Emergency Department Bed Management

Philip Bontumasi

800906399

December 10, 2021

Introduction

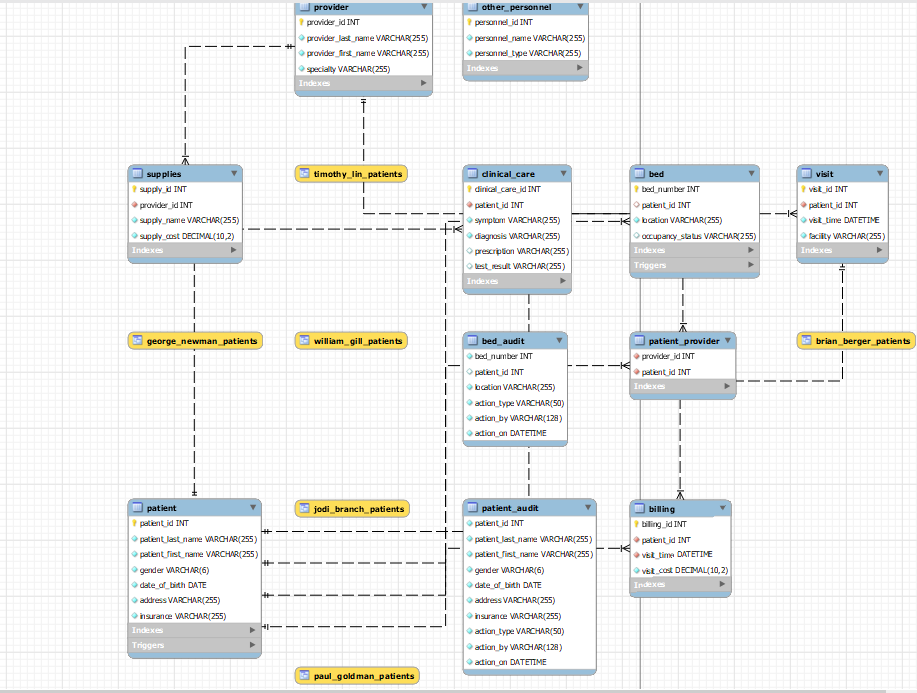
Emergency medical records are used to support patient demographic information and other records that are important to the unit. When examining an emergency department, it is important to not only have patient information, but also information about the availability of beds. This project will allow for patient information to be entered and updated as necessary, as well as updates to available beds within a particular emergency department location. The database created will support all basic functionality to store the information for easy access to the stakeholders. Having these records in a database will help to reduce human error regarding keeping patient records and ease the burden of having to manually keep track of those records.

The design of the database is based around the individual patient information. The patient\_id attribute is used throughout all tables in which the patient is associated with. Each ID is unique as patients may share the same name or the same provider. A patient may have multiple symptoms and may have multiple visits as well. The times listed in the visit table can apply to multiple patients in different locations.

Changes to the database from project one will provide easier access to the tables and make finding what is needed much work quicker. Some changes include indexing the patient names and visits, creating views for each of the providers, and adding stored procedures and triggers to deal with database edits. A new table has been added for other personnel, and the permissions granted in the database vary based on the type of personnel.

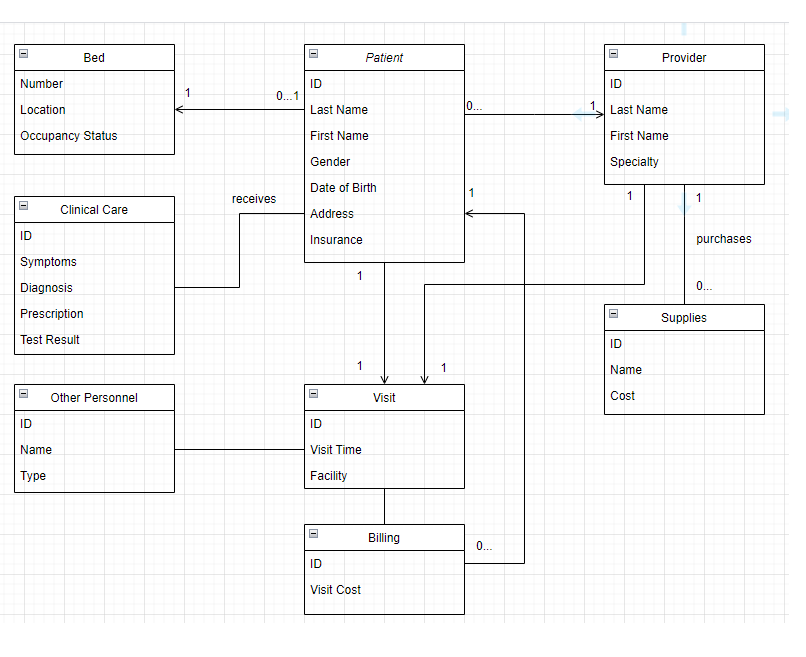
Functional Requirements  
 There are eleven different tables within the database. The bed table displays all information relating to available beds in a specified location. If the bed is available, then the patient\_id column will be a null value. The billing table displays all costs associated with a patient’s specific visit. The clinical\_care table displays results from testing as well as the diagnosis and the symptoms that the patient presented. The patient table displays demographic information about the patient such as their name and age. The provider table shows the name and specialty of the provider, and the patients that are associated with them. The patient\_provider table includes all associations between a patient and a provider that they have visited. The supplies table involves cost of supplies that the provider will need to purchase. The visit table contains the information about each individual visit that a single patient has made. The other\_personnel table providers information about the type of personnel a person is as well as their name. Two new tables to the database include audit trails for the patient and bed tables. These show any changes made to these tables, whether it be an insertion, update, or deletion.

ER Diagram

The ER diagram shows the relations between each of the eight tables (classes). Patients can be associated with providers, clinical care, and beds. Each visit will incur a visit cost. Providers are responsible for the purchase of supplies and can be associated with one or many patients. This table reflects the addition of the other\_personnel table and includes the triggers of each table as well as the views associated with the schema.

UML Diagram

Zero-to-one patients can occupy a single bed. A patient has an association with the clinical care class. A patient has a single provider, and providers many have an indefinite number of patients. A single provider may need to purchase zero to many supplies. Patients and providers are both associated with a particular visit. Each visit results in a bill to the patient, and patients may receive zero to many bills. Other personnel can be associated with one-to-many visits if they are a receptionist, and managers can manage one-to-many employees.



Proof of BCNF

Since patient and provider names are given unique column names specific to that table, and last names and first names are separated, this avoids any non-atomic values within the tables. The tables satisfy first normal form due to only containing atomic values, and each record in the tables being a unique record, typically identified by an ID.

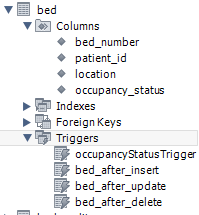
To satisfy second normal form, there must be no partial dependencies within the billing and visit tables. Since there are no tables with multiple primary keys, there will not be any partial dependencies in the tables.

Eliminating transitive dependencies is required to satisfy third normal form. This means that all non-prime keys must not depend on another non-prime key. All the patient, provider, and supply information are dependent on only the primary key of ID, as it is the only variable which must be unique. For instance, there could be multiple people named John Smith, or multiple purchases of the same supplies.

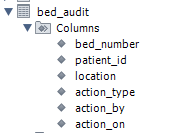
Boyce-Codd Normal Form ensures that no primary keys can be dependent on a non-primary key. Within the patient table, none of the variables can uniquely identify the primary key of ID. It is not possible to get the ID using any of the non-primary keys, as none of the variables must be unique. There could be multiple people with the same gender, date of birth, name, and insurance. For the address, there could be multiple people living at the same address as well. Therefore, this table and the others satisfy Boyce-Codd Normal Form. Using the same format that was used for the previous tables, the new other personnel table also uses BCNF by providing a unique ID for each of the personnel and avoids any dependencies.

Table Information

Bed table with 4 columns, 1 foreign key, and 4 triggers



Bed\_audit table with 6 columns



Billing table with 4 columns, 2 foreign keys

Graphical user interface, text, application

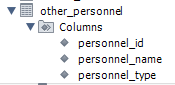
Description automatically generated

Clinical care table with 6 columns, 1 foreign key

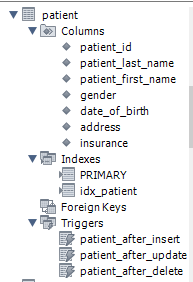
Graphical user interface, text, application

Description automatically generated

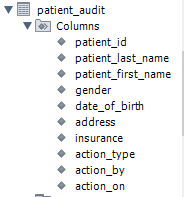
Other\_personnel table with 3 columns



Patient table with 7 columns, 1 custom index, and 3 triggers



Patient\_audit table with 10 columns



Patient\_provider table with 2 columns, 2 foreign keys

Graphical user interface, text, application

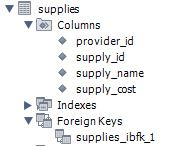
Description automatically generated

Provider table with 4 columns

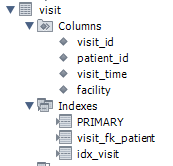
Text

Description automatically generated

Supplies table with 4 columns, 1 foreign key

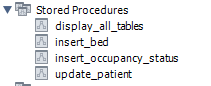


Visit table with 4 columns, 1 foreign key, and 1 custom index

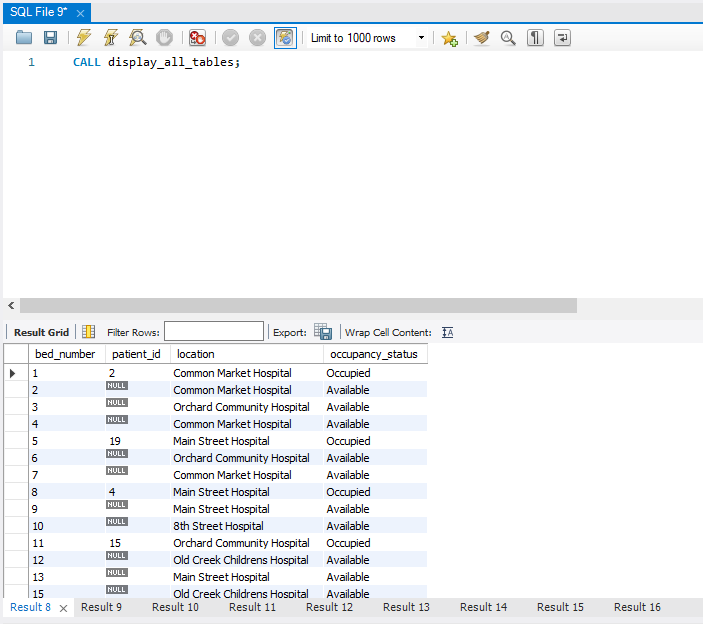


Application Programming Interface Implementation

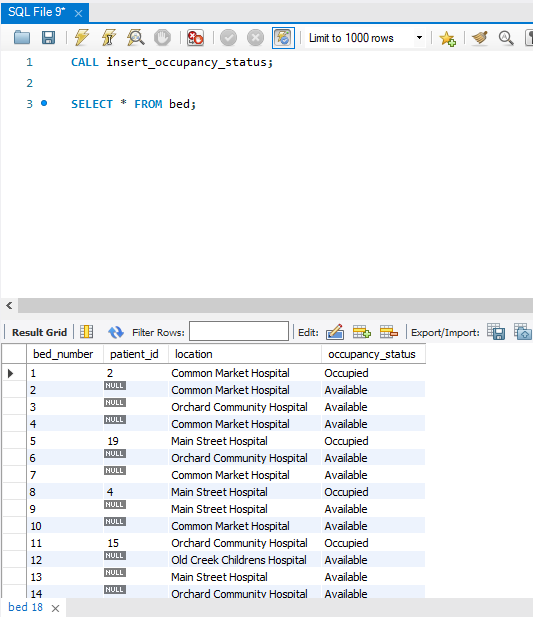
* Stored Procedures
  + Included are four different stored procedures. These are used to make it so that the data is easier to access among people who are not involved in the development of the database. By calling a procedure, the user can avoid having to write a query to find the data that they want to use.



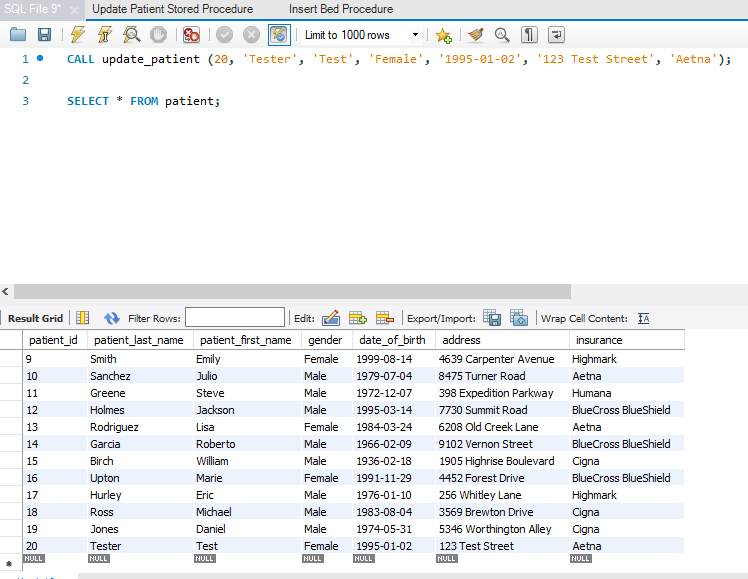
We’ll explain each of these down below:



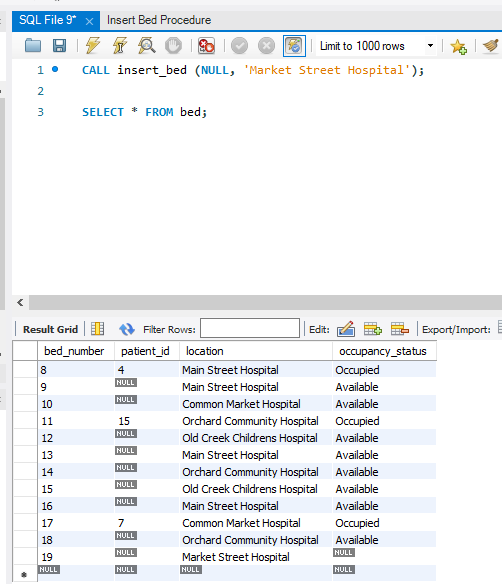
In this first example, we see an easy way to get all data from each table without having to write insert statements. Once a stored procedure is created, all one must do is call the procedure with the syntax CALL [procedure\_name]. It is helpful to familiarize oneself with what data is contained within the tables before writing queries to get useful information from it.



This is a notable change from the first project. Occupancy status is not a variable in the original database creation anymore and is managed by a stored procedure since this variable is entirely dependent on whether patient\_id is null or not. The insert\_occupancy\_status procedure adds this into the existing table so that the occupancy\_status will not require entries. Later, we’ll discuss the trigger used so that every new bed that’s added is also given an occupancy status based on what is entered.



In this procedure, the patient information can be updated by using a call statement and entering the data. Since patient information may change frequently, such as their address or possibly insurance provider, this can help make quicker changes than writing a new update statement each time.

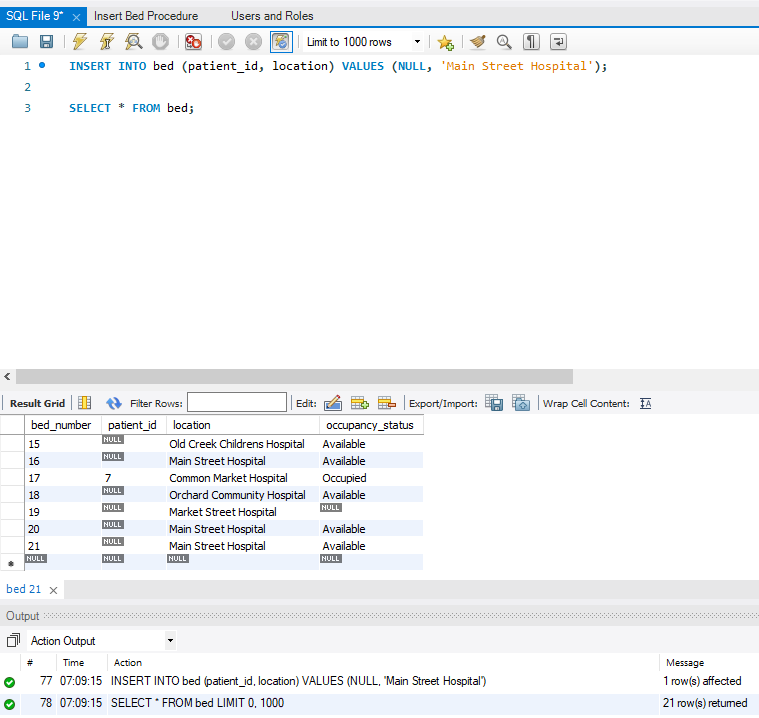


This procedure is particularly important as it adds any new beds that may be tracked by the database. Inserting a bed is as simple as calling the function and providing two values (patient\_id and location of the bed). Note that the occupancy status for the bed we just added is NULL. Therefore, we need the trigger which we will discuss later to automatically get the occupancy status for each new entry that gets added.

Graphical user interface, text, email

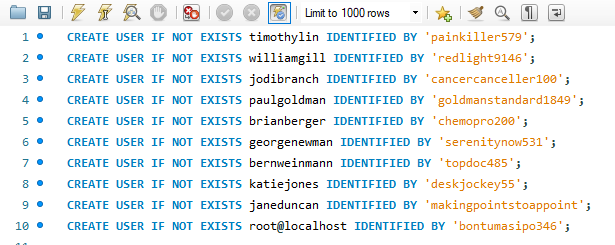
Description automatically generated

This is the trigger that we can use to make it so each new insertion into the bed table will automatically provide the occupancy status, based on if we give a null value for patient\_id or not.

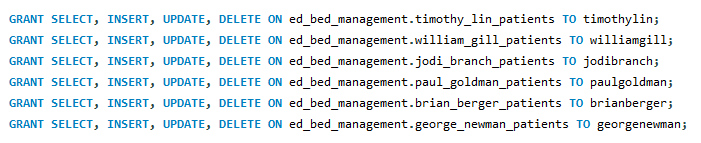


Here we see that the bed we just added has the occupancy status automatically filled based on our values provided. Since we gave patient\_id a NULL value, the occupancy status for that bed will be set to Available.

* User Authentication
  + User authentication for a database can be helpful in ensuring that only authorized users are able to access your database. In this case of potentially sensitive health information, it is imperative to ensure that only providers and other personnel will be able to access this database. We can set up a script defining the users who should be able to access the records and manage their privileges.



Here we see that all providers, managers, and receptionists can be given user access with the CREATE USER statement. We use the IDENTIFIED BY keywords after the username to create a password that that user will login with to access the database.

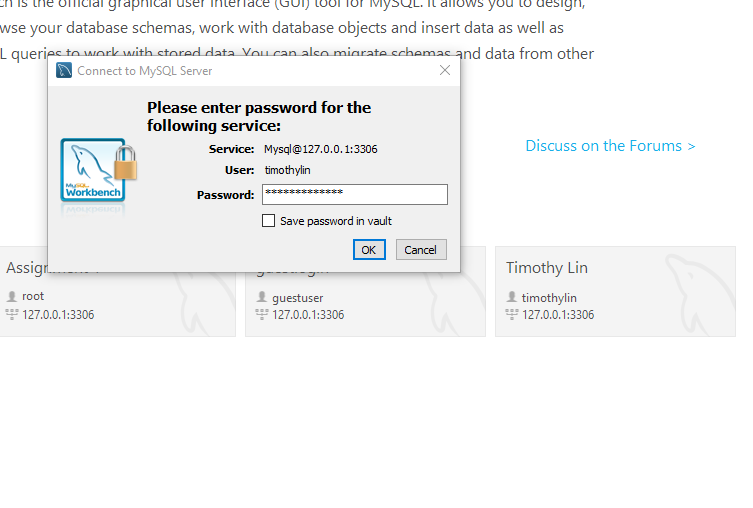


These statements grant all editing rights to providers for their respective patients. Only the patients which the providers are associated with will display within the view. These views will be discussed in the next section in more detail.

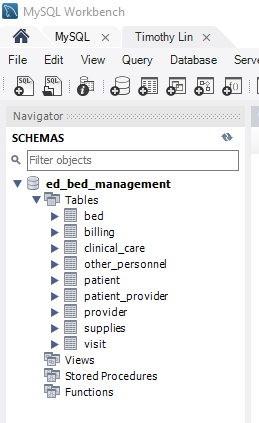
A screenshot of a computer

Description automatically generated

From the homepage you can test out a connection using the privileges that were given to a specific user. First you create a new connection and enter the username which was set as “timothylin” for the first user. Use the default schema of whatever you titled your schema in which you want them to access.

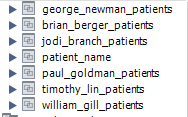


We can see the connection has been made. Once it is clicked on, you will be prompted to enter a password. In our case, we set the password for user timothylin to be painkiller579.

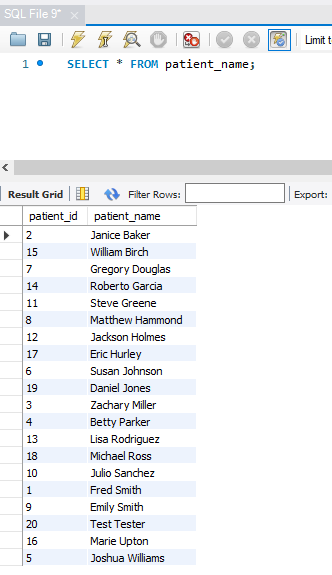


If done successfully, you will be within a new tab displaying the name of the new connection you have just made. The privileges for this connection will be based on what you set them as, so it’s useful to make sure that they are not able to access any tables that you do not want them to have access to.

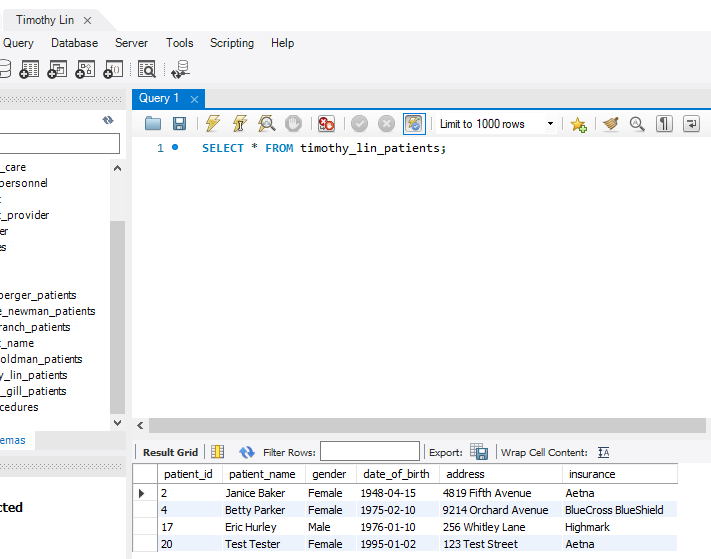
* Views
  + Next, we will address the views contained within the database. Views are useful for restricting access within this database. A view involves a select statement that can later be called with syntax SELECT \* FROM [view\_name].



Here are the views that are used within the database. First, we’ll start with the simple patient\_name view.



In this view, an aggregation function is used within a SELECT statement to combine last names and first names of a patient into a full name. It then takes the results and sorts then in alphabetical order by last name. Having this makes it easier to reference the patients by full name without having to call the CONCAT function each time you wish to display names in that format.



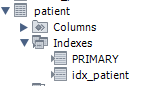
For this view, we will return to the Timothy Lin connection. This view specifically displays only patients that have records associated with Timothy Lin.

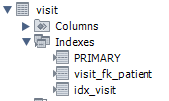
Graphical user interface, text, application, Word, email

Description automatically generated

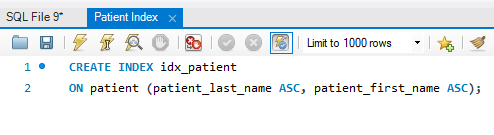
If we try to access another provider’s table on the Timothy Lin connection, we will get an error saying the command is denied. This shows that the permissions are set up properly from the original SQL file. Likewise, selecting timothy\_lin\_patients would be denied from using a connection with the other usernames.

* Indexes
  + Indexes help speed up searches within a database. These are especially useful in cases where there is a specific set of data that you wish to access frequently or have a large set of data. By default, each created table gets an index for its primary and foreign keys.

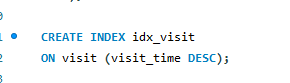




Here shown are the indexes of two tables within the database. The PRIMARY index references the primary key, and visit\_fk\_patient references the foreign key of patient\_id within the visit table. Both idx\_patient and idx\_visit contain the custom index variables.



In the patient index, we define the last name and first name as the reference to patient, and sort by last name first, followed by first name if the last names match. Using this index will make it easier to view the lists alphabetically and know exactly where to look instead of relying on the IDs which do not sort the patients in any order.



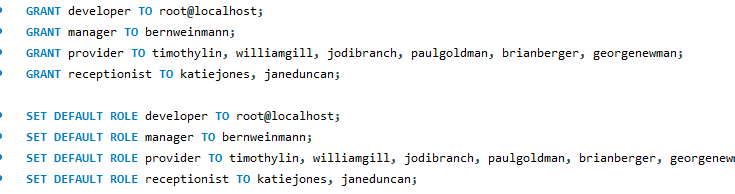
The visit index created is used to make the most recent visit times display first. This index can be thought of in a similar way to emails, where the newest results display at the top.

* Role Based Access
  + Previously we mentioned how to authenticate a user and set up permissions for that individual. But in a large database with many users, it may be too tedious to have to update permissions among individual users one by one. We can create roles given permissions and then assign users to those individual roles. This way, we can keep the permissions for a specified role constant, and simply move a user to another role in the event they change positions.

Table

Description automatically generated with low confidence

Here we have set up four different roles: developer, manager, provider, and receptionist. We use the GRANT statement in the same manner that we provided individuals with permissions, but what we put after the TO clause to be the role instead. It is also important to ensure that if a role changes among a user, they will retain their grants from their previous role, even if not included in the new role. That is why it’s important to revoke those permissions before setting up the new ones. This screenshot shows the revoking of permissions statements occurring before the new grants are issued to avoid individuals that changed roles from retaining those old permissions.



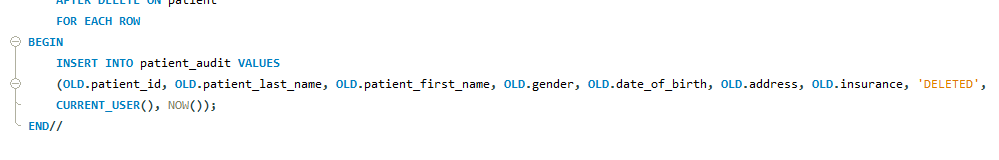
Here we grant the roles to each user we created. It is important to run these statements after the revoking and granting of permissions to each of the roles. The developer is responsible for creating the code and has all permissions to the database. The manager is responsible to managing all data so he will need to be able to perform all basic editing functions within each of the tables in the schema. The providers are solely responsible for all information pertaining to a patient’s visit as well as supplies and billing in some cases. The receptionist will also need to record information about a patient’s visit and be able to enter a patient with a specific provider. Both the providers and receptionists should be able to access information about beds, but the edits should be kept to the developers and managers.

* Audit Trails
  + Audit trails are a way of viewing all changes that have been made to a specific table within a schema. The way to set these up is to declare the same variables from the table you wish to audit, as well as important information about the change such as who made the edit and at what time the edit was made. It can also be useful to show what type of edit was made as well. We can use triggers to make these logs display once any insertion, update, or deletion occurs.

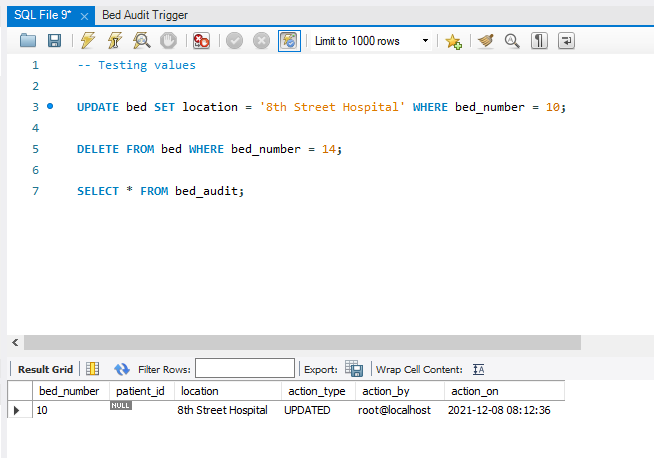
Graphical user interface, text, application, email

Description automatically generated

Here we have performed an INSERT, UPDATE, and DELETE statement. Each action type is noted in the result grid as well as the time at which it was performed. We get these values like this:



The CURRENT\_USER() function display the connected user and the NOW() function displays the date and time which the query was performed.



We can similarly do this for any other table that we want to track changes to. In this database, the patient and bed tables are the most important. Tracking changes to the beds table works similarly, and you can see what change was made to bed number 10 in this example and the time in which it was performed.

Tools Used

MySQL Workbench 8.0 CE (Database and ER Diagram)

Draw.io (UML Diagram)