULL
	3	Filch	Argus	13000	Caretaker
	4	Lupin	Remus	25000	NULL
	5	Pomfrey	Poppy	25000	Nurse
	NULL	NULL	NULL	NULL	NULL

```
# courses:
INSERT INTO courses (courseName, programmeNo, staffID,
maxNoStudents)
VALUES ("International Financial Statement Analysis", 2,
4, 30);
INSERT INTO courses (courseName, programmeNo, staffID,
maxNoStudents)
VALUES ("International Entrepreneurship", 5, 2, 15);
INSERT INTO courses (courseName, programmeNo, staffID,
maxNoStudents)
VALUES ("Machine Learning", 1, 1, 40);
INSERT INTO courses (courseName, programmeNo, staffID,
maxNoStudents)
VALUES ("Project Management", 3, 2, 50);
INSERT INTO courses (courseName, programmeNo, staffID,
maxNoStudents)
VALUES ("Marketing Management", 3, 4, 20);

# see information of courses in MySQL workbench:
SELECT * FROM university.courses;
```

```
142      # see information of courses in MySQL workbench:
143 •    SELECT * FROM university.courses;
```

100% 34:143

Result Grid Filter Rows: Search Edit: Export/

	courseNo	courseName	programmeNo	staffID	maxNoStudents
▶	1	International Financial Statement Analysis	2	4	30
	2	International Entrepreneurship	5	2	15
	3	Machine Learning	1	1	40
	4	Project Management	3	2	50
	5	Marketing Management	3	4	20
	NULL	NULL	NULL	NULL	NULL

```
# course_choice:
INSERT INTO course_choice (student, course, acceptance)
VALUES (2, 2, "approved");
INSERT INTO course_choice (student, course, acceptance)
VALUES (3, 5, "approved");
INSERT INTO course_choice (student, course, acceptance)
VALUES (2, 3, "approved");
INSERT INTO course_choice (student, course, acceptance)
VALUES (1, 5, "no");
INSERT INTO course_choice (student, course, acceptance)
VALUES (4, 2, "no");

# see information of course courses in MySQL workbench:
SELECT * FROM university.course_choice;
```

157 # see information of course courses in MySQL workbench:

158 • **SELECT * FROM university.course_choice;**

100% 40:158

Result Grid Filter Rows: Search Edit: Export/

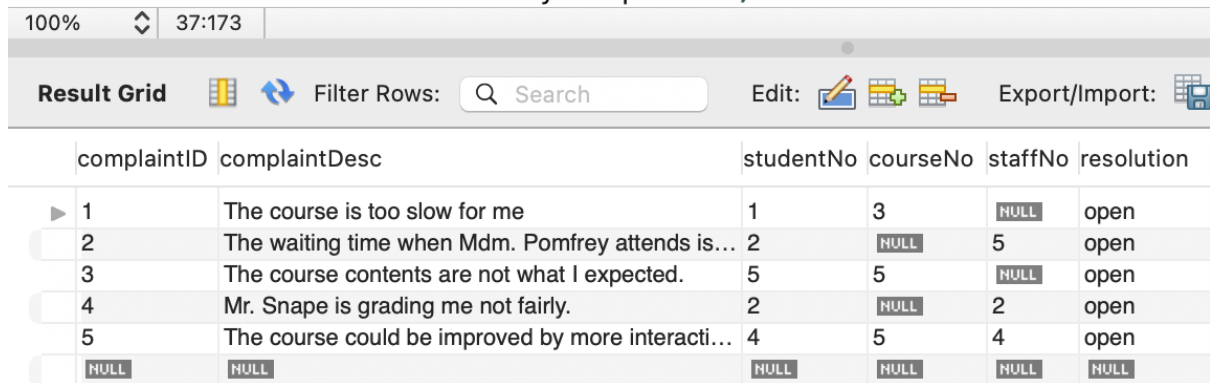
	choice	student	course	acceptation
▶	1	2	2	approved
	2	3	5	approved
	3	2	3	approved
	4	1	5	no
	5	4	2	no
	NULL	NULL	NULL	NULL

```
# complaints:
INSERT INTO complaints (complaintDesc, studentNo, courseNo)
VALUES ("The course is too slow for me", 1, 3);
INSERT INTO complaints (complaintDesc, studentNo, staffNo)
VALUES ("The waiting time when Mdm. Pomfrey attends is
too long", 2, 5);
INSERT INTO complaints (complaintDesc, studentNo, courseNo)
VALUES ("The course contents are not what I expected.",
5, 5);
INSERT INTO complaints (complaintDesc, studentNo, staffNo)
VALUES ("Mr. Snape is grading me not fairly.", 2, 2);
INSERT INTO complaints (complaintDesc, studentNo, courseNo,
staffNo)
VALUES ("The course could be improved by more interactive
contents.", 4, 5, 4);

# see information of complaints in MySQL workbench:
SELECT * FROM university.complaints;
```

172 # see information of complaints in MySQL workbench:

173 • **SELECT * FROM** university.complaints;



complaintID	complaintDesc	studentNo	courseNo	staffNo	resolution
1	The course is too slow for me	1	3	NULL	open
2	The waiting time when Mdm. Pomfrey attends is...	2	NULL	5	open
3	The course contents are not what I expected.	5	5	NULL	open
4	Mr. Snape is grading me not fairly.	2	NULL	2	open
5	The course could be improved by more interacti...	4	5	4	open
NULL	NULL	NULL	NULL	NULL	NULL

The database is a true representation of the ER diagram because it is created out of the exactly same tables and attributes and relationships (super keys and foreign keys) as presented above in Q1. It works in the way it was expected. All tables are in third normal form as already planned in Q1.

Part 2 – Question 1

Note: I assume that the unit for money in the database chinook (e.g. column invoice.Total) is kEUR.

a) Show the artists with number of albums they have in descending order.

The table “album” contains the foreign key ArtistId which is the primary key of the table “artist” which contains the Name attribute. Therefore, if I want to show the artists by name, I have to inner join both tables by the ArtistId. Also, I have to count the number of albums each artist has released and I renamed it as “Count of Albums” for better readability. In the final step, I grouped it by the artist name and ordered it in descending order by the count of albums, as requested in the task.

```
SELECT Name, count(AlbumId) AS "Count of Albums"
FROM album
INNER JOIN artist
ON album.ArtistId = artist.ArtistId
GROUP BY Name
ORDER BY count(AlbumId) DESC;
```

	Name	Count of Albums
▶	Iron Maiden	21
▶	Led Zeppelin	14
▶	Deep Purple	11
▶	Metallica	10
▶	U2	10
▶	Ozzy Osbourne	6
▶	Pearl Jam	5
▶	Various Artists	4
▶	Faith No More	4

I found out that the artist who has the most albums within the database of the music company, is Iron Maiden with 21 albums. He is followed by Led Zeppelin (14 albums) and Deep Purple (11 albums).

b) Show name and address of customer with highest total sales across the complete invoice table.

The primary key of the table "customer" is CustomerId which is a foreign key in the "invoice" table. Therefore, I can inner join them in order to create a table by CustomerId. The joined table has the name and address of customer from the customer table and the sum of all invoices from the invoice table. Within this, I can select only the Top 1 (LIMIT 1) customer sorted by descending sum of sales.

```
SELECT customer.CustomerId AS "Customer ID",
CONCAT(customer.FirstName, " ", customer.LastName) AS
"Customer", CONCAT(customer.Address, ", ", City, ",
",Country) AS "Contact Address", SUM(invoice.Total) AS
Sales
FROM customer
INNER JOIN invoice Using(CustomerId)
GROUP BY customer.CustomerId
ORDER BY Sales DESC
LIMIT 1;
```

	Customer ID	Customer	Contact Address	Sales
▶	6	Helena Holý	Rilská 3174/6, Prague, Czech Republic	49.62

I found out that the customer Helena Holy from Prague, Czech Republic, has highest sales of 49.62 kEUR in the database.

c) Find the name of the tracks that occur most frequently in playlist TV Shows and playlist 90's music.

The table "playlisttrack" lists all the tracks and their connection to playlists, where I can find the count of each track occurring how many times in the playlists TV Shows and 90's music. The foreign key TrackId makes it possible to join the table with the table "track" which holds each tracks name. The foreign key PlaylistId enables me to join the table with "playlist" which holds each playlists name. The solution to get the most frequently occurred tracks, is

to group them by the tracks ID and get the count of no. of PlaylistId per TrackId. I also ordered it in descending order by its count, so the most frequent mentioned tracks are at the top of the list. There are two ways to find it out: in two tables (one for TV shows and one for 90's music) or both in one. I included both solutions in the following code:

```
# Find the name of the tracks that occur most frequently  
in playlist TV Shows:
```

```
SELECT track.Name, count(playlisttrack.TrackId) AS  
"Occurrences in Playlist TV Shows"  
FROM playlisttrack  
INNER JOIN track ON playlisttrack.TrackId = track.TrackId  
INNER JOIN playlist ON playlisttrack.PlaylistId =  
playlist.PlaylistId  
WHERE playlist.Name = "TV Shows"  
GROUP BY playlisttrack.TrackId  
ORDER BY count(TrackId) DESC;
```

Name	Occurrences in Playlist TV Shows
► Battlestar Galactica: The Story So Far	2
Occupation / Precipice	2
Exodus, Pt. 1	2
Exodus, Pt. 2	2
Collaborators	2
Torn	2
A Measure of Salvation	2
Hero	2
Unfinished Business	2

```
# Find the name of the tracks that occur most frequently  
in playlist 90's music:
```

```
SELECT track.Name, count(playlisttrack.TrackId) AS  
"Occurrences in Playlist 90's music"  
FROM playlisttrack  
INNER JOIN track ON playlisttrack.TrackId = track.TrackId  
INNER JOIN playlist ON playlisttrack.PlaylistId =  
playlist.PlaylistId  
WHERE playlist.Name = "90's music"  
GROUP BY playlisttrack.TrackId  
ORDER BY count(TrackId) DESC;
```

Name	Occurrences in Playlist 90's m...
► Fast As a Shark	1
Restless and Wild	1
Princess of the Dawn	1
Walk On Water	1
Love In An Elevator	1
Rag Doll	1
What It Takes	1
Dude (Looks Like A Lady)	1
Janie's Got A Gun	1

```
# Find the name of the tracks that occurred most  
frequently in either one of the both playlists:
```

```
SELECT track.Name, count(playlisttrack.TrackId) AS  
"Occurrences in playlists 90's music or TV Shows"  
FROM playlisttrack  
INNER JOIN track ON playlisttrack.TrackId = track.TrackId  
INNER JOIN playlist ON playlisttrack.PlaylistId =  
playlist.PlaylistId  
WHERE (playlist.Name = '90's music') OR (playlist.Name =  
'TV Shows')  
GROUP BY playlisttrack.TrackId  
ORDER BY count(TrackId) DESC;
```

Name	Occurrences in playlists 90's music or TV Sh...
▶ Battlestar Galactica: The Story So Far	2
Occupation / Precipice	2
Exodus, Pt. 1	2
Exodus, Pt. 2	2
Collaborators	2
Torn	2
A Measure of Salvation	2
Hero	2
Unfinished Business	2

I found out that no track is standing out for occurring much higher than others in any of the two playlists.

Though, I could identify that the playlist TV shows has every track (e.g. "Battlestar Galactica: The Story So Far") occurring two times because there are two playlists (PlaylistId 3 and 10) which are called "TV Shows" with the same tracks occurring.

In the playlist 90 there is no track occurring two times because there is only one playlist 90's music and it has no duplicated tracks.

It has to be mentioned that I grouped by TrackId on purpose. During the explanatory analytics I performed on the tables, I could identify that various tracks are being duplicated by track.Name but do have two track.TrackId. The length of the songs did vary by some seconds which lets me assume that some components published different versions of their songs or songs were published on two albums. This needs further analysis and eventually, tracks can be grouped further to a combination of "one song-various versions" and therefore, the analysis of occurrences in playlists may result differently.

d) Show name of top 10 tracks by number of times they are present in different playlists i.e. a track is number 1 if it is present in maximum number of playlists.

I left joined the track table on the playlisttrack to keep all information that are in the playlisttrack by the key "TrackId". I grouped it by the TrackId to get one row per track. I also ordered it by the count of TrackId's because the table only contains the combination PlaylistId to TrackId descending. I counted the TrackId to get the number of occurrences in playlists.

```
SELECT track.Name, count(track.TrackId) AS "Number of  
Occurrences in Playlists"  
FROM playlisttrack  
LEFT JOIN track ON playlisttrack.TrackId = track.TrackId
```

```
GROUP BY playlisttrack.TrackId
ORDER BY count(playlisttrack.TrackId) DESC
LIMIT 10
;
```

Name	Number of Occurrences in Playlists
▶ Solomon HWV 67: The Arrival of the...	5
▶ Intoitus: Adorate Deum	5
▶ Die Walküre: The Ride of the Valkyries	5
▶ Suite for Solo Cello No. 1 in G Major,...	5
▶ Symphony No.5 in C Minor: I. Allegro...	5
▶ The Messiah: Behold, I Tell You a My...	5
▶ Miserere mei, Deus	5
▶ Nabucco: Chorus, "Va, Pensiero, Sull'...	5
▶ Aria Mit 30 Veränderungen, BWV 988...	5
▶ Koyaanisqatsi	5

All tracks that occurred most in playlists (in the TOP 10) occurred five times.

e) **Show the countries that the customers belong to with number of customers in each country listed in ascending order.**

For this task I did not need to join tables because all data was available in the table "customer". I counted the number of customers grouped by each country. Then, I ordered it in ascending order as requested.

```
SELECT Country, COUNT(CustomerId) AS "Number of
Customers"
FROM customer
GROUP BY Country
ORDER BY Count(CustomerId) ASC;
```

Country	Number of Custome...
▶ Norway	1
▶ Austria	1
▶ Belgium	1
▶ Denmark	1
▶ Finland	1
▶ Hungary	1
▶ Ireland	1
▶ Italy	1
▶ Netherlands	1
▶ Poland	1
▶ Spain	1
▶ Sweden	1
▶ Australia	1
▶ Argentina	1
▶ Chile	1
▶ Czech Re...	2
▶ Portugal	2
▶ India	2
▶ United Kin...	3
▶ Germany	4
▶ Brazil	5
▶ France	5
▶ Canada	8
▶ USA	13

I found out that more than half of the countries only have one customer. The USA is the country which has most customers (13) followed by Canada (8), France and Brazil (5 each).

Part 2 – Question 2

For managers of a music company trying to expand, there would be three dimensions especially important for the expansion plans:

- geographical areas that could generate more sales,
- genres that are profitable and could be invested in and
- reaching higher sales figures.

I found out the following arguments for the expansion strategy:

```
SELECT customer.Country, sum(invoice.Total)
  FROM invoice
 LEFT JOIN customer ON invoice.CustomerId =
customer.CustomerId
 GROUP BY customer.Country
 ORDER BY sum(invoice.Total) DESC;
```

Country	sum(invoice.Total)
▶ USA	523.06
Canada	303.96
France	195.10
Brazil	190.10
Germany	156.48
United Kingdom	112.86
Czech Republic	90.24
Portugal	77.24
India	75.26
Chile	46.62
Ireland	45.62
Hungary	45.62
Austria	42.62
Finland	41.62
Netherlands	40.62
Norway	39.62
Sweden	38.62
Belgium	37.62
Australia	37.62
Spain	37.62
Denmark	37.62
Italy	37.62
Poland	37.62
Argentina	37.62

First, the countries where the music companies earned the most were the USA by 523 kEUR, followed by Canada with 304 kEUR. As the company already achieves high volumes out of the market, it should continue to invest in the market.

```
SELECT BillingCountry,
       SUM(Total) AS "Sales",
       COUNT(DISTINCT CustomerId) AS "Number of Customers",
       (SUM(Total)/COUNT(DISTINCT CustomerId)) AS "Sales per
Customer",
       SUM(UnitPrice)/COUNT(UnitPrice) AS "Average Price of
Units",
       SUM(Quantity)/COUNT(UnitPrice) AS "Average Quantity
Purchased"
```



```
FROM invoice
INNER JOIN invoiceline ON invoice.InvoiceId =
invoiceline.InvoiceId
GROUP BY BillingCountry
ORDER BY (SUM(Total)/COUNT(DISTINCT CustomerId)) DESC;
```

BillingCountry	Sales	Number of Customers	Sales per Customer	Average Price of Units	Average Quantity Purchas...
► Hungary	446.62	1	446.620000	1.200526	1.0000
Ireland	446.62	1	446.620000	1.200526	1.0000
Czech Republic	879.24	2	439.620000	1.187368	1.0000
Chile	415.62	1	415.620000	1.226842	1.0000
Austria	404.62	1	404.620000	1.121579	1.0000
Norway	362.62	1	362.620000	1.042632	1.0000
USA	4667.06	13	359.004615	1.058826	1.0000
Netherlands	352.62	1	352.620000	1.068947	1.0000
Finland	350.62	1	350.620000	1.095263	1.0000
Germany	1392.48	4	348.120000	1.029474	1.0000
France	1722.10	5	344.420000	1.026842	1.0000
Portugal	687.24	2	343.620000	1.016316	1.0000
Sweden	340.62	1	340.620000	1.016316	1.0000
Canada	2689.96	8	336.245000	0.999868	1.0000
Brazil	1677.10	5	335.420000	1.000526	1.0000
Argentina	334.62	1	334.620000	0.990000	1.0000
Australia	334.62	1	334.620000	0.990000	1.0000
Belgium	334.62	1	334.620000	0.990000	1.0000
Denmark	334.62	1	334.620000	0.990000	1.0000
Italy	334.62	1	334.620000	0.990000	1.0000
Poland	334.62	1	334.620000	0.990000	1.0000
Spain	334.62	1	334.620000	0.990000	1.0000
United Kingdom	1003.86	3	334.620000	0.990000	1.0000
India	667.28	2	333.640000	1.017027	1.0000

Secondly, the TOP 3 countries with the best paying customers per purchase are Hungary, Ireland and Czech Republic, even if the music company has only one or two customers each in these countries. I advise the music managers to invest into the higher paying countries as they are still countries with potential for growth. Another argument to keep investing into these countries is that the average price per unit is also higher, so customers in these countries pay more for a track. That means that the music company should try to enforce their advertising in those countries, trying to gain new customers (market penetration marketing strategy).

```
SELECT
    genre.Name,
    customer.Country,
    COUNT(invoice.InvoiceId) AS Purchases
FROM invoice
    JOIN customer ON invoice.CustomerId =
customer.CustomerId
    JOIN invoiceline ON invoiceline.Invoiceid =
invoice.InvoiceId
    JOIN track ON track.TrackId = invoiceline.Trackid
    JOIN genre ON track.GenreId = genre.GenreId
WHERE (customer.Country = "Hungary" OR
customer.Country = "Ireland" OR customer.Country = "Czech
Republic")
GROUP BY genre.Name, customer.Country
```

```
ORDER BY Purchases DESC
LIMIT 10
;
```

Name	Country	Purchases
▶ Rock	Czech Republic	25
Rock	Ireland	12
Rock	Hungary	11
Alternative & Punk	Czech Republic	9
Latin	Czech Republic	9
TV Shows	Czech Republic	8
TV Shows	Ireland	7
Latin	Ireland	7
Metal	Czech Republic	6
Drama	Czech Republic	6

Thirdly, the managers of the music company may be unsure in which genres they should invest to successfully enter the markets in Europe (Hungary, Czech Republic and Ireland). As especially Rock is successful genre in all three those markets, they should focus on Rock as the main genre to have an advantage in all European target markets.

```
SELECT
    genre.Name,
    COUNT(invoice.InvoiceId) AS Purchases,
    SUM(invoice.Total) AS Sales,
    SUM(invoice.Total)/COUNT(invoice.InvoiceId) AS "Sales per
Purchase"
FROM invoice
    JOIN customer ON invoice.CustomerId =
customer.CustomerId
    JOIN invoiceline ON invoiceline.Invoiceid =
invoice.InvoiceId
    JOIN track ON track.TrackId = invoiceline.Trackid
    JOIN genre ON track.GenreId = genre.GenreId
GROUP BY genre.Name
ORDER BY Sales DESC;
```

Name	Purchases	Sales	Sales per Purchase
▶ Rock	835	7720.02	9.245533
Latin	386	3472.55	8.996244
Metal	264	2093.13	7.928523
Alternative & Punk	244	1961.66	8.039590
TV Shows	47	817.71	17.398085
Jazz	80	746.46	9.330750
Drama	29	544.61	18.779655
Blues	61	429.66	7.043607
R&B/Soul	41	338.62	8.259024
Reggae	30	332.64	11.088000
Classical	41	317.04	7.732683
Soundtrack	20	242.55	12.127500
Pop	28	239.75	8.562500
Alternative	14	211.17	15.083571
Sci Fi & Fantasy	20	198.87	9.943500
World	13	182.18	14.013846
Hip Hop/Rap	17	166.41	9.788824
Heavy Metal	12	161.37	13.447500
Electronica/Dance	12	149.62	12.468333
Easy Listening	10	138.60	13.860000
Comedy	9	112.30	12.477778
Science Fiction	6	102.41	17.068333
Bossa Nova	15	86.13	5.742000
Rock And Roll	6	83.16	13.860000

As a fourth step, the development that Rock is the most purchased genre can be even found in the overall sales. Rock is by far the most successful with 835 purchases, sold at 7720 kEUR

in total, then followed by Latin Music, Metal and Alternative and Punk Music. The music company already has success within those genres. The customers seem to enjoy the tracks offered which lets me assume that the music company has made good purchase decisions within these genres. Therefore, I advise the music managers to keep investing in those genres to maximise sales.

Fifth, the company earns the most revenue per purchase in the genres TV Shows, Science Fiction and Alternative. These genres are not bought frequently but double the revenue in comparison to the average purchase made. Therefore could be interesting if the costs are the same as for other genres. In this case, the music managers should keep these genres in their portfolio to gain high profit with little effort. If costs for these genres would rise or purchases drop, the managers can still drop those genres. This is called a “cash-cow-strategy” in marketing.