**Project Direction Overview:**

I would like to develop a hospital management system mobile based application named **“Health Matters”** which can be used by any size hospital to better manage their patients and their assets. I intend to target this application for government hospitals to benefit the entire public and the hospital itself. This application will contain the data for the patients (their personal information, medical history, vaccination, etc.) as well as the medical staff at the hospital. The application is initially targeted for use by patients as they are the number one priority and then extend to doctors, administrators and so on.

Now I will explain with some examples for how the application is supposed to be used. The application will be having different type of logins depending on if you are a patient, doctor, or an administrator. A patient is new to the hospital and may or may not know about the hospital application. When the patient visits the hospital, he/she will go to the reception for his/her query and receptionist may refer the patient to some doctor and bill him for the appointment accordingly. On the bill will be mentioned the patient ID number and the details provided by the patient to the receptionist. The receptionist will also inform the patient about the mobile app which can be installed by scanning the QR code or the link provided on a poster at the reception. The patient can now login to the application using the patient ID and will be asked to setup the password and profile and to link his/her email address with it. If the patient had already installed the application before coming to the hospital, then they will already be having a patient ID. Now all the data related to the patient will be visible on the application and he can track it.

People who are visiting government hospitals are usually those who are not very affording. This application is built to provide them with ease of access to their data, make payments, appointments, remind them if something is due like a vaccination or an appointment etc. There will be a lot of programming involved with the front-end GUI and everything, but I will only be focusing on the database component of this application.

**Use Cases and Fields:**

One important use case will be the patient interacting with the app for the first time and creating an account. Steps are as follows:

1. The patient visits the hospital website or app store (Google, Apple, etc.) to install the application for the first time.
2. The application asks them if they are a new user and do not have an associated patient ID and if so, then prompts them to create an account.
3. The patient creates an account and enters his/her personal information which is entered into the database at the back end.
4. The patient now has the access to all his/her information.

I am modelling this data base using a hierarchical model, so I have created a **Person** entity which can be a **Doctor** entity or **Patient** entity or both. Some important fields for the **Person** entity are as follows:

|  |  |  |
| --- | --- | --- |
| **Field** | **What it stores** | **Why it’s needed** |
| Person ID (P\_ID) | A unique number to identify the person. | This is the unique identifier of the person; the person details can be accessed using this ID. |
| First Name (P\_FNAME) | This is the first name of the person. | This will be shown in the account and will be used to address the person. |
| Last Name (P\_LNAME) | This is the last name of the person. | This will be shown in the account and will be used to address the person. |
| Mobile Number (P\_MOBNUM) | This is the mobile number of the person. | This will be used to contact the person. |
| Permanent Address (P\_PADDRESS) | This is the permanent address of the person. | This is where the person lives and can be used to send medications etc. |
| Gender (P\_GENDER) | This is the gender of the person. | This is required to identify the gender of the person. |
| Emergency Mobile Number (P\_EMOBNUM) | This is the emergency mobile number. | This will be used to contact the person appointed to contact in case of an emergency with the person. |
| Email address (P\_EADDRESS) | This is the personal email address of the person. | This will be used to send all the reports, bills, etc. to the person for record. |

**Patient** entity and **Doctor** entity will inherit P\_ID from this table. The **Patient** only has the P\_ID field so far.

Another important use case is the patient using the application to make an appointment.

1. The patient logs into the application using his/her patient ID.
2. The patient goes to the appointment tab to make an appointment with a specialized doctor.
3. The tab shows him the list of doctors working at the hospital, their related information such as name, designation, office hours, etc. which is stored in the database at the back end.
4. The patient selects the doctor to make an appointment and is asked to pay for the appointment either through the application or at the reception.
5. The patient makes the payment through application using his/her credit card and the appointment is confirmed and patient provided with an appointment number.
6. The application will remind the patient of the appointment depending on the reminder setting in the application set by the user.

Some important fields for doctor information are as follows:

|  |  |  |
| --- | --- | --- |
| **Field** | **What it stores** | **Why it’s needed** |
| Office Hours (D\_OFFICEHOURS) | This is the timing of the office. | This will be used to display when the doctor will be available for a checkup. |
| Qualification (D\_QUALIFICATION) | This is the qualification of the doctor. | These are the degrees attained by the doctor. |
| Designation (D\_DESIGNATION) | This is the designation of the doctor. | This is the designation of the doctor such as professor, etc. |
| Account Number (D\_ACCOUNT NUMBER) | This is the account number of the doctor. | This is the account to which the salary of the doctor will be transferred. |

I have modelled the relationship of the **Billing** entity with the **Person** entity, so anyone can owe bills to the hospital. The P\_ID will be the foreign key from **Person** entity and a part of the primary key of the **Billing** entity. Some important fields for payment information are as follows:

|  |  |  |
| --- | --- | --- |
| **Field** | **What it stores** | **Why it’s needed** |
| Billing ID (B\_ID) | A unique number to identify each billing. | This is the unique identifier of the bill; the billing details can be accessed using this ID. |
| Payment Type (B\_PAYMENTTYPE) | This is the type of payment cash / card. | This is required for billing purposes. |
| Billing Address (B\_BADDRESS) | This is the billing address. | This will be used for billing purposes etc. |
| Credit/Debit Card Number (B\_CARDNUMBER) | This is the credit/debit card number. | This will be used for billing purposes. |
| Security Code (CVC) (B\_CVC) | This is the CVC code of the credit/debit card. | This will be used for billing purposes. |
| Credit Card Expiry (B\_CARDEXPIRY) | This is the expiration date of the credit/debit card. | This will be used for billing purposes. |
| Credit Card Holder Name (B\_CARDNAME) | This is the name on the credit/debit card. | This will be used for billing purposes. |
| Payment Status (B\_PAYMENT STATUS) | This store status of payment such as paid, due, etc. | This is needed to check if the payment has been cleared or not. |

The Appointment entity will have the **Patient** & **Doctor** ID as foreign keys. Some important fields for appointment information are as follows:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| Appointment Number (A\_ID) | The unique number of the appointment. | To check if the appointment was made and the details. |
| Patient Complain (A\_PATIENT COMPLAIN) | The issue with the patient. | To keep record of the patient’s, complain. |

Another use case could be if the patient wants to look up his/her medical history to check his/her history of treatment at the hospital and for his/her medication:

1. The patient logs into the application using his/her patient ID.
2. The patient goes to the patient history tab in the application.
3. The patient checks his medical history stored in the database.
4. The history will show each appointment ID, the doctor with whom the appointment was made, the patient medical condition, diagnosis, the medication prescribed by the doctor.
5. The patient can then log out once they have the required information.

The **Diagnosis** entity will have appointment ID as a foreign key from the **Appointment** entity. Some important fields for the diagnosis are:

|  |  |  |
| --- | --- | --- |
| Field | What it stores | Why it’s needed |
| Doctor Diagnosis (DI\_DOCDIAG) | Diagnosis of the patient complain. | To keep record of the diagnosis. |
| Doctor Medication (DI\_DOCMED) | Medication given by the doctor. | To keep record of the medication. |
| Doctor Referral (DI\_DOCREF) | If doctor refers to some other doctor. | To keep record of doctor referral. |
| Doctor next appointment date (DI\_NEXTAPP) | Next appointment date with the doctor if needed. | To keep record of another appointment date with the doctor if needed. |
| Patient admission type (IN/OUT) (DI\_ADMTYPE) | Doctor decision if patient should be admitted. | To keep record of the admission type. |

**Health Matters Structural Rules:**

**First use case:**

Patient interacting with the app for the first time and creating an account:

1. The patient visits the hospital website or app store (Google, Apple, etc.) to install the application for the first time.
2. The application asks them if they are a new user and do not have an associated patient ID and if so, then prompts them to create an account.
3. The patient creates an account and enters his/her personal information which is entered into the database at the back end.
4. The patient now has the access to all his/her information.

As I am only focused on the database part so only the information that needs to be stored in the database is of concern to me. From step 2 of this use case, I can see one **entity Patient**. In looking at the other steps I cannot see any entity or relationship. As I only have one entity for now and no relationships therefore, I cannot infer a business rule from just this first use case. So, now I will move on to the next use case.

**Second use case:**

Patient using the application to make an appointment:

1. The patient logs into the application using his/her patient ID.
2. The patient goes to the appointment tab to make an appointment with a specialized doctor.
3. The tab shows him the list of doctors working at the hospital, their related information such as name, designation, office hours, etc. which is stored in the database at the back end.
4. The patient selects the doctor to make an appointment and is asked to pay for the appointment either through the application or at the reception.
5. The patient makes the payment through application using his/her credit card and the appointment is confirmed and patient provided with an appointment number.
6. The application will remind the patient of the appointment depending on the reminder setting in the application set by the user.

From step 2 of this use case, I can see that the **Patient entity** is related to a **Doctor entity** through an **Appointment entity** which is an associative entity. From step 4, I can see that the patient must pay his bills at the hospital so, we have the **Billing entity** related to the **Patient entity**. Now that I have identified four entities and their relationships from this second use case, I will now number and create some business rules.

1. Each patient may have many appointments; each appointment has only one patient.

This business rule is inferred from step 2 of this use case. A patient can book multiple appointments at the hospital with different or even the same doctor. A new patient will have no appointments. But an appointment will always have a must one patient.

1. Each doctor may have many appointments; each appointment has only one doctor.

This business rule is also inferred from step 2 of this use case. A doctor can have many appointments at a time. But if the doctor is new to the hospital, then he may also have no appointments at all. However, each appointment will always have a must one doctor.

1. Each patient must have a billing; each billing must have a patient.

This business rule is inferred from step 4 of this use case. Each patient must pay his / her bills at the hospital. The bills could be for labs, surgeries, appointments, etc. So, a patient must have a billing information for use by the hospital. Likewise, each billing information must have a patient related to it. A doctor at the hospital could also be a patient and he could also have a billing.

**Third use case:**

Patient wants to look up his/her medical history to check his/her history of treatment at the hospital and for his/her medication.

1. The patient logs into the application using his/her patient ID.
2. The patient goes to the patient history tab in the application.
3. The patient checks his medical history stored in the database.
4. The history will show each appointment ID, the doctor with whom the appointment was made, the patient medical condition, diagnosis, the medication prescribed by the doctor.
5. The patient can then log out once they have the required information.

From step 3 of this use case, we can see that an **Appointment entity** is related to a **Diagnosis** entity. There are no other entities or relationships that can be inferred from this use case. The business rule considering this new entity is as follows:

1. An appointment will have only one diagnosis; a diagnosis will have only one appointment.

An appointment between a doctor and a patient will always have a diagnosis provided by the doctor. Likewise, a diagnosis will only happen if an appointment was made with a doctor.

Therefore, now I have 4 structural rules for my database which are as follows:

1. Each patient may have many appointments; each appointment has only one patient.
2. Each doctor may have many appointments; each appointment has only one doctor.
3. Each patient must have a billing; each billing must have a patient.
4. An appointment will have only one diagnosis; a diagnosis will have only one appointment.

**Initial Health Matters EERD**

Diagram

Description automatically generated

The EERD above, has been implemented above using the four business rules initially determined from the use cases. Now that the EERD has been implemented, new relationships and business rules have emerged after its analysis, so the business rules need to be revised.

The third business rule involves a **patient** entity and a **billing** entity which I have implemented in the above EERD between a **person** entity and a **billing** entity. Although the business rule still does hold true, but it is ambiguous as there is no direct relation shown in the EERD between the **patient** and **billing** entity. Therefore, the third business rule after edition is now as follows:

1. A person may have none or many billing; a billing must have a person.

I have implemented a hierarchical relationship in the database of the hospital management system. All the staff (such as doctors, nurses, etc.) and the patients at the hospitals will have some common attributes. These are implemented in the entity of **Person**. A person could be a doctor or a patient or both at the hospital which is implemented by the overlapping complete constraint in the EERD. A person can have none or many **Billing** entity based on the fact if he / she has any bills related to the hospital. A patient makes an appointment with a doctor and the doctor provides a diagnosis to the patient related to his / her medical condition. A doctor might also refer the patient to one or more doctors which is implemented using the **REFERRAL** associative (link) entity. A doctor can have many qualifications which is implemented by the **HAS** associative (link) entity between the **QUALIFICATION** and **DOCTOR** entities. The **PERSON** and the **BILLING** entities must have an address which is implemented in the **ADDRESS** entity.

**Revised Database Rules after Implementation of Specialization Generalization Relationships**

The new business rules are as follows:

1. A person can be a patient or a doctor or both at the hospital.
2. Each patient may have many appointments; each appointment has only one patient.
3. Each doctor may have many appointments; each appointment has only one doctor.
4. A person may have none or many billing; a billing must have a person.
5. An appointment will have only one diagnosis; a diagnosis will have only one appointment.
6. A doctor has many qualifications; a qualification can have none or many doctors.
7. A diagnosis can have none or many referrals; a referral can have only one diagnosis.
8. A doctor can have many referrals; a referral can only have one doctor.
9. A person must have an address; an address has none or many persons.
10. A billing must have an address; an address has none or many billings.

**Initial Physical ERD before Normalization:**

Diagram, schematic

Description automatically generated

The physical ERD has been implemented above using the EERD and the keys and attribute types have been added to it. There are two entities below **PERSON** which are **DOCTOR** and **PATIENT** entities.

**Adding Attributes to DBMS Physical ERD:**

In this part, I will be adding attributes to my DBMS physical ERD and explaining the reasoning behind the inclusion of each attribute. The attributes are as follows:

|  |  |  |  |
| --- | --- | --- | --- |
| **Table Name** | **Attribute** | **Data Type** | **Reasoning** |
| PERSON | P\_ID | DECIMAL(12) | Every person in this database will have a unique person id assigned to them which is the primary key of this table. The datatype is set to DECIMAL (12) to allow for a large number of data rows. |
| ADDRESS | ADD\_ID | DECIMAL(12) | Every address will have a unique address id assigned to it which is the primary key of this table. The datatype is set to DECIMAL (12) to allow for a large number of data rows. |
| PERSON | P\_LNAME | VARCHAR(30) | This is the last name of the person in the hospital database application. VARCHAR (30) is chosen to be the data type as names might be long. |
| PERSON | P\_FNAME | VARCHAR(30) | This is the first name of the person in the hospital database application. VARCHAR(30) is chosen to be the data type as names might be long. |
| PERSON\_HAS\_TYPE | HAS\_ID | DECIMAL(12) | This attribute is the primary key of this table. DECIMAL(12) is chosen to be the data type. |
| PERSON | P\_MOBNUM | DECIMAL(10) | This variable is used to store the personal mobile number of the person in the database. DECIMAL(10) is used because the US mobile numbers are 10 digits. |
| PERSON | P\_GENDER | VARCHAR(30) | This is used to store the gender of the person in the database which is a personal information specific to a person. VARCHAR(30) is chosen to be the data type. |
| PERSON | P\_EMOBNUMBER | DECIMAL(10) | This variable stores the emergency mobile number of a person in the database. This will be used to contact in case of an emergency with the person. DECIMAL(10) is used because the US mobile numbers are 10 digits. |
| PERSON | P\_EADDRESS | VARCHAR(60) | This variable stores the email address of the person in the database which will be used to contact the person for promotions, billings, etc. VARCHAR(60) is used to allow for large email addresses. |
| ADDRESS | ADD\_ZIPCODE | DECIMAL(5) | This variable is used to store the zip code of the address. DECIMAL(5) is used because US zip codes have a length of 5. |
| ADDRESS | ADD\_STREET | VARCHAR(255) | This variable is used to store the street information of the address. VARCHAR(255) is used as the datatype as street information can vary in length. |
| ADDRESS | ADD\_STATE | VARCHAR(255) | This variable is used to store the state information of the address. VARCHAR(255) is used as the datatype as state information can vary in length. |
| ADDRESS | ADD\_CITY | VARCHAR(255) | This variable is used to store the city information of the address. VARCHAR(255) is used as the datatype as city information can vary in length. |
| BILLING | B\_ID | DECIMAL(12) | Every billing in this database will have a unique billing id assigned to it which is the primary key of this table. The datatype is set to DECIMAL(12) to allow for a large number of data rows. |
| BILLING | B\_PAYMENTTYPE | VARCHAR(20) | This attribute is used to store the payment type of the billing such as credit card or cash. VARCHAR(20) is used as the datatype. |
| BILLING | B\_PAYMENTSTATUS | VARCHAR(30) | This attribute is used to store the status of the payment to the hospital such as cleared, pending, etc. VARCHAR(30) is used as the datatype to store the status. |
| BILLING | B\_AMOUNT | DECIMAL(15,2) | This attribute is used to store the amount of each billing. DECIMAL(15,2) is used as the data type to allow for bills with large amounts. |
| BILLING | B\_PAYMENTDATE | DATE | This attribute is used to store date each billing is cleared. DATE is used as the data type to store this. |
| PERSON\_TYPE | TYPE\_ID | DECIMAL(12) | This attribute is the primary key of this table and is used to store unique id of each type of person. DECIMAL(12) is used as the datatype to allow for large number of rows of data. |
| PERSON\_TYPE | TYPE\_DESCRIPTION | VARCHAR(30) | This attribute is used to store the description of each type of person such as doctor, patient, etc. VARCHAR(30) is used as the datatype to store this. |
| CREDIT\_DEBIT\_CARD | CARD\_ID | DECIMAL(12) | This attribute is the primary key of this table. This is a unique identifier of each credit card in the system. DECIMAL(12) is used to allow for large number of rows in the data. |
| CREDIT\_DEBIT\_CARD | CARD\_NUMBER | DECIMAL(16) | This attribute is used to store the credit card number. DECIMAL(16) is used because mostly credit card numbers are 16 digits. |
| CREDIT\_DEBIT\_CARD | CARD\_CVC | DECIMAL(3) | This attribute is used to store the CVC code of the credit card. DECIMAL(3) is used because the CVC codes are usually 3 digits. |
| CREDIT\_DEBIT\_CARD | CARD\_EXPIRY | DATE | This attribute is used to store the date of expiry of the credit card. DATE is used as the datatype as it is a date of expiry. |
| CREDIT\_DEBIT\_CARD | CARD\_NAME | VARCHAR(60) | This attribute is used to store the name on the credit card. VARCHAR(60) is chosen because the names might be long. |
| APPOINTMENT | A\_ID | DECIMAL(12) | This attribute is the primary key of this table and is used to store a unique appointment number for each appointment. DECIMAL(12) is used as the datatype to allow for large number of rows of data. |
| APPOINTMENT | A\_PATIENTCOMPLAIN | VARCHAR(255) | This attribute is used to store the complain of the patient. VARCHAR(255) is chosen as the datatype for this attribute. |
| APPOINTMENT | A\_DATETIME | DATE | This attribute is used to store the date and time of each appointment. DATETIME is used as the data type to store this. |
| DOCTOR | D\_OFFICEHOURS | VARCHAR(30) | This attribute is used to store the office hours of each doctor. VARCHAR(30) is used as the data type to allow to save the hours. |
| DOCTOR | D\_DESIGNATION | VARCHAR(30) | This attribute is used to store the designation of the doctor at the hospital such as professor, assistant professor, etc. VARCHAR(30) is the datatype used to store this. |
| DOCTOR | D\_ACCOUNTNUMBER | DECIMAL(30) | This attribute is used to store the account number of the doctor in which his / her monthly salary will be deposited. DECIMAL(30) is used as the data type as different banks have different length of account numbers. |
| DOC\_HAS\_QUAL | HAS\_ID | DECIMAL(12) | This attribute is the primary key of this table showing the associative relationship between doctor and qualifications table. DECIMAL(12) is used to allow for large number of rows. |
| QUALIFICATIONS | Q\_ID | DECIMAL(12) | This attribute is used to store the unique id of each type of qualification. This is the primary key. DECIMAL(12) is used as the datatype to allow for a large number of rows. |
| QUALIFICATIONS | Q\_NAME | VARCHAR(60) | This attribute is used to store the name of each qualification. VARCHAR(60) is used as the datatype to store this. |
| QUALIFICATIONS | Q\_QUALIFICATION | VARCHAR(60) | This attribute is used to store the abbreviation of the qualification. VARCHAR(60) is used as the datatype to store this. |
| DIAGNOSIS | DI\_ID | DECIMAL(12) | This attribute is the unique identifier of each diagnosis. This is the primary key of this table. DECIMAL(12) is used as the datatype to allow for a large number of rows. |
| DIAGNOSIS | DI\_DOCDIAG | VARCHAR(1024) | This attribute is used to store the diagnosis provided by the doctor for the medical condition of the patient. VARCHAR(1024) is used to store this in case of long diagnosis. |
| DIAGNOSIS | DI\_DOCMED | VARCHAR(1024) | This attribute is used to store the medication prescribed by the doctor. VARCHAR(1024) is used to store this in case of long medications. |
| DIAGNOSIS | DI\_NEXTAPP | DATE | This attribute is used to store the next appointment date with the doctor again if needed. DATE datatype is used to store this. |
| DIAGNOSIS | DI\_ADMTYPE | VARCHAR(60) | This attribute is used to store if the patient needs to be admitted or not. VARCHAR(60) is used as the datatype to store this. |
| REFERRAL | REF\_ID | DECIMAL(12) | This attribute is the unique identifier of each referral. This is the primary key of this table. DECIMAL(12) is used as the datatype to allow for a large number of rows. |

**Normalizing the DBMS Physical ERD:**

Looking at the current DBMS physical ERD I think that only one table needs normalization and that is the **BILLING** table. The billing table keeps record of the credit card currently and for the same person or family of that person, the credit card information will keep on repeating in the billing table which will result in too much redundancy. Therefore, I will create another table to store the credit card information. This new table will be called **CREDIT\_DEBIT\_CARD** and will have the **CARD\_ID** as the primary key. Creating this new table will also mask the credit card information as the credit card number is not used as the primary key and the credit card information is in a separate table. Also, the **P\_TYPE** (person type can be a doctor, patient, or both) attribute in the person table is a multivalued attribute and the type will keep repeating for different persons. Therefore, I have also created a separate **PERSON\_TYPE** table to store the different person types in the database. The physical ERD after this normalization is as follows (on next page):

Diagram, schematic

Description automatically generated

The address table in the DBMS physical ERD could also be further normalized by creating separate tables for states, cities, etc. but that would create too many unnecessary tables and would create a mess in the database. To get the full address then many joins will have to be used. Therefore, I have chosen to keep the address table separate and have kept the information as it is. Now that I have normalized the database and created new tables, I will revise the database rules. The new business rules are as follows:

1. A person can be a patient or a doctor or both at the hospital.
2. Each patient may have many appointments; each appointment has only one patient.
3. Each doctor may have many appointments; each appointment has only one doctor.
4. A person may have none or many billing; a billing must have a person.
5. An appointment will have only one diagnosis; a diagnosis will have only one appointment.
6. A doctor has many qualifications; a qualification can have none or many doctors.
7. A diagnosis can have none or many referrals; a referral can have only one diagnosis.
8. A doctor can have many referrals; a referral can only have one doctor.
9. A person must have an address; an address has none or many persons.
10. A billing must have an address; an address has none or many billings.
11. A billing may have credit card associated with it; a credit card may have one or many billing.
12. A person has one or many person\_has\_type; a person\_has\_type has only one person.
13. A person\_type has none or many person\_has\_type; a person\_has\_type has only one person\_type.

**Tables & Constraints in SQL:**

I have created the tables and constraints shown in the full DBMS physical ERD in Microsoft SQL Server. Below is also attached a screenshot of a part of the script:

**Text

Description automatically generated**

I have also inserted the data in all the tables using the data generated from mockaroo.

**Index Placement & Creation:**

In this section, I will be placing an index on columns that will mostly be used in SQL queries to increase the processing speed of the queries. The primary keys are already indexed by default in the DBMS. The primary keys include:

1. ADD\_ID from ADDRESS table.
2. P\_ID from PERSON table.
3. B\_ID from BILLING table.
4. CARD\_ID from CREDIT\_DEBIT\_CARD table.
5. HAS\_ID from PERSON\_HAS\_TYPE table.
6. TYPE\_ID from PERSON\_TYPE table.
7. P\_ID from PATIENT table.
8. A\_ID from APPOINTMENT table.
9. P\_ID from DOCTOR table.
10. HAS\_ID from DOC\_HAS\_QUAL table.
11. Q\_ID from QUALIFICATIONS table,
12. DI\_ID & A\_ID from DIAGNOSIS table.
13. REF\_ID from REFERRAL table.

Next, we are going to index all the foreign keys in all the tables because these are usually used in join statement to get data from different tables. Below is the table for the foreign keys including the reason if the foreign key should be unique or not:

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Unique?** | **Description** |
| PERSON.ADD\_ID | Not Unique | There could be multiple people living at the same address who visit the hospital so this index will not be unique. |
| BILLING.P\_ID | Not Unique | A person could visit the hospital at different times and have multiple billings assigned to him. |
| BILLING.ADD\_ID | Not Unique | A billing associated with the same person or person of same family could have the same billing address. |
| BILLING.CARD\_ID | Not Unique | A person visiting multiple times will have the same card id through which he will make the payment. |
| PERSON\_HAS\_TYPE.TYPE\_ID | Not Unique | There could be multiple people having the same type such as doctor or patient. |
| PERSON\_HAS\_TYPE.P\_ID | Not Unique | A person could have multiple person types as he could be a doctor and a patient both. |
| APPOINTMENT.PAT\_ID | Not Unique | A patient could make multiple appointments at the hospital. |
| APPOINTMENT.DOC\_ID | Not Unique | A doctor could have multiple appointments at the hospital. |
| DOC\_HAS\_QUAL.Q\_ID | Not Unique | Multiple doctors at the hospital could have the same qualification. |
| DOC\_HAS\_QUAL.P\_ID | Not Unique | A doctor at the hospital could have multiple qualifications. |
| REFERRAL.P\_ID | Not Unique | A doctor could be referred multiple times in different diagnosis so P\_IS will repeat. |
| REFERRAL.DI\_ID | Not Unique | A diagnosis could have referral to multiple doctors so DI\_ID will repeat. |

Apart from the primary & foreign keys, I will assign index to columns that I think will mostly be used to query the database. These columns are listed below:

|  |  |  |
| --- | --- | --- |
| **Column Name** | **Unique?** | **Description** |
| BILLING.B\_PAYMENTSTATUS | Not Unique | The hospital will want to know how many billings have pending payments so this will be mostly used in queries. |
| APPOINTMENT.A\_DATETIME | Not Unique | The hospital will want to know the number of appointments in a particular time interval. Appointments could have the same time. |
| DIAGNOSIS.DI\_ADMTYPE | Not Unique | The hospital will want to know how many inpatients or outpatients they have at a particular date and time. |

Apart from this we can index columns that will be mostly used in daily SQL Queries once the application is up and running.

**Procedures in the DMBS:**

To perform some of the repeated transactions in the DBMS, instead of writing queries for each transaction I have created some procedures in the DBMS. The following are some of the procedures created in the DMBS:

1. **ADD\_PERSON**

This procedure is being used to add a new person to the database. When a person creates an account, this procedure is used at the back end to insert values into the appropriate DBMS table. Please find below the screenshot for this procedure.

Graphical user interface, text, application, email

Description automatically generated

1. **ADD\_BILLING:**

This procedure is being used to enter a new billing in the database related to a person. If a person visits the hospital as a patient, then he owes some bill to the hospital for all the appointments. Therefore, every time a person comes to the hospital for a visit, this procedure is used at the back end of the application to update the billing table in the DBMS. Please find below the screenshot for this procedure.

Graphical user interface, text, application, email

Description automatically generated

1. **PENDING\_AMOUNT:**

This procedure is used to calculate the pending amount in bills of a person in the database. In the application, the administrator can input the ID of the person for whom he wants to check the pending amount and this procedure will run at the back end and print the appropriate amount owed by that person. Please find below the screenshot for this procedure.

Graphical user interface, text, application, email

Description automatically generated

1. **COUNT\_APPOINTMENT\_PER\_DOCTOR:**

This procedure is used to count the appointments for the doctor whose ID has been entered till date. At the front end in the application, the administrator can enter the ID of the doctor and if it exists, then the number of appointments that doctor has had will be printed. Please find below the screenshot for this procedure.

Graphical user interface, text, application, email

Description automatically generated

**Triggers in the DBMS:**

To maintain history tables of the updates and to implement some checks that are natural human logic I have implemented some triggers on tables in the DBMS. The following are the triggers and their descriptions:

1. **BILLING\_HISTORY\_1:**

This trigger is used to keep a history of all the updates that are made to billings in the billing table. To do this, a new table called “BILLING\_HISTORY” has been created to keep track of all the updates. We will be adding this table to the ERD and creating a new business rule which is: **“A billing may have many billing histories; a billing history will have a must one billing”**. Whenever any update is made in the billing table, this trigger will run and save the old and new values in the “BILLING\_HISTORY” table. A new value of **“-1”** denotes that the value remained unchanged in the update. Please find below the screenshot of the trigger.

Text

Description automatically generated

1. **BILLING\_CHECK:**

This trigger has been implemented to check if natural human logic is followed in the insertion of a record into the billing table. This trigger runs whenever there is an update or insertion into the billing table. First check it performs is that the payment method is valid or not. There are only 2 payment methods in the database which are: “CREDIT/DEBIT CARD” & “CASH”. Therefore, the payment method must be one of these. The second check is that if the payment method of “CARD” is selected then “CARDID” field cannot be empty, and it must have a CARDID. Also, if the payment method is “CASH” then this field must be empty. Third check is that if the payment status is “PAID”, then the payment date must be there as well, and it cannot be null. Also, if the payment status is “PENDING” which means the payment is yet to be made, then the date field should be NULL. Please find below the screenshot of the trigger.

Graphical user interface, text, application, email

Description automatically generated

**Updated Business Rules & Physical EERD:**

The updated business rules are as follows:

1. A person can be a patient or a doctor or both at the hospital.
2. Each patient may have many appointments; each appointment has only one patient.
3. Each doctor may have many appointments; each appointment has only one doctor.
4. A person may have none or many billing; a billing must have a person.
5. An appointment will have only one diagnosis; a diagnosis will have only one appointment.
6. A doctor has many qualifications; a qualification can have none or many doctors.
7. A diagnosis can have none or many referrals; a referral can have only one diagnosis.
8. A doctor can have many referrals; a referral can only have one doctor.
9. A person must have an address; an address has none or many persons.
10. A billing must have an address; an address has none or many billings.
11. A billing may have credit card associated with it; a credit card may have one or many billing.
12. A person has one or many person\_has\_type; a person\_has\_type has only one person.
13. A person\_type has none or many person\_has\_type; a person\_has\_type has only one person\_type.
14. A billing may have many billing histories; a billing history will have a must one billing.

Diagram, schematic

Description automatically generated

**Questions & Queries:**

I have also created some procedures with queries that the hospital administration might be working with most often and included those in the Procedures section of this document. The following are some other queries as well that the hospital administration might be working with:

1. **Query for net income of hospital in last 6 months:**

**Text, application

Description automatically generated**

1. **Query for number of appointments in last 6 months:**

**Graphical user interface, text, application, email

Description automatically generated**

**Summary and Reflection:**

My database is for a hospital management system with priority to the patients for whom this application is being created. This application will also go on to include the hospital staff afterwards. What happens normally is that patients go to the hospitals, and they have a lot of paperwork to deal with such as medical history, medicines, bills, lab reports, etc. The purpose of the application is to keep a track of everything and provide billing and other services as well. The information is also visible to the patient in the application so even if they lose the paperwork, it is all there in the database of the application and accessible to them. Also, all the paperwork will be having the digital signatures of the doctors, lab staff etc. so that the information can be validated easily when used for other purposes.

The structural database rules and the EERD for my database design contain the important entities of Person, Billing, Doctor, Patient, Appointment, Diagnosis, Referral, Address, Qualifications, Has, as well as relationships between them.

I had started from three very simple use cases but ended up making 4 business rules. But then after the implementation of the EERD, more business rules and entities emerged. The total business rules increased to 10. The activity has shown me that even a very small and simple use case could require many entities to implement it and could involve complex relationships.

Using the EERD I implemented the physical ERD which included the keys and attribute types. Now it is looking like the project is ready to be implemented in SQL.

After physical ERD has been implemented, I have added relevant attributes to each of the tables in the physical ERD which are typical of a real-world scenario. After adding the attributes, I implemented normalization on the physical ERD which resulted in new tables being created. Once the new tables were implemented in the physical ERD, the business rules also had to be updated related to these tables. After the normalization, I wrote SQL scripts to implement the physical ERD in SQL and created tables with the same attributes and constraints. Once the tables were created, then I checked which columns needed indexing and indexed them.

Now that the database has been created and all the data input, I have added procedures and to the DBMS for queries that might be quite often run by the hospital administration to eliminate the time to write the queries again and again. In addition, I have also implemented a triggers on the “BILLING” table to maintain history of updates to the tables and to perform some checks on the values inserted and updated in the table. I have also written some quite simple queries that the hospital administration might be using to get data about the number of appointments or net income.

Now that I have done all of this, I can see that I have accomplished a lot with the database over the project iterations. This is still quite a simple model and still needs more checks and balances, but this is still quite a solid foundation to pave the path for its implementation on a larger scale.

It feels great to see with what I have accomplishes with this database project in such a short amount of time. Would love feedback on the project to make it better. A big thanks to my professor from whom I learnt a lot about SQL throughout the semester.