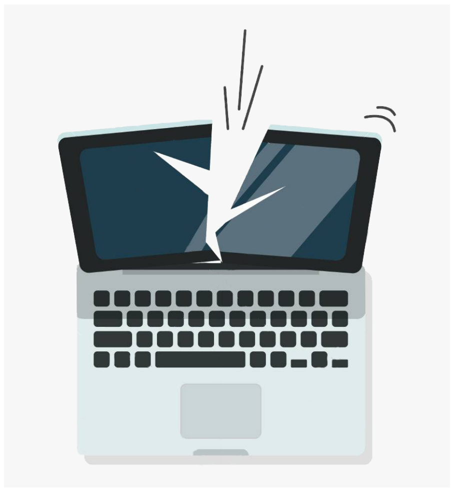
# Repair Shop



Group #06

Francisco D. Ulloa
Alfredo A. Zavala

# Phase 1: Fact-Finding, Information Gathering,and Conceptual Database Design

Phase one of our repair shop database will focus on how we obtained information regarding the types of entities, data and relationships that should be present within the schema. Once explained how we gained an idea of how to organize the repair shop database, we will then jump into introducing and defining each entity along with their respective attributes.  When defining the attributes, the meaning of the attribute will be explained.

For the final section of 1.1 it will discuss the groups who will be using this database regularly. Along with how the group plans to use this database, such as their tasks and jobs for when they interact with it.

## 1.1 Fact-Finding Techniques and Information Gathering

In order to make a database for a repair shop we needed to gather our information through multiple ways. Interviews with multiple employees who work or have worked at repairs shops really helped us choose what to add into our database. Also being a customer at a repair shop showed us what the customer sees when going in for a repair. In the later sections we discuss how we used and found our facts for the database we created.

### 1.1.1 Introduction to Enterprise/Organization

*RepairShop* is a shop that takes in customer's computers and repairs them. It offers all types of repairs, whether there is something wrong with the GPU or motherboard. Employees at *RepairShop* will be able to diagnose a problem with a computer if the customer cannot. Employees are required to be knowledgeable with computers, not only will they need to know what each part does but know what parts to order from the supplier.

### 1.1.2 Description Fact-Finding Techniques

We got information from a few repair shops around California. Some were still in the old school era where they wrote everything down on a piece of paper. A customer would have to either write down or explain their problem with the computer. We observed how employees are always looking for the paper with the problem with the computers, if they couldn't find it then they had to spend time figuring out what was wrong with it. That time employees spent diagnosis the computer was valuable time the company lost. We've talked to the employees and they all mentioned how they wanted a better way of keeping track of customer's computer problems. We also looked at the way they organized their inventory of replacement parts. They had them ordered by manufacturers then by product names.

### 1.1.3 Scope of the Database

Our goal for this database is to create a way for employees to seamlessly view repair requests and order parts for customer's computers who need repair. Customers will be able to fill out a form and describe the problem with their computer that needs to be fixed.

### 1.1.4 Itemized Descriptions of Entity Sets and Relationship Sets

**Entity Name:** Employee

Attribute: EmployeeID

Type: Int

Meaning: Unique number to identify each employee.

Instance:123456780

Attribute: EmployeeName

Type: String

Meaning: Name of employee.

Instance: John

Attribute: PhoneNum

Type: String

Meaning: Phone number of employee.

Instance: (661)555-1234

Attribute: EmpAddress

Type: String

Meaning: Address of employee.

Instance: 123 Repair Ave.

Attribute: E-Mail

Type: String

Meaning: E-mail address to contact employee

Instance: random_email@gmail.com

Attribute: SSN

Type: String

Meaning: Social Security Number of employee.

Instance: 123-45-6789

**Entity Name:** Customer

Attribute: CustomerID

Type: Int

Meaning: Unique number identifying a customer.

Instance: 1234567

Attribute: CustomerName

Type: String

Meaning: Name of the customer.

Instance: Jim Smith

Attribute: PhoneNumber

Type: String

Meaning: Phone number of customer.

Instance: (661)555-1212

Attribute: CCnum

Type: String

Meaning: Customer's credit card number.

Instance: 1231456578981234

Attribute: CCtype

Type: String

Meaning: Type of credit card a customer has.

Instance: Visa

**Entity Name:** Device

Attribute: DeviceID

Type: Int

Meaning: Unique identification number for each device.

Instance: 1231231230

**Entity Name:** Manufacturer

Attribute: ManufacturerID

Type: Int

Meaning: Unique ID for each manufacture

Instance: 12345678

Attribute: ManufacturerName

Type: String

Meaning: Name of manufacturer

Instance: Repair in Bulk

Attribute: PhoneNum

Type: String

Meaning: Phone number of manufacturer

Instance: (661)555-9876

Attribute: Address

Type: String

Meaning: Address of manufacturer

Instance: 213 Repair Ave.

**Entity Name:** Model

Attribute: ModelName

Type: String

Meaning: Name of device model

Instance:

Attribute: YearReleased

Type: Date

Meaning: Date the product was released

Instance: 01/02/2020

**Entity Name:** Orders

Attribute: OrderID

Type: Int

Meaning: Unique identification number for each order

Instance: 1236547895

Attribute: OrderDate

Type: Date

Meaning: Date the order was made

Instance: 01/03/2020

Attribute: OrderStatus

Type: String

Meaning: Status of the order

Instance: In transit

**Entity Name:** Payment

Attribute: PaymentID

Type: Int

Meaning: Unique identification number for each payment

Instance: 45678912

Attribute: PaymentType

Type: Int

Meaning: Type of payment is used

Instance: credit card

Attribute: amount

Type: Double

Meaning: Amount to pay for repair

Instance: 123.45

Attribute: date-time

Type: Date

Meaning: Date and time of payment completed

Instance: 01/23/2020 15:16:12

**Entity Name:** RepairRequest

Attribute: RepairOrderID

Type: Int

Meaning: Unique identification number for each repair order

Instance: 12321300

Attribute: Date

Type: Date

Meaning: Date the repair request being asked for

Instance: 01/10/2020

Attribute: DmgDesc

Type: String

Meaning: Description of damage on device that needs repair

Instance: Makes loud noise when using

Attribute: RepairLog

Type: String

Meaning: Log file explaining what was done to the device

Instance: Fixed loose cable

Attribute: TypeOfRepair

Type: String

Meaning: Type of repair that was done on the device

Instance: Cable management

**Entity Name:** ReplacementParts

Attribute: PartID

Type: Int

Meaning: Unique identification number for the parts that are used on the device

Instance: 11223344

Attribute: PartName

Type: String

Meaning: Name of the replacement parts used on the device

Instance:

Attribute: SellCost

Type: Double

Meaning: Price of part used in repair

Instance: $50.00

Attribute: quantity

Type: Integer

Meaning: Amount of part we have in stock

Instance: 50

**Entity Name:** Supplier

Attribute: SupplierID

Type: Int

Meaning: Unique identification number for each supplier

Instance: 12231223

Attribute: SupplierName

Type: String

Meaning: Name of supplier

Instance:

Attribute: SupplierAddress

Type: String

Meaning: Address for each supplier

Instance: 111 Repair Ave.

Attribute: PhoneNum

Type: String

Meaning: Phone number of supplier

Instance: (661)555-2222

**Relationship Name:** Works on

Meaning: Employee “works on” a repair request

Relates: Relates Employee to Repair request

Instance: Employee “works on” a repair request

Constraint: One employee can work on many repair request

**Relationship Name:** Uses

Meaning: Employee “uses” replacement parts on a device

Relates: Relates employee to replacement parts

Instance: Employee “uses” GPU on a customers computer

Constraint: Only one employee can use that replacement part at a time

**Relationship Name:** Creates

Meaning: Employee “creates” an order that is given to the supplier

Relates: Relates employee to order

Instance: Employee “creates” an order for 6 sticks of RAM

Constraint: Only employees are allowed to create an order

**Relationship Name:** Contains

Meaning: Each order “contains” a specific amount of replacement parts

Relates: Relates Order with replacement parts

Instance: Order #12345678 “contains” 12 CPUs and 4 sticks of RAM

Constraint:

**Relationship Name:** Supplies

Meaning: The Supplier “supplies” the replacement parts for the repairs

Relates: Relates Supplier with replacement parts

Instance: EVGA “supplies” the GPU to the repair shop

Constraint:

**Relationship Name:** Is_Involved

Meaning: Each device “is involved” in one or many repair request

Relates: Relates Device to Repair Request

Instance: Jim’s computer “is involved” in 1 repair request

Constraint:

**Relationship Name:** Paid with

Meaning: Customer “paid with” some type of payment

Relates: Relates Repair request with Payment

Instance: Jim “payed with” his Visa to fix the repair

Constraint: Each repair must be paid with cash or credit card

**Relationship Name:** Pays

Meaning: Customer “pays” for the “payment” of the repair done on their device

Relates: Relates Customer with Payment

Instance: Jim “pays” for his repair on his computer.

Constraint: Can only pay when the repair is finished.

**Relationship Name:** Brings

Meaning: Customer “brings” in their device for repair

Relates: Relates Customer with Device

Instance: Dan “brings” in his laptop for repair

Constraint:

**Relationship Name:** Identifies Device

Meaning: The model “identifies devices” that customers bring in

Relates: Relates Models with Device

Instance: EVGA 2080ti GPU is in Rick’s computer

Constraint:

**Relationship Name:** Narrows Search

Meaning: Manufacturer will “narrow the search” for each Model

Relates: Relates Model with Manufacturer

Instance: EVGA 2080ti

Constraint:

### 1.1.5 User Groups, Data views and Operations

There are two types of users: employee and customer.

Employees will be able to request parts from a supplier and view what device needs repair. They will be able to view customers accounts and view their device history.

Customers will be able to create an account, add their information, and fill out a repair request. They will need to identify their computer so the employee knows what parts/problem are in the computer. They will also be able to view the repair history that the employees have done to the computer. Customers will have a choice to pay with a credit card or cash after the repair is finished.

## 1.2 Conceptual Database Design

Section 1.2 will contain detailed information regarding the entity sets and relationships used in our repair shop database schema In section 1.2.1 there will be sections describing each entity used within our schema. These descriptions will include the entity's name, type, primary and a description that defines the purpose of the entity and how it relates to other entities with their relationships. There is then a table describing the attributes used in each entity and how they're defined. For section 1.2.2 There will be something similar done for relationships where there are descriptions of their purpose and how they connect the two entities to each other.

The next two sections of section 1.2 will be left to describe the vocabulary and how our diagram is arranged with the entities and relationships. Section 1.2.3 will contain definitions describing Specialization/Generalization Relationships and Aggregation. Lastly, Section 1.2.4 will contain a diagram of our schema.

### 1.2.1 Entity Type Description

**Entity Name:** Customer
**Description:** The purpose of the customer entity is to store information about the people who come into the repair store and have us repair their broken devices. The customer entity will have us store common information such as the customer's name and phone number. Some other information such as CCnum and ccType will be used to store credit card information should the customer choose to pay with a credit card.

The information stored for customers will be accessed and used with other entities, in this case, the request order entity can use the customer entity to create a more informed request order on the device. The customer entity will be having information frequently queried from it and information will be frequently entered into it.
**Primary Key:** customerID

**Entity Type:** Strong

| Name | customerID | customerName | phoneNumber | CCnum | ccType |
|---|---|---|---|---|---|
| Description | A unique number associated with a customer to help keep each customer unique. | This is the customer's name for when they apply for a repair request. | Phone number to help keep customers in the loop on their repair and when it is done. | Number on the customers credit card. | Type of credit card the customer is using, mastercard, visa. |
| Domain/Type | Int | String | String | String | String |
| Value-Range | All valid 10 digit numbers | All names | All valid phone numbers. | All valid credit card numbers from multiple banks. | Either Mastercard or Visa |
| Default Value | 0000000000 | None | None | None | None |
| Null Allowed | No | No | No | Yes | Yes |
| Unique | Yes | No | No | Yes | No |
| Single or Multivalued | Single | Multi | Single | Single | Single |
| Simple or Composite | Simple | Composite | Simple | Simple | Simple |

**Entity Name:** Device
**Description:** The purpose of the device entity is to gather information about the type of device the customer has brought in to be repaired. This entity will give each device the customer brings in a unique identity and allows the repair shop to keep track of when the device was dropped off and returned to its owner. This information has relationships directly with three other entities, those entities being the customer, so we can keep track of what device belongs to which customer, the repair order, to define and describe the device being repaired in the repair order and lastly, model entity, to help identity what model the device is that is coming in for repair.

This type of entity will have very frequent insertions into the database on a daily basis. Not only will insertions be frequent but the information from other entities will be passed along to it when it comes time. Updates and deletion of information in this entity should not happen very often but will occasionally happen. For instance, when a customer comes time to pick up their device, this entity will be updated with the devices final pick up date.

**Primary Key:** deviceID
**Entity Type:** Strong

| | |
|---|---|
| Name | deviceID |
| Description | Used to uniquely identify each device |
| Domain/Type | Int |
| Value-Range | Any 10 digit number |
| Default Value | 0000000000 |
| Null Allowed | No |
| Unique | Yes |
| Single or Multivalued | Single |
| Simple or Composite | Simple |

**Entity Name:** Employee
**Description:** This entity's responsibility is to keep track of the currently employed employees of the repair shop. This entity stores information such as the employees, name, phone number, employee address, and social security number. This information would be used to get in contact with the employee if they ever need to be informed or updated about something involving their job at the repair shop.

When it comes to relating to other entities, this entity has a relationship with repair request and order. The relationship with repair requests is to have an employee work on a repair for a customer's device. The latter relationship, order, is or when an employee creates an order shipment for a supplier for replacement parts.

This type of entity will rarely have its contents be updated or deleted but will have not-so frequent insertions. Those insertions would be for new employees coming in.
**Primary Key:** employeeID
**Entity Type:** Strong

| Name | employeeID | employeeName | phoneNum | empAddress | e-mail | SSN |
|---|---|---|---|---|---|---|
| Description | Uniquely identifies all employees | This is the customer's name when they work. | Phone number of the employee. | The current living address of the employee | A digital way to contact the employee | The government social security number used for identification and tax purposes |

|  |  |  |  |  |  |  |
|---|---|---|---|---|---|---|
| Domain/Type | Int | String | String | String | String | Int |
| Value-Range | All 8 digit numbers | All names | All valid phone number formats | Every valid address | All valid e-mail addresses | All 9 digits in the SSN |
| Default Value | 00000000 | New Employee | None | None | None | None |
| Null Allowed | No | No | No | No | No | No |
| Unique | Yes | No | No | No | Yes | Yes |
| Single or Multivalued | Single | Multi | Single | Multi | Single | Single |
| Simple or Composite | Simple | Composite | Simple | Composite | Simple | Simple |

**Entity Name:** Manufacturer
**Description:** The manufacturer entity is meant to provide manufacturer information to help narrow down the search for what kind of model of device a customer brings in to the repair shop. This entity holds information needed to narrow down the models of devices with attributes such as manufacturerName and then there are phoneNum and address for when there is a need to get in contact with manufacturers.

Other relationships this entity has is the relationship it has with the Model entity, as stated before, this is to help narrow down what model the device could be that the customer brings in. The frequency of insertion, deletion and updating will be very infrequent, the idea is to have to set this information once into the database and use it to narrow down the fields of what model a device could be. The only time information would be inserted into the database would be to update for the current year's new devices that are out whose manufacturer we haven't inserted yet.
**Primary Key:** manufacturerID
**Entity Type:** Strong

| Name | manufacturerID | manufacturerName | phoneNum | address |
|---|---|---|---|---|
| Description | Unique ID to identify manufacturers | The title of the manufacturer and how they are referred to | Phone number to get in contact with manufacturer | Address of the manufacturer |
| Domain/Type | Int | String | String | String |

| | | | | |
|---|---|---|---|---|
| Value-Range | 8 digit number | All names | Any valid phone number format | Every valid address |
| Default Value | 00000000 | NewManufacturer | None | None |
| Null Allowed | No | No | No | Yes |
| Unique | Yes | No | No | No |
| Single or Multivalued | Single | Multi | Single | Multi |
| Simple or Composite | Simple | Composite | Simple | Composite |

**Entity Name:** Model
**Description:** The purpose of the Model entities role is quite similar to that of the Manufacturers role, but instead it narrows down the search completely to the model of device that was brought in, along with other common information about the device that the employee can use to repair a device. For instance, if they are doing a battery swap, then they just need to see how large of a battery it took and grab the corresponding battery.

Information inside of this entity are modelName, yearReleased, screenSize, batterySize and RAM. All this information can be used for repair services and to identify the device the customer brings in.

The information in this entity should not have to be updated or deleted very often if ever, but instead the idea is to populate this entity once with the current years device models and then insert new models as they arise.
**Primary Key:** modelName
**Entity Type:** Strong

| Name | modelName | yearReleased |
|---|---|---|
| Description | Name of the device model | The year the model was released to the public |
| Domain/Type | String | Date |
| Value-Range | All names | All dates starting at 1950 to now |
| Default Value | EmptyName | 01/01/1950 |
| Null Allowed | No | No |
| Unique | Yes | No |
| Single or Multivalued | Multi | Multi |

| | | |
|---|---|---|
| Simple or Composite | Composite | Composite |

**Entity Name:** Order
**Description:** The order entity is to help keep track of what orders have been placed to suppliers that the employees have made for replacement parts. This entity only has two attributes for itself, the orderID and orderDate. The orderID would be the unique identifier for each order they request from the supplier and the orderDate would be to see when the order was placed.

The relationships this entity has are with Employees and Replacement Parts. Employees entity and this entity share a relationship to see which employee makes the order. While the ReplacementParts entity is used to categorize what parts were in the order.

When it comes to inserting information into this entity, it would happen whenever an order is being placed. Deletion or updates in this entity should occur when the order has to be canceled or when its been successfully delivered.
**Primary Key:** orderID
**Entity Type:** Strong

| Name | orderID | orderDate | orderStatus |
|---|---|---|---|
| Description | Holds an identification number that is unique to each order | This contains the date the order was made | Tells the status of the order, whether it was delivered, waiting to be delivered or canceled. |
| Domain/Type | Int | Date | String |
| Value-Range | All ten digit numbers | Date begins from the stores first day in business to the current day | Delivered, Not delivered, or Canceled. |
| Default Value | 0000000000 | 01/01/2020 | Not Delivered |
| Null Allowed | No | No | No |
| Unique | Yes | No | No |
| Single or Multivalued | Single | Multi | Single |
| Simple or Composite | Simple | Composite | Simple |

**Entity Name:** Payment
**Description:** The Payment entity is used to give the customer a way to pay for the repairs done on their device. They would be able to decide to pay in other cash or card. The attributes for this entity are the

paymentID to identify specific payments and the paymentType to categorize whether the customer paid in cash or with their card.

Payment has a relationship with Repair Request. This as stated before would be used for deciding what method of payment the customer used for the Repair Request. This entity should have data inserted into it every time a RepairRequest is being paid for and should not really have a need to be deleted or updated.

**Primary Key:** paymentID
**Entity Type:** Strong

| Name | paymentID | paymentType | amount | date-time |
|---|---|---|---|---|
| Description | Unique identification to identify payments | Determines what type of payment will be used, cash or card | Specifies the amount the customer pays for the repair cost | Denotes the current date and time when a customer pays |
| Domain/Type | Int | Int | Double | Date |
| Value-Range | All 8 digit number | 0-1 | All positive numbers | Current date and time |
| Default Value | 00000000 | All cards | 0.00 | Current date/time |
| Null Allowed | No | Yes | No | No |
| Unique | Yes | No | No | No |
| Single or Multivalued | Single | Single | Simple | Multi |
| Simple or Composite | Simple | Simple | Simple | Composite |

**Entity Name:** Repair Request
**Description:** The purpose of the Repair Request entity is to have an encompassing total of what type of repair the employee needed to have done on the device that a customer brought into the repair shop. This entity in particular holds a unique identifier to identify what repair was done on a device, repairOrderID, while having the date it was done, a dmgDesc attribute to describe what was wrong with the device and a repairLog to describe what was done to fix said device. Along with the typeOfRepair attribute to define the repairs being done and the hoursWorked, to bill the customer for how long it took to repair the device.

This entity has relationships with Employee, Payment , Customer, Device and Replacement Parts. The Employee relationship lets us see who has worked on the Repair Request, while the Customer relationship pays for the repairs being done on the Device, where Device can have many Repair Requests and the Payment lets us see what method the Repair Request was paid in. Lastly the relationship with Replacement Parts lets us see what parts were used and billed towards the customer.

Insertion of data in this entity will be very frequent for whenever a Customer wants to request a repair for their device. Updating this entity will also be a frequent occurrence as Employees work on the device and have to update previous information left empty at the time of creation. Deletion would only occur if a device could not be fixed and was done free of charge.

**Primary Key:** repairOrderID
**Entity Type:** Strong

| Name | repairOrderID | date | dmgDesc | repairLog |
|---|---|---|---|---|
| Description | Unique identification number for identifying each repair request | The date of the repair request being asked for | Description of the damage that the device has come in for repair | Log file describing what was done to repair the device |
| Domain/Type | Int | Date | String | String |
| Value-Range | All 8 digit number | Dates from the beginning of the store opening | All characters, numbers and symbols | All characters, numbers and symbols |
| Default Value | 00000000 | 01/01/2020 | Empty | Empty |
| Null Allowed | No | No | Yes | Yes |
| Unique | Yes | No | No | No |
| Single or Multivalued | Single | Multi | Single | Single |
| Simple or Composite | Simple | Composite | Simple | Simple |

| Name | typeOfRepair |
|---|---|
| Description | Describes the type of repair that will be done on the device |
| Domain/Type | String |
| Value-Range | Any combination of words to describe the type of repair |

| | |
|---|---|
| Default Value | None |
| Null Allowed | Yes |
| Unique | No |
| Single or Multivalued | Single |
| Simple or Composite | Simple |

**Entity Name:** Replacement Parts
**Description:** The purpose of the Replacement Parts entity is to categorize all of the replacement parts that will be used in the repair process of the customers device and to define the price each part costs for the customer. The entity has relationships with Order and Supplier. The relationship with Order is to let us see what parts the order has brought with it. While the relationship with Supplier will show us what parts the Supplier shipped out with the order the Employee ordered. We get the MSRP price from the supplier that they would sell to the public and the boughtCost we paid for them in the order.

The attributes in this entity are the partID where we can use it to distinguish each part from another, the partName and the sellCost for what the price will be sold to the customer. The frequency of inserting information into this entity would be whenever we get an order and need to update it with new parts, and updating the entity would occur when parts are either re-supplied or used.
**Primary Key:** partID
**Entity Type:** Strong

| Name | partID | partName | sellCost | quantity |
|---|---|---|---|---|
| Description | Unique identifier for parts used in repair | Name of the parts used in the repair of the device | Number of the cost of the part used in the repair when charging customer | The amount of a single part we have in stock |
| Domain/Type | Int | String | Double | Integer |
| Value-Range | All 8 digit numbers | Any combination of characters and numbers | All positive numbers | All positive numbers |
| Default Value | 00000000 | None | 01.23 | 21 |
| Null Allowed | No | No | No | No |
| Unique | Yes | No | No | No |
| Single or Multivalued | Single | Single | Single | Single |

| | | | | |
|---|---|---|---|---|
| Simple or Composite | Simple | Simple | Simple | Simple |

**Entity Name:** Supplier
**Description:** The Supplier entities purpose is to let Employees create orders for replacement parts from the list of suppliers. The attributes associated with this entity involve the supplierID that lets us identify each supplier, the supplierAddress and phoneNum lets us contact the supplier incase there was anything wrong with the order or if the store needs an update on the order. While the address lets us send back orders that were incorrect.

The relationship Supplier has is with Replacement Parts. This relationship is to give what replacement parts were shipped out according to the order and the MSRP of those parts when selling them.

Inserting data into this entity would be done when working with new suppliers and any updating of data would occur if a supplier has changed locations or phone numbers. Deletion of any records would rarely occur unless perhaps the supplier is no longer in business.
**Primary Key:** SupplierID
**Entity Type:** Strong

| Name | supplierID | supplierName | supplierAddress | phoneNum |
|---|---|---|---|---|
| Description | Unique identifier for the supplier | Name for each supplier | Address for each supplier to order from them | Phone number to get in contact with the supplier |
| Domain/Type | Int | String | String | String |
| Value-Range | All 8 digit numbers | Any combination of characters, numbers and symbols | Any valid address format | Any valid phone number format |
| Default Value | 00000000 | None | None | None |
| Null Allowed | No | No | No | No |
| Unique | Yes | No | No | No |
| Single or Multivalued | Single | Single | Multi | Single |
| Simple or Composite | Simple | Simple | Composite | Simple |

### 1.2.2 Relationship Type Description

**Relationship:** Brings
**Description:** The customer brings the device they want to get repaired to the repair shop. A single customer can bring in many different devices.
**Entities Involved:** Customer, Device
**Attributes:** None
**Cardinality:** 1 … M
**Participation:** Total:Mandatory

**Relationship:** Contains
**Description:** (A) Contains the boughCost of the replacement parts in the order. This boughtCost is how much the repair shop paid for the order.
**Entities Involved:** (A) Order, Replacement Parts
**Attributes:** (A) boughtCost, quantity
**Cardinality:** (A)M … M
**Participation:** (A) Total:Mandatory

**Relationship:** Creates
**Description:** An Employee creates an Order full of replacement parts to be shipped out to the repair shop so that they can use what they ordered to repair devices.
**Entities Involved:** Employee, Order
**Attributes:** None
**Cardinality:** 1 … M
**Participation:** Total / Mandatory

**Relationship:** Identifies_Device
**Description:** Models help identify the Device a customer has brought in with them to get repaired at the repair shop. There is a model for each type of device.
**Entities Involved:** Models, Device
**Attributes:** None
**Cardinality:** 1 … M
**Participation:** Total/Mandatory

**Relationship:** Is_Involved
**Description:** Demonstrates how the device is involved in a repair request
**Entities Involved:** Device, RepairRequest
**Attributes:** None
**Cardinality:** 1 … M
**Participation:** Total/Mandatory

**Relationship:** Narrows_Search
**Description:** When an employee selects a specific Manufacturer it narrows down the search for only those specific Models that are available from that manufacturer. A single manufacturer can have many different models.
**Entities Involved:** Manufacturer, Models
**Attributes:** None
**Cardinality:** 1 … M
**Participation:** Total/Mandatory

**Relationship:** Pays
**Description:** A customer pays for the Repair Request for the device they brought in to get fixed through a payment
**Entities Involved:** Customer, Payment
**Attributes:** None
**Cardinality:** 1 … M
**Participation:** Total/Mandatory

**Relationship:** Paid_With
**Description:** When a customer pays for a repair request, we check whether they are going to pay in cash or card. A single card or cash payment is used to pay for the single repair request, so the cardinality mapping is one to one.
**Entities Involved:** Payment, Repair_Request
**Attributes:** None
**Cardinality:** 1 … 1
**Participation:** Total/Mandatory

**Relationship:** Supplies
**Description:** When an replacement part order is made by the employee and sent to the supplier. The Supplier then supplies the Replacement Parts for the order that the employee asked for. This order is then sent to the repair shop.
**Entities Involved:** Supplier, Replacement Parts
**Attributes:** MSRP
**Cardinality:** 1 … M
**Participation:** Total

**Relationship:** Uses
**Description:** When an Employee works on a repair request for a device that a customer brought in, they will use the Replacement Parts they have in stock in order to repair said device. They will then charge the customer accordingly to the number of parts used to fix their device.
**Entities Involved:** Employee, Replacement Parts
**Attributes:** numPartsUsed, quality, sellCost
**Cardinality:** 1 … M
**Participation:** Partial/Optional

**Relationship:** Works_On
**Description:** The Employee works on the Repair Request for the appropriate device pertaining to that repair request. An employee can work on many different repair requests,
**Entities Involved:** Employee, Repair Request
**Attributes:** hoursWorked
**Cardinality:** 1 … M
**Participation:** Total/Mandatory

### 1.2.3 Related Entity Type

The generalization relationship is a relationship where you take a set of entities and find their common properties to combine them. If they have some attributes in common then you can combine the entities to create a more generalized entity that represents both of them. If there are specialized attributes then they can be added to the specialized attribute that it originated from. This type of process is known as a bottom-up approach.

Meanwhile, the specialization relationship is the opposite approach of a generalization relationship. This relationship is a top-down approach where you take a higher generalized entity and then create two or more specialized entities from the generalized one.

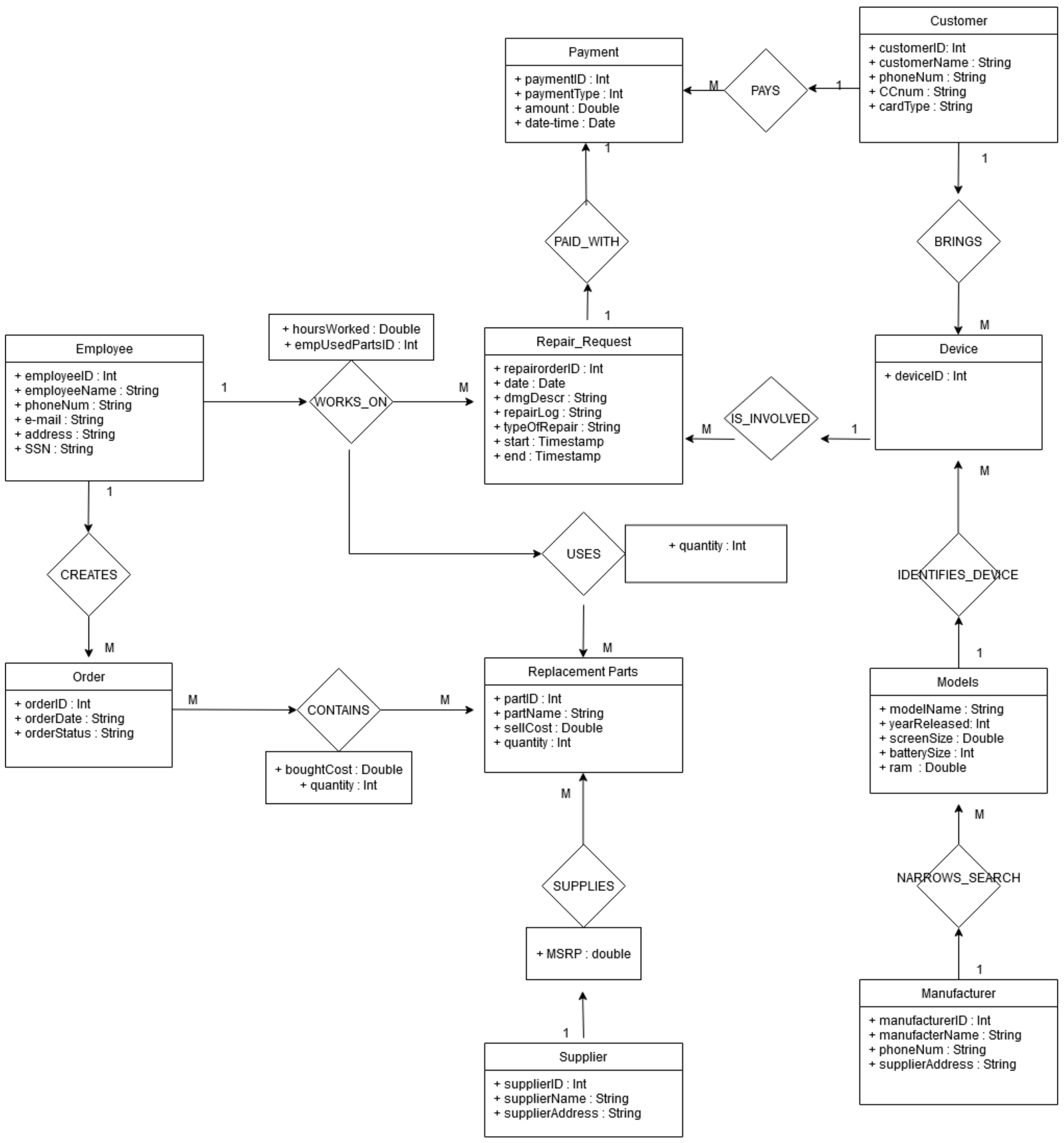
Participation constraint is when an entity that is created depends on another entity. For example Device is dependent on the repair request because without it the employee will not know what to do with the device the customer brought in. Another example would be employees depending on the replacement parts in order to fix a customer's computer. Without the parts the employee will not be able to fix anything.

Disjoint constraint is when there is a superclass and subclasses. The superclass can be in either of the subclasses, the downside is that it cannot be in both of the subclasses at the same time.

Aggregation relationships involve creating a 2nd relationship that connects to the original relationship of two already connected entities. This 2nd relationship then connects the original relationship and an entity together. The two entities that were already sharing the original relationship becomes a higher entity that then uses the newly created relationship to connect to the other higher entity.

Composite aggregation is a strong form of aggregation that specifies a whole/part relationship. You can think of this as a relationship between a file and folder on your computer. If you delete the “composite whole”, the file, then you end up deleting all of its “composites”, folders that are part of the folder, “composite whole”.

## 1.2.4 E-R Diagram



# Phase II:  From ER (Conceptual) Model to Relational (Logical) Model

Phase two focuses on the conversion of our computer repair shop from the original ER Model we created, into the Relational Model. These two models each have their own pros and cons when it comes to which you should choose to start off with and which you should choose to begin your database schema.

In section 2.1 it goes over these the technique needed to convert from one model to the other. In the second half we look at the 10 query questions and their solutions in relational algebra, tuple calculus and domain calculus.

## 2. Conceptual Database and Logical Database

In this section we analyze the different types of models and describe their goals while going over the differences and similarities between the two of them.

### 2.1 E-R Model and Relational Model

The ER model was developed by Peter Chen in 1976. The ER model is used for conceptual design of database applications. The model is used for an accurate model of a database that can be used in the real world. It can be used with entities that can be found in the real world. A major feature of this model is that it allows users to see where the flow of each entity goes. You are able to picture and view the flow of the diagram. The purpose of the model is it allows you to have key features when designing a database. It also lets you visualize how each entity relates to each other. It stores relational information so you can quickly glance at the diagram and see what you need. If there is an error with the logic then it will be easy to figure out where the problem is.

The relational model was proposed by Edgar Codd in 1969. The relational model is a table that represents data and relations in them. It is a method of structuring data using relations which can be placed in a table like fashion. The tables created are called Relations. A major feature of the relational model is that the information is stored into the tables so it is easier to extract specific data. Also you are able to use relational calculus to manipulate the data. The purpose of the data is to store the data and for processing.

#### 2.1.1 Description of E-R model and relational model

The ER model has entities, which are a group of definable things like customer, device. Entities are connected with a relationship. Relationships relate one entity with another, and it bonds two entities with each other for example, employee “works on” device. There are weak and strong entities. Strong entities can be defined on its own. Weak entities associate entities within an entity set. There are primary keys and foreign keys. A primary key are keys that are unique to each entity. For example, VIN to a vehicle. There can only be one VIN per vehicle. Foreign keys are keys that are used from other entities.

The relational model has a table that is called relation. Inside that table, there are rows. A row is called a tuple. The column header is called an attribute. Domain is the data type describing the types of values that can appear in each column. For example, USA_phone_number is the 10 digit phone numbers for the USA.

### 2.1.2 Comparison of Two Different Models

Between the two models, the ER diagram is the easiest to follow. The ER diagram allows users to follow the chart and play out how the database will run. An advantage that the ER diagram has is, everything is laid out for the user and does not require the user to think a lot to follow the flow of the diagram. It is the best when trying to reflect the real world in a database. The user can see the entities, which are things in the real world, like devices. Each entity will be connected to another entity by a relationship. That relationship can be anything from “works on” to “repairs”.

In a relational model, a table is called relations and within each table there are rows. Each row is called a tuple. The column header is called an attribute. Some people like the relational model more because it reminds them of an excel spreadsheet. People are used to seeing tables so they prefer this over the ER diagram. It is best used to represent data in a Relational Database Management System. A disadvantage is that it can provide a poor design and implementation if the user does not understand the tables. You cannot just look at the relational model and know what the database is about. You need to spend some time understanding it before you can understand it.

## 2.2 From Conceptual Database to Logical Database

In this section, we convert entity types and relationship types to relations. In order to use them in a database, you will need to use one of the many methods described below to convert them.

### 2.2.1 Converting Entity Types to Relations

Translating entity types to relations is quite easy thanks to the ER model. Each attribute will turn into a column in the table. The key attribute will become the primary key in the table. The primary key can never be null. Multi-valued attributes should be made into a table that's on its own. Add the primary column of the parent entity as a foregin key inside the new table.

The conversion from the ER model to Relational model is required in order to implement its data into the database because the ER model only describes the data requirement of each entity. A strong entity can be used as a primary key when converting. The primary key will be a unique number just like a deviceID. A weak entity cannot be alone and it relies on a primary key to represent the weak entity.

### 2.2.2 Converting Relationship Types to Relations

In a 1:1 mapping, the primary key can be copied onto the second table. Once its copied over to the second table then it becomes a foreign key. Any attributes tied to the relationship goes to the table with the foreign key. In a 1:N mapping, choose the entity on the N side and include a copy of the primary key of the 1 side. That will become a foreign key in the table on the N side. Once again, any attributes tied to the relationship goes to the table with the foreign key. In a N:M mapping, create a new table and include the primary keys from the two entities. Both of the fields will become foreign keys inside the new table. Then we can repeat the ending like the other tables, any attributes tied to the relationship goes in the new table.

You can create a table for the superclass and subclass. The superclass's primary key will be the primary key for both of the tables. For a Has a relationship, the recursive relation is formed if an entity has multiple items. A new foreign key will be created inside the relation that will reference the primary key. Categories or unions will have to link the child to the shared parent's primary key. Additional keys will be required if there is no common shared key between the parent and child.

### 2.2.3 Database Constraints

An entity constraint is where a primary key cannot be null. Only unknown values can be null, for example a person's repair date if they haven't brought anything in for repair. A primary key has to be a unique number and cannot be null. Each table will have a primary key and no two primary keys will be the same. A foreign key should match the data in the primary key it is referencing from the linked table.

A check constraint is applied to each tuple and is used to limit the value range in each of the tuples. Business rules are constraints that impose a form of constraint on the database. That database must be able to adhere to those that are imposed by the rules. In other words, they are a set of rules that the database must be able to follow and have been created by a customer.

## 2.3 Convert your E-R/Conceptual Database into a Relational/Logical Database

The purpose of section 2.3 is to help define the relations involved in the computer repair shop database while providing an example for each relation in the database. The relation tables in this section will specify the attributes, their constraints and their candidate keys.

### 2.3.1 Relation Schema for your Local Database

**Attribute:** Address
**Domain:**

| | |
|---|---|
| ADDR_ID | Integer, 000000000-999999999, PrimaryKey |
| city | Varchar (255) |
| state | Varchar (255) |

| | |
|---|---|
| zip | Integer, 00000-99999 |
| streetName | Varchar (20) |
| streetNum | Integer, 00000-99999 |

**Primary Key:** ADDR_ID
**PK Constraint:** No two addresses can have the same ADDR_ID
**Entity Integrity Constraint:** None of the attributes can be null, ADDR_ID cannot be null

**Attribute:** Customer
**Domain:**

| | |
|---|---|
| customerID | Integer, 0000000-999999, PrimaryKey |
| customerName | Varchar (255) |
| phoneNum | Varchar (20) |
| CCnum | Varchar (20) |
| cardType | Varchar (255) |

**Constraints:** customerID, customerName, and phoneNum must be given to be a valid customer
**Primary Key:** customerID
**PK Constraint:** The customerID cannot be null and no customer can have the same customerID
**Entity Integrity Constraint:** None of the attributes can be null except for CCnum and cardType

**Attribute:** Device
**Domain:**

| | |
|---|---|
| deviceID | Integer, 000000-999999, PrimaryKey |
| customerID | Integer, 000000-999999, ForeignKey |
| modelName | Varchar (255), ForeignKey |

**Primary Key:** deviceID
**PK Constraint:** deviceID cannot be null
**Entity Integrity Constraint:** Each deviceID must be unique but a device can have a similar customerID
**Referential Constraints:** customerID must exist in order for the Device to be accepted and the Model Name must exist for the device

**Attribute:** Employee
**Domain:**

| | |
|---|---|
| employeeID | Integer, 000000-999999. PrimaryKey |
| employeeName | Varchar (255) |
| e-mail | Varchar (255) |
| ADDR_ID (employee address) | Integer, 000000000-999999999, ForeignKey |
| SSN | Integer, 000000000-999999999 |

**Primary Key:** employeeID
**PK Constraint:** employeeID cannot be null
**Entity Integrity Constraint:** None of the attributes can be null to be valid
**Referential Constraints:** ADDR_ID must exist for an employee to be added

**Attribute:** Manufacturer
**Domain:**

| | |
|---|---|
| manufacturerID | Integer, 000000-999999, PrimaryKey |
| manufacturerName | Varchar (255) |
| phoneNum | Varchar (20) |
| ADRR_ID (manufacturerAddress) | Integer, 000000000-999999999, ForeignKey |

**Primary Key:** manufacturerID
**PK Constraint:** manufacturerID cannot be null
**Entity Integrity Constraint:** None of the attributes can be null
**Referential Constraints:** ADDR_ID must exist for a manufacturer to be added

**Attribute:** Model
**Domain:**

| | |
|---|---|
| modelName | Varchar (255), PrimaryKey |
| yearReleased | Date (Month - Year ), Jan 1950 - Present, PrimaryKey |
| manufacturerID | Integer, 0000000-999999, ForeignKey |

**Primary Key:** Combination of modelName and yearReleased
**PK Constraint:** No two laptop models can be the same when using modelName and yearReleased
**Referential Constraints:** A manufacturer must exist first in order for a Model to have been produced by them to be added into the catalog

**Attribute:** Order
**Domain:**

| | |
|---|---|
| orderID | Integer, 000000000-999999999, PrimaryKey |
| orderDate | Date (Month Number Year, Day) e.g. 01/23/2020, Thursday |
| orderStatus | Varchar (20) |
| employeeID | Integer, 000000-999999. ForeignKey |

**Primary Key:** orderID
**PK Constraint:** Each Order must have a unique orderID
**Entity Integrity Constraint:** None of the attributes can be null
**Referential Constraints:** An employeeID must exist in order for an Order to have been made

**Attribute:** Payment
**Domain:**

| | |
|---|---|
| paymentID | Integer, 000000000-999999999, PrimaryKey |
| paymentType | Integer, 0 - 1 |
| amount | Double (5,2) e.g 12345.99 |
| date-time | Date (Month Number Year, Time) |
| customerID | Integer, 0000000-999999, ForeignKey |
| repairOrderID | Integer, 000000000-999999999, ForeignKey |

**Primary Key:** paymentID
**PK Constraint:** Each Payment must have a unique paymentID
**Entity Integrity Constraint:** No attributes can be left null for a payment to be valid
**Referential Constraints:** A customerID and repairOrderID must exist in order for a Payment to be fulfilled

**Attribute:** RepairRequest
**Domain:**

| | |
|---|---|
| repairOrderID | Integer, 000000000-999999999, PrimaryKey |
| date-time | Date (Month Number Year, Time, Day) e.g. 01/23/2020, Thursday |
| dmgDesc | Varchar (255) |

| | |
|---|---|
| repairLog | Varchar (255) |
| typeOfRepair | Varchar (255) |
| empUsedPartsID | Integer, 000000-999999, ForeignKey |
| deviceID | Integer,  000000-999999, ForiegnKey |
| employeeID | Integer, 000000-999999. ForeignKey |

**Primary Key:** repairOrderID
**PK Constraint:** repairOrderID cannot be null
**Entity Integrity Constraint:** None of the attributes can be left null
**Referential Constraints:** empUsedPartsID cannot be empty and neither can deviceID or employeeID

**Attribute:** ReplacementParts
**Domain:**

| | |
|---|---|
| partID | Integer, 000000-999999, PrimaryKey |
| partName | Varchar (255) |
| sellCost | Double  (4,2) e.g. 1234.99 |
| quantity | Integer 000-999 |
| orderID | Integer, 000000000-999999999, ForeignKey |
| supplierID | Integer, 000000-999999, ForeignKey |

**Primary Key:** partID
**PK Constraint:**  ReplacementParts must have unique partID's
**Entity Integrity Constraint:** All attributes must be filled out to be valid
**Referential Constraints:** orderID and supplierID must exist in order for a replacement part to exist in the database

**Attribute:** Supplier
**Domain:**

| | |
|---|---|
| supplierID | Integer, 000000-999999, PrimaryKey |
| supplierName | Varchar (255) |
| ADDR_ID (supplierAddress) | Integer, 000000000-999999999, ForeignKey |

**Primary Key:** supplierID
**PK Constraint:** No two suppliers can have the same supplierID

**Entity Integrity Constraint:** No attributes can be null.
**Referential Constraints:** ADDR_ID must exist, can't be null

**Attribute:** contains
**Domain:**

| | |
|---|---|
| orderID | Integer, 000000000-999999999, ForeignKey |
| partID | Integer, 000000-999999, ForeignKey |
| boughtCost | Double (4,2) e.g. 1234.99 |
| quantity | Integer , 000-999 |

**Primary Key:** Combination of orderID and partID
**Entity Integrity Constraint:** orderID and partID cannot be null.
**Referential Constraints:** orderID and partID both must reference existing tuples.

**Attribute:** supplies
**Domain:**

| | |
|---|---|
| supplierID | Integer, 000000-999999, ForeignKey |
| partID | Integer, 000000-999999, ForeignKey |
| MSRP | Double (4,2) e.g. 1234.99 |

**Primary Key:** Combination of supplierID and partID
**Entity Integrity Constraint:** supplierID and partID cannot be null, neither can MSRP
**Referential Constraint:** supplierID and partID both must reference existing tuples.

**Attribute:** uses
**Domain:**

| | |
|---|---|
| empUsedPartsID | Integer, 000000-999999, ForeignKey |
| partID | Integer, 000000-999999, ForeignKey |
| quantity | Integer , 000-999 |

**Primary Key:** Combination of empUsedPartsID and partID
**Entity Integrity Constraint:** No attributes can be left null
**Referential Constraints:** empUsedPartsID must reference a partID that exists in replacementParts to know what was used in the repair

**Attribute:** works_on
**Domain:**

| | |
|---|---|
| EmployeeID | Integer, 000000-999999. PrimaryKey |
| RepairOrderID | Integer, 000000000-999999999, ForeignKey |
| empUsedPartsID | Integer, 000000-999999, ForeignKey |
| hoursWorked | Varchar (255) |

**Primary Key:** Combination of employeeID, repairOrderID and empUsedPartsID
**Entity Integrity Constraint:** employeeID, repairOrderID and empUsedPartsID cannot be null
**Referential Constraints:** and employeeID must exist from Employee who works on a RepairRequest where a repairOrderID exists and a list of replacement parts come from the attribute empUsedPartsID

### 2.3.2 Sample Data of Relation

**Address**

| ADDR_ID | City | State | ZipCode | Street Name | Street Num |
|---|---|---|---|---|---|
| 153678925 | Bakersfeild | CA | 93309 | Wernli St | 1046 |
| 919385282 | San Francisco | CA | 94115 | Jackson St | 224 |
| 209573958 | San Jose | CA | 95050 | Oak Dr | 7434 |
| 989034962 | New York | NY | 10005 | Capital Dr | 13263 |
| 430967223 | Dallas | TX | 75063 | Yehaw St | 3623 |
| 294363212 | Aurora | CO | 80011 | Park Ave | 123 |
| 547545223 | Bakersfield | CA | 93306 | Ming Ave | 1641 |
| 130989023 | Bakersfeild | CA | 93309 | California Ave | 134262 |
| 908262709 | Miami | FL | 33125 | Mardi St | 2364 |
| 236420839 | Chicago | IL | 60606 | Lincoln Dr | 1233 |

**Customer**

| CustomerID | CustomerName | PhoneNum | CC Num | Card Type |
|---|---|---|---|---|
| 523612 | Ara | 661-447-1235 | 2357851347531346 | VISA |
| 357352 | Andy | 121-135-1251 | 2357583477533467 | MASTER CARD |
| 686423 | Tom | 756-123-6532 | 3428134637573467 | VISA |
| 135721 | Richard | 235-575-1236 | 1234123445674125 | VISA |
| 978311 | Ceci | 136-112-1111 | 2586756653432572 | VISA |
| 3284321 | Jackson | 642-652-1363 | 8996573347686544 | MASTER CARD |
| 2375722 | Edith | 654-236-234 | 3797875465288824 | VISA |
| 1375785 | Robert | 534-123-1363 | 4686865546546333 | VISA |
| 2467823 | Deru | 765-236-1237 | 6896784754633546 | MASTER CARD |
| 1362412 | Natalie | 312-236-2361 | 4632452346367135 | VISA |

**Device**

| DeviceID | customerID | Model Name |
|---|---|---|
| 72334 | 523612 | Dell XPS 13 |
| 23421 | 357352 | Alienware 15 |
| 78523 | 686423 | Thinkpad Yoga |
| 43471 | 135721 | Thinkpad 420 |
| 78854 | 978311 | Dell XPS 15 |
| 23454 | 3284321 | Alienware m15 |
| 42346 | 2375722 | Macbook Pro 2019 |
| 23462 | 1375785 | Macbook Air 2012 |
| 23422 | 2467823 | LG Gram 17 |
| 67772 | 1362412 | Zenbook 13 UX333 |

**Employee**

| **EmployeeID** | **Employee Name** | **E-Mail** | **ADDR_ID** | **SSN** |
|---|---|---|---|---|
| 47324 | Rick Mon | rick@gmail.com | 153678925 | 373554234 |
| 57232 | Erin Snyder | eSnyder@gmail.com | 919385282 | 324753783 |
| 12472 | Kaly Lynn | Lynn@gmail.com | 209573958 | 444472341 |
| 86524 | Lily Shot | LilShot@gmail.com | 989034962 | 548757333 |
| 99813 | Chris Lopex | Lo@gmail.com | 430967223 | 357877222 |
| 08632 | Jose Aguilar | AguilarJ@gmail.com | 294363212 | 536245211 |
| 47234 | Hamilton Alex | HamAlex@gmail.com | 547545223 | 357854124 |
| 13642 | Alex Smith | ASmith@gmail.com | 130989023 | 537123411 |
| 55711 | Don Jon | DonJon@gmail.com | 908262709 | 434671212 |
| 57534 | Al Phonse | AlPhonse@gmail.com | 236420839 | 237572234 |

**Manufacturer**

| **ManufacturerID** | **Manufacturer Name** | **Phone Num** | **ADDR_ID** |
|---|---|---|---|
| 112314 | Apple | 1-800-275-2273 | 153678925 |
| 135234 | Dell | 1-800-999-3355 | 919385282 |
| 842341 | Sony | 1-800-538-7550 | 209573958 |
| 342123 | Toshiba | 1-800-457-7777 | 989034962 |
| 234623 | Lenovo | 1-855-253-6686 | 430967223 |
| 856312 | Microsoft | 1-800-642-7676 | 294363212 |
| 335411 | Asus | 1-800-820-6655 | 547545223 |
| 437312 | LG | 1-866-558-2559 | 130989023 |
| 352131 | HP | 1-800-474-6836 | 908262709 |

| | | | |
|---|---|---|---|
| 111123 | MSI | 1-888-447-6564 | 236420839 |

**Model**

| **Model Name** | **Year Released** | **Manufacturer ID** |
|---|---|---|
| Dell XPS 13 | 2019 | 112314 |
| Alienware 15 | 2019 | 135234 |
| Thinkpad Yoga | 2016 | 842341 |
| Thinkpad 420 | 2010 | 342123 |
| Dell XPS 15 | 2020 | 234623 |
| Alienware m15 | 2018 | 856312 |
| Macbook Pro | 2019 | 335411 |
| Macbook Air | 2012 | 437312 |
| LG Gram 17 | 2019 | 352131 |
| Zenbook 13 UX333 | 2017 | 111123 |

**Order**

| **OrderID** | **Order Date** | **Order Status** | **EmployeeID** |
|---|---|---|---|
| 134724122 | 01/23/2020 | Received | 47324 |
| 712412154 | 01/03/2020 | Received | 57232 |
| 124672431 | 03/13/2020 | Canceled | 12472 |
| 357253442 | 05/22/2020 | Returned | 86524 |
| 573321151 | 12/11/2020 | Received | 99813 |
| 658562311 | 11/16/2020 | Canceled | 08632 |
| 247587354 | 06/12/2020 | Canceled | 47234 |
| 357232416 | 06/26/2020 | Returned | 13642 |

| | | | |
|---|---|---|---|
| 357834251 | 03/12/2020 | Received | 55711 |
| 454624611 | 07/23/2020 | Damaged | 57534 |

**Payment**

| **PaymentID** | **Payment Type** | **Amount** | **Date-Time** | **CustomerID** | **RepairOrderID** |
|---|---|---|---|---|---|
| 358463223 | Cash | 132.23 | 12/23/2020 12:23 | 523612 | 573237962 |
| 344367671 | Cash | 35.00 | 03/03/2020 11:11 | 357352 | 345824342 |
| 137513713 | Cash | 24.99 | 06/13/2020 09:10 | 686423 | 315878235 |
| 314673517 | Card | 250.00 | 05/22/2012 09:11 | 135721 | 135735131 |
| 797863234 | Card | 70.00 | 12/11/2013 07:11 | 978311 | 248435413 |
| 53722321 | Cash | 45.25 | 10/16/2010 11:30 | 3284321 | 344689414 |
| 12345161 | Cash | 46.99 | 04/12/2015 12:22 | 2375722 | 234753734 |
| 46162321 | Card | 120.00 | 04/26/2012 03:22 | 1375785 | 358731234 |
| 31463412 | Cash | 180.25 | 02/12/2010 01:22 | 2467823 | 341871349 |
| 36745424 | Card | 123.99 | 07/23/2020 1:36 | 1362412 | 898585445 |

**RepairRequest**

| **RepairOrderID** | **Date** | **DmgDesc** | **RepairLog** | **TypeOfRepair** |
|---|---|---|---|---|
| 394689403 | 12/23/2020 12:23, Thusday | Broken screen | Removed screen and replaced | Screen replacement |
| 347585234 | 03/03/2020 11:11, Friday | Broken Hinge | Replaced hinge from donor laptop | Hinge replacement |
| 789867875 | 06/13/2020 09:10, Monday | Not booting up | Swapped out old drive for newer one | Fresh install |
| 789543434 | 05/22/2012 09:11, | Broken screen | Replaced screen | Screen |

| | | | | |
|---|---|---|---|---|
| | Tuesday | | | replacement |
| 346765473 | 12/11/2013 07:11, Friday | Not charging | Replaced charging port after checking voltages | Replace charging port |
| 567467463 | 10/16/2010 11:30, Saturday | Screen won't turn on | Ribbon cable was damaged, swapped out entire screen | Screen replacement |
| 789757334 | 04/12/2015 12:22. Monday | Charges but screen won't turn on | Dead drive, did a fresh install on a new drive | Fresh install |
| 458678245 | 04/26/2012 03:22, Monday | Random crashes | Wiped drive and installed a fresh copy of windows | Fresh Install and Data Recovery |
| 689787342 | 02/12/2010 01:22, Tuesday | Loss date | Copied all corrupted data and rebuilt it, stored recovered data on external flash drive | Data Recovery |
| 463879435 | 07/23/2020 1:36, Wednesday | Battery only lasts an hour | Replaced the battery inside of the unit | Battery swap |

| **empUsedPartsID** | **DeviceID** | **EmployeeID** |
|---|---|---|
| 425725 | 72334 | 47324 |
| 457234 | 23421 | 57232 |
| 789734 | 78523 | 12472 |
| 745232 | 43471 | 86524 |
| 988567 | 78854 | 99813 |
| 376842 | 23454 | 08632 |
| 357482 | 42346 | 47234 |
| 573321 | 23462 | 13642 |

|  |  |  |
|---|---|---|
| 358647 | 23422 | 55711 |
| 973523 | 67772 | 57534 |

**ReplacementParts**

| **PartID** | **Part Name** | **Sell Cost** |
|---|---|---|
| 456846 | Macbook screen kit | 84.99 |
| 795743 | Sandisk 128GB SSD | 40.00 |
| 548632 | Baracuda 1TB HDD | 60.00 |
| 357234 | Crucial 1TB SSD | 150.99 |
| 243216 | MacbookPro 2019 Replacement Keyboard | 50.00 |
| 132624 | Windows 10 Registry Key | 120.00 |
| 112253 | Recovery Media Tool | 12.00 |
| 234243 | Dell XPS Screen Kit | 89.99 |
| 344454 | 92 KwHr Battery | 99.99 |
| 123633 | 52 KwHr Battery | 69.99 |

| **Quantity** | **OrderID** | **SupplierID** |
|---|---|---|
| 5 | 134724122 | 346722 |
| 10 | 712412154 | 423674 |
| 10 | 124672431 | 865756 |
| 5 | 357253442 | 823121 |
| 15 | 573321151 | 133161 |
| 30 | 658562311 | 437321 |
| 15 | 247587354 | 853645 |

| | | |
|---|---|---|
| 5 | 357232416 | 473111 |
| 5 | 357834251 | 247341 |
| 5 | 454624611 | 458742 |

**Supplier**

| SupplierID | Supplier Name | ADDR_ID |
|---|---|---|
| 346722 | Apple | 153678925 |
| 423674 | Amazon | 919385282 |
| 865756 | E-Bay | 209573958 |
| 823121 | Dell | 989034962 |
| 133161 | Samsung | 430967223 |
| 437321 | Toshiba | 294363212 |
| 853645 | Sony | 547545223 |
| 473111 | Alibaba | 130989023 |
| 247341 | Aliexpress | 908262709 |
| 458742 | Newegg | 236420839 |

**Contains**

| OrderID | PartD | Quantity |
|---|---|---|
| 134724122 | 456846 | 10 |
| 712412154 | 795743 | 15 |
| 124672431 | 548632 | 12 |
| 357253442 | 357234 | 4 |
| 573321151 | 243216 | 2 |

| | | |
|---|---|---|
| 658562311 | 132624 | 16 |
| 247587354 | 112253 | 11 |
| 357232416 | 234243 | 12 |
| 357834251 | 344454 | 4 |
| 454624611 | 123633 | 4 |

**Supplies**

| **SupplierID** | **PartID** | **MSRP** |
|---|---|---|
| 346722 | 456846 | 64.99 |
| 423674 | 795743 | 50.00 |
| 865756 | 548632 | 120.00 |
| 823121 | 357234 | 45.00 |
| 133161 | 243216 | 100.00 |
| 437321 | 132624 | 10.00 |
| 853645 | 112253 | 69.99 |
| 473111 | 234243 | 89.99 |
| 247341 | 344454 | 89.99 |
| 458742 | 123633 | 55.99 |

**Uses**

| **EmpUsedPartsID** | **PartID** | **Quantity** |
|---|---|---|
| 425725 | 456846 | 10 |
| 457234 | 795743 | 15 |
| 789734 | 548632 | 12 |
| 745232 | 357234 | 4 |

| | | |
|---|---|---|
| 988567 | 243216 | 2 |
| 376842 | 132624 | 16 |
| 357482 | 112253 | 11 |
| 573321 | 234243 | 12 |
| 358647 | 344454 | 4 |
| 973523 | 123633 | 4 |

**Works_On**

| EmployeeID | RepairOrderID | EmpUsedPartsID | Hours Worked |
|---|---|---|---|
| 47324 | 573237962 | 425725 | 02:30:14 |
| 57232 | 345824342 | 457234 | 03:12:15 |
| 12472 | 315878235 | 789734 | 00:20:56 |
| 86524 | 135735131 | 745232 | 00:45:13 |
| 99813 | 248435413 | 988567 | 02:14:16 |
| 08632 | 344689414 | 376842 | 03:22:11 |
| 47234 | 234753734 | 357482 | 00:25:16 |
| 13642 | 358731234 | 573321 | 00:20:45 |
| 55711 | 341871349 | 358647 | 00:46:34 |
| 57534 | 898585445 | 973523 | 00:34:32 |

## 2.4 Sample Queries to your Database

**Relations**
Address (ADDR_ID, city, state, zip, streetName, streetNum )
Customer (customerID, customerName, phoneNum, CCnum, cardType )
Device (deviceID, customerID, modelName)
Employee (employeeID, employeeName, phoneNum, e-mail, ADDR_ID, ssn )

Manufacturer (manufacturerID, manufacturerName, phoneNum, ADDR_ID )
Model (modelName, yearReleased, manufacturerID )
Order (orderID, orderDate, orderStatus, employeeID )
Payment (paymentID, paymentType, amount, date-time, customerID, repairOrderID)
RepairRequest (repairOrderID, date, dmgDesc, repairLog, typeOfRepair, empyUsedPartsID, deviceID, employeeID )
ReplacementParts (partID, partName, sellCost, quantity, orderID, supplierID )
Supplier (supplierID, supplierName, ADDR_ID )

Contains (orderID, partID, quantity, boughtCost )
Supplies (supplierID, partID, MSRP )
Uses (empUsedPartsID, partID, quantity )
Works_On (employeeID, repairOrderID, empUsedPartsID, hoursWorked )

### 2.4.1 Design of Queries

1. What is the lowest a customer has paid for a repair ?
2. What is the highest a customer has paid for a repair ?
3. List employees who have works on at least one device for each manufacturer
4. List employees who worked on a single device for longer than 8 hours
5. List all repairs that had to deal with broken screens
6. List all repair jobs for macbooks
7. List all employees who have worked on Daves repairs
8. List all employees who have worked on Daves repairs in March
9. Which customers paid nothing for their repair ?
10. List all employees who have cancelled an order.

### 2.4.2 Relational Algebra Expressions for Queries of 4.1

1.$\sigma_{amount1<amount2}(Payment)$

2.$\sigma_{amount1>amount2}(Payment)$

3.

4.

5.$\pi_{dmgDesc="broken\ screen"}(RepairRequest)$

6.$\pi_{modelName = "macbook"}(Device)$

7.$\pi_{EmployeeName}(\sigma_{customerName = "Dave" \wedge CustomerID \wedge DeviceID} Customer \times RepairRequest \times Employee)$

8.

$\pi_{EmployeeName \wedge Date = "March"}(\sigma_{CustomerName = "Dave" \wedge CustomerID \wedge DeviceID} Customer\ x\ Device \times RepairRequest \times Emplo$

9.$\sigma_{CustomerName}(\pi_{amount = 0}(Payment \times Customer))$

10. $\pi_{EmployeeName}(\sigma_{orderStatus = "canceled" \wedge employeeID}(Order \times Employee))$

### 2.4.3 Tuple Relational Calculus Expressions for Queries

### 2.4.4 Domain Relational Calculus Expressions for Queries

# Phase III: Relational Database Normalization and Implementation

## 3.1 Normalization of Relations

### Normalization

The process of normalization is a top-down process that evaluates each relation against the criteria of normal forms and decomposing relations as necessary and can be considered relational design by analysis. It takes a relation schema through a series of tests to certify whether it does or doesn't satisfy a certain normal form. These normal forms are the First Normal Form, Second Normal Form, Third Normal Form and Boyce-Codd Normal Form. Other normal forms include Fourth Normal Form and Fifth Normal Form but these are of much rarer instances. The first three forms were all proposed by Codd in 1972.

The normalization of data can be viewed as taking said data and analyzing it against the given relation schemas based on their FD's and primary keys in order to achieve a desirable database design. The two main goals of the normalization process are to minimize redundancy and minimize the insertion, deletion and update anomalies.

If a relation schema does not pass certain conditions then there are the normal form tests in which data is decomposed into smaller relation schemas that can pass certain conditions and be used in a desirable database.

### 1st Normal Form

The First Normal Form (1NF) is part of the formal definition of a relation in the basic relational model. This means it was defined to not allow multivalued attributes, composite attributes and their combinations. First Normal Form relations  state that the domain of an attribute must include only individual atomic values.

There are 3 techniques when it comes to achieving 1NF for relations. The first includes  creating two 1NF relations by decomposing the non-1NF relation. This is typically the  best because it doesn't suffer from redundancy and is completely generalized . It has no limit placed on the maximum number of values  it can have. The second solution is to take the primary key and have it become the combination of a separate tuple, thereby expanding the key. This solution has one disadvantage of introducing redundancy into the relation. Lastly, the third solution is to replace an attribute into n-amount of atomic attributes if the maximum number of values is known. However, with this type of solution, there can be NULL values placed inside, making it more difficult to write queries.

## 2nd Normal Form

The 2nd Normal Form (2NF), is based on the concept of full functional dependency. Full functional dependency is when $X \rightarrow Y$ if removal of any attribute A inside of X means that the dependency doesn't hold anymore.For any attribute $A \varepsilon X$, $(X - \{A\})$ does not functionally determine Y. If some attribute from $A \varepsilon X$ can be removed from X and the dependency still holds, then this would be labeled as a partial dependency.

The solution to 2NF, normalization, is to decompose and set up a new relation for each partial key with its dependent attribute(s). Making sure to keep relations with the original primary key and any attributes that are fully functionally dependent on it.

## 3rd Normal Form

The 3rd Normal Form (3NF), is based on the concept of transitive dependency. Transitive dependency is when $X \rightarrow Y$ in a relation schema R if there exists a set of attributes Z in R that is neither a candidate key nor a subset of any key R and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. If a relation schema R is in 3NF then it also satisfies 2NF and no non prime attribute of R is transitively dependent on the primary key.

The way to remedy 3NF is to decompose and set up a relation that includes the nonkey attribute(s) that functionally determine other nonkey attribute(s).

## Boyce-Codd Normal Form

The Boyce-Codd Normal Form (BCNF), was proposed as a simpler form of the 3NF. However, it was found out that it was more strict than the 3NF. This means that every BCNF is a 3NF but not vice versa. The definition for BCNF is as follows, a relation schema R is in BCNF if whenever a nontrivial function dependency $X \rightarrow A$ holds in R, then X is a superkey of R.

This form was developed by two gentlemans, Raymond F. Boyce and Edgar F. Codd in the 1970's to help solve anomalies that the 3rd Normalization Form did not deal with. Some database schemas that pass the 3rd Normalization Form may pass the Boyce-Codd Normalization Form if they don't have overlapping candidate keys.

## Update Anomalies

Update anomalies are referred to as problems when storing natural joins of base relations. These sort anomalies can further be classified as insertion anomalies, deletion anomalies and modification anomalies. The goal of schema designs are to minimize these sort of anomalies that can appear. One method is to minimize the storage space used by the base relations. A method to do this would be to group attributes together into relation schemas which can have a significant effect on storage space. However with this method anomalies can appear such as repeated information.

### Insertion Anomalies

These sort of anomalies occur when trying to insert new attributes into the database without the presence of other attributes that need to be known before being able to insert the new attribute. This can lead to not being able to insert any new attributes and can lead to inserting NULL values into attributes whose information is needed but not known when attempting to insert new attributes. These anomalies are largely caused by poor database design.

### Deletion Anomalies

Deletion anomalies are the opposite side of insertion anomalies. In the case of a deletion anomaly, it is when tuples are being deleted from the database and in doing so another related set of data is lost unintentionally. This happens when a relation holds data that did not really belong together into one table such as an employee relation holding data on a company project instead of the projects having their own table. This can happen when trying to delete one tuple or a certain set of tuples from a relation. To use the example of the employee/project relation used before if we deleted all employee tuples that worked on a particular project all data about the project would be lost as it was stored with the employee's data.

The three anomalies, update anomalies, insertion anomalies and deletion anomalies are undesirable in any database. They can cause difficulties in maintaining consistent data as well as require unnecessary updates that could have been avoided. It is thanks to the process of normalization where anomalies can be avoided.

### Modification Anomalies

Modification anomalies can develop when a record of data in the database is being changed. If one piece of data that needs to be changed is a foreign key in another table then the subset of tuples in the corresponding tables that have that value as a foreign key must be changed. If every tuple is not updated then the data could be inconsistent and subsequently incorrect and would cause a modification anomaly.

### 3.1.2 Test of normalization of relations

Address (ADDR_ID, city, state, zip, streetName, streetNum )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'ADDR_ID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

If Address was its own attribute for employee, manufacturer or supplier, then it would violate the 1st normalization form because it would not be atomic. This is the reason why address has become its own relation instead of keeping it as an attribute.

Customer (customerID, customerName, phoneNum, CCnum, cardType )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'customerID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, once you move over the CCnum and cardType attributes into a new table called "Customer Card", which will have the attributes 'customerID', 'CCnum', and 'cardType'.

Device (deviceID, customerID, modelName)
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'deviceID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Employee (employeeID, employeeName, phoneNum, e-mail, ADDR_ID, ssn )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'employeeID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

If ADDR_ID was just an address attribute then the 1st normalization form would have been violated.

Manufacturer (manufacturerID, manufacturerName, phoneNum, ADDR_ID )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'manufacturerID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

If ADDR_ID was just an address attribute then the 1st normalization form would have been violated.

Model (modelName, yearReleased, manufacturerID )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'modelName' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Order (orderID, orderDate, orderStatus, employeeID )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'orderID' consists of a single attribute with no partial dependency on anything else.

**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Payment (paymentID, paymentType, amount, date-time, customerID, repairOrderID)
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'paymentID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

RepairRequest (repairOrderID, date, dmgDesc, repairLog, typeOfRepair, empUsedPartsID, deviceID, employeeID )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'repairOrderID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

ReplacementParts (partID, partName, sellCost, quantity, orderID, supplierID )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'partID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Supplier (supplierID, supplierName, ADDR_ID )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the primary key 'supplierID' consists of a single attribute with no partial dependency on anything else.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

If ADDR_ID was just an address attribute then the 1st normalization form would have been violated.

Contains (orderID, partID, quantity, boughtCost )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the two primary keys 'orderID' and 'partID' do not have partial dependency from the other attributes.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Supplies (supplierID, partID, MSRP )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the two primary keys 'supplierID' and 'partID' do not have partial dependency from the other attributes.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Uses (empUsedPartsID, partID, quantity )

**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the two primary keys ‘empUsedPartsID’ and ‘partID’ do not have partial dependency from the other attributes.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

Works_On (employeeID, repairOrderID, empUsedPartsID, hoursWorked )
**1NF:** Satisfied, all attributes are single and atomic.
**2NF:** Satisfied, the three primary keys ‘employeeID’ and ‘repairOrderID’ and ‘empUsedPartsID’ do not have partial dependency from the other attributes.
**3NF:** Satisfied, because no non-prime attribute depends on another non-prime attribute.

## 3.2 PostgreSQL - Main Purpose & Functionality

PostgreSQL, or more alternatively known as Postgres, is a general-purpose open source object-relational database management system which was written in the C-language. This management system was originally developed at the University of California, Berkley and is currently being developed by the PostgreSQL Global Development Group. The group is made up of many individuals and companies who contribute to this free and open-source project. This software can be used for a wide range of clients, including single individual users who need it for personal hobbies, or for massive projects that where multiple users are included.

The overall main purpose of a database management system like PostgreSQL is to give its users the tools and power to effectively and efficiently handle the flow of data. Giving them control of how data can be inserted, deleted and modified.

Meanwhile, PostgreSQL contains multiple types of functionality within its language. Plenty of the functionality can be traced back to another popular language, SQL, which stands for ‘Structured Query Language’. Functionality from the SQL standards include: Data Types, Data Integrity, Concurrency, Performance,Reliability, Disaster Recovery, Security, Extensibility, Internationalisation,Text Search and many others. Not only does PostgreSQL contain SQL standards, it also contains many of the SQL commands needed to manage a database. These commands help with inserting, updating, and deleting data. WIth these sorts of tools at the disposal of its users, it lets users create and efficiently maintain databases while being able to query and search through tons of information to solve problems.

## 3.3 Schema Objects in Postgres DBMS

This section of the paper outlines the multitude of objects that users can use in the postgres database management system. The schema objects in the Postgres DBMS are tables, views, procedures, triggers, packages, sequence generator and drops/insertion of data.

### Table

Definition/Purpose: Tables are units of data that are stored inside of the database you create. Each table has a column defining a single attribute of the relational schema while a row consists of data from a combination of multiple attributes from each column. In our table we have column names such as (emp_ID, emp_name, address, SSN), with each one being specified a data type, (concurrently our attributes data types are INTEGER, VARCHAR, VARCHAR and INTEGER ). Once data is inside of the database, you can control it through updating, deleting or querying languages using the PSQL language.

SchemaObjects:

- Address, Customer, Employee, etc.

Syntax:

```
CREATE TABLE my_first_table (
   first_column text,
   second_column integer
);
```

### Schemas

Definition/Purpose: Schemas are basically a number of tables grouped together. Schemas can have functionality to help make using and maintaining the database through the use of views, indexes, sequences, data types, operators and functions. Schemas allows users to use one database without interfering with others by making them into logical groups that make them manageable.

SchemaObjects:

- Public schema for our database

Syntax:

```
CREATE SCHEMA my_schema (
. . .
);
```

### Tablespaces

Definition/Purpose: Table spaces in the postgres DBMS are spaces on the actual disk, a physical location where the user can map a logical name for a database. These tablespaces come in two default tablespaces, pg_default, where they store all user data and pg_global where they store all global data. The purpose of tablespaces is that it allows users to control the size of a database cluster, if there isn't enough room for it to grow, they can just initialize space outside of that area and then move that partition over until the system is reconfigured. The later purpose is to help make database objects more efficient in their performance.

SchemaObjects:

- None in our database

Syntax:

CREATE TABLESPACE my_tablespace
OWNER user_name
LOCATION directory_path ;

### Views

Definition/Purpose: Views are the queries stored as virtual tables; Although they do not store data, they are accessible through the SELECT statement to a tailored representation of the data. The benefits of this include that the Database Administrator is able to control the data that is available. Typically, views are the preferred method of accessing data while using front-end applications. This allows the data to be made dynamically.

SchemaObjects:

- Will be created for the reports

Syntax:

CREATE VIEW view_name AS query ;

### Functions

Definition/Purpose: Functions are also known as a stored procedure. They work in the same way the name is made, functions are a set of procedures/procedural statements that are stored together to carry out a task in the database under a function name.  This function name is then called whenever you want those stored procedures to be invoked. This helps make long tedious tasks that the user usually does manually through several statements, done within a single call of the function.

SchemaObjects:

- Will create when creating functions for the website

Syntax:

CREATE FUNCTION function_name ( arguments )

```
RETURNS return_datatype AS $variable_name$
        DECLARE
                declaration;
                [ . . . ]
        BEGIN
                <function_body>
                [ . . . ]
                RETURN { variable_name | value }
        END; LANGUAGE plpgsql;
```

## Operators

Definition/Purpose: Operators in the Postgres DBMS are keywords/characters that have been reserved to be used inside of a Postgres SQL statements WHERE clause. They help facilitate operations either arithmetically or through comparison of two variables. These keywords/characters also include logical operators and bit string operators. A simple example can be a doing varA + varB.

SchemaObjects:

- Will be used for filtering queries in the reports

Syntax:

```
SELECT select_list
FROM table_name
WHERE condition ;
```

( condition is where the operators would be used.
E.g.  WHERE class_name = “Database Systems” OR class_name = “DB Systems” ;
In this case we used the “=” and “OR” operators )

## Casts

Definition/Purpose: Casts is another keyword reserved in the Postgres DBMS that defines itself as an operator. The purpose of this operator is to convert a value of  one data type into another data type. An example of this could be casting a date received from a DATE data type into a VARCHAR data type to make it easier to handle or vice versa depending on the situation. .

SchemaObjects:

- None

Syntax:

```
CAST ( expression AS target_type ) ;
```

## Sequence

Definition/Purpose: Sequences are used to create a series of integers that follow a pattern in an either increasing or decreasing fashion. The user can state where they wish the minvalue to start, the maxvalue to end and how much to shift by between those two values until the maxvalue is met. The purpose of this object is to help create sequences into a statement you're needing. For example, an insert statement where you want the id of each item to be an increment of 10 instead of 1.

SchemaObjects:

- The id who are auto incrementing in our tables following an incrementing sequence of 1.

Syntax:
CREATE SEQUENCE sequence_name ;

### Extension

Definition/Purpose: Extension objects can be attached to the current database by loading them in. One purpose for the extension object is to give databases more built-in functionality once the extension is loaded in. Some of this functionality can be creating unique data types.

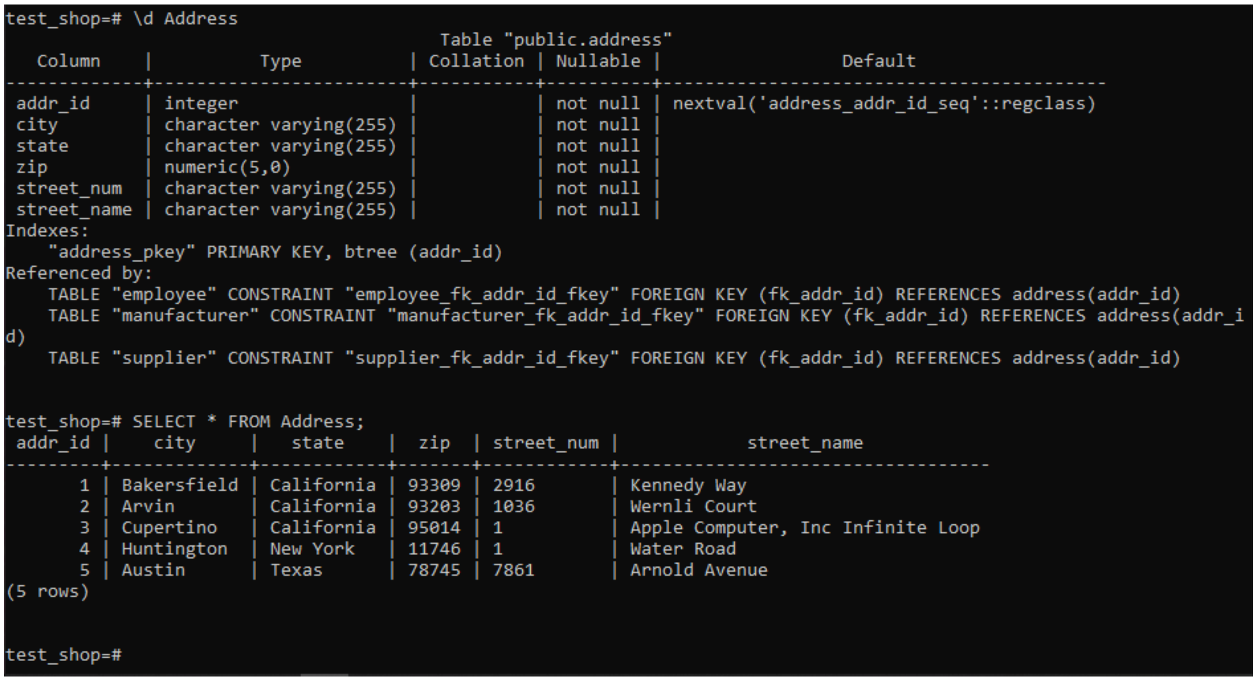
SchemaObjects:

- None

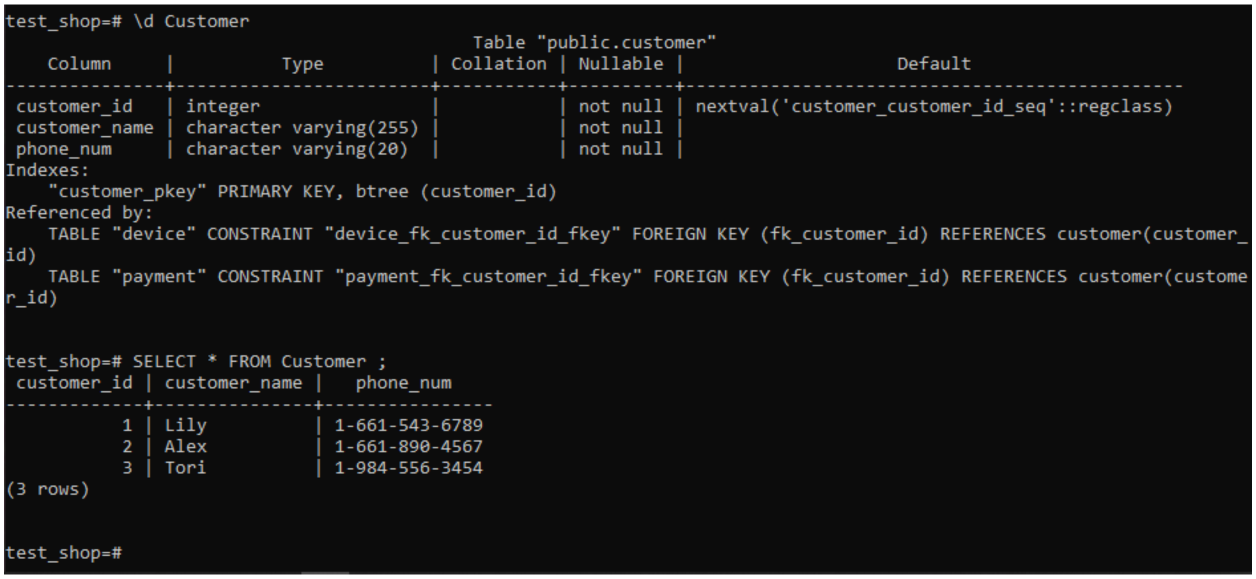
Syntax:
CREATE EXTENSION extension_name ;

## 3.4 List Relations Schema and Contents

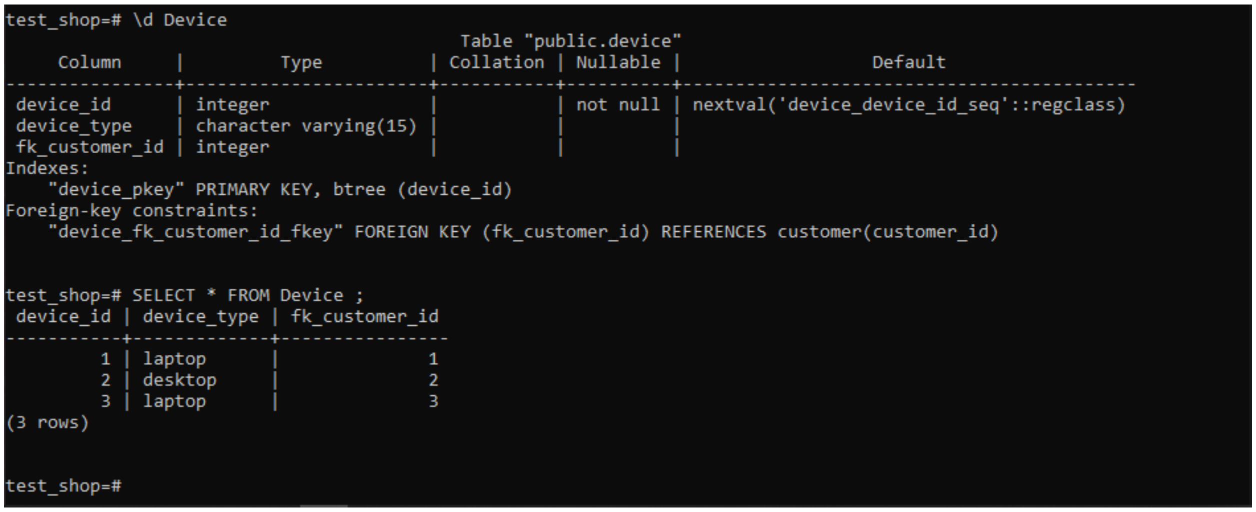
Address:


```
test_shop=# \d Address
                                        Table "public.address"
   Column    |          Type          | Collation | Nullable |                 Default
-------------+------------------------+-----------+----------+------------------------------------------
 addr_id     | integer                |           | not null | nextval('address_addr_id_seq'::regclass)
 city        | character varying(255) |           | not null |
 state       | character varying(255) |           | not null |
 zip         | numeric(5,0)           |           | not null |
 street_num  | character varying(255) |           | not null |
 street_name | character varying(255) |           | not null |
Indexes:
    "address_pkey" PRIMARY KEY, btree (addr_id)
Referenced by:
    TABLE "employee" CONSTRAINT "employee_fk_addr_id_fkey" FOREIGN KEY (fk_addr_id) REFERENCES address(addr_id)
    TABLE "manufacturer" CONSTRAINT "manufacturer_fk_addr_id_fkey" FOREIGN KEY (fk_addr_id) REFERENCES address(addr_i
d)
    TABLE "supplier" CONSTRAINT "supplier_fk_addr_id_fkey" FOREIGN KEY (fk_addr_id) REFERENCES address(addr_id)


test_shop=# SELECT * FROM Address;
 addr_id |    city    |   state    |  zip  | street_num |           street_name
---------+------------+------------+-------+------------+----------------------------------
       1 | Bakersfield | California | 93309 | 2916       | Kennedy Way
       2 | Arvin      | California | 93203 | 1036       | Wernli Court
       3 | Cupertino  | California | 95014 | 1          | Apple Computer, Inc Infinite Loop
       4 | Huntington | New York   | 11746 | 1          | Water Road
       5 | Austin     | Texas      | 78745 | 7861       | Arnold Avenue
(5 rows)


test_shop=#
```

Customer:

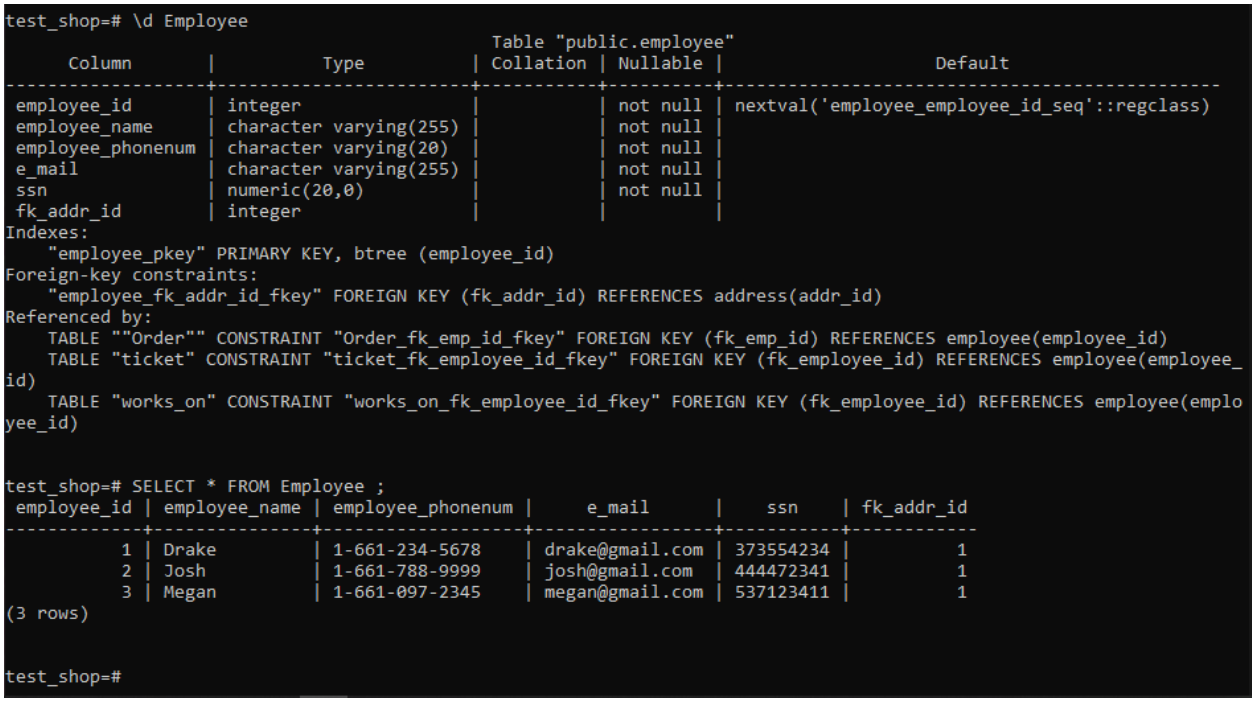

```
test_shop=# \d Customer
                                          Table "public.customer"
    Column     |          Type          | Collation | Nullable |                   Default
---------------+------------------------+-----------+----------+----------------------------------------------
 customer_id   | integer                |           | not null | nextval('customer_customer_id_seq'::regclass)
 customer_name | character varying(255) |           | not null |
 phone_num     | character varying(20)  |           | not null |
Indexes:
    "customer_pkey" PRIMARY KEY, btree (customer_id)
Referenced by:
    TABLE "device" CONSTRAINT "device_fk_customer_id_fkey" FOREIGN KEY (fk_customer_id) REFERENCES customer(customer_
id)
    TABLE "payment" CONSTRAINT "payment_fk_customer_id_fkey" FOREIGN KEY (fk_customer_id) REFERENCES customer(custome
r_id)


test_shop=# SELECT * FROM Customer ;
 customer_id | customer_name |   phone_num
-------------+---------------+---------------
           1 | Lily          | 1-661-543-6789
           2 | Alex          | 1-661-890-4567
           3 | Tori          | 1-984-556-3454
(3 rows)


test_shop=#
```

Device:



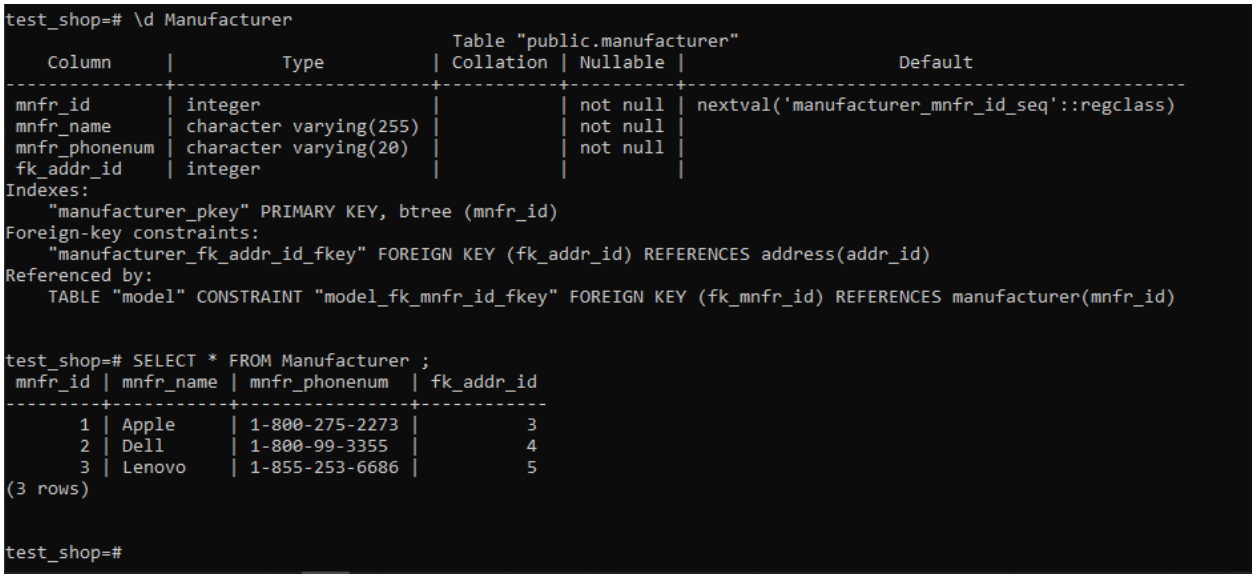
```
test_shop=# \d Device
                                        Table "public.device"
     Column     |         Type          | Collation | Nullable |                   Default
----------------+-----------------------+-----------+----------+---------------------------------------------
 device_id      | integer               |           | not null | nextval('device_device_id_seq'::regclass)
 device_type    | character varying(15) |           |          |
 fk_customer_id | integer               |           |          |
Indexes:
    "device_pkey" PRIMARY KEY, btree (device_id)
Foreign-key constraints:
    "device_fk_customer_id_fkey" FOREIGN KEY (fk_customer_id) REFERENCES customer(customer_id)


test_shop=# SELECT * FROM Device ;
 device_id | device_type | fk_customer_id
-----------+-------------+----------------
         1 | laptop      |              1
         2 | desktop     |              2
         3 | laptop      |              3
(3 rows)


test_shop=#
```

Employee:



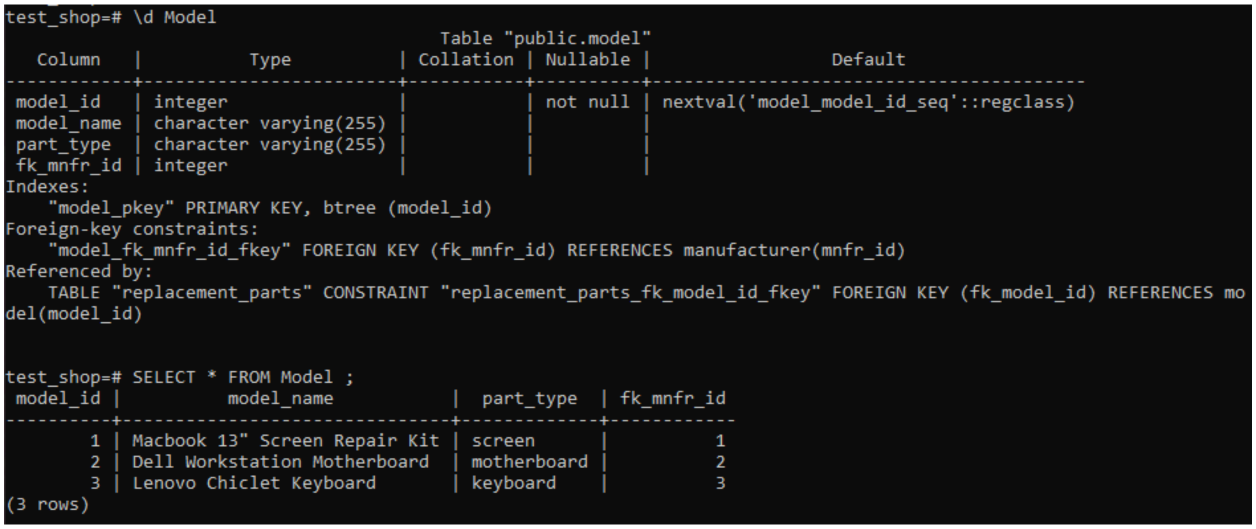
```
test_shop=# \d Employee
                                           Table "public.employee"
      Column       |          Type          | Collation | Nullable |                    Default
-------------------+------------------------+-----------+----------+-----------------------------------------------
 employee_id       | integer                |           | not null | nextval('employee_employee_id_seq'::regclass)
 employee_name     | character varying(255) |           | not null |
 employee_phonenum | character varying(20)  |           | not null |
 e_mail            | character varying(255) |           | not null |
 ssn               | numeric(20,0)          |           | not null |
 fk_addr_id        | integer                |           |          |
Indexes:
    "employee_pkey" PRIMARY KEY, btree (employee_id)
Foreign-key constraints:
    "employee_fk_addr_id_fkey" FOREIGN KEY (fk_addr_id) REFERENCES address(addr_id)
Referenced by:
    TABLE ""Order"" CONSTRAINT "Order_fk_emp_id_fkey" FOREIGN KEY (fk_emp_id) REFERENCES employee(employee_id)
    TABLE "ticket" CONSTRAINT "ticket_fk_employee_id_fkey" FOREIGN KEY (fk_employee_id) REFERENCES employee(employee_
id)
    TABLE "works_on" CONSTRAINT "works_on_fk_employee_id_fkey" FOREIGN KEY (fk_employee_id) REFERENCES employee(emplo
yee_id)


test_shop=# SELECT * FROM Employee ;
 employee_id | employee_name | employee_phonenum |      e_mail      |    ssn    | fk_addr_id
-------------+---------------+-------------------+------------------+-----------+------------
           1 | Drake         | 1-661-234-5678    | drake@gmail.com  | 373554234 |          1
           2 | Josh          | 1-661-788-9999    | josh@gmail.com   | 444472341 |          1
           3 | Megan         | 1-661-097-2345    | megan@gmail.com  | 537123411 |          1
(3 rows)


test_shop=#
```

Manufacturer:



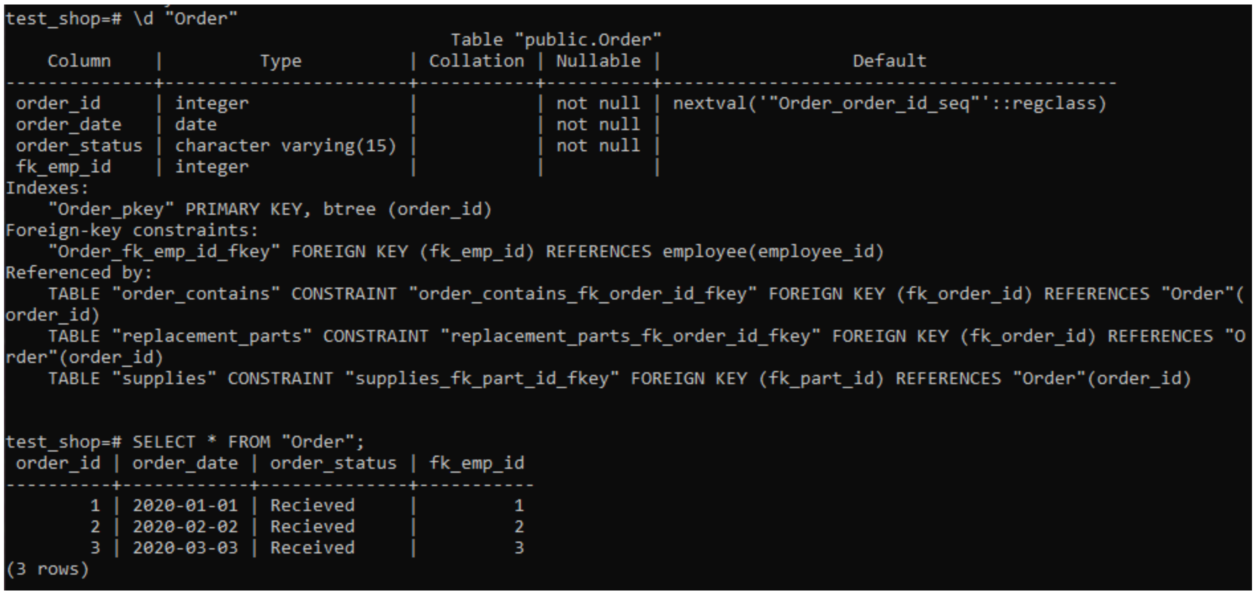
```
test_shop=# \d Manufacturer
                                      Table "public.manufacturer"
    Column     |          Type          | Collation | Nullable |                    Default
---------------+------------------------+-----------+----------+-----------------------------------------------
 mnfr_id       | integer                |           | not null | nextval('manufacturer_mnfr_id_seq'::regclass)
 mnfr_name     | character varying(255) |           | not null |
 mnfr_phonenum | character varying(20)  |           | not null |
 fk_addr_id    | integer                |           |          |
Indexes:
    "manufacturer_pkey" PRIMARY KEY, btree (mnfr_id)
Foreign-key constraints:
    "manufacturer_fk_addr_id_fkey" FOREIGN KEY (fk_addr_id) REFERENCES address(addr_id)
Referenced by:
    TABLE "model" CONSTRAINT "model_fk_mnfr_id_fkey" FOREIGN KEY (fk_mnfr_id) REFERENCES manufacturer(mnfr_id)


test_shop=# SELECT * FROM Manufacturer ;
 mnfr_id | mnfr_name | mnfr_phonenum  | fk_addr_id
---------+-----------+----------------+------------
       1 | Apple     | 1-800-275-2273 |          3
       2 | Dell      | 1-800-99-3355  |          4
       3 | Lenovo    | 1-855-253-6686 |          5
(3 rows)


test_shop=#
```

Model:



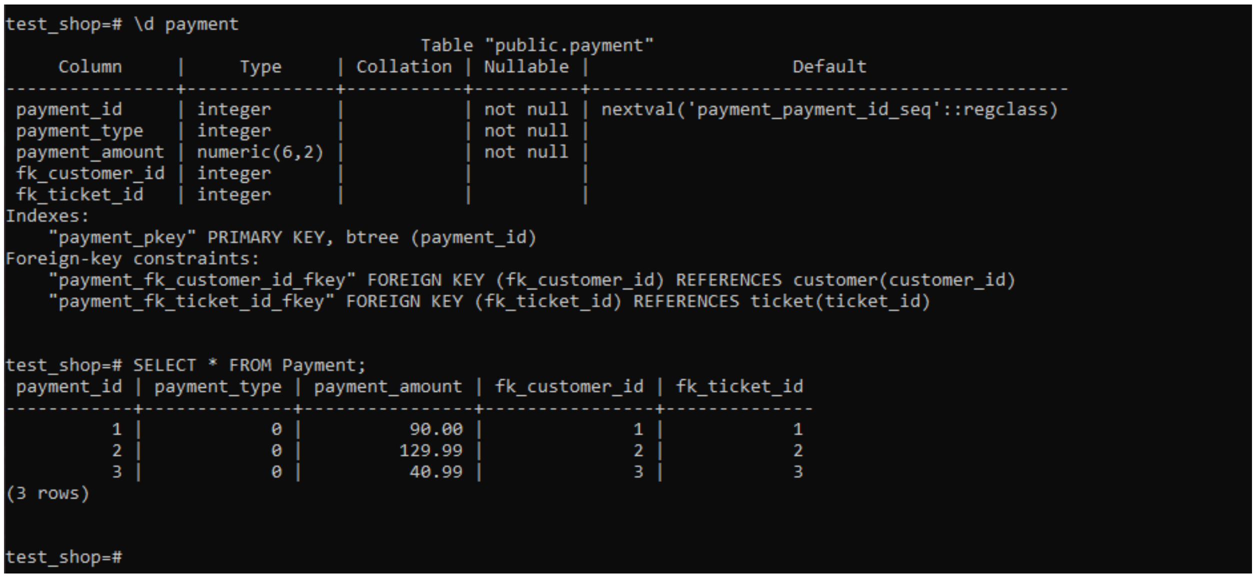
```
test_shop=# \d Model
                                      Table "public.model"
   Column   |          Type          | Collation | Nullable |                 Default
------------+------------------------+-----------+----------+-----------------------------------------
 model_id   | integer                |           | not null | nextval('model_model_id_seq'::regclass)
 model_name | character varying(255) |           |          |
 part_type  | character varying(255) |           |          |
 fk_mnfr_id | integer                |           |          |
Indexes:
    "model_pkey" PRIMARY KEY, btree (model_id)
Foreign-key constraints:
    "model_fk_mnfr_id_fkey" FOREIGN KEY (fk_mnfr_id) REFERENCES manufacturer(mnfr_id)
Referenced by:
    TABLE "replacement_parts" CONSTRAINT "replacement_parts_fk_model_id_fkey" FOREIGN KEY (fk_model_id) REFERENCES mo
del(model_id)


test_shop=# SELECT * FROM Model ;
 model_id |          model_name           |  part_type  | fk_mnfr_id
----------+-------------------------------+-------------+------------
        1 | Macbook 13" Screen Repair Kit | screen      |          1
        2 | Dell Workstation Motherboard  | motherboard |          2
        3 | Lenovo Chiclet Keyboard       | keyboard    |          3
(3 rows)
```

Order:



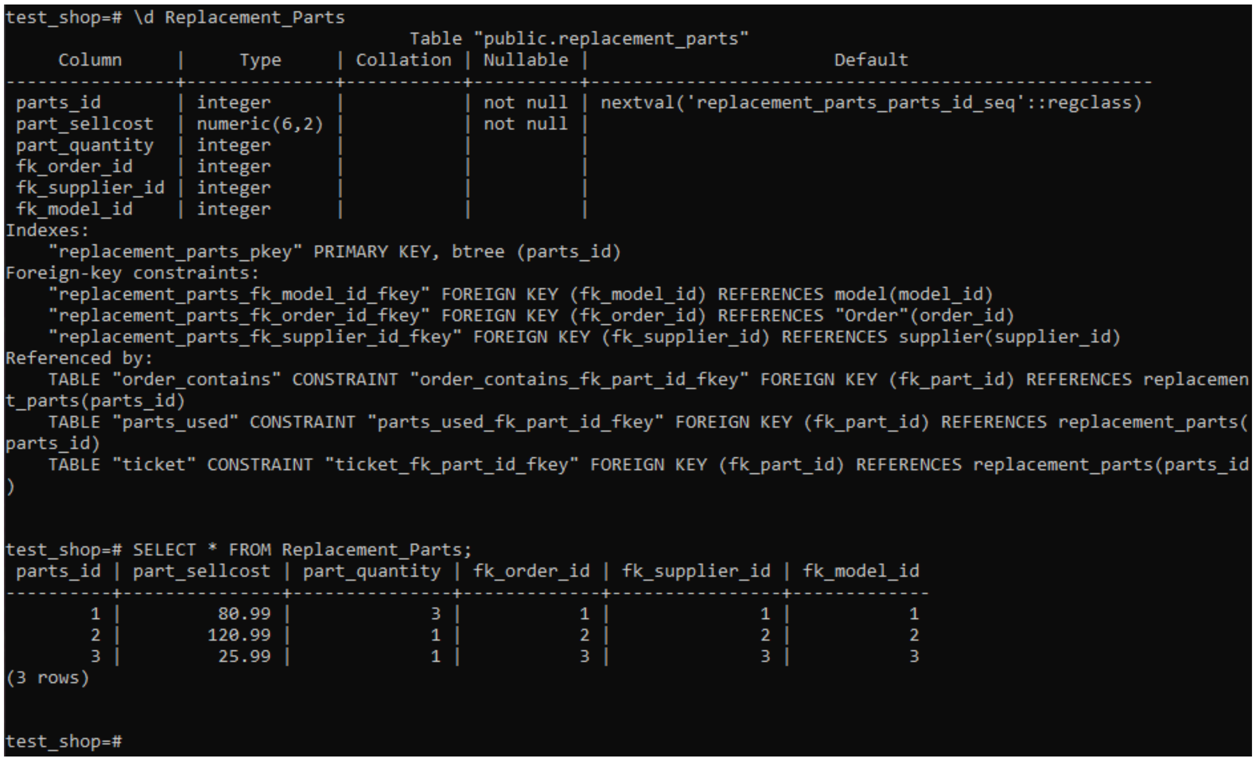
```
test_shop=# \d "Order"
                                         Table "public.Order"
    Column    |         Type          | Collation | Nullable |                  Default
--------------+-----------------------+-----------+----------+--------------------------------------------
 order_id     | integer               |           | not null | nextval('"Order_order_id_seq"'::regclass)
 order_date   | date                  |           | not null |
 order_status | character varying(15) |           | not null |
 fk_emp_id    | integer               |           |          |
Indexes:
    "Order_pkey" PRIMARY KEY, btree (order_id)
Foreign-key constraints:
    "Order_fk_emp_id_fkey" FOREIGN KEY (fk_emp_id) REFERENCES employee(employee_id)
Referenced by:
    TABLE "order_contains" CONSTRAINT "order_contains_fk_order_id_fkey" FOREIGN KEY (fk_order_id) REFERENCES "Order"(
order_id)
    TABLE "replacement_parts" CONSTRAINT "replacement_parts_fk_order_id_fkey" FOREIGN KEY (fk_order_id) REFERENCES "O
rder"(order_id)
    TABLE "supplies" CONSTRAINT "supplies_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES "Order"(order_id)


test_shop=# SELECT * FROM "Order";
 order_id | order_date | order_status | fk_emp_id
----------+------------+--------------+-----------
        1 | 2020-01-01 | Recieved     |         1
        2 | 2020-02-02 | Recieved     |         2
        3 | 2020-03-03 | Received     |         3
(3 rows)
```

Payment:



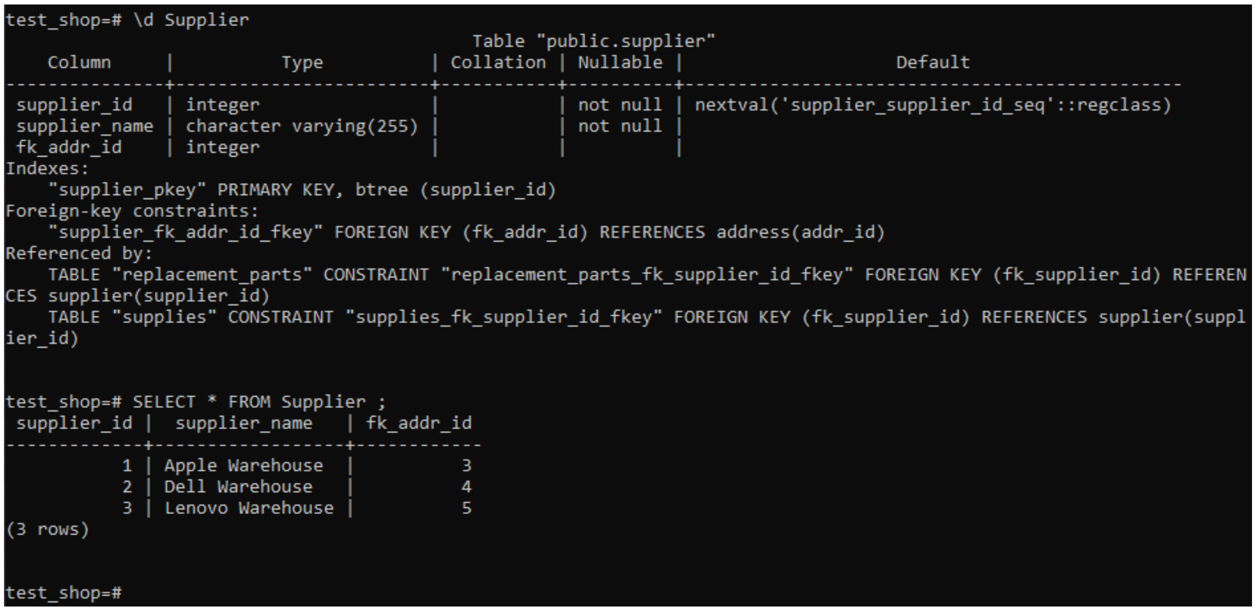
```
test_shop=# \d payment
                                        Table "public.payment"
     Column     |     Type     | Collation | Nullable |                 Default
----------------+--------------+-----------+----------+---------------------------------------------
 payment_id     | integer      |           | not null | nextval('payment_payment_id_seq'::regclass)
 payment_type   | integer      |           | not null |
 payment_amount | numeric(6,2) |           | not null |
 fk_customer_id | integer      |           |          |
 fk_ticket_id   | integer      |           |          |
Indexes:
    "payment_pkey" PRIMARY KEY, btree (payment_id)
Foreign-key constraints:
    "payment_fk_customer_id_fkey" FOREIGN KEY (fk_customer_id) REFERENCES customer(customer_id)
    "payment_fk_ticket_id_fkey" FOREIGN KEY (fk_ticket_id) REFERENCES ticket(ticket_id)


test_shop=# SELECT * FROM Payment;
 payment_id | payment_type | payment_amount | fk_customer_id | fk_ticket_id
------------+--------------+----------------+----------------+--------------
          1 |            0 |          90.00 |              1 |            1
          2 |            0 |         129.99 |              2 |            2
          3 |            0 |          40.99 |              3 |            3
(3 rows)


test_shop=#
```

Replacement_Parts:



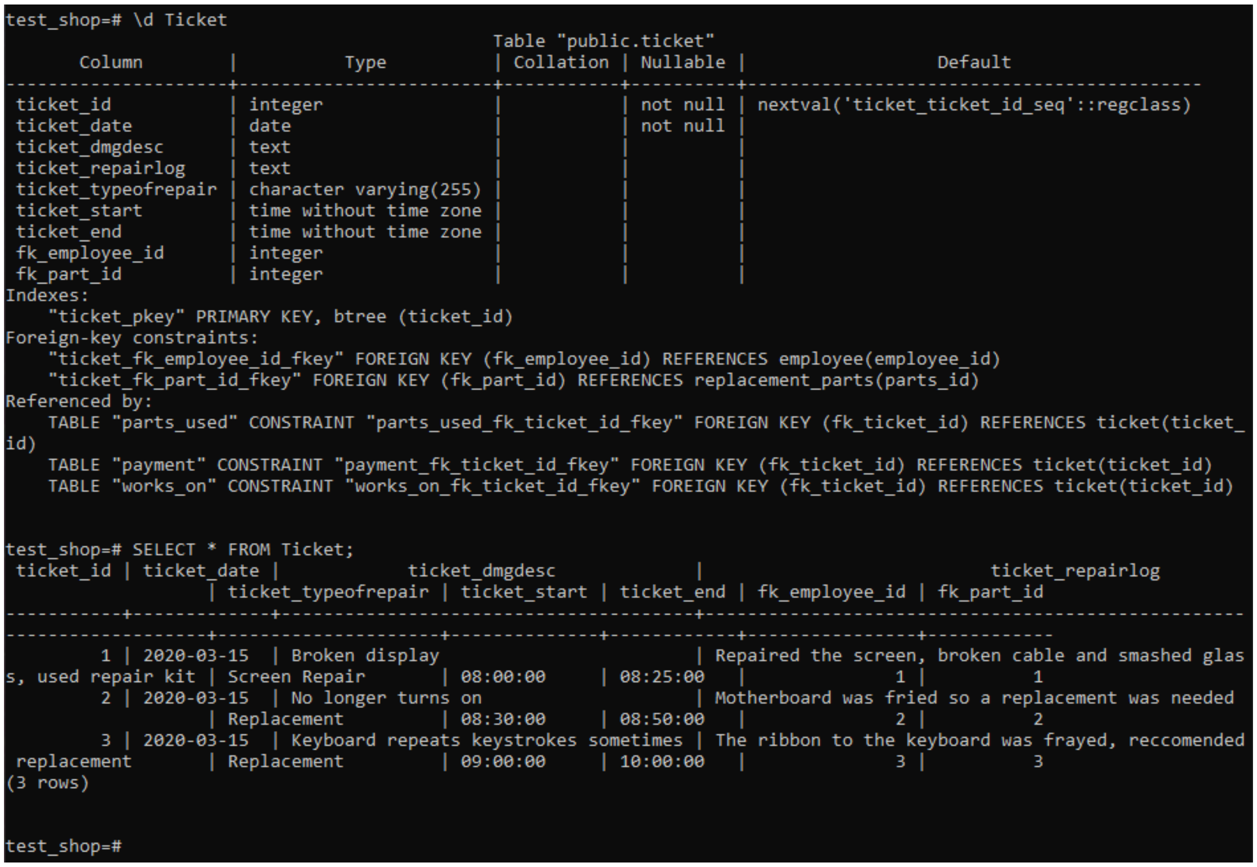
```
test_shop=# \d Replacement_Parts
                                        Table "public.replacement_parts"
     Column     |     Type     | Collation | Nullable |                       Default
----------------+--------------+-----------+----------+-----------------------------------------------------
 parts_id       | integer      |           | not null | nextval('replacement_parts_parts_id_seq'::regclass)
 part_sellcost  | numeric(6,2) |           | not null |
 part_quantity  | integer      |           |          |
 fk_order_id    | integer      |           |          |
 fk_supplier_id | integer      |           |          |
 fk_model_id    | integer      |           |          |
Indexes:
    "replacement_parts_pkey" PRIMARY KEY, btree (parts_id)
Foreign-key constraints:
    "replacement_parts_fk_model_id_fkey" FOREIGN KEY (fk_model_id) REFERENCES model(model_id)
    "replacement_parts_fk_order_id_fkey" FOREIGN KEY (fk_order_id) REFERENCES "Order"(order_id)
    "replacement_parts_fk_supplier_id_fkey" FOREIGN KEY (fk_supplier_id) REFERENCES supplier(supplier_id)
Referenced by:
    TABLE "order_contains" CONSTRAINT "order_contains_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES replacemen
t_parts(parts_id)
    TABLE "parts_used" CONSTRAINT "parts_used_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES replacement_parts(
parts_id)
    TABLE "ticket" CONSTRAINT "ticket_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES replacement_parts(parts_id
)


test_shop=# SELECT * FROM Replacement_Parts;
 parts_id | part_sellcost | part_quantity | fk_order_id | fk_supplier_id | fk_model_id
----------+---------------+---------------+-------------+----------------+-------------
        1 |         80.99 |             3 |           1 |              1 |           1
        2 |        120.99 |             1 |           2 |              2 |           2
        3 |         25.99 |             1 |           3 |              3 |           3
(3 rows)


test_shop=#
```

Supplier:



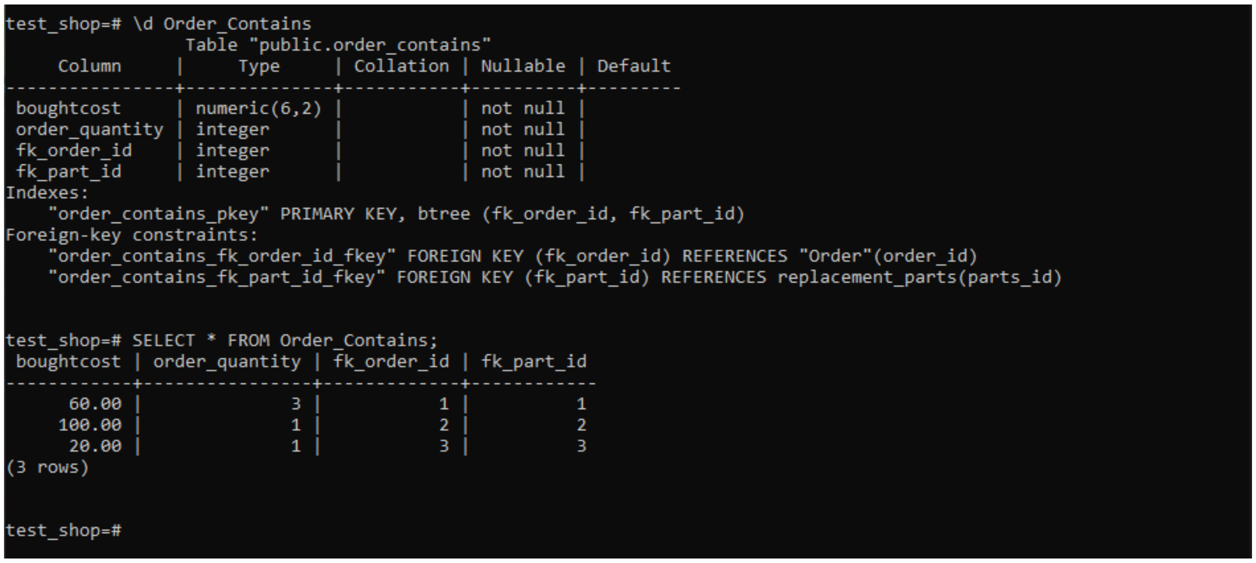
```
test_shop=# \d Supplier
                                              Table "public.supplier"
    Column     |          Type          | Collation | Nullable |                  Default
---------------+------------------------+-----------+----------+-----------------------------------------------
 supplier_id   | integer                |           | not null | nextval('supplier_supplier_id_seq'::regclass)
 supplier_name | character varying(255) |           | not null |
 fk_addr_id    | integer                |           |          |
Indexes:
    "supplier_pkey" PRIMARY KEY, btree (supplier_id)
Foreign-key constraints:
    "supplier_fk_addr_id_fkey" FOREIGN KEY (fk_addr_id) REFERENCES address(addr_id)
Referenced by:
    TABLE "replacement_parts" CONSTRAINT "replacement_parts_fk_supplier_id_fkey" FOREIGN KEY (fk_supplier_id) REFEREN
CES supplier(supplier_id)
    TABLE "supplies" CONSTRAINT "supplies_fk_supplier_id_fkey" FOREIGN KEY (fk_supplier_id) REFERENCES supplier(suppl
ier_id)


test_shop=# SELECT * FROM Supplier ;
 supplier_id |  supplier_name   | fk_addr_id
-------------+------------------+------------
           1 | Apple Warehouse  |          3
           2 | Dell Warehouse   |          4
           3 | Lenovo Warehouse |          5
(3 rows)


test_shop=#
```

Ticket (Repair_Request) :



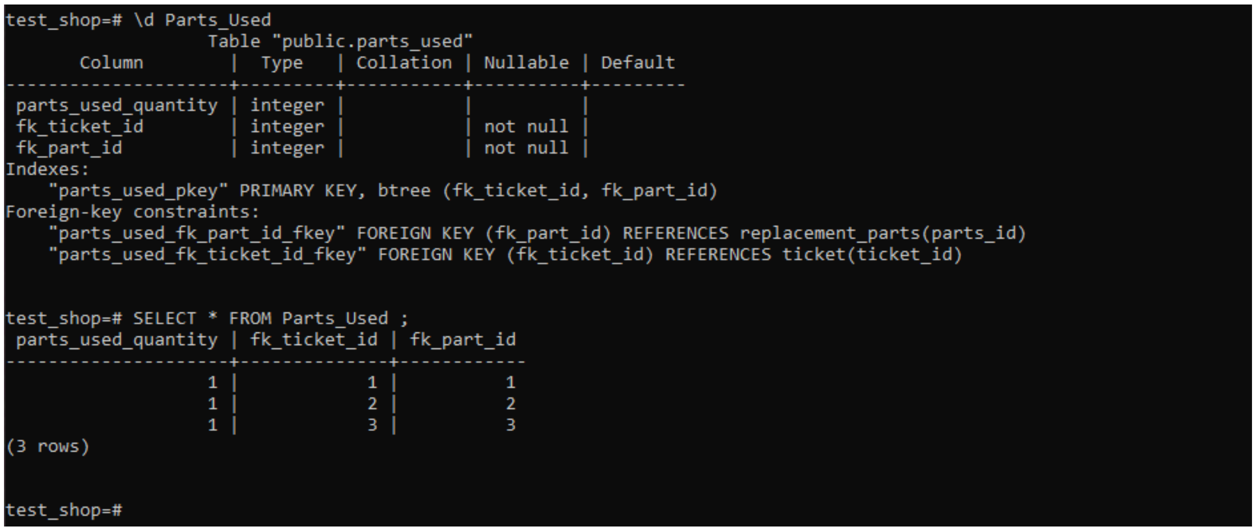
```
test_shop=# \d Ticket
                                              Table "public.ticket"
       Column        |          Type          | Collation | Nullable |                  Default
---------------------+------------------------+-----------+----------+-------------------------------------------
 ticket_id           | integer                |           | not null | nextval('ticket_ticket_id_seq'::regclass)
 ticket_date         | date                   |           | not null |
 ticket_dmgdesc      | text                   |           |          |
 ticket_repairlog    | text                   |           |          |
 ticket_typeofrepair | character varying(255) |           |          |
 ticket_start        | time without time zone |           |          |
 ticket_end          | time without time zone |           |          |
 fk_employee_id      | integer                |           |          |
 fk_part_id          | integer                |           |          |
Indexes:
    "ticket_pkey" PRIMARY KEY, btree (ticket_id)
Foreign-key constraints:
    "ticket_fk_employee_id_fkey" FOREIGN KEY (fk_employee_id) REFERENCES employee(employee_id)
    "ticket_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES replacement_parts(parts_id)
Referenced by:
    TABLE "parts_used" CONSTRAINT "parts_used_fk_ticket_id_fkey" FOREIGN KEY (fk_ticket_id) REFERENCES ticket(ticket_
id)
    TABLE "payment" CONSTRAINT "payment_fk_ticket_id_fkey" FOREIGN KEY (fk_ticket_id) REFERENCES ticket(ticket_id)
    TABLE "works_on" CONSTRAINT "works_on_fk_ticket_id_fkey" FOREIGN KEY (fk_ticket_id) REFERENCES ticket(ticket_id)


test_shop=# SELECT * FROM Ticket;
 ticket_id | ticket_date |            ticket_dmgdesc             |                           ticket_repairlog
                   | ticket_typeofrepair | ticket_start | ticket_end | fk_employee_id | fk_part_id
-----------+-------------+---------------------------------------+-----------------------------------------------
-------------------+---------------------+--------------+------------+----------------+------------
         1 | 2020-03-15  | Broken display                        | Repaired the screen, broken cable and smashed glas
s, used repair kit | Screen Repair       | 08:00:00     | 08:25:00   |              1 |          1
         2 | 2020-03-15  | No longer turns on                    | Motherboard was fried so a replacement was needed
                   | Replacement         | 08:30:00     | 08:50:00   |              2 |          2
         3 | 2020-03-15  | Keyboard repeats keystrokes sometimes | The ribbon to the keyboard was frayed, reccomended
 replacement       | Replacement         | 09:00:00     | 10:00:00   |              3 |          3
(3 rows)


test_shop=#
```

Order_Contains:



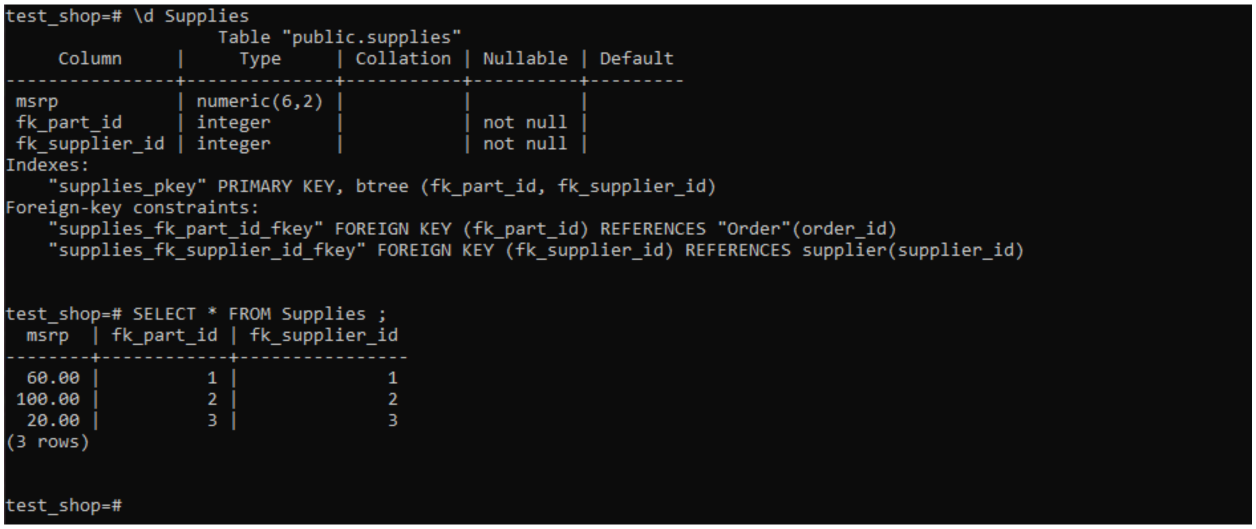
```
test_shop=# \d Order_Contains
                 Table "public.order_contains"
     Column     |     Type     | Collation | Nullable | Default
----------------+--------------+-----------+----------+---------
 boughtcost     | numeric(6,2) |           | not null |
 order_quantity | integer      |           | not null |
 fk_order_id    | integer      |           | not null |
 fk_part_id     | integer      |           | not null |
Indexes:
    "order_contains_pkey" PRIMARY KEY, btree (fk_order_id, fk_part_id)
Foreign-key constraints:
    "order_contains_fk_order_id_fkey" FOREIGN KEY (fk_order_id) REFERENCES "Order"(order_id)
    "order_contains_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES replacement_parts(parts_id)


test_shop=# SELECT * FROM Order_Contains;
 boughtcost | order_quantity | fk_order_id | fk_part_id
------------+----------------+-------------+------------
      60.00 |              3 |           1 |          1
     100.00 |              1 |           2 |          2
      20.00 |              1 |           3 |          3
(3 rows)


test_shop=#
```

Parts_Used:



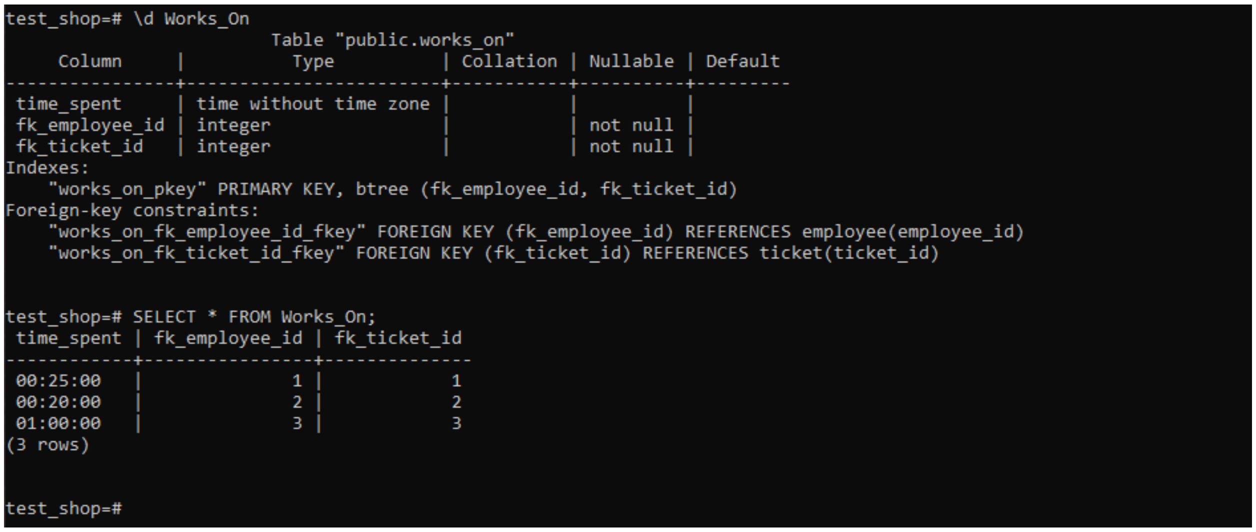
```
test_shop=# \d Parts_Used
                   Table "public.parts_used"
       Column       |  Type   | Collation | Nullable | Default
--------------------+---------+-----------+----------+---------
 parts_used_quantity | integer |           |          |
 fk_ticket_id        | integer |           | not null |
 fk_part_id          | integer |           | not null |
Indexes:
    "parts_used_pkey" PRIMARY KEY, btree (fk_ticket_id, fk_part_id)
Foreign-key constraints:
    "parts_used_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES replacement_parts(parts_id)
    "parts_used_fk_ticket_id_fkey" FOREIGN KEY (fk_ticket_id) REFERENCES ticket(ticket_id)


test_shop=# SELECT * FROM Parts_Used ;
 parts_used_quantity | fk_ticket_id | fk_part_id
---------------------+--------------+------------
                   1 |            1 |          1
                   1 |            2 |          2
                   1 |            3 |          3
(3 rows)


test_shop=#
```

Supplies:



```
test_shop=# \d Supplies
                    Table "public.supplies"
     Column     |     Type     | Collation | Nullable | Default
----------------+--------------+-----------+----------+---------
 msrp           | numeric(6,2) |           |          |
 fk_part_id     | integer      |           | not null |
 fk_supplier_id | integer      |           | not null |
Indexes:
    "supplies_pkey" PRIMARY KEY, btree (fk_part_id, fk_supplier_id)
Foreign-key constraints:
    "supplies_fk_part_id_fkey" FOREIGN KEY (fk_part_id) REFERENCES "Order"(order_id)
    "supplies_fk_supplier_id_fkey" FOREIGN KEY (fk_supplier_id) REFERENCES supplier(supplier_id)


test_shop=# SELECT * FROM Supplies ;
  msrp  | fk_part_id | fk_supplier_id
--------+------------+----------------
  60.00 |          1 |              1
 100.00 |          2 |              2
  20.00 |          3 |              3
(3 rows)


test_shop=#
```

Works_On:


```
test_shop=# \d Works_On
                        Table "public.works_on"
    Column     |          Type          | Collation | Nullable | Default
---------------+------------------------+-----------+----------+---------
 time_spent    | time without time zone |           |          |
 fk_employee_id | integer                |           | not null |
 fk_ticket_id  | integer                |           | not null |
Indexes:
    "works_on_pkey" PRIMARY KEY, btree (fk_employee_id, fk_ticket_id)
Foreign-key constraints:
    "works_on_fk_employee_id_fkey" FOREIGN KEY (fk_employee_id) REFERENCES employee(employee_id)
    "works_on_fk_ticket_id_fkey" FOREIGN KEY (fk_ticket_id) REFERENCES ticket(ticket_id)


test_shop=# SELECT * FROM Works_On;
 time_spent | fk_employee_id | fk_ticket_id
------------+----------------+--------------
 00:25:00   |              1 |            1
 00:20:00   |              2 |            2
 01:00:00   |              3 |            3
(3 rows)


test_shop=#
```

## 3.5 Queries - SQL Variations

## Phase 4: DBMS Procedural Language & Stored Procedures and Triggers

PL/PostgreSQL is a specific procedural language based on SQL. It contains variables, loops, constants. Some SQL can be used in PL/PgSQL.

## 4.1 Postgres PL/ PostgreSQL

Section 4.1 of this section will introduce the concept of PL/PSQL ( Procedural Language / PostgreSQL). It will give an introduction of how PL/PSQL came to be followed by the advantages of it. Afterwards there will be an exploration of the control statement and their syntax.

### 4.1.1 Introduction to PL/PgSQL

PL/PgSQL was made around 1995. It is Oracles Corporatio's procedural extension for SQL. It is available in multiple databases like Oracle Database, Times Ten in-memory database and many more. PL/PgSQL is a loadable procedural language for the PostgreSQL database system. It was created so it can be used to create functions and trigger procedures. It can perform complex computations and adds control structures to the SQL language. PL/PgSQL includes conditions, loops, variables, constants like other languages.

Functions, as mentioned previously, are used to computing and returning a scalar value or an array of them. Procedures are similar to functions where they can be called on multiple times. The only difference between them is that functions can be used in a SQL statement and procedures cannot. In order to make a procedure you will need to include a heading to hold the name of the procedure and a parameter list. Next you will need to add a declarative, executive and exception-handling parts. If you link multiple functions, procedures, and variables then you get a package. Packages allow you to reuse code so you do not have to write them over again.

There are five major data types in PL/PgSQL. They are char, varchar2, date, number, and timestamp. Character variables defines a character, Varchar2 is usually appended to the name definition. Date variables are used to contain the date and time. It cannot just hold the time. A numeric variable is used to hold a number, it can be from a decimal to a really small int. You can also include exceptions, which are an event that disrupts the normal flow of the program's instructions.

### 4.1.2 Advantages of PL/PgSQL

PL/pgSQL has many advantages to it that user's can take advantage of. Some of these advantages include that it's portable and easy to learn. Another of these advantages include that you can group together a block of computations and a series of queries all inside of the database server which gives the advantage of saving communication overhead of clients/servers. This means that the excess round trips between clients and servers are eliminated. With PL/pgSQL you can use the data types, operators and functions of SQL, meaning users can comfortably start using pl/pgSQL if they already have prior knowledge of using other database languages.

There are two kinds of blocks: Anonymous blocks(Do) and Named blocks(Functions and Stored Procedures). Anonymous blocks are generally constructed dynamically and executed only once by the user. Named blocks are like what the name says, they are blocks that have names that are associated with them. They are stored in the database and can be executed multiple times and accept parameters.

Due to its size, PL/PgSQL should be more efficient compared to the other languages. If your procedure has a lot of SQL statements then PL/PgSQL should be a lot shorter and cleaner for someone to read. Another advantage is parameters can have a default value so they are essentially optional.

### 4.1.3 Control Statements and Syntax.

PL/pgSQL has three different types of control statements to choose from, these control statements include conditional selection statements, loop statements, and sequential control statements.

CONDITIONAL SELECTION STATEMENTS

IF-THEN/ IF-THEN-ELSE :

```
IF boolean-expression THEN
  statements
END IF;

IF boolean-expression THEN
  statements
ELSE
  statements
END IF;
```

IF-THEN-ElSIF :

```
IF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements
[ ELSIF boolean-expression THEN
  statements
  ...]]
[ ELSE
  statements ]
```

END IF;

LOOP STATEMENTS

FOR (INT VARIANT) :

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

FOR (LOOP THROUGH QUERY RESULTS) :

```
[ <<label>> ]
FOR target IN query LOOP
    statements
END LOOP [ label ];
```

LOOP :

```
[ <<label>> ]
LOOP
    statements
END LOOP [ label ];
```

WHILE-LOOP :

```
[ <<label>> ]
WHILE boolean-expression LOOP
    statements
END LOOP [ label ];
```

ASSIGNMENT STATEMENTS

There isn't a formal keyword for assignment values to a PL/pgsql variable, but there is a proper syntax for it.

syntax:
*variable* { := | = } *expression* ;

### 4.1.4 PL/SQL Syntax of View, Function, Procedure and Trigger

<u>VIEW</u>

Creating:

```
CREATE VIEW            view_name       AS       query ;
```

Changing:

```
CREATE OR REPLACE          view_name       AS       query
ALTER VIEW             view_name       RENAME TO   new_view_name  ;
```

Removing:

```
DROP VIEW [ IF EXISTS ]      view_name  ;
```

<u>FUNCTION</u>

Creating:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
RETURN return_datatype
{IS | AS}
BEGIN
  < function_body >
END [function_name];
```

Dropping:

```
DROP FUNCTION function_name;
```

<u>PROCEDURE</u>

Creating:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter_name [IN | OUT | IN OUT] type [, ...])]
{IS | AS}
BEGIN
  < procedure_body >
```

END *procedure_name*;
Executing:
EXECUTE *procedure_name*;

TRIGGER

Simple syntax:

```
CREATE FUNCTION        trigger_function ()
        RETURN      trigger        AS
```

Basic syntax:

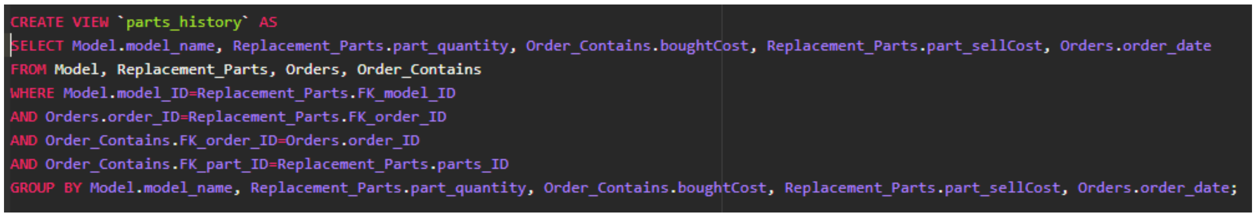
```
CREATE TRIGGER    trigger_name
{ BEFORE | AFTER | INSTEAD OF} {event  [OR . . . ] }
        ON      table_name
        [ FOR [ EACH ]  { ROW | STATEMENT } ]
                EXECUTE PROCEDURE      trigger_function
```

## 4.2 Views and Stored Subprogram for your Database

This section of phase 04 contains the code and output of the views, stored procedures/function and triggers that were created for this database.

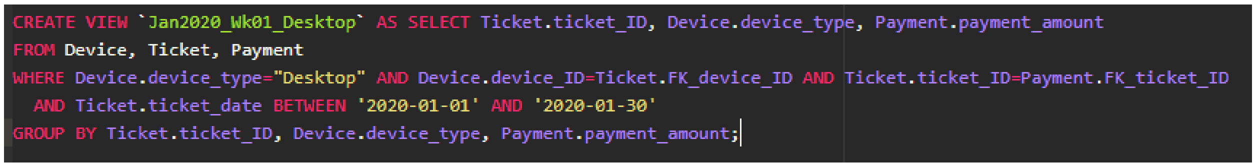
### 4.2.1 Views

This view is to be able to see what type of parts are being gotten from an oder. These tables include the sellCost where we seel it to the public for how much we want, and it also displays the boughtCost where we can see how much we paid for that specific parts in the order.



```sql
CREATE VIEW `parts_history` AS
SELECT Model.model_name, Replacement_Parts.part_quantity, Order_Contains.boughtCost, Replacement_Parts.part_sellCost, Orders.order_date
FROM Model, Replacement_Parts, Orders, Order_Contains
WHERE Model.model_ID=Replacement_Parts.FK_model_ID
AND Orders.order_ID=Replacement_Parts.FK_order_ID
AND Order_Contains.FK_order_ID=Orders.order_ID
AND Order_Contains.FK_part_ID=Replacement_Parts.parts_ID
GROUP BY Model.model_name, Replacement_Parts.part_quantity, Order_Contains.boughtCost, Replacement_Parts.part_sellCost, Orders.order_date;
```

This view is meant to show all the amount of the parts where each part must be a device_type of desktop and they must have been ordered in the month of January. `



```sql
CREATE VIEW `Jan2020_Wk01_Desktop` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Desktop" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;
```

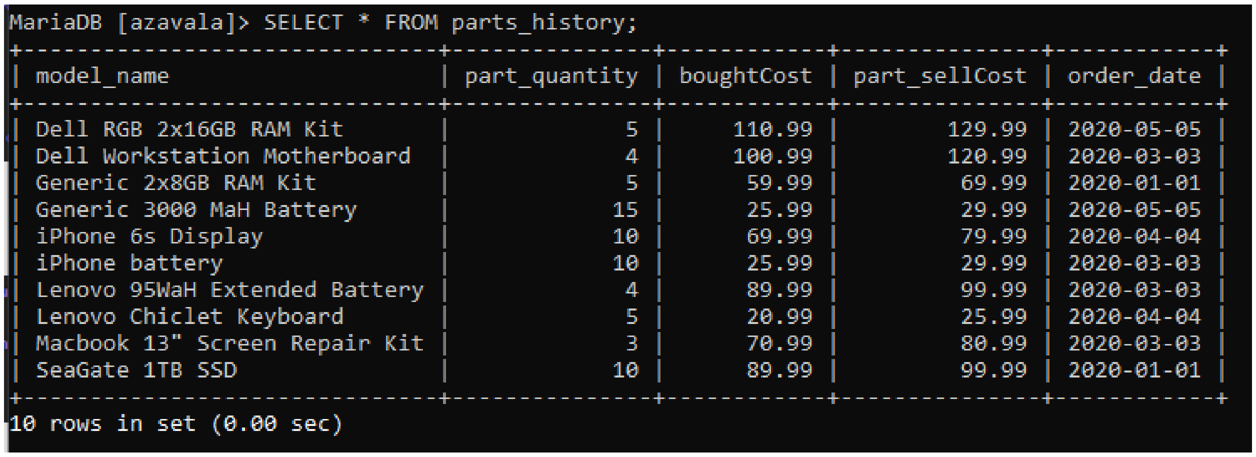
### 4.2.2 Stored Procedures and/or Functions

No functions or stored procedures were made.

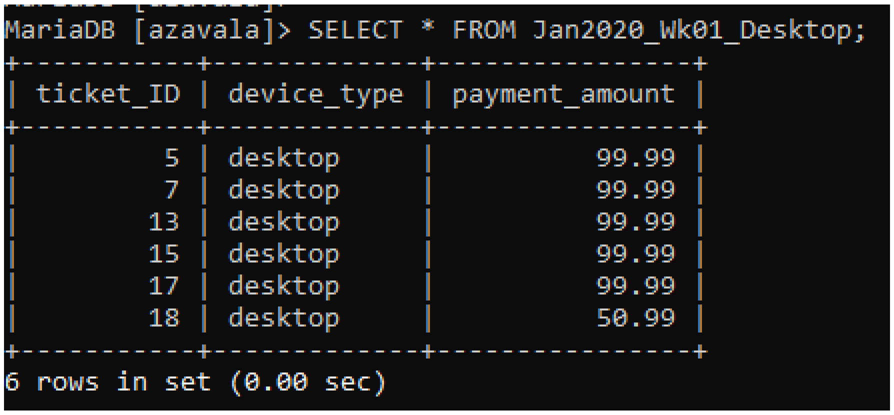
### 4.2.3 Testing Results of Views, Functions and/or Procedures

Views:

Output of the “parts_history” view.


```
MariaDB [azavala]> SELECT * FROM parts_history;
+-------------------------------+---------------+------------+---------------+------------+
| model_name                    | part_quantity | boughtCost | part_sellCost | order_date |
+-------------------------------+---------------+------------+---------------+------------+
| Dell RGB 2x16GB RAM Kit       |             5 |     110.99 |        129.99 | 2020-05-05 |
| Dell Workstation Motherboard  |             4 |     100.99 |        120.99 | 2020-03-03 |
| Generic 2x8GB RAM Kit         |             5 |      59.99 |         69.99 | 2020-01-01 |
| Generic 3000 MaH Battery      |            15 |      25.99 |         29.99 | 2020-05-05 |
| iPhone 6s Display             |            10 |      69.99 |         79.99 | 2020-04-04 |
| iPhone battery                |            10 |      25.99 |         29.99 | 2020-03-03 |
| Lenovo 95WaH Extended Battery |             4 |      89.99 |         99.99 | 2020-03-03 |
| Lenovo Chiclet Keyboard       |             5 |      20.99 |         25.99 | 2020-04-04 |
| Macbook 13" Screen Repair Kit |             3 |      70.99 |         80.99 | 2020-03-03 |
| SeaGate 1TB SSD               |            10 |      89.99 |         99.99 | 2020-01-01 |
+-------------------------------+---------------+------------+---------------+------------+
10 rows in set (0.00 sec)
```

Output of the “Jan2020_Wk01_Desktop” view.


```
MariaDB [azavala]> SELECT * FROM Jan2020_Wk01_Desktop;
+-----------+-------------+----------------+
| ticket_ID | device_type | payment_amount |
+-----------+-------------+----------------+
|         5 | desktop     |          99.99 |
|         7 | desktop     |          99.99 |
|        13 | desktop     |          99.99 |
|        15 | desktop     |          99.99 |
|        17 | desktop     |          99.99 |
|        18 | desktop     |          50.99 |
+-----------+-------------+----------------+
6 rows in set (0.00 sec)
```

## 4.3 Syntax of Stored Function and Procedures and Triggers of Three DBMS (Microsoft SQL and MySQL and Oracle)

Section 4.3 will go into detail about T/SQL, PL/SQL, and My/SQL. T/SQL is Microsoft’s extension to SQL that is used to interact with a database. My/SQL is a free and open source software that allows users to manage a database system. PL/SQL is a procedural language used to design/build applications. All these have their pros and cons, it is up to the user on what they want to decide which one fits their needs the most.

### 4.3.1 The differences between the three languagesMicrosoft  SQL,MySQL, Oracle PL\SQL

When it comes to the licenses of the databases, MySQL has an open-source development model where users who distribute applications with MySQL code must have it available through open-source under the GPL and if they can't then they can get a commercial license. Meanwhile, Oracle DB is a completely proprietary database.

When it comes to programming languages that are supported , they all share a multitude of programming languages. However, Oracle SQL supports languages that aren't supported by the other two DBMS, these languages are Scala, Fortran and other languages. Oracle SQL also includes packages to help users avoid conflicts and it also gives users the utilization of 'for-loops'.

When it comes to the case of  MySQL, they don't offer the functionality of 'for-loops'. Neither the functionality of packages for namespace management while creating procedures. When creating procedures in MySQL, the delimiter commander must be utilized while creating procedures to change the semicolon to '//'. If this isn't done then the procedure will only be what the first line of code contained.

The Microsoft T/SQL has options available for encrypting the text of procedures and they also allow the user to restrict the user permissions for those texts. Inside of procedures they allow for multiple try-catch blocks.

### 4.3.2 The similarities between Microsoft T\SQL, Oracle PL\SQL, and MY\SQL

The three database management systems, Microsoft Transact SQL, PL/SQL and MySQL, support the XML language and secondary indexes. Just like how Oracle supports many different programming languages that the other two DBMS don't, there are languages that all three DBMS have in common. These languages include Delphi, Lisp, Java, C++, C#, Ruby, PHP, Visual Basic and Javascript.

Meanwhile, some of the functionality that all of them share amongst themselves includes triggers, a type of SQL language for querying information from the database and several different API's. They also support concurrency, durability and foreign key concepts.

Now, when it comes to T/SQL and Oracle SQL, they seem to have the functionality to return tables and scalar values from their own functions. However, when it comes to returning tables and scalar values with Oracle SQL, it isn't done so easily. One other similarity that two DBMS share in common come from MySQL and Oracle SQL, they share the same parameter-passing technique.

### 4.3.3 Syntax of Stored Functions and/or Procedures of the three DBMSs

Microsoft SQL (Transact-SQL)

(Stored Procedure)
Creating:
CREATE [ OR ALTER ] { PROC | PROCEDURE }

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

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

Dropping:

```
DROP { PROC | PROCEDURE } { [ schema_name. ] procedure_name }
```

MySQL

(Stored Procedure)

Creating:

```
CREATE [DEFINER = { user | CURRENT_USER }]
PROCEDURE sp_name ([proc_parameter[,...]])
[characteristic ...] routine_body
proc_parameter: [ IN | OUT | INOUT ] param_name type
type:
Any valid MySQL data type
characteristic:
COMMENT 'string'
| LANGUAGE SQL
| [NOT] DETERMINISTIC
| { CONTAINS SQL | NO SQL | READS SQL DATA
| MODIFIES SQL DATA }
| SQL SECURITY { DEFINER | INVOKER }
routine_body:
Valid SQL routine statement
```

Dropping:

```
DROP {PROCEDURE | FUNCTION} [IF EXISTS] sp_name
```

(Stored Function)
Creating:

```
CREATE FUNCTION function_name(func_parameter1, func_parameter2, ..)
        RETURN datatype [characteristics]
        func_body
```

Dropping:
DROP FUNCTION *function_name* ;

Oracle SQL

(Stored Procedure)
Creating:
CREATE OR REPLACE *procedure_name*(arg1 data_type, ...) AS
BEGIN
  ....
END *procedure_name* ;

Dropping:
DROP PROCEDURE *procedure_name* ;

(Stored Function)
Creating:
CREATE OR REPLACE *function_name*(arg1 data_type, ...) AS
BEGIN
  ....
END *function_name*;

Dropping:
DROP FUNCTION *function_name* ;

# Phase V:  Graphic User Interface for XYZ Users

## 5.1 Functionalities and User group of the GUI Application.

Here we describe the type of people who are going to be using this database project GUI application and what purpose the GUI application will serve in order to help the employee do a better job or to make their daily tasks more manageable and efficient.

### 5.1.1 Itemized Description of Application

Designing, developing and tweaking a database can be quite difficult and time-consuming to make sure everything is done right. Throughout that time it is important to remember who the database is going to be used by and why the database is being built. Who is going to use this database; employees, managers, clients or customers ? It needs to be able to handle a multitude of users, some who are more tech-savvy than others, but with an interface that is still easily understandable and efficient.

Employees
==========================================================

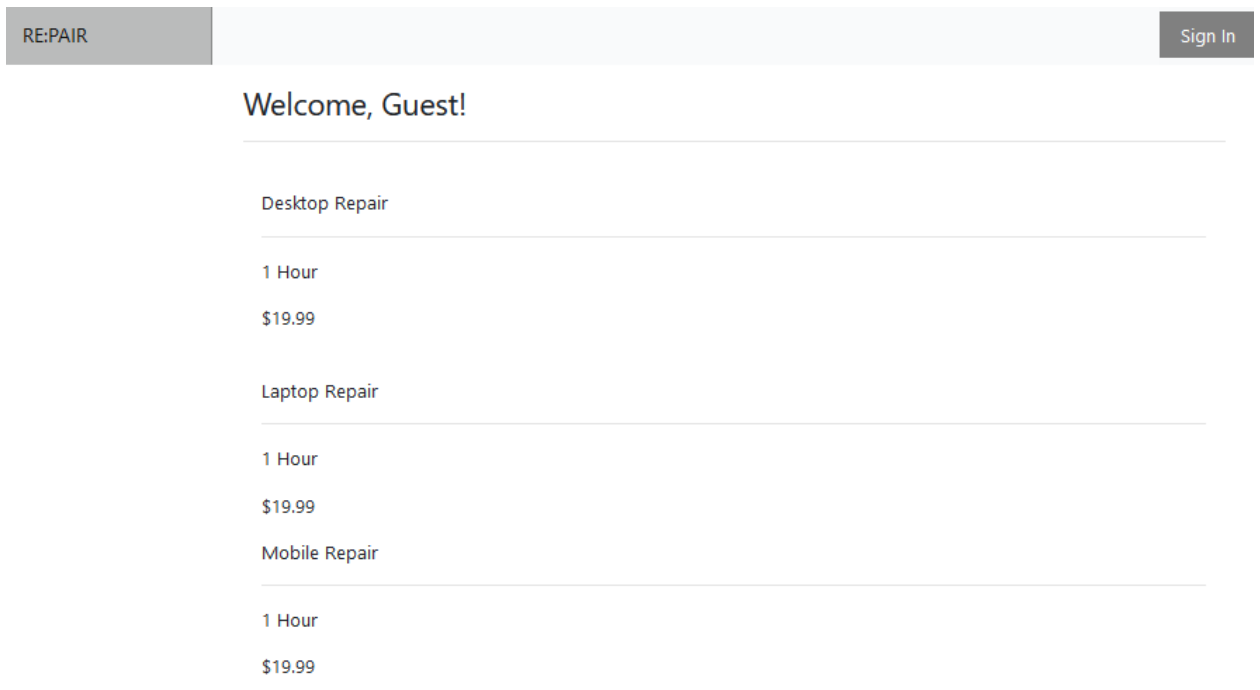
For this database, it was important for the employee to be able to enter new tickets and view what devices needed to be repaired for that day. So the interface had to be simple and intuitive.

Required Functions :

- Repair queue
- Ticket Form
- Report Generator

### 5.1.2  GUI Application Screenshots

**Home**



Welcome, Guest!

Desktop Repair

1 Hour

$19.99

Laptop Repair

1 Hour

$19.99

Mobile Repair

1 Hour

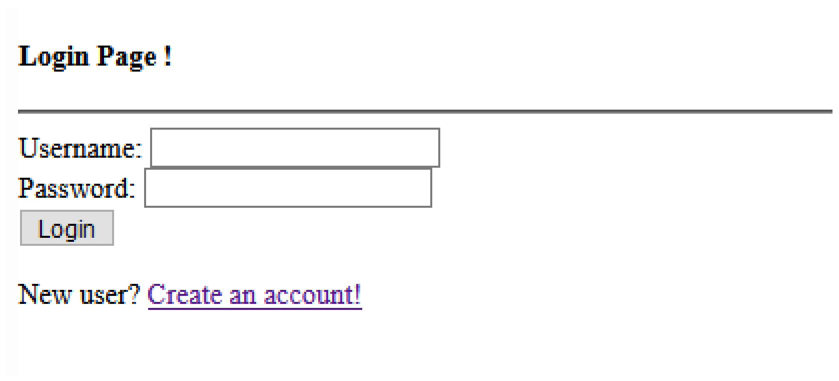
$19.99

Functionality:

Top Left - Logo of repair shop
Top Right - Sign in button where employees can log in

The home page presented to the user is just a quick home page where they have the option to log into the dashboard for employees and displays the price per hour of the repair they want.

**Login**



**Login Page !**

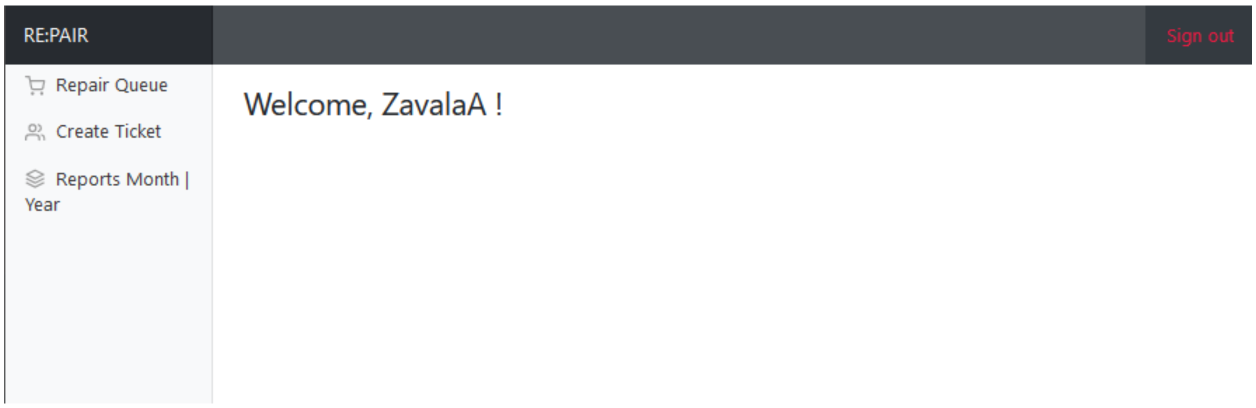
Username:
Password:

New user? Create an account!

Functionality :

The functionality in this page is a basic login where the employee logs in with their username and password.
The purple hyperlink in the image above is for the user to create a new account.

## Logged In

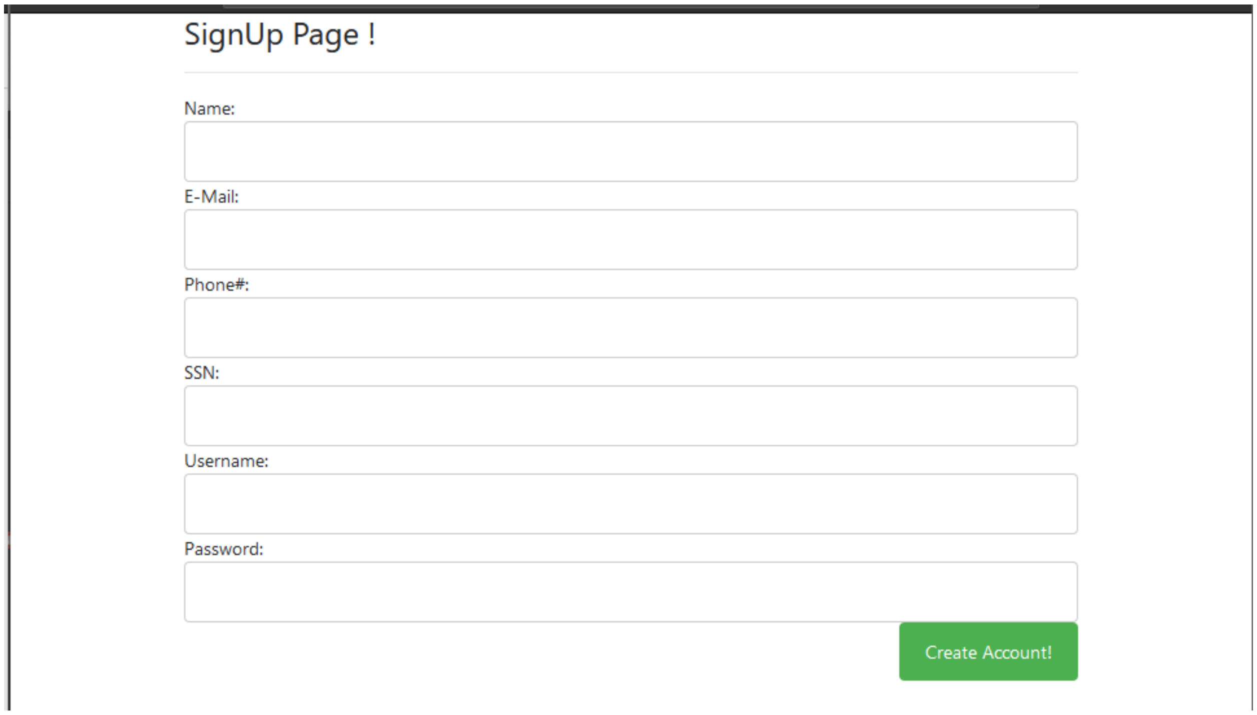


Welcome, ZavalaA !

Functionality:

The home landing page of the dashboard once the employee logs in.

## Sign Up


SignUp Page !

Name:

E-Mail:

Phone#:

SSN:

Username:

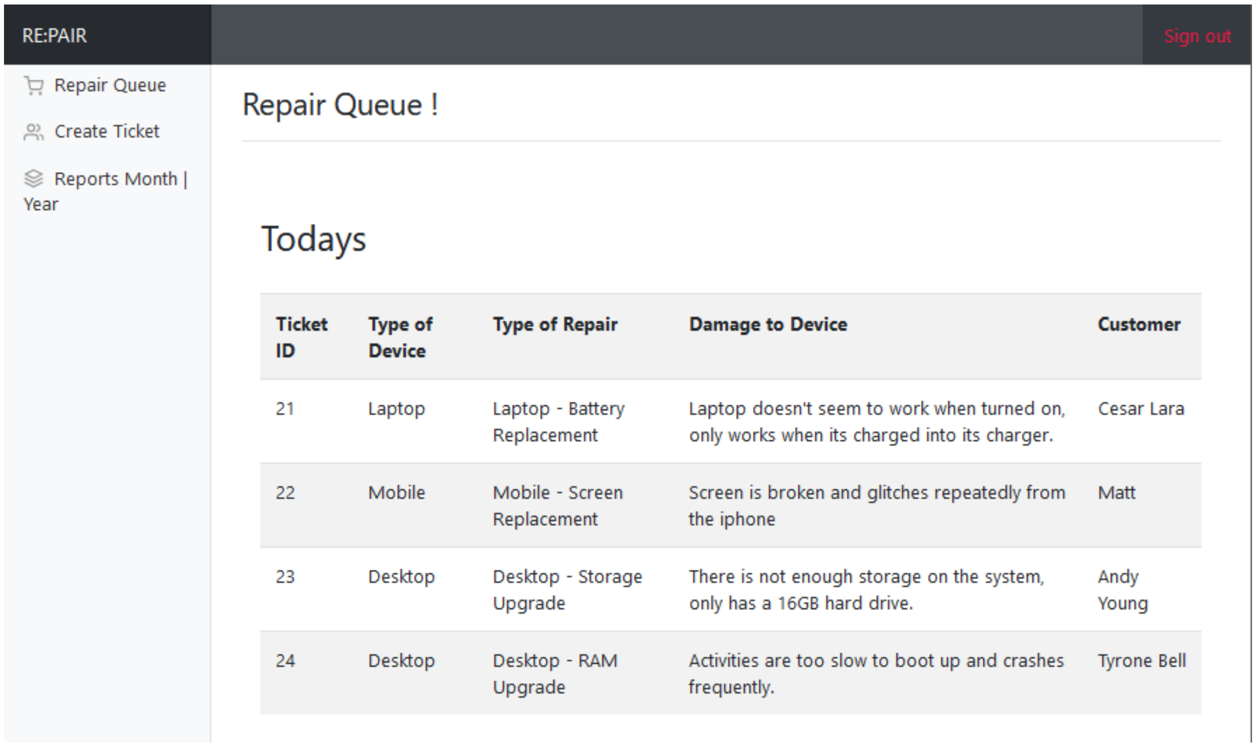
Password:

Create Account!

Functionality:

This webpage is meant to be used for the employee to sign up for access to the dashboard in case they don’t have an account.

## Repair Queue



Repair Queue !

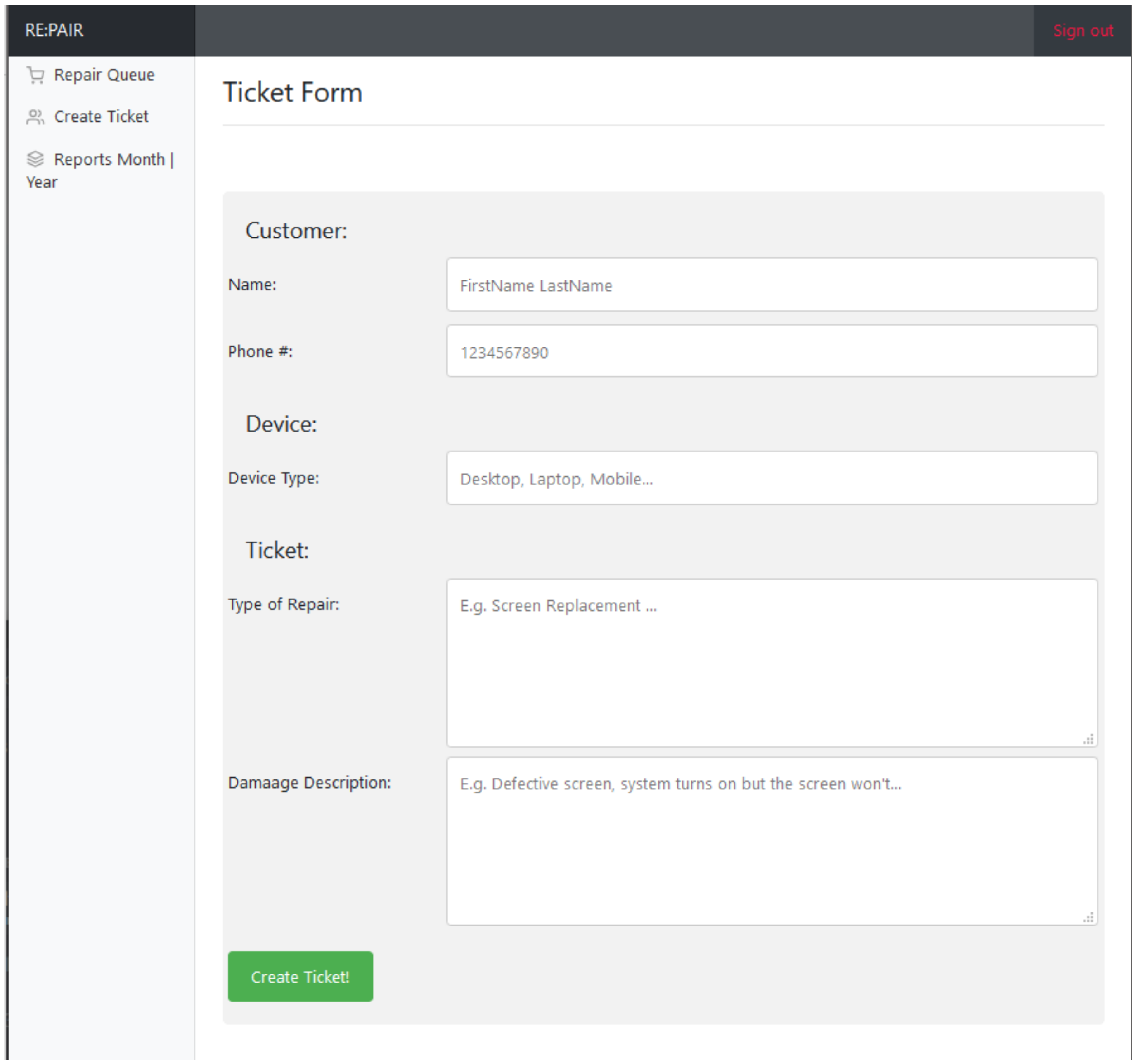
Todays

| Ticket ID | Type of Device | Type of Repair | Damage to Device | Customer |
|---|---|---|---|---|
| 21 | Laptop | Laptop - Battery Replacement | Laptop doesn't seem to work when turned on, only works when its charged into its charger. | Cesar Lara |
| 22 | Mobile | Mobile - Screen Replacement | Screen is broken and glitches repeatedly from the iphone | Matt |
| 23 | Desktop | Desktop - Storage Upgrade | There is not enough storage on the system, only has a 16GB hard drive. | Andy Young |
| 24 | Desktop | Desktop - RAM Upgrade | Activities are too slow to boot up and crashes frequently. | Tyrone Bell |

Functionality:

Displays the repair queue of the current tickets that need to be worked on that were put in the day.

**Ticket_Form**



Ticket Form

Customer:

| | |
|---|---|
| Name: | FirstName LastName |
| Phone #: | 1234567890 |

Device:

| | |
|---|---|
| Device Type: | Desktop, Laptop, Mobile... |

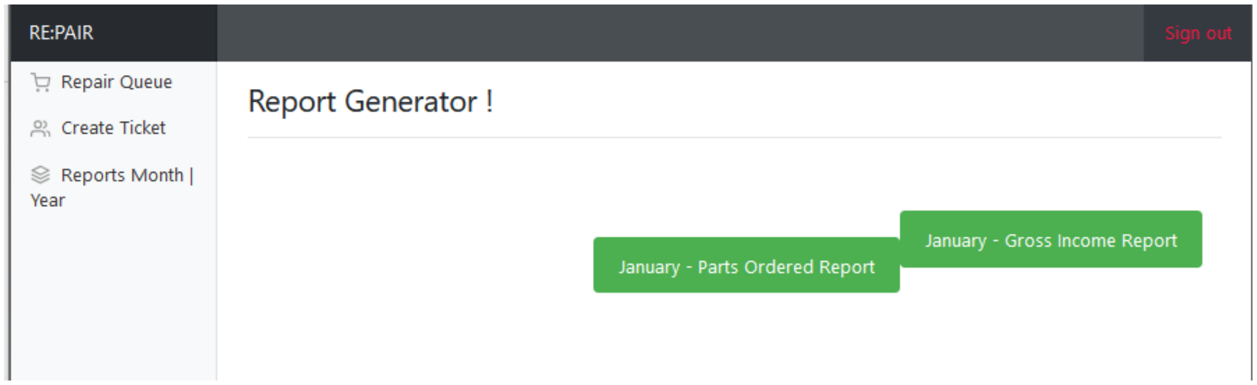
Ticket:

| | |
|---|---|
| Type of Repair: | E.g. Screen Replacement ... |
| Damaage Description: | E.g. Defective screen, system turns on but the screen won't... |

Functionality:

A ticket form where the employee can input a new ticket into the system for the customer on the device they want to get repaired. The new ticket will then be displayed in the repair queue tab on the left.

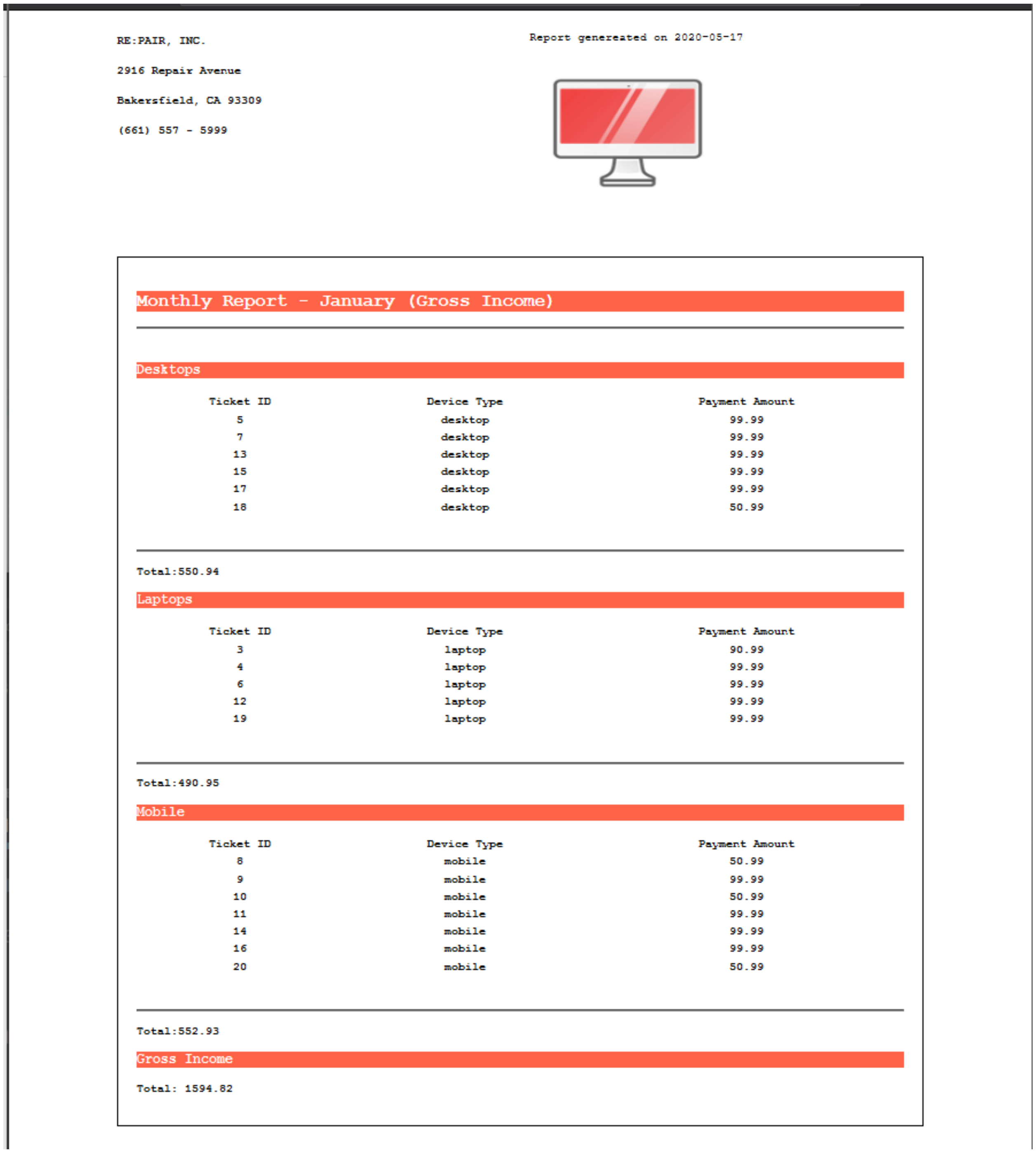
## <u>**Report Generator**</u>



Functionality:

Each button shows a php report showing the month of January's parts ordered and the gross income based off of the tables in the database.

Gross Income Report - January



RE:PAIR, INC.

2916 Repair Avenue

Bakersfield, CA 93309

(661) 557 - 5999

Report genereated on 2020-05-17

## Monthly Report - January (Gross Income)

### Desktops

| Ticket ID | Device Type | Payment Amount |
|---|---|---|
| 5 | desktop | 99.99 |
| 7 | desktop | 99.99 |
| 13 | desktop | 99.99 |
| 15 | desktop | 99.99 |
| 17 | desktop | 99.99 |
| 18 | desktop | 50.99 |

Total:550.94

### Laptops

| Ticket ID | Device Type | Payment Amount |
|---|---|---|
| 3 | laptop | 90.99 |
| 4 | laptop | 99.99 |
| 6 | laptop | 99.99 |
| 12 | laptop | 99.99 |
| 19 | laptop | 99.99 |

Total:490.95

### Mobile

| Ticket ID | Device Type | Payment Amount |
|---|---|---|
| 8 | mobile | 50.99 |
| 9 | mobile | 99.99 |
| 10 | mobile | 50.99 |
| 11 | mobile | 99.99 |
| 14 | mobile | 99.99 |
| 16 | mobile | 99.99 |
| 20 | mobile | 50.99 |

Total:552.93

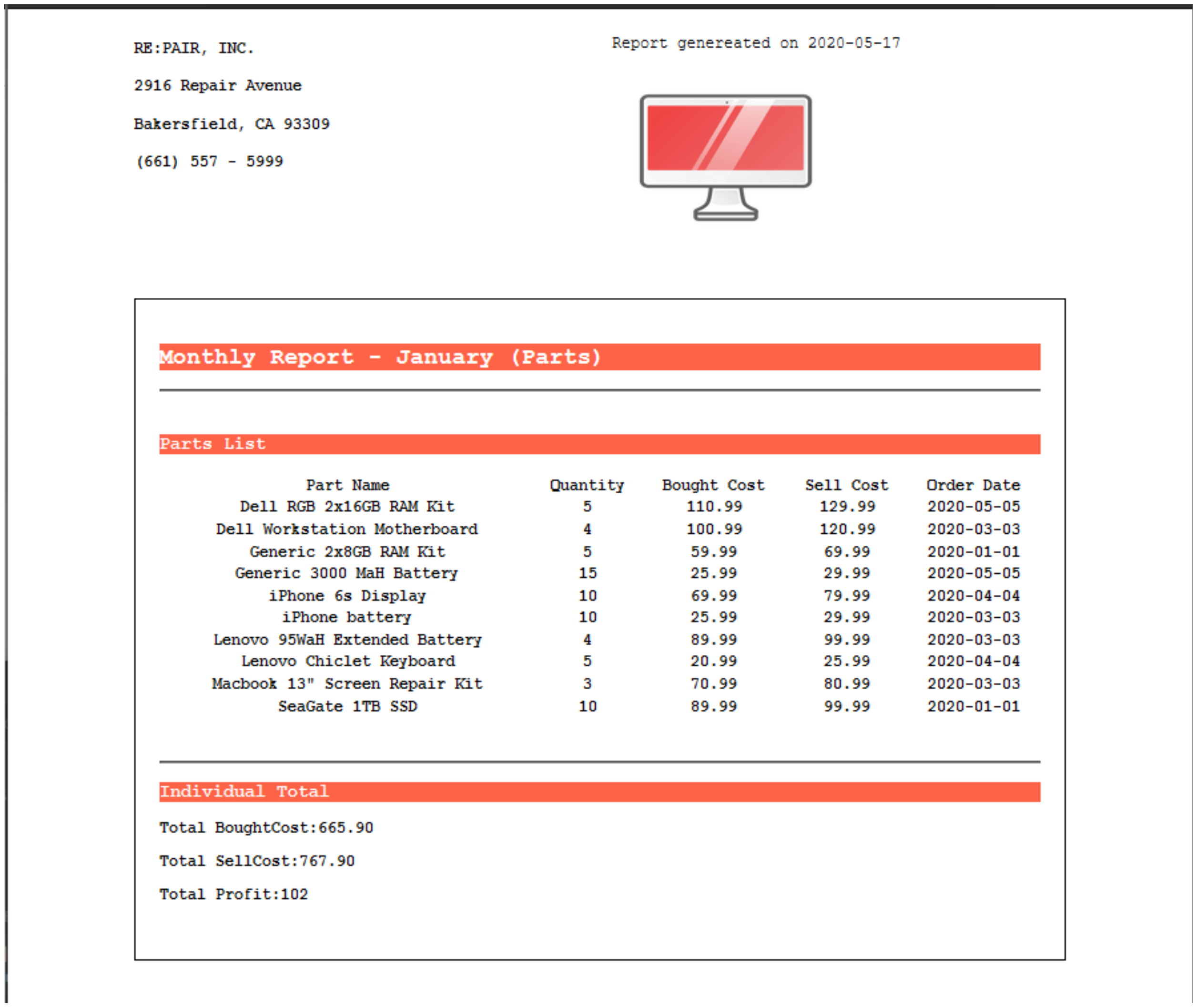
### Gross Income

Total: 1594.82

Functionality:

This report is to show a breakdown of the types of repairs that were done, how much each costed and then the total gross income that was made for the month of January.

Parts List Report - January



RE:PAIR, INC.

2916 Repair Avenue

Bakersfield, CA 93309

(661) 557 - 5999

Report genereated on 2020-05-17

Monthly Report - January (Parts)

Parts List

| Part Name | Quantity | Bought Cost | Sell Cost | Order Date |
|---|---|---|---|---|
| Dell RGB 2x16GB RAM Kit | 5 | 110.99 | 129.99 | 2020-05-05 |
| Dell Workstation Motherboard | 4 | 100.99 | 120.99 | 2020-03-03 |
| Generic 2x8GB RAM Kit | 5 | 59.99 | 69.99 | 2020-01-01 |
| Generic 3000 MaH Battery | 15 | 25.99 | 29.99 | 2020-05-05 |
| iPhone 6s Display | 10 | 69.99 | 79.99 | 2020-04-04 |
| iPhone battery | 10 | 25.99 | 29.99 | 2020-03-03 |
| Lenovo 95WaH Extended Battery | 4 | 89.99 | 99.99 | 2020-03-03 |
| Lenovo Chiclet Keyboard | 5 | 20.99 | 25.99 | 2020-04-04 |
| Macbook 13" Screen Repair Kit | 3 | 70.99 | 80.99 | 2020-03-03 |
| SeaGate 1TB SSD | 10 | 89.99 | 99.99 | 2020-01-01 |

Individual Total

Total BoughtCost:665.90

Total SellCost:767.90

Total Profit:102

Functionality:

This report is meant to show the parts that are in the inventory of what the repair shop has, as well as including the name, the price that the part was bought for and the price the repair shop is deciding to sell it for. At the end is a rough estimate of what they should hope to earn after selling 1 of each part in the list.

### 5.1.3 Tables, Views, Stored Subprograms, and Triggers used.

For this database project, the tables that were mostly used were Order, ReplacementParts, Models, Device, Ticket, Customer, Payment and Employee. These tables were used for the views as well, these views are called, Repair_Queue, Parts_History, Jan2020Wk_01_Desktop, Jan2020Wk_01_Laptop and Jan2020Wk_01_Mobile. No triggers were used or created for this database If any  triggers or stored procedures were created, then they would be for inserting data into the database or creating a special query that the user has defined of the output of data they want to see.

## 5.2. Programming Sections

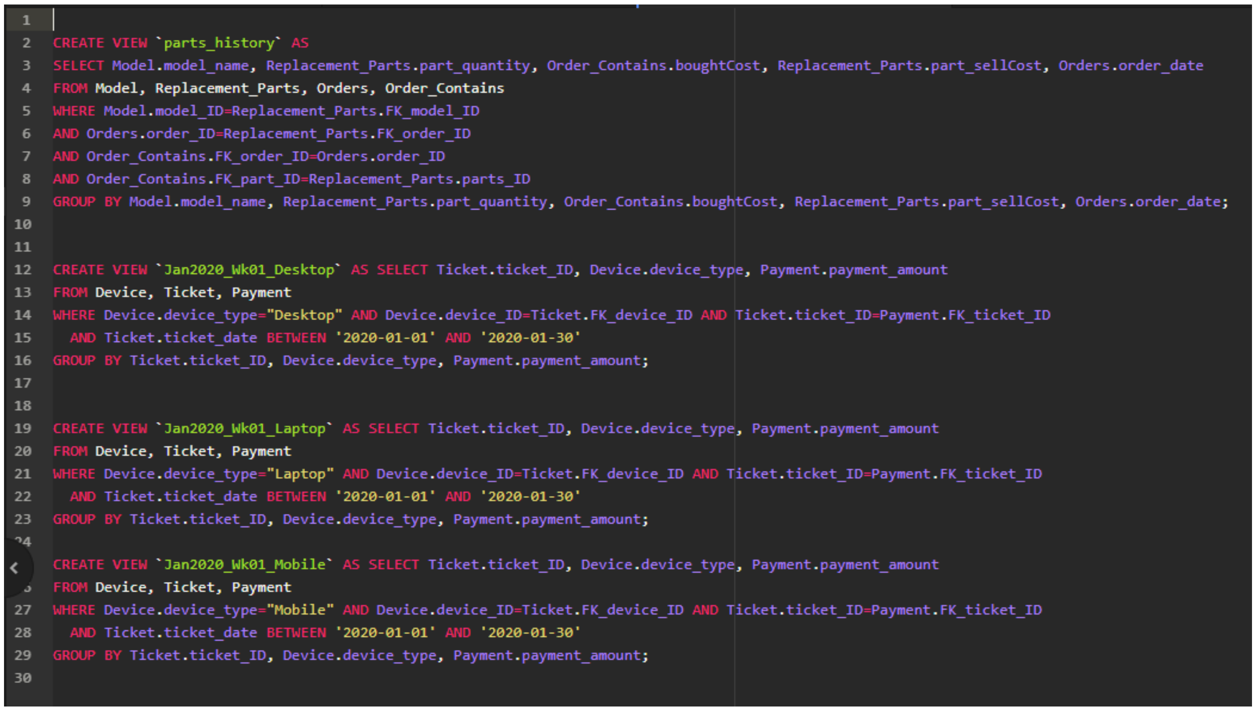
While developing this project, a number of different languages and software was used to help me build the database, create the front-end and get the back-end functionality working. The development that was primarily used for this database project involved HTML, CSS and Bootstrap for the front-end, MySQL and PHP for the back-end. Primarily using PHP to connect both the front-end and back-end together.

### 5.2.1 Server-Side Programming

The server-side languages used for this database project involved MySQL and PHP for the manipulation of data. The type of data manipulation that needed to be done involved querying said information from the database. Not only was querying information an important feature, but being able to create views and have data input possible to create the reports and to be able to have the front-end functionality of inserting information into the tables inside of the repair shop database.

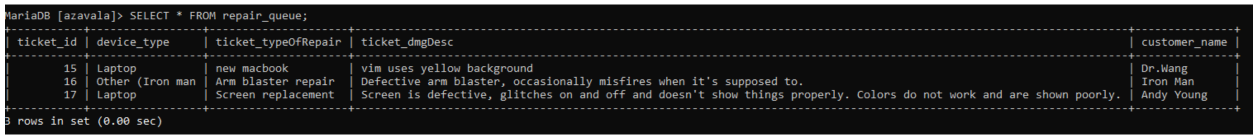
Multiple views were created from the tables.
These views and their results can be seen below.

Image of the sql code for each view, followed by the outputs.



```

CREATE VIEW `parts_history` AS
SELECT Model.model_name, Replacement_Parts.part_quantity, Order_Contains.boughtCost, Replacement_Parts.part_sellCost, Orders.order_date
FROM Model, Replacement_Parts, Orders, Order_Contains
WHERE Model.model_ID=Replacement_Parts.FK_model_ID
AND Orders.order_ID=Replacement_Parts.FK_order_ID
AND Order_Contains.FK_order_ID=Orders.order_ID
AND Order_Contains.FK_part_ID=Replacement_Parts.parts_ID
GROUP BY Model.model_name, Replacement_Parts.part_quantity, Order_Contains.boughtCost, Replacement_Parts.part_sellCost, Orders.order_date;


CREATE VIEW `Jan2020_Wk01_Desktop` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Desktop" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;


CREATE VIEW `Jan2020_Wk01_Laptop` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Laptop" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;

CREATE VIEW `Jan2020_Wk01_Mobile` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Mobile" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;

```

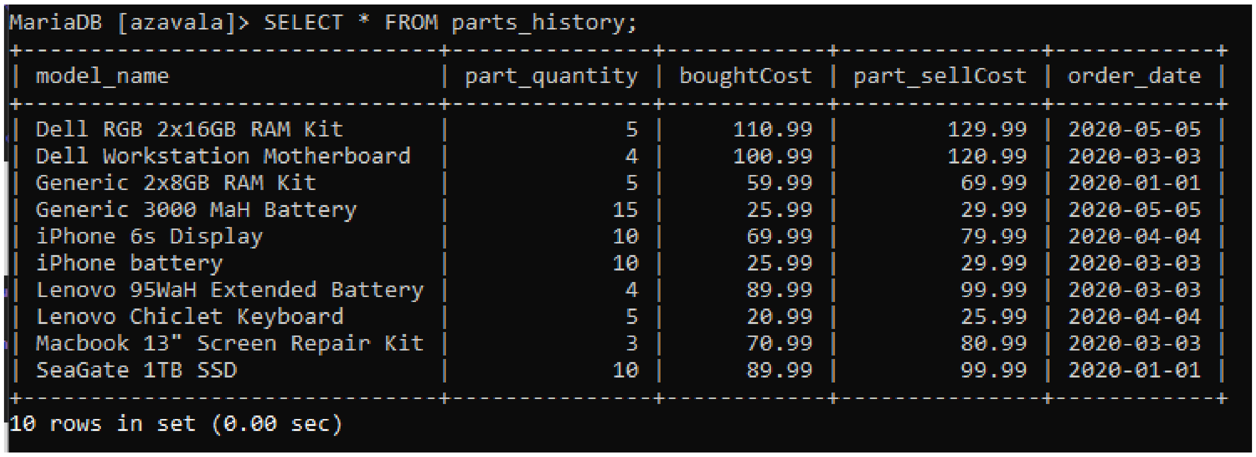
<u>Repair_Queue</u>



```
MariaDB [azavala]> SELECT * FROM repair_queue;
+-----------+------------------+---------------------+--------------------------------------------------------------------------------------------------+---------------+
| ticket_id | device_type      | ticket_typeOfRepair | ticket_dmgDesc                                                                                   | customer_name |
+-----------+------------------+---------------------+--------------------------------------------------------------------------------------------------+---------------+
|        15 | Laptop           | new macbook         | vim uses yellow background                                                                       | Dr.Wang       |
|        16 | Other (Iron man  | Arm blaster repair  | Defective arm blaster, occasionally misfires when it's supposed to.                              | Iron Man      |
|        17 | Laptop           | Screen replacement  | Screen is defective, glitches on and off and doesn't show things properly. Colors do not work and are shown poorly. | Andy Young    |
+-----------+------------------+---------------------+--------------------------------------------------------------------------------------------------+---------------+
3 rows in set (0.00 sec)
```

Purpose: The purpose of this view is to show the queue of repairs that need to be done for the current day at the repair shop. This updates whenever a new ticket is created from the ticket form on the dashboard.

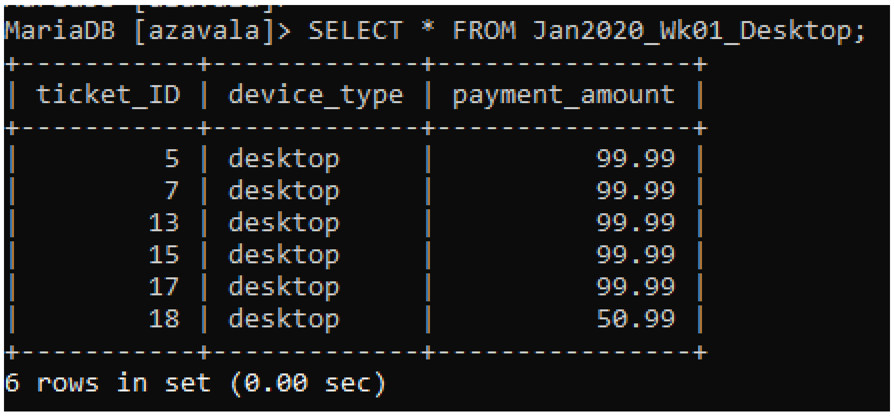
<u>Parts_History</u>



```
MariaDB [azavala]> SELECT * FROM parts_history;
+--------------------------------+---------------+------------+---------------+------------+
| model_name                     | part_quantity | boughtCost | part_sellCost | order_date |
+--------------------------------+---------------+------------+---------------+------------+
| Dell RGB 2x16GB RAM Kit        |             5 |     110.99 |        129.99 | 2020-05-05 |
| Dell Workstation Motherboard   |             4 |     100.99 |        120.99 | 2020-03-03 |
| Generic 2x8GB RAM Kit          |             5 |      59.99 |         69.99 | 2020-01-01 |
| Generic 3000 MaH Battery       |            15 |      25.99 |         29.99 | 2020-05-05 |
| iPhone 6s Display              |            10 |      69.99 |         79.99 | 2020-04-04 |
| iPhone battery                 |            10 |      25.99 |         29.99 | 2020-03-03 |
| Lenovo 95WaH Extended Battery  |             4 |      89.99 |         99.99 | 2020-03-03 |
| Lenovo Chiclet Keyboard        |             5 |      20.99 |         25.99 | 2020-04-04 |
| Macbook 13" Screen Repair Kit  |             3 |      70.99 |         80.99 | 2020-03-03 |
| SeaGate 1TB SSD                |            10 |      89.99 |         99.99 | 2020-01-01 |
+--------------------------------+---------------+------------+---------------+------------+
10 rows in set (0.00 sec)
```

Purpose: Displays the price of each item for the month of january that was ordered, the price repair shop bought it for, the price the repair shop is going to sell it for, the quantity and what the order date of the part when the employee ordered it. Used for the report.

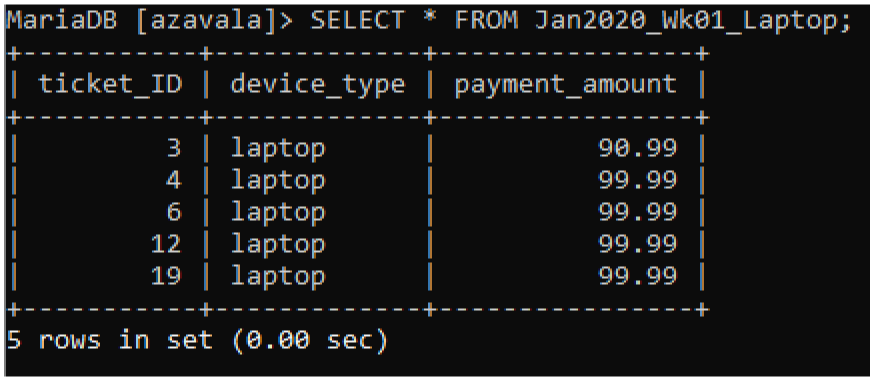
<u>Jan2020_Wk01_Desktop</u>



```
MariaDB [azavala]> SELECT * FROM Jan2020_Wk01_Desktop;
+-----------+-------------+----------------+
| ticket_ID | device_type | payment_amount |
+-----------+-------------+----------------+
|         5 | desktop     |          99.99 |
|         7 | desktop     |          99.99 |
|        13 | desktop     |          99.99 |
|        15 | desktop     |          99.99 |
|        17 | desktop     |          99.99 |
|        18 | desktop     |          50.99 |
+-----------+-------------+----------------+
6 rows in set (0.00 sec)
```

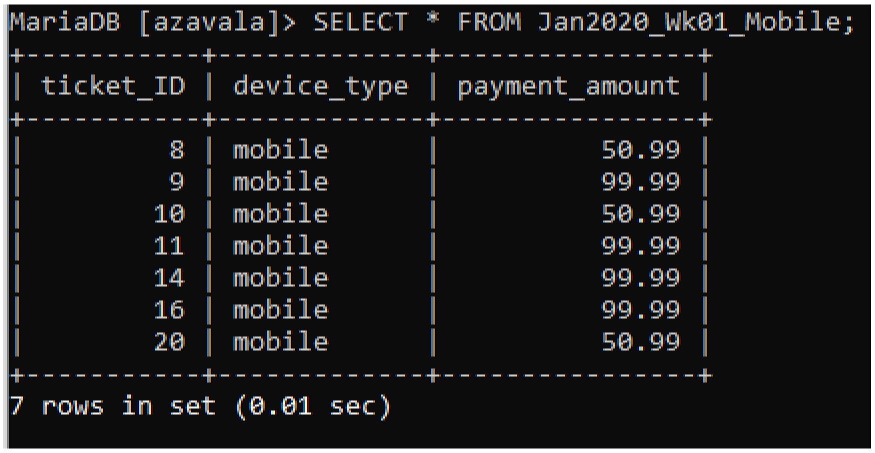
Purpose:  Displays the ticket number, the type of device that was worked on and the amount that was paid for that ticket. Used for the report.

<u>Jan2020_Wk01_Laptop</u>



```
MariaDB [azavala]> SELECT * FROM Jan2020_Wk01_Laptop;
+-----------+-------------+----------------+
| ticket_ID | device_type | payment_amount |
+-----------+-------------+----------------+
|         3 | laptop      |          90.99 |
|         4 | laptop      |          99.99 |
|         6 | laptop      |          99.99 |
|        12 | laptop      |          99.99 |
|        19 | laptop      |          99.99 |
+-----------+-------------+----------------+
5 rows in set (0.00 sec)
```

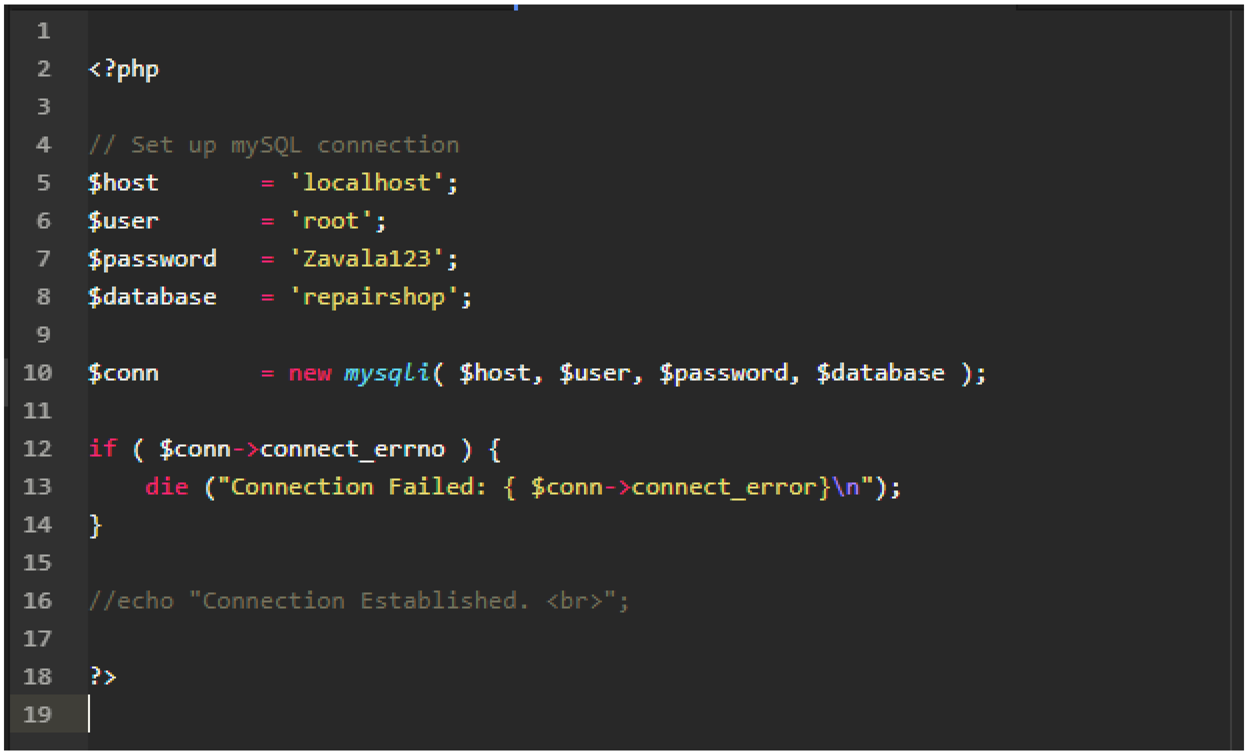
<u>Jan2020_Wk01_Mobile</u>


```
MariaDB [azavala]> SELECT * FROM Jan2020_Wk01_Mobile;
+-----------+-------------+----------------+
| ticket_ID | device_type | payment_amount |
+-----------+-------------+----------------+
|         8 | mobile      |          50.99 |
|         9 | mobile      |          99.99 |
|        10 | mobile      |          50.99 |
|        11 | mobile      |          99.99 |
|        14 | mobile      |          99.99 |
|        16 | mobile      |          99.99 |
|        20 | mobile      |          50.99 |
+-----------+-------------+----------------+
7 rows in set (0.01 sec)
```

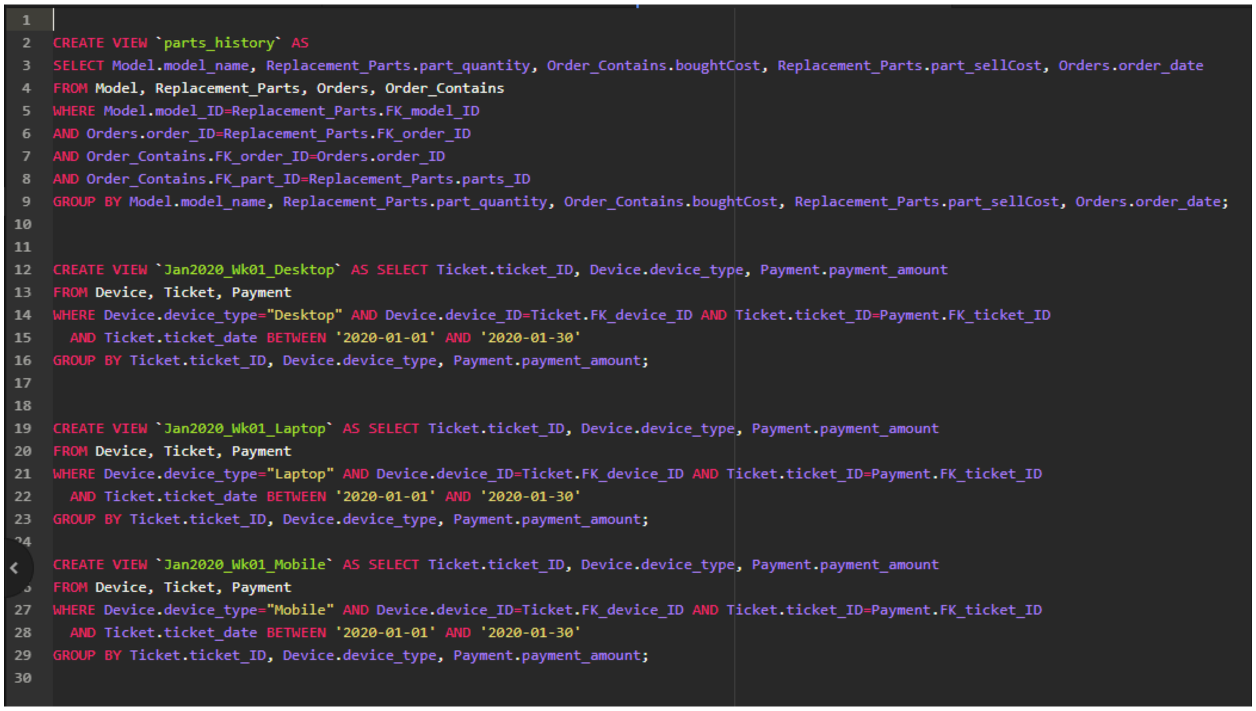
### 5.2.2 Middle-Tier Programming

Middle-tier programming languages that were used to tie together the front-end of the dashboard and the back-end of the repair shop database.

<u>Database Connection Code:</u>

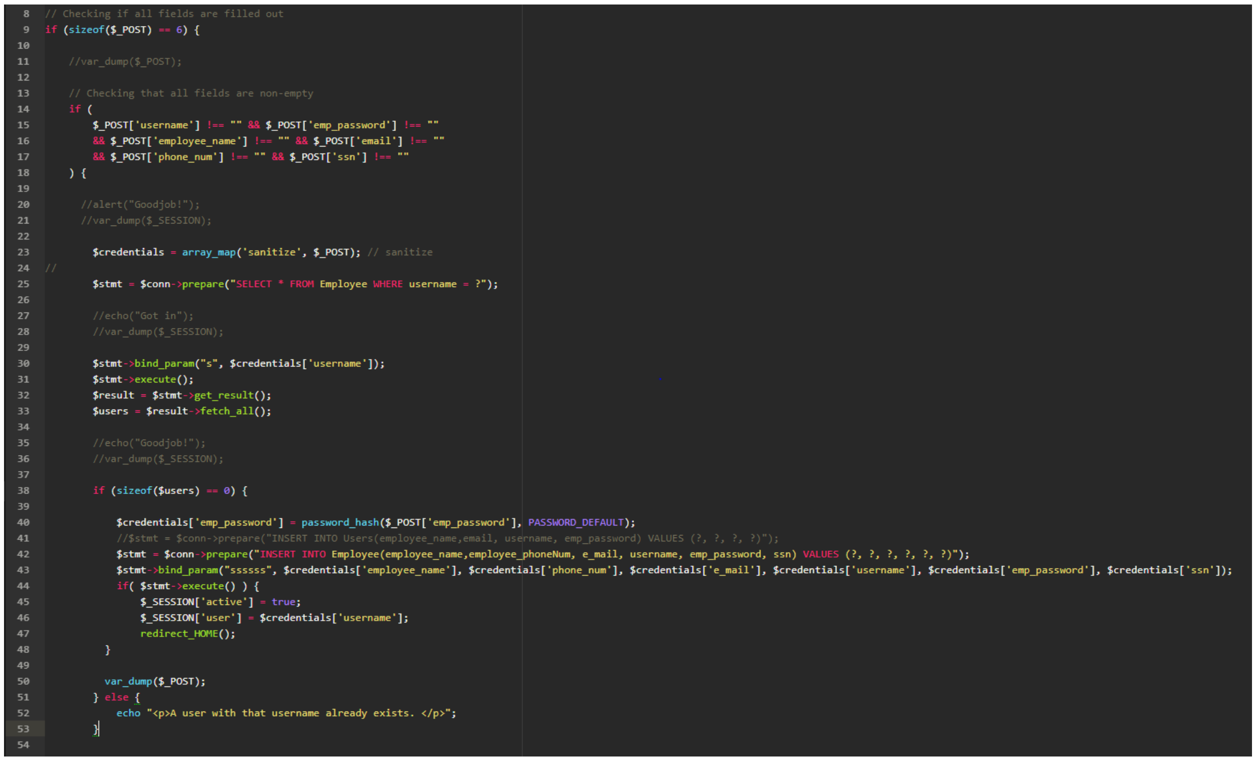

```php

<?php

// Set up mySQL connection
$host       = 'localhost';
$user       = 'root';
$password   = 'Zavala123';
$database   = 'repairshop';

$conn       = new mysqli( $host, $user, $password, $database );

if ( $conn->connect_errno ) {
    die ("Connection Failed: { $conn->connect_error}\n");
}

//echo "Connection Established. <br>";

?>

```

## Code for Views:



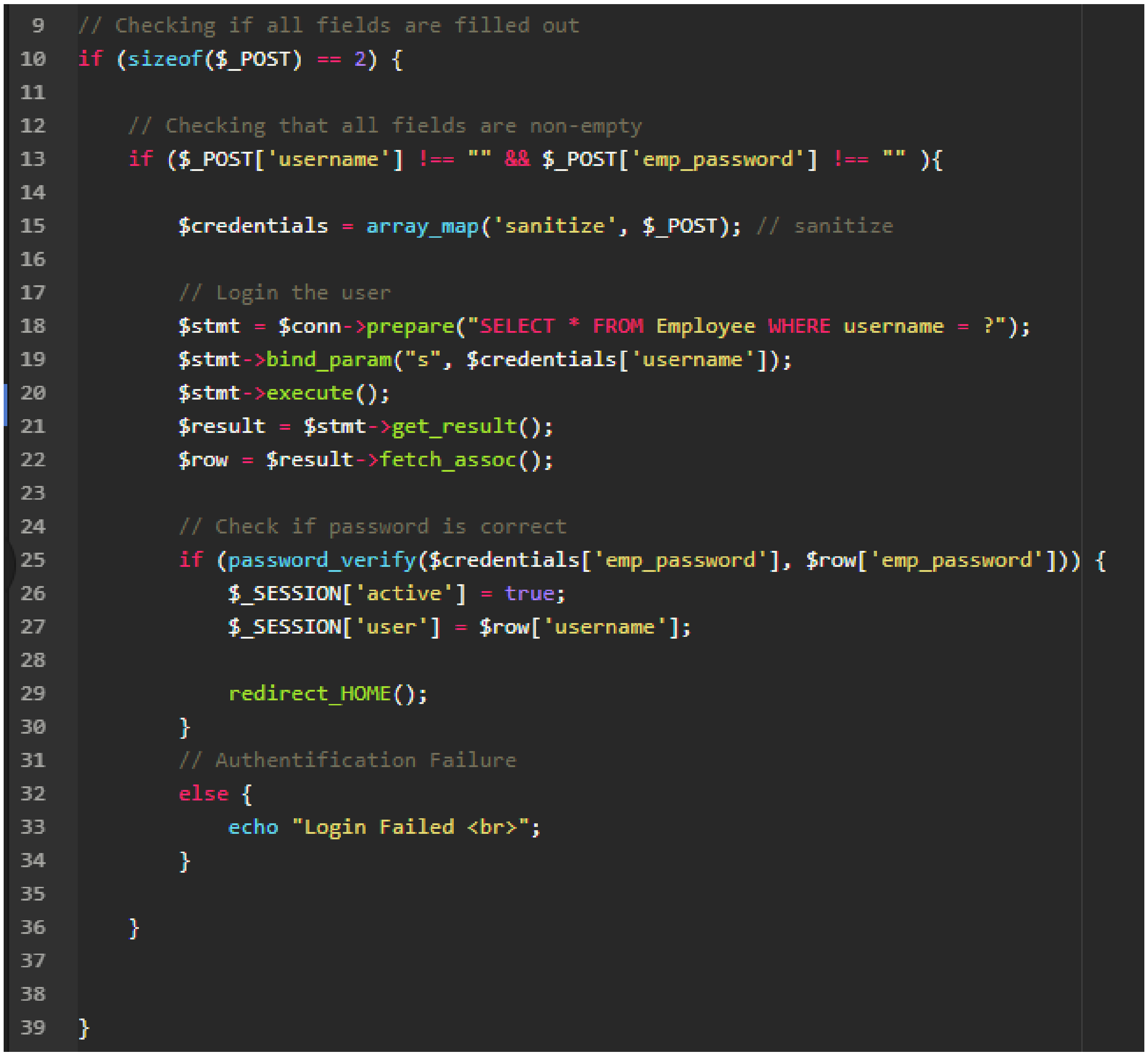
```sql

CREATE VIEW `parts_history` AS
SELECT Model.model_name, Replacement_Parts.part_quantity, Order_Contains.boughtCost, Replacement_Parts.part_sellCost, Orders.order_date
FROM Model, Replacement_Parts, Orders, Order_Contains
WHERE Model.model_ID=Replacement_Parts.FK_model_ID
AND Orders.order_ID=Replacement_Parts.FK_order_ID
AND Order_Contains.FK_order_ID=Orders.order_ID
AND Order_Contains.FK_part_ID=Replacement_Parts.parts_ID
GROUP BY Model.model_name, Replacement_Parts.part_quantity, Order_Contains.boughtCost, Replacement_Parts.part_sellCost, Orders.order_date;


CREATE VIEW `Jan2020_Wk01_Desktop` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Desktop" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;


CREATE VIEW `Jan2020_Wk01_Laptop` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Laptop" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;

CREATE VIEW `Jan2020_Wk01_Mobile` AS SELECT Ticket.ticket_ID, Device.device_type, Payment.payment_amount
FROM Device, Ticket, Payment
WHERE Device.device_type="Mobile" AND Device.device_ID=Ticket.FK_device_ID AND Ticket.ticket_ID=Payment.FK_ticket_ID
  AND Ticket.ticket_date BETWEEN '2020-01-01' AND '2020-01-30'
GROUP BY Ticket.ticket_ID, Device.device_type, Payment.payment_amount;

```

## Signing Up :



```php
// Checking if all fields are filled out
if (sizeof($_POST) == 6) {

    //var_dump($_POST);

    // Checking that all fields are non-empty
    if (
        $_POST['username'] !== "" && $_POST['emp_password'] !== ""
        && $_POST['employee_name'] !== "" && $_POST['email'] !== ""
        && $_POST['phone_num'] !== "" && $_POST['ssn'] !== ""
    ) {

      //alert("Goodjob!");
      //var_dump($_SESSION);

        $credentials = array_map('sanitize', $_POST); // sanitize
//
        $stmt = $conn->prepare("SELECT * FROM Employee WHERE username = ?");

        //echo("Got in");
        //var_dump($_SESSION);

        $stmt->bind_param("s", $credentials['username']);
        $stmt->execute();
        $result = $stmt->get_result();
        $users = $result->fetch_all();

        //echo("Goodjob!");
        //var_dump($_SESSION);

        if (sizeof($users) == 0) {

            $credentials['emp_password'] = password_hash($_POST['emp_password'], PASSWORD_DEFAULT);
            //$stmt = $conn->prepare("INSERT INTO Users(employee_name,email, username, emp_password) VALUES (?, ?, ?, ?)");
            $stmt = $conn->prepare("INSERT INTO Employee(employee_name,employee_phoneNum, e_mail, username, emp_password, ssn) VALUES (?, ?, ?, ?, ?, ?)");
            $stmt->bind_param("ssssss", $credentials['employee_name'], $credentials['phone_num'], $credentials['e_mail'], $credentials['username'], $credentials['emp_password'], $credentials['ssn']);
            if( $stmt->execute() ) {
                $_SESSION['active'] = true;
                $_SESSION['user'] = $credentials['username'];
                redirect_HOME();
          }

          var_dump($_POST);
        } else {
            echo "<p>A user with that username already exists. </p>";
        }

```

## Logging In:



```php
// Checking if all fields are filled out
if (sizeof($_POST) == 2) {

    // Checking that all fields are non-empty
    if ($_POST['username'] !== "" && $_POST['emp_password'] !== "" ){

        $credentials = array_map('sanitize', $_POST); // sanitize

        // Login the user
        $stmt = $conn->prepare("SELECT * FROM Employee WHERE username = ?");
        $stmt->bind_param("s", $credentials['username']);
        $stmt->execute();
        $result = $stmt->get_result();
        $row = $result->fetch_assoc();

        // Check if password is correct
        if (password_verify($credentials['emp_password'], $row['emp_password'])) {
            $_SESSION['active'] = true;
            $_SESSION['user'] = $row['username'];

            redirect_HOME();
        }
        // Authentification Failure
        else {
            echo "Login Failed <br>";
        }

    }


}
```

### 5.2.3 Client-side programming

For this database project, a majority of the programming was done using PHP and mySQL queries and views, There is no javascript or client-side programming that was done. The main goal of this project with the limited time I had was to get the functionality working as best as I could, rather than focusing on the aesthetics.

## 5.3. Survey Questions

| Outcome | Answers ( 1 - 10 ) |
|---|---|
| (3b) An ability to analyze a problem, and identify and define the computing requirements and specifications appropriate to its solution. | 7 |
| (3e) 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 |
| (3f) 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 |
| (3j) 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. | 7 |