29570220—SQL Programming & Creative Writing

Task 1

Databases are used to store different kinds of data such as names, currency, text, graphics, etc. Determining the appropriate data type for each attribute is important for database performance, storage size, and conducting accurate analysis on databases. As with other programming languages, SQL supports many data types. However, you will mostly use two categories while writing SQL queries: Characters, and numbers.

Consider the following standard SQL data types:

- Characters: `char`, `varchar`.
- Numbers: `integer`, `decimal`, `float`.

Different data types could be used for the same attribute. For example, both `char` and `varchar` could be used to store "names" in a database. Examine the attributes for the Characters and Numbers data types.

- Using specific examples, compare data types for each category. For example, why you should choose `char` rather than `varchar` or opposite.

> *Your response should be at least **250 words** in length, but not more than 750 words. Cite any source in APA format.*

The character string data types of `char` and `varchar` both store text characters. Yet, the `char` data type is designed to contain a fixed number of characters while the `varchar` data type is designed to contain characters of varying lengths. Thus when defining a `char` relation, one needs to specify `char(n)`, where `n` is the number of characters that will be contained by the data type. In contrast a `varchar(m)` declaration specifies that the relation can contain any number of characters in the closed range [0, m]. This features mean that designers should use the `char` data type when it is desired that the relation should contain letters whose length will always remain constant. Thus a relation that requires the letters `Y` or `N` to mean "yes" or "no" can be served adequately by the `char(1)` declaration. On the other hand, the `varchar` data type can serve relations such that of `first_name(25)` well. Here, it means that the database will hold a `first_name` relation that has at most 25 characters. When a database query exceeds that limit during insertion, for instance, then the database should throw an exception (Elmasri & Navathe, 2016).

The numeric data types of `integer`, `decimal`, and `float` serve distinct purposes of handling numbers. The `integer` data type, for instance works with whole numbers that do not contain fractions. The remaining two data types work with real/floating-point numbers. The major distinction between the `decimal` and `float` data types is that while the `decimal` type works well when you want to format numbers into a precision (`i`) and a scale (`j`)—that is, `decimal(i, j` —the `float` type can have as many decimal places as desired. Yet, `float` types present a challenge when one wants to work with value classes such as those that represent monetary values. Because after some calculations one could end with figures that read as $25.35788 when using the `float` data type, it is crucial that database designers make careful choices when settling between `decimal` and `float` data types.

Reference

Elmasri, R., & Navathe, S. (2016). *Fundamentals of Database Systems*. Pearson.

Task 2

Using the relations defined in the Attachment for a hospital software system, normalize your relations into first, second, and third normal form. First, you must describe the changes required to your relations to meet the standard of first 1NF and then 2NF and lastly 3NF. Then, you should provide an Entity Relationship diagram that is in 3NF. All attributes, entities, keys, and relationships should be included and labeled.

> *Remember that in the hospital software system, the `Patient` relation includes `allergies` attribute and the `Appointment` relation includes `medicines` attribute. Allergies describe a list of any known allergies and medicines describe a list of medicines prescribed for a patient.*

Instructions:

- The work must define or describe the changes required to get the relations into the 1st normal form.
- The work must define or describe the changes required to get the relations into the 2nd normal form.
- The work should include an Entity Relationship diagram in 3rd normal form.
- The primary and foreign keys, at a minimum, are defined for each relation in the Entity Relationship diagram.
- The work should include an Entity Relationship diagram that details the relationships between Relations.

> *Cite any source in APA format.*

- Relations of the `DOCTOR` table are:
  - `DOCTOR` (`Doctor_ID`, `Doctor_Name`, `Doctor_Phone`, `Specialty_Number`, `Specialty`)
  - Primary key: `Doctor_ID`
- Relations of the `APPOINTMENT` table
  - `APPOINTMENT`(`Appointment_ID`, `Appointment_Date`, `Doctor_ID`, `Patient_ID`, `BP`, `Pulse`, `Notes`, `Medicines`)
  - Primary key: `Appointment_ID`
  - Foreign keys: `Doctor_ID`, `Patient_ID`
- Relations of the `PATIENT` table:
  - `PATIENT` (`Patient_ID`, `Patient_Name`, `Patient_Phone`, `Email`, `Address`, `Added_Date`, `Allergies`, `Doctor_ID`)
  - Primary key: `Patient_ID`
  - Foreign key: `Doctor_ID`

Conversion to 1NF

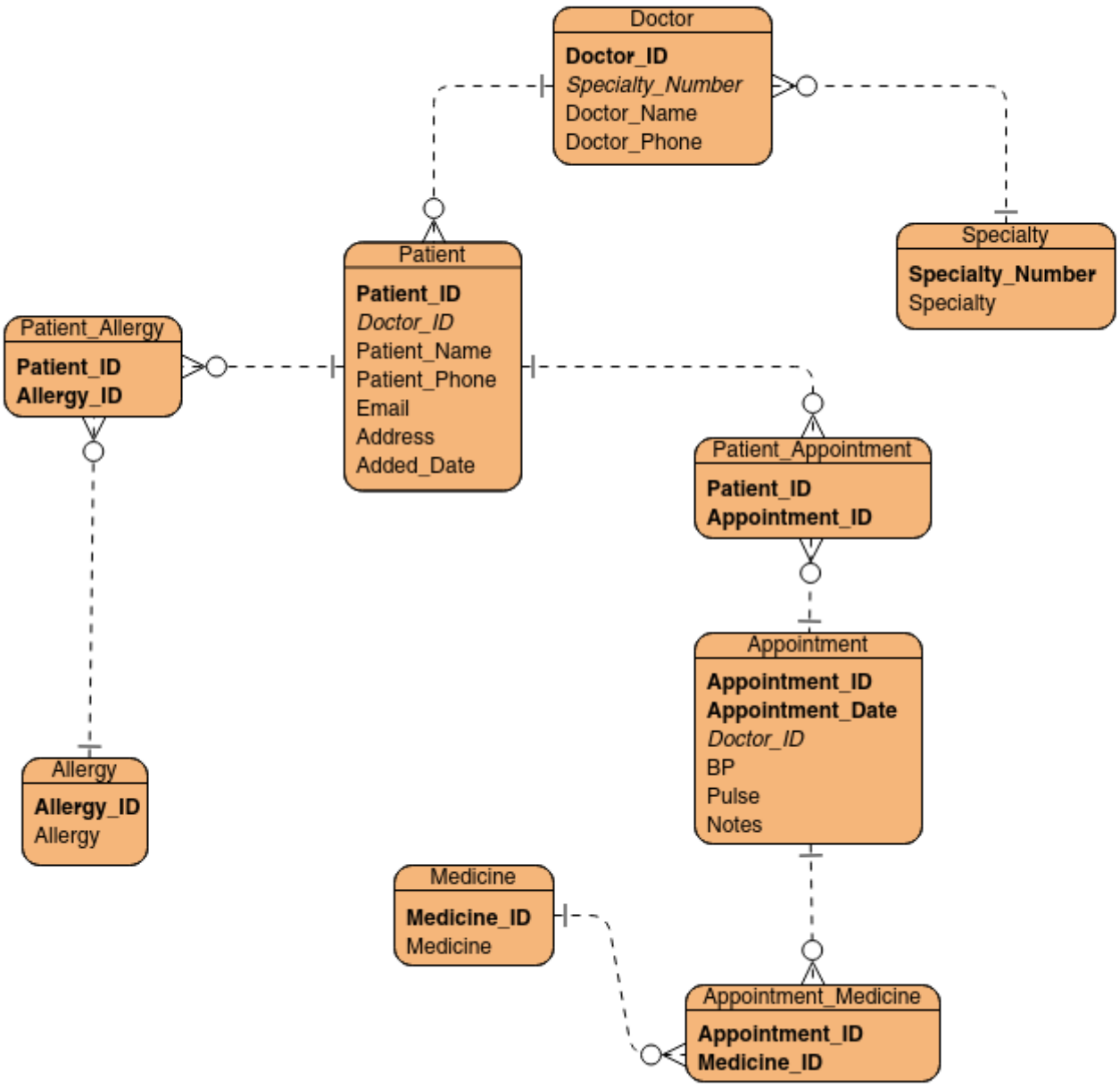
- Eliminating possibly repeating groups:
  - `Specialty_Number` and `Specialty` have the possibility of getting repeated in the `DOCTOR` table. That is, several doctors could have matching `Specialty_Number` and `Specialty`
  - `Medicines` has the possibility of getting repeated in the `APPOINTMENT` table
  - `Allergies` has the possibility of getting repeated in the `PATIENT` table
- Identifying the primary keys:
  - `Specialty_Number` from the `DOCTOR` table can serve as a new primary key. It's dependency is the `Specialty` relation
  - The `Medicines` relation needs a primary key it can be fully dependent on (`Medicines_ID`) because it is not fully dependent on the `Appointment_ID` key
  - The `Allergies` relation needs a new primary of its own (`Allergies_ID`) because it is not fully dependent on the `Patient_ID` key

Conversion to 2NF

- Identify partial dependencies
  - `Appointment_ID` and `Appointment_Date` can serve as the new composite key for the `APPOINTMENT` table.

Conversion to 3NF

- Removing the transitive dependencies



Task 3

Based on Task 2, answer the following using Introduction-Body-Conclusion format:

- Describe what you did (This does not mean that you copy and paste from what you have posted or the assignments you have prepared. You need to describe what you did and how you did it), what you learned, your weekly activities, in what ways are you able to apply the ideas and concepts gained, and finally, describe one important thing that you are thinking about in relation to the activity.
- What are the implementation problems of `Date` and `Time` data types? Discuss solutions for these problems.

> *Your response should be at least **400 words** in length, but not more than 750 words. Cite any source in APA format.*

By analyzing the relations in the three original tables; `DOCTOR`, `APPOINTMENT`, and `PATIENT`; I managed to identify those that would result in repeating sets of data. The `Medicines` relation for instance would cause several rows to contain redundant data. For instance, if in a particular appointment event the doctor prescribes several medicine types—such as, Synthroid, Amoxil, and Levoxyl—then the original, non-normalized `APPOINTMENT` table would contain three rows where every relation detail is repeated except the three `Medicines` relations. Such a waste of table rows clearly shows poor table design. Thus, I normalized the three original tables to remove partial dependencies and to create primary key dependencies which are direct and fully dependent on the resulting primary key alone. That activity resulted in the creation of five new tables—namely, the `SPECIALTY`, `MEDICINE`, `APPOINTMENT_MEDICINE`, `ALLERGY`, and `PATIENT_ALLERGY` tables.

As I worked on the 1NF, 2NF, and 3NF conversions I realized how simple relations can lead to the generation of very many tables. I was tempted to create even more tables with the requisite, full, and direct primary dependencies but then it looked like it would lead to tables that contain very rows. Hence, without being careful, the exercise would have resulted in excessively normalized tables that only contribute to increased complexity of the database design. The concept of using bridging tables played a major role in how I managed to design many-to-many relations without creating unnecessarily excessive foreign key constraints. The `PATIENT_ALLEGRY` table is a good example of such a bridging table. Without it, the design would have been forced to contain a `PATIENT` table with an `Allergy_ID` relation. Also, the `ALLERGY` table would have had a `PATIENT_ID` relation.

The `Date` SQL data type contains ten positions, namely: YYYY-MM-DD. Where YYYY represents the year; MM—month; and, DD—day. Yet, various localizations mean that users expect to find dates written in specific formats. In the US, for instance, users expect to find date written as MM-DD-YYYY. In addition, in the UK, users expect the format: DD-MM-YYYY. Thus, if not careful, database users from these two regions could cause a given database system to throw errors if the user from a given region fails to observe the SQL format of YYYY-MM-DD. Additionally, the conventional `TIME` data type contains eight positions, namely: HH:MM:SS. This format could, however, prove problematic if a user desires to get the time zone info from a `TIME` data type instance. Thus, in such cases, users are encouraged to use a data type that contains time zone data such as the TIME WITH TIME ZONE data type (Elmasri & Navathe, 2016).

Reference

Elmasri, R., & Navathe, S. (2016). *Fundamentals of Database Systems*. Pearson.