Kern Medical Database

CMPS 3420 – Fall 2017
Bijan Mirkazemi & Robert Pierucci
Group 5

Table of Contents

1.1 Fact Finding Techniques and Information Gathering

In order to create a database for Kern Medical we need to gather certain facts and information to first design a conceptual database. In this section we will present methods for finding the necessary facts and show in detail how the conceptual database will be created for Kern Medical.

1.1.1: Introduction to Enterprise/Organization

Kern Medical is a hospital located in Bakersfield California. Kern Medical has 222 beds and is the only hospital that offers advanced trauma care between Fresno and Los Angeles. When a patient comes to the hospital with a sudden illness, they go to the emergency room where a nurse takes an initial assessment of their vitals. An emergency room physician makes a diagnosis and a care team is formed. This care team sees the patient and formulates a treatment for the patient. Our database will track this information.

1.1.2: Description of Fact Finding Techniques

They key to having a good conceptual database is to have a detailed understanding of the data that will be held within the database as well as who will be using the database and modifying this information. In gathering information for our database, we consulted with group member Robert Pierucci's wife Melanie Pierucci who is a registered dietitian at Kern Medical and is familiar with how the hospital is maintained. Using this valuable resource we are able to gather ideas and construct an effective conceptual database.

1.1.3:Scope of the Conceptual Database

When designing a conceptual database it is important to have a clear understanding of the part of Kern Medical that we would like our database to represent. We will be designing our database with the goal as keeping track of interactions between patients and hospital employees. For this database we will be focusing on our employees as being doctors, nurses and dietitians. The database will document patients and detail their diagnoses (or reason they come to the hospital) and treatment plans formulated by care teams consisting of doctors and dietitians. While a hospital database is a massive undertaking, we limit ourselves to a small portion of this in order to meet the time constraints that we have on this project.

1.1.4: Itemized Descriptions of Entity Sets and Relationship Sets

After gathering the needed information for the conceptual database, we are able to construct an ER model which represents the data as entities and relationships. Below is an description of each of the entity type definitions that will be needed for our database as well as the relationship among these entity types.

Entity Type Definitions

Hospital Employee: Employee ID, SSN, Name, Address, Phone, License Type, License Number, Start Date, End Date
A **hospital employee** works for the hospital. They have a unique ID as well as basic identifying information, and a start date and end date to employment. The License Type describes the type of employee, which our database covers nurses (RN), doctor (MD), and dietitian (RD). They each also have a unique license number.

Case: Case ID, Blood Pressure Sys, Blood Pressure Dia, Heart Rate, Respiration Rate, Visit Date
For each visit to the hospital a case record is created by a Nurse. This record has blood pressure systolic and diastolic fields as well as heart rate, respiration rate and the date of the visit.

Patient: Patient ID, SSN, First Name, Last Name, Address, Phone Number, Date of Birth, Sex, Insurance, Primary Language
A **patient** is admitted to the hospital, given a patient id, and has basic identifying information. They also have their insurance type and primary language spoken.

Care Team: Team ID, Department
A **care team** is formed by a group of hospital employees. One of the members of the care team is a Registered Dietitian, while the other members are various doctors. The care team belongs to a department and has a unique ID.

Prescription: <u>Prescription ID</u>, Medication, Dosage, Frequency, Start Date, End Date
A **prescription** is created whenever medication is prescribed to a patient. The prescription has a unique ID as well as medication name, dosage, frequency of use, a start and end date.

Diagnosis: <u>Diagnosis ID</u>, Diagnosis, Status
A **diagnosis** is made by doctors. This diagnosis determines the patient's illness. A diagnosis has a unique ID as well as the diagnosis itself and a status field.

Labs: <u>Lab ID</u>, Blood Urea Nitrogen, Calcium, Carbon Dioxide, Chloride, Creatinine, Glucose, Potassium, Sodium
Labs are the traits of a blood draw that is made once per visit. The initial blood draw takes a panel of traits called a basic metabolic panel, which consists of measuring Blood Urea Nitrogen, calcium, carbon dioxide, chloride, creatinine, glucose, potassium and sodium levels. Additionally the Labs has a unique ID.

Relationship Types Definitions

Hospital Employee **takes initial** Case; Cardinality 1:N;
Participation: Hospital Employee; partial / Case; total

Hospital Employee **make** Diagnosis; Cardinality 1:N;
Participation: Hospital Employee; Partial / Diagnosis; Total

Hospital Employee **form a** Care Team; Cardinality M:N;
Participation: Hospital Employee; Partial / Care Team; Total

Patient **seen by** Care Team; Cardinality M:N;
Participation: Care Team; Total / Patient; Total

Labs **recorded in** Diagnosis; Cardinality 1:1;
Participation: Labs; Total / Diagnosis; Total

Case **recorded in** Diagnosis; Cardinality 1:1;
Participation: Case; Total / Diagnosis; Total

Patient **undergoes** Case; Cardinality 1:N;
Participation: Patient; Total / Assessment; Total

Diagnosis **prescribes** prescription; Cardinality 1:N;
Participation: Diagnosis; Partial/ Prescription; Total

1.2 Conceptual Database Design

In this section we will be using our ER Model to create a conceptual database. A conceptual database is an accurate representation of the organization, in our case, Kern Medical. Using our entities and relationships briefly described in section 1.1.4, we will now create sets based on these entities and relationships. An *entity set* is rather self explanatory, it is the a collection of all the entities of one type in the database. Similarly, a *relationship set* is a gathering of the relationships in the database that show the connections between the entities. In the following sections we will give a detailed description of these various entity and relationship sets.

1.2.1 Entity Set Description

Entity Name: Hospital Employee
Description: The hospital employee entity stores basic information for each employee such as name, address, SSN, phone number, and a start date and end date of employment. Employees are designated as either a registered dietitian (RD), Medical Doctor (MD) or registered Nurse (RN) on the License Type Field. A unique License Number is also provided. An entry is created only when a new employee is hired. Entries are never deleted except in extreme circumstances. Entries may be updated when a hospital employee updates their address. Hospital employees are distinguished from one another by their Employee ID, Social Security Number and License Number.

Candidate Keys: Employee ID, SSN, License Number
Primary Keys: Employee ID
Entity Type: Strong
Indexed Fields: Employee ID, SSN, License Number

Attributes:

(on next page)

Attribute Name	**EmployeeID**	**SSN**	**Name**	**Address**
Description	Unique distinguishing ID. Can be auto-increment	Social security number	First name, Middle name, Last name	Street Address, City, State, Zip code
Domain/Type	Integer	Integer	String, String, String,	String, String, String, Integer
Value/Range	0 - MaxID	Any 9-digit integer	Any, Any, Any,	Any, Any, Any, 00000-99999
Default Value	None	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	Yes	No	No
Single or Multi-Valued	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Composite	Composite

Attribute Name	**Phone**	**License Type**	**License Number**	**Start Date**	**End Date**
Description	Employee primary contact phone number	Type of Employee	Unique License Number	Hire date	Resignation date
Domain/Type	Integer	String	Integer	Date	Date
Value/Range	Any 10-digit integer	“MD”, “RD”, or “RN”	000000000 - 999999999	Any	Any
Default Value	None	None	None	None	None
Null Value Allowed	No	No	No	No	Yes
Unique	No	No	Yes	No	No
Single or Multi-Valued	Single	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple	Simple

Entity Name: Case
Description: When a patient first comes to the hospital, a nurse hospital employee will create a case. A case has the patient’s vitals recorded as well as the date of the visit. Entries are created whenever a patient comes for a visit. Entries are never deleted except in extreme circumstances. Entries are unable to be updated.

Candidate Keys: Case ID
Primary Keys: Case ID
Entity Type: Strong
Indexed Fields: Case ID

Attributes:

Attribute Name	**Case ID**	**Blood Pressure Sys**	**Blood Pressure Dia**	**Heart Rate**
Description	Unique distinguishing ID. Can be auto-increment	Systolic Blood Pressure Reading	Diastolic Blood Pressure Reading	Patient's current heart rate reading
Domain/Type	Integer	Integer	Integer	Integer
Value/Range	0 - MaxID	0 - 300	0-300	0-500
Default Value	MaxID + 1	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	No	No	No
Single or Multi-Valued	Single	Single	Single	Single
Simple or Composite	Simple	Composite	Simple	Simple

(Cont on next page)

Attribute Name	**Respiration Rate**	**Visit Date**
Description	Current respiration rate of patient.	Date of patient visit to hospital
Domain/Type	Integer	Date
Value/Range	0 - 100	Any

Default Value	None	None
Null Value Allowed	No	No
Unique	Yes	No
Single or Multi-Valued	Single	Single
Simple or Composite	Simple	Composite

Entity Name: Patient
Description: The patient entity stores basic information about a patient such as name, address, phone number, etc. It also has a unique patient ID to distinguish patients from one another. The patient entity also has fields that describe the patient's primary language, self-diagnosis and the insurance that they carry. Entries are created whenever a new patient arrives at the hospital. Entries are never deleted except in extreme circumstances. Entries may be updated when patient information changes (such as address, insurance carried, or phone number.)

Candidate Keys: Patient ID, SSN,
Primary Keys: Patient ID,
Entity Type: Strong
Indexed Fields: Patient ID, SSN, Last Name,

Attributes:

Attribute Name	**Patient ID**	**SSN**	**First Name**	**Last Name**
Description	Unique distinguishing ID. Can be auto-increment	Social security number	First name	Last name
Domain/Type	Integer	Integer	String	String
Value/Range	0 - MaxID	Any 9-digit integer	Any	Any
Default Value	MaxID + 1	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	Yes	No	No
Single or Multi-Valued	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

(continued on next page)

Attribute Name	**Address**	**Phone Number**	**Date of Birth**	**Sex**
Description	Street Address, City, State, Zip code	Patient primary contact phone number	Birth date	Gender of patient
Domain/Type	String, String, String, Integer	Integer	Date	Character
Value/Range	Any, Any, Any, 00000-99999	Any 10-digit integer	Any	"M" - Male, "F" - Female
Default Value	None	None	None	None

Null Value Allowed	No	No	No	No
Unique	No	No	No	No
Single or Multi-Valued	Single	Single	Single	Single
Simple or Composite	Composite	Simple	Simple	Simple

Attribute Name	**Insurance**	**Primary Language**
Description	Type of insurance	Primary language patient speaks
Domain/Type	String	String
Value/Range	Any	Any
Default Value	None	None
Null Value Allowed	No	No
Unique	No	No
Single or Multi-Valued	Single	Single
Simple or Composite	Simple	Simple

Entity Name: Care Team
Description: The care team entity stores basic information about the care team who follows patients and determines a treatment plan based on their needs. The entity has information such as the department and a unique identifying ID. Entries are created whenever a care team is formed. Entries are never deleted except in extreme circumstances. Entries are rarely updated (only to fix errors).

Candidate Keys: Care Team ID
Primary Keys: Care Team ID
Entity Type: Strong
Indexed Fields: Care Team ID

Attributes:

Attribute Name	Care Team ID	Department
Description	Unique distinguishing ID. Can be auto-increment	Identifies the department of the team (Medical Team, Surgical Team, etc)
Domain/Type	Integer	String
Value/Range	0 - MaxID	Any
Default Value	MaxID + 1	None
Null Value Allowed	No	No
Unique	Yes	No
Single or Multi-Valued	Single	Single
Simple or Composite	Simple	Simple

Entity Name: Prescription
Description: The prescription entity contains information about what the care team believes the patient needs based on their illness. This includes prescriptions prescribed in the diagnosis and includes the medication, dosage, frequency, and a start and end date. The care team formulates the treatment and then it is recorded in the diagnosis for each patient. Entries are created whenever an assessment is done. Entries are never deleted except in extreme circumstances. Entries are unable to be updated.

Candidate Keys: Prescription ID
Primary Keys: Prescription ID
Entity Type: Strong
Indexed Fields: Prescription ID

Attributes:

Attribute Name	Prescription ID	Medication	Dosage	Frequency
Description	Unique distinguishing ID. Can be auto-increment	Identifies the Medication prescribed to the patient	Identifies the dosage to take of the prescription given (MG)	Identifies the frequency to take of the prescription given
Domain/Type	Integer	String	Float	String
Value/Range	0 - MaxID	Any	Any	Any
Default Value	MaxID + 1	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	No	No	No
Single or Multi-Valued	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

(continued on next page)

Attribute Name	Start Date	End Date
Description	Date to start prescription	Date to end prescription
Domain/Type	Date	Date
Value/Range	Any	Any
Default Value	None	None
Null Value Allowed	No	No
Unique	No	No
Single or Multi-Valued	Single	Single
Simple or Composite	Simple	Simple

Entity Name: Diagnosis
Description: The diagnosis entity contains the evaluation of the Doctor on the patient. When the physician sees the patient he will assess what is wrong, this is the diagnosis. This entity will also contain a chronological status patient’s visit. (A key explaining the values follows the attribute table). Entries are created whenever a diagnosis is done. Entries are never deleted except in extreme circumstances. Entries are unable to be updated.

Candidate Keys: Diagnosis ID
Primary Keys: Diagnosis ID
Entity Type: Strong
Indexed Fields: Diagnosis ID

Attributes:

Attribute Name	Diagnosis ID	Diagnosis	Status
Description	Unique distinguishing ID. Can be auto-increment	Describes the diagnosis the doctor has given the patient	Status indicating where the staff is in the diagnosis for the patient (chronological)
Domain/Type	Integer	String	Integer
Value/Range	0 - MaxID	Any	0-2*
Default Value	MaxID + 1	None	None
Null Value Allowed	No	No	No
Unique	Yes	No	No
Single or Multi-Valued	Single	Single	Single
Simple or Composite	Simple	Simple	Simple

***Status Key:**
0 - Awaiting Assessment
1 - Assessment Complete, Awaiting Diagnosis (Seen by Nurse)
2 - Diagnosis Complete, Awaiting Treatment (Seen by Nurse & Doctor)

Entity Name: Labs
Description: The labs entity contains the lab values that are taken from the patient. When a patient comes to the hospital, a standardized set of lab values are drawn from the patient called a basic metabolic panel. These values are stored here.Entries are created whenever a lab draw is done. Entries are never deleted except in extreme circumstances. Entries are unable to be updated.

Candidate Keys: LabID
Primary Keys: LabID
Entity Type: Strong
Indexed Fields: LabID

Attributes:

Attribute Name	LabID	Blood Urea Nitrogen	Calcium	Carbon Dioxide

Description	Unique distinguishing ID. Can be auto-increment	Describes the blood urea nitrogen level of the patient	Describes the calcium level of the patient	Describes the carbon dioxide level of the patient
Domain/Type	Integer	Integer	Float	Integer
Value/Range	0 - MaxID	Any	Any	Any
Default Value	MaxID + 1	None	None	None
Null Value Allowed	No	No	No	No
Unique	Yes	No	No	No
Single or Multi-Valued	Single	Single	Single	Single
Simple or Composite	Simple	Simple	Simple	Simple

Chloride	**Creatinine**	**Glucose**	**Potassium**	**Sodium**
Describes the chloride level of the patient	Describes the creatinine level of the patient	Describes the glucose level of the patient	Describes the potassium level of the patient	Describes the sodium level of the patient
Integer	Float	Integer	Float	Integer
Any	Any	Any	Any	Any
None	None	None	None	None
No	No	No	No	No
No	No	No	No	No
Single	Single	Single	Single	Single
Simple	Simple	Simple	Simple	Simple

1.2.2 Relationship Set Description

Relationship: takes initial
Description: A nurse hospital employee takes an initial case of the patient. This initial case taken by the nurse is then recorded in the diagnosis which corresponds to a patient. One nurse can take many initial cases. All nurses participate in this relationship as an assessment is required upon a patient's visit.
Entity Sets Involved: Nurse, Case
Mapping Cardinality: 1:N
Descriptive Field: None
Participation Constraint: Total

Relationship: undergoes
Description: A patient undergoes a case upon each visit to the hospital. Each patient can undergo many cases (when they visit the hospital multiple times) but each case corresponds to one patient. Every patient receives a case.
Entity Sets Involved: Patient, Case
Mapping Cardinality: 1:N
Descriptive Field: None

Participation Constraint: Total

Relationship: recorded in
Description: When a case is completed, that case is recorded in the diagnosis for others to see. Each case is recorded in a single diagnosis, and each diagnosis can have only one case. Every case is recorded in a diagnosis.
Entity Sets Involved: Case, Diagnosis
Mapping Cardinality: 1:1
Descriptive Field: none
Participation Constraint: Total

Relationship: recorded in
Description: When a set of labs are completed, that lab set is recorded in the diagnosis for others to see. Each lab is recorded in a single diagnosis, and each diagnosis can have only one lab set. Every lab set is recorded in a diagnosis.
Entity Sets Involved: Labs, Diagnosis
Mapping Cardinality: 1:1
Descriptive Field: none
Participation Constraint: Total

Relationship: forms
Description: Various hospital employees form a care team. A care team is made up of a group of employees based on the patient's needs. Each employee can form many care teams as well as each care team can be formed by many employees. All employees do not form care teams.
Entity Sets Involved: Hospital Employee, Care Team
Mapping Cardinality: M:N
Descriptive Field: None
Participation Constraint: Partial

Relationship: makes
Description: A diagnosis is made on a patient. The diagnosis includes all information related to the patient during their stay at the hospital. Each diagnosis is on one specific patient but each patient can have many diagnoses. (In the instance of visiting the hospital multiple times)
Entity Sets Involved: Diagnosis, Patient
Mapping Cardinality: 1:N
Descriptive Field: None
Participation Constraint: Total

Relationship: seen by

Description: During their stay at the hospital, a patient is seen by a care team. The care team is made up of doctors based on the patient's specific needs. Over time, each patient is seen by many care teams and each care team can see many patients. All patients are seen by a care team.
Entity Sets Involved: Patient, Care Team
Mapping Cardinality: M:N
Descriptive Field: None
Participation Constraint: Total

Relationship: prescribes
Description: Within a diagnosis the doctor may prescribe medication to a patient. Not all diagnoses will have prescriptions. A diagnosis may have many prescriptions.
Entity Sets Involved: Diagnosis, Prescription
Mapping Cardinality: M:N
Descriptive Field: None
Participation Constraint: Partial

1.2.3 Related Entity Set

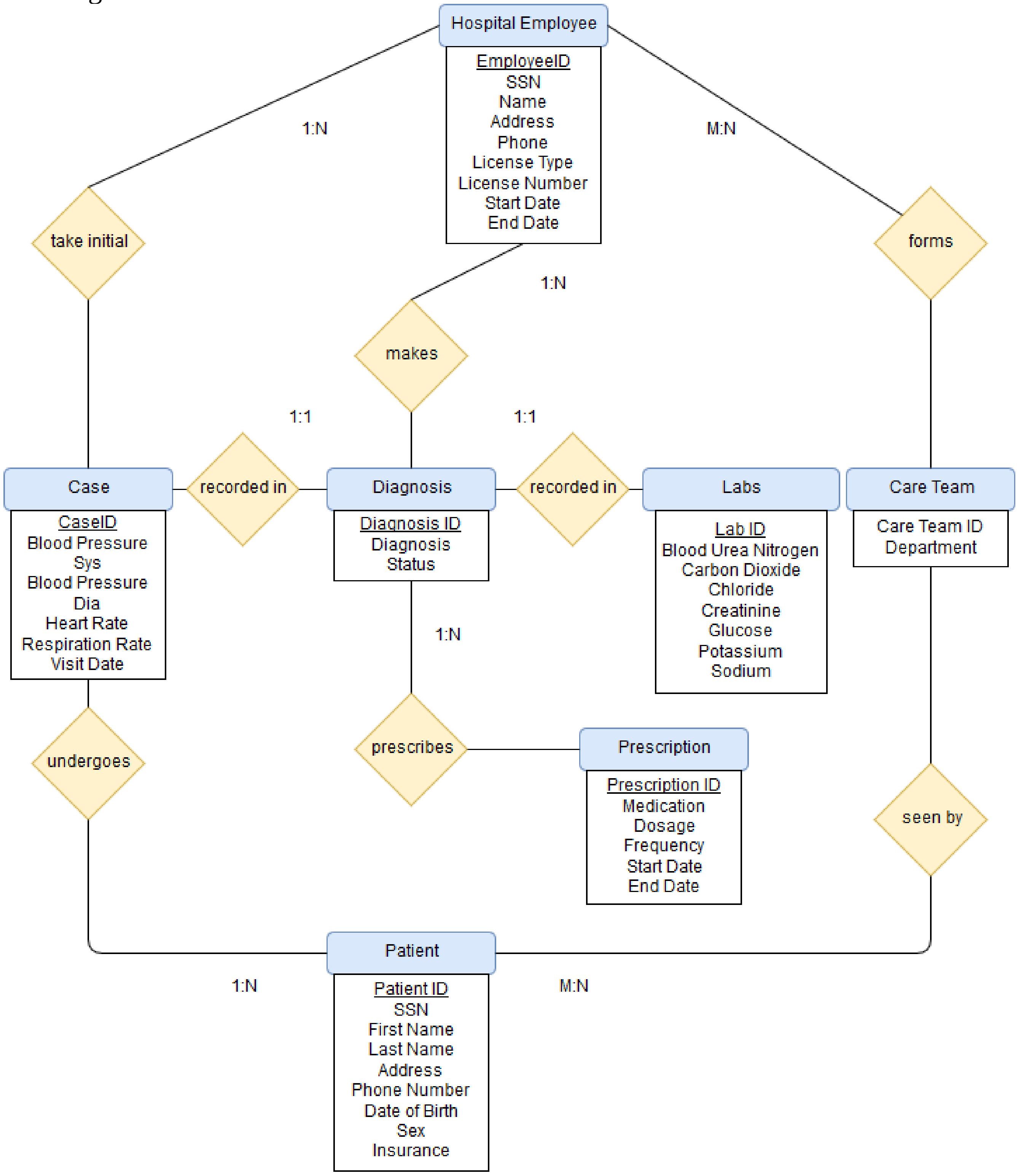
Specialization is the process of defining a set of subclasses of an entity type. When using specialization, the superclass of the specialization is the entity. The subclasses then inherit the attributes of the entity that they are derived from. For our database we used Specialization on the Hospital Employee entity to distinguish employee types from each other.

Generalization is the process of unifying multiple related subclasses together under a single superclass. After much discussion and revisions of our entity relation model, it was determined that our nurses, doctors and dietitians were very similar and it was best to bring them together under the hospital employee superclass.

Both specialization and generalization share constraints on "disjointedness" and "completeness". Our database specialization of hospital employee to doctor, nurse and dietitian is disjoint because an employee can only be one of the different job types. The completeness constraint on our specialization is total because a hospital employee must be one of three possible types (doctor, nurse, dietitian). This constraint exists because the scope of our conceptual database excludes other types of employees.

1.2.4 ER Diagram

The Entity-Relationship Diagram (or ER Model) is essential to have as a visual representation of the entities and relationships of an organization's database. The ER Diagram displays all of the entities and relationships as well as cardinalities and participation constraints that are detailed throughout the report. On the next page is our ER Diagram.



Entity Relationship Diagram

[This page is left intentionally blank]

2 Conceptual Database and Logical Database

While a conceptual database's purpose is for one to easily visualize how a database will look, a logical database is used in implementing the database into software. For this reason, we must convert our database from a conceptual to a logical design. The following section will detail our steps to convert our E-R model from the previous chapter into a relational model to prepare our database for implementation.

Section 2 will describe the E-R and relational database models, detailing their origins, purpose as well as similarities and differences. Later we will discuss the methods used to convert from the E-R model to the relational model. Continuing on we will use the methods described to convert our own E-R Model to a relational model for our database.

Beyond this, we will create sample data and design queries for our data using relational algebra, tuple relational calculus and domain relational calculus and will present that information.

2.1 E-R Model and Relational Model

The ER Model and Relational Models both represent different database designs. While the ER Model represents a conceptual database design the Relational Model represents a logical database design. Now we will discuss and compare both the ER and Relational Models.

2.1.1 Description of E-R Model and Relational Model

Although the E-R Model and Relational Model are both used for Databases these two methods were not developed by the same person nor at the same time. The E-R Model was first introduced in a publication in 1976 by Peter Chen. When Peter Chen developed the E-R Model he intended for it to represent a conceptual database that represents objects and the ways in which they are related to one another. This model is depicted best using a simple, easy-to-read, diagram that the average person would have no trouble understanding.

The Relational Model, ironically, was developed 6 years earlier than the E-R Model in 1970 by Edgar F. Codd. This model is more complex than the E-R Model and represents how the data will be implemented into software. In the Relational Model all data is represented as tuples that are grouped into relations. Several commercial level Database Management Systems (DBMS) such as Oracle and MySQL have adopted the Relational Model design. There are three formal query languages that go hand in hand with the Relational Model; relational algebra, tuple relational calculus, and domain relational calculus.

Another way in which the E-R Model and Relational Model differ is in how they are defined. The E-R Model, which stands for Entity-Relationship Model, is defined by two key features, entities and relationships. The objects in an E-R Model are known as the “entities” and the ways in which they are related to one another is, obviously, the “relationships”.

The Relational Model, on the other hand, is defined slightly differently. Rather than having entities which are linked to one another with relationships, the Relational Model does away with this concept and introduces converting all of the entities and relationships to simply, relations. The Relational Model consists of tuples which are flat lists of related values and these tuples are grouped into relations. Essentially, each tuple is a row in the table of data.

The E-R Model and Relational Model also have different purposes. The E-R Model is designed to better understand the objects which will be making up the database and how they are related to one another. Once we have a clear understanding of the different entities and relationships, the E-R Model can be converted into a Relational Model. The Relational Model is slightly more difficult to understand for a person who does not have a background in Computer Science but it accurately represents the data how it will be implemented in the software. This is the purpose of the Relational Model.

2.1.2 Comparison of Two Different Models

The E-R model has many advantages over the relational model. The E-R model is a much better method of communicating ideas between those in business who are incapable of understanding the terminology of software engineering with those who implement the database software for the company. The E-R model excludes details of software implementation which accomplishes this task. It focuses on the conceptual design of the database.

Because the E-R model is geared toward creating an understanding among non-engineers, it lacks much of the logical qualities used in software implementation. For this reason, it is impossible to create queries with mathematical logic from the E-R model alone. There are also multiple ways to design a single E-R model which means that the same database can have multiple E-R models.

The relational model also has multiple advantages and disadvantages. The relational model is formal and standardized, making it a good tool for software implementation of the database. Because the relational model is as such in its use of discrete mathematics and formal query languages, it can be easily translated into computer data.

The disadvantages are mainly due to the technicality of the relational model. Those in business with no knowledge of software engineering have difficulty understanding the relational model. Because of this, the relational model typically only has use for those implementing the database and is otherwise useless to the rest of the organization.

The E-R and relational models do share some similarities. Both models can represent the structure of data in a clear and concise fashion. The E-R model uses entity and relationships types while the relational model uses relations alone. Both the entity and relationships contain names and lists of attributes as do relations. Both models incorporate constraints which defines how data is to be related to one another. Also, both models implement the idea of Tuples and Schema (aptly named instances and types in E-R model).

The E-R and relational models also have differences. The main difference between them is that the relational model describes everything through the concept of a relation while the E-R model does so through entities and relationships. Also, the relational model does not allow the use of composite, multivalued attributes while the E-R model does. The relational model also has no cardinality or participation constraints and instead utilized integrity constraints to create consistency between relations referencing each other.

2.2 Conversion of Conceptual Design to Logical Design

Now that we have a better understanding of both models, the process by which an E-R Model is converted into a Relational Model will be explained. This process is completed in multiple steps that can be broken into two different areas. The first will be converting the various entity types to relations and then, the slightly more extensive task of converting the different relationship types to relations. And finally the methods of defining the database constraints that will ensure the integrity of the data.

2.2.1 Converting Entity Types to Relations

When an E-R Model is converted into a Relational Model all of the entity types in the E-R Model will be converted into relations in the Relational Model. Each relation in the Relational Model has a list of simple attributes therefore the entity types with composite or multi-valued attributes must be modified in some way.

In this section we will explain how each entity type in the E-R Model will be converted into relations in the relational model. First we will explain how to represent weak and strong entity types, then simple and composite attributes, and finally single and multiple valued attributes.

Strong Entities:

Each strong entity, E, in the E-R Model is converted into a relation, R, in the Relational Model. The simple attributes are included in the relation as attributes and only the simple component attributes of composite attributes are included (See mapping of composite attributes). Then one of the key attributes of E is chosen as the primary key for R. If the chosen key of E is composite, then the set of simple attributes that form it will together form the primary key of R.

Weak Entities:

For each weak entity type W in the E-R Model with an owner of entity type E, a relation R is created. This relation R includes all of the simple attributes and simple components of components of composite attributes of W. In addition, a foreign key attribute of R, is included and is the primary key of the owner, E, of the weak entity W. This will take care of mapping the identifying relationship type of W. If there is a partial key for the weak entity W then the partial key of W and the primary key of E will form the primary key for the relation R.

Simple and Composite Attributes:

Simple attributes from the E-R Model are transferred directly to the Relational Model and are not changed at all. Composite attributes are different in that the attribute is broken into the simple components of the attribute and each of those is a corresponding attribute in the Relational Model.

Single and Multivalued Attributes:

The ways to represent single valued attributes and multivalued attributes are different. Single valued attributes of the entity E simply become attributes of the relation R. The process for mapping multivalued attributes is slightly more complex. For each multivalued attribute A, a new relation R is created which will include an attribute corresponding to A plus the primary key of the entity type or relationship type that has A as a multivalued attribute. The primary key of R will be a combination of A and the primary key mentioned above. If the multivalued attribute is composite, then it’s simple components are included.

2.2.2 Converting Relationship Types to Relations

While the ER model has two primary concepts - entities and relationships - the relational model has only one - the relation. Relationship types are to be represented by relationship schemas. Converting relationship types to relations is not the same as with entities as there are specific rules that must be followed. This section will explain the methods for converting the following relationships:

- Cardinality Constraints (1:1, 1:N, M:N)
- Superclass / Subclass concepts “IsA” relationship
- “HasA” relationship
- Relationship types involving other relationship types
- Recursive Relationships(involving a single entity type)
- Relationships with 2 or more entity types
- Relationship and Union Types (or categories)

<u>Mapping of binary relationship types with a 1:1 cardinality constraint</u>

For each binary 1:1 relationship type R, relations A and B correspond to R and are one to one. This means that each instance of Relation A should be related to exactly one instance of Relation B. Three methods to represent this constraint are listed below:

1. **Foreign Key Approach:** This method takes the primary key of R_A and makes it into a foreign key attribute of R_B or vice versa. All simple attributes as well as simple components of composite attributes that belong to the relationship type also become attributes of the relation with the foreign key.

2. **Merged Relation Approach:** This approach combines the attributes of R_A and R_B into a single relation.

3. **Cross Reference Approach:** This approach creates a new relation R to represent the relationship type. R is referred to as a “relationship relation”. This new relation will contain the primary keys of both R_A and R_B as foreign key attributes as well as the simple attributes of the relationship type. The primary key for R is one of the foreign keys.

Advantages/Disadvantages:

The **Foreign Key Approach** decreases the number of join operations needed when querying. However this approach should only be used if one of the entity types has total participation in the relationship. Otherwise it will end up wasting storage space by allowing the foreign key attribute to be NULL for relations that are not participating.

The **Merged Relation Approach** is not a very good approach considering that if you can merge two relations into one, then this should have been done initially during the conceptual design phase.

The **Cross Reference Approach** increases the number of join operations needed when querying, but it is a good method if neither of the participating entity types have total participation.

Mapping of binary relationship types with a 1:N cardinality constraint

_____For each binary 1:N relationship type R, relations A and B correspond to R and are one to many. This means that one instance of relation A can be related to multiple instances of relation B. Only one instance of relation B can be related to one instance of relation A. Two methods to represent this constraint are listed below:

1. **Foreign Key Approach:** This approach is the same as the 1:1 relationship type approach previously described. The difference here lies in that the foreign key and relationship type attributes must belong to the relation derived from the entity on the N-side of the relationship (or relation B in this case). This is due to the fact that entities on that side can only be related to entities of relation A one time making their participation unique.

2. **Cross Reference Approach:** This approach is the same as the 1:1 relationship type approach except that the primary key of relation R must be the foreign key for the relation that represents the entity on the N side of the relationship.

Advantages/Disadvantages
The advantages for the **Foreign Key Approach** and **Cross Reference Approach** in 1:N relationship types is the same as it is in the 1:1 relationship types.

Mapping of binary relationship types with a M:N cardinaltiy constraint

For each binary M:N relationship type R, a new relation S is created which represents R. An M:N relationship means that each instance of relation A can be related to multiple instances of relation B, as well as each instance of relation B can also be related to multiple instances of relation A.

The **Cross Reference Approach** is the only method of converting an M:N relationship type. A new relationship relation is created to represent the relationship type. This new relationship contains the primary keys of R_A and R_B as foreign keys. It also contains all of the simple and simple-component relationship type attributes. Both foreign keys are combined to form the primary key.

Mapping of superclass / subclass for the "IsA" relationship

The "IsA" relationship refers to the disjoint subclasses of a superclass entity type. This means that the entity belongs to a single subclass. Three methods to represent these relationships with relations are listed below:

1. **Multiple relations - superclass and subclass:** In this approach a relation S is created for superclass C and relation L is created for subclass S. The relation S contains the attributes of superclass entity C. The subclass L contains the attributes of the subclass entity as well as the primary key of superclass relation S which is a foreign key. While it is a foreign key, it acts as the primary key in the subclass relation L.

   The advantages of this approach are that this approach works for every type of superclass/subclass relationship (disjoint, overlapping, total, partial). Because it creates a separate superclass relation, this method requires more join operations.

2. **Multiple relations - subclass only:** In this approach only entities that are subclasses are given their own relations. The subclass relation L contains the union of the attributes from the superclass entity type C and the subclass entity type C_{sub}. This approach will only work for a specialization in which the subclasses are total meaning that every entity in the superclass must belong to at minimum a single subclass. This approach is only recommended if the specialization has the disjointedness constraint. If the specialization is overlapping, the same entity may be duplicated in multiple relations.

   This approach requires fewer join operations, yet it only works for relationships with total participation. By this I mean that when an entity must belong to one of the subclasses.

3. **Single relation with one type attribute:** This approach has a single relation L that is created containing the attributes from superclass C and the attributes from all of subclasses L_i combined. The relation L also has a discriminating attribute ("Type") in which this value indicates the subclass L_i that each tuple belongs to, if any at all.

This approach again requires fewer join operations than all of the other approaches. However, the relation can be become extremely large due to all of the attributes contained within. Also attributes of the subclasses in which an entity does not belong will be NULL resulting in possibly many NULL values and wasted space. This approach is best when the subclasses are disjoint and have few unique attributes.

Mapping of superclass / subclass for the "HasA" relationship

The "HasA" relationship refers to when entity types are overlapping subclasses of a superclass entity type. In other words, this means that an entity may belong to multiple subclasses. Two methods to represent these relationships as relations are listed below:

1. **Multiple relations - superclass and subclass:** This approach is the same here as it was for the "IsA" superclass/subclass relationship. The same advantages and disadvantages are also present.

2. **Single relation with multiple type attributes:** This approach has a single relation L created with the union of attributes from both the superclass C and all of the subclasses L_i combined. Also a boolean attribute is included for each of the subclasses. The boolean's purpose is to indicate that a relation instance tuple belongs to the subclass L_i in which the boolean is true.

   This approach requires less joins on queries. However, many NULL values will exist since the attributes for the subclasses will not always be present if the entity does not belong to the subclass. In turn, this has a great amount of wasted data.

Mapping of relationship types involving other relationship types

_____When mapping relationship types where an entity or relationship is associated with another relationship type (entity or relationship) it is best to create a primary key for each relationship involved. Once each relationship has a primary key, then the foreign key approach or the cross reference approach can be used depending on the cardinality.

Mapping of recursive relationships

_____A recursive relationship occurs when an entity type converted to relation R is related to itself. There are two ways to map this relationship:

1. **Foreign Key Approach**: This approach requires that the foreign key attribute of R references the primary key of itself.

This approach requires less join operations, but a negative is that relation instance tuples that do not participate in the relationship will have NULL values for the foreign key, in turn wasting data.

2. **Cross-Reference Approach**: This approach requires the creation of a new relation R_1 that represents the recursive relationship. Two foreign keys are in use that both reference the primary key of R. The combination of both foreign keys also form the primary key of R_1.

   This approach will not have problems having relation instance tuples with NULL values. However, the disadvantage is that using queries will require more join operations.

Mapping of relationships with more than 2 entity types

When a relationship type links more than two entity types, a relation R is created to represent that relationship type. R will hold primary keys of each participating entity type as foreign keys. These foreign keys will be combined to form the primary key of R, although some foreign keys could be excluded from the primary key if their corresponding relation represents an entity on the 1-Side of 1:N cardinality constraint.

Mapping of relationships and union types

_____A union type or category occurs when a relation for a subclass entity belongs to multiple superclass entities. For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, which is called a surrogate key, when creating a relation to correspond to the category or union type. The keys of the defining classes are not the same, so none of them can be used exclusively in defining all entities from a specific category. Also if multiple attributes are superclasses of the same subclass entity, then the relation tuples of the superclass entities share a value for the surrogate key.

2.2.3 Database Constraints

To ensure our data in our database is meaningful and coherent we must implement constraints on our database. In order for the relational database model to accurately represent the ER Model, the tuples in the relation state must satisfy certain rules and conditions. The data will be meaningless if any rules are violated. This section will detail constraint types and also explain how they are enforced in a database management system. The constraints to discuss are:

- Domain Constraints
- Entity Constraint
- Primary and Unique Key Constraint
- Referential Constraint
- Check Constraints and Business Rules

Domain Constraints

The purpose of domain constraints are to ensure that tuple values in a relation state are within the domains of their corresponding attributes in the relation schema. Restricting the value for a given attribute to a specific data type and to a subset of values within that data type is an example of a domain constraint. Domain constraints are enforced by the database management system using a method of rejecting INSERT or UPDATE operations which cause tuples to have invalid values. It can also set those values to be NULL or “default”.

Entity Constraints

To ensure that all tuples that are part of a relation state have a primary key value that is not NULL, the entity constraint is used. This is important in ensuring that each tuple can be uniquely identified. By rejecting INSERT or UPDATE operations when the primary key value is not set, the database management system enforces the entity constraint.

Primary and Unique Key Constraints

In order to uniquely identify tuples, a “key” attribute is necessary within a tuple which is never the same for any two tuples in a relation state. The primary key constraint is used to ensure that there are never two tuples that have the same primary key attributes. Additionally, some attributes in a relation schema can have a uniqueness constraint even if they are not the primary key. The database management system will enforce these constraints by rejecting INSERT or UPDATE commands where the primary or unique key attributes of a tuple match another tuple in the relation state. By using a feature called auto-incrementing, the database management system can enforce this constraint during each INSERT operation if the attribute has an integer domain.

Referential Constraints

If a tuple in one relation state references a tuple in another relational state using a foreign key, the referenced tuple must exist. The referential constraint ensures that if relation R_a

contains a foreign key that references relation R_b and a Tuple T_a exists in a relation state of R_a, then the foreign key of T_a matches the primary key for a tuple T_b that exists in relation state of R_b.

The database management system enforces referential constraints for INSERT, UPDATE and DELETE operations.

For the INSERT operation, any new tuple that has an invalid foreign key value will be rejected or, if possible, the value is set to NULL or a default value.

For the DELETE operation, three outcomes can occur:

1. **Restrict**: The operation will be rejected.
2. **Cascade:** Delete all tuples that reference the deleted tuple through the foreign key
3. **Set Default**: Set the foreign key values for all other tuples that reference the one being deleted to NULL or a default value.

For the UPDATE operation, if the foreign key value is invalid then the operation will either be rejected or the foreign key value is set to NULL or a default value if able. When making changes to the primary key value of a tuple referenced by other tuples with the UPDATE operation, the **Restrict, Cascade** and **Set Default** options are also valid.

Check Constraints and Business Rules

Check Constraints and Business Rules ensure that data fits the user's expectations of how the business should run. These constraints are custom-written by the database designer and are enforced by code in applications implementing the database. There is no such way to directly express these constraints in the schemas of the data model.

2.3 Convert Entity Relationship Model to Relational Model

Since we have explained the process for converting between a conceptual to logical database, we will implement this process on the Kern Medical Database. All of the entity and relationship types will be converted into relational schema. Constraints must be created to preserve the validity of the relational database that fits the needs of the organization. Sample tuples will also be created for each relation to display how the relational database model will function in real life.

2.3.1 Relational Schema for Logical Database

Each relational schema in the logical design of the database will now be listed. Included will be all of the attributes as well as their type, where the relation came from regarding the E-R Model, and all of the constraints related to the database.

Relation Schema: Hospital Employee
Hospital Employee(**eID**, SSN, fName, mName, lName, street, city, state, zip, phone, licType, licNo, sDate, eDate)

eID*	integer, 0-maxID, primary key
SSN	000000000-999999999
fName	varchar2(20)
mName	varchar2(20)
lName	varchar2(25)
street	varchar2(25)
city	varchar2(25)
state	varchar2(2)
zip	integer, 00000-99999
phone	integer 0000000-9999999
licType	varchar(2) “MD”, “RN”, or “RD” representing Doctor, Nurse, or Dietitian

licNo	integer 00000-99999
sDate	Date (timestamp)
eDate	Date (timestamp)

Candidate Keys: eID (primary key) and SSN

Primary Key/Entity Integrity Constraint: eID is unique and cannot be NULL

Uniqueness Constraint: SSN must be unique

Business Constraint: None

Derivation from Entity and Relationship Types: Hospital Employee
This relational schema "employee" is derived from the Hospital Employee entity type. The way we differentiate between nurses, doctors, and dietitians is through the attribute licType. The attributes name and address are also broken into simple attributes.

Relational Schema: Patient
Patient(**<u>pID</u>**, SSN, fName, mName, lName, street, city, state, zip, phone, dob, sex, insType, language)

<u>pID</u>	integer, 0-maxID, primary key

SSN	000000000-999999999
fName	varchar2(20)
mName	varchar2(20)
lName	varchar2(25)
street	varchar2(25)
city	varchar2(25)
state	varchar2(2)
zip	integer, 00000-99999
phone	integer 0000000-9999999
dob	Date
sex	varchar2(10) “Male” or “Female”
insType	varchar2(50)
language	varchar2(20)

Candidate Keys: pID (primary key) and SSN

Primary Key/Entity Integrity Constraint: pID is unique and cannot be NULL

Uniqueness Constraint: SSN must be unique

Business Constraint: None

Derivation from Entity and Relationship Types: Patient

This relational schema is derived from the patient entity type. The only things that changed is the attributes name and address were broken into simple attributes.

Relational Schema: Care Team
Care Team(**ctID**, dept)

ctID	integer, 0-maxID, primary key

dept	varchar2(25)

Candidate Keys: ctID

Primary Key/Entity Integrity Constraint: ctID is unique and cannot be NULL

Uniqueness Constraint: None

Business Constraint: None

Derivation from Entity and Relationship Types: Care Team

This relational schema is derived from the care team entity type. Nothing was changed in converting this to a relation.

Relational Schema: Cases
Cases(**<u>cID</u>**, bpSys, bpDia, hRate, rRate, vDate, **pID, dID eID**)

<u>cID</u>	integer, 0-maxID, primary key
bpSys	integer, 70-200
bpDia	integer, 40-100
hRate	integer, 0-200

rRate	integer, 0-100
vDate	Date
pID	integer, 00000-99999
dID	integer, 00000-99999
eID	integer, 00000-99999

Candidate Keys: cID, (pID vData), (eID vDate)

Primary Key/Entity Integrity Constraint: cID is unique and cannot be NULL

Referential Integrity Constraint: pID is a foreign key for patient, dID is a foreign key for diagnosis, and eID is a foreign key for employee (Nurse).

Business Constraint: eID must be licType = ‘RN’

Derivation from Entity and Relationship Types: Case

This relational schema was derived from the case entity type. The N:1 relationships between patient & case, hospital employee (nurse) & case, and diagnosis & case are represented using the foreign key approach since participation is total.

Relational Schema: Diagnosis
Diagnosis(**dID**, diagnosis, status, **lID, eID**)

dID*	integer, 0-maxID, primary key
diagnosis	varchar2(50)
status	integer, 0-2*
lID*	integer, 0-maxID, foreign key
eID*	integer, 0-maxID, foreign key

***Status Key:**

0 - Awaiting Assessment
1 - Assessment Complete, Awaiting Diagnosis (Seen by Nurse)
2 - Diagnosis Complete, Awaiting Treatment (Seen by Nurse & Doctor)

Candidate Keys: dID

Primary Key/Entity Integrity Constraint: dID is unique and cannot be NULL

Referential Integrity Constraint: lID is a foreign key for labs and eID is a foreign key for employee.

Business Constraint: None, eID must be licType = 'MD'

Derivation from Entity and Relationship Types: Diagnosis

This relational schema was derived from the diagnosis entity type. The N:1 relationships between hospital employee (doctor) & diagnosis are represented using the foreign key approach since participation is total. Since the relationship between labs and diagnosis is 1:1 we can use the foreign key approach as well.

Relational Schema: Prescription
Prescription(**<u>prID</u>**, medication, dosage, frequency, sDate, eDate)

<u>prID</u>	integer, 0-maxID, primary key
medication	varchar2(50)
dosage	float
frequency	varchar2(20)

Candidate Keys: pID
Primary Key/Entity Integrity Constraint: prID is unique and cannot be NULL

Uniqueness Constraint: None

Business Constraint: None

Derivation from Entity and Relationship Types: Prescription

This relational schema is derived from the prescription entity type. Nothing was changed in changing this entity into a relation.

Relational Schema: Labs
Labs(<u>**lID**</u>, BUN, calcum, c02, chloride, creatinine, glucose, potassium, sodium)

<u>**lID**</u>	integer, 0-maxID, primary key
BUN	integer
calcium	float
c02	integer
chloride	integer
creatinine	float
glucose	integer

potassium	float
sodium	integer

Candidate Keys: lID

Primary Key/Entity Integrity Constraint: pID is unique and cannot be NULL

Uniqueness Constraint: None

Business Constraint: None

Derivation from Entity and Relationship Types: Labs

This relational schema is derived from the labs entity type. Nothing was changed in changing this entity to a relation.

Relational Schema: Forms
Forms(**eID, ctID**, sdate)

eID	integer, 00000-99999, foreign key
ctID	integer, 00000-99999, foreign key
sDate	Date

Candidate Keys: (eID, ctID, and sDate), (eID, ctID, and sDate)

Primary Key/Entity Integrity Constraint: The combination of eID, ctID, and sDate must be unique and cannot be NULL

Referential Integrity Constraints: eID is a foreign key for employee and ctID is a foreign key for care team.

Business Constraint: sDate cannot be greater than eDate and eID must be licType = ‘MD’ or ‘RD’

Derivation from Entity and Relationship Types: forms

This relational schema was derived from the forms relationship. The relationship ‘forms” between care team and employees is M:N therefore we must use the cross reference approach. This relation is a “relationship relation.

Relational Schema: prescribes
prescribes(**prID, dID**, date)

prID	integer, 00000-99999, foreign key
dID	integer, 00000-99999, foreign key
sDate	Date
eDate	Date

Candidate Keys: (prescID and date) , (dID and date)

Primary Key/Entity Integrity Constraint: The combination of prescID & date and dID & date must be unique and cannot be NULL

Referential Integrity Constraints: prescID is a foreign key for prescriptions and dID is a foreign key for diagnosis

Business Constraint: sDate cannot be larger than eDate

Derivation from Entity and Relationship Types: Prescription and Diagnosis

This relational schema was formed from the “prescribes” relationship using the cross reference approach with prescription and diagnosis. This relation is a “relationship relation”

Relational Schema: seen by
seen by(**pID, ctID**, date)

pID	integer, 00000-99999, foreign key
ctID	integer, 00000-99999, foreign key
date	Date

Candidate Keys: (pID and date) , (ctID and date)

Primary Key/Entity Integrity Constraint: The combinations of pID & date and ctID & date must be unique and cannot be NULL

Referential Integrity Constraints: pID is a foreign key for patient and ctID is a foreign key for care team.

Business Constraint: None

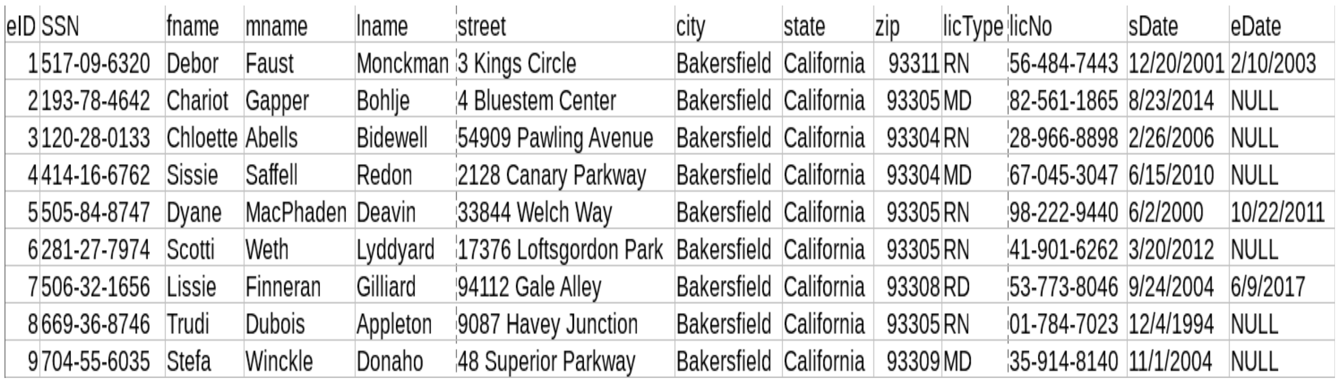
Derivation from Entity and Relationship Types: seen by

This relational schema was derived from the “seen by” relationship using the cross reference approach with patient and care team. This relation is a “relationship relation.”

2.3.2 Sample Data of Relation

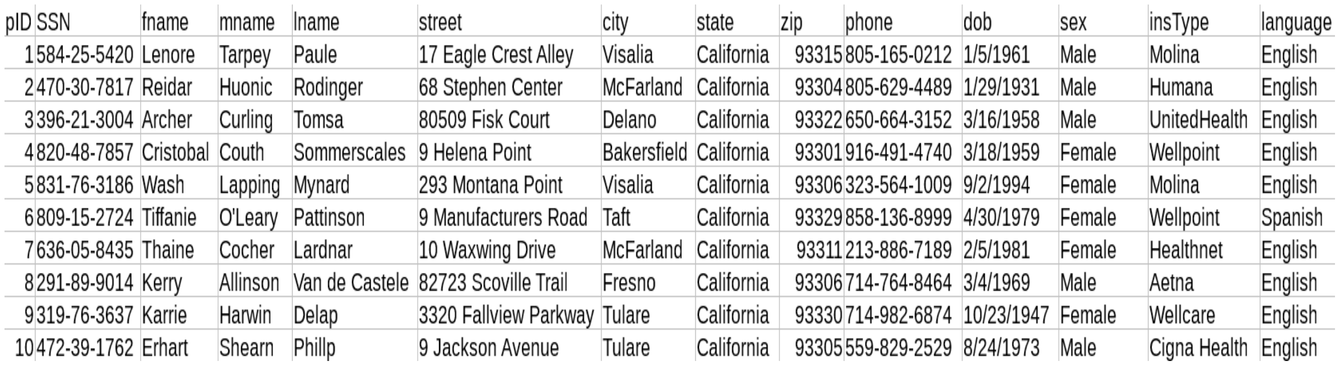
Since we have now described each Relation Schema for our Relational Model, we will display a list of Tuples that are to belong to hypothetical Relation States for each Relation Schema in our database. The Tuples will be listed in a table format in which attributes are columns and each individual Tuple is a row. Relations corresponding to entity sets will be displayed with roughly 10 tuples while relations that correspond to Relationship Sets will be displayed with between 60 to 100 tuples.

<u>**Hospital Employee**</u>



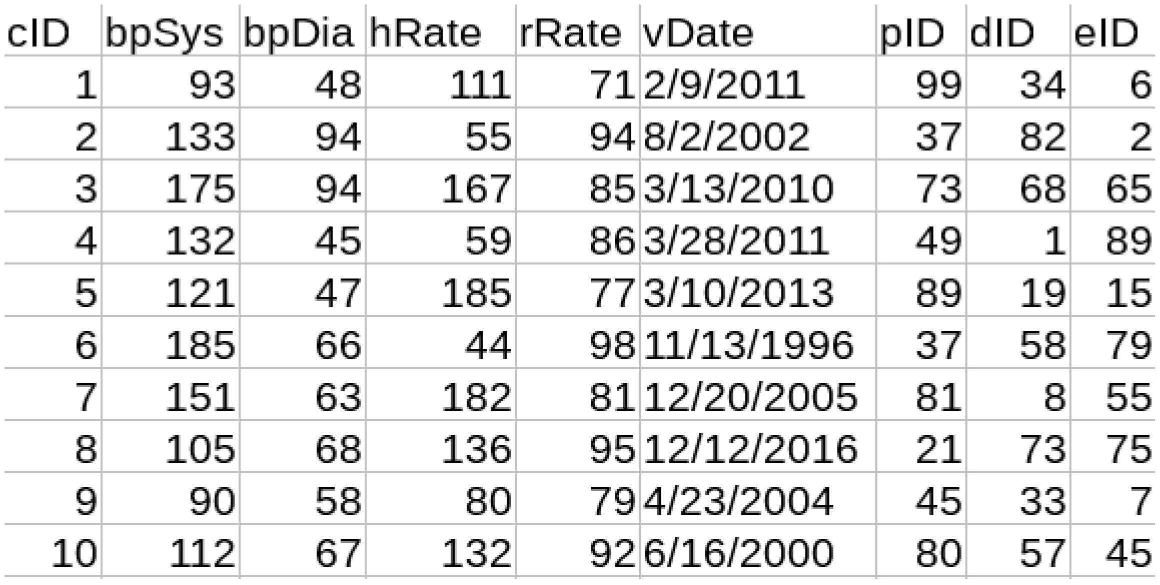
eID	SSN	fname	mname	lname	street	city	state	zip	licType	licNo	sDate	eDate
1	517-09-6320	Debor	Faust	Monckman	3 Kings Circle	Bakersfield	California	93311	RN	56-484-7443	12/20/2001	2/10/2003
2	193-78-4642	Chariot	Gapper	Bohlje	4 Bluestem Center	Bakersfield	California	93305	MD	82-561-1865	8/23/2014	NULL
3	120-28-0133	Chloette	Abells	Bidewell	54909 Pawling Avenue	Bakersfield	California	93304	RN	28-966-8898	2/26/2006	NULL
4	414-16-6762	Sissie	Saffell	Redon	2128 Canary Parkway	Bakersfield	California	93304	MD	67-045-3047	6/15/2010	NULL
5	505-84-8747	Dyane	MacPhaden	Deavin	33844 Welch Way	Bakersfield	California	93305	RN	98-222-9440	6/2/2000	10/22/2011
6	281-27-7974	Scotti	Weth	Lyddyard	17376 Loftsgordon Park	Bakersfield	California	93305	RN	41-901-6262	3/20/2012	NULL
7	506-32-1656	Lissie	Finneran	Gilliard	94112 Gale Alley	Bakersfield	California	93308	RD	53-773-8046	9/24/2004	6/9/2017
8	669-36-8746	Trudi	Dubois	Appleton	9087 Havey Junction	Bakersfield	California	93305	RN	01-784-7023	12/4/1994	NULL
9	704-55-6035	Stefa	Winckle	Donaho	48 Superior Parkway	Bakersfield	California	93309	MD	35-914-8140	11/1/2004	NULL

Patient



pID	SSN	fname	mname	lname	street	city	state	zip	phone	dob	sex	insType	language
1	584-25-5420	Lenore	Tarpey	Paule	17 Eagle Crest Alley	Visalia	California	93315	805-165-0212	1/5/1961	Male	Molina	English
2	470-30-7817	Reidar	Huonic	Rodinger	68 Stephen Center	McFarland	California	93304	805-629-4489	1/29/1931	Male	Humana	English
3	396-21-3004	Archer	Curling	Tomsa	80509 Fisk Court	Delano	California	93322	650-664-3152	3/16/1958	Male	UnitedHealth	English
4	820-48-7857	Cristobal	Couth	Sommerscales	9 Helena Point	Bakersfield	California	93301	916-491-4740	3/18/1959	Female	Wellpoint	English
5	831-76-3186	Wash	Lapping	Mynard	293 Montana Point	Visalia	California	93306	323-564-1009	9/2/1994	Female	Molina	English
6	809-15-2724	Tiffanie	O'Leary	Pattinson	9 Manufacturers Road	Taft	California	93329	858-136-8999	4/30/1979	Female	Wellpoint	Spanish
7	636-05-8435	Thaine	Cocher	Lardnar	10 Waxwing Drive	McFarland	California	93311	213-886-7189	2/5/1981	Female	Healthnet	English
8	291-89-9014	Kerry	Allinson	Van de Castele	82723 Scoville Trail	Fresno	California	93306	714-764-8464	3/4/1969	Male	Aetna	English
9	319-76-3637	Karrie	Harwin	Delap	3320 Fallview Parkway	Tulare	California	93330	714-982-6874	10/23/1947	Female	Wellcare	English
10	472-39-1762	Erhart	Shearn	Phillp	9 Jackson Avenue	Tulare	California	93305	559-829-2529	8/24/1973	Male	Cigna Health	English

Case



cID	bpSys	bpDia	hRate	rRate	vDate	pID	dID	eID
1	93	48	111	71	2/9/2011	99	34	6
2	133	94	55	94	8/2/2002	37	82	2
3	175	94	167	85	3/13/2010	73	68	65
4	132	45	59	86	3/28/2011	49	1	89
5	121	47	185	77	3/10/2013	89	19	15
6	185	66	44	98	11/13/1996	37	58	79
7	151	63	182	81	12/20/2005	81	8	55
8	105	68	136	95	12/12/2016	21	73	75
9	90	58	80	79	4/23/2004	45	33	7
10	112	67	132	92	6/16/2000	80	57	45

Care Team

ctID	dept
1	Orthopedics
2	Family Practice
3	Trauma
4	Psychiatry
5	Infectious Disease
6	Psychiatry
7	General Surgery
8	Family Practice
9	Hemotology/Oncology
10	Internal Medicine

<u>Diagnosis</u>

dID	diagnosis	status	lID	eID
1	Hepatitis A; GI Bleed	1	73	45
2	MVA with Multiple Fractures; Subdural Hemotoma	0	3	7
3	Stroke	1	90	17
4	Flu; Intractable Nausea & Vomiting	2	11	70
5	GI Bleed	2	70	59
6	Hepatitis B	1	58	40
7	Intractable Nausea & Vomiting	2	1	90
8	HIV; Failure to Thrive	1	0	64
9	Cholecystitis	0	11	35
10	Cystic Fibrosis	0	20	64

<u>Prescription</u>

prID	medication	dosage	frequency
1	Carvedilol Tab	3.125	BID with meals
2	Vancomycin Injection	1	Q 12 hrs infuse over 90 minutes
3	Acetaminophen	650	Q 6 hr
4	Ondansetron Injection	4	Q 6 hr
5	Heparin Injection	0.5	Q 8 hr
6	Lisinopril	5	Daily
7	Furosemide	4	Daily
8	Hydrocodone	325	Q 6 hr
9	Oxycodone	10	Q 6 hr
10	Acetaminophen	500	Q 6 hr

<u>Labs</u>

lID	BUN	calcium	c02	chloride	creatinine	glucose	potassium	sodium
1	6	8.1	35	91	1.2	99	3.8	150
2	12	10.5	22	106	0.72	74	3.5	142
3	28	7.1	16	103	1.2	88	5.1	144
4	7	8.8	26	100	1.49	102	3.4	135
5	11	7.2	20	103	1.35	93	4.4	137
6	14	8.6	28	90	1.17	75	3.5	149
7	25	10.7	27	93	1.4	88	3	144
8	15	9.3	25	99	0.86	70	3.9	146
9	16	8.5	27	94	1.41	91	3.4	134
10	23	10.5	22	103	1.21	102	4.5	139

Forms

eID	ctID	sDate
2	1	10/16/14
4	1	10/16/14
7	1	10/16/14
2	2	10/17/14
7	2	10/17/14
2	3	11/21/14
4	3	11/21/14
7	3	11/21/14
13	3	11/21/11
2	4	02/25/14
13	4	02/25/14
7	4	02/25/14
18	5	01/05/13
24	5	01/05/13
16	5	01/05/13
16	6	01/06/13
25	6	01/06/13
29	6	01/06/13
16	7	03/05/17
7	7	03/05/17
2	8	03/10/14
7	8	03/10/14
16	8	03/10/14
29	9	08/16/16
32	9	08/16/16
7	9	08/16/16
16	10	05/11/99
23	10	05/11/99
12	11	05/09/14
14	12	07/04/12
7	12	07/04/12
9	12	07/04/12
16	13	02/02/05
7	13	02/02/05
9	13	02/02/05
9	14	11/01/08
7	14	11/01/08
13	15	11/01/12
18	15	11/01/12
16	16	05/18/17
7	16	05/18/17
4	16	05/18/17
2	17	12/01/14
4	17	12/01/14
7	17	12/01/14
2	18	04/04/15
4	18	04/04/15
2	19	05/04/15
4	19	05/04/15
7	19	05/04/15
16	20	11/10/05
7	20	11/10/05
9	20	11/10/05
18	20	11/10/05
2	21	02/01/16
7	21	02/01/16
4	21	02/01/16
2	22	08/16/15
4	23	08/16/15
7	23	08/16/15

Seen By

pID	ctID	date
77	75	2/28/2009
48	26	10/23/1998
77	6	1/12/2000
58	27	12/8/2016
77	96	12/27/2000
33	99	5/14/2006
6	84	5/20/2014
36	94	9/16/2001
2	19	6/9/2017
28	15	7/6/2008
11	51	10/13/2000
88	8	11/24/2004
95	1	4/25/2015
31	27	11/26/1995
38	10	2/12/1999
22	63	6/13/2002
11	70	1/20/1996
94	27	7/13/2014
27	89	7/5/1996
93	56	4/1/2014
43	20	3/20/2008
88	9	8/2/2008
26	35	9/9/2009
18	79	6/6/2005
34	82	4/14/1996
50	53	7/5/2012
58	16	3/20/2001
65	38	9/26/2002
59	26	7/26/2014
7	18	10/8/2001
7	18	10/8/2001
37	15	7/14/2006
27	74	7/20/2006
73	63	3/14/2004
48	12	4/7/2004
44	63	7/14/2008
66	77	12/21/2013
94	23	9/2/2016
84	9	1/3/2007
24	41	3/16/2017
27	68	10/8/1996
94	10	4/4/1999
74	78	8/17/2009
47	75	10/31/2014
100	52	12/2/2008
7	36	5/11/2003
19	47	8/31/1998
8	16	9/22/2001
10	60	6/15/2010
26	52	9/7/2012
50	9	6/29/1996
92	62	2/6/2000
82	7	11/12/2007
86	1	5/21/2015
43	22	1/15/2012
57	89	4/4/2000
64	12	4/9/2009
70	22	7/30/2010
24	56	1/10/2000
28	33	5/19/1997
1	57	6/16/2005

Prescribes

prID	dID	sDate	eDate
1	17	9/18/2003	6/6/2004
2	69	5/11/2006	9/13/2006
3	94	8/8/2000	12/11/2001
4	76	4/19/2008	6/30/2009
5	41	9/24/1999	7/7/2000
6	59	3/18/2011	11/15/2011
7	50	6/8/2010	4/17/2011
8	66	8/19/1996	12/1/1996
9	49	2/24/1997	8/16/1997
10	100	4/7/2007	5/28/2007
11	87	1/26/2004	5/15/2004
12	51	10/29/2002	10/29/2003
13	93	6/29/2002	7/31/2002
14	4	9/24/2014	10/3/2014
15	81	9/22/2014	3/7/2015
16	75	1/16/2000	4/2/2000
17	7	4/3/2013	4/8/2014
18	92	4/17/2012	3/9/2013
19	18	4/9/2000	2/15/2001
20	54	3/16/2007	8/16/2008
21	70	1/11/1997	1/12/2005
22	35	8/8/2000	7/22/2001
23	43	11/11/2015	11/5/2016
24	96	6/22/2009	10/22/2009
25	87	4/11/2013	2/7/2014
26	51	11/12/1998	9/10/1999
27	12	9/11/2005	3/22/2006
28	72	1/14/2014	11/29/2014
29	18	12/6/1997	3/2/1998
30	87	4/3/2011	8/20/2011
30	87	4/3/2011	8/20/2011
31	62	12/16/1998	3/27/1999
32	27	8/2/2012	10/10/2012
33	68	12/23/2010	9/14/2011
34	1	9/25/1996	11/27/1996
35	31	5/30/2011	4/19/2012
36	25	7/2/1997	11/18/1997
37	84	5/18/1999	8/14/1999
38	72	12/27/1998	6/25/1999
39	12	5/23/1996	9/14/1996
40	43	1/27/1996	7/26/1996
41	71	9/19/2016	5/13/2017
42	16	8/30/2014	3/18/2015
43	68	11/18/1997	10/14/1998
44	86	4/26/2002	7/11/2002
45	21	11/1/2012	7/5/2013
46	94	10/12/2003	10/26/2003
47	69	7/19/2008	3/14/2009
48	13	12/16/2011	6/21/2012
49	100	4/12/2012	6/10/2012
50	77	12/5/1998	11/11/1999
51	85	6/13/1999	10/12/1999
52	27	9/10/2010	3/2/2011
53	99	7/18/1996	5/5/1997
54	85	1/3/2014	10/4/2014
55	79	4/15/2010	10/10/2010
56	93	8/6/2017	7/18/2018
57	32	5/6/2000	5/20/2000
58	63	3/26/2014	8/29/2014
59	48	6/19/1996	11/16/1996
60	88	2/10/2003	1/28/2004

2.4 Sample Queries

Now that we have specified all of the data in our database, we will need to show methods of retrieving this data. For this we will be using sample queries designed to search for specific data throughout our database. All of the queries will be expressed in three different types of mathematical expressions.

2.4.1 Design of Queries

The design of these queries will be in three different formal querying languages. These languages are: relational algebra, domain relational calculus, and tuple relational calculus. These sample queries will be expressed using all three of these formal querying languages.

2.4.2 Relational Algebra Expressions for Queries

Relational algebra combines operation sets to retrieve tuples from a relational database. Being procedural, it is important that relational algebra expressions are nested properly. Here are our sample queries in relational algebra format:

1. **List all patients who were at the hospital in 1999 but did not get prescribed any medication**

pr d p c

Result $\leftarrow \pi_{P.*¿¿¿}$

pr.dID = d.dID **d.dID = c.dID** **p.pID = c.pID ^ vDate $\geq$ 01-01-99**

^ vDate $\leq$ 12-31-99

p **c**

π ❑$_{P.*¿¿¿}$ $-$¿ Result)

p.pID = c.pID ^ vDate $\geq$ 01-01-99

^ vDate $\leq$ 12-31-99

2. **List all nurses who have seen at least 3 cases but no longer work at the hospital**

c3 **c2** **c1** **he**

$\pi_{he.*¿¿¿}$

c3.cID != c2.cID $\wedge$ **c2.cID != c.cID $\wedge$** **licType = 'RN' $\wedge$ eDate != NULL**

c3.cID != c.cID $\wedge$ **c2.eID = he.eID** **$\wedge$ c.eID = he.eID**

c3.eID = he.eID

3. **List the patient with the lowest heart rate recorded in the hospital.**

P **P2** **C2** **P1** **C1**

$\pi_{P.*¿¿¿}\sigma\ Patient)-¿¿(\sigma\ Patient \times Case)¿$

P2.pID = C2.pID $\wedge$ **P1.pID = C2.pID**

(P1.pID != P2.pID $\vee$

C1.cID != C2.cID) $\wedge$ C1.hr < C2.hr

4. **List the youngest patient who has been to the hospital.**

p3 c2 p2 p c

$\pi_{P3.*¿¿¿}\sigma\ Patient \times Case) - ¿(\sigma\ Patient \times (\sigma\ Patient \times Case))$

p3.pID = c2.pID p2.pID != p.pID p.pID = c.pID

$\wedge$ p2.dob ¿ p.dob

5. **List all patients with 2 or more current prescriptions in a single Diagnosis.**

pr2 pr d p c

$\pi_{p.*}¿$

pr2.pID !- pr.pID $\wedge$ pr.dID = d.dID $\wedge$ d.dID = c.dID p.pID = c.pID

pr2.dID = d.dID pr.eDate $\geq$ Today

6. **List all patients who have been to the hospital exactly once.**

p2 c2 c p

$\pi_{p2.*}$ ¿

c2.pID = p.pID $\wedge$ c.pID = p.pID

c2.cID != c.cID

7. **List all patient who have received a diagnosis by every doctor currently employed at the hospital.**

p c d he

Result1 $\leftarrow \pi_{p.pID, he.eID}¿$

p.pID = c.pID c.dID = d.dID d.eID = he.eID $\wedge$ licType = 'MD' $\wedge$ status $\geq$ 2

$\wedge$ eDate = NULL

he

Result2 $\leftarrow \pi_{he.eID}(\sigma\ H.Employee)$

licType = 'MD'

P

π❑$_{P.*¿¿¿} \times Patient)$

8. **List all patients who have been seen by a doctor and a nurse but do not have any active prescriptions.**

d p c

Result1 $\leftarrow (\sigma\ Diagnosis \times (\sigma\ Patient \times Case)¿$

d.dID = c.dID p.pID = c.pID

$\wedge$ status = 2

p d p c

$$\text{Result2} \leftarrow \pi_{p.*}\ \text{¿}(\sigma\ Diagnosis \times (\sigma\ Patient \times Case)\text{¿¿}$$

pr.dID = d.dID d.dID = c.dID p.pID = c.pID

$\wedge$ **pr.sDate $\leq$ today** $\wedge$ **status = 2**

$$\text{Result3} \leftarrow \pi_{p.*}(\text{Result1 - Result2})$$

9. List all male patients over the age of 50 with calcium levels over 9.0 (very high).

l d p c

$$\pi_{p.*\text{¿¿¿}}\sigma\ Diagnosis) \times (\sigma\ Patient \times Case)\text{¿¿}$$

l.lID = d.lID d.dID = c.dID p.pID = c.pID $\wedge$ **p.sex = 'Male'** $\wedge$ **p.dob $\geq$ 50 yrs**

$\wedge$ **calcium ¿ 9.0**

10.List all patients who were only seen by a nurse but not a doctor.

D P C

$$\pi_{P.*\text{¿¿¿}}\sigma\ Diagnosis) \times (\sigma\ Patient \times Case)\text{¿}$$

D.dID = c.dID $\wedge$ **P.pID = C.pID**

status = 1

***Status Key:**
0 - Awaiting Assessment
1 - Assessment Complete, Awaiting Diagnosis (Seen by Nurse)
2 - Diagnosis Complete (Seen by Nurse & Doctor)

2.4.3 Tuple Relational Calculus Expressions for Queries

Relational calculus is a querying language that uses declarative, nonprocedural expressions. A calculus expression specifies *what is* to be retrieved rather than *how* to retrieve it. The order in which the tuples will be retrieved is not specified in relational calculus. Two types of variables are used in relational calculus, free variables and bound variables. The free variable describes what the query will retrieve while bound variables are "bound" by existential and universal quantifiers. There are two types of relational calculus, tuple relational calculus and domain relational calculus. The tuple relational calculus is based on specifying a number of tuple variables where a tuple represents a list of values that represent a list of values that satisfy the attribute names of the relational schema. The following queries are written in tuple relational calculus

1. **List all patients who were at the hospital in 1999 but did not get prescribed any medication**

{p | patient(p) ∧(∃ *c*¿(case(c) ∧p.pID = c.pid ∧c.vDate ≤12/31/1999 ∧c.vDate ≥ 1/1/1999 ∧(
∃ *d*) (diagnosis (d) ∧d.dID = c.dID ∧
¬(∃ *pr*¿(prescribes(pr) ∧pr.dID = d.dID)))}

2. **List all nurses who have seen at least 3 cases but no longer work at the hospital**

{h | hospital employee (h) ∧h.licType = 'RN' ∧h.eDate != NULL ∧
(∃ *c*) (case (c) ∧c.eID = h.eID ∧
(∃ *c*2) (case (c2) ∧c2.eID = h.eID ∧c.cID != c2.cID ∧
(∃ *c*3) (case (c3) ∧c3.eID = h.eID ∧c3.cID != c2.cID ∧c3.cID != c.cID)))}

3. **List the patient with the lowest heart rate recorded in the hospital.**

{p | patient(p) ∧
(∃ *c*) (case (c) ∧c.pID = p.pID ∧
¬(∃ *c*2) (case (c2) ∧c2.pID = p.pID ∧c2.cID != c.cID ∧c2.hr < c.hr))}

4. **List the youngest patient who has been to the hospital.**

{p | patient(p) ∧
(∀ *p*2) (patient(p2) ∧p.pID != p2.pID → p.dob < p2.dob)}

5. **List all patients with 2 or more current prescriptions in a single Diagnosis.**

{p | patient(p) ∧
(∃ *c*) (case (c) ∧
(∃ *d*) (diagnosis (d) ∧c.dID = d.dID ∧
(∃ *pr*) (prescribes(pr) ∧pr.dID = d.dID ∧pr.eDate ≥Today ∧
(∃ *pr* 2) (prescribes(pr2) ∧pr2.dID = d.dID ∧pr2.prID != pr.prID ∧pr2.eDate ≥Today))))}

6. **List all patients who have been to the hospital exactly once.**

{p | patient(p) ∧
(∃ *c*) (case (c) ∧p.pID = c.pID ∧
¬(∃ *c*2) (case(c2) ∧c.dID = d.dID ∧c2.cID != c.cID ∧c2.pID =p.pID))}

7. **List all patients who have had a diagnosis made by all doctors currently employed at the hospital.**

$\{p \mid \text{patient}(p) \wedge$
$(\forall\ h)\ (\text{hospital employee}\ (h) \wedge h.\text{eDate} \mathrel{!=} \text{NULL} \rightarrow$
$(\exists\ d)(\exists\ c)\ (\text{diagnosis}(d) \wedge \text{case}(c) \wedge h.\text{eID} = d.\text{eID} \wedge p.\text{pID} = c.\text{pID} \wedge c.\text{dID} = d.\text{dID}))\}$

8. **List all patients who have been seen by a doctor and a nurse but do not have any active prescriptions.**

$\{p \mid \text{patient}(p) \wedge (\exists\ c\text{¿}(\text{case}(c) \wedge p.\text{pID} = c.\text{pid} \wedge$
$(\exists\ d)\ (\text{diagnosis}\ (d) \wedge d.\text{dID} = c.\text{dID} \wedge \text{status} = 2 \wedge$
$\neg(\exists\ pr\text{¿}(\text{prescribes}(pr) \wedge pr.\text{dID} = d.\text{dID})))\}$

9. **List all male patients over the age of 50 with calcium levels over 9.0 (very high).**

$\{p \mid \text{patient}(p) \wedge p.\text{dob} \geq 50\ \text{yrs} \wedge p.\text{sex} = \text{'Male'}$
$(\exists\ c)\ (\text{case}\ (c) \wedge p.\text{pID} = c.\text{pID} \wedge$
$(\exists\ d)\ (\text{diagnosis}(d) \wedge c.\text{dID} = d.\text{dID})$
$(\exists\ l)\ (\text{labs}(l) \wedge l.\text{lID} = d.\text{lID} \wedge \text{calcium}\ \text{¿}9.0)\}$

10. **List all patients who were only seen by a nurse but not a doctor.**

$\{p \mid \text{patient}(p) \wedge$
$(\exists\ c)\ (\text{case}\ (c) \wedge p.\text{pID} = c.\text{pID} \wedge$
$(\exists\ d)\ (\text{diagnosis}(d) \wedge c.\text{dID} = d.\text{dID} \wedge d.\text{status} = 1))\}$

2.4.4 Domain Relational Calculus Expressions for Queries

There is a second variation of relational calculus which is called domain relational calculus or simply domain calculus. Domain calculus varies from tuple relational calculus in the types of variables that are used in the formulas. Rather than having variables range over entire tuples, the variables range over single values from domains of attributes. The following queries are done in domain relational calculus.

1. **List all patients who were at the hospital in 1999 but did not get prescribed any medication**

$\{<i, f, l> \mid \text{patient}\ (i,\_,f,\_,l,\_,\_,\_,\_,\_,\_,\_,\_,\_) \wedge (\exists\ d\text{¿}(\exists\ dt)(\text{case}(\_,\_,\_,\_,\_,dt,i,d,\_)$
$\wedge\ dt \geq 1/1/1999 \wedge dt \leq 12/31/1999 \wedge (\text{diagnosis}(d,\_,\_,\_,\_,\_)$
$\wedge \neg(\exists\ pr)(\text{prescribes}(pr,d,\_,\_))))\}$

2. **List all nurses who have seen at least 3 cases but no longer work at the hospital**

{<i, f, l> | Hospital Employee(i,f,_,l_,_,_,_,_,’RN’,_,_, != NULL) ∧ (∃ *c*)(∃ *c*2)(∃ *c*3)
(Case(c,_,_,_,_,_,_,_,i) ∧ Case(c2,_,_,_,_,_,_,_,i) ∧ Case(c3,_,_,_,_,_,_,_,i)
)}

3. **List the patient with the lowest heart rate recorded in the hospital.**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧(∃ *hr*)(∃ *c*)(Case(c,_,_,hr,_,_,i,_,_) ∧(∀ *hr* 2)
Case(!=c,_,_,hr2,_,_i,_,_,_) → hr < hr2)}

4. **List the youngest patient who has been to the hospital.**

{<i, f, l, a> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧(∃ *a*)(Patient(i,_,_,_,_,_,_,_,_,_,_,a,_,_,_)
∧ ¬Patient(_,_,_,_,_,_,_,_,_,_,_,<a,_,_,_))}

5. **List all patients with 2 or more current prescriptions in a single Diagnosis.**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧(∃ *d*)diagnosis(d,_,_,_,_,_)
∧(case(_,_,_,_,_,_,i,d,_) ∧prescribes(_,d,_,≥Today)∧prescribes(_,d,_,≥Today))}

6. **List all patients who have been to the hospital exactly once.**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧(∃ *c*)(Case(c,_,_,_,_,_,i,_,_) ∧
¬Case(_,_,_,_,_,_,i,_,_))}

7. **List all patients who have had a diagnosis made by all doctors currently employed at the hospital.**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧
(∀ *h*)(H.Employee(h,_,_,_,_,_,_,_,_,_,_,_,!=NULL) →
(∃ *d*)(diagnosis(d,_,_,_,e) ∧Case(_,_,_,_,_,_,_,d,_)))}

8. **List all patients who have been seen by a doctor and a nurse but do not have any active prescriptions.**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧(∃ *d*¿(diagnosis(d,_,2,_,_,_)
∧(case(_,_,_,_,_,_,i,d,_)
∧ ¬(∃ *pr*)(prescribes(pr,d,_,_))))}

9. **List all male patients over the age of 50 with calcium levels over 9.0 (very high).**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,≥50 yrs,'Male',_,_) ∧(∃ *c*)(∃ *d*¿(∃ *l*¿
(case(c,_,_,_,_,_,i,d,_) ∧diagnosis(d,_,_,l,_,_)∧labs(l,_,¿9.0,_,_,_,_,_,_))}

10. **List all patients who were only seen by a nurse but not a doctor.**

{<i, f, l> | patient (i,_,f,_,l,_,_,_,_,_,_,_,_,_) ∧(∃ *c*)(∃ *d*)(case(c,_,_,_,_,_,i,d,_)∧
diagnosis(d,_,'1',_,_,_))}

3 PostgreSQL

3.1 Normalization of Relations

In order to ensure a database schema is well design, formal method called normalization is used. This is used to determine the quality of our relational database schema design. In this section we will discuss some of the problems that can occur when applying operations to a poorly implemented database design. Following this will we analyze our own relation schemas for the Kern Medical database.

3.1.1 Normalization and Normal Forms

The quality of the relational schemas as well as the concepts of normalization are key aspects of Kern Medical Database. Next we will define the concepts of normalization and how to measure normalization using the three normal forms tests (NF).

Description of Normalization and Normal Forms

A relational database schema is said to be poorly designed if the tuples contain repetitive data. This can cause problems because when the redundant attribute value is changed in one tuple, it will need to be changed in other tuple(s) if the data is to remain consistent and coherent. This phenomenon is known as a modification anomaly. The decomposing or "breaking up" of relation schemas is known as Normalization. This is done to remove redundancy as well as preventing modification anomalies.

A formal method called the *normal forms test* will be used to test how each relational schema will be normalized.A relation schema can belong to four different main normal forms which measure normalization: first, second, third, and Boyce-Codd. These normal forms are also a useful ranking system for the relation schemas - the higher the normal form, the more normalized the schema will be. All schemas of a well designed database should satisfy at least the third normal form.

First Normal Form (1NF)

The key to the first normal form (1NF) being satisfied is that all attributes of the relation schema have to be single and atomic. The 1NF does not allow nested tuples of values within the relation tuples. There is multiple methods in which a relation schema that does not satisfy 1NF can be decomposed or broken up. One method is to create a new relation for the multi-valued attribute that contains the original relation's primary key as a foreign key. A second method is to create a new single-value attribute is added to the relation for each value in the multi-valued attribute, this can only be done is there is a specific number of values for each tuple. The third and final method is to replace the multi-valued attribute with a new single-value attribute and then store each value from the multi-valued attribute in a separate or "duplicate" tuple. The third solution will create redundant data so it should be avoided if possible.

Second Normal Form (2NF)

The key to the second normal form (2NF) and what will be evaluated is functional dependencies. Functional dependency is when the attributes X of a set only map to one set of values in a second set Y. In other words, the values of X can be used to determine what the values of Y will be. The primary key of a tuple is an example of functional dependency because each primary key is mapped to only one corresponding tuple.

$$X \rightarrow Y$$

Y is functionally dependent of X

The satisfaction of 1NF is necessary for a relation schema to satisfy 2NF. All attributes of the relation schema that are not part of the primary key must *fully* functionally depend on that corresponding primary key. Attributes that are not part of the primary key must functionally depend on the entire primary key, not just part of it. Any relational schema that have a single-attribute primary key will automatically pass the

2NF test. When a relational schema fails the 2NF test, normalization can be achieved by breaking the it into smaller schemas with primary keys that are subsets of the original relation’s primary key.

Third Normal Form (3NF)

In order for a relation schema to satisfy the third normal form (3NF), it is imperative that it satisfy both 1NF and 2NF. In addition, non-prime attributes cannot functionally depend on other non-prime attributes. In other words, non-prime attributes can not *transitively* depend on the primary key.

If the 3NF test results in a failure, normalization can be achieved by decomposing the relation into smaller relations where the left side of a functional dependency is always a primary key attribute. It can also be a superkey which contains the primary key.

Boyce-Codd Normal Form (BCNF)

The Boyce-Codd Normal Form (BCNF) is very similar to 3NF in that all previous normal forms must be satisfied first, only it is slightly stricter. In BCNF the left side of any functional dependencies must also be a primary key (or superkey similar to 3NF) of the relation schema. The way that BCNF is stricter than 3NF is that it does not allow any prime attributes to depend on non-prime attributes, 3NF allows this.

When a relation schema fails the BCNF test, it can be normalized by decomposing the schema into relations in which non-prime attributes at the left side of any functional dependencies will now be prime attributes in the new relation schema.

Anomalies that Result from Poor Normalization

At minimum, 3NF or BCNF must be satisfied. If this is not the case, relations will contain redundant data that can cause anomalies to occur when the data is modified. There are three classes of anomalies: insertion, modification, and deletion.

Insertion:

Redundant data which violates 2NF, 3NF, or BCNF tests can be created by storing the natural join of two relations as a single relation. Many anomalies may occur when trying to insert new tuples for the “joined” relation schema.

In order to insert two tuples that represent two relations that are both joined to the same relation, the attribute values for the joined relation must be exactly the same for both tuples for the data to remain coherent.

To insert a tuple representing a relation that is not joined to any other relation, the attribute values for the other relation schema must be all set to NULL. Since NULL values have different interpretations (including “not applicable”, “unknown”, and an absent value), this can create problems. Additionally, if any of the attributes that are set to NULL help compose the primary key of the joined relation, then the entity integrity constraint will be violated.

Update:

If a set of tuples that can represent one single relation is joined to several other relations then the attributed values representing the single relation will appear in all of the tuples. For the data to remain coherent, one must update all tuples instead of just one for the data to remain coherent.

Deletion:

If a set of tuples that represent a single relation is joined to multiple other relations and all of the tuples are removed, then any record of a single relation will be completely removed from the database. This means that it will be impossible to join the single relation to other relations in the future.

3.1.2 Normal Forms for this Database

We will now analyze our own relation schemas from the Kern Medical database design to determine if they satisfy, at minimum, 3NF. We will then analyze a relation schema resulting from the joining of two relation schemas and detail the anomalies that can result from this.

Employee

Functional Dependencies

FD1. {**eID**} → {SSN, fName, …, eDate}

FD2. {SSN} → {**eID,** fName, …, eDate}

Candidate Keys

eID

SSN

Normal Form

1NF is satisfied; all attributes have atomic domains.

2NF is satisfied; the primary key has only one attribute.

3NF is satisfied; no non-prime attributes depend on other non-prime attributes.

BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Patient

Functional Dependencies
FD1. {**pID**} → {SSN, fName, …, eDate}
FD2. {SSN} → {**pID,** fName, …, eDate}

Candidate Keys
pID
SSN

Normal Form
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Cases

Functional Dependencies
FD1. {**cID**} → {bpSys, bpDia, …, eID}

Candidate Keys
cID

Normal Form
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Diagnosis

Functional Dependencies
FD1. {**dID**} → {diagnosis, status, lID, eID}

Candidate Keys
dID

Normal Form
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.

3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Labs

Functional Dependencies

FD1. {**lID**} → {BUN, calcium, …, sodium}

Candidate Keys

lID

Normal Form

1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Prescription

Functional Dependencies

FD1. {**preID**} → {medication, dosage, frequency}

Candidate Keys

preID

Normal Form

1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

CareTeam

Functional Dependencies

FD1. {**ctID**} → {dept}

Candidate Keys

ctID

Normal Form
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

SeenBy

Functional Dependencies
FD1. {**pID**} → {**ctID**, date}
FD2. {**ctID**} → {**pID**, date}

Candidate Keys
pID
ctID

Normal Form
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Prescribes

Functional Dependencies
FD1. {**preID**} → {**dID**, sDate, dDate}
FD2. {**dID**} → {**preID**, sDate, eDate}

Candidate Keys
preID
dID

Normal Form
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

Form

<u>Functional Dependencies</u>
FD1. {**eID**} → {**ctID**, sDate}
FD2. {**ctID**} → {**eID**, sDate}

<u>Candidate Keys</u>
eID
ctID

<u>Normal Form</u>
1NF is satisfied; all attributes have atomic domains.
2NF is satisfied; the primary key has only one attribute.
3NF is satisfied; no non-prime attributes depend on other non-prime attributes.
BCNF is satisfied; the left side of all functional dependencies are candidate keys.

Therefore No inherent modification anomalies.

3.2 Psql: Main Purpose and Functionality

As we have completed checking the quality of the relational design, we will now describe the process for physical implementation. The physical database will be implemented using psql, in which we will also fill it with our sample data. psql is the command-line user interface which interacts with postgreSQL. psql is used by database administrators to quickly define and easily maintain a database. psql allows one to input SQL commands to manage and define schema objects, manipulate and query data, and control the output formatting. It also allows users to create and run scripts that execute those same type of commands at one time.

3.3 Schema Objects for postgreSQL

Sequences

Sequences are created by using a mathematical function that produces a set of unique values. Each time the sequence generator is called upon, it responds with the next number of the sequence. This is primarily used to generate primary key attributes ensuring that unique values are sued for new tuples being inserted by multiple users simultaneously. Sequence generators also have a caching option which allows it to pre-calculate and store the next n numbers in the sequence in memory.

Syntax:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE [ IF NOT EXISTS ] name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

Examples in this implementation:

- seqEmpID
- seqPatientID
- seqCtID
- seqCaseID
- seqDiagID
- seqPreID
- seqLabID

Indexes

Indexes provide faster access paths to specified table columns which can increase the speed of queries. Columns may be utilized in multiple indexes if each index contains a unique set of columns. Indexes are logically independent meaning they may be created or dropped at any time which will not affect the table data or other indexes within the database. However previously indexed data may be slower. The following indexing schemes are available:

- B-tree
- B-tree cluster
- Hash cluster
- Reverse key
- Bitmap
- Bitmap join

Syntax:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table [ USING method ]
    ( { column | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace ]
    [ WHERE predicate ]
```

Examples in this implementation:

- idx_employee_id
- idx_employee_ssn
- idx_employee_licno
- idx_cases_id

- idx_patient_id
- idx_patient_ssn
- idx_patient_lname
- idx_diagnosis_id
- idx_prescription_id
- idx_lab_id

Tables

Tables are the primary form of data storage within our database. The data is stored within rows (or tuples) which contain the attributes of the relational schema. Columns of a table have column names, a specified data type and a width. When the data is inserted into tables it can be updated, deleted or queried using structured query language. Tables can also have virtual columns which derive values on demand through user-specified functions.

Syntax:

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name
    OF type_name [ (
  { column_name WITH OPTIONS [ column_constraint [ ... ] ]
    | table_constraint }
    [, ... ]
) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]

where column_constraint is:

[ CONSTRAINT constraint_name ]
{ NOT NULL |
  NULL |
  CHECK ( expression ) [ NO INHERIT ] |
  DEFAULT default_expr |
  UNIQUE index_parameters |
  PRIMARY KEY index_parameters |
  REFERENCES reftable [ ( refcolumn ) ] [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ]
    [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and table_constraint is:
```

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [ WHERE ( predicate ) ]
|
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]

and like_option is:

{ INCLUDING | EXCLUDING } { DEFAULTS | CONSTRAINTS | INDEXES | STORAGE | COMMENTS | ALL }

index_parameters in UNIQUE, PRIMARY KEY, and EXCLUDE constraints are:

[ WITH ( storage_parameter [= value] [, ... ] ) ]
[ USING INDEX TABLESPACE tablespace_name ]

exclude_element in an EXCLUDE constraint is:

{ column_name | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ]
```

Examples in this implementation:

- employee
- patient
- careteam
- prescription
- labs
- diagnosis
- cases
- forms
- prescribes
- seenby

Views

When a query is stored a virtual table, the result is known as a view. Views do not store data, but rather are a SELECT statement that will generate a specific representation of data. By using views, Database Administrators (DBAs) are able to control, present, as well as format data before it is queried by other users of the database. In order to simplify queries and save time, front-end applications often query from views rather than the actual tables. The common operations SELECT, CREATE, INSERT, UPDATE, and DELETE can be applied to views in the same way they can be applied to tables of the database. When these operations are applied to a view, they are really being applied to the base table which the view is derived from. If you change information in the base table, the view will be dynamically recreated.

Syntax:

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Triggers

Triggers are stored blocks of SQL code that is automatically executed each time a specific event occurs (for example, an insertion operation).

Syntax:

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] [ INITIALLY IMMEDIATE | INITIALLY DEFERRED ] ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

Function

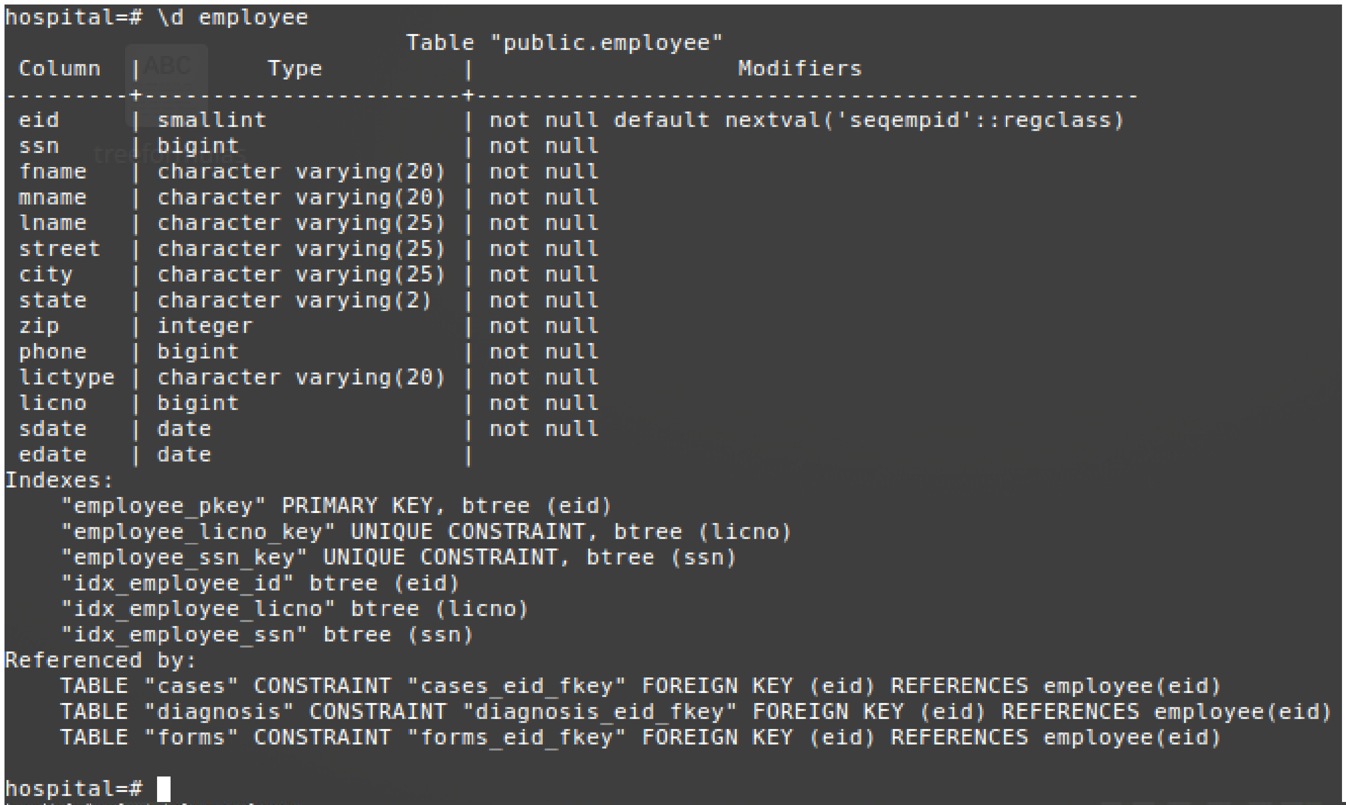
Any collection of commands in SQL can be packaged together and defined as a function. The command can include all query types including insertion, selection, update and deletion. A function can also include many other SQL commands, although not all of them. For a function to work, the final command must be a select query or have a returning clause that returns whatever is specified as the functions return type. One can also create a function that returns void.

Syntax:

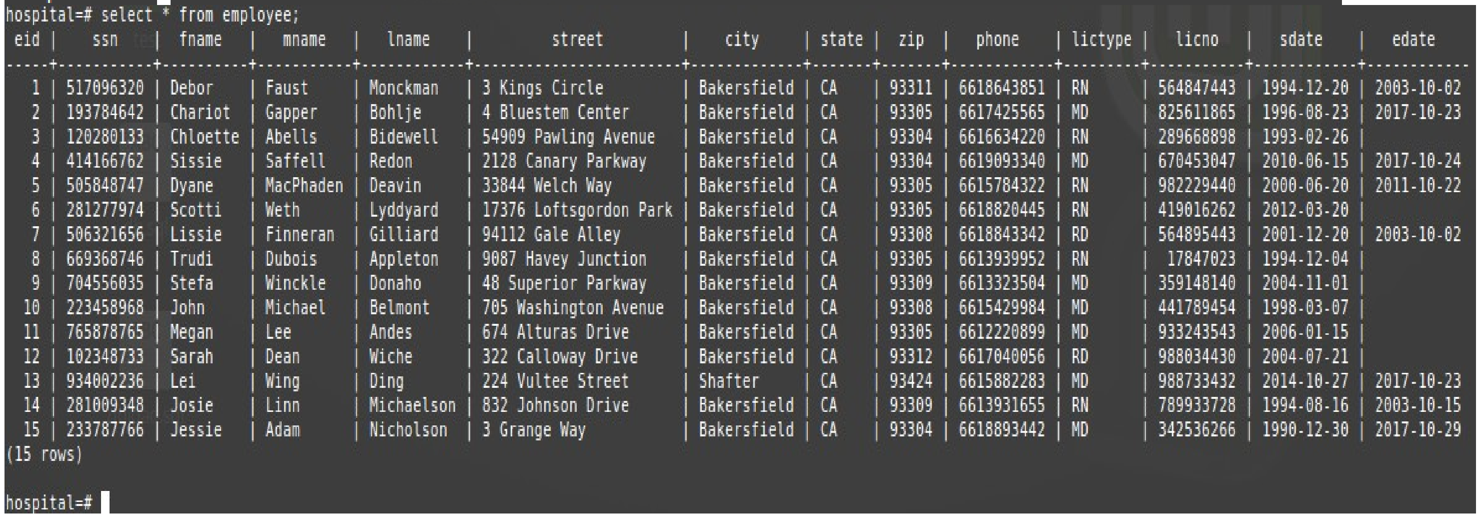
```
CREATE [ OR REPLACE ] FUNCTION
    name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...] ] )
    [ RETURNS rettype
      | RETURNS TABLE ( column_name column_type [, ...] ) ]
  { LANGUAGE lang_name
    | TRANSFORM { FOR TYPE type_name } [, ... ]
    | WINDOW
    | IMMUTABLE | STABLE | VOLATILE | [ NOT ] LEAKPROOF
    | CALLED ON NULL INPUT | RETURNS NULL ON NULL INPUT | STRICT
    | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
    | PARALLEL { UNSAFE | RESTRICTED | SAFE }
    | COST execution_cost
    | ROWS result_rows
    | SET configuration_parameter { TO value | = value | FROM CURRENT }
    | AS 'definition'
    | AS 'obj_file', 'link_symbol'
  } ...
    [ WITH ( attribute [, ...] ) ]
```

3.4 List Relations with SQL commands

Employee

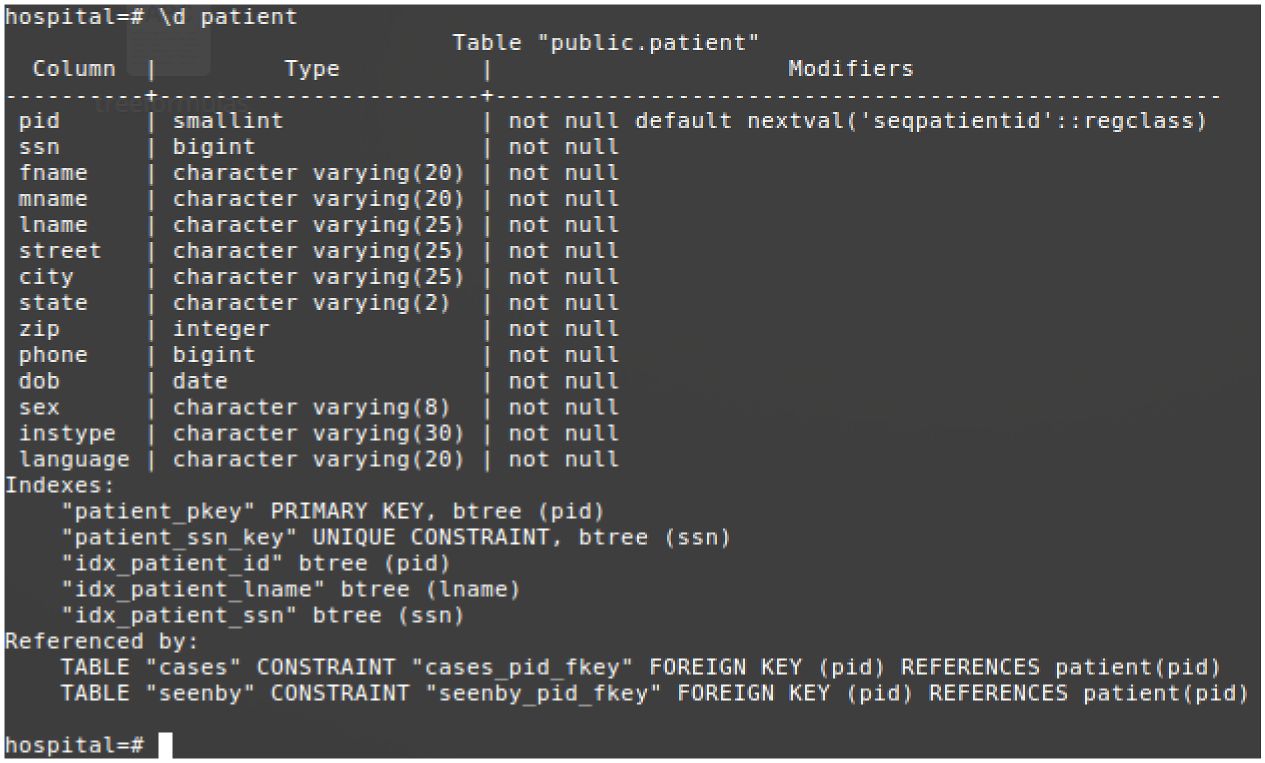


```
hospital=# \d employee
                             Table "public.employee"
 Column  |         Type          |                   Modifiers
---------+-----------------------+------------------------------------------------
 eid     | smallint              | not null default nextval('seqempid'::regclass)
 ssn     | bigint                | not null
 fname   | character varying(20) | not null
 mname   | character varying(20) | not null
 lname   | character varying(25) | not null
 street  | character varying(25) | not null
 city    | character varying(25) | not null
 state   | character varying(2)  | not null
 zip     | integer               | not null
 phone   | bigint                | not null
 lictype | character varying(20) | not null
 licno   | bigint                | not null
 sdate   | date                  | not null
 edate   | date                  |
Indexes:
    "employee_pkey" PRIMARY KEY, btree (eid)
    "employee_licno_key" UNIQUE CONSTRAINT, btree (licno)
    "employee_ssn_key" UNIQUE CONSTRAINT, btree (ssn)
    "idx_employee_id" btree (eid)
    "idx_employee_licno" btree (licno)
    "idx_employee_ssn" btree (ssn)
Referenced by:
    TABLE "cases" CONSTRAINT "cases_eid_fkey" FOREIGN KEY (eid) REFERENCES employee(eid)
    TABLE "diagnosis" CONSTRAINT "diagnosis_eid_fkey" FOREIGN KEY (eid) REFERENCES employee(eid)
    TABLE "forms" CONSTRAINT "forms_eid_fkey" FOREIGN KEY (eid) REFERENCES employee(eid)

hospital=#
```

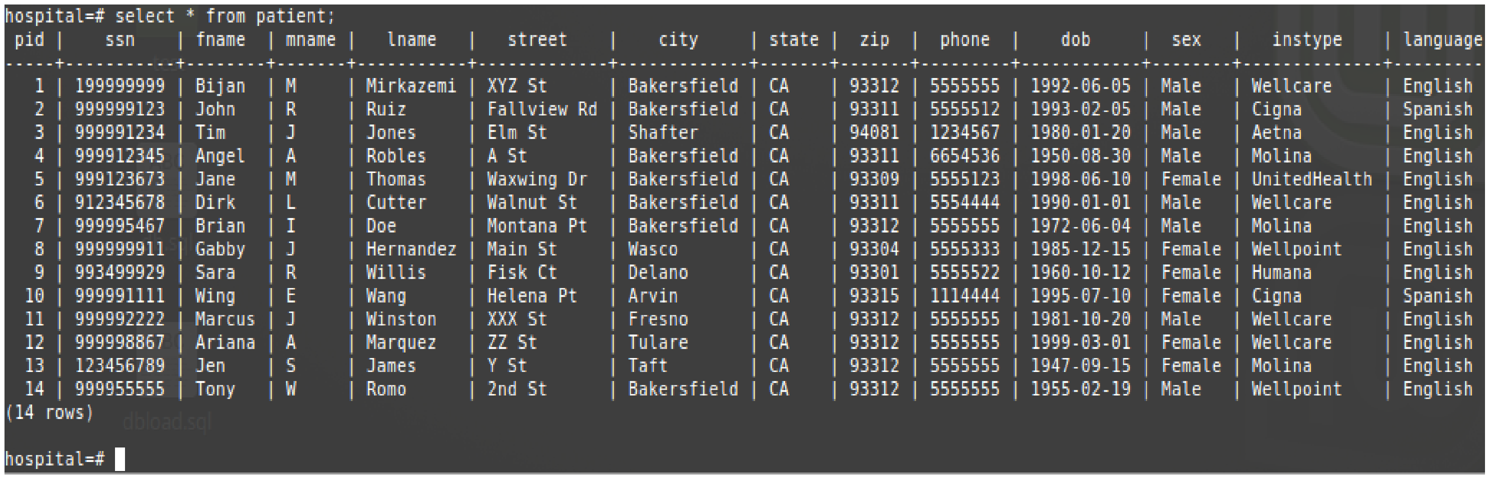


```
hospital=# select * from employee;
 eid |    ssn    |  fname   |   mname   |   lname    |         street          |    city     | state |  zip  |   phone    | lictype |   licno   |   sdate    |   edate
-----+-----------+----------+-----------+------------+-------------------------+-------------+-------+-------+------------+---------+-----------+------------+------------
   1 | 517096320 | Debor    | Faust     | Monckman   | 3 Kings Circle          | Bakersfield | CA    | 93311 | 6618643851 | RN      | 564847443 | 1994-12-20 | 2003-10-02
   2 | 193784642 | Chariot  | Gapper    | Bohlje     | 4 Bluestem Center       | Bakersfield | CA    | 93305 | 6617425565 | MD      | 825611865 | 1996-08-23 | 2017-10-23
   3 | 120280133 | Chloette | Abells    | Bidewell   | 54909 Pawling Avenue    | Bakersfield | CA    | 93304 | 6616634220 | RN      | 289668898 | 1993-02-26 |
   4 | 414166762 | Sissie   | Saffell   | Redon      | 2128 Canary Parkway     | Bakersfield | CA    | 93304 | 6619093340 | MD      | 670453047 | 2010-06-15 | 2017-10-24
   5 | 505848747 | Dyane    | MacPhaden | Deavin     | 33844 Welch Way         | Bakersfield | CA    | 93305 | 6615784322 | RN      | 982229440 | 2000-06-20 | 2011-10-22
   6 | 281277974 | Scotti   | Weth      | Lyddyard   | 17376 Loftsgordon Park  | Bakersfield | CA    | 93305 | 6618820445 | RN      | 419016262 | 2012-03-20 |
   7 | 506321656 | Lissie   | Finneran  | Gilliard   | 94112 Gale Alley        | Bakersfield | CA    | 93308 | 6618843342 | RD      | 564895443 | 2001-12-20 | 2003-10-02
   8 | 669368746 | Trudi    | Dubois    | Appleton   | 9087 Havey Junction     | Bakersfield | CA    | 93305 | 6613939952 | RN      |  17847023 | 1994-12-04 |
   9 | 704556035 | Stefa    | Winckle   | Donaho     | 48 Superior Parkway     | Bakersfield | CA    | 93309 | 6613323504 | MD      | 359148140 | 2004-11-01 |
  10 | 223458968 | John     | Michael   | Belmont    | 705 Washington Avenue   | Bakersfield | CA    | 93308 | 6615429984 | MD      | 441789454 | 1998-03-07 |
  11 | 765878765 | Megan    | Lee       | Andes      | 674 Alturas Drive       | Bakersfield | CA    | 93305 | 6612220899 | MD      | 933243543 | 2006-01-15 |
  12 | 102348733 | Sarah    | Dean      | Wiche      | 322 Calloway Drive      | Bakersfield | CA    | 93312 | 6617040056 | RD      | 988034430 | 2004-07-21 |
  13 | 934002236 | Lei      | Wing      | Ding       | 224 Vultee Street       | Shafter     | CA    | 93424 | 6615882283 | MD      | 988733432 | 2014-10-27 | 2017-10-23
  14 | 281009348 | Josie    | Linn      | Michaelson | 832 Johnson Drive       | Bakersfield | CA    | 93309 | 6613931655 | RN      | 789933728 | 1994-08-16 | 2003-10-15
  15 | 233787766 | Jessie   | Adam      | Nicholson  | 3 Grange Way            | Bakersfield | CA    | 93304 | 6618893442 | MD      | 342536266 | 1990-12-30 | 2017-10-29
(15 rows)

hospital=#
```

Patient

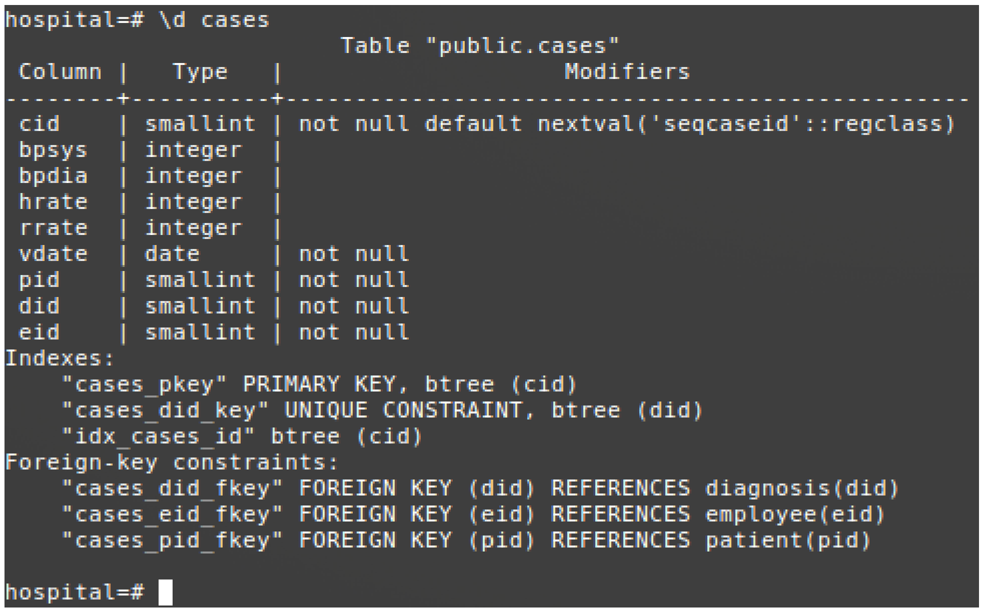


```
hospital=# \d patient
                                Table "public.patient"
  Column  |          Type          |                       Modifiers
----------+------------------------+----------------------------------------------------
 pid      | smallint               | not null default nextval('seqpatientid'::regclass)
 ssn      | bigint                 | not null
 fname    | character varying(20)  | not null
 mname    | character varying(20)  | not null
 lname    | character varying(25)  | not null
 street   | character varying(25)  | not null
 city     | character varying(25)  | not null
 state    | character varying(2)   | not null
 zip      | integer                | not null
 phone    | bigint                 | not null
 dob      | date                   | not null
 sex      | character varying(8)   | not null
 instype  | character varying(30)  | not null
 language | character varying(20)  | not null
Indexes:
    "patient_pkey" PRIMARY KEY, btree (pid)
    "patient_ssn_key" UNIQUE CONSTRAINT, btree (ssn)
    "idx_patient_id" btree (pid)
    "idx_patient_lname" btree (lname)
    "idx_patient_ssn" btree (ssn)
Referenced by:
    TABLE "cases" CONSTRAINT "cases_pid_fkey" FOREIGN KEY (pid) REFERENCES patient(pid)
    TABLE "seenby" CONSTRAINT "seenby_pid_fkey" FOREIGN KEY (pid) REFERENCES patient(pid)

hospital=#
```

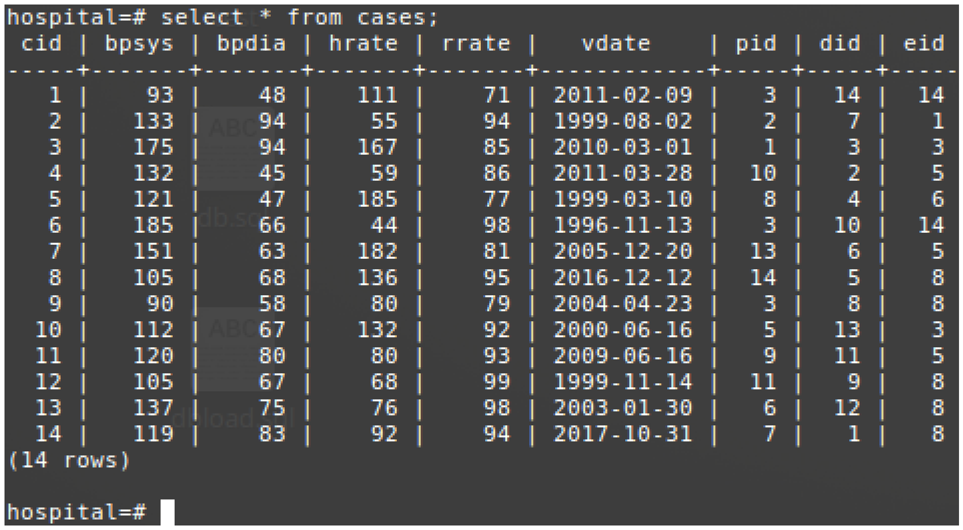


```
hospital=# select * from patient;
 pid |    ssn    | fname  | mname |   lname   |   street    |    city     | state |  zip  |  phone  |    dob     |  sex   |   instype    | language
-----+-----------+--------+-------+-----------+-------------+-------------+-------+-------+---------+------------+--------+--------------+---------
   1 | 199999999 | Bijan  | M     | Mirkazemi | XYZ St      | Bakersfield | CA    | 93312 | 5555555 | 1992-06-05 | Male   | Wellcare     | English
   2 | 999999123 | John   | R     | Ruiz      | Fallview Rd | Bakersfield | CA    | 93311 | 5555512 | 1993-02-05 | Male   | Cigna        | Spanish
   3 | 999991234 | Tim    | J     | Jones     | Elm St      | Shafter     | CA    | 94081 | 1234567 | 1980-01-20 | Male   | Aetna        | English
   4 | 999912345 | Angel  | A     | Robles    | A St        | Bakersfield | CA    | 93311 | 6654536 | 1950-08-30 | Male   | Molina       | English
   5 | 999123673 | Jane   | M     | Thomas    | Waxwing Dr  | Bakersfield | CA    | 93309 | 5555123 | 1998-06-10 | Female | UnitedHealth | English
   6 | 912345678 | Dirk   | L     | Cutter    | Walnut St   | Bakersfield | CA    | 93311 | 5554444 | 1990-01-01 | Male   | Wellcare     | English
   7 | 999995467 | Brian  | I     | Doe       | Montana Pt  | Bakersfield | CA    | 93312 | 5555555 | 1972-06-04 | Male   | Molina       | English
   8 | 999999911 | Gabby  | J     | Hernandez | Main St     | Wasco       | CA    | 93304 | 5555333 | 1985-12-15 | Female | Wellpoint    | English
   9 | 993499929 | Sara   | R     | Willis    | Fisk Ct     | Delano      | CA    | 93301 | 5555522 | 1960-10-12 | Female | Humana       | English
  10 | 999991111 | Wing   | E     | Wang      | Helena Pt   | Arvin       | CA    | 93315 | 1114444 | 1995-07-10 | Female | Cigna        | Spanish
  11 | 999992222 | Marcus | J     | Winston   | XXX St      | Fresno      | CA    | 93312 | 5555555 | 1981-10-20 | Male   | Wellcare     | English
  12 | 999998867 | Ariana | A     | Marquez   | ZZ St       | Tulare      | CA    | 93312 | 5555555 | 1999-03-01 | Female | Wellcare     | English
  13 | 123456789 | Jen    | S     | James     | Y St        | Taft        | CA    | 93312 | 5555555 | 1947-09-15 | Female | Molina       | English
  14 | 999955555 | Tony   | W     | Romo      | 2nd St      | Bakersfield | CA    | 93312 | 5555555 | 1955-02-19 | Male   | Wellpoint    | English
(14 rows)

hospital=#
```

Cases

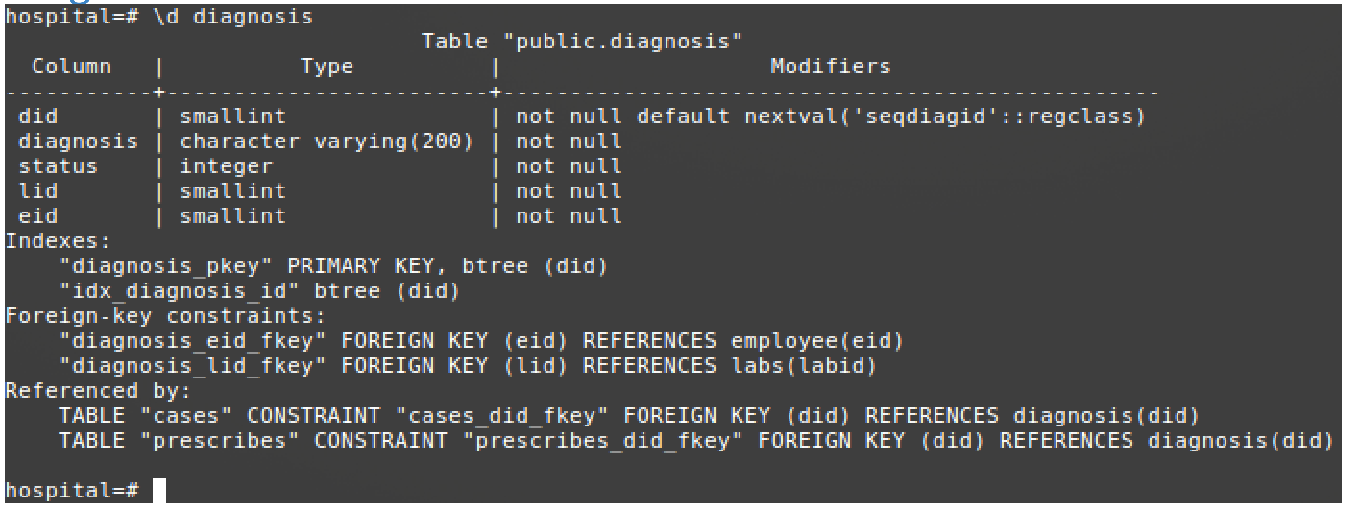


```
hospital=# \d cases
                        Table "public.cases"
 Column |   Type   |                       Modifiers
--------+----------+--------------------------------------------------
 cid    | smallint | not null default nextval('seqcaseid'::regclass)
 bpsys  | integer  |
 bpdia  | integer  |
 hrate  | integer  |
 rrate  | integer  |
 vdate  | date     | not null
 pid    | smallint | not null
 did    | smallint | not null
 eid    | smallint | not null
Indexes:
    "cases_pkey" PRIMARY KEY, btree (cid)
    "cases_did_key" UNIQUE CONSTRAINT, btree (did)
    "idx_cases_id" btree (cid)
Foreign-key constraints:
    "cases_did_fkey" FOREIGN KEY (did) REFERENCES diagnosis(did)
    "cases_eid_fkey" FOREIGN KEY (eid) REFERENCES employee(eid)
    "cases_pid_fkey" FOREIGN KEY (pid) REFERENCES patient(pid)

hospital=#
```

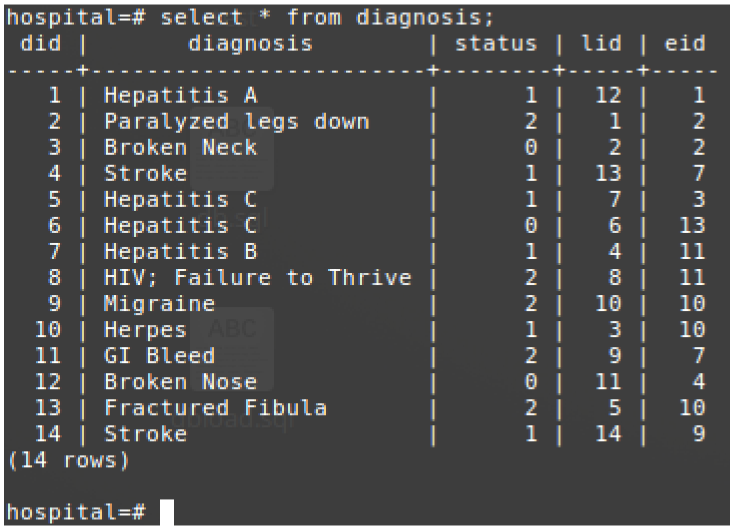


```
hospital=# select * from cases;
 cid | bpsys | bpdia | hrate | rrate |    vdate     | pid | did | eid
-----+-------+-------+-------+-------+------------+-----+-----+-----
   1 |    93 |    48 |   111 |    71 | 2011-02-09 |   3 |  14 |  14
   2 |   133 |    94 |    55 |    94 | 1999-08-02 |   2 |   7 |   1
   3 |   175 |    94 |   167 |    85 | 2010-03-01 |   1 |   3 |   3
   4 |   132 |    45 |    59 |    86 | 2011-03-28 |  10 |   2 |   5
   5 |   121 |    47 |   185 |    77 | 1999-03-10 |   8 |   4 |   6
   6 |   185 |    66 |    44 |    98 | 1996-11-13 |   3 |  10 |  14
   7 |   151 |    63 |   182 |    81 | 2005-12-20 |  13 |   6 |   5
   8 |   105 |    68 |   136 |    95 | 2016-12-12 |  14 |   5 |   8
   9 |    90 |    58 |    80 |    79 | 2004-04-23 |   3 |   8 |   8
  10 |   112 |    67 |   132 |    92 | 2000-06-16 |   5 |  13 |   3
  11 |   120 |    80 |    80 |    93 | 2009-06-16 |   9 |  11 |   5
  12 |   105 |    67 |    68 |    99 | 1999-11-14 |  11 |   9 |   8
  13 |   137 |    75 |    76 |    98 | 2003-01-30 |   6 |  12 |   8
  14 |   119 |    83 |    92 |    94 | 2017-10-31 |   7 |   1 |   8
(14 rows)

hospital=#
```

Diagnosis

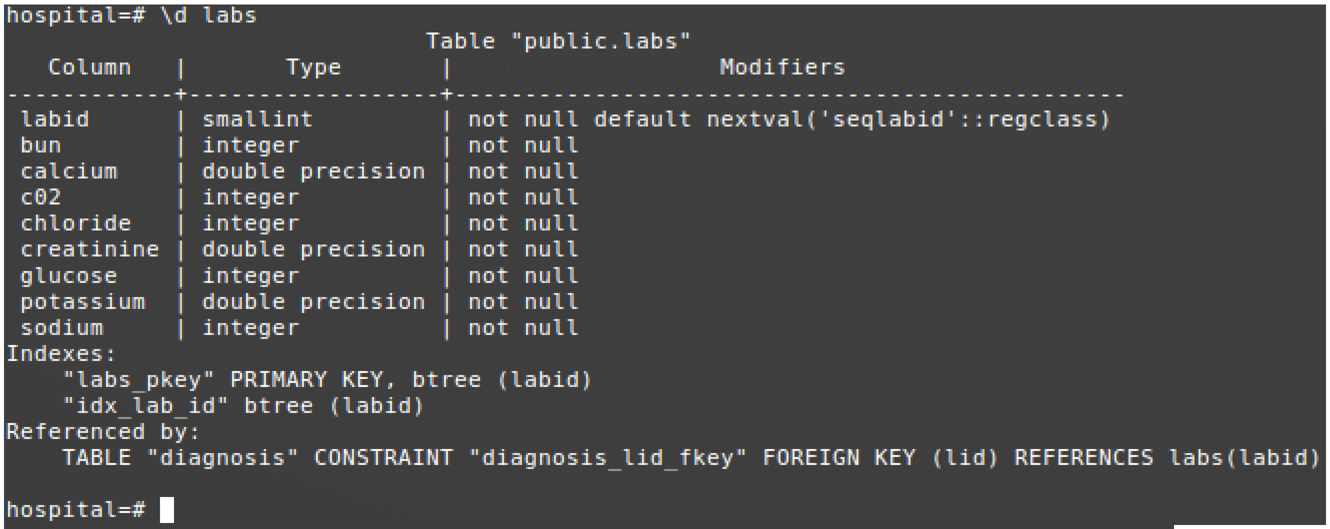


```
hospital=# \d diagnosis
                              Table "public.diagnosis"
  Column   |          Type          |                    Modifiers
-----------+------------------------+--------------------------------------------------
 did       | smallint               | not null default nextval('seqdiagid'::regclass)
 diagnosis | character varying(200) | not null
 status    | integer                | not null
 lid       | smallint               | not null
 eid       | smallint               | not null
Indexes:
    "diagnosis_pkey" PRIMARY KEY, btree (did)
    "idx_diagnosis_id" btree (did)
Foreign-key constraints:
    "diagnosis_eid_fkey" FOREIGN KEY (eid) REFERENCES employee(eid)
    "diagnosis_lid_fkey" FOREIGN KEY (lid) REFERENCES labs(labid)
Referenced by:
    TABLE "cases" CONSTRAINT "cases_did_fkey" FOREIGN KEY (did) REFERENCES diagnosis(did)
    TABLE "prescribes" CONSTRAINT "prescribes_did_fkey" FOREIGN KEY (did) REFERENCES diagnosis(did)

hospital=#
```

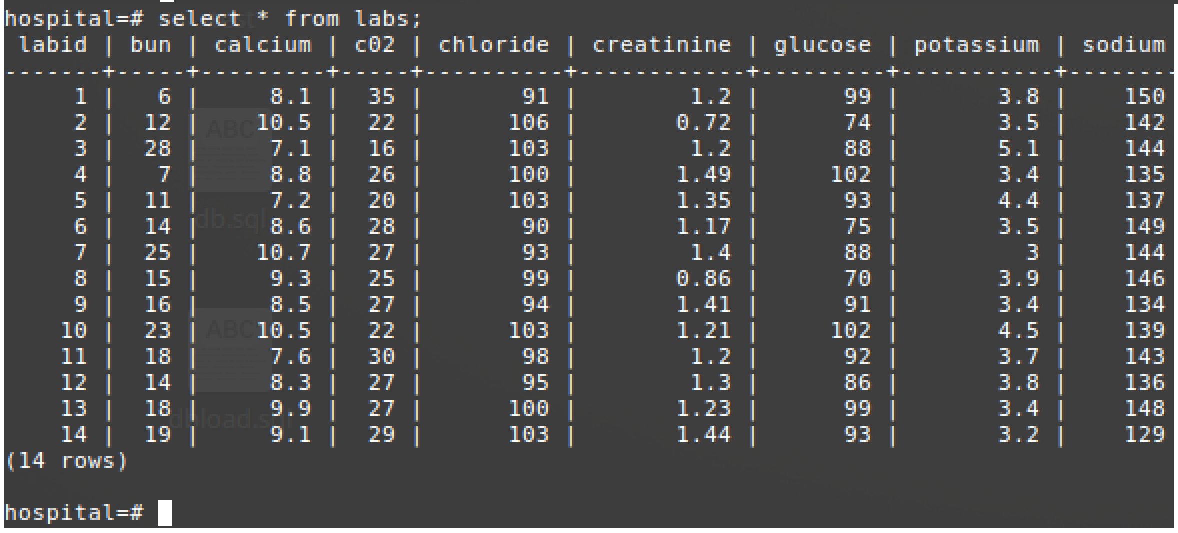


```
hospital=# select * from diagnosis;
 did |        diagnosis         | status | lid | eid
-----+--------------------------+--------+-----+-----
   1 | Hepatitis A              |      1 |  12 |   1
   2 | Paralyzed legs down      |      2 |   1 |   2
   3 | Broken Neck              |      0 |   2 |   2
   4 | Stroke                   |      1 |  13 |   7
   5 | Hepatitis C              |      1 |   7 |   3
   6 | Hepatitis C              |      0 |   6 |  13
   7 | Hepatitis B              |      1 |   4 |  11
   8 | HIV; Failure to Thrive   |      2 |   8 |  11
   9 | Migraine                 |      2 |  10 |  10
  10 | Herpes                   |      1 |   3 |  10
  11 | GI Bleed                 |      2 |   9 |   7
  12 | Broken Nose              |      0 |  11 |   4
  13 | Fractured Fibula         |      2 |   5 |  10
  14 | Stroke                   |      1 |  14 |   9
(14 rows)

hospital=#
```

Labs

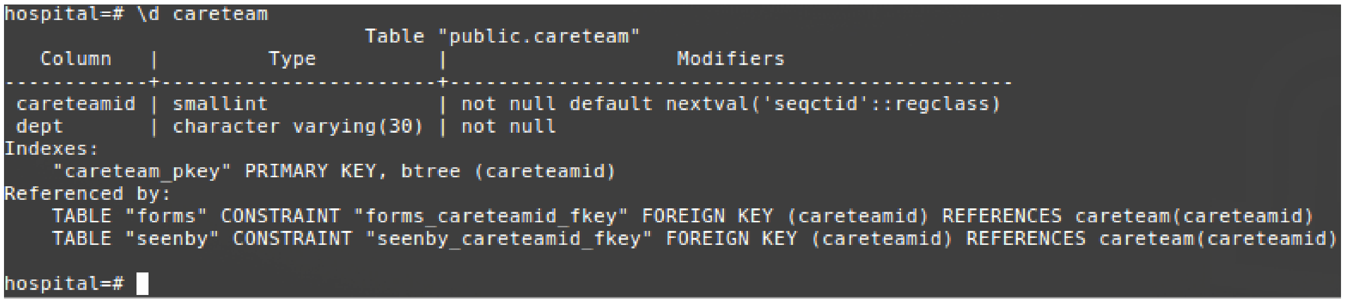


```
hospital=# \d labs
                              Table "public.labs"
   Column   |       Type       |                    Modifiers
------------+------------------+------------------------------------------------
 labid      | smallint         | not null default nextval('seqlabid'::regclass)
 bun        | integer          | not null
 calcium    | double precision | not null
 c02        | integer          | not null
 chloride   | integer          | not null
 creatinine | double precision | not null
 glucose    | integer          | not null
 potassium  | double precision | not null
 sodium     | integer          | not null
Indexes:
    "labs_pkey" PRIMARY KEY, btree (labid)
    "idx_lab_id" btree (labid)
Referenced by:
    TABLE "diagnosis" CONSTRAINT "diagnosis_lid_fkey" FOREIGN KEY (lid) REFERENCES labs(labid)

hospital=#
```

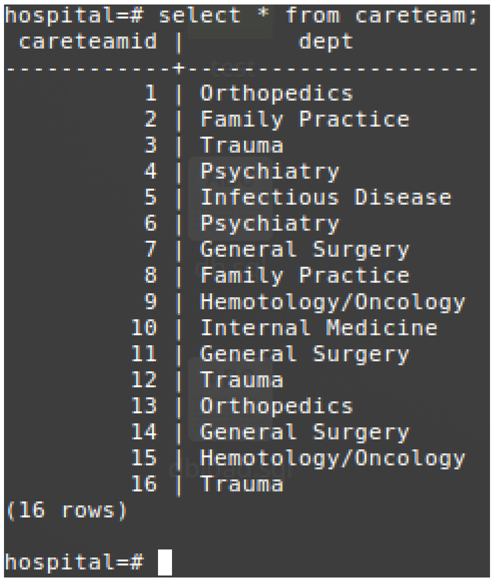


```
hospital=# select * from labs;
 labid | bun | calcium | c02 | chloride | creatinine | glucose | potassium | sodium
-------+-----+---------+-----+----------+------------+---------+-----------+--------
     1 |   6 |     8.1 |  35 |       91 |        1.2 |      99 |       3.8 |    150
     2 |  12 |    10.5 |  22 |      106 |       0.72 |      74 |       3.5 |    142
     3 |  28 |     7.1 |  16 |      103 |        1.2 |      88 |       5.1 |    144
     4 |   7 |     8.8 |  26 |      100 |       1.49 |     102 |       3.4 |    135
     5 |  11 |     7.2 |  20 |      103 |       1.35 |      93 |       4.4 |    137
     6 |  14 |     8.6 |  28 |       90 |       1.17 |      75 |       3.5 |    149
     7 |  25 |    10.7 |  27 |       93 |        1.4 |      88 |         3 |    144
     8 |  15 |     9.3 |  25 |       99 |       0.86 |      70 |       3.9 |    146
     9 |  16 |     8.5 |  27 |       94 |       1.41 |      91 |       3.4 |    134
    10 |  23 |    10.5 |  22 |      103 |       1.21 |     102 |       4.5 |    139
    11 |  18 |     7.6 |  30 |       98 |        1.2 |      92 |       3.7 |    143
    12 |  14 |     8.3 |  27 |       95 |        1.3 |      86 |       3.8 |    136
    13 |  18 |     9.9 |  27 |      100 |       1.23 |      99 |       3.4 |    148
    14 |  19 |     9.1 |  29 |      103 |       1.44 |      93 |       3.2 |    129
(14 rows)

hospital=#
```

CareTeam

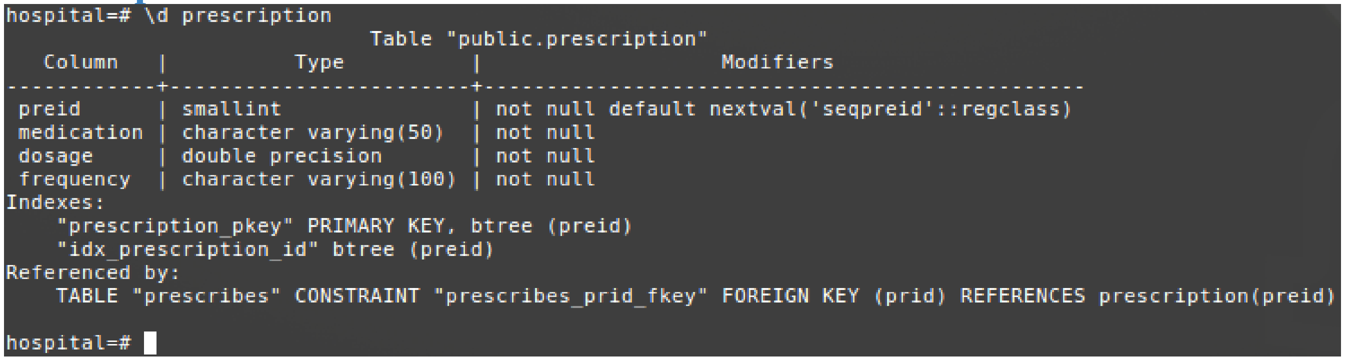


```
hospital=# \d careteam
                              Table "public.careteam"
   Column   |         Type          |                   Modifiers
------------+-----------------------+------------------------------------------------
 careteamid | smallint              | not null default nextval('seqctid'::regclass)
 dept       | character varying(30) | not null
Indexes:
    "careteam_pkey" PRIMARY KEY, btree (careteamid)
Referenced by:
    TABLE "forms" CONSTRAINT "forms_careteamid_fkey" FOREIGN KEY (careteamid) REFERENCES careteam(careteamid)
    TABLE "seenby" CONSTRAINT "seenby_careteamid_fkey" FOREIGN KEY (careteamid) REFERENCES careteam(careteamid)

hospital=#
```

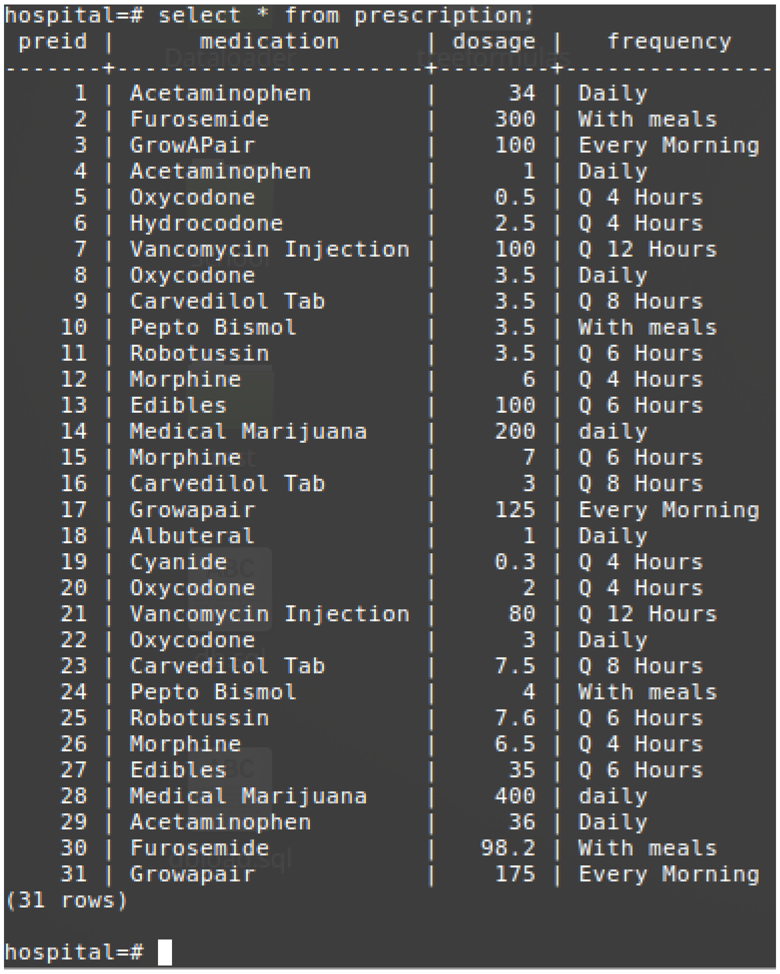


```
hospital=# select * from careteam;
 careteamid |         dept
------------+---------------------
          1 | Orthopedics
          2 | Family Practice
          3 | Trauma
          4 | Psychiatry
          5 | Infectious Disease
          6 | Psychiatry
          7 | General Surgery
          8 | Family Practice
          9 | Hemotology/Oncology
         10 | Internal Medicine
         11 | General Surgery
         12 | Trauma
         13 | Orthopedics
         14 | General Surgery
         15 | Hemotology/Oncology
         16 | Trauma
(16 rows)

hospital=#
```

Prescription

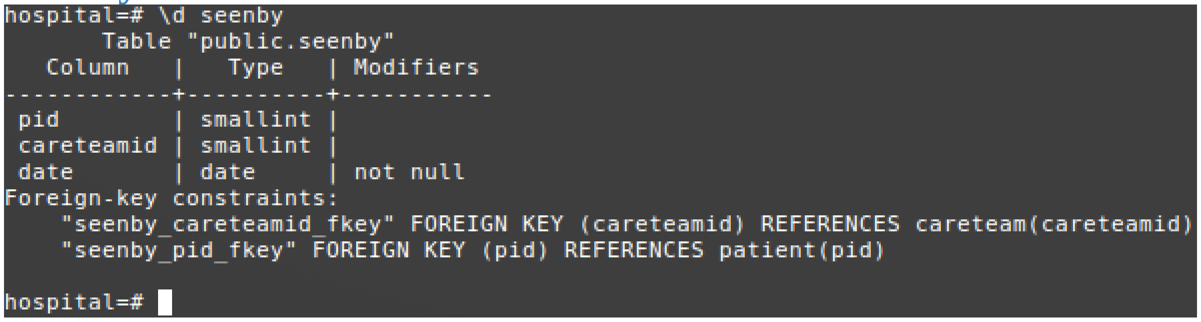


```
hospital=# \d prescription
                              Table "public.prescription"
   Column   |          Type          |                   Modifiers
------------+------------------------+------------------------------------------------
 preid      | smallint               | not null default nextval('seqpreid'::regclass)
 medication | character varying(50)  | not null
 dosage     | double precision       | not null
 frequency  | character varying(100) | not null
Indexes:
    "prescription_pkey" PRIMARY KEY, btree (preid)
    "idx_prescription_id" btree (preid)
Referenced by:
    TABLE "prescribes" CONSTRAINT "prescribes_prid_fkey" FOREIGN KEY (prid) REFERENCES prescription(preid)

hospital=#
```

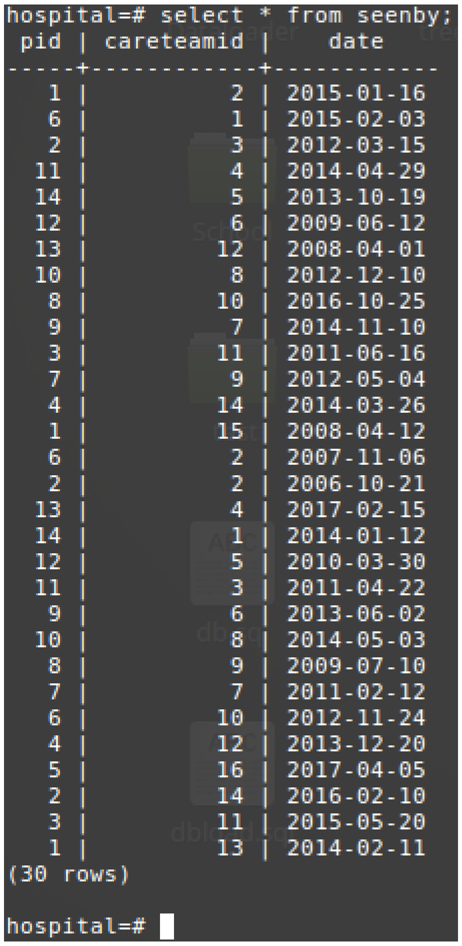


```
hospital=# select * from prescription;
 preid |      medication      | dosage |   frequency
-------+----------------------+--------+---------------
     1 | Acetaminophen        |     34 | Daily
     2 | Furosemide           |    300 | With meals
     3 | GrowAPair            |    100 | Every Morning
     4 | Acetaminophen        |      1 | Daily
     5 | Oxycodone            |    0.5 | Q 4 Hours
     6 | Hydrocodone          |    2.5 | Q 4 Hours
     7 | Vancomycin Injection |    100 | Q 12 Hours
     8 | Oxycodone            |    3.5 | Daily
     9 | Carvedilol Tab       |    3.5 | Q 8 Hours
    10 | Pepto Bismol         |    3.5 | With meals
    11 | Robotussin           |    3.5 | Q 6 Hours
    12 | Morphine             |      6 | Q 4 Hours
    13 | Edibles              |    100 | Q 6 Hours
    14 | Medical Marijuana    |    200 | daily
    15 | Morphine             |      7 | Q 6 Hours
    16 | Carvedilol Tab       |      3 | Q 8 Hours
    17 | Growapair            |    125 | Every Morning
    18 | Albuteral            |      1 | Daily
    19 | Cyanide              |    0.3 | Q 4 Hours
    20 | Oxycodone            |      2 | Q 4 Hours
    21 | Vancomycin Injection |     80 | Q 12 Hours
    22 | Oxycodone            |      3 | Daily
    23 | Carvedilol Tab       |    7.5 | Q 8 Hours
    24 | Pepto Bismol         |      4 | With meals
    25 | Robotussin           |    7.6 | Q 6 Hours
    26 | Morphine             |    6.5 | Q 4 Hours
    27 | Edibles              |     35 | Q 6 Hours
    28 | Medical Marijuana    |    400 | daily
    29 | Acetaminophen        |     36 | Daily
    30 | Furosemide           |   98.2 | With meals
    31 | Growapair            |    175 | Every Morning
(31 rows)

hospital=# 
```

SeenBy

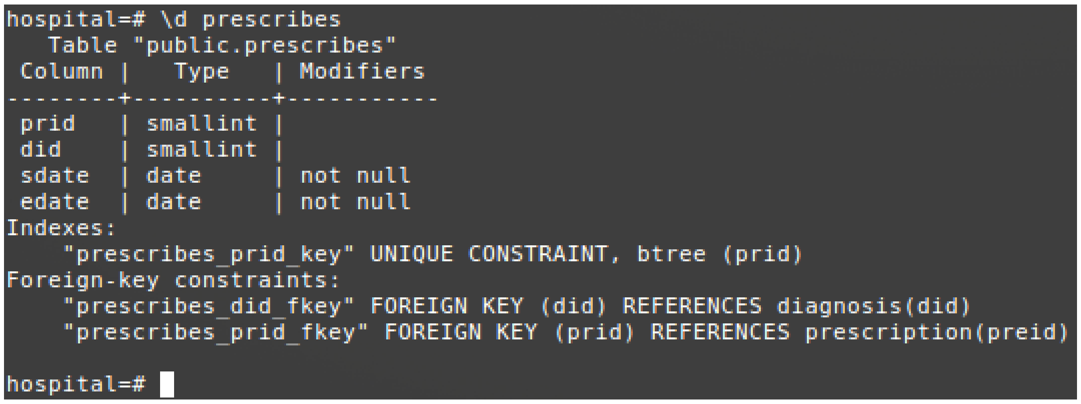


```
hospital=# \d seenby
       Table "public.seenby"
   Column   |   Type   | Modifiers
------------+----------+-----------
 pid        | smallint |
 careteamid | smallint |
 date       | date     | not null
Foreign-key constraints:
    "seenby_careteamid_fkey" FOREIGN KEY (careteamid) REFERENCES careteam(careteamid)
    "seenby_pid_fkey" FOREIGN KEY (pid) REFERENCES patient(pid)

hospital=# 
```

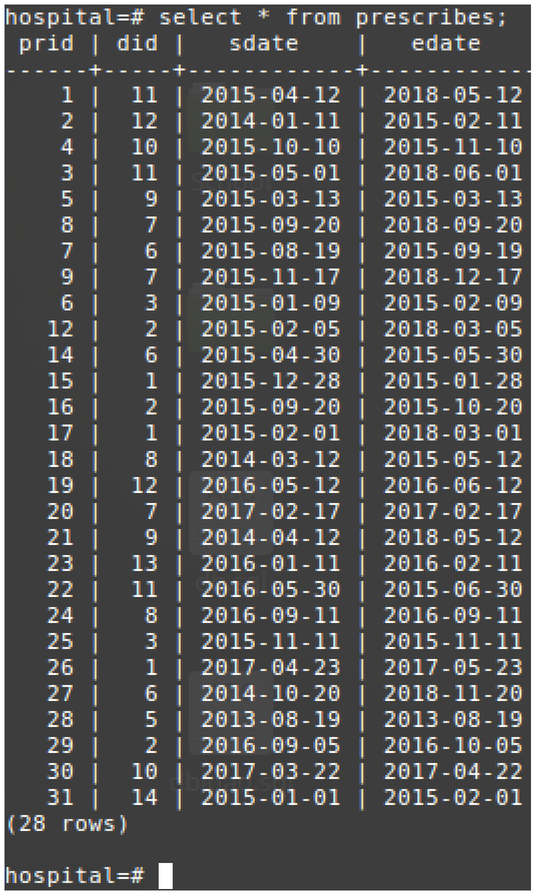


```
hospital=# select * from seenby;
 pid | careteamid |    date
-----+------------+------------
   1 |          2 | 2015-01-16
   6 |          1 | 2015-02-03
   2 |          3 | 2012-03-15
  11 |          4 | 2014-04-29
  14 |          5 | 2013-10-19
  12 |          6 | 2009-06-12
  13 |         12 | 2008-04-01
  10 |          8 | 2012-12-10
   8 |         10 | 2016-10-25
   9 |          7 | 2014-11-10
   3 |         11 | 2011-06-16
   7 |          9 | 2012-05-04
   4 |         14 | 2014-03-26
   1 |         15 | 2008-04-12
   6 |          2 | 2007-11-06
   2 |          2 | 2006-10-21
  13 |          4 | 2017-02-15
  14 |          1 | 2014-01-12
  12 |          5 | 2010-03-30
  11 |          3 | 2011-04-22
   9 |          6 | 2013-06-02
  10 |          8 | 2014-05-03
   8 |          9 | 2009-07-10
   7 |          7 | 2011-02-12
   6 |         10 | 2012-11-24
   4 |         12 | 2013-12-20
   5 |         16 | 2017-04-05
   2 |         14 | 2016-02-10
   3 |         11 | 2015-05-20
   1 |         13 | 2014-02-11
(30 rows)

hospital=#
```

Prescribes

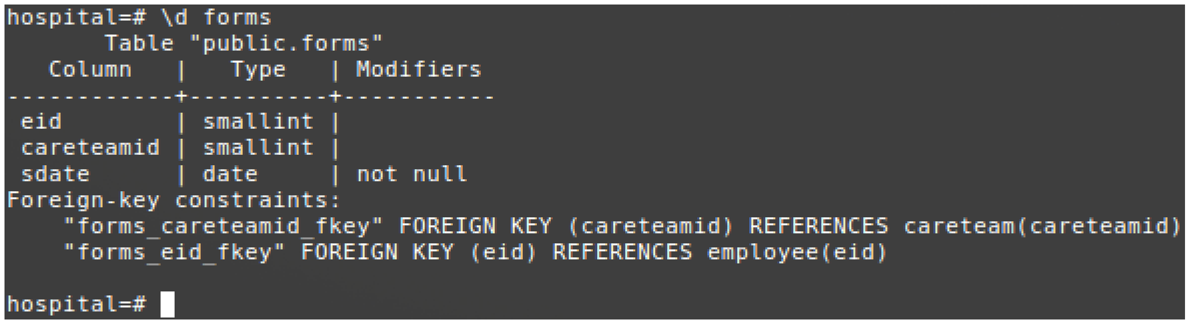


```
hospital=# \d prescribes
   Table "public.prescribes"
 Column |   Type   | Modifiers
--------+----------+-----------
 prid   | smallint |
 did    | smallint |
 sdate  | date     | not null
 edate  | date     | not null
Indexes:
    "prescribes_prid_key" UNIQUE CONSTRAINT, btree (prid)
Foreign-key constraints:
    "prescribes_did_fkey" FOREIGN KEY (did) REFERENCES diagnosis(did)
    "prescribes_prid_fkey" FOREIGN KEY (prid) REFERENCES prescription(preid)

hospital=#
```

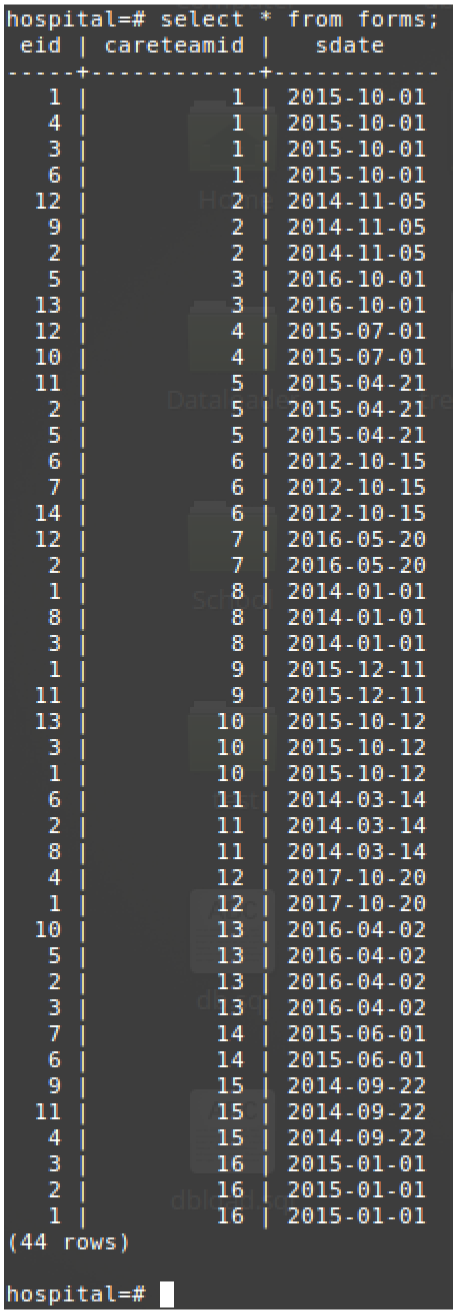


```
hospital=# select * from prescribes;
 prid | did |   sdate    |   edate
------+-----+------------+------------
    1 |  11 | 2015-04-12 | 2018-05-12
    2 |  12 | 2014-01-11 | 2015-02-11
    4 |  10 | 2015-10-10 | 2015-11-10
    3 |  11 | 2015-05-01 | 2018-06-01
    5 |   9 | 2015-03-13 | 2015-03-13
    8 |   7 | 2015-09-20 | 2018-09-20
    7 |   6 | 2015-08-19 | 2015-09-19
    9 |   7 | 2015-11-17 | 2018-12-17
    6 |   3 | 2015-01-09 | 2015-02-09
   12 |   2 | 2015-02-05 | 2018-03-05
   14 |   6 | 2015-04-30 | 2015-05-30
   15 |   1 | 2015-12-28 | 2015-01-28
   16 |   2 | 2015-09-20 | 2015-10-20
   17 |   1 | 2015-02-01 | 2018-03-01
   18 |   8 | 2014-03-12 | 2015-05-12
   19 |  12 | 2016-05-12 | 2016-06-12
   20 |   7 | 2017-02-17 | 2017-02-17
   21 |   9 | 2014-04-12 | 2018-05-12
   23 |  13 | 2016-01-11 | 2016-02-11
   22 |  11 | 2016-05-30 | 2015-06-30
   24 |   8 | 2016-09-11 | 2016-09-11
   25 |   3 | 2015-11-11 | 2015-11-11
   26 |   1 | 2017-04-23 | 2017-05-23
   27 |   6 | 2014-10-20 | 2018-11-20
   28 |   5 | 2013-08-19 | 2013-08-19
   29 |   2 | 2016-09-05 | 2016-10-05
   30 |  10 | 2017-03-22 | 2017-04-22
   31 |  14 | 2015-01-01 | 2015-02-01
(28 rows)

hospital=#
```

Forms



```
hospital=# \d forms
       Table "public.forms"
   Column    |   Type   | Modifiers
-------------+----------+-----------
 eid         | smallint |
 careteamid  | smallint |
 sdate       | date     | not null
Foreign-key constraints:
    "forms_careteamid_fkey" FOREIGN KEY (careteamid) REFERENCES careteam(careteamid)
    "forms_eid_fkey" FOREIGN KEY (eid) REFERENCES employee(eid)

hospital=#
```



```
hospital=# select * from forms;
 eid | careteamid |   sdate
-----+------------+------------
   1 |          1 | 2015-10-01
   4 |          1 | 2015-10-01
   3 |          1 | 2015-10-01
   6 |          1 | 2015-10-01
  12 |          2 | 2014-11-05
   9 |          2 | 2014-11-05
   2 |          2 | 2014-11-05
   5 |          3 | 2016-10-01
  13 |          3 | 2016-10-01
  12 |          4 | 2015-07-01
  10 |          4 | 2015-07-01
  11 |          5 | 2015-04-21
   2 |          5 | 2015-04-21
   5 |          5 | 2015-04-21
   6 |          6 | 2012-10-15
   7 |          6 | 2012-10-15
  14 |          6 | 2012-10-15
  12 |          7 | 2016-05-20
   2 |          7 | 2016-05-20
   1 |          8 | 2014-01-01
   8 |          8 | 2014-01-01
   3 |          8 | 2014-01-01
   1 |          9 | 2015-12-11
  11 |          9 | 2015-12-11
  13 |         10 | 2015-10-12
   3 |         10 | 2015-10-12
   1 |         10 | 2015-10-12
   6 |         11 | 2014-03-14
   2 |         11 | 2014-03-14
   8 |         11 | 2014-03-14
   4 |         12 | 2017-10-20
   1 |         12 | 2017-10-20
  10 |         13 | 2016-04-02
   5 |         13 | 2016-04-02
   2 |         13 | 2016-04-02
   3 |         13 | 2016-04-02
   7 |         14 | 2015-06-01
   6 |         14 | 2015-06-01
   9 |         15 | 2014-09-22
  11 |         15 | 2014-09-22
   4 |         15 | 2014-09-22
   3 |         16 | 2015-01-01
   2 |         16 | 2015-01-01
   1 |         16 | 2015-01-01
(44 rows)

hospital=#
```

3.5 Example Queries in SQL

This section will detail the SQL implementation of each of the queries mentioned in section 2. We will also be adding three additional queries in order to display psql functionality. For each query, we will display the query as well as a grouping of sample data the query returns. Test data will be used and will not represent the final result of our database design in future phases.

1.) List all patients who were at the hospital in 1999 but did not get prescribed any medication.

```
CREATE OR REPLACE VIEW query1 AS
SELECT DISTINCT p.pID, fname, lname
        FROM patient p WHERE EXISTS (SELECT * FROM diagnosis d, cases c
        WHERE p.pID = c.pID AND c.vDate <= '12/31/1999'::date AND c.vDate >= '1999-01-01'::date
        AND d.did = c.did AND NOT EXISTS (SELECT * FROM prescribes pr WHERE pr.dID =
        d.dID
        ))
;
```

```
hospital=# SELECT * FROM query1;
 pid | fname |   lname
-----+-------+-----------
   8 | Gabby | Hernandez
(1 row)
```

2.) List all nurses who have seen at least 3 cases but no longer work at the hospital.

```
CREATE OR REPLACE VIEW query2 AS
        SELECT DISTINCT e.eID, fname, lname
        FROM employee e, cases c1, cases c2, cases c3
        WHERE e.eDate IS NOT NULL
        AND e.eID = c1.eID
        AND e.eID = c2.eID
        AND e.eID = c3.eID
        AND c1.cID != c2.cID
        AND c2.cID != c3.cID
        AND c1.cID != c3.cID
;
```

```
hospital=# SELECT * FROM query2;
 eid | fname | lname
-----+-------+--------
   5 | Dyane | Deavin
(1 row)
```

3.) List the patient with the lowest heart rate recorded in the hospital.

```
CREATE OR REPLACE VIEW query3 AS
  SELECT DISTINCT p.pID, fname, lname
        FROM patient p WHERE EXISTS (SELECT * FROM cases c WHERE
        c.pID = p.pID AND NOT EXISTS (SELECT * FROM cases c2 WHERE
        c2.cID != c.cID AND c2.hrate < c.hrate))
;
```

```
hospital=# SELECT * FROM query3;
 pid | fname | lname
-----+-------+-------
   3 | Tim   |
(1 row)
```

4.) List the youngest patient who has been to the hospital.

```
CREATE OR REPLACE VIEW query4 AS
SELECT DISTINCT p.pid, fname, lname, p.dob
        From Patient p WHERE NOT EXISTS(SELECT * FROM patient p2 WHERE
        p.pID != p2.pID AND p.dob < p2.dob)
;
```

```
hospital=# SELECT * FROM query4;
 pid | fname  |  lname  |    dob
-----+--------+---------+------------
  12 | Ariana | Marquez | 1999-03-01
(1 row)
```

5.) List all patients with 2 or more current prescriptions in a single diagnosis.

```
CREATE OR REPLACE VIEW query5 AS
        SELECT DISTINCT  p.pid, fname, lname
        FROM patient p, cases c, diagnosis d, prescribes pr, prescribes pr2
        WHERE c.did = d.did AND p.pid = c.pid AND pr.did = d.did AND pr.eDATE >= now()
        AND pr2.did = d.did AND pr2.prid != pr.prid AND pr2.edate >= now()
```

```
;

hospital=# SELECT * FROM query5;
 pid | fname | lname
-----+-------+--------
   9 | Sara  | Willis
   2 | John  | Ruiz
(2 rows)
```

6.) List all patients who have been to the hospital exactly once.

```
CREATE OR REPLACE VIEW query6 AS
  SELECT DISTINCT p.pID, fname, lname
            FROM patient p
            WHERE EXISTS (SELECT * FROM cases c
                        WHERE c.pID = p.pID AND
                        NOT EXISTS (SELECT * FROM cases c2
                        WHERE c2.pID = p.pID AND c.vDate != c2.vDate
                        )
            )
;

hospital=# SELECT * FROM query6;
 pid | fname  |   lname
-----+--------+-----------
   1 | Bijan  | Mirkazemi
   2 | John   | Ruiz
   5 | Jane   | Thomas
   6 | Dirk   | Cutter
   7 | Brian  | Doe
   8 | Gabby  | Hernandez
   9 | Sara   | Willis
  10 | Wing   | Wang
  11 | Marcus | Winston
  13 | Jen    | James
  14 | Tony   | Romo
(11 rows)
```

7.) List all patients who have received a diagnosis by every doctor currently employed at the hospital.

```
CREATE OR REPLACE VIEW query7 AS
        SELECT DISTINCT p.pID, fname, lname
        FROM patient p
        WHERE NOT EXISTS (SELECT *
                FROM employee e
```

```
            WHERE e.eDate IS NULL AND e.licType = 'MD'
            AND NOT EXISTS ( SELECT *
                  FROM cases c, diagnosis d
                  WHERE p.pID = c.pID
                  AND e.eID = d.eID
                  AND c.dID = d.dID
      ))
;

hospital=# SELECT * FROM query7;
 pid   | fname | lname
-------+---------+---------
   3    | Tim     | Jones
(1 row)
```

8.) List all patients who have been seen by a doctor or nurse but do not have any active. prescriptions.

```
CREATE OR REPLACE VIEW query8 AS
         SELECT DISTINCT p.pID, fname, lname
         FROM patient p
         WHERE EXISTS (SELECT * FROM cases c, diagnosis d
                  WHERE c.pID = p.pID AND d.status=2 AND c.dID = d.dID
                  AND NOT EXISTS (SELECT * FROM prescribes pr
                  WHERE pr.sDate <= now() ANd pr.eDate >= now()
                  AND pr.dID = d.dID
                  )
         )
;

hospital=# SELECT * FROM query8;
 pid | fname | lname
-----+-------+--------
   3 | Tim   | Jones
   5 | Jane  | Thomas
(2 rows)
```

9.) List all male patients over the age of 50 with calcium levels over 9.0 (very high).

```
CREATE OR REPLACE VIEW query9 AS
         SELECT DISTINCT p.pID, fname, lname, calcium
         FROM patient p, cases c, diagnosis d, labs l
         WHERE p.sex = 'Male' AND p.dob < now()-interval'50 years'
         AND p.pID = c.pID AND c.dID = d.dID
         AND d.lID = l.labID AND l.calcium > 9.0
```

```
;

hospital=# SELECT * FROM query9;
 pid | fname | lname | calcium
-----+-------+-------+---------
  14 | Tony  | Romo  |    10.7
(1 row)
```

10.) List all patients who were only seen by a nurse but not a doctor.

```
CREATE OR REPLACE VIEW query10 AS
SELECT DISTINCT p.pid, fname, lname
        FROM patient p, cases c, diagnosis d WHERE
        p.pid = c.pid AND c.did = d.did AND d.status = 1
;

hospital=# SELECT * FROM query10;
 pid | fname |   lname
-----+---------+-----------
   2 | John    | Ruiz
   3 | Tim      | Jones
   7 | Brian   | Doe
   8 | Gabby | Hernandez
  14| Tony    | Romo
```

Additional Queries:

In addition to the ten queries we created and implemented above, we have included a few additional queries which demonstrate some special functionalities of the PostgreSQL that are not possible in relational algebra.

11.) List all patients and how many cases (or number of visits) they have had at the hospital.

```
CREATE OR REPLACE VIEW query11 AS
        SELECT p.fname "First Name", p.lname "Last Name", count(*) "Number of Visits"
        FROM patient p  NATURAL JOIN cases c
        GROUP BY p.fname, p.lname
        ORDER BY count(*) DESC;
;

hospital=# SELECT * FROM query11;
 First Name | Last Name | Number of Visits
------------+---------------+------------------
 Tim        | Jones         |              3
 Sara       | Willis        |              1
 Wing       | Wang          |              1
 Brian      | Doe           |              1
 Tony       | Romo          |              1
 Jane       | Thomas        |              1
 John       | Ruiz          |              1
 Dirk       | Cutter        |              1
 Jen        | James         |              1
 Gabby      | Hernandez |              1
 Marcus     | Winston       |              1
 Bijan      | Mirkazemi  |              1
(12 rows)
```

The above query uses a GROUP BY clause as well as a count aggregate function which shows the number of visits a patient has made to the hospital. The GROUP BY clause combines tuples that have the same value for a set of attributes (or attribute expressions) into a single tuple. For this, an aggregate function is required. It is ran over attribute expressions from the grouped tuples and returns the result as one of the columns. It also utilizes an ORDER BY clause which organizes the data by Number of Visits in descending order.

12.) List the patients names heart rates that are greater than the average heart rate of all patients.

```
CREATE OR REPLACE VIEW query12 AS
        SELECT p.fname, p.lname, hrate
        FROM patient p NATURAL JOIN cases c
        WHERE hrate > (SELECT avg(hrate) FROM cases)
        ORDER by hrate
;

hospital=# SELECT * FROM query12;
 fname  |   lname       | hrate
----------+---------------+-------
 Tim      | Jones        |   111
 Jane    | Thomas      |   132
 Tony    | Romo        |   136
 Bijan   | Mirkazemi  |   167
 Jen     | James       |   182
 Gabby | Hernandez |   185
(6 rows)
```

The above query uses a NATURAL JOIN to automatically group together attributes from the patient relation to their corresponding case relation. It also uses a function avg() to gather the average of all heart rates of patients that have been seen.

13.) List all nurses and how many cases they have taken.

```
CREATE OR REPLACE VIEW query13 AS
        SELECT e.eid, e.fname, e.lname, count(c)
        FROM employee e FULL OUTER JOIN cases c ON e.eid = c.eid
        WHERE licTYPE = 'RN'
        GROUP BY e.eid, e.fname, e.lname
        ORDER by count(c) ASC
;
hospital=# SELECT * FROM query13;
 eid |  fname   |   lname     | count
------+-----------+---------------+-------
   1  | Debor    | Monckman |     1
   6  | Scotti     | Lyddyard    |     1
   3  | Chloette | Bidewell     |     2
  14 | Josie      | Michaelson |     2
   5  | Dyane   | Deavin        |     3
   8  | Trudi      | Appleton    |     5
(6 rows)
```

The above query utilizes a FULL OUTER JOIN which gathers all information from both the employee relation and case relation. It again utilizes the count function but only counts within the 'c' relation to determine the number of cases each nurse has taken at the hospital. It also again uses the GROUP BY and ORDER by clauses as explained on query 11.

3.6 Data Loader

To add large amounts of data to our database, a method of loading this data is necessary. There are multiple ways to do this such as writing SQL statements to insert data into the database. Also, there are various programs which will automatically generate SQL insertion scripts to put the data into the database quickly.

Insertion Statements:

To insert data into postgreSQL, one can do so manually with SQL statements. SQL statements are the simplest way to insert data into the database.

Example I:
```
INSERT INTO [table name]
        [column name 1, … columnt name n]
VALUES
        [expression 1 … expression n]
```

This example allows one to specify value expressions for each column in the table when inserting a record.

Example II:
```
INSERT INTO [table name]
        [select query]
```

This example allows one to use the result of a query as column values.

When loading large amounts of data, this method is not the best to use as it can be very time consuming. This is how we loaded our data into our database.

Data Loader:
The data loader is a program written by Dr. Wang which uses command-line interface to insert data into tables from a text file. The user of the application must specify the database name, password, and the text file for which to use in the command line. The text file shall follow a specific format that specifies the data and the table in which the tuple should be inserted. The user can also determine which character is used as a delimiter to separate columns through the command line (such as "," or "|"). "INSERT INTO" SQL statements are generated and ran based on the information within the text file.

Oracle SQL Developer:
While Oracle SQL Developer is made for the Oracle Database Management System, many have found ways to use the software with PostgreSQL. This software is provided by Oracle free of charge. It allows users to develop and manage a database. It utilizes a graphical user interface that demonstrates all of the user's tables. The SQL developer GUI also has the ability to import data into tables with CSV files. This then returns an insert script with SQL commands if the user wants to paste the commands onto files.

pgAdmin:
pgAdmin is a graphical user interface program made specifically for postgreSQL. This application allows users to develop and manage databases with ease. It uses a graphical user interface to display all of the user's tables and important information. This software is freely available. It allows users to easily insert data into their database with CSV files.

4. PostgreSQL Database Management System PL/pgSQL Components

In previous sections we described how to build a physical database with PostgreSQL as well as some of the most very basic operations that could be performed on the data within the database. However, when adhering to integrity constraints as well as business constraints, we will need some more complex operations to manipulate data within a database.

In phase four we will explore how to implement complex operations using the PostgreSQL procedural extension of SQL which is called PL/pgSQL. First we will introduce the purpose of the procedural language as well as the benefits of using PL/pgSQL. Then we will explore some of the key features of PL/pgSQL along with the syntax for utilizing each feature correctly. After that we will provide some example procedural operations using our database. And finally we will explore SQL extensions that are provided by other popular Database Management Systems such as Oracle or MySQL.

4.1 PostgreSQL PL/pgSQL

The procedural language extension of SQL for PostgreSQL is PL/pgSQL. It uses *flow control structures* such as if statements, case statements, and loops to define the order in which statements are executed. This procedural language extension is used to build precompiled blocks of PL/pgSQL code that can be run at any time called stored procedures or stored functions (User defined functions).

Using stored procedures in a database application has several advantages over manually writing PL/pgSQL blocks of code and sending them to the server. The first advantage is that stored procedures and functions are precompiled, this means that the PL/pgSQL does not need to be compiled every time it is run, therefore saving time during execution. Another advantage is that these stored procedures are reusable or in other words they condense complex operations into a single function or functions that can be repeatedly used by other users. The final advantage is the fact that stored procedures hide complex functionality of this procedural extension from the users which will make developing code much easier and faster.

4.1.1 Program Structure and Control Statements

The PL/pgSQL language is divided into collections of different statements called *blocks.* Each block has its own corresponding local variables as well as scope. Blocks can either be unnamed (anonymous) or named and named blocks can be either procedures or functions. The syntax for an anonymous block is below.

Anonymous Block Syntax:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements;
...
END [ label ];
```

Another advantage of the PL/pgSQL language is a feature called *flow control structures*. They control the order in which statements are executed. Flow control structures include conditional statements (“if” statements, case statements) and different types of loops (*for loops, while loops, and basic loops*). The syntax for these statements are described below.

”IF” Statements Syntax:

```
IF condition THEN
  statements;
ELSE
```

```
  alternative-statements;
END IF;
```

Case Statements Syntax:

```
CASE search-expression
   WHEN expression_1 [, expression_2, ...] THEN
      when-statements
  [ ... ]
  [ELSE
      else-statements ]
END CASE;
```

Basic Loop Syntax:

```
<<label>>
LOOP
   statements;
   EXIT [<<label>>] WHEN condition;
END LOOP;
```

“While” Loop Syntax:

```
<<label>>
WHILE condition LOOP
   statements;
END LOOP;
```

“For” Loop Syntax (Integer Variant):

```
<<label>>
   FOR loop_counter IN [ REVERSE ] from.. to [ BY
   expression ] LOOP
   statements;
END LOOP [ label ];
```

4.1.2 Stored Procedures

A stored procedure is defined as a group of SQL statements that perform a given task. The procedure encapsulates a set of queries or operations to be executed on a database. In Oracle PL/SQL stored procedures are slightly different than stored functions in that stored procedures do not return values but stored functions do. However in PostgreSQL, there are only “functions”, which are referred to as *User-Defined Functions* (UDF’s). Since there are only functions in PL/pgSQL, they have all of the same key features as procedures from different database systems such as preventing SQL injections. This is done by checking the type of parameter to make sure it matches the type of the stored procedure when parameters are passed to a stored procedure.

Syntax for Stored Procedures:

```
CREATE FUNCTION function_name(p1 type, p2 type)
    RETURNS type AS
        BEGIN
        -- logic
        END;
    LANGUAGE language_name;
```

4.1.3 Stored Functions

In PostgreSQL “functions” are referred to as *User-Defined Functions* or UDF’s. As you can see the syntax for stored procedures and stored functions is virtually identical. Rather than just simply modifying or altering the data in the database, the purpose of stored functions is to calculate something and return a value.

Syntax for Stored Functions:

```
CREATE FUNCTION function_name(p1 type, p2 type)
    RETURNS type AS
        BEGIN
        -- logic
        END;
    LANGUAGE language_name;
```

4.1.4 Packages

There are no packages in PL/pgSQL. This is one of the differences between Oracle PL/SQL and PostgreSQL PL/pgSQL. Although both languages are block structured and imperative, functions are organized into groups using schemas rather than packages in PL/pgSQL. Since there are no packages, this means there are also no package-level variables, but, you can keep per-session states in a temporary table instead.

4.1.5 Triggers

In PL/pgSQL, a trigger is a stored procedure that is associated with a table, view, schema, or the actual database itself. Triggers are set to automatically execute whenever a specific event occurs and they can be executed either before, after, or in place of this specific event. A very common example of where a trigger can be utilized is with a cascade delete operation. A cascade delete operation is the idea that whenever a tuple, t, is deleted, any other tuples that references that tuple, t, must be deleted first. With this being the case, a cascade operation is required. Some cascade operations may be implemented to delete multiple levels of tuples.

Syntax for Triggers:

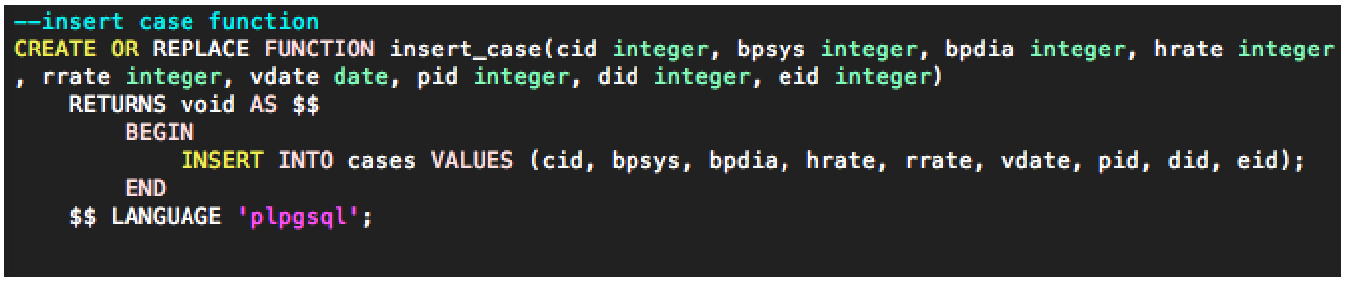
```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD
OF} {event [OR ...]}
   ON table_name
   [ FROM referenced_table_name ]
   [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY
IMMEDIATE | INITIALLY DEFERRED } ]
   [ FOR [ EACH ] { ROW | STATEMENT } ]
   [ WHEN ( condition ) ]
   EXECUTE PROCEDURE function_name ( arguments )
```

4.2 PostgreSQL PL/pgSQL Subprogram Examples

In the following section we will explore some subprograms of PL/pgSQL. Since PL/pgSQL does not utilize packages, the method in which it groups functions is by schema. Therefore we will list and define three procedures/functions as well as three triggers for our database. The functions include a function to insert a case, one that will delete a patient, and a third function that will update a case.

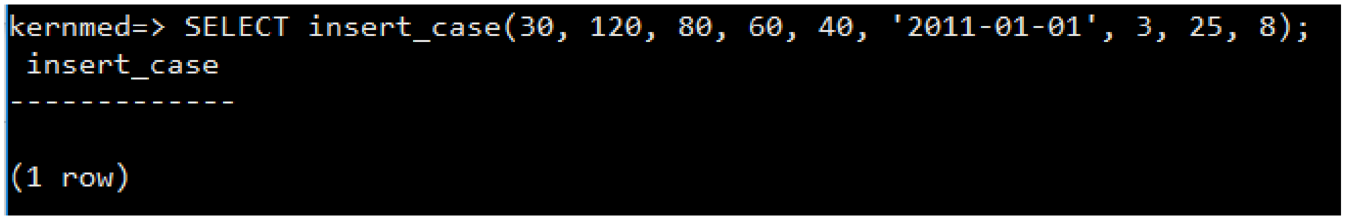
Insert Procedure Definition:

This procedure will insert a case into the database using specified parameters. When this procedure executes, each and every attribute from the “cases” table will be passed in as parameters.



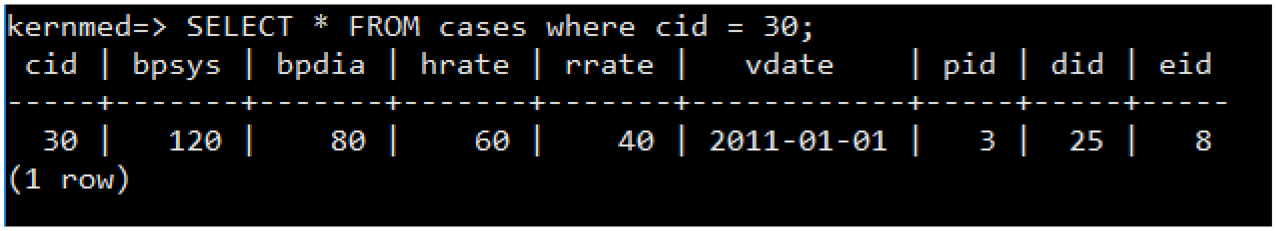
```
--insert case function
CREATE OR REPLACE FUNCTION insert_case(cid integer, bpsys integer, bpdia integer, hrate integer
, rrate integer, vdate date, pid integer, did integer, eid integer)
    RETURNS void AS $$
        BEGIN
            INSERT INTO cases VALUES (cid, bpsys, bpdia, hrate, rrate, vdate, pid, did, eid);
        END
    $$ LANGUAGE 'plpgsql';
```

Insert Procedure Execution:



```
kernmed=> SELECT insert_case(30, 120, 80, 60, 40, '2011-01-01', 3, 25, 8);
 insert_case
-------------

(1 row)
```

Insert Procedure Results:

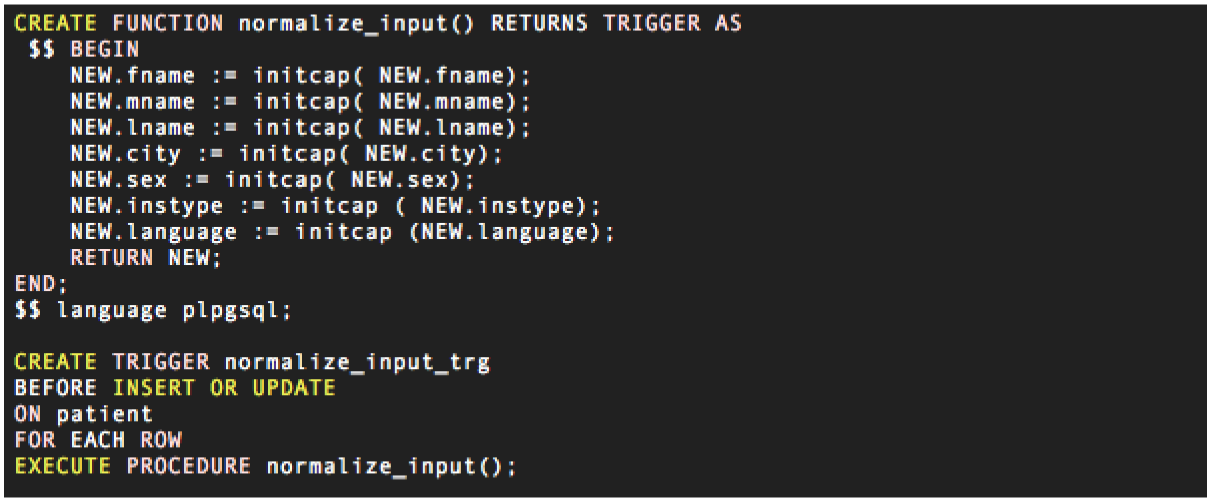


```
kernmed=> SELECT * FROM cases where cid = 30;
 cid | bpsys | bpdia | hrate | rrate |    vdate    | pid | did | eid
-----+-------+-------+-------+-------+------------+-----+-----+-----
  30 |   120 |    80 |    60 |    40 | 2011-01-01 |   3 |  25 |   8
(1 row)
```

After the Insert procedure executes successfully, it will insert the case into the next row with the specified cid.

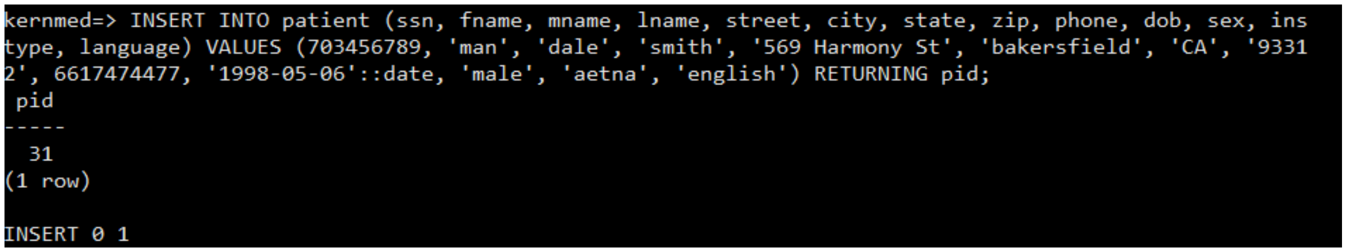
Insert/Update Patient Trigger Definition:

To ensure our database is clean and the data input into it is easy to read, a procedure and trigger are created to normalize user input when adding new patients to the database. This trigger will ensure that tuples input have a single first capital letter followed by lower case letters so our data is neat and readable for standard users.



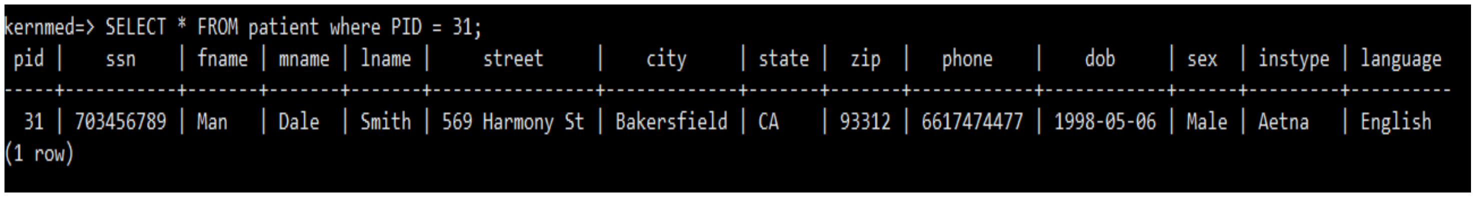
```
CREATE FUNCTION normalize_input() RETURNS TRIGGER AS
 $$ BEGIN
    NEW.fname := initcap( NEW.fname);
    NEW.mname := initcap( NEW.mname);
    NEW.lname := initcap( NEW.lname);
    NEW.city := initcap( NEW.city);
    NEW.sex := initcap( NEW.sex);
    NEW.instype := initcap ( NEW.instype);
    NEW.language := initcap (NEW.language);
    RETURN NEW;
END;
$$ language plpgsql;

CREATE TRIGGER normalize_input_trg
BEFORE INSERT OR UPDATE
ON patient
FOR EACH ROW
EXECUTE PROCEDURE normalize_input();
```

Insert/Update Patient Trigger Execution:



```
kernmed=> INSERT INTO patient (ssn, fname, mname, lname, street, city, state, zip, phone, dob, sex, ins
type, language) VALUES (703456789, 'man', 'dale', 'smith', '569 Harmony St', 'bakersfield', 'CA', '9331
2', 6617474477, '1998-05-06'::date, 'male', 'aetna', 'english') RETURNING pid;
 pid
-----
  31
(1 row)

INSERT 0 1
```

Insert/Update Patient Trigger Results:

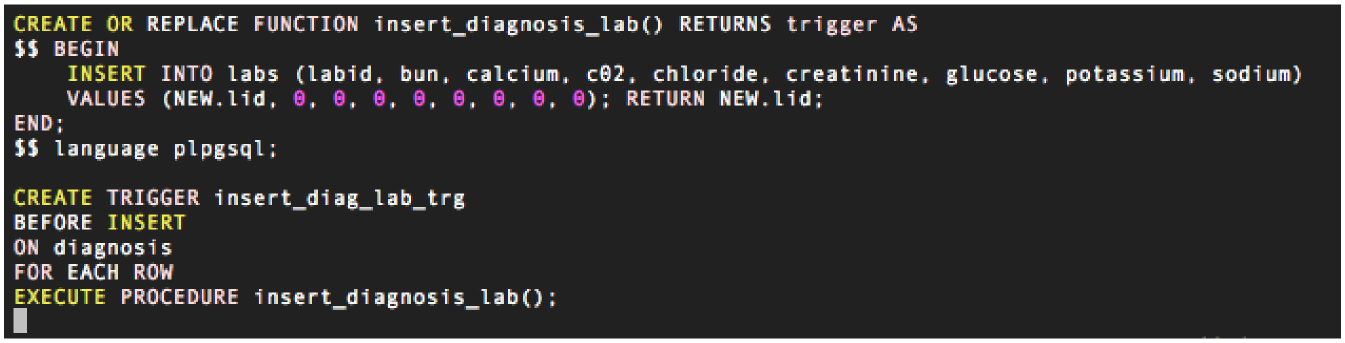


```
kernmed=> SELECT * FROM patient where PID = 31;
 pid |    ssn    | fname | mname | lname |     street     |    city     | state |  zip  |   phone    |    dob     | sex  | instype | language
-----+-----------+-------+-------+-------+----------------+-------------+-------+-------+------------+------------+------+---------+----------
  31 | 703456789 | Man   | Dale  | Smith | 569 Harmony St | Bakersfield | CA    | 93312 | 6617474477 | 1998-05-06 | Male | Aetna   | English
(1 row)
```

After the Insert/Update Patient Trigger is executed successfully, you can see that each tuple within the row has capitalization on the first letter of the string within the tuple followed by lowercase. Above I inserted the information in the database without capitalization. This should aid in keeping the database clean and neat.

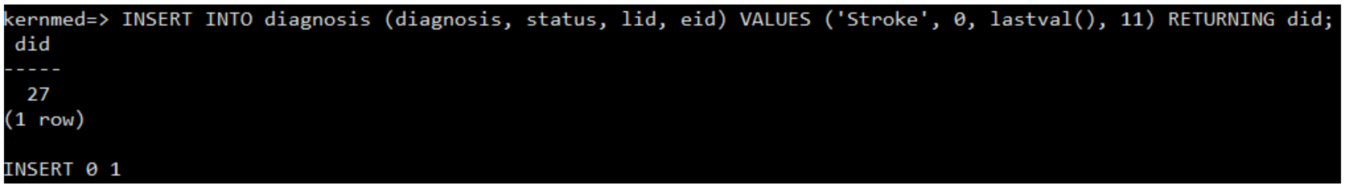
Insert Diagnosis Trigger Definition:

When creating a new diagnosis for our database, the user will manually assign the employee id to each record but the corresponding lab tuple should be created at the same time. For this reason the Insert Diagnosis trigger is used, which will fire before the Insert occurs. It will create a lab with default 0 values and link that lab to the newly inserted diagnosis the user defines.



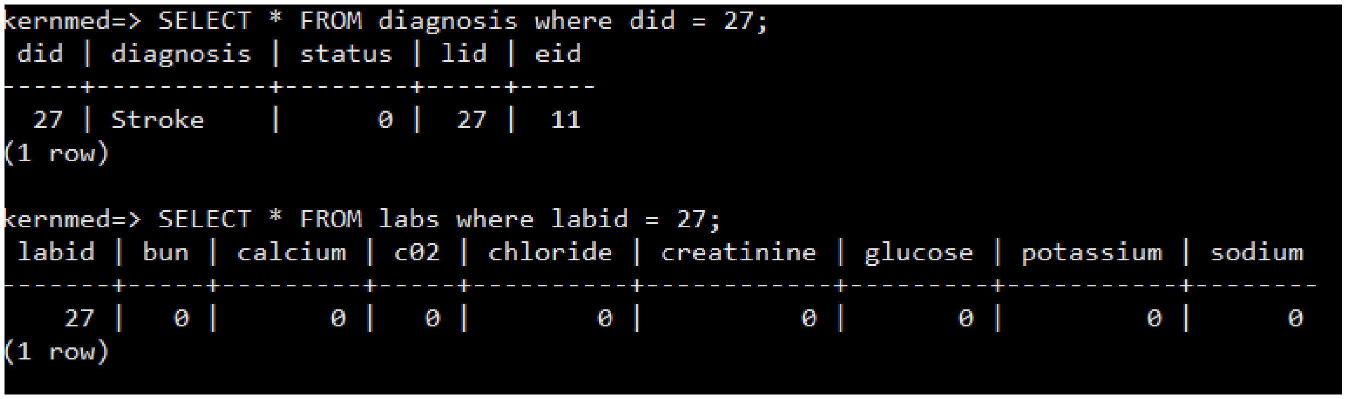
```
CREATE OR REPLACE FUNCTION insert_diagnosis_lab() RETURNS trigger AS
$$ BEGIN
    INSERT INTO labs (labid, bun, calcium, c02, chloride, creatinine, glucose, potassium, sodium)
    VALUES (NEW.lid, 0, 0, 0, 0, 0, 0, 0, 0); RETURN NEW.lid;
END;
$$ language plpgsql;

CREATE TRIGGER insert_diag_lab_trg
BEFORE INSERT
ON diagnosis
FOR EACH ROW
EXECUTE PROCEDURE insert_diagnosis_lab();
```

Insert Diagnosis Trigger Execution:



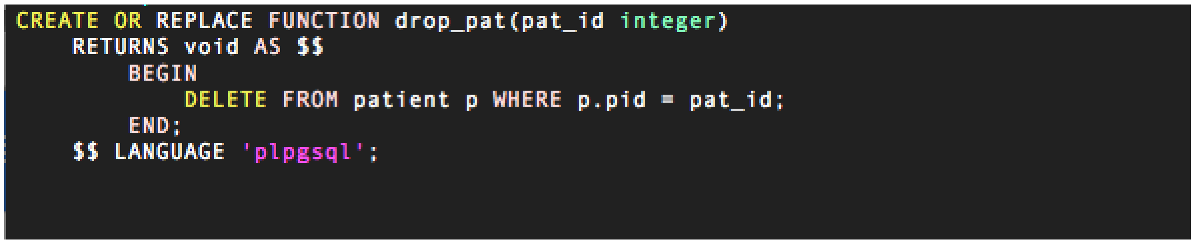
```
kernmed=> INSERT INTO diagnosis (diagnosis, status, lid, eid) VALUES ('Stroke', 0, lastval(), 11) RETURNING did;
 did
-----
  27
(1 row)

INSERT 0 1
```

Insert Diagnosis Trigger Results:

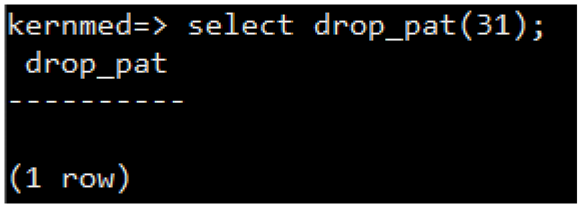

```
kernmed=> SELECT * FROM diagnosis where did = 27;
 did | diagnosis | status | lid | eid
-----+-----------+--------+-----+-----
  27 | Stroke    |      0 |  27 |  11
(1 row)

kernmed=> SELECT * FROM labs where labid = 27;
 labid | bun | calcium | c02 | chloride | creatinine | glucose | potassium | sodium
-------+-----+---------+-----+----------+------------+---------+-----------+--------
    27 |   0 |       0 |   0 |        0 |          0 |       0 |         0 |      0
(1 row)
```
After inserting a diagnosis, the trigger creates the blank lab record. Using the lastval() function in the insert statement, the lid inserted into the diagnosis table is the labid of the newly created tuple in the lab table, correctly linking them all together.

Delete Procedure Definition:

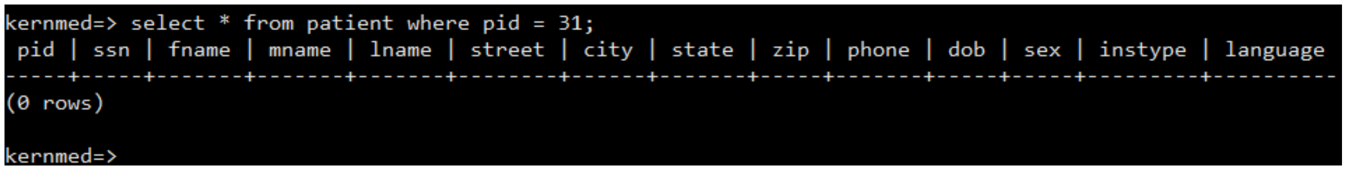
The delete procedure simply drops a patient from the database. It accepts the patient id number to remove the patient from the database completely.


```
CREATE OR REPLACE FUNCTION drop_pat(pat_id integer)
    RETURNS void AS $$
        BEGIN
            DELETE FROM patient p WHERE p.pid = pat_id;
        END;
    $$ LANGUAGE 'plpgsql';
```

Delete Procedure Execution:

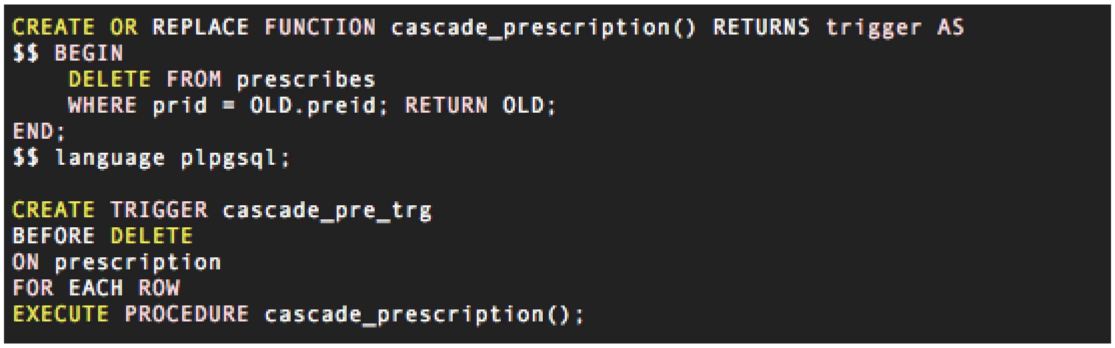

```
kernmed=> select drop_pat(31);
 drop_pat
----------

(1 row)
```

Delete Procedure Results:


```
kernmed=> select * from patient where pid = 31;
 pid | ssn | fname | mname | lname | street | city | state | zip | phone | dob | sex | instype | language
-----+-----+-------+-------+-------+--------+------+-------+-----+-------+-----+-----+---------+----------
(0 rows)

kernmed=>
```
As can be seen above, the patient id is input and removes the patient quickly and efficiently.

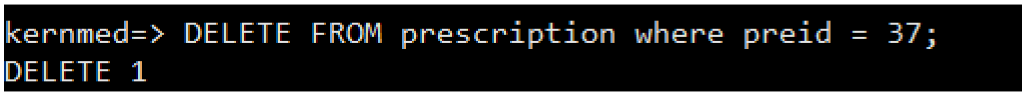
Delete Prescription Trigger Definition:

This trigger/procedure combination works when a user deletes a prescription from the database. In turn, this action also deletes the associated tuple from the prescribles table using a cascade delete to ensure that foreign key constraints are properly met.



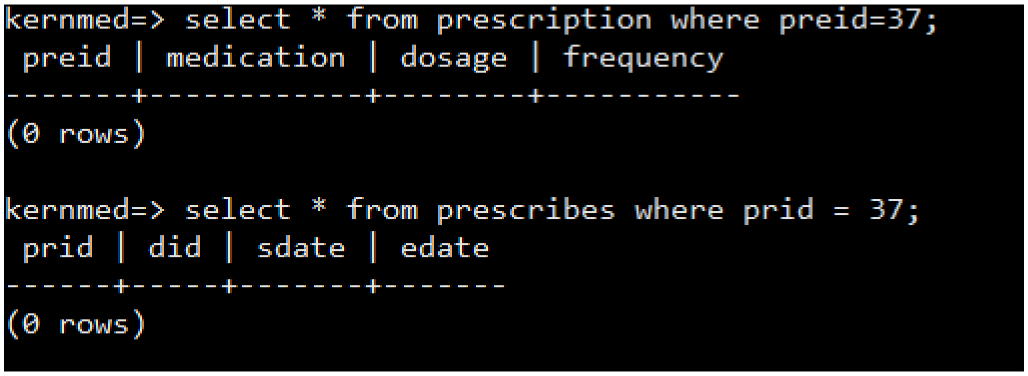
```
CREATE OR REPLACE FUNCTION cascade_prescription() RETURNS trigger AS
$$ BEGIN
    DELETE FROM prescribes
    WHERE prid = OLD.preid; RETURN OLD;
END;
$$ language plpgsql;

CREATE TRIGGER cascade_pre_trg
BEFORE DELETE
ON prescription
FOR EACH ROW
EXECUTE PROCEDURE cascade_prescription();
```

Delete Prescription Trigger Execution:



```
kernmed=> DELETE FROM prescription where preid = 37;
DELETE 1
```

Delete Prescription Trigger Results:

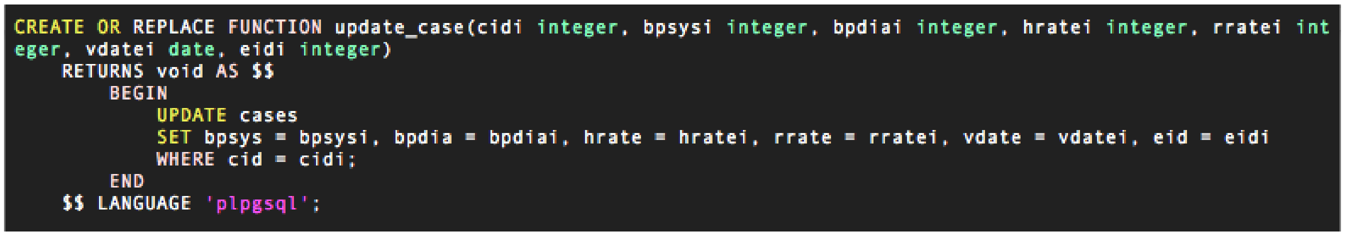


```
kernmed=> select * from prescription where preid=37;
 preid | medication | dosage | frequency
-------+------------+--------+-----------
(0 rows)

kernmed=> select * from prescribes where prid = 37;
 prid | did | sdate | edate
------+-----+-------+-------
(0 rows)
```

After executing the Delete Prescription trigger, the prescribes table is referenced by the prescription id number and deleted before the deletion of the record on the prescription table. This ensures that foreign key constraints are met.

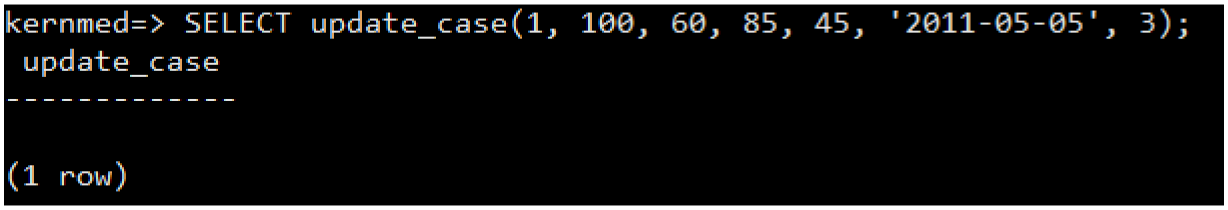
Update Procedure Definition:

The Update Procedure is used to update values within a case. This procedure uses each and every attribute from the cases table passed in via parameters.



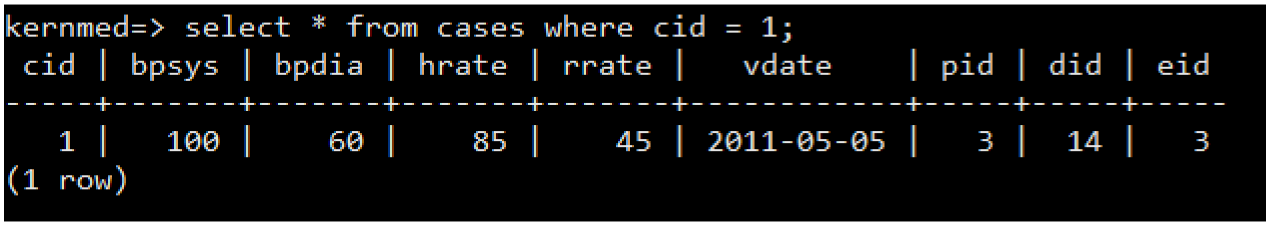
```
CREATE OR REPLACE FUNCTION update_case(cidi integer, bpsysi integer, bpdiai integer, hratei integer, rratei int
eger, vdatei date, eidi integer)
    RETURNS void AS $$
        BEGIN
            UPDATE cases
            SET bpsys = bpsysi, bpdia = bpdiai, hrate = hratei, rrate = rratei, vdate = vdatei, eid = eidi
            WHERE cid = cidi;
        END
    $$ LANGUAGE 'plpgsql';
```

Update Procedure Execution:



```
kernmed=> SELECT update_case(1, 100, 60, 85, 45, '2011-05-05', 3);
 update_case
-------------

(1 row)
```

Update Procedure Results:



```
kernmed=> select * from cases where cid = 1;
 cid | bpsys | bpdia | hrate | rrate |   vdate    | pid | did | eid
-----+-------+-------+-------+-------+------------+-----+-----+-----
   1 |   100 |    60 |    85 |    45 | 2011-05-05 |   3 |  14 |   3
(1 row)
```

After execution the case has been updated to the values specified by the user.

4.3 PL/pgSQL Like Tools (Oracle, Microsoft SQL Server, and MySQL)

PL/pgSQL was used for this project to implement the physical database for Kern Medical. Although PostgreSQL's commercial DBMS was used in this project, there are many other popular database management systems that each offer unique stored procedure functionality. In this section, we will explore Oracle PL/SQL, Microsoft SQL Server T-SQL (Transact-SQL), and MySQL as well as compare the stored procedure functionality and syntax.

Microsoft SQL Server T-SQL

Comparison to other commercial DBMS languages:

This commercial DBMS offers some unique functionality compared to other database management systems. You can restrict user permissions (*with* clause) as well as encrypt the text for stored procedures. Another unique feature is that T-SQL allows for multiple *try-catch* blocks in a procedure, where in PL/SQL and MySQL both use a separate section for handling these types of exceptions. T-SQL functions can also easily return both tables as well as scalar values which cannot be done in MySQL and can be done in PL/SQL by using *pipelined table functions*, but it is not easy). Microsoft's DBMS also offers a unique method for passing as well as using parameters; by forcing the '@' character to precede all parameters. MySQL and Oracle do not use this feature.

Microsoft SQL Server's DBMS, T-SQL, offers a lot of unique functionality, but it also lacks some key features of other DBMS's. Unlike PL/SQL, T-SQL does not offer *for* loops but only *basic* loops and *while* loops. There is also no grouping of procedure into packages in T-SQL.

Syntax for creating procedures/functions:

```
CREATE { PROC | PROCEDURE } [schema_name.] procedure_name [
; number ]
     [ { @parameter [ type_schema_name. ] data_type }
          [ VARYING ] [ = default ] [ OUT | OUTPUT |
[READONLY]
      ] [ ,...n ]
[ WITH <procedure_option> [ ,...n ] ]
[ FOR REPLICATION ]
AS { [ BEGIN ] sql_statement [;] [ ...n ] [ END ] }
```

```
 [;]

<procedure_option> ::=
    [ ENCRYPTION ]
    [ RECOMPILE ]
    [ EXECUTE AS Clause ]

CREATE FUNCTION [ schema_name. ] function_name

( [ { @parameter_name [ AS ][ type_schema_name. ]
parameter_data_type
    [ = default ] [ READONLY ] }
    [ ,...n ]
  ]
)

RETURNS return_data_type
    [ WITH <function_option> [ ,...n ] ]
    [ AS ]
    BEGIN
        function_body
        RETURN scalar_expression
    END
[;]
```

Syntax for *basic* loop:

```
WHILE @cnt < cnt_total
BEGIN
    {...statements...}
    SET @cnt = @cnt + 1;
END;
```

MySQL

Comparison to other commercial DBMS languages:

MySQL offers many of the same features and functionality as PL/SQL and T-SQL but it is also missing a few of the features. Similarly to T-SQL, MySQL does not offer *for* loops but rather only *basic* loops and *while* loops. MySQL also does not offer packages for namespace management like Oracle PL/SQL does. But parameters are passed in the same way as PL/SQL rather than using the '@' character like T-SQL. One key aspect of MySQL is to know that when creating a procedure, the *delimiter* command *must* be used to change the default end-line character from (;) to (//); if this is not done then only the first line of the procedure will be stored.

Syntax for creating procedures/functions:

```
CREATE
    [DEFINER = { user | CURRENT_USER }]
    PROCEDURE sp_name ([proc_parameter[,...]])
    [characteristic ...] routine_body

CREATE
    [DEFINER = { user | CURRENT_USER }]
    FUNCTION sp_name ([func_parameter[,...]])
    RETURNS type
    [characteristic ...] routine_body
```

Syntax for *basic* loop:

```
[begin_label:] WHILE search_condition DO
    statement_list
END WHILE [end_label]
```

Oracle PL/SQL

Comparison to other commercial DBMS languages:

The procedural SQL-based language which Oracle's DBMS utilizes is called PL/SQL. This procedural language provides many of the same features as Postgres' procedural SQL-based language, PL/pgSQL. One of the differences between PL/SQL and PL/pgSQL is packages. Oracle uses packages to prevent name conflicts and provide a way to group functions together. It also offers more complex control structures like to *for* loop (this feature is also offered in PL/pgSQL but not in other procedural SQL-based languages such as MySQL and T-SQL). The method in which parameters are passed is, for the most part, the same as MySQL.

Syntax for procedures/functions:

```
create [or replace] procedure <procedure name> (
    <parameter list>
)
return <return type>
as
    <declarations>
begin
    <executable section>
end;
```

Syntax for *basic* loop:

```
loop
    <PL/SQL statements>
end loop;

for <counter variable> in <range> loop
    <PL/SQL statements>
end loop;


while <condition> loop
    <PL/SQL statements>
end loop;
```

5 Graphical User Interface Implementation

In the final phase, we will implement a database application which includes a graphical user interface (GUI). The first task will be listing all of the different types of user groups who will be accessing the application as well as each of their special needs. Then, we will go into depth about the Postgres PL/pgSQL features which are required to implement the software application, including views and stored procedures. The next step will be to provide screenshots corresponding to all of the features in a component of the application for *one* of the different user groups due to hefty time consumption. After this we will describe some of the implementation processes for the application complete with screenshots as well. Lastly, we will provide a detailed overview of the entire database implementation process from start to finish.

5.1 Daily User Activities

When developing a front-end application for the database, it is important to remember that the application will be accessed by a variety of users who each have different needs and expectations from the software. Each user group will require a different graphical user interface that provides the functionalities they need. We will now describe the user groups for the Kern Medical database and the unique required functionalities.

5.1.1 Nurse Users

The nurse will assess patients upon initial arrival and collect their vitals such as blood pressure, heart rate and respiratory rate. They will also assign a doctor to the patient and view general data.

Required Functionalities:

- View patients
- Record vitals and admission information
- Assign doctor to patients

5.1.2 Doctor Users

The doctor will diagnose the patient with a sickness and record it within the case. Doctors will also write prescriptions for patients and assess lab values through charts and other various reports.

Required Functionalities

- View assigned patients
- Record Diagnoses on each patient
- Assign Prescriptions
- View reports and other general data

5.1.3 Executive/Administrative Users

Executive and Administrative users of the database will ensure that the database accurately reflects the hospital and its status. Executive users will view usage reports and cost/profit analysis to make financial decisions. They will also make human resource decisions.

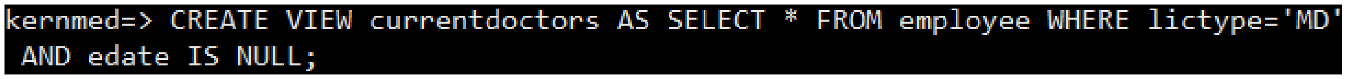
Required Functionalities

- View hospital resource reports
- Manage all patient and employee information
- Analyze income reports for financial decisions

5.2 Relations, Views, and Subprograms

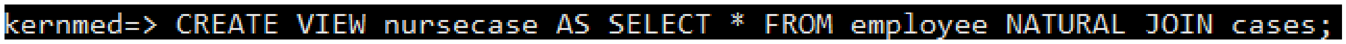
To provide the services required of our user groups, the database application will utilize some of the functionalities provided by postgreSQL such as views and procedures. The purpose of our views and procedures are listed below.

View: Current Doctors
This view displays all information pertaining to currently hired doctors from the employee table. This view is very useful because it allows us to easily populate our lists of available doctors on the application and filter out those doctors who are no longer working at the hospital.



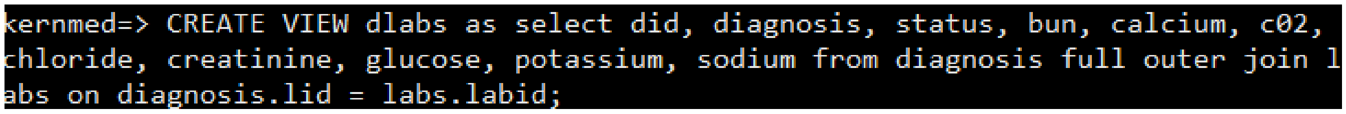
```
kernmed=> CREATE VIEW currentdoctors AS SELECT * FROM employee WHERE lictype='MD'
 AND edate IS NULL;
```

View: Nurse Cases
The Nurse cases view displays information which links each nurse to their corresponding assessments completed. This is valuable and aids greatly in determining who performed an assessment on a patient.



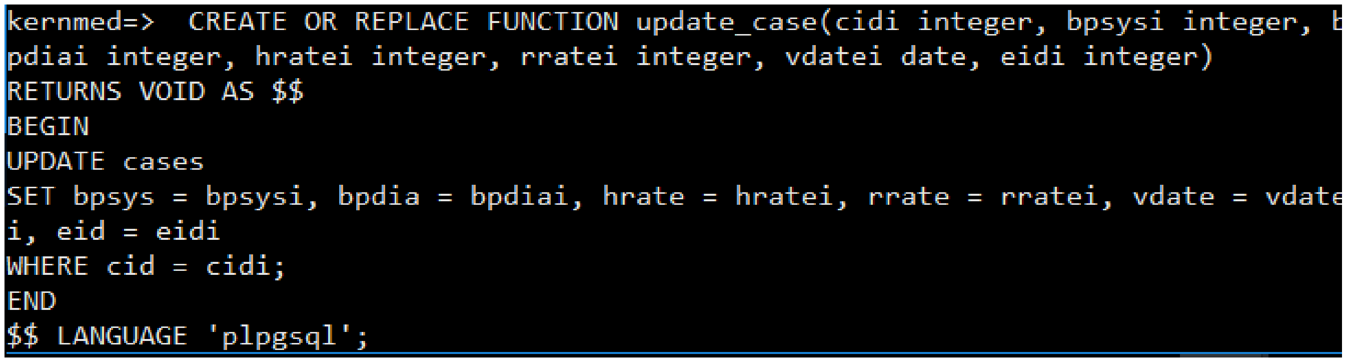
```
kernmed=> CREATE VIEW nursecase AS SELECT * FROM employee NATURAL JOIN cases;
```

View: Diagnosis Labs
This view displays the information linking each lab to their corresponding diagnosis which aids greatly in the development of the front end database application.



```
kernmed=> CREATE VIEW dlabs as select did, diagnosis, status, bun, calcium, c02,
chloride, creatinine, glucose, potassium, sodium from diagnosis full outer join l
abs on diagnosis.lid = labs.labid;
```

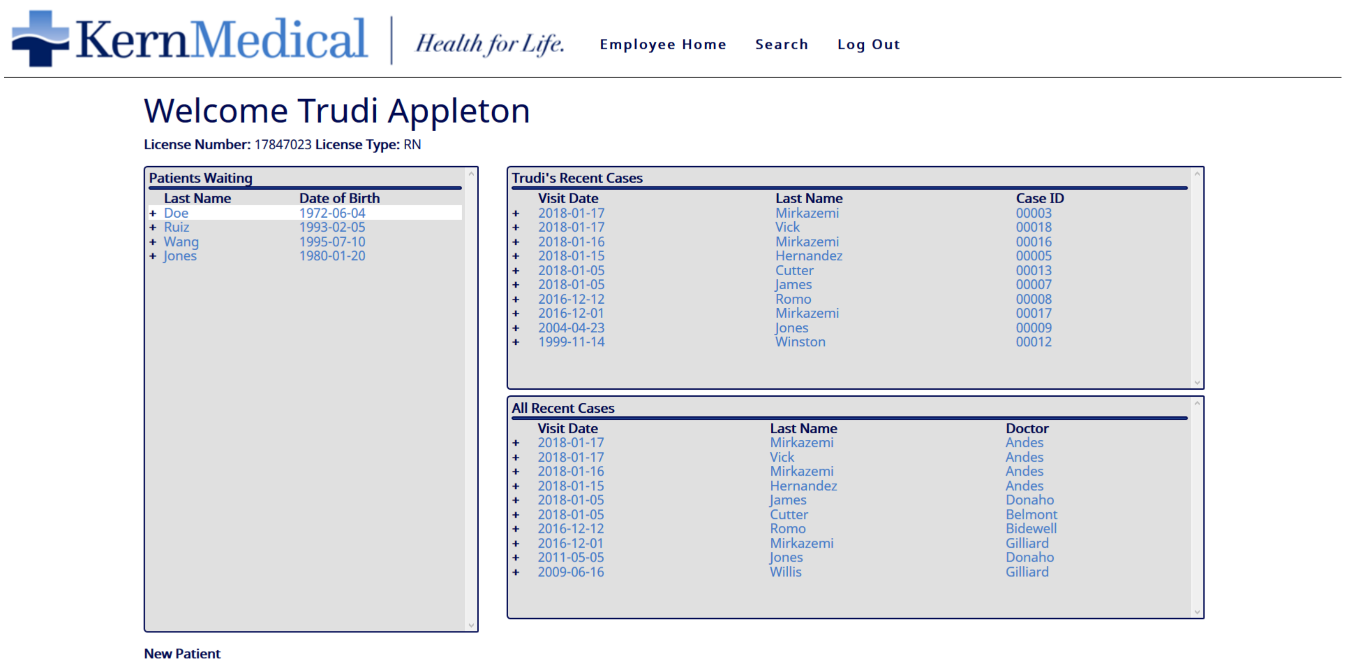
Stored Procedure: Update Case
This procedure allows us to update case records quickly and efficiently with our front end application. This is useful because it helps in reducing errors and miscalculation when modifying data manually.



```
kernmed=>  CREATE OR REPLACE FUNCTION update_case(cidi integer, bpsysi integer, b
pdiai integer, hratei integer, rratei integer, vdatei date, eidi integer)
RETURNS VOID AS $$
BEGIN
UPDATE cases
SET bpsys = bpsysi, bpdia = bpdiai, hrate = hratei, rrate = rratei, vdate = vdate
i, eid = eidi
WHERE cid = cidi;
END
$$ LANGUAGE 'plpgsql';
```

5.3 Menus and Displays

Nurse Specific
Upon logging into the application, the nurse is presented with a web page which displays helpful information. The left box signifies patients that are awaiting an assessment, while the top right box signifies patients that specific nurse has already assessed. The bottom right box displays the patients that all nurses have seen recently. This information is displayed with a variety of queries to the database.



Welcome Trudi Appleton

License Number: 17847023 **License Type:** RN

Patients Waiting

	Last Name	Date of Birth
+	Doe	1972-06-04
+	Ruiz	1993-02-05
+	Wang	1995-07-10
+	Jones	1980-01-20

Trudi's Recent Cases

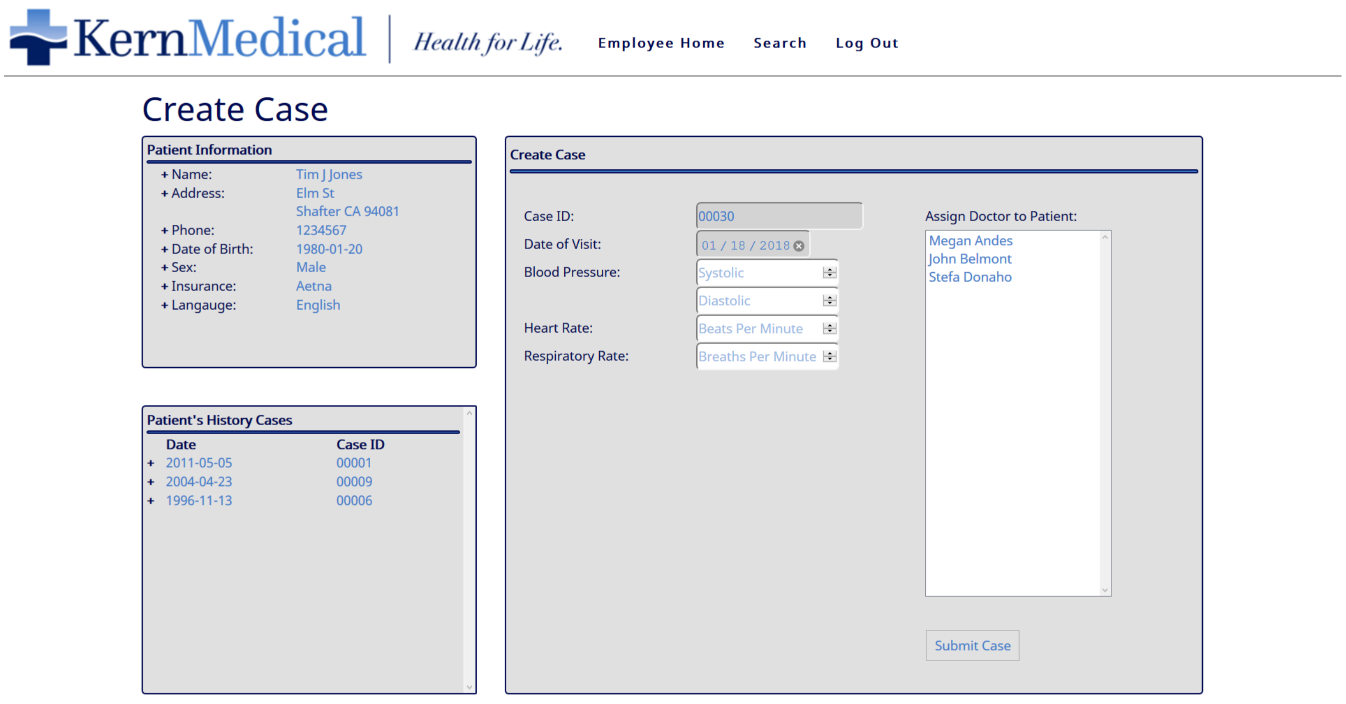
	Visit Date	Last Name	Case ID
+	2018-01-17	Mirkazemi	00003
+	2018-01-17	Vick	00018
+	2018-01-16	Mirkazemi	00016
+	2018-01-15	Hernandez	00005
+	2018-01-05	Cutter	00013
+	2018-01-05	James	00007
+	2016-12-12	Romo	00008
+	2016-12-01	Mirkazemi	00017
+	2004-04-23	Jones	00009
+	1999-11-14	Winston	00012

All Recent Cases

	Visit Date	Last Name	Doctor
+	2018-01-17	Mirkazemi	Andes
+	2018-01-17	Vick	Andes
+	2018-01-16	Mirkazemi	Andes
+	2018-01-15	Hernandez	Andes
+	2018-01-05	James	Donaho
+	2018-01-05	Cutter	Belmont
+	2016-12-12	Romo	Bidewell
+	2016-12-01	Mirkazemi	Gilliard
+	2011-05-05	Jones	Donaho
+	2009-06-16	Willis	Gilliard

New Patient

When a nurse clicks on a name on the left “Patient’s Waiting” table, they are brought to a screen where they can do their initial assessment including adding vitals and assigning a doctor to a patient. Patient’s past cases can also be accessed on this page. Once a nurse has assessed a patient, that patient is moved from the “Patient’s Waiting” box on the main page to the “Recent Cases” box so that the next patient can be assisted.



Create Case

Patient Information

+ Name:	Tim J Jones
+ Address:	Elm St Shafter CA 94081
+ Phone:	1234567
+ Date of Birth:	1980-01-20
+ Sex:	Male
+ Insurance:	Aetna
+ Langauge:	English

Patient's History Cases

	Date	Case ID
+	2011-05-05	00001
+	2004-04-23	00009
+	1996-11-13	00006

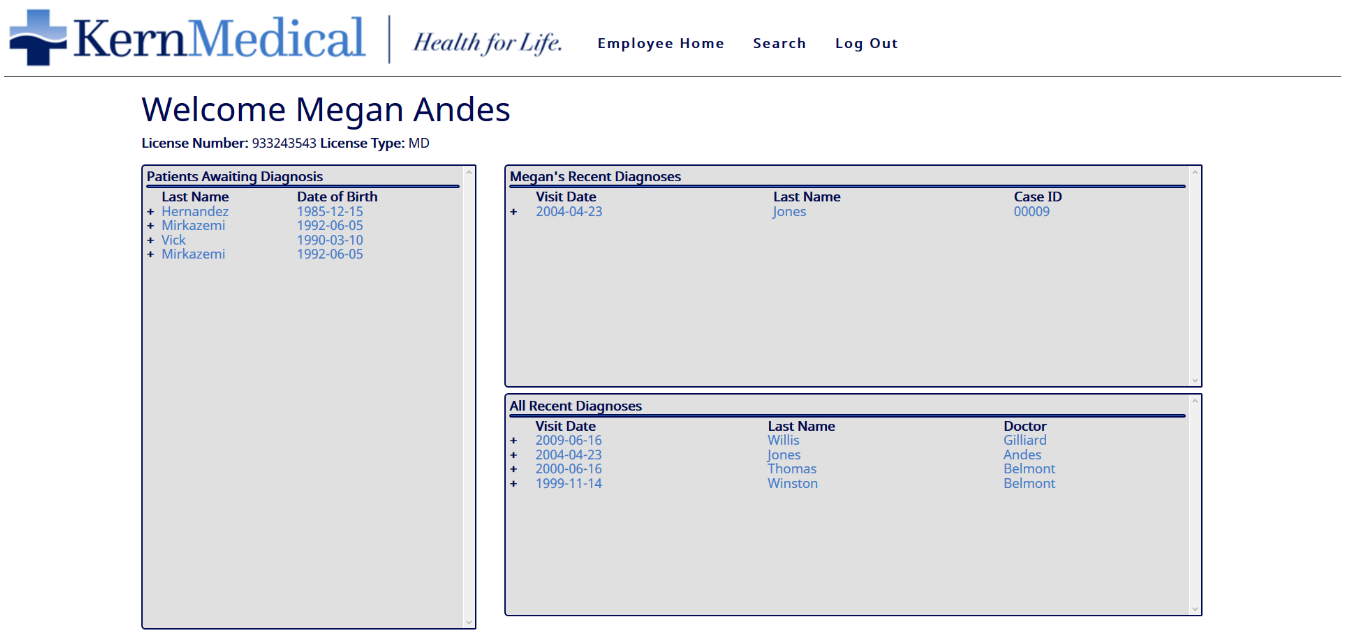
Create Case

Case ID:	00030
Date of Visit:	01 / 18 / 2018
Blood Pressure:	Systolic Diastolic
Heart Rate:	Beats Per Minute
Respiratory Rate:	Breaths Per Minute

Assign Doctor to Patient:

Megan Andes
John Belmont
Stefa Donaho

Doctor Specific
After logging into the application, doctors are greeted with a similar page that the nurse receives which has patients that are awaiting a diagnosis on the left and the doctors recently done diagnoses on the top right and all doctors recently done diagnoses on the bottom right.



Welcome Megan Andes

License Number: 933243543 **License Type:** MD

Patients Awaiting Diagnosis

	Last Name	Date of Birth
+	Hernandez	1985-12-15
+	Mirkazemi	1992-06-05
+	Vick	1990-03-10
+	Mirkazemi	1992-06-05

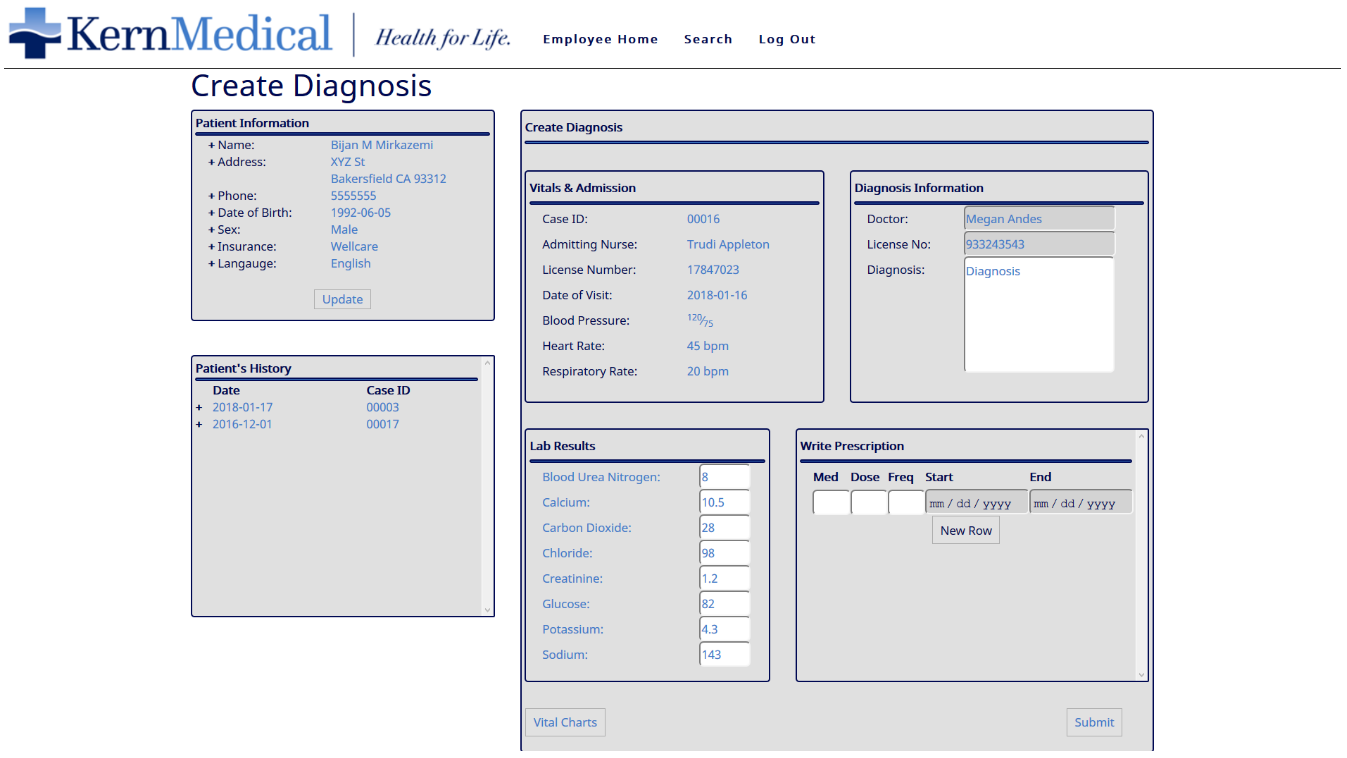
Megan's Recent Diagnoses

	Visit Date	Last Name	Case ID
+	2004-04-23	Jones	00009

All Recent Diagnoses

	Visit Date	Last Name	Doctor
+	2009-06-16	Willis	Gilliard
+	2004-04-23	Jones	Andes
+	2000-06-16	Thomas	Belmont
+	1999-11-14	Winston	Belmont

Once a doctor clicks on a patient on the left to start their diagnosis, they are brought to a screen with information pertaining to their patient. This includes the assessment the nurse gave as well as lab results, a space to write prescriptions and access to charts which show the patient’s vital history (which I will describe in the next section). Once the diagnosis is submitted, the patient will move from the main page’s “Patient’s Awaiting Diagnosis” box to the doctor’s “Recent Diagnoses” box.


KernMedical | Health for Life. Employee Home Search Log Out

Create Diagnosis

Patient Information

+ Name:	Bijan M Mirkazemi
+ Address:	XYZ St Bakersfield CA 93312
+ Phone:	5555555
+ Date of Birth:	1992-06-05
+ Sex:	Male
+ Insurance:	Wellcare
+ Langauge:	English

Update

Patient's History

	Date	Case ID
+	2018-01-17	00003
+	2016-12-01	00017

Create Diagnosis

Vitals & Admission

Case ID:	00016
Admitting Nurse:	Trudi Appleton
License Number:	17847023
Date of Visit:	2018-01-16
Blood Pressure:	$^{120}/_{75}$
Heart Rate:	45 bpm
Respiratory Rate:	20 bpm

Diagnosis Information

Doctor:	Megan Andes
License No:	933243543
Diagnosis:	Diagnosis

Lab Results

Blood Urea Nitrogen:	8
Calcium:	10.5
Carbon Dioxide:	28
Chloride:	98
Creatinine:	1.2
Glucose:	82
Potassium:	4.3
Sodium:	143

Write Prescription

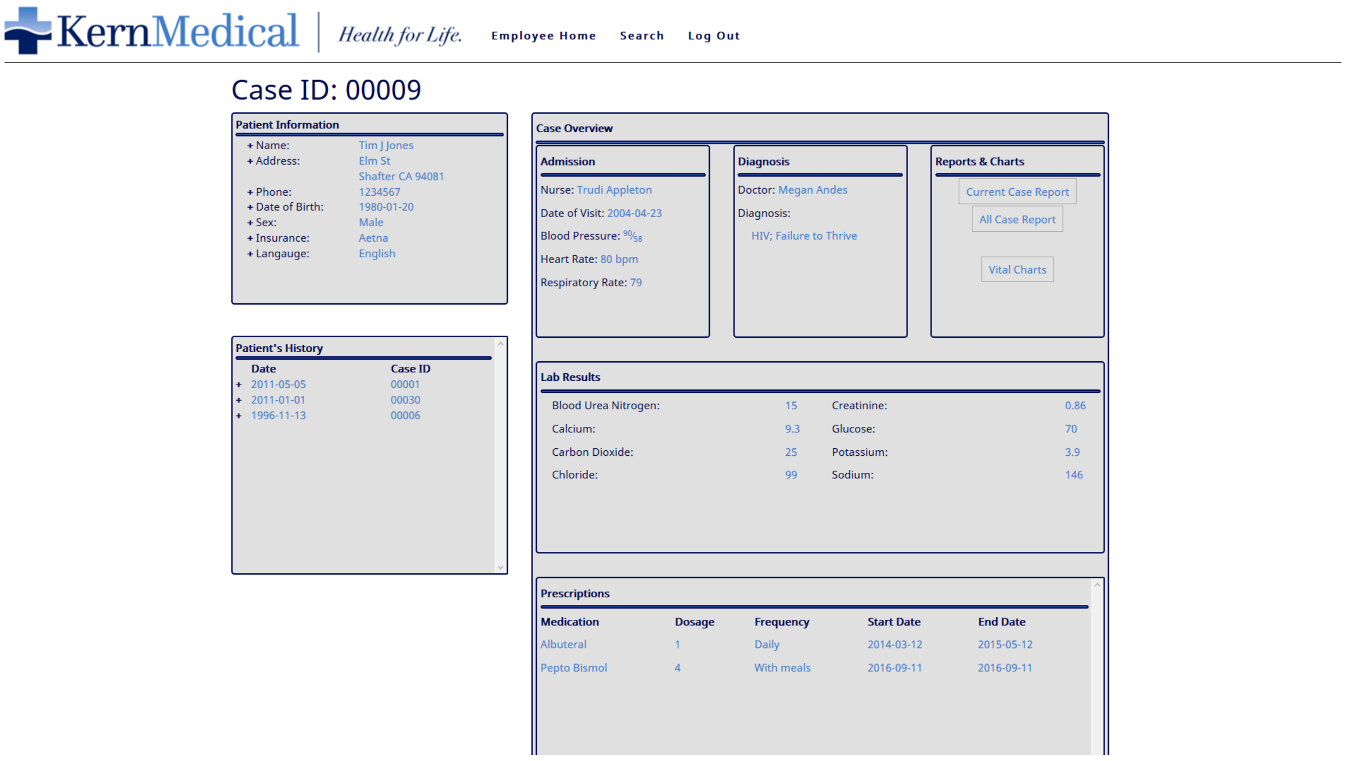
Med	Dose	Freq	Start	End
			mm / dd / yyyy	mm / dd / yyyy

New Row

Vital Charts Submit

Shared Functionality

Both application frontends share similar functionality. Both the doctor and the nurse can access a case page which gives a full overview of an entire case including reports, charts, and all prescriptions.


KernMedical | Health for Life. Employee Home Search Log Out

Case ID: 00009

Patient Information

+ Name:	Tim J Jones
+ Address:	Elm St Shafter CA 94081
+ Phone:	1234567
+ Date of Birth:	1980-01-20
+ Sex:	Male
+ Insurance:	Aetna
+ Langauge:	English

Patient's History

	Date	Case ID
+	2011-05-05	00001
+	2011-01-01	00030
+	1996-11-13	00006

Case Overview

Admission

Nurse: Trudi Appleton
Date of Visit: 2004-04-23
Blood Pressure: $^{90}/_{58}$
Heart Rate: 80 bpm
Respiratory Rate: 79

Diagnosis

Doctor: Megan Andes
Diagnosis:
HIV; Failure to Thrive

Reports & Charts

Current Case Report
All Case Report
Vital Charts

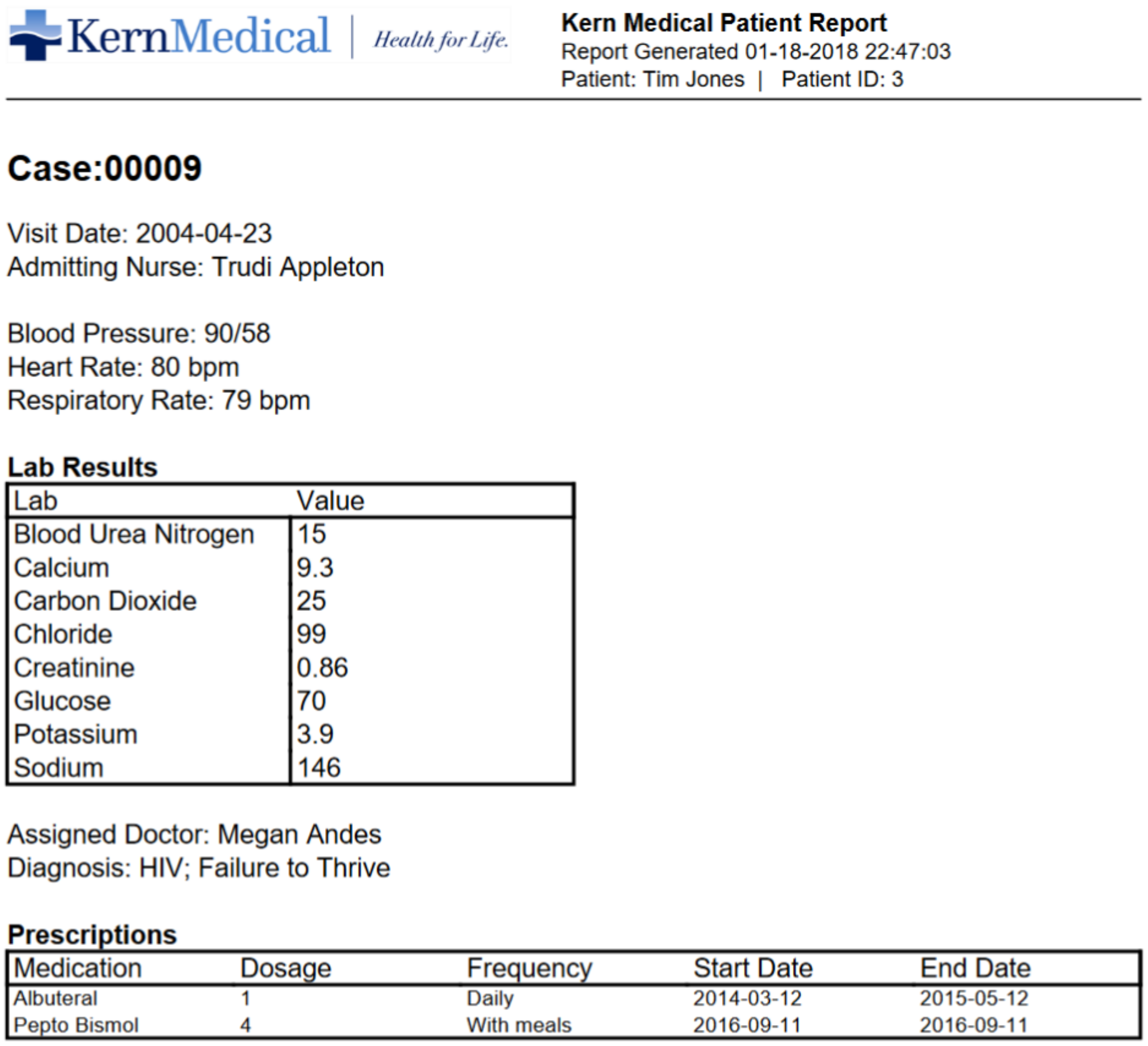
Lab Results

Blood Urea Nitrogen:	15	Creatinine:	0.86
Calcium:	9.3	Glucose:	70
Carbon Dioxide:	25	Potassium:	3.9
Chloride:	99	Sodium:	146

Prescriptions

Medication	Dosage	Frequency	Start Date	End Date
Albuteral	1	Daily	2014-03-12	2015-05-12
Pepto Bismol	4	With meals	2016-09-11	2016-09-11

Two types of reports can be generated, one for an individual case and one for all cases. The report has a general synopsis of the case which can be printed out for record keeping purposes.



KernMedical | Health for Life.

Kern Medical Patient Report
Report Generated 01-18-2018 22:47:03
Patient: Tim Jones | Patient ID: 3

Case:00009

Visit Date: 2004-04-23
Admitting Nurse: Trudi Appleton

Blood Pressure: 90/58
Heart Rate: 80 bpm
Respiratory Rate: 79 bpm

Lab Results

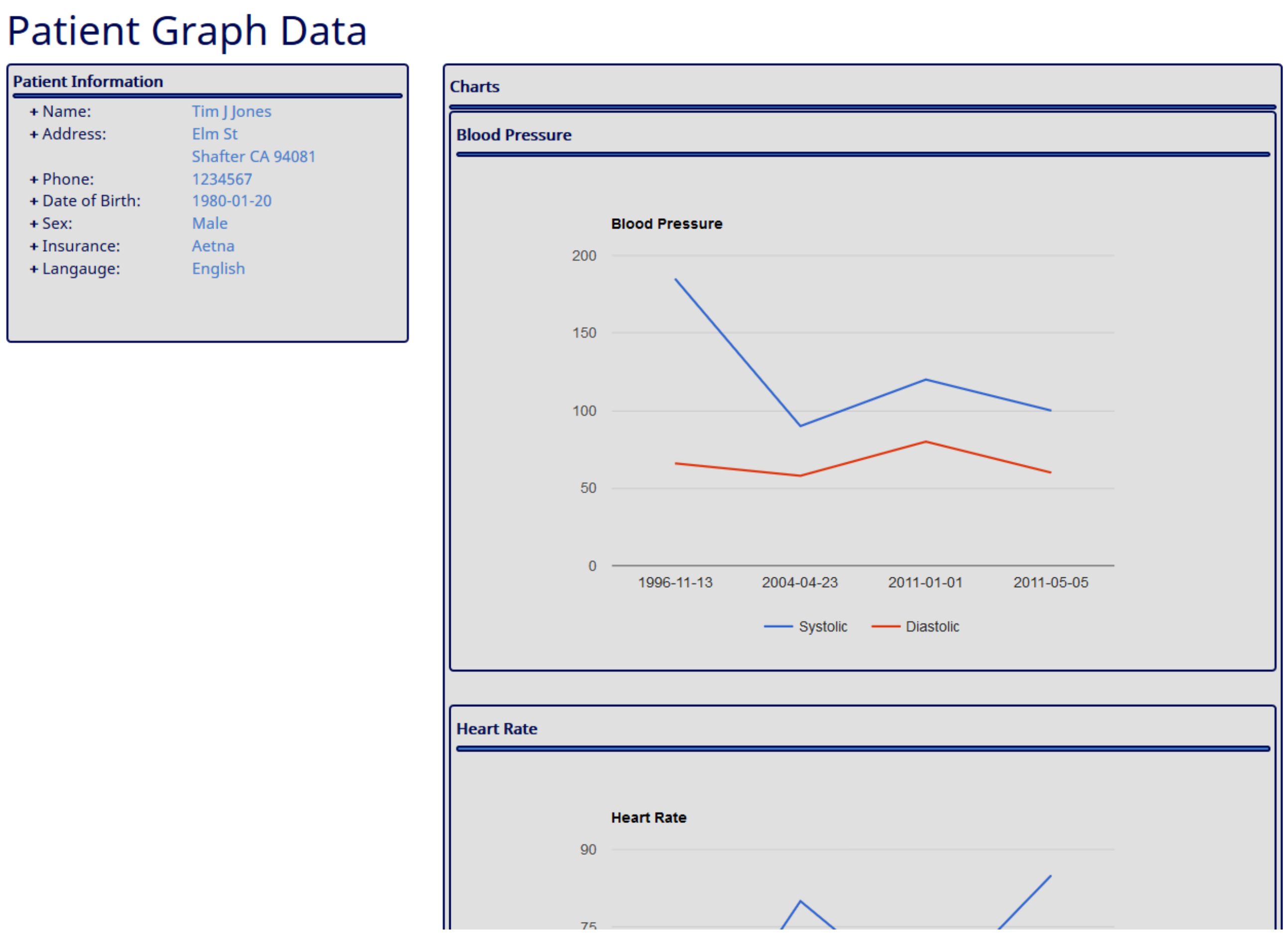
Lab	Value
Blood Urea Nitrogen	15
Calcium	9.3
Carbon Dioxide	25
Chloride	99
Creatinine	0.86
Glucose	70
Potassium	3.9
Sodium	146

Assigned Doctor: Megan Andes
Diagnosis: HIV; Failure to Thrive

Prescriptions

Medication	Dosage	Frequency	Start Date	End Date
Albuteral	1	Daily	2014-03-12	2015-05-12
Pepto Bismol	4	With meals	2016-09-11	2016-09-11

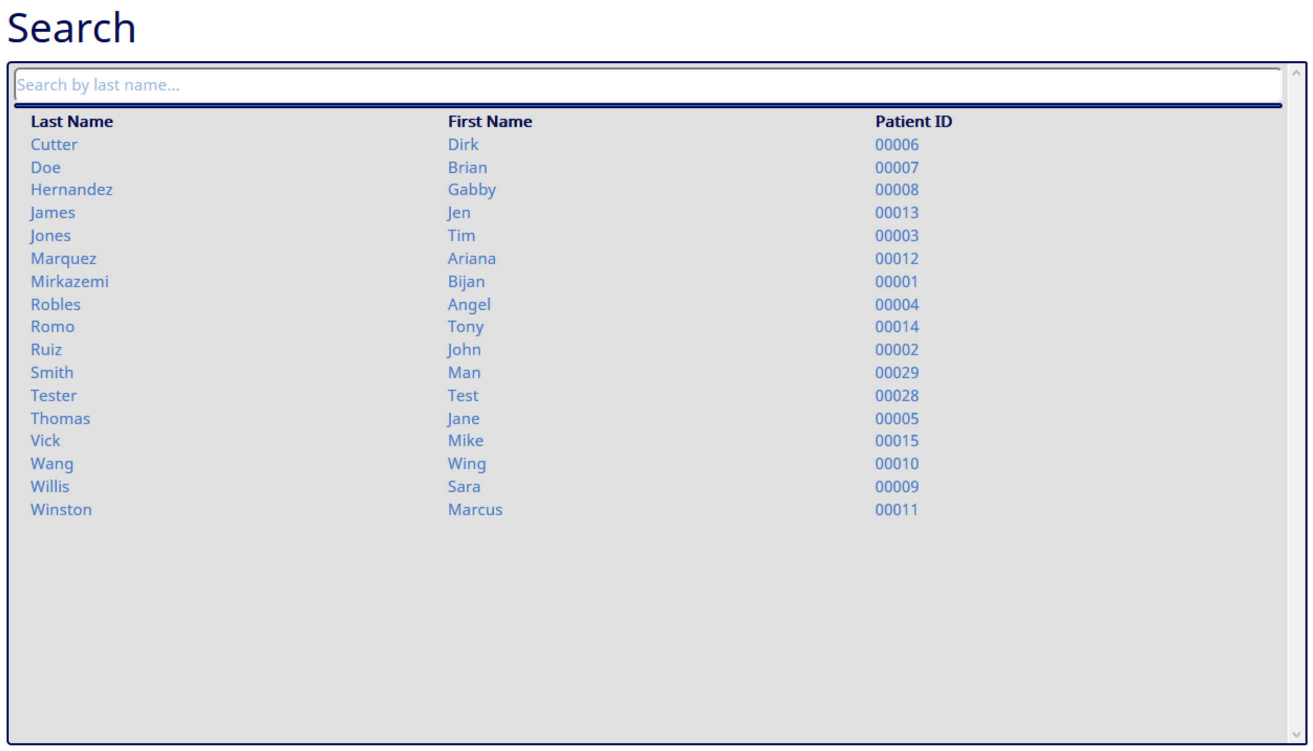
Additionally nurses and doctors can access patient vital charts which shows line graphs detailing each visit to the hospital and corresponding values for blood pressure, heart rate and respiratory rate.


Patient Graph Data

Patient Information

+ Name:	Tim J Jones
+ Address:	Elm St Shafter CA 94081
+ Phone:	1234567
+ Date of Birth:	1980-01-20
+ Sex:	Male
+ Insurance:	Aetna
+ Langauge:	English

Another feature is a dynamic search of patients which will narrow search results as you type patient’s last name so doctors and nurses can easily find the patients they need to view.



Search

Search by last name...

Last Name	First Name	Patient ID
Cutter	Dirk	00006
Doe	Brian	00007
Hernandez	Gabby	00008
James	Jen	00013
Jones	Tim	00003
Marquez	Ariana	00012
Mirkazemi	Bijan	00001
Robles	Angel	00004
Romo	Tony	00014
Ruiz	John	00002
Smith	Man	00029
Tester	Test	00028
Thomas	Jane	00005
Vick	Mike	00015
Wang	Wing	00010
Willis	Sara	00009
Winston	Marcus	00011

5.4 Description of Code

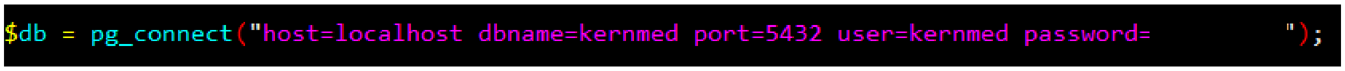
This section will give a brief overview of how the functionalities for the database front-end application was developed and implemented.

PHP, HTML and Javascript

The application was created using Hypertext Markup Language(html), Hypertext Preprocessor (php), and Javascript. Due to the application being produced with HTML, the front-end for the database is accessible via any standard web browser. HTML provides a clean and aesthetic look for the front-end of the application to be developed so users that may not have much experience with a computer will be able to comfortably and efficiently use the application. Javascript was also implemented to allow dynamic actions such as a search feature that updates in real time and adding prescription fields at the push of a button. PHP was used in order to access the database as well as manipulate information in the form of variables from the database which provides an easy way to interact with the database.

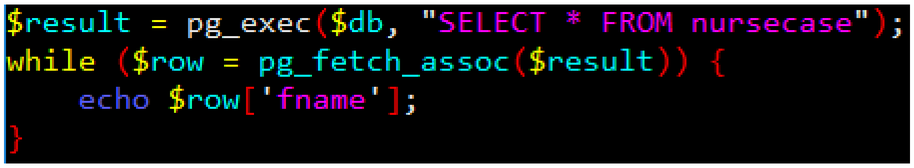
5.4.1 Database Connection & Manipulation

PHP allows developers to connect easily to a postgreSQL database with the function pg_connect. Developers can also quickly query and update the database with this type of accessibility. As such, the following connection string was used to connect to the database:



```
$db = pg_connect("host=localhost dbname=kernmed port=5432 user=kernmed password=         ");
```

With a connection made to the database, functions and views can be used using the pg_exec() function which takes the database resource value and a string which will call the view or function. To access the nursecase view in this way we use the following code:

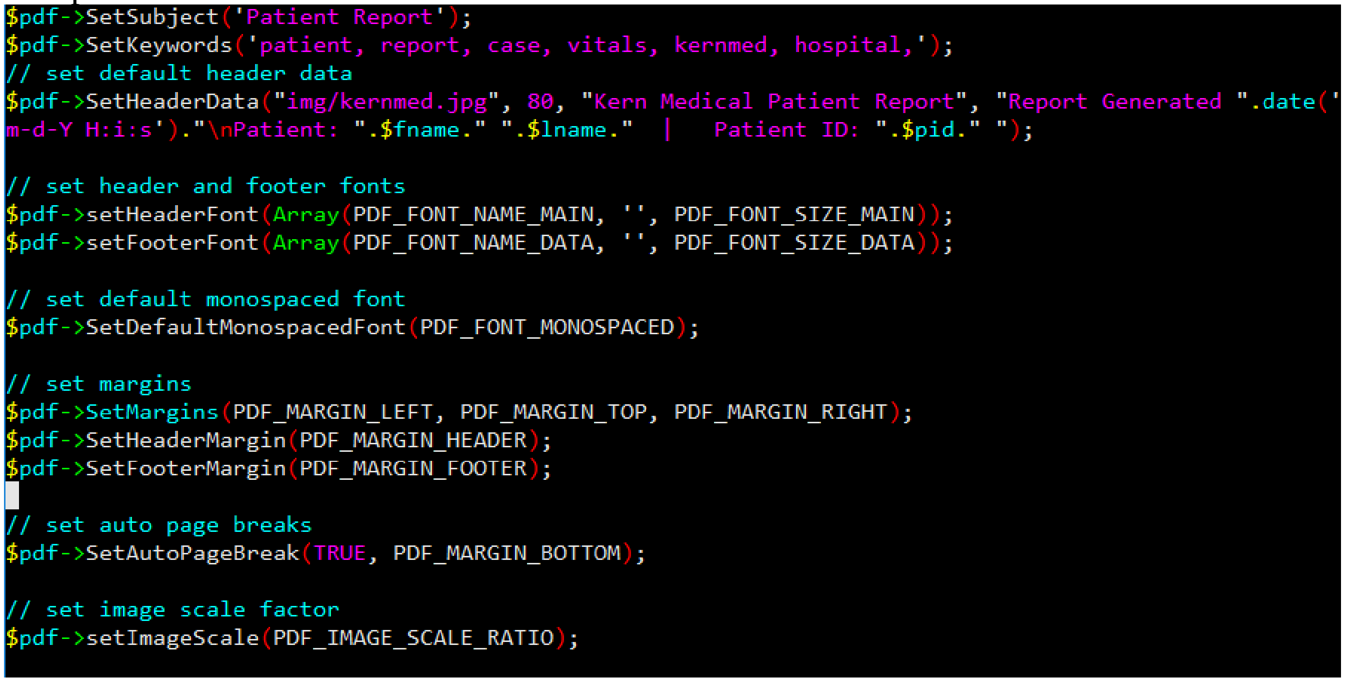


```
$result = pg_exec($db, "SELECT * FROM nursecase");
while ($row = pg_fetch_assoc($result)) {
    echo $row['fname'];
}
```

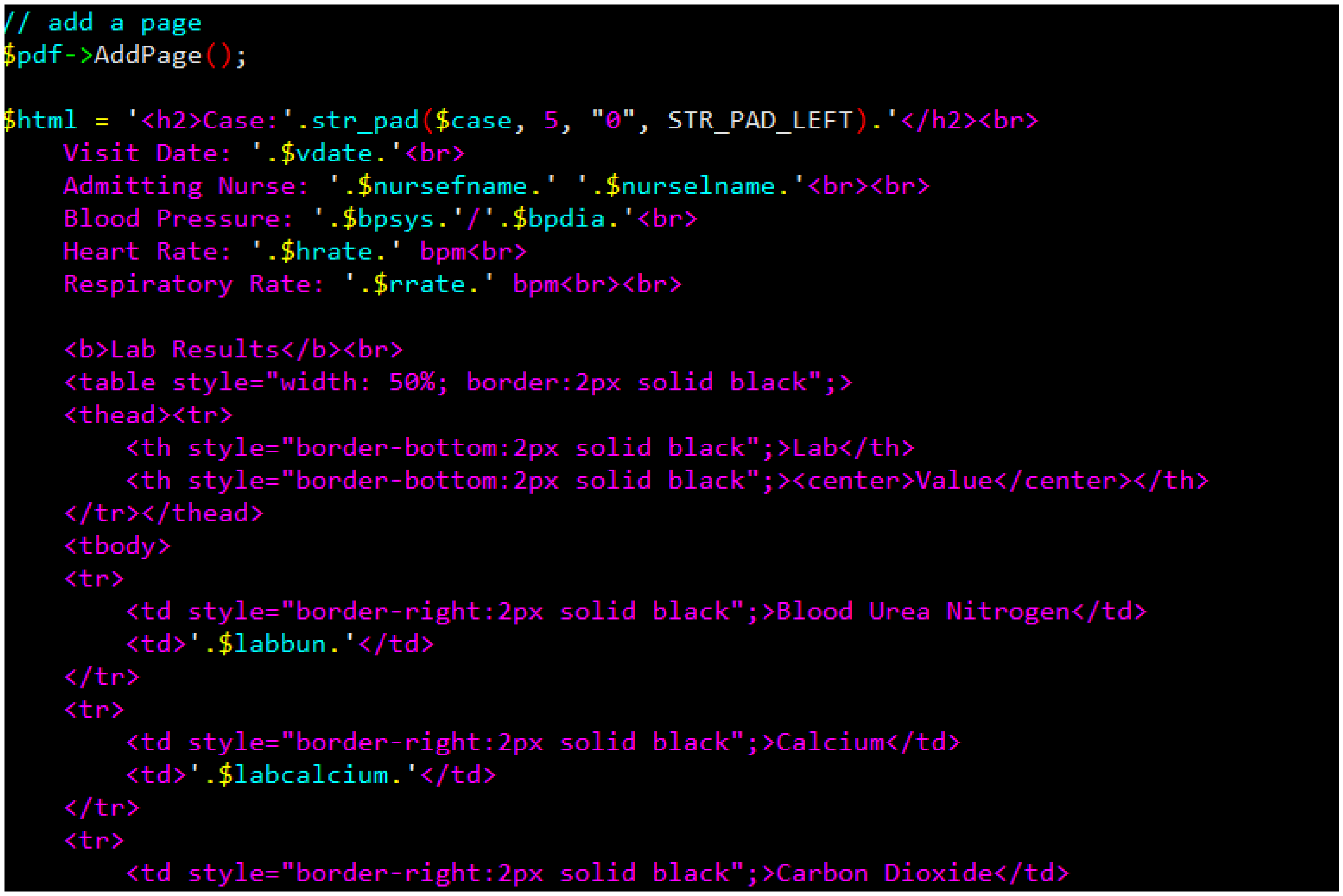
pg_fetch_assoc() is a function that allows us to traverse the results of the query and with that we can output them into PHP variables for later use or use them within a while loop. This is the primary way of getting information from our database to the application and also works in a similar way when submitting SQL statements back to the database.

5.4.2 Reports

In order to generate reports for the application an open source PHP class for generating PDF documents called TCPDF was implemented. TCPDF allows one to easily implement reports with their custom library. Implementing TCPDF was simple in that there are a variety of examples on their website to learn from. (https://tcpdf.org). Sample code is below:



```
$pdf->SetSubject('Patient Report');
$pdf->SetKeywords('patient, report, case, vitals, kernmed, hospital,');
// set default header data
$pdf->SetHeaderData("img/kernmed.jpg", 80, "Kern Medical Patient Report", "Report Generated ".date('
m-d-Y H:i:s')."\nPatient: ".$fname." ".$lname."  |   Patient ID: ".$pid." ");

// set header and footer fonts
$pdf->setHeaderFont(Array(PDF_FONT_NAME_MAIN, '', PDF_FONT_SIZE_MAIN));
$pdf->setFooterFont(Array(PDF_FONT_NAME_DATA, '', PDF_FONT_SIZE_DATA));

// set default monospaced font
$pdf->SetDefaultMonospacedFont(PDF_FONT_MONOSPACED);

// set margins
$pdf->SetMargins(PDF_MARGIN_LEFT, PDF_MARGIN_TOP, PDF_MARGIN_RIGHT);
$pdf->SetHeaderMargin(PDF_MARGIN_HEADER);
$pdf->SetFooterMargin(PDF_MARGIN_FOOTER);

// set auto page breaks
$pdf->SetAutoPageBreak(TRUE, PDF_MARGIN_BOTTOM);

// set image scale factor
$pdf->setImageScale(PDF_IMAGE_SCALE_RATIO);
```



```
// add a page
$pdf->AddPage();

$html = '<h2>Case:'.str_pad($case, 5, "0", STR_PAD_LEFT).'</h2><br>
    Visit Date: '.$vdate.'<br>
    Admitting Nurse: '.$nursefname.' '.$nurselname.'<br><br>
    Blood Pressure: '.$bpsys.'/'.$bpdia.'<br>
    Heart Rate: '.$hrate.' bpm<br>
    Respiratory Rate: '.$rrate.' bpm<br><br>

    <b>Lab Results</b><br>
    <table style="width: 50%; border:2px solid black";>
    <thead><tr>
        <th style="border-bottom:2px solid black";>Lab</th>
        <th style="border-bottom:2px solid black";><center>Value</center></th>
    </tr></thead>
    <tbody>
    <tr>
        <td style="border-right:2px solid black";>Blood Urea Nitrogen</td>
        <td>'.$labbun.'</td>
    </tr>
    <tr>
        <td style="border-right:2px solid black";>Calcium</td>
        <td>'.$labcalcium.'</td>
    </tr>
    <tr>
        <td style="border-right:2px solid black";>Carbon Dioxide</td>
```

The code above is the report page. As you can see, TCPDF becomes as simple as writing basic html code into a variable and outputting that variable to a PDF file. A PDF file is generated which can be printed and saved which makes developing a report not too difficult.

5.5 Design and Implementation Process

Before we conclude the documentation for the application, a useful exercise is to provide a detailed overview of all the steps for implementing a database application. This will include lessons learned from the development process.

Requirements Collection and Analysis

During the “requirements collection and analysis” phase, the developer should start to learn about the business and fully understand the components and how they are related. It is very important to perform a detailed survey of the business so the operations can be completely and effectively understood.

Conceptual Database Design

During the conceptual database design, one of the most important aspects is to develop a detailed, visual representation of the organization's structure based on the data that was collected in the previous phase. One useful way to test potential problems is to create multiple designs that way you can fully explore all potential problems. The conceptual design will form the groundwork for the physical database therefore all possible problems should be tested and run though.

Logical Database Design

Once the conceptual design is finished, the database can be converted into a logical database. The logical database will be the design that is used in software implementation. Due to the fact that the logical design will be used in software implementation, the conversion process between the conceptual and logical designs must be followed accurately and precisely and the resulting relations must then be normalized to ensure the design of the database is sound.

Physical Database Implementation

There are many DBMS's that were explored in previous sections of this document. Each DBMS provides a slightly different functionality therefore it is very important when deciding which DBMS will be most useful for a specific database. Once a useful DBMS is chosen, the logical design must be followed closely in the construction of the database. After construction, the database should be loaded with plenty of data so any potential problems will be spotted easily. The use of stored procedures, triggers, and views will be designed and implemented to make application development easier.

Database Application

Once the physical database is implemented and up and running the next step is to create a user interface that connects to the database and will be designed for different types of users therefore it should be easy to use and understand. The concept of "user-friendly" is the key focus of the design. Making the user interface easy to use is absolutely necessary. Users should be able to access as well as manipulate the database without any issues.

5.6 Embedded Questions

Outcome	Bijan Mirkazemi	Robert Pierucci
An ability to analyze a problem, and identify and define the computing requirements and specifications appropriate to its solution.	9	9
An ability to design, implement and evaluate a computer- based system, process, component, or program to meet desired needs, An ability to understand the analysis, design, and implementation of a computerized solution to a real-life problem.	9	8
An ability to communicate effectively with a range of audiences. An ability to write a technical document such as a software specification white paper or a user manual.	9	9

An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer-based systems in a way that demonstrates comprehension of the tradeoffs involved in design choices.	8	9