Plumbing Program

CMPS 3420 – Database Systems Project

Cody Graves

Last Updated: 5/17/2018

Table Of Contents

Phase I

Phase II

Phase III

Phase IV

Phase V

1.1 Fact Finding and Information Gathering

1.2 Target Organization

This software is targeted towards service businesses which compensate employees (or technicians) with commission-based pay. For the purpose of this project, our example business is a plumbing company. This company's focus is on sending technicians out to customer's properties to take care of plumbing issues, then paying the technicians what they have earned based on their commission rate and the cost of the project.

1.3 Fact Finding Methods

The primary fact finding method used to gather data and operational data for this project was personal work experience. As a manager of a small plumbing company, I have experience not only in taking customer information and dispatching technicians to solve plumbing problems, but also in calculating worker compensation based on commissions for services completed.

1.4 Organization Structure and Software Scope

The database for this software will be based on the the structure of our plumbing company. The company keeps track of customers, customer addresses (including properties which are leased or rented out), employees, the scheduling and records of jobs or estimates, as well as the commission earned by each employee for each job.

The basic flow of the company is as follows: a customer calls in a plumbing problem at a certain address, which is then received and turned into a ticket. This ticket is assigned a technician (or technicians) who are scheduled a specific time to diagnose or fix the plumbing problem. After that, the technicians either create an estimate for the customer, describing the problem, proposed solution, and estimated cost to provide their services, or an invoice (or receipt) which describes the work details and the total amount the customer is responsible for paying (either upon completion or in some cases, within an agreed upon payment term). Once an invoice is completed, a manager or qualified employee calculates each participating technician's commission amount from the total of the invoice and that technician's commission rate. Any non-commissionable items (items that do not count towards a technician's commission), such as parts used, are deducted from the total of the invoice before a commission is calculated.

This software will keep track of customers, their addresses and problems, job tickets, technicians, ticket schedules, technician commissions, and finally technician pay records.

2.1 Conceptual Database Design

2.2 Entities

Entity Name: EMPLOYEE

- Description: Technician's who provide services.
- Primary Key: Employee_id
- Candidate Keys: Employee_id
- Weak/Strong: Strong
- Attributes:

Name: Employee_id
Data Type: integer
Description: Unique identifier for each employee.
Value Set: 0000-9999
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Name
Data Type: string
Description: Employee's first and last name.
Value Set: 100 chars
Atomic / Composite: Composite
Single-value / Multi-value: Single
Stored / Derived: Derived from First_name and Last_name.
NULL Values: No

Name: First_name
Data Type: string
Description: Employee's first name.
Value Set: 50 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Last_name
Data Type: string
Description: Employee's last name.
Value Set: 50 chars

Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Phone
Data Type: integer
Description: Employee's phone number.
Value Set: 10 digit integer
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: Active
Data Type: boolean
Description: Whether employee is active or not.
Value Set: True / False
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Amount_owed
Data Type: float
Description: Current amount owed from commissions for employee.
Value Set: 0.00 – 999,999,999.99
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Derived from sum of job commissions.
NULL Values: No. If no amount is owed, value will be zero.

Name: Emp_total_sales
Data Type: float
Description: Total amount employee has made from commissions.
Value Set: 0.00 – 999,999,999.99
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Derived from sum of payment records and current amount owed.
NULL Values: No

Entity Name: COMMISSION_RATE

- Description: Rate of commission for technicians.
- Primary Key: Position

- Candidate Keys: Position
- Weak/Strong: Strong
- Attributes:

  Name: Position
  Data Type: int
  Description: Number of the position for this rate.
  Value Set: 0-999
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

  Name: Pos_name
  Data Type: String
  Description: Name of the position for this rate.
  Value Set: 50 chars
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

  Name: Rate
  Data Type: float
  Description: Percentage of total job amount which employee will be paid for.
  Value Set: 0.00% - 100.00%
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

Entity Name: EMP_PAYMENT_RECORD

- Description: Record which shows how much employee was last paid and when.
- Primary Key: Payment_id
- Candidate Keys: Payment_id
- Weak/Strong: Strong
- Attributes:

  Name: Payment_id
  Data Type: int
  Description: Unique identifier for the payment record.
  Value Set: 0000 - 9999
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

Name: Payment_amount
Data Type: float
Description: Amount employee was paid for.
Value Set: 00.00 – 999,999,999.99
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Derived from EMPLOYEE's Amount_owed attribute.
NULL Values: No

Name: Payment_date
Data Type: timestamp
Description: Date employee was paid for this record.
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Entity Name: CUSTOMER

- Description: Customers
- Primary Key: Customer_id
- Candidate Keys: Customer_id, Phone (if NULL values are ignored)
- Weak/Strong: Strong
- Attributes:

Name: Customer_id
Data Type: int
Description: Unique customer identifier.
Value Set: 000000 - 999999
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Name
Data Type: string
Description: Customer's first and last name
Value Set: 100 chars
Atomic / Composite: Composite
Single-value / Multi-value: Single
Stored / Derived: Derived from CUSTOMER's First_name and Last_name.
NULL Values: No

Name: First_name
Data Type: string
Description: Customer's first name

Value Set: 50 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Last_name
Data Type: string
Description: Customer's first and last name
Value Set: 50 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Phone
Data Type: integer
Description: Customer's phone number
Value Set: 10 digit integer
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: Cust_total_sales
Data Type: float
Description: Total amount customer has paid in sales.
Value Set: 00.00 – 999,999,999.99
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Derived from Invoice_total for every ticket of this customer.
NULL Values: No

Entity Name: LOCATION

- Description: Address information.
- Primary Key: Location_id
- Candidate Keys: Location_id, Address
- Weak/Strong: Strong
- Attributes:

  Name:Location_id
  Data Type: int
  Description: Key for locations
  Value Set 1-999999
  Atomic / Composite: Atomic
  Single-value/ Multi_value: Single

Stored / Derived: Derived
NULL Values: No

Name: Address
Data Type: string
Description: Physical street address.
Value Set: 100 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: City
Data Type: string
Description: City for location.
Value Set: 100 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: Zipcode
Data Type: int
Description: Zipcode for location.
Value Set: 5 digit integer
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: State
Data Type: string
Description: Location state.
Value Set: 2 chars (state abbreviation)
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Entity Name: PROBLEM

- Description: Plumbing problem the customer called in for.
- Primary Key: Problem_id
- Candidate Keys: Problem_id
- Weak/Strong: Strong
- Attributes:

Name: Problem_id
Data Type: integer
Description: Unique identifier for problem.
Value Set: 000000000 - 999999999
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Problem_description
Data Type: string
Description: Description of the plumbing problem.
Value Set: 255 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Work_type
Data Type: string
Description: Type of plumbing work being done
Value Set: Drain clog / Water Line / Gas Line / Fixture / Septic / Water Heater / Other
Atomic / Composite: Atomic
Single-value / Multi-value: Multi-value. Customer can call in for multiple issues.
Stored / Derived: Stored
NULL Values: No

Entity Name: TICKET

- Description: Problem ticket which will be assigned to technicians.
- Primary Key: Ticket_id
- Candidate Keys: Ticket_id, Invoice_number
- Weak/Strong: Strong
- Attributes:

Name: Ticket_id
Data Type: integer
Description: Unique ticket identifier number.
Value Set: 000000000 - 999999999
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Estimated_start
Data Type: timestamp

Description: Estimated start of work
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: Estimated_end
Data Type: timestamp
Description: Estimated end of work
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: Invoice_number
Data Type: integer
Description: Unique invoice number written up by technician for job or estimate.
Value Set: 00000 - 99999
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Invoice_total
Data Type: float
Description: Total amount written on invoice.
Value Set: 00.00 – 999,999,999.99
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes. If no work was done and no estimate given.

Name: Invoice_date
Data Type: timestamp
Description: Date the technician visited location and wrote an invoice.
Value Set: 00/00/0000 – 31/12/9999
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: Invoice_desc

Data Type: string
Description: Description of work completed or estimate by technician.
Value Set: 1000 chars
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes. This is important, but not necessary for just calculating commissions.

Name: Start_time
Data Type: timestamp
Description: Time work began.
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: End_time
Data Type: timestamp
Description:Time work began.
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Name: Payment_type
Data Type: string
Description: Method of customer payment for job
Value Set: Cash / Check / Credit Card / E-Check / Charge / Other
Atomic / Composite: Atomic
Single-value / Multi-value: Multi-value. Ex: Pay half with cash and half with credit card.
Stored / Derived: Stored
NULL Values: Yes

Name: Is_estimate
Data Type: boolean
Description: If true, then Invoice_total does not contribute to employee commission.
Value Set: True / False
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored

NULL Values: No

Entity Name: NON_COMM_ITEM

- Description: Non-commissionable items which don't contribute to employee commission.
- Primary Key: Item_id
- Candidate Keys: Item_id
- Weak/Strong: Strong
- Attributes:

  Name: Item_id
  Data Type: integer
  Description: Unique item identifier.
  Value Set: 000000 - 999999
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

  Name: Item_desc
  Data Type: string
  Description: Description of the item, parts, or equipment used.
  Value Set: 255 chars
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

  Name: Item_amount
  Data Type: float
  Description: Amount for item to be deducted from invoice total
  Value Set: 00.00 – 999,999,999.99
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No.

2.3 Relationships

Relationship Name: WAS_PAID

- Description: After an employee is paid, a payment record is made for that payment.
- Participating Entities: EMPLOYEE, EMP_PAYMENT_RECORD
- Cardinality Ratio: 1:M
- Participation Constraint: Partial. Not every employee will have a payment record
- Recursive: No

Relationship Name: HAS_RATE

- Description: Each employee has a commission rate.
- Participating Entities: EMPLOYEE, COMMISSION_RATE
- Cardinality Ratio: 1:M
- Participation Constraint: Total
- Recursive: No
- Attributes:

  Name: Start_date
  Data Type: timestamp
  Description: Employee start date at this commission rate
  Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: No

  Name: End_date
  Data Type: timestamp
  Description: Employee end date at this commission rate
  Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
  Atomic / Composite: Atomic
  Single-value / Multi-value: Single
  Stored / Derived: Stored
  NULL Values: Yes

Relationship Name: DID_WORK

- Description: Employee does work for a ticket.
- Participating Entities: EMPLOYEE, TICKET
- Cardinality Ratio: 1:M
- Participation Constraint: Partial. Not all employees have a ticket.
- Recursive: No
- Attributes:

  Name: Start_time
  Data Type: timestamp

Description: Time work started for employee
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: No

Name: End_time
Data Type: timestamp
Description: Time work ended for employee
Value Set: 00/00/0000 00:00:00 – 31/12/9999 23:59:59
Atomic / Composite: Atomic
Single-value / Multi-value: Single
Stored / Derived: Stored
NULL Values: Yes

Relationship Name: OWNS_LOC

- Description: Customer owns a location.
- Participating Entities: CUSTOMER, LOCATION
- Cardinality Ratio: 1:M
- Participation Constraint: Total
- Recursive: No

Relationship Name: PROBLEM_AT

- Description: Associates problem with the location of the problem.
- Participating Entities: LOCATION, PROBLEM
- Cardinality Ratio: M..M
- Participation Constraint: Total
- Recursive: No

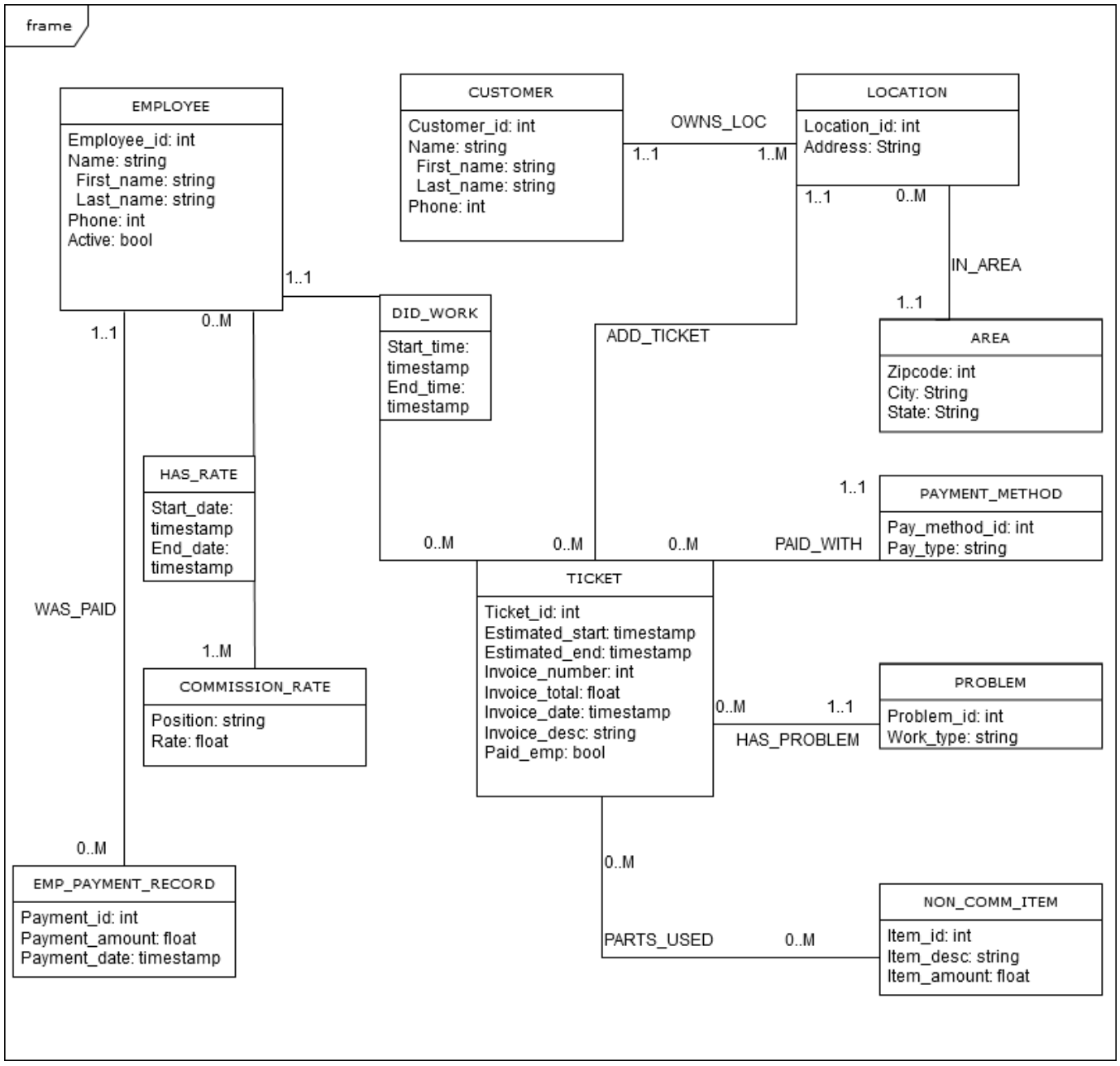
Relationship Name: ADD_TICKET

- Description: A ticket is created from a problem to be assigned.
- Participating Entities: PROBLEM, TICKET
- Cardinality Ratio: 1:M
- Participation Constraint: Total
- Recursive: No

Relationship Name: PARTS_USED

- Description: Associates ticket with non-commissionable items (usually parts).
- Participating Entities: TICKET, NON_COMM_ITEM
- Cardinality Ratio: M:M
- Participation Constraint: Partial
- Recursive: No

2.4 ER-Diagram



3.1 Conceptual Database and Logical Database

3.2 ER Model and Relational Model

Description

The ER (Entity-Relationship) model was developed by Peter Chen in 1976 in an attempt to describe real world business objects and their relationships in database modeling. The ER model describes things of a certain domain as well as the relationships between them. The ER model consists of entity types, which are classes of things of interest and relationship types, which classify the relationships between entities. Entities and relationships are also described by their associated properties, called attributes. Today, however, the ER model is commonly used to represent the data structure based on business needs in an easy to understand visual. It is typically used to describe data and define the composition of a relational database.

The relational model was first introduced by Edgar Codd in 1969 as a database management method based on first-order predicate logic. In the relational model, data is represented as tables of values, called relations. Tables consists of attributes as columns and tuples as rows. Attributes are the name of the role played by the domain, which describes the types of values that can appear for each attribute. An attribute which is used to identify a tuple is called a primary key. A tuple is a list of values within the domain. Rather than relationship types to describe the relations between two objects, the relational model relies on relations having foreign keys which point to the primary key of another relation. A relation schema is used to describe a relation and consists of a relation name and a list of attributes. Relational databases are the most common type of database systems and are all based on the relational model.

Comparison

The ER model represents a collection of entities and relationships between them. It is easy to visualize and understand for non-technical users. The ER model also has a mapping cardinality constraint. The relational model is more complicated for the non-technical user, but can describe the same collection of data represented by relations or tables. The relational model has no mapping cardinality constraint. The ER model is useful for visualizing a business's data needs, but it is not supported by database management systems. On the other hand, the relational model is supported by database management systems.

3.3 Conversion from Conceptual Database to Logical Database Overview

The ER model is a conceptual model, which represents data in an abstract form. This is useful for visualizing data, but in order to be used in a relational database, it must be converted to a logical model. The logical model represents data in the format required for most databases, including the relational database.

The first step is converting entity types to relations. For strong entity types, relations can be created using the attributes of the entity type and a unique attribute as the primary key. For weak entity types, a relation is created using the simple attributes of the weak entity type, as well as simple components of composite attributes, and a foreign key for each primary key of the owner entity type. The primary key of the weak entity type is the combination of foreign keys for the owner entity type. For multivalued attributes, a new relation must be created.

The next step is converting relationship types to relations. For 1:1 relationship types, a foreign key for the first entity type is created from the primary key of the second entity type and a foreign key for the second entity type is created from the primary key of the first entity type. For 1:M relationship types, a foreign key is created for each entity type related to the first entity type from the first entity type's primary key. For M:M relationships, a new relation must be created with the primary keys of each related entity type. For superclass relationship types, there are four methods which may be used. The first method involves creating a relation for the superclass as well as one for each subclass. This can be used for every type of specialization, total or partial and disjoint or overlapping. The second method involves creating a relation for every subclass, but this only works for total and disjoint specializations. The third method involves creating a single relation with one type attribute that indicates which subclass the tuple belongs to. This method can be used only with disjoint specializations. The fourth method involves creating a single relation with multiple type attributes, which include a boolean attribute that indicates whether or not a specific tuple belongs to a certain subclass. For n-ary relationship types, a new relationship is created to represent that relationship type with the foreign key is the primary keys of the participating entity types.

3.4 Database Constraints

Database constraints define restrictions which data in a database must adhere to. The purpose of these rules is to enforce data validity so that the data of a database does not become corrupt over time.

Entity Constraint: Every primary key must not equal NULL. When an entity's primary key is NULL the value can't be used to identify certain entity tuple id. Location requires all address, city, zip code, and

state to use that data if anything needs to be shipped to that customer.

UNIQUE- Never has duplicates in the table.
NOT NULL- Never accepts null values.
Primary key- Unique Only, one primary key attribute per table.
Foreign key- Referencing a primary key of another table.

3.5 Relation Schema

EMPLOYEE(Employee_id, First_name, Last_name, Phone, Active)

Attributes

- Employee_id (Primary Key)
  Domain: Integer (0-9999). Value must not be NULL.
- First_name
  Domain: String (50 Chars). Value must not be NULL.
- Last_name
  Domain: String (50 Chars). Value must not be NULL.
- Phone
  Domain: String (10 digit integer). Value can be NULL.
- Active
  Domain: Boolean (True/False). Value must not be NULL.

Constraints

Primary Key: Employee_id - Must be unique.

Candidate Keys

Employee_id, Phone

EMP_PAYMENT_RECORD(Payment_id, Employee_id, Payment_amount, Payment_date)

Attributes

- Payment_id (Primary Key)
  Domain: Integer (0-9999). Value must not be NULL.
- Employee_id (Foreign Key)
  Domain: Integer (0-9999). Value must not be NULL.
- Payment_amount
  Domain: Float (00.00 - 999,999,999.99). Value must not be NULL.
- Payment_date
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value must not be NULL.

Constraints

Primary Key: Payment_id - Must be unique.
Foreign Key: Must match an existing Employee_id.
Business Rule: Payment_date must be valid.

Candidate Keys

Payment_id

HAS_RATE(Employee_id, Position, Start_date, End_date)

Attributes

- Employee_id (Foreign Key)
  Domain: Integer (0-9999). Value must not be NULL.
- Position (Foreign Key)
  Domain: String (50 Chars). Value must not be NULL.
- Start_date
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value must not be NULL.
- End_date
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value can be NULL.

Constraints

Primary Key: Position and Employee_id must match existing.
Foreign Key: Must match existing Employee_id and Position
Business Rule: Pos_name must be matched to a rate and thus must be unique.

Candidate Keys

Position

COMMISSION_RATE(Position, Rate)

Attributes

- Position (Primary Key)
  Domain: String (50 Chars). Value must not be NULL.
- Rate
  Domain: Float (0.00%-100.00%). Value must not be NULL.

Constraints

Primary Key: Position - Must be unique.

Candidate Keys

Position

CUSTOMER(Customer_id, First_name, Last_name, Phone)

Attributes

- Customer_id (Primary Key)
  Domain: Integer (000000-999999). Value must not be NULL.

- First_name
  Domain: String (50 chars). Value must not be NULL.
- Last_name
  Domain: String(50 chars). Value must not be NULL.
- Phone
  Domain: String(10 digits). Value must not be NULL.

Constraints

Primary Key: Customer_id - Must be unique.
Business Rule: N/A

Candidate Keys

Customer_id, Phone

DID_WORK(Ticket_id, Employee_id, Start_time, End_time)

Attributes

- Ticket_id (Primary Key/Foreign Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Employee_id (Foreign Key)
  Domain: Integer (0-9999). Value must not be NULL.
- Start_time
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value can be NULL.
- End_time
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value can be NULL.

Constraints

Primary Key: Ticket_id- Must match an existing Ticket.
Foreign Key: Must match existing Employee_id and Ticket_id

Candidate Keys

Ticket_id

TICKET(Ticket_id, Location_id, Problem_id, Estimated_start, Estimated_end, Invoice_number, Invoice_total, Invoice_date, Invoice_desc, Payment_type, Is_estimate, Paid_emp)

Attributes

- Ticket_id (Primary Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Location_id (Foreign Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Problem_id (Foreign Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Estimated_start
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value can be NULL.
- Estimated_end

Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value can be NULL.

- Invoice_number
  Domain: Integer (00000-99999). Value must not be NULL.
- Invoice_total
  Domain: float (00.00-999,999,999.99). Value must not be NULL.
- Invoice_date
  Domain: Timestamp (Format: dd/mm/yyyy hh:mm:ss). Value must not be NULL.
- Invoice_desc
  Domain: String (1000 Chars). Value can be NULL.
- Payment_type
  Domain: String (40 Chars). Value can be NULL.
- Is_estimate
  Domain: Boolean. Value must not be NULL.
- Paid_emp
  Domain: Boolean. Value must not be NULL.

Constraints

Primary Key: Ticket_id - Must be unique.
Foreign Key: Must match existing Problem_id and Location_id.
Business Rule: Estimated_start must be before Estimated_end. Start_time must be before End_time.

Candidate Keys

Ticket_id, Invoice_number

LOCATION(Location_id, Customer_id, Address, City, Zipcode, State)

Attributes

- Location_id (Primary Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Customer_id (Foreign Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Address
  Domain: String (100 Chars). Value must not be NULL.
- City
  Domain: String (100 Chars). Value must not be NULL.
- Zipcode
  Domain: String (5 chars). Value must not be NULL.
- State
  Domain: String (2 Chars). Value must not be NULL.

Constraints

Primary Key: Location_id
Foreign Key: Must match existing Customer_id
Business Rule: State must match 2 letter state abbreviations in the US.

Candidate Keys

Location_id, Customer_id

NON_COMM_ITEM(Item_id, Item_desc, Item_amount)

Attributes

- Item_id (Primary Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Item_desc
  Domain: String (255 Chars). Value must not be null NULL.
- Item_amount
  Domain: Integer (0-999999999.99). Value must not be NULL.

Constraints

Primary Key: item_id - Must be unique.
Business Rule: N/A

Candidate Keys

Item_id

PARTS_USED(Ticket_id, Item_id)

Attributes

- Item_id (Foreign Key)
  Domain: Integer (0-999999). Value must not be NULL.
- Ticket_id (Foreign Key)
  Domain: Integer (0-999999). Value must not be NULL.

Constraints

Composite Primary Key: Ticket_id and Item_id - Must match existing Ticket and Non_comm_item
Business Rule: N/A

Candidate Keys

Ticket_id and Item_id

PROBLEM(Problem_id,Work_type)

Attributes

- Problem_id (Primary key)
  Domain: Integer (0-999999). Value must not be NULL.
- Work_type
  Domain: String(100 chars). Value must not be NULL.

Constraints

Primary Key: Problem_id must be unique.
Ticket and Non_comm_item
Business Rule: N/A

Candidate Keys

Ticket_id and Item_id

3.6 Relation Instances

EMPLOYEE(Employee_id, First_name, Last_name, Phone, Active, Amount_owed, Emp_total_sales)

Employee_id	First_name	Last_name	Phone	Active	Amount_owed	Emp_total_sales
1	John	Doe	6616232111	True	49.23	109352.69
2	Jane	Doe	6616251485	True	100.00	2231.36
3	Kenny	Martinez	5588751258	False	0.00	1000.00
4	Allan	Price	6621548758	True	3002.33	49253.66
5	Linn	Lee	6613629948	True	0.00	9003.00

EMP_PAYMENT_RECORD(Payment_id, Employee_id, Payment_amount, Payment_date)

Payment_id	Employee_id	Payment_amount	Payment_date
112	1	908.50	02/05/2018
113	2	1040.00	02/05/2018
114	3	300.00	02/05/2018
115	4	1200.50	02/12/2018
116	5	607.73	02/12/2018

HAS_RATE(Employee_id, Position, Start_date, End_date)

Employee_id	Position	Start_date	End_date
03	Apprentice	01/22/2018	02/05/2018
01	Master	05/01/2015	NULL
04	Journeyman	03/02/2013	09/28/2017

COMMISSION_RATE(Position, Rate)

Position	Rate
Apprentice	10%
Journeyman	20%
Master	30%

CUSTOMER(Customer_id, First_name, Last_name, Phone, Cust_total_sales)

Customer_id	First_name	Last_name	Phone	Cust_total_sales
321	Luke	Evans	5548754155	19800.00
322	Anna	Barnett	6612515487	155.00
323	Diane	Cummings	6612154874	3175.50

DID_WORK(Ticket_id, Employee_id, Start_time, End_time)

Ticket_id	Employee_id	Start_time	End_time
112	2	01/01/2018 08:00:00	01/02/2018 11:00:00
113	4	01/25/2018 11:30:00	01/25/2018 12:30:00
114	5	NULL	NULL
115	1	11/21/2017 09:00:00	11/21/2017 09:45:00

TICKET(Ticket_id, Problem_id, Estimated_start, Estimated_end, Invoice_number, Invoice_total, Invoice_date, Invoice_desc, Payment_type, Is_estimate)

Ticket_id	Problem_id	Estimated_start	Estimated_end	Invoice_number	Invoice_total	Invoice_date	Invoice_desc	Payment_type	is_estimate
112	01	01/01/2018 08:00:00	01/02/2018 12:00:00	98837	2200.00	01/01/2018	Repaired broken pipe	Cash	False
113	02	01/25/2018 11:30:00	01/25/2018 13:00:00	98938	100.00	01/25/2018	Cleared drain	Check	False
114	03	11/20/2017 14:30:00	11/20/2017 15:30:00	99839	225.00	11/20/2017	Fix leaking toilet	NULL	True
115	02	11/21/2017 9:00:00	11/21/2017 10:00:00	99340	100.00	11/21/2017	Cleared drain	Check	False

LOCATION(Location_id, Customer_id, Address, City, Zipcode, State)

Location_id	Customer_id	Address	City	Zipcode	State
905	321	542 Woodrow st.	Bakersfield	93308	CA
359	322	424 Warren st.	Bakersfield	93308	CA
909	323	231 Union ave.	Bakersfield	93307	CA

PROBLEM(Problem_id, Work_type)

Problem_id	Work_type
01	broken water pipe
02	clogged drain
03	leaking toilet

PROBLEM_AT(Location_id, Problem_id)

Location_id	**Problem_id**
905	02
359	03
909	02

NON_COMM_ITEM(Item_id, Ticket_id, Item_desc, Item_amount)

Item_id	Item_desc	Item_amount
01	sink	199.99
02	toilet	105.99
03	shower head	35.00

PARTS_USED(Ticket_id, Item_id)

Ticket_id	Item_id
125	01
3120	02
1001	03

4.1 Sample Queries

Purpose

In this section, we will describe 10 sample queries along with their representation in relational algebra, tuple relational calculus, and domain relational calculus. Since this is a relational database, queries are expressed in either relational algebra or relational calculus. Relational algebra is a procedural method in which every sequence of performing the query must be defined. In relational algebra, operations are used on existing relations to create new relations until the desired result is found. Relational calculus is nonprocedural and uses formulas to reach the result. In relational calculus, we must specify the requirements for the query result, but not the specific sequences to reach that result.

4.2 Query Design

1. List all Employees that worked on tickets for every type of problem
2. List all Employees that worked on at least one invoice greater than $2000.00
3. List all Employees which were promoted in less than one year
4. List all tickets which have estimates which were given by “John Smith” since 2017
5. List all tickets which have invoices which are greater than $2000.00 and were completed in single day
6. List all customers which have had tickets only in 2015
7. List all locations in the 93312 zip code which have no more than 1 problem
8. List the employees who have had every position
9. List all tickets with work type “Sewer Repair” which were given an estimate
10. List the problems which were associated with the most expensive tickets in 2017

4.3 Relational Algebra for Sample Queries

1. List all Employees that worked on tickets for every type of problem
   T1 <- π e.Employee_id, p.Problem_id (σ e.Employee_id = d.Employee_id ^ d.Ticket_id = t.Ticket_id ^ t.Problem_id = p.Problem_id (EMPLOYEE e X DID_WORK d X TICKET t X PROBLEM p))
   T2 <- T1 % π p.Problem_id (PROBLEM)
   π EMPLOYEE.* (T2)

2. List all Employees that worked on at least one invoice greater than $2000.00
   T1 <- σ t.Invoice_total>2000 (DID_WORK d $\bowtie$ d.Ticket_id = t.Ticket_id (TICKET t))
   π e.* (EMPLOYEE e $\bowtie$ e.Employee_id = T1.Employee_id (T1)

3. List all Employees which were promoted in less than one year
   T1 <- σ hr1.Start_date < hr2.Start_date ^ hr2.Start_date < (hr1.Start_date + 00/00/0001) ^ hr1.Employee_id = hr2.Employee_id (HAS_RATE hr1 X HAS_RATE hr2)

π e.* (EMPLOYEE * T1)

4. List all tickets which have estimates which were given by “John Smith” since 2017
   T1 <- (EMPLOYEE e ⋈ e.Name = ‘John Smith’ ^ e.Employee_id = d.Employee_id (DID_WORK d))
   T2 <- T1 ⋈ T1.Ticket_id = t.Ticket_id ^ t.Invoice_date > 01/01/2017 ^ t.Is_estimate = true (TICKET t)
   π TICKET.* (T2)

5. List all tickets which have invoices which are greater than $2000.00 and were completed in single day
   T1 <- DID_WORK d ⋈ d.Ticket_id = t.Ticket_id ^ t.Invoice_total > 2000.00 (TICKET t)
   π TICKET.* (σ T1.Is_Estimate = false ^ T1.End_time - T1.Start_time < 24:00:00 (T1))

6. List all customers which have had tickets only in 2015
   T1 <- π CUSTOMER.*, TICKET.* ((((CUSTOMER * LOCATION) * PROBLEM_AT)* PROBLEM) * TICKET)
   T2 <- σ t1.Invoice_date >= 01/01/2015 ^ t1.Invoice_date < 01/01/2016 (T1)
   π CUSTOMER.* CUSTOMER % T2

7. List all locations in the 93312 zip code which have had no more than one problem
   π LOCATION.* (LOCATION - (π LOCATION.* (σ l.Location_id = p1.Location_id ^ l.Zipcode = 93312 ^ p1.Location_id = p2.Location_id ^ p1.Problem_id != p2.Problem_id(LOCATION L X PROBLEM_AT p1 X PROBLEM_AT p2))))

8. List employees who have had every position
   T1 <- π EMPLOYEE.*,HAS_RATE.*,COMMISSION_RATE.* (EMPLOYEE * HAS_RATE * COMMISSION_RATE)
   π EMPLOYEE.* (T1 % COMMISSION_RATE)

9. List all tickets with a problem work type “Sewer Repair” which were given an estimate
   π TICKET.* (σ t.Is_estimate = true (TICKET t) * σ p.Work_type=”Sewer Repair” (PROBLEM p))}

10. List the problems which were associated with the most expensive tickets in 2017
    T1 <- TICKET - π TICKET.* (TICKET a ⋈ a.Invoice_total < b.Invoice_total ^ a.Invoice_date >= 01/01/2017 ^ a.Invoice_date <= 12/31/2017 ^ b.Invoice_date >= 01/01/2017 ^ b.Invoice_date <= 12/31/2017 (TICKET b))
    π Problem.* (T1 ⋈ T1.Problem_id = p.Problem_id (PROBLEM p))

4.4 Tuple Relational Calculus for Sample Queries

1. List all Employees that worked on tickets for every type of problem
   {e | Employee(e) AND (∀ p(Problem(p) -> (∃ t)(Ticket(t) ^ (∃ d)(Did_work(d) ^ t.Problem_id = p.Problem_id ^ d.Employee_id = e.Employee_id ^ d.Ticket_id = t.Ticket_id)))}

2. List all Employees that worked on at least one invoice greater than $2000.00

{e | Employee(e) AND (∃ t)(Ticket(t) ^ (∃ d)(Did_work(d) ^ t.Ticket_id = d.Ticket_id ^ d.Employee_id = e.Employee_id ^ t.Invoice_total > 2000.00))}

3. List all Employees which were promoted in less than one year
   {e | Employee(e) ^ (∃ r1)(∃ r2)(Has_rate(r1) ^ (Has_rate(r2) ^ r1.Employee_id = r2.Employee_id ^ e.Employee_id = r1.Employee_id ^ r1.Start_date < r2.Start_date ^ r2.Start_date < r1.Start_date + 00/00/0001))}

4. List all tickets which have estimates which were given by “John Smith” since 2017
   {t | Ticket(t) ^ (∃ e)(Employee(e) ^ (∃ d)(Did_work(d) ^ d.Ticket_id = t.Ticket_id ^ d.Employee_id = e.Employee_id ^ e.Name = “John Smith” ^ t.Invoice_date > 01/01/2017 ^ t.Is_estimate = true))}

5. List all tickets which have invoices which are greater than $2000.00 and were completed in single day
   {t | Ticket(t) ^ t.Invoice_total > 2000.00 ^ t.End_time - t.Start_time <= 23:59:59 ^ t.Is_estimate = false }

6. List all customers which have had tickets only in 2015
   {c | Customer(c) ^ ~(∃ t)(Ticket(t) ^ (∃ p)(Problem_at(p) ^ (∃ l)(Location(l) ^ t.Problem_id = p.Problem_id ^ p.Location_id = l.Location_id ^ l.Customer_id = c.Customer_id ^ t.Invoice_date < 01-01-2015 OR t.Invoice_date > 12-31-2015) }

7. List all locations in the 93312 zip code which have no more than 1 problem
   {l | Location(l) ^ ~(∃ p1)(∃ p2)(Problem_at(p1) ^ Problem_at(p2) ^ p1.Location_id = p2.Location_id ^ p1.Location_id = l.Location_id ^ p1.Problem_id != p2.Problem_id)}

8. List employees who have had every position
   {e | Employee(e) ^ (∀ p)(Commission_rate(p) -> (∃ h)(Has_rate(h) ^ h.Employee_id = e.Employee_id ^ h.Position = p.Position))}

9. List all tickets with a problem work type “Sewer Repair” which were given an estimate
   {t | Ticket(t) ^ (∃ p)(Problem(p) ^ t.is_estimate = true ^ t.Problem_id = p.Problem_id ^ p.Work_type = “Sewer Repair”)}

10. List the problems which were associated with the most expensive tickets in 2017
    {p | Problem(p) ^ (∃ t1)(Ticket(t1) ^ p.Problem_id = t1.Problem_id ^ t1.Invoice_date <= 12/31/2017 ^ t1.Invoice_date >= 01/01/2017 ^ ~(∃ t2)(Ticket(t2) ^ t1.Ticket_id != t2.Ticket_id ^ t2.Invoice_total > t1.Invoice_total ^ t2.Invoice_date <= 12/31/2017 ^ t2.Invoice_date >= 01/01/2017))}

4.5 Domain Relational Calculus for Sample Queries

1. List all Employees that worked on tickets for every type of problem
   {<a,b,c,d,e,f,g>| Employee(a,b,c,d,e,f,g) ^ (∀ p)(~Problem(p,_) V (Did_work(t,a,_,_) ^ Ticket(t,p,_,_,_,_,_,_,_,_,_)))}

2. List all Employees that worked on at least one invoice greater than $2000.00
   {<a,b,c,d,e,f,g> | Employee(a,b,c,d,e,f,g) ^ (∃ t)(Ticket(t,_,>2000) ^ Did_work(t,a,_,))}

3. List all Employees which were promoted in less than one year
   {<a,b,c,d,e,f,g> | Employee(a,b,c,d,e,f,g) ^ (∃ s1)(Has_rate(a,_,s1,_) ^ Has_rate(a,_,<s1+00/00/0001,_))}

4. List all tickets which have estimates which were given by “John Smith” since 2017
   {<a,b,c,d,e,f,g,h,i,j>| Ticket(a,b,c,d,e,f,g,h,i,j) ^ (∃ eid)(Employee(eid, “Smith”, “John”, _,_,_,_) ^ Did_work(a,eid,>01/01/2017,_))}

5. List all tickets which have invoices which are greater than $2000.00 and were completed in single day
   {<a,b,c,d,e,f,g,h,i,j>| Ticket(a,b,c,d,e,f,g,h,i,j) ^ j = false ^ f > 2000.00 ^ (∃ start)(∃ end)(Did_work(a,_,start,end) ^ end - start < 024:00:00)}

6. List all customers which have had tickets only in 2015
   {<cid,b,c,d,e>| Customer(cid,b,c,d,e) ^ (∀ tid)(∃ d2)(∃ pid)(∃ lid)(∃ cid) (~Ticket(tid,pid,_,_,_,_,d2,_,_,_) V (Problem_at(lid,pid) ^ Location(lid,cid,_,_,_,_) ^ Customer(cid,_,_,_,_) ^ (d2 < 01/01/2016 V d2 >= 01/01/2015)))}

7. List all locations in the 93312 zip code which have no more than 1 problem
   {<lid,b,c,d,e,f>| Location(lid,b,c,d,e,f) ^ e = 93312 ^ ~(∃ pid1)(Problem_at(lid,pid1) ^ (Problem_at(lid,!=pid1)))}

8. List employees who have had every position
   {<eid,b,c,d,e,f,g> | Employee(eid,b,c,d,e,f,g) ^ (∀ pid)(~Commission_rate(pid,_) V Has_rate(eid,pid,_,_))}

9. List all tickets with a problem work type “Sewer Repair” which were given an estimate
   {<tid,pid,c,d,e,f,g,h,i,isest>|Ticket(tid,pid,c,d,e,f,g,h,i,isest) ^ isest = true ^ Problem(pid,”Sewer Repair”)}

10. List the problems which were associated with the most expensives ticket in 2017
    {<pid,type> | Problem(pid,type) ^ (∃ tot)(∃ date)(∃ tid)(Ticket(tid,pid,_,_,_,tot,date,_,_,_) ^ ~Ticket(!tid,_,_,_,_,>tot,date,_,_,_) ^ date >= 01/01/2017 ^ date <= 12/31/2017)}

5.1 Normalization

Normalization

Database normalization is a technique of testing and restructuring relation schemas in order to minimize data redundancy as well as minimize insertion, deletion, and update/modification anomalies. The process consists of a top-down series of tests on the relation schema to show that they satisfy a specific normal form. If the relation schema fails the test for a particular normal form, it is decomposed or broken down into a smaller relation schema that does pass the test.

The purpose of normalization is to ensure a level of quality in the design of the database by reducing anomalies as well as reducing storage space by minimizing data redundancy. Aside from redundant information in the tuples, other properties used to measure the quality of a relation schema in a database are: clarity of the semantics for attributes, limited NULL values in tuples, and disallowing the possibility of generating spurious tuples (tuples which are joined from two or more tables where the joining attributes are neither primary keys nor foreign keys).

Normal Forms

Normal forms define conditions that must be met for a relation schema to achieve a certain quality of design. The normal form of a relation schema describes the normalization rules that a relation schema follows. In other words, the normal form refers to the highest normal form condition that a relation schema meets. There are four normal forms discussed here: First Normal Form, Second Normal Form, Third Normal Form, and Boyce-Codd Normal Form.

First Normal Form specifies that a relation schema must have attribute domains which only contain atomic, single values, that attributes in a column all be in a matching domain, and that columns have unique names. For example, if a relation were to include an attribute which allowed for multiple values, it would not be in first normal form and would be considered to have bad design. To fix this issue, the relation schema would need to be redesigned in a way that ensured each attribute contained an atomic, single value.

Second Normal Form relations are relations that meet the criteria for first normal form as well as the additional property that every attribute (column) in the relational schema must be fully functionally dependent on the primary key of that relation, which means that the attributes must be uniquely identified only by the entirety of every candidate key of that relation. In other words, assuming a relation is in first normal form, if any nonprime attributes (e.g. nonkey attributes, or attributes which cannot be used to uniquely identify a tuple), are functionally determined by any proper subset of the candidate key, it is not in second normal form. To make a relation schema be in second normal form, the relation schema must be broken down so that every attribute depends only on the primary key or entire composite primary key.

Third Normal Form relations are second normal form relations that also have no nonprime attribute which is transitively dependent on the primary key. This means that a nonprime attribute should not determine another nonprime attribute. In other words, all attributes on the relation should be determined only by the candidate keys. To make a relation schema in third normal form, the relation must be decomposed into a new relation that includes any nonprime attributes which determine other nonprime attributes.

Boyce-Codd Normal Form is a stricter form fashioned after third normal form. Relations in Boyce-Codd form are in third normal form with the added restriction that for every functional dependency in the relation, the key which functionally determines other attributes must be a super key. Some non-Boyce-Codd normal form relations may not be able to be decomposed into Boyce-Codd normal form.

Anomalies

Database anomalies are inconsistencies that exist within the database between two sets of data. Anomalies are caused by data redundancies that occur when storing the joins of relations which are not normalized and can be described as three different types of anomalies: Insertion Anomalies, Deletion Anomalies, and Modification Anomalies.

Insertion Anomalies occur when relations contain redundant attributes which must be added each time new data is added to that relation. For example, if instead of two relations, STUDENT and CLASS, there were a single relation STUDENT_CLASS which had the attributes for students as well as the attributes for classes, we would not be able be able to insert a student without class information or a class without student information.

Deletion Anomalies are similar to insertion anomalies, but occur when removing certain data causes additional data to be lost unintentionally. Using the above example, if there exists a single STUDENT_CLASS tuple and we try to remove that last student, the class data will also be lost.

Modification/Update Anomalies occur when attempting to make a change to a single attribute, because of redundant data, that change must be made to several attributes. If a there are 20 students in a certain class, there would be 20 STUDENT_CLASS tuples. If we were to update the class information, for example the room number, that value would have to be changed for each of the 20 STUDENT_CLASS tuples.

Normal Forms of Relations

1. **EMPLOYEE**(Employee_id, First_name, Last_name, Phone, Active)

   The EMPLOYEE relation is in third normal form.

2. **EMP_PAYMENT_RECORD**(Payment_id, Employee_id, Payment_amount, Payment_date)

The EMP_PAYMENT_RECORD relation is in third normal form.

3. **HAS_RATE**(Employee_id, Position, Start_date, End_date)

   The HAS_RATE relation is in third normal form.

4. **COMMISSION_RATE**(Position, Rate)

   The COMMISSION_RATE relation is in Boyce-Codd normal form.

5. **CUSTOMER**(Customer_id, First_name, Last_name, Phone)

   The CUSTOMER relation is in third normal form.

6. **DID_WORK**(Ticket_id, Employee_id, Start_time, End_time)

   The DID_WORK relation is in third normal form.

7. **TICKET**(Ticket_id, Location_id, Problem_id, Estimated_start, Estimated_end, Invoice_number, Invoice_total, Invoice_date, Invoice_desc, Payment_type, Paid_emp)

The TICKET relation is in second normal form. To make this relation in third normal form, we break it into the following relations:

TICKET(Ticket_id, Location_id, Problem_id, Estimated_start, Estimated_end, Invoice_number, Invoice_total, Invoice_date, Invoice_desc, Pay_method_id, Paid_emp)

PAYMENT_METHOD(Pay_method_id, Pay_type)

8. **LOCATION**(Location_id, Customer_id, Address, City, Zipcode, State)

The LOCATION relation is in second normal form but not third normal form. Zipcode is functionally dependent on Location_id and City and State are functionally dependent on Zipcode, therefore City and State are transitively dependent on Location_id. To solve this, we must split the LOCATION into two relations.

LOCATION(Location_id, Customer_id, Address, Zipcode)

AREA(Zipcode, City, State)

9. **NON_COMM_ITEM**(Item_id, Item_desc, Item_amount)

   The NON_COMM_ITEM relation is in third normal form.

10. **PARTS_USED**(Ticket_id, Item_id)

The PARTS_USED relation is in Boyce-Codd normal form.

11. **PROBLEM**(Problem_id,Work_type)

    The PROBLEM relation is in Boyce-Codd normal form.

5.2 Postgres

PostgresSQL or simply Postgres is an object relational database management system. As a database management system Postgres handles large and small workloads that deal with many concurrent users. Postgres is installed by default on macOS servers and run on Windows and Linux operating systems as well.

Postgres has updated views that stores triggers, foreign keys and supports stored procedures. Postgres initial release was over 21 years ago and it is still being used and updated today. Postgres supports a wide range of data types for the developers to use, such as:

- Boolean
- Arbitrary precision numerics
- Character (text, varchar, char)
- Binary
- Date/time (timestamp/time with/without timezone, date, interval)
- Money
- Enum
- Bit strings
- Text search type
- Composite
- HStore (an extension enabled key-value store within PostgreSQL)
- Arrays (variable length and can be of any data type, including text and composite types) up to 1 GB in total storage size
- Geometric primitives
- IPv4 and IPv6 addresses

5.2.1 Postgres Schema Objects

Schema Objects are used

- To allow many users to use one database.
- To organize database objects into logical groups.
- Control privileges between various schemas.
- Store data types, functions, and operators.

Postgres Schema Syntax

CREATE SCHEMA myschema;

Schemas tables can be accessed with a dot operator

schema.table

Schemas can be dropped using DROP

DROP SCHEMA myschema;

5.3 Relation Instances

Relational Schema Objects Contents

- WAS_PAID Assigned to EMPLOYEE
- HAS_RATE EMPLOYEE can have many COMMISON_RATE
- DID_WORK EMPLOYEE can work on many TICKET
- PARTS_USED PARTS may or may not be used in TICKET
- OWNS_LOC CUSTOMER can have many LOCATION

Employee

```
MariaDB [dalden]> DESC Employee;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| Employee_id | int(11)     | NO   | PRI | NULL    |       |
| First_name  | varchar(50) | NO   |     | NULL    |       |
| Last_name   | varchar(50) | NO   |     | NULL    |       |
| Phone       | int(11)     | NO   |     | NULL    |       |
| Active      | tinyint(1)  | YES  |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

Has_rate

```
MariaDB [dalden]> DESC Has_rate;
+-------------+-------------+------+-----+---------------------+-----------------------------+
| Field       | Type        | Null | Key | Default             | Extra                       |
+-------------+-------------+------+-----+---------------------+-----------------------------+
| Employee_id | int(11)     | NO   |     | NULL                |                             |
| Position    | varchar(50) | NO   |     | NULL                |                             |
| Start_date  | timestamp   | NO   |     | CURRENT_TIMESTAMP   | on update CURRENT_TIMESTAMP |
| End_date    | timestamp   | NO   |     | 0000-00-00 00:00:00 |                             |
+-------------+-------------+------+-----+---------------------+-----------------------------+
```

Commisson_rate

```
MariaDB [dalden]> DESC Commisson_rate;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| Position | varchar(50) | NO   | PRI | NULL    |       |
| Rate     | float(5,2)  | NO   |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

Emp_payment_record

```
MariaDB [dalden]> DESC Emp_payment_record;
+----------------+-------------+------+-----+---------------------+-----------------------------+
| Field          | Type        | Null | Key | Default             | Extra                       |
+----------------+-------------+------+-----+---------------------+-----------------------------+
| Payment_id     | int(11)     | NO   | PRI | NULL                |                             |
| Employee_id    | int(11)     | NO   |     | NULL                |                             |
| Payment_amount | float(11,2) | NO   |     | NULL                |                             |
| Payment_date   | timestamp   | NO   |     | CURRENT_TIMESTAMP   | on update CURRENT_TIMESTAMP |
+----------------+-------------+------+-----+---------------------+-----------------------------+
```

Commisson_rate

```
MariaDB [dalden]> DESC Commisson_rate;
+----------+-------------+------+-----+---------+-------+
| Field    | Type        | Null | Key | Default | Extra |
+----------+-------------+------+-----+---------+-------+
| Position | varchar(50) | NO   | PRI | NULL    |       |
| Rate     | float(5,2)  | NO   |     | NULL    |       |
+----------+-------------+------+-----+---------+-------+
```

Customer

```
MariaDB [dalden]> DESC Customer;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| Customer_id | int(11)     | NO   | PRI | NULL    |       |
| First_name  | varchar(50) | YES  |     | NULL    |       |
| Last_name   | varchar(50) | NO   |     | NULL    |       |
| Phone       | tinyint(4)  | NO   |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

Did_work

```
MariaDB [dalden]> DESC Did_work;
+-------------+-----------+------+-----+---------------------+-----------------------------+
| Field       | Type      | Null | Key | Default             | Extra                       |
+-------------+-----------+------+-----+---------------------+-----------------------------+
| Ticket_id   | int(11)   | NO   | PRI | NULL                |                             |
| Employee_id | int(11)   | NO   |     | NULL                |                             |
| Start_time  | timestamp | NO   |     | CURRENT_TIMESTAMP   | on update CURRENT_TIMESTAMP |
| End_time    | timestamp | NO   |     | 0000-00-00 00:00:00 |                             |
+-------------+-----------+------+-----+---------------------+-----------------------------+
```

Ticket

```
MariaDB [dalden]> DESC Ticket;
+-----------------+---------------+------+-----+---------------------+-----------------------------+
| Field           | Type          | Null | Key | Default             | Extra                       |
+-----------------+---------------+------+-----+---------------------+-----------------------------+
| Ticket_id       | int(11)       | NO   | PRI | NULL                |                             |
| Location_id     | int(11)       | NO   |     | NULL                |                             |
| Problem_id      | int(11)       | NO   |     | NULL                |                             |
| Estimated_start | timestamp     | NO   |     | CURRENT_TIMESTAMP   | on update CURRENT_TIMESTAMP |
| Estimated_end   | timestamp     | NO   |     | 0000-00-00 00:00:00 |                             |
| Invoice_number  | int(11)       | NO   |     | NULL                |                             |
| Invoice_total   | float(11,2)   | YES  |     | NULL                |                             |
| Invoice_date    | timestamp     | NO   |     | 0000-00-00 00:00:00 |                             |
| Invoice_desc    | varchar(1000) | NO   |     | NULL                |                             |
| Paid_emp        | tinyint(1)    | YES  |     | NULL                |                             |
| Customer_id     | tinyint(4)    | YES  |     | NULL                |                             |
| Pay_method_id   | int(11)       | YES  |     | NULL                |                             |
+-----------------+---------------+------+-----+---------------------+-----------------------------+
```

Payment_method

```
MariaDB [dalden]> DESC Payment_method;
+---------------+--------------+------+-----+---------+----------------+
| Field         | Type         | Null | Key | Default | Extra          |
+---------------+--------------+------+-----+---------+----------------+
| Pay_method_id | int(11)      | NO   | PRI | NULL    | auto_increment |
| Pay_type      | varchar(255) | YES  |     | NULL    |                |
+---------------+--------------+------+-----+---------+----------------+
```

Location

```
MariaDB [dalden]> DESC Location;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| Location_id | int(11)     | NO   | PRI | NULL    |       |
| Address     | varchar(100)| NO   |     | NULL    |       |
| City        | varchar(100)| NO   |     | NULL    |       |
| Zipcode     | varchar(5)  | NO   |     | NULL    |       |
| State       | varchar(2)  | NO   |     | NULL    |       |
| Customer_id | int(11)     | YES  |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

Area

```
MariaDB [dalden]> DESC Area;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| City    | varchar(255)| YES  |     | NULL    |       |
| Zipcode | int(11)     | YES  |     | NULL    |       |
| State   | varchar(10) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+
```

Non_comm_item

```
MariaDB [dalden]> DESC Non_comm_item;
+-------------+-------------+------+-----+---------+-------+
| Field       | Type        | Null | Key | Default | Extra |
+-------------+-------------+------+-----+---------+-------+
| Item_id     | int(11)     | NO   | PRI | NULL    |       |
| Item_desc   | varchar(255)| NO   |     | NULL    |       |
| item_amount | float(11,2) | NO   |     | NULL    |       |
+-------------+-------------+------+-----+---------+-------+
```

Parts_used

```
MariaDB [dalden]> DESC Parts_used;
+---------+---------+------+-----+---------+-------+
| Field   | Type    | Null | Key | Default | Extra |
+---------+---------+------+-----+---------+-------+
| Item_id | int(11) | NO   |     | NULL    |       |
| Ticket  | int(11) | NO   |     | NULL    |       |
+---------+---------+------+-----+---------+-------+
```

Problem

```
MariaDB [dalden]> DESC Problem;
+------------+-------------+------+-----+---------+----------------+
| Field      | Type        | Null | Key | Default | Extra          |
+------------+-------------+------+-----+---------+----------------+
| Problem_id | int(11)     | NO   | PRI | NULL    | auto_increment |
| Work_type  | varchar(100)| NO   |     | NULL    |                |
+------------+-------------+------+-----+---------+----------------+
```

5.4 SQL Queries

1. List all Employees that worked on tickets for every type of problem

```
SELECT employee.*
FROM employee e, did_work d
WHERE e.employee_id = d.employee_id AND
NOT EXISTS (
        SELECT *
        FROM problem p
        WHERE NOT EXISTS (
                SELECT *
                FROM ticket t
                WHERE t.problem_id = p.problem_id AND d.ticket_id = t.ticket_id
                )
        )
```

2. List all Employees that worked on at least one invoice greater than $2000.00

```
SELECT employee.*
FROM employee a, ticket b, did_work c
WHERE a.employee_id = c.employee_id AND b.ticket_id = c.ticket_id AND b.is_estimate = false
AND b.invoice_total > 2000
ORDER BY employee.lastName
```

3. List all tickets which have estimates which were given by “John Smith” since 2017

```
SELECT ticket_id
FROM ticket a, employee b, did_work c, payment_method d
WHERE a.ticket_id = c.ticket_id AND c.employee_id = b.employee_id AND  b.Name = “John
Smith” AND c.invoice_date BETWEEN ‘2017-01-01’ AND ‘2017-12-31’ AND d.pay_type =
“Estimate” AND d.pay_method_id = a.pay_method_id
```

4. List all Employees which were promoted in less than one year

```
SELECT * FROM employee e, has_rate h, has_rate h2
WHERE e.employee_id = h.employee_id AND e.employee_id = h2.employee_id AND h.position
!= h2.position AND h.start_date - h2.start_date < 365
ORDER BY has_rate.position
```

5. List all tickets which have invoices which are greater than $2000.00 and were completed in single day

```
SELECT *
FROM ticket t, did_work d
WHERE t.ticket_id = d.ticket_id AND t.is_estimate = false AND t.invoice_total > 2000 AND
(d.end_time - d.start_time) < 1
```

6. List all customers which have had tickets only in 2015

```
SELECT *
```

```
FROM customer c
WHERE NOT EXISTS (
        SELECT *
        FROM  location l
        WHERE c.customer_id = l.customer_id
        AND NOT EXISTS (
                SELECT *
                FROM ticket t
                WHERE t.location_id = l.location_id AND t.Invoice_date BETWEEN '2015-01-01'
                AND '2015-12-31'
                )
        )
```

7. List all locations in the 93312 zip code which have no more than 1 problem

```
SELECT *
FROM location l, ticket t
WHERE zipcode = 93312 AND l.location_id = t.location_id AND
NOT EXISTS (
        SELECT *
        FROM ticket t2
        WHERE t2.location_id = l.location_id AND t.problem_id != t2.problem_id
ORDER BY location.zipcode
```

8. List the employees who have had every position

```
SELECT first_name, last_name
FROM employee e
WHERE NOT EXISTS (
        SELECT *
        FROM commission_rate c
        WHERE NOT EXISTS (
                SELECT *
                FROM has_rate h
                WHERE h.employee_id = e.employee_id AND h.position = c.position
                )
        )
```

9. List all tickets with work type "Sewer Repair" which were given an estimate

```
SELECT ticket.ticket_id, ticket.invoice_number
FROM ticket t NATURAL JOIN problem p
WHERE p.work_type = "Sewer Repair" AND t.is_estimate = true
ORDER BY ticket.invoice_number
```

10. List the problems which were associated with the most expensive tickets in 2017

```
SELECT problem.work_type
FROM problem p NATURAL JOIN ticket t1
```

```
WHERE NOT EXISTS (
        SELECT *
        FROM ticket t2
        WHERE t2.invoice_total > t1.invoice_total;
        )
```

5.5 Data Loader

Data Loading Methods

After creating relations, Postgres offers three methods of loading data into the tables that have been created. The first method is an INSERT INTO command followed by the values you wish to fill the table with. The second method is an INSERT INTO command followed by a SELECT query which allows you to add data from another table into the new table. The third method is the COPY command which allows you to copy data from a file and put it into the new table.

INSERT INTO Using VALUES

To insert new values into a table, the first method is used with the following syntax:

INSERT INTO <*table_name*> (<*column_names*>) VALUES (<*value*>)

This inserts data into '<table_name>'. '<column_names>' is an optional expression which specifies the list of columns of the table. 'VALUES' precedes a list of values to be inserted into the table as data and '<values>' specifies the list of values to be inserted into the columns, one value per column.

Example:

```
INSERT INTO employee (Employee_id, First_name, Last_name, Phone, Active)
VALUES (12, 'John', 'Doe', 6613334389, true)
```

INSERT INTO Using Query

To insert values into a table from another table, the second method is used with the following syntax:

INSERT INTO <*table_name*> (<*column_name*s>) <query>;

This inserts data '<table_name>' similar to the first method, with '<column_names>' still being an optional expression; however, with an additional field for a query to retrieve the values from another relation.

Example:

```
INSERT INTO parts_used (ticket_id, item_id)
        SELECT ticket_id, item_id FROM ticket, non_comm_item
        WHERE ticket.invoice_number = 80113 AND comm_item.item_id = 3;
```

COPY From File

To insert values into a table from an external file, the COPY command is used. The COPY command is stricter than the INSERT INTO command, but is much faster. It uses the following syntax:

COPY <table_name> FROM '<filename>' DELIMITER '<delimiter>' WITH NULL AS '<null string>';

This copies data into '<table_name>' from a file specified as '<filename>'. The '<delimiter>' field specifies the character used to parse the data, usually "," or ";". The '<null string>' field describes the string which will be read as NULL values when copying the data into the table.

Example:
COPY customer FROM '/data/customers.sql' DELIMITER ',' WITH NULL AS 'NULL';

COPY To File

PSQL also provides a tool for exporting database data to a file. To do this, we use the COPY command followed by a 'TO' instead of 'FROM' after the table name. The syntax is similar to the COPY FROM command:

COPY <table_name> TO '<filename>' DELIMITER '<delimiter>' WITH NULL AS '<null string>';

This copies data from '<table_name>' to a file specified as '<filename>'. The '<delimiter>' field and '<null string>' field are similar to the COPY FROM example.

Example:
COPY customer TO '/data/customers.sql' DELIMITER ',' WITH NULL AS 'NULL';

Java DataLoader Program

The DataLoader program takes in a text file with data, then creates new tables from that data. It first prompts the user for a file name where data is currently located, then stores the input in 'fileName'. The program then reads each line in a loop. In the loop, the two main functions ran are 'buildPreparedStatement()' and 'addRecordToCurrentTable()'. The first function, 'buildPreparedStatement()', prepares the SQL statement to be executed. It first separates the name of the table into 'tableName' and each attribute value by a specified delimiter, 'colSepChars', into tokens which are appended to an SQL INSERT INTO command. Essentially, this function prepares each line into a SQL statement in order to create a record of the data. The program then runs the second function, 'addRecordToCurrentTable()', in order to add the record into the table. This function goes through

each column to make sure the format is correct for timestamp, time, and date data types. It then executes the SQL statement prepared earlier, which creates a record of the data and inserts it into the appropriate table.

The functionality I've added to the DataLoader program allows the user to enter a range of rows they would like to insert into their table, if for some reason they would like to enter only some data from a file. It first prompts the user if they would like to enter a range, stored in the boolean 'hasRange', then if the user answers 'true', allows the user to enter a beginning row and ending row. If the user entered a range, all other rows are skipped when inserting records of tables.

```
public static void main(String argv[]) throws IOException {
   String user = null, passwd = null;
   user = ScreenIO.promptForString(  "        Oracle user name: ");
   passwd = ScreenIO.promptForString("    Oracle user password: ");
   DataLoader ldr = new DataLoader(user, passwd); // get connected, and create stmt.
   ldr.colSepChars = ScreenIO.promptForString("   Colum Separating char: ");

   boolean    succ;
   String  line = null, fileName = null;
   String  upperCaseLine = null;
   fileName = ScreenIO.promptForString("Enter the data file name: ");
   BufferedReader spFile = new BufferedReader(new FileReader(fileName));

    //Prompts user if they would like to define a range of rows to insert. If false, program enters all rows
   int minRange, maxRange;
   boolean hasRange = Boolean.parseBoolean(ScreenIO.promptForString("Enter range of rows to insert?(true/false): "));
   if (hasRange)
   {
     minRange =  Integer.parseInt(ScreenIO.promptForString("Enter the beginning row: "));
     maxRange =  Integer.parseInt(ScreenIO.promptForString("Enter the ending row: "));
   }

   while ( (line = spFile.readLine()) != null ) {
  if (hasRange) //if user entered range of rows, only insert those rows. Otherwise program continues normally
  {
     if (ldr.lineNo < minRange || ldr.lineNo > maxRange)
     {
         ldr.lineNo ++;
         continue; // skip lines which are not within the user defined range
     }
  }
```

6.1 Postgres PL/pgSQL

PL/pgSQL is a procedural programming language which is supported by PostgreSQL database management systems. PL/pgSQL allows for the use of functions, trigger functions, control structures such as loops, and performing more complex computations. The primary advantage to using PL/pgSQL is efficiency. When communication is made between the client and server, each SQL statement is executed one at a time on the server side when a query is sent by the client. The client then must wait until the statement is executed and returned before the next query is handled. With PL/pgSQL, the client can send the entire group of queries within a function to be handled by the server, reducing the number of required communications between the client and server. Another benefit is that the functions are reusable.

The program structure of PL/pgSQL fits within a block. An example of the syntax is as follows:

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

Where the label is an optional field used to identify the block. Variables can be defined as declarations and are terminated by semicolons. Statements are placed between BEGIN and END and are also terminated by semicolons. The entire block must be terminated by a semicolon as well.

6.1.1 Stored Procedures

In PL/pgSQL, stored procedures allow users to define custom functions that are stored in the database to be called upon using Postgres. Stored procedures have a name which is used to call the function, parameters which may be passed into the function as arguments, statements, and return a value of a valid datatype (integer, boolean, trigger, table, etc.).

The following is an example to demonstrate the syntax of stored procedures:

```
CREATE OR REPLACE FUNCTION double (i integer)
RETURNS integer
AS $$
BEGIN
        RETURN i * 2;
END;
$$ LANGUAGE plpgsql;
```

The name of the stored procedure in the above example is double(int), it takes in an integer as an argument and returns that integer multiplied by 2.

6.1.2 Packages

Postgres allows for multiple SQL objects to be collected into a single package. This package is called an extension and is useful for simplifying and organizing the database. The primary advantage of packages is that when multiple objects are packaged together, PostgreSQL recognizes that these objects are of the same package, so operations can easily be done to every object within that package at the same time. For example, you can add, modify, or drop entire packages of objects in one statement.

To install a new extension, the CREATE EXTENSION <extension name> command is used.

6.1.3 Triggers

In Postgres, triggers are functions created for tables which are called automatically when a specified event occurs on that table. There are insert, update, and delete events. Triggers can also be specified to occur before, after, or instead of an event takes place. For example, if a table has an after update trigger, then anytime an update is made to a row of that table, the after update trigger is automatically called after that row is updated.

The following is an example syntax of a trigger and trigger function:

```
CREATE OR REPLACE FUNCTION  dept_update()
RETURNS trigger AS
$$
BEGIN
UPDATE employee
SET employee.did = did.new
WHERE employee.did = did.old;
RETURN NEW;
END; $$ LANGUAGE plpgsql;

CREATE TRIGGER dept_update_trigger
AFTER UPDATE
ON department
FOR EACH ROW
EXECUTE PROCEDURE dept_update();
```

This trigger will occur after the update of a department. When the department id is changed, any employee that had a department id which matched the old department's id would be updated to have the new department id.

6.2 Postgres Subprograms

Stored Procedures

1. INSERT INTO stored procedure

This function is defined as addticket(int,int). It takes in an integer for the location_id and problem_id, then inserts that ticket and returns the newly created ticket_id.

```
CREATE OR REPLACE FUNCTION public.addticket(loc integer, prob integer)
 RETURNS integer
 LANGUAGE plpgsql
AS $function$
declare tmpid int; begin
insert into ticket
(location_id, problem_id)
values (loc, prob) returning ticket_id into tmpid; return tmpid; end;
$function$
```

In the following example, I call the addticket(int,int) stored procedure. I pass the integers 85 and 9 as the location_id and problem_id arguments, respectively. After the function is called, we can see the ticket_id is returned so we can easily check that this ticket was indeed inserted into our ticket table with a location_id of 85 and problem_id of 9. This stored procedure is overloaded. There is another variation which allows for other parameters to be passed, such as 'eta'; however, at minimum, the location_id and problem_id must be passed so I used this function for simplicity.

```
plumbingdb=# select addticket(85,9);
 addticket
-----------
       224
(1 row)

plumbingdb=# select * from ticket where ticket_id = 224;
 ticket_id | location_id | problem_id | eta |   status   |          received
-----------+-------------+------------+-----+------------+----------------------------
       224 |          85 |          9 |     | Unassigned | 2018-05-17 13:44:23.670377
(1 row)
```

2. DELETE FROM stored procedure

This function is defined as deleteticket(int). It takes in an integer for the ticket_id, then deletes the ticket with the matching ticket_id.

```
CREATE OR REPLACE FUNCTION public.deleteticket(id integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
delete from ticket
where ticket_id = id;
end;
$function$
```

In the following example, I delete the ticket we created in the previous example (ticket_id 224). I pass the ticket_id as the parameter, then demonstrate that the ticket no longer exists.

```
plumbingdb=# select deleteticket(224);
 deleteticket
--------------

(1 row)

plumbingdb=# select * from ticket where ticket_id = 224;
 ticket_id | location_id | problem_id | eta | status | received
-----------+-------------+------------+-----+--------+----------
(0 rows)
```

3. Average stored Procedure

This function is defined as highavginvoice(int). It takes in an integer for the variable k, then averages the highest k invoice totals. To do this, I use a subquery which selects invoice totals from the invoice table, ordered from highest to lowest, limited by k. Then my main query averages the invoice totals from the results of the subquery.

```
CREATE OR REPLACE FUNCTION public.highavginvoice(k integer)
 RETURNS numeric
 LANGUAGE plpgsql
AS $function$
declare average numeric; begin
select avg(invoice_total) into average
from (select invoice_total from invoice
order by invoice_total desc
limit k) as total; return average; end; $function$
```

In the following example, I select the top 5 invoice totals from the invoice table, so that the user may test the function by manually calculating the average from the resulting values. I then call the highavginvoice(int) stored procedure, passing 5 as the integer parameter, which returns the correct average of the listed 5 values.

```
plumbingdb=# select invoice_total from invoice
order by invoice_total desc limit 5;
 invoice_total
---------------
        795.20
        782.79
        782.26
        777.94
        755.17
(5 rows)

plumbingdb=# select highavginvoice(5);
    highavginvoice
----------------------
 778.6720000000000000
(1 row)
```

Triggers

The following triggers rely on the location and zipcode relations. These area relation contains a zipcode, city, and state. The location relation contains a primary key location_id, a customer_id which references the customer who owns the location, an address, and a zipcode which references the zipcode of the area the location is at. The relations are defined as follows:

```
plumbingdb=# \d location
                                  Table "public.location"
   Column    |         Type          |                           Modifiers
-------------+-----------------------+---------------------------------------------------------------
 location_id | integer               | not null default nextval('location_location_id_seq'::regclass)
 customer_id | integer               | not null
 address     | character varying(50) | not null
 zipcode     | character varying(5)  | not null
Indexes:
    "location_pkey" PRIMARY KEY, btree (location_id)
Foreign-key constraints:
    "location_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(customer_id)
Referenced by:
    TABLE "billing_address" CONSTRAINT "billing_address_location_id_fkey" FOREIGN KEY (location_id) REFERENCES location(location_id)
    TABLE "ticket" CONSTRAINT "ticket_location_id_fkey" FOREIGN KEY (location_id) REFERENCES location(location_id)
Tablespace: "data"

plumbingdb=# \d area
           Table "public.area"
 Column  |         Type          | Modifiers
---------+-----------------------+-----------
 zipcode | character varying(5)  | not null
 city    | character varying(50) |
 state   | character varying(2)  |
Indexes:
    "area_pkey" PRIMARY KEY, btree (zipcode)
Triggers:
    zipcodedeletetrigger AFTER DELETE ON area FOR EACH ROW EXECUTE PROCEDURE zipcodedelete()
    zipcodetrigger BEFORE UPDATE ON area FOR EACH ROW EXECUTE PROCEDURE zipcodeupdate()
Tablespace: "data"
```

1. BEFORE UPDATE trigger

   This trigger is defined as zipcodetrigger. It is a BEFORE UPDATE trigger which is created on the

area relation. When an area relation is updated, the stored procedure zipcodeupdate() is called.

```
zipcodetrigger BEFORE UPDATE ON area FOR EACH ROW EXECUTE PROCEDURE zipcodeupdate()
```

This function is a cascade update function and is defined as zipcodeupdate(). When the zipcode of an area relation is updated, this stored procedure updates the zipcode of every location which references the old zipcode to the new zipcode.

```
CREATE OR REPLACE FUNCTION public.zipcodeupdate()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
begin
update location
set zipcode = new.zipcode
where zipcode = old.zipcode;
return new; end; $function$
```

In the following example, I show that three locations are at the zipcode ‘93316’. I then update the area with the zipcode ‘93316’ to the new zipcode, ‘93306’. I think show that all three locations which were at the zipcode ‘93316’, are now at ‘93306’.

```
plumbingdb=# select * from location natural join area where zipcode = '93316';
 zipcode | location_id | customer_id |         address          |    city     | state
---------+-------------+-------------+--------------------------+-------------+-------
 93316   |         212 |          17 | 9835 Snow Rd             | Bakersfield | CA
 93316   |         213 |          19 | 1353 Jacobs Lane         | Bakersfield | CA
 93316   |         214 |          55 | 9001 Crystal Springs Way | Bakersfield | CA
(3 rows)

plumbingdb=# update area set zipcode = '93306' where zipcode = '93316';
UPDATE 1
plumbingdb=# select * from location natural join area where zipcode = '93306';
 zipcode | location_id | customer_id |         address          |    city     | state
---------+-------------+-------------+--------------------------+-------------+-------
 93306   |         212 |          17 | 9835 Snow Rd             | Bakersfield | CA
 93306   |         213 |          19 | 1353 Jacobs Lane         | Bakersfield | CA
 93306   |         214 |          55 | 9001 Crystal Springs Way | Bakersfield | CA
(3 rows)
```

2. AFTER DELETE trigger

This trigger is defined as zipcodedeletetrigger. It is an AFTER DELETE trigger which is created on the area relation. When an area object is deleted, the zipcodedelete() stored procedure is called.

```
zipcodedeletetrigger AFTER DELETE ON area FOR EACH ROW EXECUTE PROCEDURE zipcodedelete()
```

This is a cascade delete function and is named zipcodedelete(). When an area is deleted, this

function is called, and any location which matches the zipcode of the deleted area is deleted.

```
CREATE OR REPLACE FUNCTION public.zipcodedelete()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
begin
delete from location
where zipcode = old.zipcode;
return new; end; $function$
```

In the following example, I show the three locations which exist at zipcode ‘93316’. I then delete the area with that same zipcode. The cascade delete function is triggered, which goes through and deletes all three locations that were at that zipcode, which is demonstrated with a select statement.

```
plumbingdb=# select * from location where zipcode = '93316';
 location_id | customer_id |          address          | zipcode
-------------+-------------+---------------------------+---------
         212 |          17 | 9835 Snow Rd              | 93316
         213 |          19 | 1353 Jacobs Lane          | 93316
         214 |          55 | 9001 Crystal Springs Way | 93316
(3 rows)

plumbingdb=# delete from area where zipcode = '93316';
DELETE 1
plumbingdb=# select * from location where zipcode = '93316';
 location_id | customer_id | address | zipcode
-------------+-------------+---------+---------
(0 rows)
```

3. INSTEAD OF UPDATE trigger

An INSTEAD OF trigger acts on a view. When a statement is performed on the view which is associated with the INSTEAD OF trigger, the statement is performed on the base tables instead.

First, I created a view called locationView, which simply returns all columns from the natural join of location and area.

```
plumbingdb=# \d+ locationView
                    View "public.locationview"
   Column    |         Type          | Modifiers | Storage  | Description
-------------+-----------------------+-----------+----------+-------------
 zipcode     | character varying(5)  |           | extended |
 city        | character varying(50) |           | extended |
 state       | character varying(2)  |           | extended |
 location_id | integer               |           | plain    |
 customer_id | integer               |           | plain    |
 address     | character varying(50) |           | extended |
View definition:
 SELECT area.zipcode,
    area.city,
    area.state,
    location.location_id,
    location.customer_id,
    location.address
   FROM area
     JOIN location USING (zipcode);
Triggers:
    location_view_trigger INSTEAD OF UPDATE ON locationview FOR EACH ROW EXECUTE PROCEDURE location_area_view_update(
)
```

The location_view_trigger which is added to the locationView is an INSTEAD OF UPDATE trigger, which calls the stored procedure location_area_view_update() whenever the user attempts to perform an update on the locationView.

```
    location_view_trigger INSTEAD OF UPDATE ON locationview FOR EACH ROW EXECUTE PROCEDURE location_
area_view_update()
```

This function is defined as location_area_view_update(). When an update is attempted on the locationView, this stored procedure is called, which updates the zipcode of the area and locations to the new zipcode which was updated on the view.

```
CREATE OR REPLACE FUNCTION public.location_area_view_update()
 RETURNS trigger
 LANGUAGE plpgsql
AS $function$
begin
update location set zipcode = new.zipcode
where zipcode = old.zipcode;
update area set zipcode = new.zipcode
where zipcode = old.zipcode;
return new; end; $function$
```

In the following example, I select from the locationView where the zipcode is ‘93306’, which shows 3 locations. I then update the locationView, setting the zipcode from ‘93306’ to ‘93316’. This triggers the location_area_view_update() stored procedure, which updates the zipcode of the area and location from ‘93306’ to ‘93316’. This is demonstrated in the following select statement.

```
plumbingdb=# select * from locationView where zipcode = '93306';
 zipcode |    city     | state | location_id | customer_id |         address
---------+-------------+-------+-------------+-------------+-------------------------
 93306   | Bakersfield | CA    |         212 |          17 | 9835 Snow Rd
 93306   | Bakersfield | CA    |         213 |          19 | 1353 Jacobs Lane
 93306   | Bakersfield | CA    |         214 |          55 | 9001 Crystal Springs Way
(3 rows)

plumbingdb=# update locationView set zipcode = '93316' where zipcode = '93306';
UPDATE 3
plumbingdb=# select * from locationView where zipcode = '93316';
 zipcode |    city     | state | location_id | customer_id |         address
---------+-------------+-------+-------------+-------------+-------------------------
 93316   | Bakersfield | CA    |         212 |          17 | 9835 Snow Rd
 93316   | Bakersfield | CA    |         213 |          19 | 1353 Jacobs Lane
 93316   | Bakersfield | CA    |         214 |          55 | 9001 Crystal Springs Way
(3 rows)
```

6.3 Other Tools

Some advantages of the PostgreSQL database management system are that it is open source, it has a wide variety of data types supported, fully supports SQL features and follows SQL standards, supports powerful procedural languages (PL/pgSQL), and is efficient at joining many tables together. PostgreSQL is ideal when making a complex database or complex stored procedures, and when connecting with Java applications.

MS SQL is closed-source, but is designed to cater towards corporations. Using it requires more understanding of databases, but has more tools to customize security features. It offers a higher degree of control at the cost of higher resource use. MS SQL is ideal when working on a large corporate database and require a lot of control over the database.

Oracle is closed-source, but has a limited free version. It is designed to be very flexible in terms of database scaling as well as backing up the database. Oracle is ideal when creating a large database that requires a lot of scalability and needs to be used on multiple platforms.

7.1 Graphical User Interface

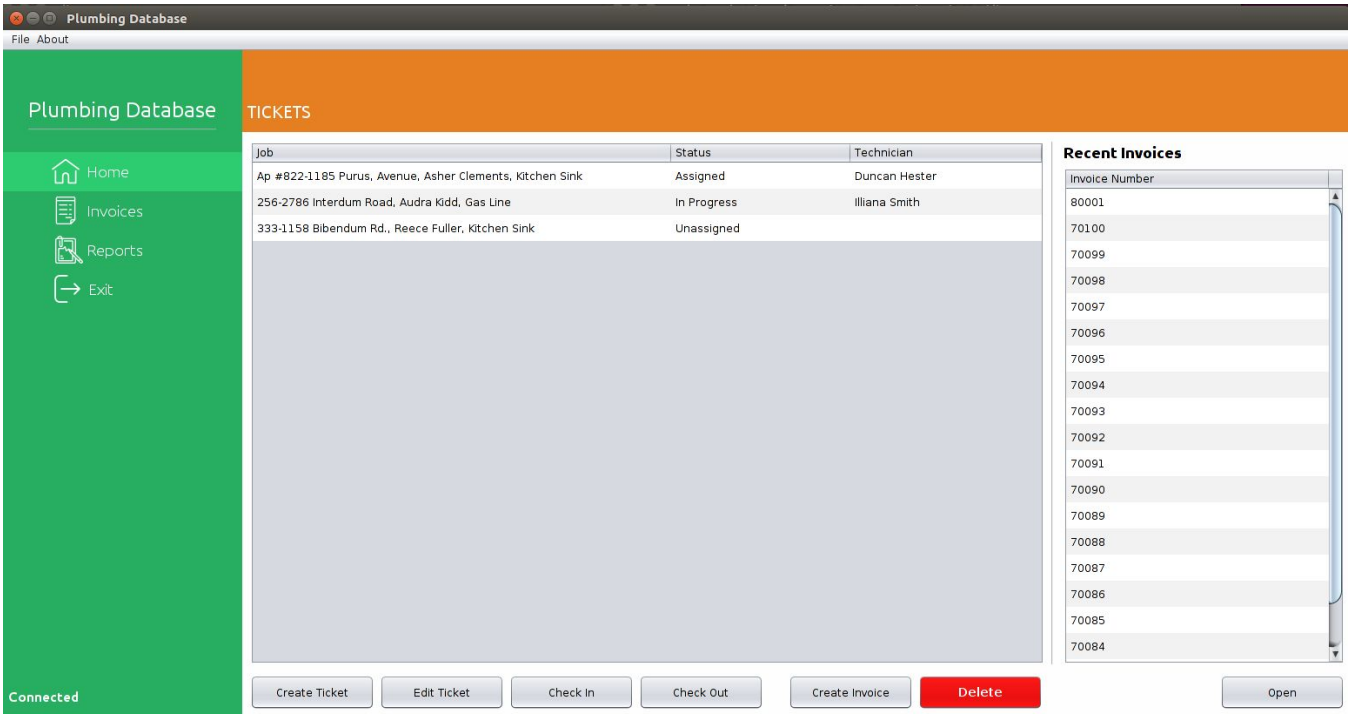
7.1.1 Application Description

My application is designed for a small plumbing business to keep track of current jobs as well as past jobs completed. The main functionalities of the application are to create, edit, and delete tickets, to create, edit, and delete invoices, and to generate an invoice report which details all relevant job information for that invoice. The program contains a simple, yet powerful and easy to understand

interface for viewing and manipulating data, as well as fast and efficient querying tools for finding data quickly.

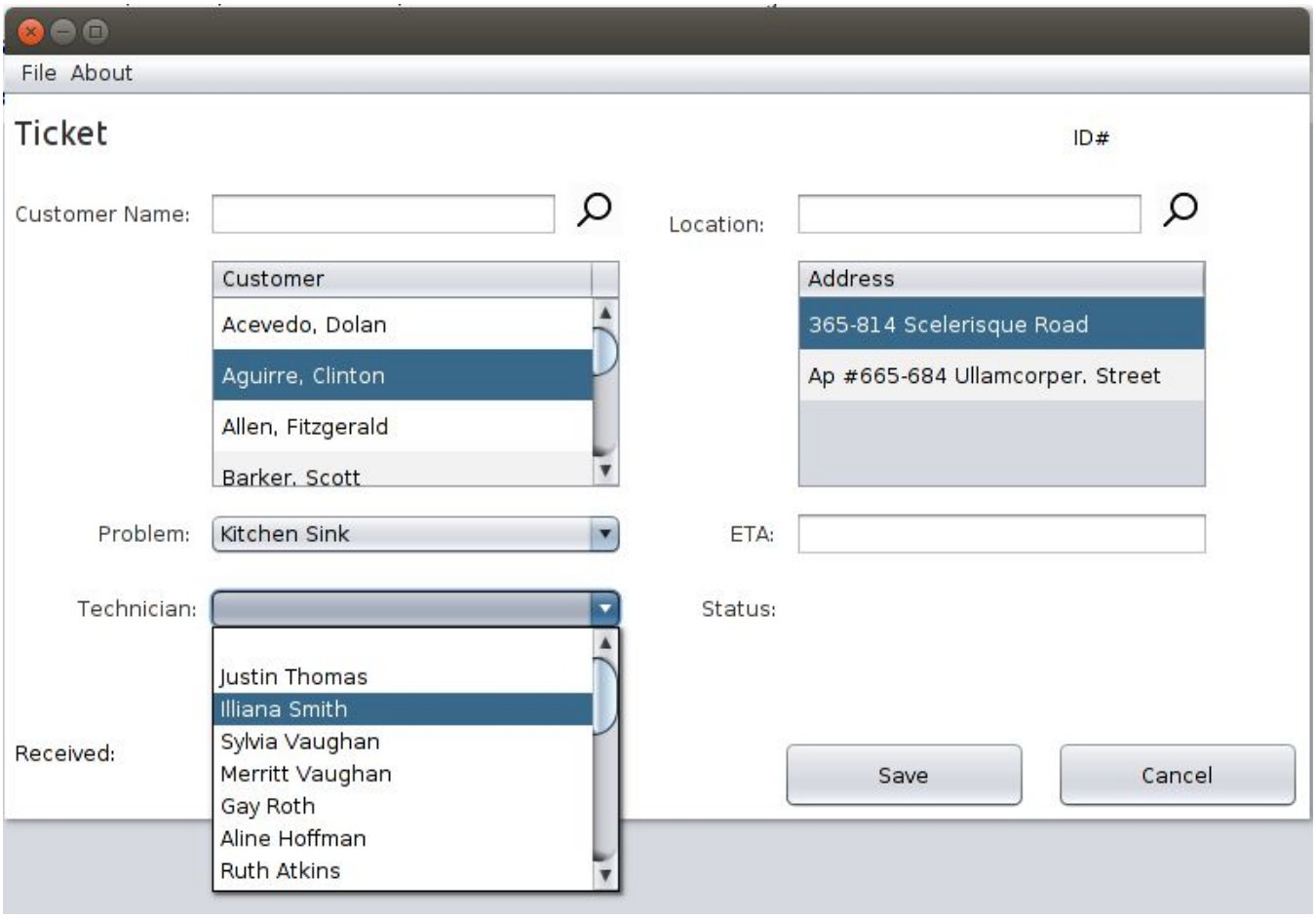
The general use of the application is as follows: A ticket is first created, specifying the customer, location, problem, and if applicable, the technician assigned to the ticket. That ticket will then show up in the tickets table with the current ticket status (either “Assigned” or “Unassigned”, depending on whether or not a technician was assigned yet). After a technician has been assigned, that technician may be checked in to the job, which changes the status to “In Progress”. Then, the technician can be checked out, which changes the ticket status to “Complete”. Then, an invoice for that job may be created for that ticket, which archives the information and keeps the ticket board filled with only current, open tickets.

7.1.2 Application Details

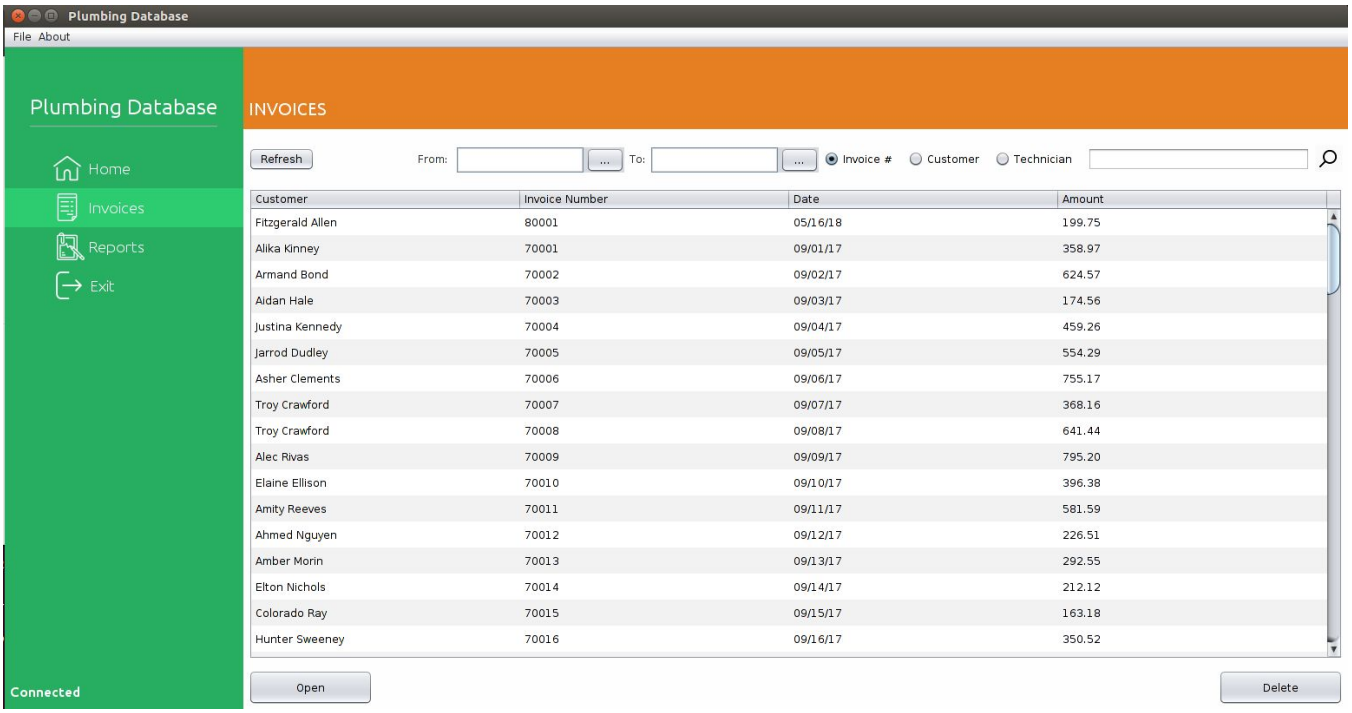


The Home panel of the application lists all current tickets which are still open. The main feature is the ticket table, which displays all open tickets with job information, current status, and assigned technicians (if applicable). There is also a “Recent Invoices” section to the right, which shows the 20 most recent invoices that have been entered. Several functionalities are available on the tickets, which are accessible by the buttons at the bottom. The “Create Ticket” button allows the user to create a new ticket. After selecting on a ticket on the ticket table, the user may click the “Edit Ticket” button to change details about that ticket. They can also select “Check In” or “Check Out” which updates the status of the ticket. After the ticket is completed by clicking “Check Out”, the user may click “Create Invoice” to generate a new invoice for that ticket. At any time, the user may click “Delete” to delete a

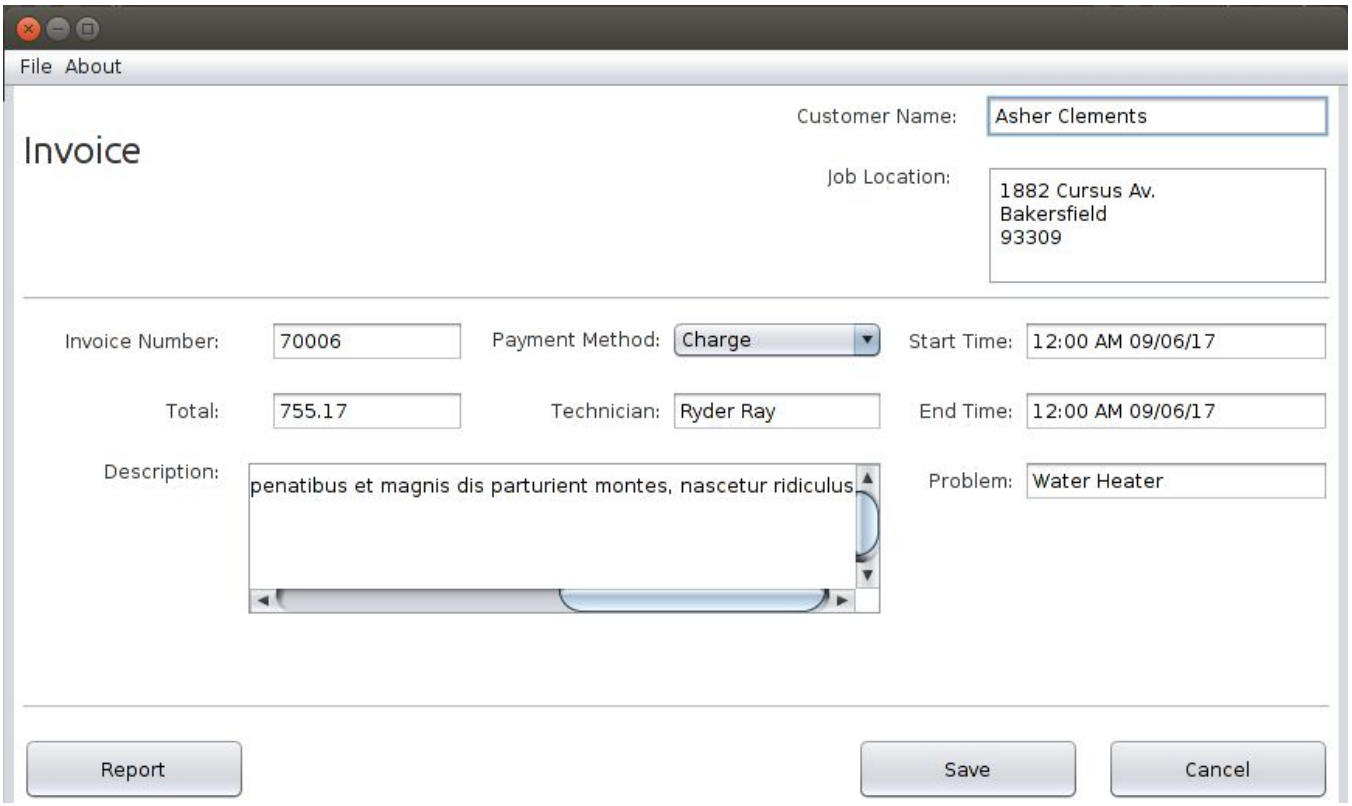
ticket. The user may also click “Open” under Recent Invoices to open an invoice in that list.



After clicking “Create Ticket” or “Edit Ticket”, the above window will open. If the user clicked “Create Ticket”, all fields will be empty and the user may select the data they wish to include in the ticket. If the user selected “Edit Ticket”, fields will be filled with data from the ticket the user wishes to edit. The “Save” button will create a new ticket, or save changes to an existing ticket if the user selected “Edit Ticket”. The Customer table contains all customers in the database, these can be filtered by typing in the above search bar. The Address table contains all locations owned by the selected customer, and may also be filtered by the above search bar. The Problem combo box contains all existing problems in the database, and the Technician combo box contains all existing technicians. The ETA field is an optional field to enter an estimated time of arrival for a ticket and the Status label simply prints out the current ticket status.



The Invoices panel lists all invoices that currently exist in the database. It contains a customer name, invoice number, date the job was completed, and amount of that invoice. There are several search features as well, such as search by date, which returns invoices only within the entered dates, or search by invoice, customer name, or technician name, which filters only invoices with matching queries. The “Refresh” button refreshes the list with entered query filters. The “Open” button opens an invoice for editing or viewing, and the “Delete” button deletes an invoice as well as the ticket it refers to.



After selecting the “Open” button in the invoice panel, the above window is opened with the information pulled from the selected invoice. It describes the Invoice number, total, and other relevant data from that invoice and ticket associated with it. Clicking “Save” will save any changes made to the invoice, and clicking “Report” will generate a pdf report of that invoice.

Robo Plumbing
3535 Bowman Ct.
Bakersfield, CA
93308

Invoice

Customer

Asher Clements
1882 Cursus Av. Bakersfield, CA 93309

Invoice#: 70006
Date: 09/06/2017

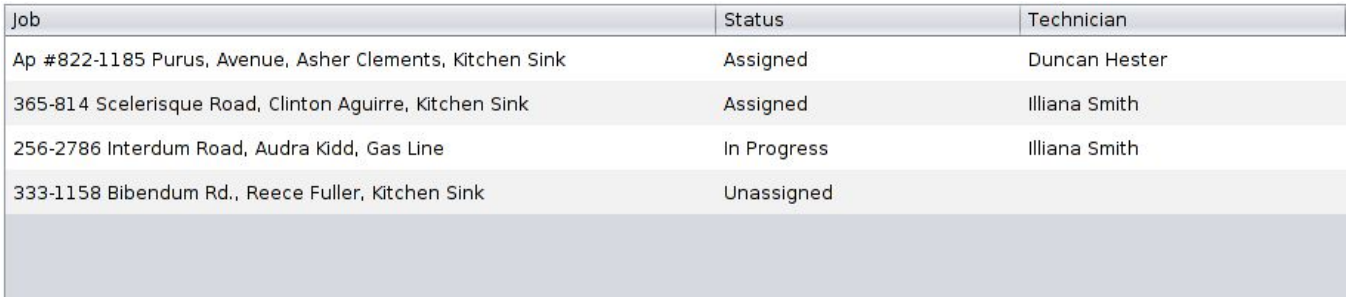
Description	Amount
sem magna nec quam. Curabitur vel lectus. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus	755.17
Amount Owed	755.17

Technician: Ryder Ray
Start Time: 09/06/2017 00:00 AM
End Time: 09/06/2017 00:00 AM

The pdf report contains a logo and company information on the top left, as well as customer information and invoice information from the invoice which the report was generated on.

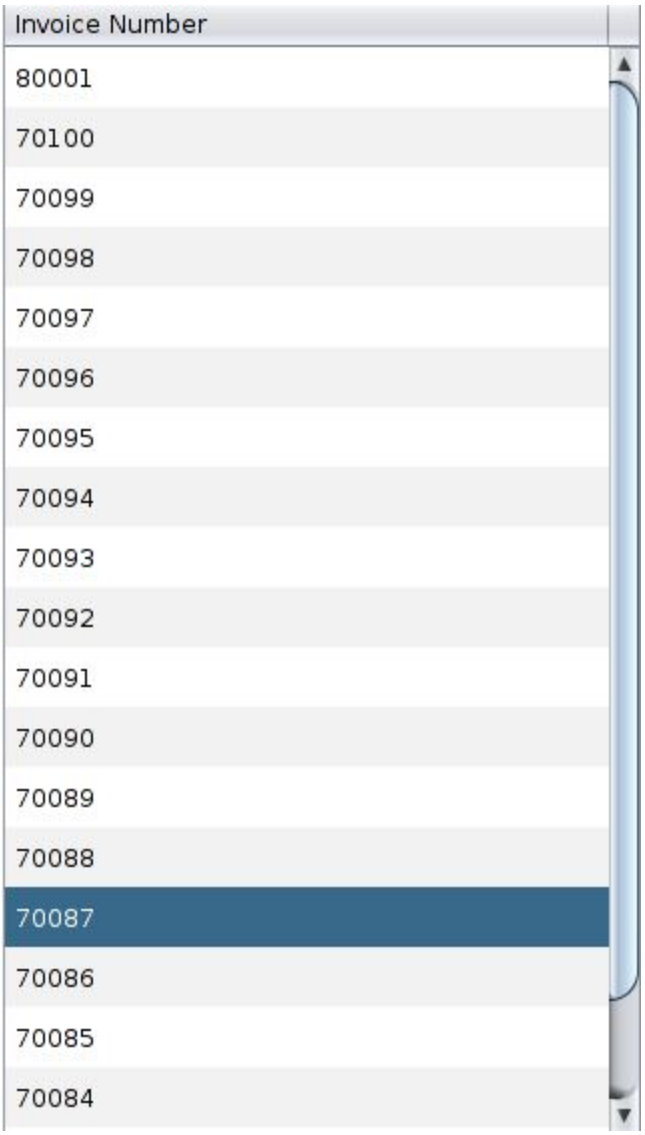
7.1.3 Tables and Views

The following is the open ticket table on the Home panel of the application. It selects all rows from the view “openticketview” and adds the returning data to the rows of the application table. The Job column contains the address, customer, and problem. The Status column describes the current ticket status. The Technician column shows the assigned technician, if any. A fourth hidden column is also included on the table, which contains the ticket_id. This is included to easily fetch the ticket_id of the selected row.

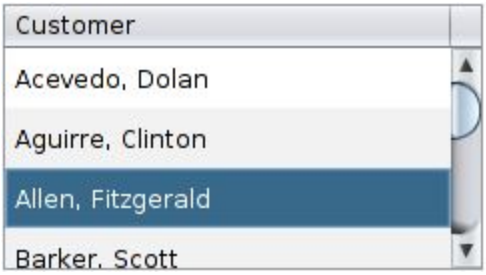


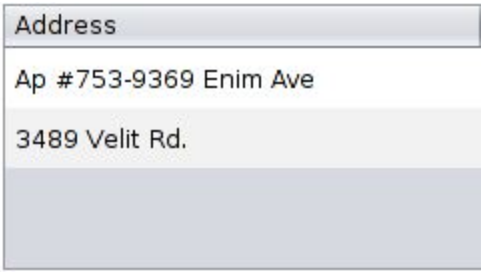
Job	Status	Technician
Ap #822-1185 Purus, Avenue, Asher Clements, Kitchen Sink	Assigned	Duncan Hester
365-814 Scelerisque Road, Clinton Aguirre, Kitchen Sink	Assigned	Illiana Smith
256-2786 Interdum Road, Audra Kidd, Gas Line	In Progress	Illiana Smith
333-1158 Bibendum Rd., Reece Fuller, Kitchen Sink	Unassigned	

The Recent Invoices table on the Home panel shows the 20 most recent invoices, listed in descending date order by invoice number. It gets the data to fill the rows from the result of a select statement on the view “recentinvoiceview”.



In the “Create Ticket” or “Edit Ticket” window, the Customer table is filled with all customers in the database from the view titled “customerview”. It contains a hidden column for the customer_id to easily fetch the selected customer’s customer_id. The Address table does not call a view, rather a stored procedure which takes in the selected customer_id as an argument to fetch only that customer’s locations. It also has a hidden column for the location_id.





The Invoices table calls the “invoiceview” to select all invoices in the database. The invoice view returns the customer, date, amount, and invoice number for each invoice.

Customer	Invoice Number	Date	Amount
Fitzgerald Allen	80001	05/16/18	199.75
Alika Kinney	70001	09/01/17	358.97
Armand Bond	70002	09/02/17	624.57
Aidan Hale	70003	09/03/17	174.56
Justina Kennedy	70004	09/04/17	459.26
Jarrod Dudley	70005	09/05/17	554.29
Asher Clements	70006	09/06/17	755.17
Troy Crawford	70007	09/07/17	368.16
Troy Crawford	70008	09/08/17	641.44
Alec Rivas	70009	09/09/17	795.20
Elaine Ellison	70010	09/10/17	396.38
Amity Reeves	70011	09/11/17	581.59
Ahmed Nguyen	70012	09/12/17	226.51
Amber Morin	70013	09/13/17	292.55
Elton Nichols	70014	09/14/17	212.12
Colorado Ray	70015	09/15/17	163.18
Hunter Sweeney	70016	09/16/17	350.52

7.2 Programming

This project consisted of a PSQL database for the back-end and a Java application for the graphical user interface.

7.2.1 Server-Side Programming

The server-side contained many views and subprograms to effectively fetch and manipulate data from the front-end.

7.2.1.1 PSQL Views

The customerview selects the first and last name from the customer relation. It is used to fill the Customer table when adding a ticket.

```
plumbingdb=# \d+ customerview
                      View "public.customerview"
   Column    |         Type          | Modifiers | Storage  | Description
-------------+-----------------------+-----------+----------+-------------
 customer_id | integer               |           | plain    |
 first_name  | character varying(35) |           | extended |
 last_name   | character varying(35) |           | extended |
View definition:
 SELECT customer.customer_id,
    customer.first_name,
    customer.last_name
   FROM customer
  ORDER BY customer.last_name;
```

The invoiceview selects the customer's first and last name, invoice number, date the invoice work was completed, and the invoice total. It is used for the invoice table when selecting an invoice.

```
plumbingdb=# \d+ invoiceview
                        View "public.invoiceview"
     Column     |         Type          | Modifiers | Storage  | Description
----------------+-----------------------+-----------+----------+-------------
 first_name     | character varying(35) |           | extended |
 last_name      | character varying(35) |           | extended |
 invoice_number | character varying(10) |           | extended |
 date           | text                  |           | extended |
 invoice_total  | numeric(10,2)         |           | main     |
View definition:
 SELECT customer.first_name,
    customer.last_name,
    invoice.invoice_number,
    to_char(did_work.end_time, 'MM/DD/YY'::text) AS date,
    invoice.invoice_total
   FROM invoice
     JOIN ticket USING (ticket_id)
     JOIN location USING (location_id)
     JOIN customer USING (customer_id)
     JOIN did_work USING (ticket_id);
```

The openticketview selects all relevant ticket and assignment information from tickets which are not closed. This is used for the main ticket table in the Home panel.

```
plumbingdb=# \d+ openticketview
                    View "public.openticketview"
   Column   |         Type          | Modifiers | Storage  | Description
------------+-----------------------+-----------+----------+-------------
 ticket_id  | integer               |           | plain    |
 address    | character varying(50) |           | extended |
 first_name | character varying(35) |           | extended |
 last_name  | character varying(35) |           | extended |
 work_type  | character varying(50) |           | extended |
 received   | text                  |           | extended |
 eta        | text                  |           | extended |
 status     | character varying(15) |           | extended |
 fname      | character varying(35) |           | extended |
 lname      | character varying(35) |           | extended |
View definition:
 SELECT ticket.ticket_id,
    location.address,
    customer.first_name,
    customer.last_name,
    problem.work_type,
    to_char(ticket.received, 'HH12:MI AM MM/DD/YY'::text) AS received,
    to_char(ticket.eta, 'HH12:MI AM MM/DD/YY'::text) AS eta,
    ticket.status,
    employee.fname,
    employee.lname
   FROM ticket
     JOIN location USING (location_id)
     JOIN customer USING (customer_id)
     JOIN problem USING (problem_id)
     LEFT JOIN did_work ON ticket.ticket_id = did_work.ticket_id
     LEFT JOIN employee ON did_work.employee_id = employee.employee_id
  WHERE ticket.status::text <> 'Closed'::text;
```

The problemview selects the problem_id and work_type from the problem relation. It is used when selecting a problem for a ticket.

```
plumbingdb=# \d+ problemview
                    View "public.problemview"
   Column   |         Type          | Modifiers | Storage  | Description
------------+-----------------------+-----------+----------+-------------
 problem_id | integer               |           | plain    |
 work_type  | character varying(50) |           | extended |
View definition:
 SELECT problem.problem_id,
    problem.work_type
   FROM problem;
```

The recentinvoice view selects the invoice number from the 20 most recent invoices for the Recent Invoices table on the home panel.

```
plumbingdb=# \d+ recentinvoiceview
                    View "public.recentinvoiceview"
     Column     |         Type          | Modifiers | Storage  | Description
----------------+-----------------------+-----------+----------+-------------
 invoice_number | character varying(10) |           | extended |
View definition:
 SELECT invoice.invoice_number
   FROM invoice
     JOIN ticket USING (ticket_id)
     JOIN did_work USING (ticket_id)
  ORDER BY did_work.end_time DESC
 LIMIT 20;
```

The technicianview selects the employee's first and last name from the employee relation. It is used when selecting a technician for a ticket.

```
plumbingdb=# \d+ technicianview
                     View "public.technicianview"
   Column    |         Type          | Modifiers | Storage  | Description
-------------+-----------------------+-----------+----------+-------------
 employee_id | integer               |           | plain    |
 fname       | character varying(35) |           | extended |
 lname       | character varying(35) |           | extended |
View definition:
 SELECT employee.employee_id,
    employee.fname,
    employee.lname
   FROM employee;
```

7.2.1.2 PSQL Stored Procedures

The addassign(int,int) function inserts a did_work relation using a ticket_id and employee_id. This keeps track of who was assigned to which ticket and when they got there and finished.

```
CREATE OR REPLACE FUNCTION public.addassign(tic integer, emp integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
insert into did_work
(ticket_id, employee_id)
values (tic, emp);
end;
$function$
```

The addinvoice() function takes in necessary fields to create an invoice and inserts a new invoice into the invoice relation. When an invoice with that number already exists, it updates the existing invoice with the passed information. It also returns the invoice_number that was just created.

```
CREATE OR REPLACE FUNCTION public.addinvoice(inv character varying, tic_id integ
er, total numeric, pay_id integer, des character varying)
 RETURNS integer
 LANGUAGE plpgsql
AS $function$
declare tmpnum varchar(10);
begin
insert into invoice
(invoice_number, ticket_id, invoice_total,
pay_method_id, invoice_desc)
values (inv, tic_id, total, pay_id, des)
on conflict (invoice_number) do update
set invoice_total = total,
pay_method_id = pay_id,
invoice_desc = des returning invoice_number into tmpnum;
return tmpnum; end; $function$
```

The addticket() function inserts a ticket using a location_id and problem_id. It also returns the ticket_id that was just created.

```
CREATE OR REPLACE FUNCTION public.addticket(loc integer, prob integer)
 RETURNS integer
 LANGUAGE plpgsql
AS $function$
declare tmpid int; begin
insert into ticket
(location_id, problem_id)
values (loc, prob) returning ticket_id into tmpid; return tmpid; end;
$function$
```

check_date() takes in an invoice number, beginning date, and ending date, and returns 0 if the invoice does not exist within those dates or 1 if it does. The purpose of this function is essentially to act as a boolean function to check if an invoice occurred within two dates. This is used for the date filter.

```
CREATE OR REPLACE FUNCTION public.check_date(id character varying, begining date
, ending date)
 RETURNS integer
 LANGUAGE plpgsql
AS $function$
declare exists integer;
begin
select count("invoice_number") into exists
from invoice natural join ticket natural join did_work
where invoice.invoice_number = id and
did_work.end_time >= begining and
did_work.end_time <= ending;
return exists; end; $function$
```

The check_in() function takes in a ticket_id, then updates the start time associated with that ticket. The start time is the time at which the function is called.

```
CREATE OR REPLACE FUNCTION public.check_in(id integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
update did_work
set start_time = now()
where ticket_id = id;
end; $function$
```

The check_out() function takes in a ticket_id, then updates the end time associated with that ticket. The end time is the time at which the function is called.

```
CREATE OR REPLACE FUNCTION public.check_out(id integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
update did_work
set end_time = now()
where ticket_id = id;
end; $function$
```

The delete_assigned() function deletes the did_work object for the specified ticket passed in by the ticket_id.

```
CREATE OR REPLACE FUNCTION public.delete_assigned(id integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
delete from did_work
where did_work.ticket_id = id;
end; $function$
```

The deleteinvoice() function deletes an invoice from the invoice number passed in as an argument.

```
CREATE OR REPLACE FUNCTION public.deleteinvoice(id character varying)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
delete from invoice
where invoice_number = id;
end;
$function$
```

deleteticket() takes in a ticket_id as an argument, then deletes that ticket.

```
CREATE OR REPLACE FUNCTION public.deleteticket(id integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
delete from ticket
where ticket_id = id;
end;
$function$
```

The get_tech() function returns the name of the technician associated with an invoice.

```
CREATE OR REPLACE FUNCTION public.get_tech(id character varying)
 RETURNS character varying
 LANGUAGE plpgsql
AS $function$
declare name varchar;
begin
select employee.fname || ' ' || employee.lname
into name from invoice natural join ticket
natural join did_work natural join employee
where invoice.invoice_number = id;
return name; end; $function$
```

The get_invoice_info() function gets ticket info for creating an invoice from the selected ticket.

```
CREATE OR REPLACE FUNCTION public.get_invoice_info(id integer)
 RETURNS TABLE(first_name character varying, last_name character varying, addres
s character varying, city character varying, zipcode character varying, fname ch
aracter varying, lname character varying, work_type character varying, start_tim
e text, end_time text)
 LANGUAGE plpgsql
AS $function$
begin
return query
select
 customer.first_name, customer.last_name, location.address, area.city,
 area.zipcode, employee.fname, employee.lname, problem.work_type,
 to_char(did_work.start_time, 'HH12:MI AM MM/DD/YY'::text),
 to_char(did_work.end_time, 'HH12:MI AM MM/DD/YY'::text)
from ticket natural join location natural join customer natural join area
 natural join problem natural join did_work inner join employee
 on did_work.employee_id = employee.employee_id
where ticket_id = id;
 end; $function$
```

The get_invoice_info_full() function gets the invoice info from a selected invoice when opening it. When opening an invoice, both the get_invoice_info() and get_invoice_info_full() functions are called.

```
CREATE OR REPLACE FUNCTION public.get_invoice_info_full(id character varying)
 RETURNS TABLE(tot numeric, paymeth integer, des character varying)
 LANGUAGE plpgsql
AS $function$
begin
return query
select
invoice.invoice_total,
invoice.pay_method_id,
invoice.invoice_desc
from invoice
where invoice.invoice_number = id;
end; $function$
```

The get_invoice_report() function takes in an invoice number as a parameter, then returns a table with all the relevant invoice and ticket information for printing a report.

```
CREATE OR REPLACE FUNCTION public.get_invoice_report(id character varying)
 RETURNS TABLE(invnum character varying, cfname character varying, clname character varying, caddress character varying, cci
ty character varying, cstate character varying, czip character varying, des character varying, total numeric, paymethod char
acter varying, efname character varying, elname character varying, starttime timestamp without time zone, endtime timestamp
without time zone)
 LANGUAGE plpgsql
AS $function$
begin
return query
select
invoice.invoice_number,
customer.first_name,
customer.last_name,
location.address,
area.city,
area.state,
area.zipcode,
invoice.invoice_desc,
invoice.invoice_total,
payment_method.payment_type,
employee.fname,
employee.lname,
did_work.start_time,
did_work.end_time
from invoice natural join ticket natural join did_work
natural join employee,
location natural join area natural join customer, payment_method
where invoice.invoice_number = id and
invoice.ticket_id = ticket.ticket_id and
ticket.ticket_id = did_work.ticket_id and
ticket.location_id = location.location_id and
invoice.pay_method_id = payment_method.pay_method_id;
end; $function$
```

The get_locations() function takes in a customer_id and returns all locations owned by that customer. This is useful when selecting a job location, you can limit the total locations to only the selected customer.

```
CREATE OR REPLACE FUNCTION public.get_locations(cust integer)
 RETURNS TABLE(loc_id integer, addr character varying, zip character varying, city character varying, sta
te character varying)
 LANGUAGE plpgsql
AS $function$
BEGIN
 return query select
 location_id, address, zipcode, area.city, area.state
from location natural join customer natural join area
where customer_id = cust;
END; $function$
```

The getreceived() function returns the time a ticket was received.

```
CREATE OR REPLACE FUNCTION public.getreceived(id integer)
 RETURNS text
 LANGUAGE plpgsql
AS $function$
declare tmptime text;
begin
select to_char(received, 'HH12:MI AM MM/DD/YY'::text) into tmptime
from ticket
where ticket_id = id; return tmptime;
end; $function$
```

The get_status() function takes in a ticket_id and returns the status of that ticket.

```
CREATE OR REPLACE FUNCTION public.get_status(id integer)
 RETURNS character varying
 LANGUAGE plpgsql
AS $function$
declare stat varchar;
begin
select ticket.status into stat from ticket
where ticket.ticket_id = id;
return stat;
end; $function$
```

The get_ticket_from_invoice() function takes in an invoice number and returns the ticket_id associated with it. This is used on application tables where ticket information is needed, but only the invoice number is given.

```
CREATE OR REPLACE FUNCTION public.get_ticket_from_invoice(id character varying)
 RETURNS integer
 LANGUAGE plpgsql
AS $function$
declare result int;
begin
select invoice.ticket_id into result
from invoice
where invoice.invoice_number = id;
return result;
end;$function$
```

The get_ticket_info() function takes in the ticket_id and returns the information associated with that ticket. This is used when opening an existing ticket.

```
CREATE OR REPLACE FUNCTION public.get_ticket_info(id integer)
 RETURNS TABLE(c_id integer, l_id integer, p_id integer, e_id integer, est timestamp without time zone, s
tat character varying)
 LANGUAGE plpgsql
AS $function$
begin
return query select
customer_id, location_id, problem_id, employee_id,
eta, status
from ticket natural join location natural join customer
natural join problem left join did_work
on did_work.ticket_id = ticket.ticket_id
where ticket.ticket_id = id;
end; $function$
```

The set_status() function is used to manually set a ticket's status. It takes in the ticket_id of the ticket you want to update, as well as a string of what the new status is.

```
CREATE OR REPLACE FUNCTION public.set_status(id integer, stat character varying)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
update ticket
set status = stat
where ticket_id = id;
end; $function$
```

updateassign() takes in a ticket_id and employee_id and updates the employee_id for that ticket. It is used when changing technicians on an existing ticket.

```
CREATE OR REPLACE FUNCTION public.updateassign(tic integer, emp integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
update did_work
set employee_id = emp
where ticket_id = tic;
end; $function$
```

The updateticket() function is used to update an existing ticket with a new location and problem.

```
CREATE OR REPLACE FUNCTION public.updateticket(id integer, loc integer, prob integer)
 RETURNS void
 LANGUAGE plpgsql
AS $function$
begin
update ticket
set location_id = loc, problem_id = prob
where ticket_id = id;
end; $function$
```

7.2.2 Database to Front-end Connection

The Main Window contains the following code, which should be self-documenting by the function names, for calling views and stored procedures:

```
    public void updateRecentInvoices(){
    String sql = "Select * FROM recentinvoiceview";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        ResultSet rs = stmt.executeQuery();

        String rowData[] = new String[4];
        DefaultTableModel model = (DefaultTableModel) jTable3.getModel();
        model.setRowCount(0);
        while (rs.next()) {
            rowData[0] = rs.getString(1);
            model.addRow(rowData);
        }
        jTable3.setModel(model);
        jTable3.setRowHeight(30);
        model.fireTableDataChanged();

    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

The Delete Ticket button:

```
private void jButton2ActionPerformed(java.awt.event.ActionEvent evt) {
    int row = -1;
    row = jTable1.getSelectedRow();
    if (row < 1)
        return;
    String sql = "select deleteticket(?)";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setInt(1,Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString()));
        stmt.executeQuery();
        updateTickets();

    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

The Delete Invoice button:

```
private void jButton12ActionPerformed(java.awt.event.ActionEvent evt) {
   int row = -1;
    row = jTable2.getSelectedRow();
    if (row < 0)
    {
        System.out.println("No invoice selected");
        return;
    }
    Globals.invoiceid = jTable2.getModel().getValueAt(row,1).toString();
    Globals.ticketid = getTicketFromInvoice(Globals.invoiceid);

    String sql = "select deleteinvoice(?)";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setString(1, Globals.invoiceid);
    stmt.executeQuery();
    }
    catch(Exception e) {
    System.out.println(e.getMessage());
    }

    sql = "select delete_assigned(?)";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setInt(1, Globals.ticketid);
    stmt.executeQuery();
    }
    catch(Exception e) {
    System.out.println(e.getMessage());
    }

    sql = "select deleteticket(?)";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setInt(1, Globals.ticketid);
    stmt.executeQuery();
    }
    catch(Exception e) {
    System.out.println(e.getMessage());
    }
    Globals.invoiceid = "-1";
    Globals.ticketid = -1;
    updateInvoices();
    updateTickets();
    updateRecentInvoices();

}
```

The Create Ticket button:

```
private void jButton5ActionPerformed(java.awt.event.ActionEvent evt) {
    int row = -1;
    row = jTable1.getSelectedRow();
    if (row < 0)
    {
        System.out.println("No ticket selected");
        return;
    }
    Globals.ticketid = Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString());
    String sql = "select * from getreceived(?)";

    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setInt(1,Globals.ticketid);
        ResultSet rs = stmt.executeQuery();
        if (rs.next())
            Globals.ticketreceived = rs.getString(1);

        }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    AddTicket a = new AddTicket();
    a.setParentObject(this);
    a.setVisible(true);
}
```

The Check In button:

```
private void jButton1ActionPerformed(java.awt.event.ActionEvent evt) {
    // CHECK IN
    int row = -1;
    row = jTable1.getSelectedRow();
    if (row < 0)
    {
        System.out.println("No ticket selected");
        return;
    }
    if (!getStatus(Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString())).equals("Assigned"))
    {
        System.out.println("Ticket must be assigned before checking in");
        return;
    }

    changeStatus(Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString()), "In Progress");
    String sql = "select check_in(?)";

    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setInt(1,Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString()));
        stmt.executeQuery();
        }

    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    updateTickets();
}
```

The Check Out button:

```
private void jButton4ActionPerformed(java.awt.event.ActionEvent evt) {
    // CHECK OUT
    int row = -1;
    row = jTable1.getSelectedRow();
    if (row < 0)
    {
        System.out.println("No ticket selected");
        return;
    }
    if (!getStatus(Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString())).equals("In Progress"))
    {
        System.out.println("Ticket must be in progress before checking out");
        return;
    }
    //Globals.ticketid = Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString());
    changeStatus(Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString()), "Completed");
            String sql = "select check_out(?)";

    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setInt(1,Integer.parseInt(jTable1.getModel().getValueAt(row,0).toString()));
        stmt.executeQuery();
        }

    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    updateTickets();
}
```

```
public String getStatus(int id)
{
    String sql = "select get_status(?)";
    String status = "";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setInt(1, id);
    ResultSet rs = stmt.executeQuery();
    if (rs.next())
        status = rs.getString(1);
    System.out.println(status);
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    return status;
}
public void changeStatus(int id, String newStatus)
{
    String sql = "select set_status(?,?)";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setInt(1, id);
    stmt.setString(2, newStatus);
    stmt.executeQuery();
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```
public void updateFields()
{
    //update technicians
    jComboBox4.addItem("");
    DefaultComboBoxModel model = (DefaultComboBoxModel) jComboBox4.getModel();
    String sql = "Select * from technicianview";
    Item item;
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        ResultSet rs = stmt.executeQuery();
        while (rs.next()){
            int id = rs.getInt(1);
            String name = rs.getString(2) + " " + rs.getString(3);
            item = new Item(id,name);
            techMap.put(item.getDescription(), item.getId());
        }
        for (String s : techMap.keySet())
                jComboBox4.addItem(s);
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
    //update problems
    model = (DefaultComboBoxModel) jComboBox3.getModel();
    sql = "Select * from problemview";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        ResultSet rs = stmt.executeQuery();
        while (rs.next()){
            int id = rs.getInt(1);
            String name = rs.getString(2);
            item = new Item(id,name);
            probMap.put(item.getDescription(),item.getId());
        }
        for (String s : probMap.keySet())
            jComboBox3.addItem(s);
    }
    catch(Exception e)
    {
        System.out.println(e);
    }


}
```

```
public void updateCustomers(){

    DefaultTableModel model = (DefaultTableModel) jTable1.getModel();
    String sql = "Select * FROM customerview";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        ResultSet rs = stmt.executeQuery();

        String rowData[] = new String[2];

        model.setRowCount(0);
        while (rs.next()) {
            rowData[0] = rs.getString(1);
            rowData[1] = rs.getString(3).substring(0,1).toUpperCase() +
                    rs.getString(3).substring(1) + ", " +
                    rs.getString(2).substring(0,1).toUpperCase() +
                    rs.getString(2).substring(1);
            //System.out.println(rowData[0]);

            if (!jTextField4.getText().isEmpty())
            {
                if (rowData[1].toLowerCase().contains(jTextField4.getText().toLowerCase()))
                    model.addRow(rowData);
            }
            else
                model.addRow(rowData);
        }
        jTable1.setModel(model);
        jTable1.setRowHeight(30);
        model.fireTableDataChanged();

    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```
public void updateLocations(int id){
    DefaultTableModel model = (DefaultTableModel) jTable2.getModel();
    String sql = "Select * FROM get_locations(?)";
    try {
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setInt(1,id);
        ResultSet rs = stmt.executeQuery();
        String rowData[] = new String[2];
        model.setRowCount(0);
        while (rs.next()) {
            rowData[0] = rs.getString(1);
            rowData[1] = rs.getString(2);
            rs.getString(3);
            rs.getString(4);
            rs.getString(5);
            if (!jTextField5.getText().isEmpty())
            {
                if (rowData[1].toLowerCase().contains(jTextField5.getText().toLowerCase()))
                    model.addRow(rowData);
            }
            else
                model.addRow(rowData);
        }
        jTable2.setModel(model);
        jTable2.setRowHeight(30);
        model.fireTableDataChanged();
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

When adding a ticket, the program checks if the user is editing an existing ticket set by the boolean “editMode”. If the user is editing, updateTicket() is called, otherwise addticket() is called.

```
if (jTextField1.getText().isEmpty())
{
    if (editMode)
        sql = "select * from updateticket(?,?,?)";
    else
        sql = "select * from addticket(?,?)";
}
else
{
    if (editMode)
        sql = "select * from updateticket(?,?,?,?)";
    else
        sql = "select * from addticket(?,?,?)";
}
```

If a technician is selected, they are assigned as well.

```
else
{
    sql = "select addassign(?,?)";
    changeStatus(ticket_id, "Assigned");
}
try (
PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
stmt.setInt(1, ticket_id);
stmt.setInt(2, techMap.get(jComboBox4.getSelectedItem().toString()));
stmt.executeQuery();
//update status of ticket
```

```
public void updateFieldsEditMode()
{
    String sql = "Select * FROM get_invoice_info_full(?)";
    double total = -1;
    int paymeth = -1;
    String desc = "";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setString(1,Globals.invoiceid);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            total = rs.getDouble(1);
            paymeth = rs.getInt(2);
            desc = rs.getString(3);
        }
        jTextField3.setText(Globals.invoiceid);
        jTextField7.setText(""+total);
        jComboBox1.setSelectedIndex(paymeth-1);
        jTextArea1.setText(desc);
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```
public void updateFields()
{
    String sql = "Select * FROM get_invoice_info(?)";
    String customerName = "";
    String address = "";
    String city = "";
    String zipcode = "";
    String employeeName = "";
    String problem = "";
    String startTime = "";
    String endTime = "";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setInt(1,Globals.ticketid);
        ResultSet rs = stmt.executeQuery();
        if (rs.next()) {
            jTextField5.setText(rs.getString(1) + " " + rs.getString(2));
            jTextArea2.setText(rs.getString(3) + "\n" + rs.getString(4)
                    + "\n" + rs.getString(5));

            jTextField8.setText(rs.getString(6) + " " + rs.getString(7));
            jTextField12.setText(rs.getString(8));
            jTextField9.setText(rs.getString(9));
            jTextField10.setText(rs.getString(10));
        }

    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

The Add Invoice button:

```
private void jButton10ActionPerformed(java.awt.event.ActionEvent evt) {
    //Globals.invoiceid = "-1";
    //validate
    if (jTextField3.getText().isEmpty() || jTextField7.getText().isEmpty()
            || !isDouble(jTextField7.getText()))
    {
        System.out.println("Invalid entry.");
        return;
    }
    //CREATE INVOICE
    String sql = "select * from addinvoice(?,?,?::numeric(10,2),?,?)";
    //System.out.println(jComboBox1.getSelectedIndex());
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        stmt.setString(1, jTextField3.getText());
        stmt.setInt(2, Globals.ticketid);
        stmt.setDouble(3, Double.parseDouble(jTextField7.getText()));
        stmt.setInt(4, jComboBox1.getSelectedIndex() + 1);
        stmt.setString(5, jTextArea1.getText());
        ResultSet rs = stmt.executeQuery();
        //if (rs.next())
            //System.out.println(rs.getInt(1));
    }
    catch (SQLException se){
        System.out.println(se.getMessage());
        return;
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    changeStatus(Globals.ticketid, "Closed");
    parentObject.updateTickets();
    parentObject.updateInvoices();
    parentObject.updateRecentInvoices();

    this.setVisible(false);

}
```

The Report button:

```
String sql = "select * from get_invoice_report(?)";
try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setString(1, Globals.invoiceid);
    ResultSet rs = stmt.executeQuery();
    if (rs.next())
    {
        invoiceNumber = rs.getString(1);
        customerName = rs.getString(2) + " " + rs.getString(3);
        address = rs.getString(4) + "\n" + rs.getString(5) + ", " +
                rs.getString(6) + " " + rs.getString(7);
        description = rs.getString(8);
        jobAmount = Double.toString(rs.getDouble(9));
        paymethod = rs.getString(10);
        technicianName = rs.getString(11) + " " + rs.getString(12);
        SimpleDateFormat sdf = new SimpleDateFormat("MM/dd/yyyy KK:mm a");
        SimpleDateFormat sdf2 = new SimpleDateFormat("MM/dd/yyyy");
        startTime = sdf.format(rs.getTimestamp(13));
        endTime = sdf.format(rs.getTimestamp(14));
        date = sdf2.format(rs.getTimestamp(14));
    }


}
catch (SQLException se){
    System.out.println(se.getMessage());
    return;
}
```

```
public int getTicketFromInvoice(String invNum)
{
    String sql = "select get_ticket_from_invoice(?)";
    int ticketid = -1;
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setString(1, invNum);
    ResultSet rs = stmt.executeQuery();
    if (rs.next())
        ticketid = rs.getInt(1);
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    return ticketid;
}
```

```
public String getStatus(int id)
{
    String sql = "select get_status(?)";
    String status = "";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setInt(1, id);
    ResultSet rs = stmt.executeQuery();
    if (rs.next())
        status = rs.getString(1);
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    return status;
}
```

```
public int checkDateQuery(String id, java.sql.Date begin, java.sql.Date end)
{
    String sql = "select invoice_in_date(?,?,?)";
    int status = 0;
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setString(1, id);
    stmt.setDate(2, begin);
    stmt.setDate(3,end);
    ResultSet rs = stmt.executeQuery();
    if (rs.next())
        status = rs.getInt(1);
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    return status;
}

public void changeStatus(int id, String newStatus)
{
    String sql = "select set_status(?,?)";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setInt(1, id);
    stmt.setString(2, newStatus);
    stmt.executeQuery();
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
}
```

```
public void updateTickets(){
    String sql = "Select * FROM openticketview order by status asc, ticket_id asc";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        ResultSet rs = stmt.executeQuery();

        String rowData[] = new String[4];
        DefaultTableModel model = (DefaultTableModel) jTable1.getModel();
        model.setRowCount(0);
        while (rs.next()) {
            rowData[0] = rs.getString(1);
            rowData[1] = rs.getString(2).substring(0,1).toUpperCase() +
                    rs.getString(2).substring(1) + ", " + rs.getString(3).substring(0,1).toUpperCase() +
                    rs.getString(3).substring(1) + " " +
                    rs.getString(4).substring(0,1).toUpperCase() +
                    rs.getString(4).substring(1) + ", " + rs.getString(5);
            rs.getString(6);
            rs.getString(7);
            rowData[2] = rs.getString(8);
            rowData[3] = rs.getString(9) + " " + rs.getString(10);
            //delete null
            for (int i = 0; i < 4; i++) {
                if (rowData[i].equals("null") || rowData[i].equals("null null"))
                    rowData[i] = "";
            }

            model.addRow(rowData);

        }
```

```
public String getTech(String invNum)
{
    String sql = "select get_tech(?)";
    String name = "";
    try (
    PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
    stmt.setString(1, invNum);
    ResultSet rs = stmt.executeQuery();
    if (rs.next())
        name = rs.getString(1);
    }
    catch(Exception e) {
        System.out.println(e.getMessage());
    }
    return name;
}
```

```
public void updateInvoices(){
    String sql = "Select * FROM invoiceview";
    try (
        PreparedStatement stmt = dbc.conn.prepareStatement(sql)){
        ResultSet rs = stmt.executeQuery();

        String rowData[] = new String[4];
        DefaultTableModel model = (DefaultTableModel) jTable2.getModel();
        model.setRowCount(0);
        while (rs.next()) {
            rowData[0] = rs.getString(1) + " " + rs.getString(2);
            rowData[1] = rs.getString(3);
            rowData[2] = rs.getString(4);
            rowData[3] = rs.getString(5);

            //delete null
            for (int i = 0; i < 4; i++) {
                if (rowData[i].equals("null") || rowData[i].equals("null null"))
                    rowData[i] = "";
            }
            /////////NEW WIP
            SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");


            boolean withinDates = false;

            if(datePicker1.toString().isEmpty() || datePicker2.toString().isEmpty())
                withinDates = true;
            else
            {
                java.sql.Date date1 = java.sql.Date.valueOf(datePicker1.getDate());
                java.sql.Date date2 = java.sql.Date.valueOf(datePicker2.getDate());
                if (checkDateQuery(rowData[1], date1, date2) > 0)
                    withinDates = true;
            }
```

Survey

1. An ability to analyze a problem, and identify and define the computing requirements and specifications appropriate to its solution. --- 8
2. 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. --- 8
3. 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. --- 8
4. 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