Yun Zhou, 300442776

School of Engineering and Computer Science

SWEN 304/439 Database System Engineering

Project 1

Due: Monday 25 April, 23:59 pm

This project gives you practice in developing and using relational databases using PostgreSQL. The project is worth **20%** of your final grade. It will be marked out of 100.

Submission Instructions

Please submit your project via the submission system:

1. **Your answers to all questions in a pdf file.** For each question please include:
   1) Your SQL code **and**
   2) PostgreSQL's responses to your SQL statements and messages.
   Note: *marks will be deducted if responses are not provided.*

2. Additionally, for the following questions **submit SQL code in sql files:**
   - Question 1,
   - Question 4, one for each task, and
   - Question 5, one for each task, and
   - Question 6, submit only for your nested queries, but not for the queries of your stepwise approach, one for each task.

Note: details about what should be included in your submissions can be found at the end of each question. ***Marks will be deducted if sql files are not submitted.***

The Database Server

For this project we will use the PostgreSQL Database Management System. A brief tutorial on using PostgreSQL is given at the end of this handout.

For more detailed information on PostgreSQL, please refer to the online PostgreSQL Manual. The link is given on the SWEN304 home page. **Be careful: The PostgreSQL manual is HUGE!!! Please do not print out the entire manual!**

The Business Case

The story: You are a database engineer hired by the Chicago police department. Chicago is the capital of Cook county which includes also several smaller cities like Burbank, Deerfield and Evanston. For quite some time, the police have been investigating a gang of elusive bank robbers who have been operating in Cook county. The police have collected quite a lot of information about the gang, some from an informant close to one of the gang members. So far, the police department has been keeping the data in a set of spreadsheets, but they realize that they cannot do many of the queries they want in the spreadsheets and they are also worried that the data entry is introducing errors and inconsistencies that the spreadsheets does not check for. From the spreadsheets, they have produced a collection of tab-separated files of data. They now

want you to convert the data into a well-designed relational database and generate some standard queries.

The data: You find the data files stored on the Assignment page of the SWEN304 website. They contain information about the gang and the banks in the cities where they have been operating:

`banks_22.data`: lists all the bank branches in Cook county. The banks are specified by the name of the bank and the city where the branch is located in. The data file also includes the number of accounts held in the bank (an indicator of size) and the level of the security measures installed by the bank.

`robbers_22.data`: contains the name (actually, the nickname), age, and number of years spent in prison of each gang member.

`hasaccounts_22.data`: lists the banks at which the various robbers have accounts.

`hasskills_22.data`: specifies the skills of the robbers. Each robber may have several skills, ranked by preference – what activity the robber prefers to be engaged in. The robbers are also graded on each skill. The file contains a line for each skill of each robber listing the robber's nickname, the skill description, the preference rank (a number where 1 represents first preference), and the grade.

`robberies_22.data`: contains the banks that have been robbed by the gang so far. For each robbery, it lists the bank branch, the date of the robbery, and the amount that was stolen. Note that some banks may have been robbed more than once.

`accomplices_22.data`: lists the robbers that were involved in each robbery and their estimated share of the money.

`plans_22.data`: contains information from the informant about banks that the gang is planning to rob in the future, along with the planed robbery date and number of gang members that would be needed. Note that gang may plan to rob some banks more than once.

Each of these files could be converted directly to a relation in the database. However, this would not be a great design.

The nickname problem: The robbers are currently identified by their nicknames. Although the current list has no duplicates, it is quite possible to have two robbers with the same nickname. It would be better to give each robber a unique Id, and to use the Id for identifying the robber in all the tables. This way, adding a new robber with a duplicate nickname would not require redesigning the whole database schema.

The skills problem: The list of robber skills uses the descriptions of the skills. There should be a finite set of possible skills, and we would like to ensure that data entry does not misspell skills. Misspelled skills constitute a severe concern since queries might then miss out some answers because of the misspelling. One approach is to define a constraint on the skill attribute of the *HasSkills* table that checks that every value is one of the possible skills. However, if we then wanted to add a further kind of skill, we would have to change the database schema. A better design can be achieved if we introduce an additional *Skills* table listing all the possible skills, and define a constraint on the *HasSkills* table to ensure that every skill there is also in the new *Skills* table.

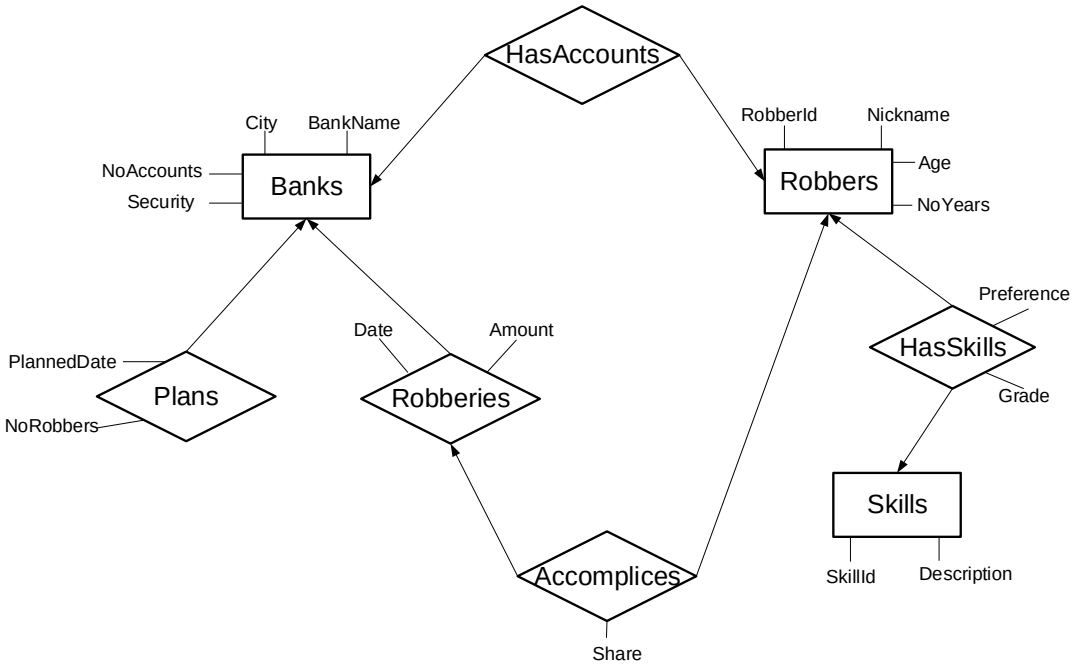
Further assumptions: The banks are identified by their name and city rather than by an Id. The business rules set by the local banking authority ensure that the combination of name and city is unique, so that it is not necessary to create an Id for the banks.

You will need to:

- Define the schema of the *RobbersGang* database using PostgreSQL (Question 1);
- Populate the schema using the data files. This will require some transformation of the data files, and probably making some temporary files or tables (Question 2);
- Check that the database enforces the required consistency checks by submitting a series of data manipulations to the database that should all be rejected by PostgreSQL (Question 3);
- Write a set of queries for the database (Question 4, 5 and 6).

QUESTION 1: Defining the Database [15 marks]

A partial EER diagram is designed for the ***RobbersGang*** database. Note that keys are not yet defined for all the types in the diagram.



You are expected to use SQL as a data definition language. Define relation schemas by CREATE TABLE statements for each of the following database relations, derived from the EER diagram above:

- **Banks**, which stores information about banks, including the bank name, the city where the bank is located, the number of accounts and the security level of the bank.
  Attributes: *BankName, City, NoAccounts, Security*
- **Robberies**, which stores information about bank robberies that the gang has already performed, including when the robbery took place and how much money was stolen.
  Attributes: *BankName, City, Date, Amount*
- **Plans**, which stores information about robbery plans of the gang, including the number of gang members needed and when the robbery will take place.
  Attributes: *BankName, City, NoRobbers, PlannedDate*
- **Robbers**, which stores information about gang members. Note that it is not possible to be in prison for more years than you have been alive!
  Attributes: *RobberId, Nickname, Age, NoYears*
- **Skills**, which stores the possible robbery skills.
  Attributes: *SkillId, Description*
- **HasSkills**, which stores information about the skills that particular gang members possess. Each skill of a gang member has a preference rank and a grade.
  Attributes: *RobberId, SkillId, Preference, Grade*

- **HasAccounts**, which stores information about the banks where individual gang members have accounts.
  Attributes: *RobberId, BankName, City*
- **Accomplices**, which stores information about which gang members participated in which robbery, and what share of the money they got.
  Attributes: *RobberId, BankName, City, Date, Share*

You are expected to design the database with appropriate choices of:

- *Keys.* Choose appropriate attributes or sets of attributes to be keys, and decide on the primary key.
- *Foreign keys*. Determine all foreign keys, and decide what should be done if the tuple referred to is deleted or modified.
- *Attribute constraints.* Choose suitable basic data types and additional constraints, such as NOT NULL constraints, CHECK constraints, or DEFAULT values.

Note: *Question 3 will be used to check if the above constraints are set up properly.*

Your answer to Question 1 should include:

1. A list of the primary keys and foreign keys for each relation, along with a brief justification for your choice of keys and foreign keys.

Banks:

- Attributes: (BankName, City, NoAccounts, Security)
- Primary key: (BankName, City)

There will be duplicates if we use only BankName or City as the primary key. But from the further assumption, we can see that the combination of BankName and City is guaranteed to be unique. Consequently, by using them as primary key, the ability to uniquely identify the specified Banks entry is ensured. There is no foreign key.

Robberies*:*

- Attributes: (BankName, City, Date, Amount)
- Primary key: (BankName, City, Date)
- Foreign key: (BankName, City)

Since the BankName and City is the primary key of the Banks relation, so it makes them become the foreign key of this Robberies table. For the primary key, due to the same bank might be robbed more than once, so the primary key will be BankName, City and Date. They can uniquely identify the specified Robbery entry and no duplicates.

Plans:

- Attributes: (BankName, City, NoRobbers, PlannedDate)
- Primary key: (BankName, City, PlannedDate)
- Foreign key: (BankName, City)

Since the BankName and City is the primary key of the Banks relation, so it makes them become the foreign key of this Plans table. For the primary key, due to the gang may plan to rob some banks more than once, so NoRobbers attribute is useless. And BankName, City and PlanedDate can uniquely identify the specified Plans entry and will not have duplicates.

Robbers:

- Attributes: (RobberId, Nickname, Age, NoYears)
- Primary key: (RobberId)

The RobberId is proposed for solving the possible duplicate nickname problem that might occur in the future. Therefore, it is guaranteed to uniquely identify the specified Robber entry and no duplicates. There is no foreign key in this relation.

Skills:

- Attributes: (SkillId, Description)
- Primary key: (SkillId)

The SkillId is unique in the Skills relation. Therefore, it can uniquely identify the specified Skills entry and no duplicates. There is no foreign key in this relation.

HasSkills:

- Attributes: (RobberId, SkillId, Preference, Grade)
- Primary Key: (RobberId, SkillId)
- Foreign key: (SkillId, RobberId)

The SkillId is the primary key of the Skills relation and RobberId is the primary key of Robbers relation, so that they are the foreign key of this HasSkills relation. For the primary key, to determine the which Robber has which skill will need RobberId and SkillId, so they are the primary key of this HasSkills relation.

HasAccounts:

- Attributes: (RobberId, BankName, City)
- Primary key: (RobberId, BankName, City)
- Foreign key: (BankName, City, RobberId)

BankName and City are the primary key of the Banks relation, and RobberId is the primary key of the Robbers relation, so they are all the foreign key of this HasAccounts relation. For the primary key, it needs all attributes since to determine which Robber at which Bank has accounts require RobberId, City and Bank in order to uniquely identify the specified row without duplicates.

Accomplices:

- Attributes: (RobberId, BankName, City, Date, Share)
- Primary key: (RobberId. BankName, City, Date)
- Foreign key: (RobberId. BankName, City, Date)

To uniquely identify the specified accomplice, we need to know which Robber at which Robbery get what estimated amount of share, so RobberId. BankName, City, Date are the primary keys of this relation. Also, through different attributes combinations, they are primary keys of other relation tables, such as BankName, City and Date are they primary key of the Robberies relation, RobberId is the primary key of Robbers relation, etc. so they are also the foreign keys.

2. A list of all your CREATE TABLE statements.

```
CREATE TABLE Banks (
   BankName CHAR(30) NOT NULL,
   City CHAR(30) NOT NULL,
   NoAccounts INT DEFAULT 0,
   Security CHAR(15) CHECK(
      Security = 'weak'
      OR Security = 'good'
      OR Security = 'very good'
      OR Security = 'excellent'
   ),
   CONSTRAINT BanksPK PRIMARY KEY (BankName, City),
   CONSTRAINT NoAccountsCheck CHECK(NoAccounts >= 0)
);
```

```
swen439p1=> REVOKE CONNECT ON DATABASE  swen439p1 FROM PUBLIC;
REVOKE
swen439p1=> CREATE TABLE Banks (
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL,
swen439p1(>     NoAccounts INT DEFAULT 0,
swen439p1(>     Security CHAR(15) CHECK(
swen439p1(>         Security = 'weak'
swen439p1(>         OR Security = 'good'
swen439p1(>         OR Security = 'very good'
swen439p1(>         OR Security = 'excellent'
swen439p1(>     ),
swen439p1(>     CONSTRAINT BanksPK PRIMARY KEY (BankName, City),
swen439p1(>     CONSTRAINT NoAccountsCheck CHECK(NoAccounts >= 0)
swen439p1(> );
CREATE TABLE
swen439p1=>
```

```
CREATE TABLE Robberies(
   BankName CHAR(30) NOT NULL,
   City CHAR(30) NOT NULL,
   Date DATE NOT NULL,
   Amount REAL NOT NULL CHECK(Amount >= 0),
   CONSTRAINT RobberiesPK PRIMARY KEY (BankName, City, Date),
   CONSTRAINT RobberiesFK FOREIGN KEY (BankName, City) REFERENCES
Banks(BankName, City)
);
```

```
swen439p1=> CREATE TABLE Robberies(
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL,
swen439p1(>     Date DATE NOT NULL,
swen439p1(>     Amount REAL NOT NULL CHECK(Amount >= 0),
swen439p1(>     CONSTRAINT RobberiesPK PRIMARY KEY (BankName, City, Date),
swen439p1(>     CONSTRAINT RobberiesFK FOREIGN KEY (BankName, City) REFERENCES B
anks(BankName, City)
swen439p1(> );
CREATE TABLE
swen439p1=>
```

```
CREATE TABLE Plans(
   BankName CHAR(30) NOT NULL,
   City CHAR(30) NOT NULL,
   NoRobbers INT CHECK(NoRobbers > 0),
   PlannedDate DATE NOT NULL,
   CONSTRAINT PlansPK PRIMARY KEY (BankName, City, PlannedDate),
   CONSTRAINT PlansFK FOREIGN KEY (BankName, City) REFERENCES
Banks(BankName, City)
);
```

```
swen439p1=> CREATE TABLE Plans(
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL,
swen439p1(>     NoRobbers INT CHECK(NoRobbers > 0),
swen439p1(>     PlannedDate DATE NOT NULL,
swen439p1(>     CONSTRAINT PlansPK PRIMARY KEY (BankName, City, PlannedDate),
swen439p1(>     CONSTRAINT PlansFK FOREIGN KEY (BankName, City) REFERENCES Banks(BankName, City)
swen439p1(> );
CREATE TABLE
swen439p1=>
```

```
CREATE TABLE Robbers(
   RobberId INT PRIMARY KEY NOT NULL,
   Nickname CHAR(25) NOT NULL,
   Age INT NOT NULL,
   NoYears INT DEFAULT 0,
   CONSTRAINT NoYearsCheck CHECK(NoYears >= 0),
   CONSTRAINT AgeCheck CHECK(Age >= NoYears)
);
```

```
swen439p1=> CREATE TABLE Robbers(
swen439p1(>     RobberId INT PRIMARY KEY NOT NULL,
swen439p1(>     Nickname CHAR(25) NOT NULL,
swen439p1(>     Age INT NOT NULL,
swen439p1(>     NoYears INT DEFAULT 0,
swen439p1(>     CONSTRAINT NoYearsCheck CHECK(NoYears >= 0),
swen439p1(>     CONSTRAINT AgeCheck CHECK(Age >= NoYears)
swen439p1(> );
CREATE TABLE
```

```
CREATE TABLE Skills(
   SkillId INT PRIMARY KEY NOT NULL CHECK(SkillId >= 0),
   Description CHAR(25) NOT NULL
);
```

```
swen439p1=> CREATE TABLE Skills(
swen439p1(>     SkillId INT PRIMARY KEY NOT NULL CHECK(SkillId >= 0),
swen439p1(>     Description CHAR(25) NOT NULL
swen439p1(> );
CREATE TABLE
swen439p1=>
```

When doing Q3 task1a, I find that the should my database is not yet correct, I realize that Description should add unique constraint:

- ALTER TABLE Skills ADD CONSTRAINT Description_unique UNIQUE(Description);

```
swen439p1=> ALTER TABLE Skills ADD CONSTRAINT Description_unique UNIQUE(Descript
ion);
ALTER TABLE
```

```
CREATE TABLE HasSkills(
   RobberId INT NOT NULL REFERENCES Robbers(RobberId),
   SkillId INT NOT NULL REFERENCES Skills(SkillId),
   Preference INT NOT NULL,
   Grade CHAR(3) NOT NULL,
   CONSTRAINT HasSkillsPK PRIMARY KEY(RobberId, SkillId)
);
```

```
swen439p1=> CREATE TABLE HasSkills(
swen439p1(>     RobberId INT NOT NULL REFERENCES Robbers(RobberId),
swen439p1(>     SkillId INT NOT NULL REFERENCES Skills(SkillId),
swen439p1(>     Preference INT NOT NULL,
swen439p1(>     Grade CHAR(3) NOT NULL,
swen439p1(>     CONSTRAINT HasSkillsPK PRIMARY KEY(RobberId, SkillId)
swen439p1(> );
CREATE TABLE
```

When doing Q3 8a, I find out that I should add a UNIQUE constraint that a specified Robber can not have duplicate Preference, so I add the unique constraint below.

- ALTER TABLE Hasskills
- ADD CONSTRAINT id_preference_unique UNIQUE (RobberId, Preference);

```
swen439p1=> ALTER TABLE Hasskills
swen439p1-> ADD CONSTRAINT  id_preference_unique UNIQUE (RobberId, Preference);
ALTER TABLE
```

```
CREATE TABLE HasAccounts(
   RobberId INT NOT NULL REFERENCES Robbers(RobberId),
   BankName CHAR(30) NOT NULL,
   City CHAR(30) NOT NULL,
   CONSTRAINT HasAccountsPK PRIMARY KEY (RobberId, BankName, City),
   CONSTRAINT HasAccountsFK_2 FOREIGN KEY(BankName, City) REFERENCES
Banks(BankName, City)
);
```

```
swen439p1=> CREATE TABLE HasAccounts(
swen439p1(>     RobberId INT NOT NULL REFERENCES Robbers(RobberId),
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL,
swen439p1(>     CONSTRAINT HasAccountsPK PRIMARY KEY (RobberId, BankName, City),
swen439p1(>     CONSTRAINT HasAccountsFK_2 FOREIGN KEY(BankName, City) REFERENCES Banks(BankName,
City)
swen439p1(> );
CREATE TABLE
```

```
CREATE TABLE Accomplices(
   RobberId INT NOT NULL REFERENCES Robbers(RobberId),
   BankName CHAR(30) NOT NULL,
   City CHAR(30) NOT NULL,
   Date DATE NOT NULL,
   Share REAL NOT NULL,
   CONSTRAINT AccomplicesPK PRIMARY KEY (RobberId, BankName, City, Date),
   CONSTRAINT AccomplicesFK_2 FOREIGN KEY(BankName, City) REFERENCES
Banks(BankName, City),
   CONSTRAINT AccomplicesFK_3 FOREIGN KEY(BankName, City, Date) REFERENCES
Robberies(BankName, City, Date)
);
```

```
swen439p1=> CREATE TABLE Accomplices(
swen439p1(>     RobberId INT NOT NULL REFERENCES Robbers(RobberId),
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL,
swen439p1(>     Date DATE NOT NULL,
swen439p1(>     Share REAL NOT NULL,
swen439p1(>     CONSTRAINT AccomplicesPK PRIMARY KEY (RobberId, BankName, City, Date),
swen439p1(>     CONSTRAINT AccomplicesFK_2 FOREIGN KEY(BankName, City) REFERENCES Banks(BankName,
City),
swen439p1(>     CONSTRAINT AccomplicesFK_3 FOREIGN KEY(BankName, City, Date) REFERENCES Robberies(
BankName, City, Date)
swen439p1(> );
CREATE TABLE
swen439p1=>
```

All tables that are created:

```
swen439p1=> \dt
           List of relations
 Schema |     Name     | Type  |  Owner
--------+--------------+-------+---------
 public | accomplices  | table | zhouyun
 public | banks        | table | zhouyun
 public | hasaccounts  | table | zhouyun
 public | hasskills    | table | zhouyun
 public | plans        | table | zhouyun
 public | robberies    | table | zhouyun
 public | robbers      | table | zhouyun
 public | skills       | table | zhouyun
(8 rows)

swen439p1=>
```

3. A justification for your choice of actions on delete or on update for each foreign key.

For the update action,

- If I want to update an existing row, then for the foreign key attribute(s) of that row, they must exist and match the corresponding tuple in the reference table so that the referential integrity constraint will not be violated. For example, (BankName, City) are the foreign key in Plans table and they point to the Banks table as the primary key. Therefore, for the updated value, (BankName, City) must exist and match the corresponding tuple in Banks table.
- If I want to update the type of the foreign key, such as change the type from INT to DOUBLE, the primary key of the table that the foreign key points to would also need to update the type. Alternatively, there is a dummy way that we can drop and re-create both tables again. The type must be consistent.

For the delete action, if I want to delete the foreign key, I should first need to drop the foreign key constraint then I should be able to delete the foreign key.

By the way, the referential integrity constraint will be violated if the deleted tuple is referred by foreign keys in some tuples in other relations. For instances, if I delete a tuple in the Robbers then it will most likely violate the constraint since the RobberId is both the foreign key of HasSkills and HasAccounts, so it will cause the corresponding tuples in HasSkills and HasAccounts refer to nothing so that cause errors. In this way, I can reject the deletion or set the corresponding tuples in HasSkills and HasAccounts to null/default then delete the tuple in Robbers Or delete all affected tuples.

For this question, since it just simply deletes the foreign key which means no other relations will refer to the foreign key, so we can just simply drop the foreign key constraint then delete foreign key. However, there is an outlier that the foreign key is also the primary key in which other relations will refer to.

4. A brief justification for your choice of attribute constraints (other than the basic data).

Except for the basic data constraints such as the primary key and foreign key constraints, I have also set up some other attribute constraints that are inspired from the given data. For instances:

- In Banks table, I add the check constraints on the Security to ensure that the value must be one of "weak", "good", "very good" and "excellent"
- I've also add some check constraints to be greater than 0 in various attributes, such as Amount, Age, NoAccounts(stand for number of accounts), etc, they are should be a positive number, so I add >0 constraints on them.
- For the Id attributes, since they are used for the primary key purpose, so it would be better if they are non-negative integer values, so I've also added the check constraints on them.
- I've also applied the check constraint based on the attribute relationship. For example, in the Robbers table, there is a logic that a Robber cannot in prison for more years than its age, so I add a check constraint that Age >= NoYears
- For the skills table, when doing Q3, I realize that the Description col should add a UNIQUE constraint since skills table is generated based on hasskills table, which means the description need to be set as Unique.

All my choice of attribute constraints makes senses and follow the logic as shown above.

QUESTION 2: Populating your Database with Data [15 marks]

Now that you have your relation schemas defined, you are expected to insert data into your database. On the SWEN304 website you find tab-separated text files for the tables of your database. To begin with, copy these files to a folder (say Pro1) in your private directory. If the data in the text file matches the relation directly (which should be true for some of the files), you can insert the data using the `\copy` command inside the PostgreSQL interpreter:

```
dbname=> \copy Banks FROM ~/Pro1/Banks_22.data
```

or

```
dbname=> \copy Banks(bankname,city,noaccounts,security) FROM
~/Pro1/Banks_22.data
```

The first form assumes that each line of the `Banks_22.data` file contains the right number of attributes for the relation and in exactly the same order as they were specified in the CREATE TABLE statement. The second form allows you to specify which attributes are present in the file and in what order. If not all attributes of the relation are specified, the other attributes will be assigned a default value or a null.

For other files, you will need to do more work to convert the data. Although you could (for this little database) convert the text files by hand, this would no longer be feasible for large amounts of data. Therefore, we want you to practice the use of PostgreSQL to do the conversion.

Dealing with the Robber Ids is a little trickier. You are expected to generate these Ids. You can use PostgreSQL to generate Ids with the help of the *Serial* data type. Please consult Section II (8. Data Types) of the online PostgreSQL manual. You will also need to convert the data in `Hasskills_22.data`, `Hasaccounts_22.data`, and `Accomplices_22.data` to use the Robber Ids instead of the nicknames. You will probably need to make temporary relations and do various joins. You may wish to use the INSERT statement in the form:

```
dbname=> INSERT INTO <table_name> (<attribute_list>) SELECT …
```

Moreover, you are expected to construct the *Skills* table based on the data in the `Hasskills_22.data` file. To do this, you will need to load the data into a table, then extract

the *Description* column, and put it into the *Skills* table. Note that you should not be able to use the *HasSkills* table to do this because of the foreign key in the *HasSkills* table that depends on the *Skills* table. Rather you will need to construct a temporary relation, copy the `Hasskills_22.data` file into that relation then extract the values from that.

Please note:

1. You need to keep a record of the steps that you went through during the data conversion. This can be just the sequence of PostgreSQL statements you performed.
2. The data in the data files is consistent. We trust (or at least hope) that we have removed all errors from it. In a real situation, there are likely to be errors and inconsistencies in the data, which would make the data conversion process a lot trickier.

Your answer to Question 2 should include:

1. A description of how you performed all the data conversion, for example, a sequence of the PostgreSQL statements that accomplished the conversion. [12 marks]

Populate the Banks Table:

\copy Banks(BankName, City, NoAccounts, Security) FROM ~/git/439p1/data2022/banks_22.data

```
swen439p1=> \copy Banks(BankName, City, NoAccounts, Security) FROM ~/git/439p1/data2022/banks_22.data
COPY 20
swen439p1=>
```

Populate the Robberies Table:

\copy Robberies(BankName, City, Date, Amount) FROM ~/git/439p1/data2022/robberies_22.data

```
swen439p1=> \copy Robberies(BankName, City, Date, Amount) FROM ~/git/439p1/data2022/robberies_22.data
COPY 21
```

Populate Plans Table:

\copy Plans(BankName, City, PlannedDate, NoRobbers) FROM ~/git/439p1/data2022/plans_22.data

```
swen439p1=> \copy Plans(BankName, City, PlannedDate, NoRobbers) FROM ~/git/439p1/data2022/plans_22.data
COPY 11
```

Populate Robbers Table:

Due to I've set the id to int type first, so through the online tutorial, I obtain that first I need to CREATE SEQUENCE Robbers_id_seq AS integer;

```
swen439p1=> CREATE SEQUENCE Robbers_id_seq AS integer;
CREATE SEQUENCE
```

Then,

ALTER TABLE Robbers
ALTER COLUMN RobberId SET DEFAULT nextval('Robbers_id_seq');
ALTER SEQUENCE Robbers_id_seq OWNED BY Robbers.RobberId;

```
swen439p1=> ALTER TABLE Robbers
swen439p1-> ALTER COLUMN RobberId SET DEFAULT nextval('Robbers_id_seq');
ALTER TABLE
swen439p1=> ALTER SEQUENCE Robbers_id_seq OWNED BY Robbers.RobberId
swen439p1-> ;
ALTER SEQUENCE
```

Finally, populate the Robbers table by using the second form :

\copy Robbers(NickName, Age, NoYears) FROM ~/git/439p1/data2022/robbers_22.data

```
swen439p1=> \copy Robbers(NickName, Age, NoYears) FROM ~/git/439p1/data2022/robb
ers_22.data
COPY 24
```

List all rows for checking, from below screenshot we can see that 24 rows are added

successfully and their corresponding robberid is increased as expected. By the way, I further explore how the id works by deleting all rows (i.e. DELETE from Robbers;) and add them again. I find out that the robberid will then start at 25 instead of start at 1, it shows that no matter old rows are deleted or not, their id will not be rest. Additionally, if I want all rows be deleted as well as reset the id start from 1, I should type:

- TRUNCATE Robbers RESTART IDENTITY CASCADE;

```
swen439p1=> select * FROM Robbers
swen439p1-> \G
Invalid command \G. Try \? for help.
swen439p1-> \g
 robberid |         nickname          | age | noyears
----------+---------------------------+-----+---------
        1 | Al Capone                 |  31 |       2
        2 | Bugsy Malone              |  42 |      15
        3 | Lucky Luchiano            |  42 |      15
        4 | Anastazia                 |  48 |      15
        5 | Mimmy The Mau Mau         |  18 |       0
        6 | Tony Genovese             |  28 |      16
        7 | Dutch Schulz              |  64 |      31
        8 | Clyde                     |  20 |       0
        9 | Calamity Jane             |  44 |       3
       10 | Bonnie                    |  19 |       0
       11 | Meyer Lansky              |  34 |       6
       12 | Moe Dalitz                |  41 |       3
       13 | Mickey Cohen              |  24 |       3
       14 | Kid Cann                  |  14 |       0
       15 | Boo Boo Hoff              |  54 |      13
       16 | King Solomon              |  74 |      43
       17 | Bugsy Siegel              |  48 |      13
       18 | Vito Genovese             |  66 |       0
       19 | Mike Genovese             |  35 |       0
       20 | Longy Zwillman            |  35 |       6
       21 | Waxey Gordon              |  15 |       0
       22 | Greasy Guzik              |  25 |       1
       23 | Lepke Buchalter           |  25 |       1
       24 | Sonny Genovese            |  39 |       0
(24 rows)
```

Populate Skills Table:

First, for the auto increment id, we do the same as Robbers id.

- CREATE SEQUENCE Skills_id_seq AS integer;
- ALTER TABLE Skills
- ALTER COLUMN SkillId SET DEFAULT nextval('Skills_id_seq');
- ALTER SEQUENCE Skills_id_seq OWNED BY Skills.SkillId;

```
swen439p1=> CREATE SEQUENCE Skills_id_seq AS integer;
CREATE SEQUENCE
swen439p1=> ALTER TABLE Skills
swen439p1-> ALTER COLUMN SkillId SET DEFAULT nextval('Skills_id_seq');
ALTER TABLE
swen439p1=> ALTER SEQUENCE Skills_id_seq OWNED BY Skills.SkillId;
ALTER SEQUENCE
```

Then, as handout said, create a temp table to copy data from HasSkills.data since Skills table is constructed based on HasSkills table:

- CREATE TABLE TempSkills (
  RobberNicknameTemp CHAR(25) NOT NULL,
  DescriptionTemp CHAR(25) NOT NULL,
  PreferenceTemp INT NOT NULL CHECK(PreferenceTemp >= 0),
  GradeTemp CHAR(3) NOT NULL
  );
  \copy TempSkills FROM ~/git/439p1/data2022/hasskills_22.data

```
swen439p1=> CREATE TABLE TempSkills (
swen439p1(>     RobberNicknameTemp CHAR(25) NOT NULL,
swen439p1(>     DescriptionTemp CHAR(25) NOT NULL,
swen439p1(>     PreferenceTemp INT NOT NULL CHECK(PreferenceTemp >= 0),
swen439p1(>     GradeTemp CHAR(3) NOT NULL
swen439p1(> );
CREATE TABLE
swen439p1=> \copy TempSkills FROM ~/git/439p1/data2022/Hasskills_22.data
/home/zhouyun/git/439p1/data2022/Hasskills_22.data: No such file or directory
swen439p1=> \copy TempSkills FROM ~/git/439p1/data2022/hasskills_22.data
COPY 38
```

Then, extract the descriptionTemp column and put it to the Skills table:

- INSERT INTO Skills(Description) SELECT DescriptionTemp FROM TempSkills;

```
swen439p1=> INSERT INTO Skills(Description) SELECT DescriptionTemp FROM TempSkil
ls;
INSERT 0 38
```

However, by observing all inserted rows, I find out that there are duplicate descriptions which should not occur.

```
 27 | Safe-Cracking
 28 | Money Counting
 29 | Money Counting
 30 | Safe-Cracking
```

Therefore, by googling, I use the command shown below to delete all duplicates, and we can see that there are 12 rows left.

```
DELETE FROM Skills rowA USING (
        SELECT MIN(ctid) as ctid,
            description
        FROM Skills
        GROUP BY description
        HAVING COUNT(*) > 1
    ) rowB
WHERE rowA.description = rowB.description
    AND rowA.ctid <> rowB.ctid;
```

```
swen439p1=> DELETE FROM Skills rowA USING (
swen439p1(>         SELECT MIN(ctid) as ctid,
swen439p1(>             description
swen439p1(>         FROM Skills
swen439p1(>         GROUP BY description
swen439p1(>         HAVING COUNT(*) > 1
swen439p1(>     ) rowB
swen439p1-> WHERE rowA.description = rowB.description
swen439p1->     AND rowA.ctid <> rowB.ctid;
DELETE 26
swen439p1=> select * from skills
swen439p1-> \g
 skillid |        description
---------+---------------------------
       1 | Planning
       2 | Safe-Cracking
       3 | Preaching
       5 | Driving
       6 | Guarding
       9 | Explosives
      12 | Gun-Shooting
      13 | Lock-Picking
      14 | Scouting
      20 | Money Counting
      34 | Eating
      36 | Cooking
(12 rows)
```

However, we can see the skillId is not in order, so I need to keep them in order from 1 to 12, screenshot and commands is shown below:

- CREATE SEQUENCE Skills_id_seq2 AS integer;
- ALTER SEQUENCE Skills_id_seq2 OWNED BY Skills.SkillId;
- update Skills set SkillId = nextval('Skills_id_seq2')
- \g

- select * from Skills
- \g

```
swen439p1=> CREATE SEQUENCE Skills_id_seq2 AS integer;
CREATE SEQUENCE
swen439p1=> ALTER SEQUENCE Skills_id_seq2 OWNED BY Skills.SkillId;
ALTER SEQUENCE
swen439p1=> update Skills set SkillId = nextval('Skills_id_seq2')
swen439p1-> \g
UPDATE 12
swen439p1=> select * from SKills
swen439p1-> \g
 skillid |        description
---------+---------------------------
       1 | Planning
       2 | Safe-Cracking
       3 | Preaching
       4 | Driving
       5 | Guarding
       6 | Explosives
       7 | Gun-Shooting
       8 | Lock-Picking
       9 | Scouting
      10 | Money Counting
      11 | Eating
      12 | Cooking
(12 rows)
```

Populate HasSkills Table:

extract the RobberId and SkillId columns from Robber and Skills table, for the Preference and Grade, use tempSkills table directly.

```
INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
SELECT robber.RobberId,
   skill.SkillId,
   tempSkill.PreferenceTemp,
   tempSkill.GradeTemp
FROM TempSkills tempSkill,
   Robbers robber,
   Skills skill
WHERE robber.Nickname = tempSkill.RobberNicknameTemp
   AND skill.Description = tempSkill.DescriptionTemp;
```

```
swen439p1=> INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
swen439p1-> SELECT robber.RobberId,
swen439p1->     skill.SkillId,
swen439p1->     tempSkill.PreferenceTemp,
swen439p1->     tempSkill.GradeTemp
swen439p1-> FROM TempSkills tempSkill,
swen439p1->     Robbers robber,
swen439p1->     Skills skill
swen439p1-> WHERE robber.Nickname = tempSkill.RobberNicknameTemp
swen439p1->     AND skill.Description = tempSkill.DescriptionTemp;
INSERT 0 38
```

We can see that 38 rows are inserted which match the given hasskills.data file. For further review, we can see all rows shown below:

```
swen439p1=> select  * from hasSkills
swen439p1-> \g
 robberid | skillid | preference | grade
----------+---------+------------+-------
        1 |       3 |          3 | A+
        1 |       2 |          2 | C+
        1 |       1 |          1 | A+
        2 |       6 |          1 | A
        3 |       4 |          2 | B+
        3 |       8 |          1 | B+
        4 |       5 |          1 | A
        5 |       4 |          2 | C
        5 |       1 |          1 | A+
        6 |      11 |          1 | B+
        7 |       4 |          2 | C+
        7 |       8 |          1 | A+
        8 |       1 |          3 | C
        8 |       9 |          2 | C+
        8 |       8 |          1 | C+
        9 |       7 |          1 | B
       10 |       3 |          1 | B
       11 |       2 |          1 | A+
       12 |       2 |          1 | A
       13 |      10 |          1 | B+
       14 |      10 |          1 | B
       15 |       1 |          1 | A+
       16 |       1 |          1 | A
       17 |       5 |          2 | C+
       17 |       4 |          1 | A+
       18 |      11 |          3 | A+
       18 |      12 |          2 | A
       18 |       9 |          1 | B+
       19 |      10 |          1 | C
       20 |       4 |          1 | C
       21 |       7 |          1 | C
       22 |       8 |          2 | C
       22 |       3 |          1 | A+
       23 |       5 |          2 | C
       23 |       4 |          1 | A
       24 |       8 |          3 | B
       24 |       2 |          2 | C+
       24 |       6 |          1 | B
(38 rows)
```

Populate HasAccounts Table:

This one is similar as the hasskills table. First, construct a temp table to hold the data from the given data files, then copy them into the temp table.

```
CREATE TABLE TempHasaccounts (
   RobberNicknameTemp CHAR(25) NOT NULL,
   BankName CHAR(30) NOT NULL,
   City CHAR(30) NOT NULL
);
```

```
swen439p1=> CREATE TABLE TempHasaccounts (
swen439p1(>     RobberNicknameTemp CHAR(25) NOT NULL,
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL
swen439p1(> );
CREATE TABLE
swen439p1=> \copy TempHasaccounts FROM ~/git/439p1/data2022/hasaccounts_22.data
COPY 31
```

Then, extract the RobberId columns from Robbers table, for the BankName and City, use TempHasAccounts table directly.

```
INSERT INTO HasAccounts(RobberId, BankName, City)
SELECT robber.RobberId, temp.BankName, temp.City FROM Robbers robber,TempHasaccounts temp
WHERE robber.Nickname = temp.RobberNicknameTemp;
```

From below screenshot, we can see that 31 rows are inserted successfully as expect.

```
swen439p1=> INSERT INTO HasAccounts(RobberId, BankName, City)
swen439p1-> SELECT robber.RobberId,
swen439p1->     temp.BankName,
swen439p1->     temp.City
swen439p1-> FROM Robbers robber,
swen439p1->     TempHasaccounts temp
swen439p1-> WHERE robber.Nickname = temp.RobberNicknameTemp;
INSERT 0 31
swen439p1=> select * from hasaccounts
swen439p1-> \g
 robberid |           bankname            |             city
----------+-------------------------------+---------------------
        1 | NXP Bank                      | Chicago
        1 | Inter-Gang Bank               | Evanston
        1 | Bad Bank                      | Chicago
        3 | Bankrupt Bank                 | Evanston
        3 | NXP Bank                      | Chicago
        4 | Loanshark Bank                | Evanston
        2 | Loanshark Bank                | Deerfield
        2 | Loanshark Bank                | Chicago
        7 | Inter-Gang Bank               | Chicago
        8 | Penny Pinchers                | Evanston
        9 | Dollar Grabbers               | Chicago
        9 | Bad Bank                      | Chicago
        9 | PickPocket Bank               | Evanston
        9 | PickPocket Bank               | Chicago
       11 | Penny Pinchers                | Evanston
       12 | Gun Chase Bank                | Evanston
       12 | Dollar Grabbers               | Evanston
       13 | Gun Chase Bank                | Burbank
       14 | PickPocket Bank               | Evanston
       15 | PickPocket Bank               | Deerfield
       17 | PickPocket Bank               | Chicago
       24 | Hidden Treasure               | Chicago
       18 | Gun Chase Bank                | Evanston
       18 | Bad Bank                      | Chicago
       19 | Gun Chase Bank                | Burbank
       20 | PickPocket Bank               | Evanston
       21 | PickPocket Bank               | Evanston
       22 | PickPocket Bank               | Chicago
       23 | Hidden Treasure               | Chicago
        5 | Loanshark Bank                | Evanston
        5 | Inter-Gang Bank               | Evanston
(31 rows)
```

Populate Accomplices Table:

This one is similar as the HasAccounts table. First, construct a temp table to hold the data from the given data files, then copy them into the temp table.

```
CREATE TABLE TempAccomplices (
    RobberNicknameTemp CHAR(25) NOT NULL,
    BankName CHAR(30) NOT NULL,
    City CHAR(30) NOT NULL,
    DateTemp DATE NOT NULL,
    ShareTemp REAL NOT NULL
);
```

```
swen439p1=> CREATE TABLE TempAccomplices (
swen439p1(>     RobberNicknameTemp CHAR(25) NOT NULL,
swen439p1(>     BankName CHAR(30) NOT NULL,
swen439p1(>     City CHAR(30) NOT NULL,
swen439p1(>     DateTemp DATE NOT NULL,
swen439p1(>     ShareTemp REAL NOT NULL
swen439p1(> );
CREATE TABLE
swen439p1=> \copy TempAccomplices FROM ~/git/439p1/data2022/accomplices_22.data

COPY 76
```

Then, extract the RobberId column from Robbers table, for remaining cols, use TempAccomplices table directly. We can see that 76 rows are inserted successfully.

```
INSERT INTO Accomplices(RobberId, BankName, City, Date, Share)
SELECT robber.RobberId, temp.BankName, temp.City, temp.DateTemp, temp.ShareTemp
FROM Robbers robber, TempAccomplices temp
```

WHERE robber.Nickname = temp.RobberNicknameTemp;

```
swen439p1=> INSERT INTO Accomplices(RobberId, BankName, City, Date, Share)
swen439p1-> SELECT robber.RobberId,
swen439p1->     temp.BankName,
swen439p1->     temp.City,
swen439p1->     temp.DateTemp,
swen439p1->     temp.ShareTemp
swen439p1-> FROM Robbers robber,
swen439p1->     TempAccomplices temp
swen439p1-> WHERE robber.Nickname = temp.RobberNicknameTemp;
INSERT 0 76
```

```
swen439p1=> select * from  accomplices
swen439p1-> \g
 robberid |          bankname          |            city            |    date    |  share
----------+----------------------------+----------------------------+------------+---------
        1 | Bad Bank                   | Chicago                    | 2017-02-02 |    3010
        1 | NXP Bank                   | Chicago                    | 2019-01-08 |    6406
        1 | Loanshark Bank             | Evanston                   | 2019-02-28 |    4997
        1 | Loanshark Bank             | Chicago                    | 2019-03-30 |    4201
        1 | Inter-Gang Bank            | Evanston                   | 2016-02-16 |   12103
        1 | Inter-Gang Bank            | Evanston                   | 2018-02-14 |    8769
        2 | NXP Bank                   | Chicago                    | 2019-01-08 |    2300
        3 | Penny Pinchers             | Evanston                   | 2016-08-30 |   16500
        3 | Loanshark Bank             | Evanston                   | 2019-02-28 |    4997
```

For the tail, we can see that there're 76 rows:

```
       23 | NXP Bank                   | Chicago                    | 2019-01-08 |    6406
       24 | PickPocket Bank            | Evanston                   | 2018-01-30 |     500
       24 | PickPocket Bank            | Evanston                   | 2016-03-30 |    2000
       24 | PickPocket Bank            | Chicago                    | 2015-09-21 |     681
       24 | Penny Pinchers             | Evanston                   | 2017-10-30 |    3000
       24 | Loanshark Bank             | Chicago                    | 2019-03-30 |    4201
       24 | Gun Chase Bank             | Evanston                   | 2016-04-30 |    3282
(76 rows)
```

Finally, drop/delete all these temp tables for clarity.

```
swen439p1=> drop table tempskills

swen439p1-> ;
DROP TABLE
swen439p1=> drop table temphasaccounts;
DROP TABLE
swen439p1=> drop table tempaccomplics;
ERROR:  table "tempaccomplics" does not exist
swen439p1=> drop table tempaccomplices;
DROP TABLE
swen439p1=>
```

2. A brief description of the order in which you have implemented the tables of the *RobbersGang* database. Justify your answer. [3 marks]

First, I need to implement tables that are independent, which means the table do not have foreign keys that refer to other tables and do not have other tables' foreign key refer to itself. After the check, I find that for this RobbersGane database, we don't have this kind of independent tables.

Then, due to the feature of the foreign key, I first implement tables that other tables' foreign keys will refer to, then I implement these tables where their foreign key refer to other tables. For instances, Banks table is implemented before Robberies, Plans table, Robbers and Skills tables are implemented before HasSkills table, etc. It is essential since these later implemented tables depends on these earlier implemented tables. If it is the other way around, then the implementations will be rejected due to it violate the referential integrity constraints that the values of the attributes of a foreign key do not match any tuple in the other relation. For example, if we implement the HasSkills table before Robbers and Skills tables, then it cannot find the corresponding tuples in Robbers and Skills tables which result that it refers to nothing.

QUESTION 3: Checking your Database [10 marks]

You are now expected to check that your database design enforces all the mentioned consistency checks. Use SQL as a data manipulation language to perform the tasks listed below.

For each task, record the feedback from PostgreSQL. **If your database is created correctly, you should receive error messages from PostgreSQL**.

For each task, briefly state which kind of constraint it violates. If no error message is returned, then your database is probably not yet correct. You should at least say what the constraint ought to be, even if you cannot implement it.

Please note: If you give names to your constraints, the error messages are more informative.

The tasks:

1. Insert the following tuple into the *Skills* table:
    a. (21, 'Driving')

INSERT INTO Skills(skillId, Description)
VALUES(21, 'Driving');

```
swen439p1=> INSERT INTO Skills(skillId, Description)
swen439p1-> VALUES(21, 'Driving');
ERROR:  duplicate key value violates unique constraint "description_unique"
DETAIL:  Key (description)=(Driving                ) already exists.
```

It violates the unique constraint of the Description column.

2. Insert the following tuples into the *Banks* table:
    a. ('Loanshark Bank', 'Evanston', 100, 'very good')

```
INSERT INTO Banks(BankName, City, NoAccounts, Security)
VALUES('Loanshark Bank', 'Evanston', 100, 'very good');
```

```
swen439p1=> INSERT INTO Banks(BankName, City, NoAccounts, Security)
swen439p1-> VALUES('Loanshark Bank', 'Evanston', 100, 'very good');
ERROR:  duplicate key value violates unique constraint "bankspk"
DETAIL:  Key (bankname, city)=(Loanshark Bank              , Evanston
            ) already exists.
```

It violates the unique constraint of the primary key (BankName, City).

    b. ('EasyLoan Bank', 'Evanston', -5, 'excellent')

INSERT INTO Banks(BankName, City, NoAccounts, Security)
VALUES('EasyLoan Bank', 'Evanston', -5, 'excellent');

```
swen439p1=> INSERT INTO Banks(BankName, City, NoAccounts, Security)
swen439p1-> VALUES('EasyLoan Bank', 'Evanston', -5, 'excellent');
ERROR:  new row for relation "banks" violates check constraint "noaccountscheck"
DETAIL:  Failing row contains (EasyLoan Bank               , Evanston
            , -5, excellent      ).
```

It violates the "noaccountscheck" constraint that I personally add, which means the number of accounts must be a non-negative integer, but in there it is negative number: -5.

    c. ('EasyLoan Bank', 'Evanston', 100, 'poor')

INSERT INTO Banks(BankName, City, NoAccounts, Security)
VALUES('EasyLoan Bank', 'Evanston', 100, 'poor');

```
swen439p1=> INSERT INTO Banks(BankName, City, NoAccounts, Security)
swen439p1-> VALUES('EasyLoan Bank', 'Evanston', 100, 'poor');
ERROR:  new row for relation "banks" violates check constraint "banks_security_c
heck"
DETAIL:  Failing row contains (EasyLoan Bank                , Evanston
           , 100, poor          ).
```

It violates the "banks_security_check" constraint that I personally add, which means the security value of the inserted tuple must be one of predefined values, which is one of 'weak', 'good', 'very good', 'excellent'

3. Insert the following tuple into the *Robberies* table:
    a. ('NXP Bank', 'Chicago', '2019-01-08', 1000)

INSERT INTO Robberies(BankName, City, Date, Amount)
VALUES('NXP Bank', 'Chicago', '2019-01-08', 1000);

```
swen439p1=> INSERT INTO Robberies(BankName, City, Date, Amount)
swen439p1-> VALUES('NXP Bank', 'Chicago', '2019-01-08', 1000);
ERROR:  duplicate key value violates unique constraint "robberiespk"
DETAIL:  Key (bankname, city, date)=(NXP Bank                       , Chicago
                  , 2019-01-08) already exists.
```

It violates the unique constraint of the primary key (BankName, City, Date).

4. Delete the following tuple from the *Skills* table:
    a. (1, 'Driving')

DELETE FROM Skills
WHERE SkillId = 1
  AND Description = 'Driving';

```
swen439p1=> DELETE FROM Skills
swen439p1-> WHERE SkillId = 1
swen439p1->     AND Description = 'Driving';
DELETE 0
```

From above screenshot, we can see that the error message is not reported since we do not have an existing tuple that perfectly match the condition where SkillId=1 AND Description= 'Driving'.
For further check, I decide to try to delete an existing tuple and it should return the error message that it violates the foreign key constraint, to be more specific, it violates the referential integrity constraint. We can observe this from below.

DELETE FROM Skills
WHERE SkillId = 1
  AND Description = 'Planning';

```
swen439p1=> DELETE FROM Skills
swen439p1-> WHERE SkillId = 1
swen439p1->     AND Description = 'Planning';
ERROR:  update or delete on table "skills" violates foreign key constraint "hass
kills_skillid_fkey" on table "hasskills"
DETAIL:  Key (skillid)=(1) is still referenced from table "hasskills".
```

5. Delete the following tuples from the *Banks* table:
    a. ('PickPocket Bank', 'Evanston', 2000, 'very good')

DELETE FROM Banks WHERE BankName = 'PickPocket Bank' AND City = 'Evanston' AND NoAccounts = 2000 AND Security = 'very good';

```
swen439p1=> DELETE FROM Banks
swen439p1-> WHERE BankName = 'PickPocket Bank'
swen439p1->     AND City = 'Evanston'
swen439p1->     AND NoAccounts = 2000
swen439p1->     AND Security = 'very good';
ERROR:  update or delete on table "banks" violates foreign key constraint "robbe
riesfk" on table "robberies"
DETAIL:  Key (bankname, city)=(PickPocket Bank              , Evanston
           ) is still referenced from table "robberies".
swen439p1=>
```

It violates the foreign key constraint, to be more specific, it violates the referential integrity constraint. It means the foreign key of the Robberies table refer to this tuple, so delete it will affect tuples in Robberies table point to nothing.

6. Delete the following tuple from the *Robberies* table:
    a. (‘Loanshark Bank’, ‘Chicago’, ‘’, ‘’)

DELETE FROM Robberies WHERE BankName = 'Loanshark Bank' AND City = 'Chicago' AND Date = '' AND Amount = '';
DELETE FROM Robberies WHERE BankName = 'Loanshark Bank' AND City = 'Chicago' AND Date = NULL AND Amount = NULL;

```
swen439p1=> DELETE FROM Robberies
swen439p1-> WHERE BankName = 'Loanshark Bank'
swen439p1->     AND City = 'Chicago'
swen439p1->     AND Date = NULL
swen439p1->     AND Amount = NULL;
DELETE 0
swen439p1=> DELETE FROM Robberies
swen439p1-> WHERE BankName = 'Loanshark Bank'
swen439p1->     AND City = 'Chicago'
swen439p1->     AND Date = ''
swen439p1->     AND Amount = '';
ERROR:  invalid input syntax for type date: ""
LINE 4:     AND Date = ''
                       ^
```

For this one, it is a little bit tricky since the given tuple specify that the value of 2 columns is empty. Therefore, I try to replace them with NULL and quotes,.
We can see that for NULL, it just report DELETE 0, which means there is no matched tuple that can be deleted. For the quotes, it just report the error that the input syntax error.

In the following two tasks, we assume that there is a robber with Id 3, but no robber with Id 333.

7. Insert the following tuples into the *Robbers* table:
    a. (1, ‘Shotgun’, 70, 0)

```
INSERT INTO Robbers(RobberId, NickName, Age, NoYears)
VALUES (1, 'Shotgun', 70, 0);
```

```
swen439p1=> INSERT INTO Robbers(RobberId, NickName, Age, NoYears)
swen439p1-> VALUES (1, 'Shotgun', 70, 0);
ERROR:  duplicate key value violates unique constraint "robbers_pkey"
DETAIL:  Key (robberid)=(1) already exists.
```

It violates the unique constraint of the primary key (RobberId).

b. (333, ‘Jail Mouse’, 25, 35)

```
INSERT INTO Robbers(RobberId, NickName, Age, NoYears)
VALUES(333, 'Jail Mouse', 25, 35);
```

It violates the “agecheck” constraint that I personally add, which means it is not possible for a Robber to be in prison for more years than the Robber have been alive!

```
swen439p1=> INSERT INTO Robbers(RobberId, NickName, Age, NoYears)
swen439p1-> VALUES(333, 'Jail Mouse', 25, 35);
ERROR:  new row for relation "robbers" violates check constraint "agecheck"
DETAIL:  Failing row contains (333, Jail Mouse                , 25, 35).
```

8. Insert the following tuples into the *HasSkills* table:

a. (1, 7, 1, ‘A+’)

For this one, I first insert this tuple into the HasSkills table successfully, which confuse me for a while, I even think that there is nothing wrong with this tuple. However, then, by looking at existing tuples more carefully, I observe that the Preference column value is unique for each RobberId, which means a Robber can have lots of skills, but for the preference ranking of different skills, it cannot be the same like 2 skills have the same preference ranking. Therefore, I add a UNIQUE constraint to the Preference column to fix this bug.

- ALTER TABLE Hasskills
- ADD CONSTRAINT id_preference_unique UNIQUE (RobberId, Preference);

```
swen439p1=> ALTER TABLE Hasskills
swen439p1-> ADD CONSTRAINT  id_preference_unique UNIQUE (RobberId, Preference);
ALTER TABLE
```

Then, after delete it, **insert this tuple again to see whether it works:**

```
INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
VALUES (1, 7, 1, 'A+');
```

We can see it violates the unique constraint that (robberid,preference) must be unique.

```
swen439p1=> INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
swen439p1-> VALUES (1, 7, 1, 'A+');
ERROR:  duplicate key value violates unique constraint "id_preference_unique"
DETAIL:  Key (robberid, preference)=(1, 1) already exists.
```

b. (1, 2, 0, ‘A’)

```
INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
VALUES (1, 2, 0, 'A');
```

```
swen439p1=> INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
swen439p1-> VALUES (1, 2, 0, 'A');
ERROR:  duplicate key value violates unique constraint "hasskillspk"
DETAIL:  Key (robberid, skillid)=(1, 2) already exists.
```

It violates the unique constraint of the primary key(RobberId, SkillId).

c. (333, 1, 1, ‘B-’)

```
INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
VALUES (333, 1, 1, 'B-');
```

```
swen439p1=> INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
swen439p1-> VALUES (333, 1, 1, 'B-');
ERROR:  insert or update on table "hasskills" violates foreign key constraint "hasskills_robberid_fkey"
DETAIL:  Key (robberid)=(333) is not present in table "robbers".
```

It violates the foreign key constraint that this robberId:333 does not exist in the table Robbers.

d. (3, 20, 3, ‘B+’)

```
INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
VALUES (3, 20, 3, 'B+');
```

```
swen439p1=> INSERT INTO HasSkills(RobberId, SkillId, Preference, Grade)
swen439p1-> VALUES (3, 20, 3, 'B+');
ERROR:  insert or update on table "hasskills" violates foreign key constraint "hasskills_skillid_fkey"
DETAIL:  Key (skillid)=(20) is not present in table "skills".
```

It violates the foreign key constraint that this skillId:20 does not exist in the table Skills.

In the following task, we assume that Al Capone has the robber Id 1. If Al Capone has a different Id in your database, then please change the first entry in the following tuple to be your Id of Al Capone.

9. Delete the following tuple from the *Robbers* table:

   a. (1, ‘Al Capone’, 31, 2).

```
DELETE FROM Robbers
WHERE RobberId = 1
   AND Nickname = 'Al Capone'
   AND Age = 31
   AND NoYears = 2;
```

```
swen439p1=> DELETE FROM Robbers
swen439p1-> WHERE RobberId = 1
swen439p1->     AND Nickname = 'Al Capone'
swen439p1->     AND Age = 31
swen439p1->     AND NoYears = 2;
ERROR:  update or delete on table "robbers" violates foreign key constraint "has
skills_robberid_fkey" on table "hasskills"
DETAIL:  Key (robberid)=(1) is still referenced from table "hasskills".
```

It violates the foreign key constraint on the HasSkills table. For the detail, we can see that this tuple is still referenced from the table ‘hasskills’, so delete this tuple will result that tuples in HasSkills table reference to nothing.

Your answer to Question 3 should include:

- Your SQL statements for each task, the feedback from PostgreSQL, and the constraint that has been violated in case of an error message.

QUESTION 4: Simple Database Queries [24 marks]

You are now expected to use SQL as a query language to retrieve data from the database. Perform the series of tasks listed below.

For each task, record the answer from PostgreSQL.

The tasks:

1. Retrieve *BankName* and *City* of all banks that have never been robbed. [3 marks]

```
SELECT bank.BankName,
   bank.City
FROM Banks bank
WHERE NOT EXISTS(
      SELECT 1
      FROM Robberies robbery
      WHERE robbery.BankName = bank.BankName
         AND robbery.City = bank.City
   );
```

```
swen439p1=> SELECT bank.BankName,
swen439p1->     bank.City
swen439p1-> FROM Banks bank
swen439p1-> WHERE NOT EXISTS(
swen439p1(>         SELECT 1
swen439p1(>         FROM Robberies robbery
swen439p1(>         WHERE robbery.BankName = bank.BankName
swen439p1(>             AND robbery.City = bank.City
swen439p1(>     );
           bankname            |             city
-------------------------------+---------------------------
 Bankrupt Bank                 | Evanston
 Loanshark Bank                | Deerfield
 Inter-Gang Bank               | Chicago
 NXP Bank                      | Evanston
 Dollar Grabbers               | Chicago
 Gun Chase Bank                | Burbank
 PickPocket Bank               | Deerfield
 Hidden Treasure               | Chicago
 Outside Bank                  | Chicago
(9 rows)
```

2. Retrieve *RobberId*, *Nickname*, *Age*, and all skill descriptions of all robbers who are older than *40* years old. [3 marks]

```
SELECT robber.RobberId,
   robber.NickName,
   robber.Age,
   skill.Description
FROM Robbers robber,
   Skills skill,
   HasSkills hasSkill
WHERE robber.Age > 40
   AND robber.RobberId = hasSkill.RobberId
   AND skill.SkillId = hasSkill.SkillId;
```

```
swen439p1=> SELECT robber.RobberId,
swen439p1->     robber.NickName,
swen439p1->     robber.Age,
swen439p1->     skill.Description
swen439p1-> FROM Robbers robber,
swen439p1->     Skills skill,
swen439p1->     HasSkills hasSkill
swen439p1-> WHERE robber.Age > 40
swen439p1->     AND robber.RobberId = hasSkill.RobberId
swen439p1->     AND skill.SkillId = hasSkill.SkillId;
 robberid |         nickname          | age |       description
----------+---------------------------+-----+--------------------
        2 | Bugsy Malone              |  42 | Explosives
        3 | Lucky Luchiano            |  42 | Lock-Picking
        3 | Lucky Luchiano            |  42 | Driving
        4 | Anastazia                 |  48 | Guarding
        7 | Dutch Schulz              |  64 | Lock-Picking
        7 | Dutch Schulz              |  64 | Driving
        9 | Calamity Jane             |  44 | Gun-Shooting
       12 | Moe Dalitz                |  41 | Safe-Cracking
       15 | Boo Boo Hoff              |  54 | Planning
       16 | King Solomon              |  74 | Planning
       17 | Bugsy Siegel              |  48 | Driving
       17 | Bugsy Siegel              |  48 | Guarding
       18 | Vito Genovese             |  66 | Scouting
       18 | Vito Genovese             |  66 | Cooking
       18 | Vito Genovese             |  66 | Eating
(15 rows)
```

3. Retrieve *BankName* and city of all banks where Al Capone has an account. The answer should list every bank at most once. [3 marks]

```
SELECT DISTINCT bank.BankName,
   bank.City
FROM Banks bank
   INNER JOIN HasAccounts acc USING(BankName, City)
   INNER JOIN Robbers robber USING(RobberId)
WHERE acc.RobberId = robber.RobberId
   AND robber.NickName = 'Al Capone';
```

```
swen439p1=> SELECT DISTINCT bank.BankName,
swen439p1->     bank.City
swen439p1-> FROM Banks bank
swen439p1->     INNER JOIN HasAccounts acc USING(BankName, City)
swen439p1->     INNER JOIN Robbers robber USING(RobberId)
swen439p1-> WHERE acc.RobberId = robber.RobberId
swen439p1->     AND robber.NickName = 'Al Capone';
           bankname            |             city
-------------------------------+-------------------------------
 Bad Bank                      | Chicago
 Inter-Gang Bank               | Evanston
 NXP Bank                      | Chicago
(3 rows)
```

4. Retrieve *BankName* and *City* and *NoAccounts* of all banks that have no branch in Chicago. The answer should be sorted in increasing order of the number of accounts. [3 marks]

```
SELECT BankName,
   City,
   NoAccounts
FROM Banks bank
WHERE City != 'Chicago'
ORDER BY NoAccounts ASC;
```

```
swen439p1=> SELECT BankName,
swen439p1->     City,
swen439p1->     NoAccounts
swen439p1-> FROM Banks bank
swen439p1-> WHERE City != 'Chicago'
swen439p1-> ORDER BY NoAccounts ASC;
           bankname           |             city             | noaccounts
------------------------------+------------------------------+------------
 Gun Chase Bank               | Burbank                      |       1999
 PickPocket Bank              | Evanston                     |       2000
 PickPocket Bank              | Deerfield                    |       6565
 Penny Pinchers               | Evanston                     |     130013
 Bankrupt Bank                | Evanston                     |     444000
 Inter-Gang Bank              | Evanston                     |     555555
 Gun Chase Bank               | Evanston                     |     656565
 NXP Bank                     | Evanston                     |     656565
 Dollar Grabbers              | Evanston                     |     909090
 Loanshark Bank               | Deerfield                    |    3456789
 Loanshark Bank               | Evanston                     |    7654321
(11 rows)
```

5. Retrieve *RobberId, Nickname* and individual total "earnings" of those robbers who have earned more than $40,000 by robbing banks. The answer should be sorted in decreasing order of the total earnings. [3 marks]

```
SELECT robber.RobberId,
   robber.Nickname,
   SUM(accomplice.Share) AS TotalEarning
FROM Accomplices accomplice,
   Robbers robber
WHERE robber.RobberId = accomplice.RobberId
GROUP BY robber.RobberId,
   robber.Nickname
HAVING SUM(accomplice.Share) > 40000
ORDER BY TotalEarning DESC;
```

```
swen439p1=> SELECT robber.RobberId,
swen439p1->     robber.Nickname,
swen439p1->     SUM(accomplice.Share) AS TotalEarning
swen439p1-> FROM Accomplices accomplice,
swen439p1->     Robbers robber
swen439p1-> WHERE robber.RobberId = accomplice.RobberId
swen439p1-> GROUP BY robber.RobberId,
swen439p1->     robber.Nickname
swen439p1-> HAVING SUM(accomplice.Share) > 40000
swen439p1-> ORDER BY TotalEarning DESC;
 robberid |         nickname          | totalearning
----------+---------------------------+--------------
        5 | Mimmy The Mau Mau         |        70000
       15 | Boo Boo Hoff              |      61447.6
       16 | King Solomon              |      59725.8
       17 | Bugsy Siegel              |      52601.1
        3 | Lucky Luchiano            |        42667
       10 | Bonnie                    |        40085
(6 rows)
```

6. Retrieve *RobberId*, *NickName*, and the *Number of Years* in prison for all robbers who were in prison for more than ten years. [3 marks]

```
SELECT RobberId,
    NickName,
    NoYears
FROM Robbers
WHERE NoYears > 10;
```

```
swen439p1=> SELECT RobberId,
swen439p1->     NickName,
swen439p1->     NoYears
swen439p1-> FROM Robbers
swen439p1-> WHERE NoYears > 10;
 robberid |         nickname          | noyears
----------+---------------------------+---------
        2 | Bugsy Malone              |      15
        3 | Lucky Luchiano            |      15
        4 | Anastazia                 |      15
        6 | Tony Genovese             |      16
        7 | Dutch Schulz              |      31
       15 | Boo Boo Hoff              |      13
       16 | King Solomon              |      43
       17 | Bugsy Siegel              |      13
(8 rows)
```

7. Retrieve *RobberId*, *Nickname* and the *Number of Years* **not** spent in prison for all robbers who spent more than half of their life in prison. [3 marks]

```
SELECT robber.RobberId,
   robber.Nickname,
   (robber.Age - robber.NoYears) AS NotInPrisonYears
FROM Robbers robber
WHERE robber.NoYears > (robber.Age / 2);
```

```
swen439p1=> SELECT robber.RobberId,
swen439p1->     robber.Nickname,
swen439p1->     (robber.Age - robber.NoYears) AS NotInPrisonYears
swen439p1-> FROM Robbers robber
swen439p1-> WHERE robber.NoYears > (robber.Age / 2);
 robberid |         nickname          | notinprisonyears
----------+---------------------------+------------------
        6 | Tony Genovese             |               12
       16 | King Solomon              |               31
(2 rows)
```

8. Retrieve the *Description* of all skills together with *RobberId* and *NickName* of all robbers who possess this skill. The answer should be ordered by skill description. [3 marks]

```
SELECT robber.RobberId,
   robber.NickName,
   skill.Description
FROM Robbers robber
   INNER JOIN HasSkills hasSkill USING (RobberId)
   INNER JOIN Skills skill USING(SkillId)
ORDER BY skill.Description;
```

```
swen439p1=> SELECT robber.RobberId, robber.NickName, skill.Description
swen439p1-> FROM Robbers robber INNER JOIN HasSkills hasSkill USING (RobberId) INNER JOIN Skills skill USING(SkillId)
swen439p1-> ORDER BY skill.Description;
 robberid |         nickname          |        description
----------+---------------------------+---------------------------
       18 | Vito Genovese             | Cooking
       20 | Longy Zwillman            | Driving
       17 | Bugsy Siegel              | Driving
        3 | Lucky Luchiano            | Driving
        5 | Mimmy The Mau Mau         | Driving
        7 | Dutch Schulz              | Driving
       23 | Lepke Buchalter           | Driving
       18 | Vito Genovese             | Eating
        6 | Tony Genovese             | Eating
       24 | Sonny Genovese            | Explosives
        2 | Bugsy Malone              | Explosives
        4 | Anastazia                 | Guarding
       23 | Lepke Buchalter           | Guarding
       17 | Bugsy Siegel              | Guarding
        9 | Calamity Jane             | Gun-Shooting
       21 | Waxey Gordon              | Gun-Shooting
       24 | Sonny Genovese            | Lock-Picking
        3 | Lucky Luchiano            | Lock-Picking
        7 | Dutch Schulz              | Lock-Picking
        8 | Clyde                     | Lock-Picking
       22 | Greasy Guzik              | Lock-Picking
       13 | Mickey Cohen              | Money Counting
       14 | Kid Cann                  | Money Counting
       19 | Mike Genovese             | Money Counting
       15 | Boo Boo Hoff              | Planning
        1 | Al Capone                 | Planning
        8 | Clyde                     | Planning
        5 | Mimmy The Mau Mau         | Planning
       16 | King Solomon              | Planning
        1 | Al Capone                 | Preaching
       10 | Bonnie                    | Preaching
       22 | Greasy Guzik              | Preaching
       11 | Meyer Lansky              | Safe-Cracking
       12 | Moe Dalitz                | Safe-Cracking
        1 | Al Capone                 | Safe-Cracking
       24 | Sonny Genovese            | Safe-Cracking
        8 | Clyde                     | Scouting
       18 | Vito Genovese             | Scouting
(38 rows)
```

Your answer to Question 4 should include:

- Your SQL statement for each task, and the answer from PostgreSQL.
- Also, submit your SQL queries, with each query (just SQL code) as a separate .sql file. Name files in the following way: Query4_TaskX.sql, where X stands for the task number 1, 2, ...

QUESTION 5: Complex Database Queries [20 marks]

You are again expected to use SQL as a query language to retrieve data from the database to perform the tasks listed below. For each of the following tasks, you are asked to construct SQL queries.

1. Retrieve *BankName* and *City* of all banks that were not robbed in the year, in which there were robbery plans for that bank. [4 marks]

```
SELECT DISTINCT plan.BankName, plan.City
FROM Plans plan
WHERE NOT EXISTS(
    SELECT 1
    FROM Robberies robbery
    WHERE extract(
        year
        from plan.PlannedDate
    ) = extract(
        year
        from robbery.Date
    )
    AND plan.BankName = robbery.BankName
    AND plan.City = robbery.City
);
```

```
swen439p1=> SELECT DISTINCT plan.BankName,
swen439p1->     plan.City
swen439p1-> FROM Plans plan
swen439p1-> WHERE NOT EXISTS(
swen439p1(>         SELECT 1
swen439p1(>         FROM Robberies robbery
swen439p1(>         WHERE extract(
swen439p1(>                 year
swen439p1(>                 from plan.PlannedDate
swen439p1(>             ) = extract(
swen439p1(>                 year
swen439p1(>                 from robbery.Date
swen439p1(>             )
swen439p1(>             AND plan.BankName = robbery.BankName
swen439p1(>             AND plan.City = robbery.City
swen439p1(>     );
          bankname            |             city
------------------------------+------------------------------
 Bad Bank                     | Chicago
 Dollar Grabbers              | Chicago
 Gun Chase Bank               | Evanston
 Hidden Treasure              | Chicago
 Inter-Gang Bank              | Evanston
 Loanshark Bank               | Deerfield
 PickPocket Bank              | Chicago
 PickPocket Bank              | Deerfield
(8 rows)
```

2. Retrieve *RobberId* and *Nickname* of all robbers who never robbed the banks at which they have an account. [4 marks]

```
SELECT DISTINCT robber.RobberId, robber.Nickname
FROM Robbers robber
    INNER JOIN HasAccounts hasAccount USING(RobberId)
WHERE NOT EXISTS(
        SELECT 1
        FROM Accomplices accomplice
        WHERE accomplice.RobberId = robber.RobberId
            AND accomplice.BankName = hasAccount.BankName
            AND accomplice.City = hasAccount.City
```

```
        )
    ORDER BY robber.RobberId;
```

```
swen439p1=> SELECT DISTINCT robber.RobberId,
swen439p1->     robber.Nickname
swen439p1-> FROM Robbers robber
swen439p1->     INNER JOIN HasAccounts hasAccount USING(RobberId)
swen439p1-> WHERE NOT EXISTS(
swen439p1(>         SELECT 1
swen439p1(>         FROM Accomplices accomplice
swen439p1(>         WHERE accomplice.RobberId = robber.RobberId
swen439p1(>             AND accomplice.BankName = hasAccount.BankName
swen439p1(>             AND accomplice.City = hasAccount.City
swen439p1(>     )
swen439p1-> ORDER BY robber.RobberId;
 robberid |         nickname
----------+--------------------------
        2 | Bugsy Malone
        3 | Lucky Luchiano
        4 | Anastazia
        7 | Dutch Schulz
        9 | Calamity Jane
       12 | Moe Dalitz
       13 | Mickey Cohen
       14 | Kid Cann
       15 | Boo Boo Hoff
       18 | Vito Genovese
       19 | Mike Genovese
       21 | Waxey Gordon
       23 | Lepke Buchalter
       24 | Sonny Genovese
(14 rows)
```

3. Retrieve *RobberId*, *Nickname*, and *Description* of the first *preferred* skill of all robbers who have two or more skills. [4 marks]

```
SELECT hs.RobberId,
    robber.Nickname,
        -- hs.Preference,
        -- temp.skillNums,
    skill.Description
FROM Robbers robber
    NATURAL JOIN Hasskills hs
    NATURAL JOIN Skills skill
    NATURAL JOIN(
        -- remove robbers who do NOT have 2 or more skills
        SELECT robberid,
            COUNT(robberid) AS skillNums
        from HasSkills
        GROUP BY robberid
        HAVING COUNT(robberid) >= 2
    ) temp
WHERE hs.Preference = 1
ORDER BY hs.RobberId;
```

```
swen439p1=> SELECT hs.RobberId,
swen439p1->     robber.Nickname,
swen439p1->     -- hs.Preference,
swen439p1->     -- temp.skillNums,
swen439p1->     skill.Description
swen439p1-> FROM Robbers robber
swen439p1->     NATURAL JOIN Hasskills hs
swen439p1->     NATURAL JOIN Skills skill
swen439p1->     NATURAL JOIN(
swen439p1(>         -- remove robbers who do NOT have 2 or more skills
swen439p1(>         SELECT robberid,
swen439p1(>             COUNT(robberid) AS skillNums
swen439p1(>         from HasSkills
swen439p1(>         GROUP BY robberid
swen439p1(>         HAVING COUNT(robberid) >= 2
swen439p1(>     ) temp
swen439p1-> WHERE hs.Preference = 1
swen439p1-> ORDER BY hs.RobberId;
 robberid |         nickname          |        description
----------+---------------------------+---------------------------
        1 | Al Capone                 | Planning
        3 | Lucky Luchiano            | Lock-Picking
        5 | Mimmy The Mau Mau         | Planning
        7 | Dutch Schulz              | Lock-Picking
        8 | Clyde                     | Lock-Picking
       17 | Bugsy Siegel              | Driving
       18 | Vito Genovese             | Scouting
       22 | Greasy Guzik              | Preaching
       23 | Lepke Buchalter           | Driving
       24 | Sonny Genovese            | Explosives
(10 rows)
```

4. Retrieve *BankName*, *City* and *Date* of all robberies in the city that observes the highest *Share* among all robberies. [4 marks]

```
SELECT robbery.BankName,
   robbery.City,
   robbery.Date,
   temp.HighestShare
FROM Robberies robbery
   NATURAL JOIN(
      SELECT city,
         MAX(amount) AS HighestShare
      FROM Robberies
      GROUP BY city
      ORDER BY MAX(amount)
   ) temp
WHERE robbery.amount = temp.HighestShare
ORDER BY robbery.City;
```

```
swen439p1=> SELECT  robbery.BankName,
swen439p1->     robbery.City,
swen439p1->     robbery.Date,
swen439p1->     temp.HighestShare
swen439p1-> FROM Robberies robbery
swen439p1->     NATURAL JOIN(
swen439p1(>         SELECT city,
swen439p1(>             MAX(amount) AS HighestShare
swen439p1(>         FROM Robberies
swen439p1(>         GROUP BY city
swen439p1(>         ORDER BY MAX(amount)
swen439p1(>     ) temp
swen439p1-> WHERE robbery.amount = temp.HighestShare
swen439p1-> ORDER BY robbery.City;
          bankname           |             city             |    date    | highestshare
-----------------------------+------------------------------+------------+--------------
 Loanshark Bank              | Chicago                      | 2017-11-09 |        41000
 Penny Pinchers              | Evanston                     | 2016-08-30 |      99000.8
(2 rows)
```

5. Retrieve *BankName* and *City* of all banks that were robbed by all robbers. [4 marks]

```
swen439p1=> SELECT a.bankName,
swen439p1->     a.city
swen439p1-> FROM Robbers r
swen439p1->     NATURAL JOIN Accomplices a
swen439p1->     NATURAL JOIN(
swen439p1(>         --  annoying, spend me lots of time,
swen439p1(>         -- it will find the number of the bank that has been robb
ed from different robbers. WHich means no matter how many times a robber rob
this specificed banks, it will count only once.
swen439p1(>         SELECT a1.BankName,
swen439p1(>             a1.City,
swen439p1(>             COUNT(DISTINCT a1.RobberId) AS robbedCounts
swen439p1(>         FROM Accomplices a1
swen439p1(>         GROUP BY BankName,
swen439p1(>             City
swen439p1(>         ORDER BY robbedCounts
swen439p1(>     ) temp
swen439p1->     NATURAL JOIN (
swen439p1(>         Select Count(*) as robberNumbers
swen439p1(>         From Robbers
swen439p1(>     ) temp1
swen439p1-> WHERE temp.robbedCounts = temp1.robberNumbers
swen439p1-> ORDER BY a.robberId;
 bankname | city
----------+------
(0 rows)
```

BELOW is the SQL statement for query 5, task 5:

```sql
SELECT a.bankName,
    a.city
FROM Robbers r
    NATURAL JOIN Accomplices a
    NATURAL JOIN(
        --  annoying, spend me lots of time,
        -- it will find the number of the bank that has been robbed from different robbers.
WHich means no matter how many times a robber rob this specificed banks, it will count only
once.
        SELECT a1.BankName,
            a1.City,
            COUNT(DISTINCT a1.RobberId) AS robbedCounts
        FROM Accomplices a1
        GROUP BY BankName,
            City
        ORDER BY robbedCounts
    ) temp
    NATURAL JOIN (
        Select Count(*) as robberNumbers
        From Robbers
    ) temp1
WHERE temp.robbedCounts = temp1.robberNumbers
ORDER BY a.robberId;
```

Your answer to Question 5 should include:

- Your SQL statement for each task, and the answer from PostgreSQL.
- Also, submit your SQL queries, with each query (just SQL code) as a separate .sql file. Name files in the following way: Query5_TaskX.sql, where X stands for the task number 1, 2, ...

QUESTION 6: Complex Database Queries [16 marks]

You are again expected to use SQL as a query language to retrieve data from the database to perform the tasks listed below. **For each of the following tasks, you are asked to construct SQL queries in two ways**: using the stepwise approach, and as a single nested SQL query.

The stepwise approach of computing complex queries consists of a sequence of basic (not nested) SQL queries. The results of each query must be put into a virtual or a materialised view (with the CREATE VIEW … AS SELECT … command, or the CREATE TABLE … AS SELECT … command). The output of the last query should be the requested result. The first query in the sequence uses the base relations as input. Each subsequent query in the sequence may use the base relations and/or the intermediate results of the preceding views as input.

1. *The police department wants to know which robbers are most active, but were never penalised.*

   Construct a view that contains the *Nicknames* of all robbers who participated in more robberies than the average, but spent no time in prison. The answer should be sorted in decreasing order of the individual total "earnings" of the robbers. [8 marks]

Stepwise approach:

First, create a view that contains the robberies number per robber.

```
CREATE VIEW RobberiesPerRobber AS(
   SELECT RobberId,
      COUNT(RobberId) AS robberyTimes
   FROM Accomplices a
   GROUP BY robberid
   ORDER BY Robberid
);
```

```
swen439p1=> CREATE VIEW RobberiesPerRobber AS(
swen439p1(>     SELECT RobberId,
swen439p1(>         COUNT(RobberId) AS robberyTimes
swen439p1(>     FROM Accomplices a
swen439p1(>     GROUP BY robberid
swen439p1(>     ORDER BY Robberid
swen439p1(> );
CREATE VIEW
```

Then, create a View that calculate average robberies by dividing the all robbery times to all robbers.

```
CREATE VIEW averageRobberies AS(
   SELECT SUM(perRobberView.robberyTimes) AS TotalRobberies,
      COUNT(robber.*) AS TotalRobbers,
      (
         SUM(perRobberView.robberyTimes) / COUNT(robber.*)
      ) AS AVGRobberies
   FROM Robbers robber
      NATURAL JOIN RobberiesPerRobber perRobberView
);
```

```
swen439p1=> CREATE VIEW averageRobberies AS(
swen439p1(>     SELECT SUM(perRobberView.robberyTimes) AS TotalRobberies,
swen439p1(>         COUNT(robber.*) AS TotalRobbers,
swen439p1(>         (
swen439p1(>             SUM(perRobberView.robberyTimes) / COUNT(robber.*)
swen439p1(>         ) AS AVGRobberies
swen439p1(>     FROM Robbers robber
swen439p1(>         NATURAL JOIN RobberiesPerRobber perRobberView
swen439p1(> );
CREATE VIEW
```

Then, create a view that only contains robbers who participated in more robberies than the average, but spent no time in prison.

```
CREATE VIEW RobbersAboveAvgNotInPrison AS(
   SELECT RobberID,
      NickName,
      robberyTimes,
      NoYears,
      avg.AVGRobberies
   FROM Robbers
      NATURAL JOIN RobberiesPerRobber,
      averageRobberies avg
   WHERE robberyTimes > AVGRobberies
      AND NoYears = 0
);
```

```
swen439p1=> CREATE VIEW RobbersAboveAvgNotInPrison AS(
swen439p1(>     SELECT RobberID,
swen439p1(>         NickName,
swen439p1(>         robberyTimes,
swen439p1(>         NoYears,
swen439p1(>         avg.AVGRobberies
swen439p1(>     FROM Robbers
swen439p1(>         NATURAL JOIN RobberiesPerRobber,
swen439p1(>         averageRobberies avg
swen439p1(>     WHERE robberyTimes > AVGRobberies
swen439p1(>         AND NoYears = 0
swen439p1(> );
CREATE VIEW
```

Since the answer should be sorted in decreasing order of the individual total “earnings” of the robbers. So, finally, do the followings:

```
SELECT r.RobberId,
   r.NickName,
   SUM(a.Share) AS TotalEarning
FROM RobbersAboveAvgNotInPrison r
   NATURAL JOIN Accomplices a
GROUP BY r.RobberId,
   r.Nickname
ORDER BY TotalEarning DESC;
```

We can see the result from below screenshot, 3 robbers are found and they’re sorted in decreasing order of the individual total earnings of the robbers:

```
swen439p1=> SELECT r.RobberId,
swen439p1->     r.NickName,
swen439p1->     SUM(a.Share) AS TotalEarning
swen439p1-> FROM RobbersAboveAvgNotInPrison r
swen439p1->     NATURAL JOIN Accomplices a
swen439p1-> GROUP BY r.RobberId,
swen439p1->     r.Nickname
swen439p1-> ORDER BY TotalEarning DESC;
 robberid |         nickname          | totalearning
----------+---------------------------+--------------
       10 | Bonnie                    |        40085
        8 | Clyde                     |        31800
       24 | Sonny Genovese            |        13664
(3 rows)
```

Single nested SQL query:

```
SELECT RobberId,
   NickName,
   SUM(a.Share) AS TotalEarning
FROM (
      SELECT RobberId,
         COUNT(RobberId) AS robberyTimes
      FROM Accomplices a
      GROUP BY robberid
      ORDER BY Robberid
   ) SB
   NATURAL JOIN (
      SELECT (
            SUM(SB.robberyTimes) / COUNT(robber.*)
         ) AS AVGRobberies
      FROM Robbers robber
         NATURAL JOIN (
            SELECT RobberId,
               COUNT(RobberId) AS robberyTimes
            FROM Accomplices a
            GROUP BY robberid
            ORDER BY Robberid
         ) SB
      WHERE NoYears = 0
   ) avg
   NATURAL JOIN Accomplices a
   NATURAL JOIN Robbers r
WHERE avg.AVGRobberies < SB.robberyTimes
   AND noYears = 0
GROUP BY RobberId,
   Nickname
ORDER BY TotalEarning DESC;
```

```
swen439p1=> SELECT RobberId,
swen439p1->     NickName,
swen439p1->     SUM(a.Share) AS TotalEarning
swen439p1-> FROM (
swen439p1(>         SELECT RobberId,
swen439p1(>             COUNT(RobberId) AS robberyTimes
swen439p1(>         FROM Accomplices a
swen439p1(>         GROUP BY robberid
swen439p1(>         ORDER BY Robberid
swen439p1(>     ) SB
swen439p1->     NATURAL JOIN (
swen439p1(>         SELECT (
swen439p1(>                 SUM(SB.robberyTimes) / COUNT(robber.*)
swen439p1(>             ) AS AVGRobberies
swen439p1(>         FROM Robbers robber
swen439p1(>             NATURAL JOIN (
swen439p1(>                 SELECT RobberId,
swen439p1(>                     COUNT(RobberId) AS robberyTimes
swen439p1(>                 FROM Accomplices a
swen439p1(>                 GROUP BY robberid
swen439p1(>                 ORDER BY Robberid
swen439p1(>             ) SB
swen439p1(>         WHERE NoYears = 0
swen439p1(>     ) avg
swen439p1->     NATURAL JOIN Accomplices a
swen439p1->     NATURAL JOIN Robbers r
swen439p1-> WHERE avg.AVGRobberies < SB.robberyTimes
swen439p1->     AND noYears = 0
swen439p1-> GROUP BY RobberId,
swen439p1->     Nickname
swen439p1-> ORDER BY TotalEarning DESC;
 robberid |         nickname          | totalearning
----------+---------------------------+--------------
       10 | Bonnie                    |        40085
        8 | Clyde                     |        31800
       24 | Sonny Genovese            |        13664
(3 rows)
```

2. *The police department wants to know whether bank branches with lower security levels are more attractive for robbers than those with higher security levels.*

   Construct a view *containing* the *Security* level, the total *Number* of robberies that occurred in bank branches of that security level, and the average *Amount* of money that was stolen during these robberies. [8 marks]

Stepwise approach:

First, construct a VIEW that contains the security level, total number of robberies with the level and the corresponding total money.

```
CREATE VIEW securityWithSumCount AS(
   SELECT bank.Security,
      COUNT(robbery.*) AS RobberiesNumber,
      SUM(robbery.Amount) AS TotalMoney
   FROM Robberies robbery
      INNER JOIN Banks bank USING(BankName, City)
   GROUP BY bank.security -- ORDER BY bank.security
);
```

```
swen439p1=> CREATE VIEW securityWithSumCount AS(
swen439p1(>     SELECT bank.Security,
swen439p1(>         COUNT(robbery.*) AS RobberiesNumber,
swen439p1(>         SUM(robbery.Amount) AS TotalMoney
swen439p1(>     FROM Robberies robbery
swen439p1(>         INNER JOIN Banks bank USING(BankName, City)
swen439p1(>     GROUP BY bank.security
swen439p1(>     -- ORDER BY bank.security
swen439p1(> );
CREATE VIEW
```

Then, construct a VIEW that using the previous view to calculate the average amount of money that was stolen during these robberies, and keep the cols of the security level and the total number of robberies with the level.

```
CREATE VIEW securityWithAVG AS(
    SELECT sumView.Security,
        sumView.RobberiesNumber,
        (sumView.TotalMoney / sumView.RobberiesNumber) AS AverageMoneyAmount
    FROM securityWithSumCount sumView
    -- ORDER BY sumView.Security
);
```

```
swen439p1=> CREATE VIEW securityWithAVG AS(
swen439p1(>     SELECT sumView.Security,
swen439p1(>         sumView.RobberiesNumber,
swen439p1(>         (sumView.TotalMoney / sumView.RobberiesNumber) AS Average
MoneyAmount
swen439p1(>     FROM securityWithSumCount sumView
swen439p1(>     -- ORDER BY sumView.Security
swen439p1(> );
CREATE VIEW
```

Finally, we can see the output:

```
SELECT * FROM securityWithAVG;
```

```
swen439p1=> SELECT *
swen439p1-> FROM securityWithAVG;
    security     | robberiesnumber | averagemoneyamount
-----------------+-----------------+--------------------
 weak            |               4 |             2299.5
 good            |               2 |               3980
 very good       |               3 |     12292.42578125
 excellent       |              12 |   39238.0833333333
(4 rows)
```

Single nested SQL query:

```
SELECT CTE.Security,
    CTE.RobberiesNumber,
    (CTE.TotalMoney / CTE.RobberiesNumber) AS AverageMoneyAmount
FROM (
        SELECT bank.Security,
            COUNT(robbery.*) AS RobberiesNumber,
            SUM(robbery.Amount) AS TotalMoney
        FROM Robberies robbery
            NATURAL JOIN Banks bank
        GROUP By bank.Security -- ORDER BY bank.Security
    ) CTE;
```

Output:

```
swen439p1=> SELECT CTE.Security,
swen439p1->     CTE.RobberiesNumber,
swen439p1->     (CTE.TotalMoney / CTE.RobberiesNumber) AS AverageMoneyAmount
swen439p1-> FROM (
swen439p1(>         SELECT bank.Security,
swen439p1(>             COUNT(robbery.*) AS RobberiesNumber,
swen439p1(>             SUM(robbery.Amount) AS TotalMoney
swen439p1(>         FROM Robberies robbery
swen439p1(>             NATURAL JOIN Banks bank
swen439p1(>         GROUP By bank.Security -- ORDER BY bank.Security
swen439p1(>     ) CTE;
    security     | robberiesnumber | averagemoneyamount
-----------------+-----------------+--------------------
 weak            |               4 |             2299.5
 good            |               2 |               3980
 very good       |               3 |     12292.42578125
 excellent       |              12 |   39238.0833333333
(4 rows)
```

Your answer to Question 6 should include:

- A sequence of SQL statements for the basic queries and the views/tables you created, and the output of the final query.
- A single nested SQL query, with its output from PostgreSQL (hopefully the same).
- Also, submit your SQL nested queries, with each nested query (just SQL code) as a separate sql file. Name files in the following way: Query6_TaskX.sql, where X stands for the task number 1, 2.

Using PostgreSQL on the workstations

We have a command line interface to PostgreSQL server, so you need to run it from a Unix prompt in a shell window. To enable the various applications required, first type either

> need comp302tools

or

> need postgresql

You may wish to add either "`need comp302tools`", or the "`need postgresql`" command to your `.cshrc` file so that it is run automatically. Add this command after the command `need SYSfirst`, which has to be the first `need` command in your `.cshrc` file.

There are several commands you can type at the unix prompt:

> createdb 〈db name〉

Creates an empty database. The database is stored in the same PostgreSQL server used by all the students in the class. You may freely name your database. But to ensure security, you must issue the following command as soon as you log-in into your database for the first time:

```
REVOKE CONNECT ON DATABASE <database_name> FROM PUBLIC;
```

You only need to do this once (unless you get rid of your database to start again). **Note**, your markers may check whether you have issued this command and if they find you didn't, you may be **penalized**.

> **psql** [**–d** 〈db name〉]

Starts an interactive SQL session with PostgreSQL to create, update, and query tables in the database. The db name is optional (unless you have multiple databases)

> dropdb 〈db name〉

Gets rid of a database. (In order to start again, you will need to create a database again)

> pg_dump -i 〈db name〉 > 〈file name〉

Dumps your database into a file in a form consisting of a set of SQL commands that would reconstruct the database if you loaded that file.

> **psql –d** <database_name> **-f** <file_name>

Copies the file <file_name> into your database <database_name>.

Inside and interactive SQL session, you can type SQL commands. You can type the command on multiple lines (note how the prompt changes on a continuation line). End commands with a ';'

There are also many single line PostgreSQL commands starting with '\' . No ';' is required. The most useful are

\? to list the commands,

\i 〈file name〉
loads the commands from a file (eg, a file of your table definitions or the file of data we provide).

\dt to list your tables.

\d〈table name〉 to describe a table.

\q to quit the interpreter

copy <table_name> **to** <file_name>
Copy your table_name data into the file file_name.

copy <table_name> **from** <file_name>
Copy data from the file file_name into your table table_name.

Note also that the PostgreSQL interpreter has some line editing facilities, including up and down arrow to repeat previous commands.
For longer commands, it is safer (and faster) to type your commands in an editor, then paste them into the interpreter!

****