**UNIVERSITY OF ARIZONA**

Prepared for

Enterprise Data Management

**TUGO BIKE SHARE**



By

**Group H**: Data Diggers

**Group Members**

Abhishek Kakani

Gaurav Mahajan

Tanya Sharma

Asmita Singh

Kanika Singh

Teja Tumma

TABLE OF CONTENTS

[REQUIREMENTS ANALYSIS 5](#_Toc152796835)

[1.1 About the client 5](#_Toc152796836)

[1.2 Requirements 5](#_Toc152796837)

[CONCEPTUAL SCHEMA 8](#_Toc152796838)

[2.1 Data Dictionary (Conceptual / for ER Modeling) 8](#_Toc152796839)

[2.2 ER DIAGRAM 17](#_Toc152796840)

[RELATIONAL SCHEMA 18](#_Toc152796841)

[3.1 Entity Classes 18](#_Toc152796842)

[3.2 Relationship Classes & Weak Entity Class 19](#_Toc152796843)

[3.3 Data Dictionary (for Relational Design) 21](#_Toc152796844)

[3.4 SQL Create Statements 32](#_Toc152796845)

[3. 5 Sequences 50](#_Toc152796846)

[QUERIES 51](#_Toc152796847)

[4.1. Finding out top customers based on maximum rental time 51](#_Toc152796848)

[4.2. Analyze Net Income Performance for Consecutive Quarters 51](#_Toc152796849)

[4.3. Identify Bike Hotspots 52](#_Toc152796850)

[4.4. Calculate the time between consecutive bike rentals for each bike 53](#_Toc152796851)

[4.5. Analyzing the frequency of the most common type of service done on Bikes and Stations 54](#_Toc152796852)

[4.6. Identify Top 5 customers who paid the highest Excess Charges for the Bike rentals 55](#_Toc152796853)

[4.7. Customer Payment Categories Analysis 56](#_Toc152796854)

[4.8. Monthly Revenue Analysis 57](#_Toc152796855)

[4.9. Active Subscriber Percentage Analysis 57](#_Toc152796856)

[4.10. Subscription Count by Plan Analysis 58](#_Toc152796857)

[4.11. Top 3 Bike Models by Trip Count Analysis 58](#_Toc152796858)

[4.12. On-Call Employee with the Highest Complaint Count 59](#_Toc152796859)

[4.13. Identify the top 5 revenue-generating vendors for Spare Parts 60](#_Toc152796860)

[4.14. Customer Unresolved Complaints Analysis 60](#_Toc152796861)

[TRIGGERS and PROCEDURES 62](#_Toc152796862)

[5.1 Triggers 62](#_Toc152796863)

[5.1.1. Trigger for calculating Excess Charge for a trip 62](#_Toc152796864)

[5.1.2. Trigger for calculating Free Bike Space at a station 63](#_Toc152796865)

[5.2 Procedures 63](#_Toc152796866)

[5.2.1. Procedure for Calculating Monthly Net Income 63](#_Toc152796867)

[5.2.2 Procedure for Rewarding Loyalty Points for all Customers 68](#_Toc152796868)

[5.2.3 Procedure for Updating Reward Points for Customers 69](#_Toc152796869)

[USER INTERFACE 72](#_Toc152796870)

[6.1. User Interface walk-through (Recorded Video Walkthrough) 72](#_Toc152796871)

[6.1.1 Home page or Landing page: 72](#_Toc152796872)

[6.2 Perform Add, Update and Delete Operations on Table 76](#_Toc152796873)

[6.2.1. CUSTOMERS Table 76](#_Toc152796874)

[6.2.2. Other Data Tables 76](#_Toc152796875)

[6.2.3: Business Insights using SQL Queries. 77](#_Toc152796876)

[IMPLEMENTATION PLAN 79](#_Toc152796877)

[APPENDIX A 88](#_Toc152796878)

[REQUIREMENTS ANALYSIS 4](#_Toc152792172)

[1.1 About the client 4](#_Toc152792173)

[1.2 Requirements 4](#_Toc152792174)

[CONCEPTUAL SCHEMA 7](#_Toc152792175)

[2.1 Data Dictionary (Conceptual / for ER Modeling) 7](#_Toc152792176)

[2.2 ER DIAGRAM 16](#_Toc152792177)

[RELATIONAL SCHEMA 17](#_Toc152792178)

[3.1 Entity Classes 17](#_Toc152792179)

[3.2 Relationship Classes & Weak Entity Class 18](#_Toc152792180)

[3.3 Data Dictionary (for Relational Design) 20](#_Toc152792181)

[3.4 SQL Create Statements 29](#_Toc152792182)

[QUERIES 54](#_Toc152792183)

[4.1. Finding out top customers based on maximum rental time 54](#_Toc152792184)

[4.2. Analyze Net Income Performance for Consecutive Quarters 54](#_Toc152792185)

[4.3. Identify Bike Hotspots 55](#_Toc152792186)

[4.4. Calculate the time between consecutive bike rentals for each bike 56](#_Toc152792187)

[4.5. Analyzing the frequency of the most common type of service done on Bikes and Stations 57](#_Toc152792188)

[4.6. Identify Top 5 customers who paid the highest Excess Charges for the Bike rentals 58](#_Toc152792189)

[4.7. Customer Payment Categories Analysis 59](#_Toc152792190)

[4.8. Monthly Revenue Analysis 60](#_Toc152792191)

[4.9. Active Subscriber Percentage Analysis 60](#_Toc152792192)

[4.10. Subscription Count by Plan Analysis 61](#_Toc152792193)

[4.11. Top 3 Bike Models by Trip Count Analysis 61](#_Toc152792194)

[4.12. On-Call Employee with the Highest Complaint Count 62](#_Toc152792195)

[4.13. Identify the top 5 revenue-generating vendors for Spare Parts 63](#_Toc152792196)

[4.14. Customer Unresolved Complaints Analysis 63](#_Toc152792197)

[TRIGGERS and PROCEDURES 65](#_Toc152792198)

[5.1 Triggers 65](#_Toc152792199)

[5.1.1. Trigger for calculating Excess Charge for a trip 65](#_Toc152792200)

[5.1.2. Trigger for calculating Free Bike Space at a station 66](#_Toc152792201)

[5.2 Procedures 66](#_Toc152792202)

[5.2.1. Procedure for Calculating Monthly Net Income 66](#_Toc152792203)

[5.2.2 Procedure for Rewarding Loyalty Points for all Customers 71](#_Toc152792204)

[5.2.3 Procedure for Updating Reward Points for Customers 72](#_Toc152792205)

[USER INTERFACE 75](#_Toc152792206)

[6.1. User Interface walk-through (Recorded Video Walkthrough) 75](#_Toc152792207)

[6.1.1 Home page or Landing page: 75](#_Toc152792208)

[6.2 Perform Add, Update and Delete Operations on Table 79](#_Toc152792209)

[6.2.1. CUSTOMERS Table 79](#_Toc152792210)

[6.2.2. Other Data Tables 79](#_Toc152792211)

[6.2.3: Business Insights using SQL Queries. 80](#_Toc152792212)

[IMPLEMENTATION PLAN 82](#_Toc152792213)

[APPENDIX A 91](#_Toc152792214)

**CHAPTER 1**

# REQUIREMENTS ANALYSIS

## 1.1 About the client

Tugo Bike Share is a short-term bike rental company. It offers a 30-minute ride allowing the user to unlock a bike, ride for 30 min, and return it at the same or another Tugo Station. The application is used by the customers who are the end users and employees of the company.

## 1.2 Requirements

Some of the client requirements are to manage diverse bike models, their rental prices, and station capacities for effective bike deployment, the ability to operate in multiple cities with a focus on station management, the ability to track trips in real-time, efficient tracking of employee details, roles, and payroll information for streamlined operation, efficient supply chain management through vendor coordination and maintenance records.

The company has different employees to manage several different operations for a seamless experience. The employees working for TugoBike have an employee ID, an employee name (last name, first name), date of birth, age, SSN, gender, date of joining, date of leaving, and location of work. TugoBike also tracks the contact information of the employees such as contact number, email, and address. An employee can have multiple contact numbers. The address is stored as a combination of street, city, state, and zip code. We also track the current job title (e.g., manager, mechanics, on-call support, etc.), manager, and department of each employee. TugoBike also tracks the manager to whom an employee reports using a manager ID. It is important to store emergency contact information that contains the emergency contact name and number. Employees must provide one emergency contact. Employees will be paid according to their salary grade.

An employee can be a manager and supervise employees. TugoBike also wants to track which employee is managed by which manager which is done by the managerID for each employee.

Each employee works for at most one department at a time. Each department is distinguished by its unique department ID and name and will have at least one employee.

On-call support employees are responsible for handling the complaints made by customers. All employees may not handle complaints and an employee can handle more than one complaint. Mechanics are responsible for handling the maintenance of bikes and stations.

While creating a profile, userID (authID), password, email, security question, and security answer are stored commonly along with type to distinguish between employee and customer login. An authentication history is logged for both employees and customers separately. Each time a login is attempted, login-in time and log-out time are stored along with the coordinates. Attempt to make a login into the accounts (employee or customer) is validated by storing userID and password. An employee or customer may not have an authentication history.

On the other hand, customers have first and last names, date of birth, age, address, email, and multiple mobile numbers stored in the database. We also want to track how long the customer has been with us and what is the current status (active or inactive). Each customer is assigned a uniqueID.

Customers also provide ratings for their user experience. The ratings contain the date of post was created, bikeID for which the review was written, and the review content. Each customer can provide multiple ratings for the bikes they have used.

Customers can also register complaints. While registering the complaint, an ID is generated, and raised time, resolved time, nature of complaint and the status (in progress, resolved, etc.) of the complaint is stored. A complaint will have a customer associated and a customer may make more than one complaint.

Payments are another element of focus. We want to track the payment history for the customers. When a payment is made, the time, amount and type of payment (subscription or late payment) is stored. Payments will be make by many customers and a customer can make many payments as they continue to use the services. If a refund has to be initiated, we have the refundID and the amount which is associated with the customer's payment. A refund could be made multiple times for a particular customer and a refund is not initiated for every payment.

Based on the payment amount, customers receive loyalty points. The loyalty points increase or decrease as they make payments or redeem the points respectively.

TugoBIke has different discounts for the customers which can be applied on the payments. Discounts have discountIDs, a discount code, validity period, the amount of discount and the text description for the given code. We also have different subscription models where we have monthly or annual plans. So, each plan is distinguished by a unique plan ID. A subscription model has a base price, title, and description. Whenever a customer subscribes to the plan, the information is recorded as the subscription status where the start data and end date of the subscription is recorded. A customer will have a long subscriptions history.

The main element of focus for TugoBikes is the trips made by the users. For a trip, there will be a start and end time associated with each trip and is identified by an ID. We also have an additional fee if the ride exceeds 30 min and it is calculated on a per minute basis. Since it is a bike sharing model, a trip can only have one user, but a user can make take many rides.

Bikes belong to different bike models. Bike models include model number(modelID), model name, type, description, rental price and cost to company. The bikes include bike number(bikeID), bike name and year of manufacture. A bike will have multiple trips.

These bikes are parked at bike stations from where the bikes are picked up and dropped off. There are several stations in many cities. So, it is important to store the details of every station. Each station has its own ID, name, bike capacity, available bikes, bikes under repair, free bike spaces for parking, location and city of the station.

Station needs maintenance on regular basis, and this is stored under station maintenance history. The mechanics handle the repair and maintenance. For each maintenance, we have the start and end time, type of maintenance (regular or repair) is recorded. Since a bike will have multiple maintenance records, we have an ID for each instance.

Bike maintenance works the same way. For every bike repair or maintenance, there is a mechanic associated and start and end time, type of maintenance is recorded.

Vendors supply us the bikes. Each vendor has a name, phone number, email, and address for contact. Address contains street, city, state, zip code and country. We also purchase spare parts from vendors required for maintenance. Spare parts have name, brand, weight, color, and price. It is identified by unique ID.

Purchase of spare parts are recorded separately. A spare part will be brought many times. Each time a part is bought we store the date, billing amount and the quantity bought.

**CHAPTER 2**

# CONCEPTUAL SCHEMA

## 2.1 Data Dictionary (Conceptual / for ER Modeling)

|  |  |  |
| --- | --- | --- |
| **Schema Construct** | **Construct Meaning** | **Structure and/or Constraints** |
| **ENTITIES** | | |
| **Authentication** | Entity Class, to model authentication while logging |  |
| * authID | Identifying Attribute, Username while creating profile | Varchar containing alphanumeric combination |
| * password | Password for logging |  |
| * email | Email for account recovery | unique |
| * type | Stores for whom the authID and password are (e.g., employee or customer) | Contains only 2 values, employees and customer |
| * securityQuestion | Question for security |  |
| * securityAnswer | Corresponding answer for the question |  |
| **Bikes** | Entity Class, to model bikes |  |
| * bikeID | Identifying Attribute | Format: “B” prefix followed by 3 digits. |
| * bikeName | Name associated with the bike |  |
| * yearOfManufacture | Year when the bike was manufactured | Cannot be NULL |
|  |  |  |
| **Bike Maintenance** | Weak Entity Class, to model bike maintenances |  |
| * bMaintenanceID | Identifying Attribute | Format: “BM” prefix followed by 3 digits. |
| * bStartTime | Bike maintenance start time |  |
| * bEndTime | Bike maintenance end time |  |
| * bTypeOfService | Type of maintenance (regular or repair) |  |
| **Bike Models** | Entity Class, to model bike models | Format: “M” prefix followed by 3 digits. |
| * modelID | Identifying Attribute |  |
| * modelName | Model of the Bike |  |
| * description | Description of the bike and its utility |  |
| * type | Nature of bike(urban, mountain etc.) |  |
| * rentalPrice | price (in $) for renting the bike to customer for one time use | Cannot be negative |
| * costToCompany | price (in $) for buying bikes | Cannot be negative |
| **Complaints** | Entity Class, to model customer complaints |  |
| * complaintID | Identifying Attribute | Format: “C” prefix followed by 3 digits. |
| * typeOfComplaint | Contains the name of entity the complaint is for | Allows only 3 values ‘Bike’, ‘Station’ or ‘Payment’ |
| * status | Stores the status of the complaint | Allows only 3 values:  'N', ‘P’ or ‘R’ which stands for New, In Progress and Resolved respectively |
| * raisedTime | Stores the timestamp information when complaint was registered |  |
| * resolveTime | Stores the timestamp information when complaint was resolved |  |
| **Customer Authentication History** | Weak Entity Class, to model customer’s login/authentication history |  |
| * loggedInTime | Identifying attribute, stores the timestamp when customer is authenticated to log in |  |
| * loggedOutTime | Stores the timestamp when customer logs out |  |
| * custAuthLongitude | Stores the latitude co-ordinates of the place from where customer is logging in |  |
| * custAuthLatitude | Stores the longitude co-ordinates of the place from where customer is logging in |  |
| **Customers** | Entity Class, to model customers |  |
| * custID | Identifying Attribute | Format: “CUST” prefix followed by 6 digits. |
| * fName | Stores first name of the customer |  |
| * lName | Stores last name of the customer |  |
| * age | Stores the age of the customer | Derived based on the date of birth |
| * dob | Stores the date of birth |  |
| * custSince | Store the date on which customer register | Takes the system date by default |
| * street | Street name for address |  |
| * city | City for address |  |
| * state | State for address |  |
| * zipcode | Zipcode for address |  |
| * email | Customer’s email address |  |
| * activeStatus | Indicates if the customer currently has an active subscription | Holds on two values either ‘Y’ or ‘N’ |
| **Cust Rewards** | Entity Class, to model Customer Rewards |  |
| * rewardPoints | Stores the reward points |  |
| **Departments** | Entity Class, to model departments |  |
| * deptID | Identifying attribute | Format: “D” prefix followed by 3 digits. |
| * deptName | Department Name (e.g., Finance, |  |
| **Discounts** | Entity Class, to model discounts |  |
| * discountID | Identifying attribute | Format: “D” prefix followed by 4 digits. |
| * disCode | Discount Code |  |
| * creditPercent | Percentage of credit amount | Number |
| * description | Information about the discount |  |
| * validFrom | Start of discount period | Date |
| * validTo | End of discount period | Date |
| **Employees** | Entity Class, to model employees |  |
| * empID | Identifying attribute | Format: “EMP” prefix followed by 7 digits. |
| * fName | First name of employee |  |
| * lName | Last name of employee |  |
| * salaryGrade | Salary level for payroll (A, B, C, D) |  |
| * dateOfJoining | Date of joining the company |  |
| * ssn | SSN of employee |  |
| * dateOfLeaving | Date of leaving the company |  |
| * gender | Gender (Male, Female) |  |
| * workLocation | Offlice type such as Office, Branch Office, Headquarter |  |
| * managerID | ManagerID of the manager assigned to employee |  |
| * departmentID | Department ID of the employee’s department |  |
| * email | Work email of employee |  |
| * street | Employee address |  |
| * city | City where employee stays |  |
| * state | State of the address |  |
| * zipcode | Address’s zipcode |  |
| * dob | Date of Birth |  |
| * age | Derived Attribute |  |
| **EmployeeAuthentication History** | Weak Entity Class, to model employee’s login/authentication history |  |
| * loggedInTime | Identifying attribute, stores the timestamp when employee is authenticated to log in |  |
| * loggedOutTime | Stores the timestamp when employee logs out |  |
| * empAuthLongitude | Stores the latitude co-ordinates of the place from where employee is logging in |  |
| * empAuthLatitude | Stores the longitude co-ordinates of the place from where employee is logging in |  |
| **Payment** | Entity Class, to model the payments made by customers |  |
| * paymentID | Identifying attribute | Format: “PM” prefix followed by 3 digits. |
| * amount | amount (in $) paid by the customer |  |
| * paymentTime | Time when payment was made |  |
| * typeOfPayment | Category of payment | Can hold only 2 values, ‘D’ or ‘S’ which stands for defaulted/excess charge and subscription respectively |
| **Payroll History** | Weak Entity Class, to model payment history of employees |  |
| * paymentid | Identify Attribute | Format: “P” prefix followed by 3 digits. |
| * yearofpayment | Year for which employee got the salary |  |
| * monthOfPayment | Month for which employee got the salary |  |
| **Payrolls** | Entity Class, to model salary grades |  |
| * payrollID | Identifying attribute | Format: “PAY” prefix followed by 3 digits. |
| * baseSalary | Base salary a grade |  |
| * salaryGrade | Salary Grade(ex: A, B, C) | Should be a Unique value |
| * taxPercent | Percent of tax imposed on the base salary |  |
| * taxAmount | Tax amount (in $), calculated from base salary and tax percent |  |
| * incentivePercent | Percent of incentives per grade |  |
| * incentiveAmount | Amount(in $) of incentive per grade based on base salary and incentive percentage |  |
| * totalSalary | Net salary(in $), considering incentives and taxes |  |
| **Purchase History** | Weak Entity Class, to model purchase history of spare parts |  |
| * purchaseID | Identifying attribute | Format: “PU” prefix followed by 3 digits. |
| * pDate | Date of purchase |  |
| * quantity | Quantity of spare parts purchased |  |
| * billingAmt | Total amount(in $) for the purchase |  |
| **Ratings** | Entity Class, to model ratings given by customers |  |
| * reviewID | Identifying attribute | Format: “R” prefix followed by 6 digits. |
| * userID | Customer’s user ID |  |
| * datePosted | Date of review |  |
| * bikeID | Bike ID for which the review is posted |  |
| * reviewText | Review text description |  |
| **Refunds** | Weak Entity Class, to model refunds related to payments made by customers |  |
| * refundID | Identifying attribute | Format: “RF” prefix followed by 3 digits. |
| * amount | Refund amount (in $) | Cannot be negative |
| **Spare Parts** | Entity Class, to model spare parts required for maintenance |  |
| * spartID | Identifying attribute | Format: “S” prefix followed by 3 digits. |
| * spName | Spare part name |  |
| * spWeight | Spare part weight (in kilograms) |  |
| * spColor | Spare part color |  |
| * spBrand | Brand of the spare part |  |
| * price | Price (in $) for the spare part | Cannot be negative |
| **Station Maintenance** | Weak Entity Class, to model station maintenances |  |
| * sMaintenanceID | Identifying Attribute | Format: “SM” prefix followed by 3 digits. |
| * sStartTime | Station maintenance start time |  |
| * sEndTime | Station maintenance end time |  |
| * sTypeOfService | Type of maintenance (regular or repair) | Can hold only 2 values, ‘Repair’ or ‘Regular Maintenance’ |
| **Stations** | Entity Class, to model stations |  |
| * stationID | Identifying attribute | Format: “S” prefix followed by 3 digits. |
| * sName | Station Name |  |
| * city | City in which the station exists |  |
| * bikecapacity | Total bikes that station can accommodate |  |
| * availablebikes | Number of bikes available for pick up by customers |  |
| * maintenancebikes | Number of bikes which are down for maintenace |  |
| * freebikespaces | Number of free spaces available at the station for bike drop off |  |
| * stationlatitude | Station location latitude co-ordinate |  |
| * stationlongitude | Station location longitude co-ordinate |  |
| **Subscription Models** | Entity Class, to model subscription plans |  |
| * planID | Identifying attribute | Format: “P” prefix followed by 3 digits. |
| * title | Title of the subscription plan |  |
| * basePrice | Price (in $) to subscribe a plan | Cannot be nagative |
| * description | Description of the subscription |  |
| **Subscription Status** | Weak Entity Class, to model subscription history of a customer |  |
| * joinedDate | Identifying attribute | Date with timestamp |
| * endDate |  | Date with timestamp |
| **Trips** | Entity Class, to model spare parts required for maintenance |  |
| * tripID | Identifying attribute | Format: “TRIP” prefix followed by 3 digits. |
| * startTime | Trip start time |  |
| * endTime | Trip end time |  |
| * excessCharge | Charge (in $) if time exceeds after 30 mins before end time logged | Cannot be negative |
| **Vendors** |  |  |
| * vendorID | Identifying attribute | Format: “VEN” prefix followed by 6 digits. |
| * vPhone | Vendor’s phone nymber |  |
| * vName | Vendor’s name |  |
| * vEmail | Vendor’s email |  |
| * street | Street name for address |  |
| * city | City for address |  |
| * state | State for address |  |
| * zipcode | Zipcode for address |  |
| * country | Country of address |  |
| **RELATIONSHIPS** | | |
| hasEmerContact | Relationship that models the emergency contact that the employee has |  |
| Works for | Relationship that models the department for which employees works |  |
| empRequires | Relationship that models the authentication of employees |  |
| supervises | Relationship that models managers supervising the employees |  |
| custRequires | Relationship that models the authentication of employees |  |
| worksOn | Relationship that models the complaint handled by On Call support employees |  |
| provides | Relationship that models ratings given by customers |  |
| Registers | Relationship that models complaints registerd by customers |  |
| earns | Relationship that models loyalty points earned by customers |  |
| Initiate refund | Relationship that models refunds initiated on payments |  |
| Applied on | Relationship that models discounts applied on payments |  |
| makes | Relationship that models payments made by customers |  |
| Gets paid | Relationship that models salary given to employees |  |
| Subscribed to | Relationship that models subscription plan customers can have |  |
| canRequest | Relationship that models trips requested by customers |  |
| pickUpStation | Relationship that models pick up station |  |
| dropOffStation | Relationship that models drop off station |  |
| servicesStation | Relationship that models maintenance service of stations |  |
| servicesBikes | Relationship that models maintenance service of bikes |  |
| contains | Relationship that models the stations containing bikes |  |
| includes | Relationship that models trips which includes bike |  |
| canSupply | Relationship that models bike models supplied by vendors | Contraining relationship |
| supplies | Relationship that models bike supplied by vendors | Constrained relationship |
| purchases | Relationship that models spare parts purchased from the vendors |  |

## 2.2 ER DIAGRAM



**CHAPTER 3**

# RELATIONAL SCHEMA

## 3.1 Entity Classes

1. AUTHENTICATION (AUTHID, PASSWORD, EMAIL, SECURITYQUESTION, SECURITYANSWER, TYPE)
2. BIKE\_MODELS (MODELID, MODELNAME, TYPE, DESCRIPTION, MODELCOST, VENDORID)

Foreign Key VENDORID references VENDORS (VENDORID)

1. BIKES (BIKEID, BIKENAME, YEAROFPURCHASE, MODELID, STATIONID, MONTHOFPURCHASE)

Foreign Key MODELID references BIKE\_MODELS (MODELID)

Foreign Key STATIONID references STATIONS (STATIONID)

1. COMPLAINTS (COMPLAINTID, TYPEOFCOMPLAINT, STATUS, RAISEDTIME, RESOLVETIME, CUSTID, EMPID)

Foreign Key CUSTID references CUSTOMERS (CUSTID)

Foreign Key EMPID references EMPLOYEES (EMPID)

1. CUSTOMER\_CONTACTS (CUSTID, MOBILENUM)

Foreign Key CUSTID references CUSTOMERS(custID)

1. CUSTOMERS (CUSTID, FNAME, LNAME, AGE, DOB, CUSTSINCE, STREET, CITY, STATE, ZIPCODE, EMAIL, ACTIVESTATUS, AUTHID)

Foreign Key AUTHID references AUTHENTICATION(AUTHID)

1. DISCOUNTS (DISCOUNTID, DISCODE, CREDITPERCENT, DESCRIPTION, VALIDFROM, VALIDTO)
2. DEPARTMENTS (DEPTID, DEPTNAME)
3. EMPLOYEES(EMPID, FNAME, LNAME, SALARYGRADE, DATEOFLEAVING, SSN, DATEOFJOINING, GENDER, WORKLOCATION, MANAGERID, DEPARTMENTID, AUTHID, EMAIL, STREET, CITY, STATE, ZIPCODE, DOB, AGE, ISMANAGER, ISONCALLSUPPORT, ISMECHANIC)

Foreign Key AUTHID references AUTHENTICATION(AUTHID)

Foreign Key SALARYGRADE references PAYROLL(SALARYGRADE)

Foreign Key MANAGERID references EMPLOYEES (EMPID)

1. PAYMENT (PAYMENTID, AMOUNT, PAYMENTTIME, TYPEOFPAYMENT, CUSTID)

Foreign key CUSTID references CUSTOMERS (CUSTID)

1. PAYROLLS (PAYROLLID, BASESALARY, SALARYGRADE, TAXPERCENT, TAXAMOUNT, INCENTIVEPERCENT, INCENTIVEAMOUNT, TOTALSALARY)
2. RATINGS (REVIEWID, USERID, DATEPOSTED, BIKEID, REVIEWTEXT, CUSTID)

Foreign Key CUSTID references CUSTOMERS (CUSTID)

1. SPARE\_PARTS (SPARTID, SPNAME, SPWEIGHT, SPCOLOR, SPBRAND, PRICE, VENDORID)

Foreign Key VENDORID references VENDORS (VENDORID)

1. STATIONS (STATIONID, SNAME, CITY, BIKECAPACITY, AVAILABLEBIKES, MAINTENANCEBIKES, FREEBIKESPACES, STATIONLATITUDE, STATIONLONGITUDE)
2. SUBSCRIPTION\_MODELS (PLANID, TITLE, BASEPRICE, DESCRIPTION)

1. TRIPS (TRIPID, STARTTIME, ENDTIME, CUSTID, PICKUPSTAIONID, DROPOFFSTATIONID, BIKEID, EXCESSCHARGE)

Foreign Key PICKUPSTATIONID references STATIONS (STATIONID)

Foreign Key DROPOFFSTATIONID references STATIONS (STATIONID)

Foreign Key CUSTID references CUSTOMERS (CUSTID)

Foreign Key BIKEID references BIKES (BIKEID)

1. VENDORS (VENDORID, VPHONE, VNAME, VEMAIL, STREET, CITY, STATE, ZIPCODE, COUNTRY)

## 3.2 Relationship Classes & Weak Entity Class

1. BIKE\_MAINTENANCE (MAINTENCEID, BIKEID, M\_START\_TIME, M\_END\_TIME, TYPEOFSERVICE)

Foreign key BIKEID references BIKES (BIKEID)

1. CUST\_REWARDS (CUSTID, REWARDPOINTS)

Foreign key CUSTID references CUSTOMERS (CUSTID)

1. CUSTOMER\_AUTHENTICATION\_HISTORY (AUTHID, CUSTID, LOGGEDINTIME, LOGGEDOUTTIME, CUSTAUTHLONGITUDE, CUSTAUTHLATITUDE)

Foreign Key AUTHID password references AUTHENTICATION (AUTHID)

Foreign Key CUSTID password references CUSTOMERS (CUSTID)

1. EMP\_CONTACTS (EMPID, MOBILENUM)

Foreign Key EMPID references EMPLOYEES (EMPID)

1. EMP\_EMERGENCY\_CONTACTS (EMPID, EMERGENCYCONTACTNAME, EMERGENCYCONTACTNUM)

Foreign Key EMPID references EMPLOYEES (EMPID)

1. EMPLOYEE\_AUTHENTICATION\_HISTORY (AUTHID, EMPID, LOGGEDINTIME, LOGGEDOUTTIME, EMPAUTHLONGITUDE, EMPAUTHLATITUDE)

Foreign Key AUTHID references AUTHENTICATION (AUTHID)

Foreign Key EMPID references EMPLOYEES (EMPID)

1. MONTHLY\_REVENUE (REVID, YEAR, MONTH, CUSTPAYMENTS, EMPSALARY, REFUNDAMOUNT, SPAREPARTSPURCHASES, BIKESPURCHASES, MONTHTOTAL)
2. PAYMENTS\_DISCOUNTS (DISCOUNTID, PAYMENTID)

Foreign Key DISCOUNTID references D ISCOUNTS (DISCOUNTID)

Foreign Key PAYMENTID references PAYMENT (PAYMENTID)

1. PAYROLL\_HISTORY (PAYROLLID, EMPID, PAYMENTID, YEAROFPAYMENT, MONTHOFPAYMENT)

Foreign key EMPID references EMPLOYEES (EMPID)

Foreign key PAYROLLID references PAYROLLS (PAYROLLID)

1. PURCHASE\_HISTORY(VENDORID, SPARTID, PURCHASEID, PDATE, QUANTITY, BILLINGAMT)

Foreign Key VENDORID references VENDORS (VENDORID)

Foreign Key SPARTID references SPARE\_PARTS (SPARTID)

1. REFUNDS (PAYMENTID, REFUNDID, AMOUNT)

Foreign Key PAYMENTID references PAYMENTS (PAYMENTID)

1. STATION\_MAINTENANCE (MAINTENCEID, STATIONID, M\_START\_TIME, M\_END\_TIME, TYPEOFSERVICE)

Foreign key STATIONID references STATIONS (STATIONID)

1. SUBSCRIPTION\_STATUS (CUSTID, PLANID, JOINEDDATE, ENDDATE)

Foreign Key CUSTID references CUSTOMERS (CUSTID)

Foreign Key PLANID references SUBSCRIPTION\_MODELS (PLANID)

## 3.3 Data Dictionary (for Relational Design)

|  |  |  |
| --- | --- | --- |
| **Schema Construct** | **Data type** | **Constraint** |
| **Authentication** | Entity Class, to model authentication while logging |  |
| * AUTHID | VARCHAR2(255 BYTE) | **Primary Key** |
| * PASSWORD | VARCHAR2(255 BYTE) |  |
| * EMAIL | VARCHAR2(255 BYTE) |  |
| * SECURITYQUESTION | VARCHAR2(255 BYTE) |  |
| * SECURITYANSWER | VARCHAR2(255 BYTE) |  |
| * TYPE | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Bikes** | Entity Class, to model bikes |  |
| * BIKEID | VARCHAR2(255 BYTE) | **Primary Key** |
| * BIKENAME | VARCHAR2(255 BYTE) |  |
| * YEAROFPURCHASE | NUMBER(4,0) |  |
| * MODELID | VARCHAR2(255 BYTE) | **Foreign Key to Bike\_Models** |
| * STATIONID | VARCHAR2(255 BYTE) | **Foreign Key to Stations** |
| * MONTHOFPURCHASE | NUMBER(2,0) |  |
|  |  |  |
| **Bike\_Maintenance** | Weak Entity Class, to model bike maintenances |  |
| * MAINTENCEID | VARCHAR2(255 BYTE) | **Primary Key** |
| * BIKEID | VARCHAR2(255 BYTE) | **Primary Key, Foreign key BIKEID references BIKES (BIKEID)** |
| * M\_START\_TIME | TIMESTAMP(6) |  |
| * M\_END\_TIME | TIMESTAMP(6) |  |
| * TYPEOFSERVICE | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Bike\_Models** | Entity Class, to model bike models |  |
| * MODELID | VARCHAR2(255 BYTE) | **Primary Key** |
| * MODELNAME | VARCHAR2(255 BYTE) |  |
| * TYPE | VARCHAR2(255 BYTE) |  |
| * DESCRIPTION | VARCHAR2(255 BYTE) |  |
| * MODELCOST | NUMBER |  |
| * VENDORID | VARCHAR2(255 BYTE) | **Foreign Key VENDORID references VENDORS (VENDORID)** |
|  |  |  |
| **Complaints** | Entity Class, to model customer complaints |  |
| * COMPLAINTID | VARCHAR2(255 BYTE) | **Primary Key** |
| * TYPEOFCOMPLAINT | VARCHAR2(255 BYTE) |  |
| * STATUS | CHAR(1 BYTE) |  |
| * RAISEDTIME | TIMESTAMP(6) |  |
| * RESOLVETIME | TIMESTAMP(6) |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Foreign Key CUSTID references CUSTOMERS (CUSTID)** |
| * EMPID | VARCHAR2(255 BYTE) | **Foreign Key EMPID references EMPLOYEES (EMPID)** |
|  |  |  |
| **Customer\_Authentication\_History** | Weak Entity Class, to model customer’s login/authentication history |  |
| * AUTHID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key AUTHID password references AUTHENTICATION (AUTHID)** |
| * CUSTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key CUSTID password references CUSTOMERS (CUSTID)** |
| * LOGGEDINTIME | TIMESTAMP(6) | **Primary Key** |
| * LOGGEDOUTTIME | TIMESTAMP(6) |  |
| * CUSTAUTHLONGITUDE | VARCHAR2(255 BYTE) |  |
| * CUSTAUTHLATITUDE | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Customers** | Entity Class, to model customers |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Primary Key** |
| * FNAME | VARCHAR2(255 BYTE) |  |
| * LNAME | VARCHAR2(255 BYTE) |  |
| * AGE | NUMBER(38,0) |  |
| * DOB | DATE |  |
| * CUSTSINCE | DATE |  |
| * STREET | VARCHAR2(255 BYTE) |  |
| * CITY | VARCHAR2(255 BYTE) |  |
| * STATE | VARCHAR2(255 BYTE) |  |
| * ZIPCODE | NUMBER(5,0) |  |
| * EMAIL | VARCHAR2(255 BYTE) |  |
| * ACTIVESTATUS | CHAR(1 BYTE) |  |
| * AUTHID | VARCHAR2(255 BYTE) | **Foreign Key AUTHID password references AUTHENTICATION (AUTHID)** |
|  |  |  |
| **Cust\_Rewards** | Weak Entity Class, to model Customer Rewards |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key CUSTID password references CUSTOMERS (CUSTID)** |
| * REWARDPOINTS | NUMBER |  |
|  |  |  |
| **Cust\_Contacts** | Entity Class, to model customer multi-valued phone numbers |  |
| CUSTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key CUSTID password references CUSTOMERS (CUSTID)** |
| MOBILENUM | NUMBER(10,0) |  |
|  |  |  |
| **Departments** | Entity Class, to model departments |  |
| * DEPTID | VARCHAR2(255 BYTE) | **Primary Key** |
| * DEPTNAME | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Discounts** | Entity Class, to model discounts |  |
| * DISCOUNTID | VARCHAR2(255 BYTE) | **Primary Key** |
| * DISCODE | VARCHAR2(255 BYTE) |  |
| * CREDITPERCENT | NUMBER |  |
| * DESCRIPTION | VARCHAR2(255 BYTE) |  |
| * VALIDFROM | DATE |  |
| * VALIDTO | DATE |  |
|  |  |  |
| **Emp\_Contacts** | Weak Entity Class, to model employees multi-valued phone numbers |  |
| EMPID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key EMPID references EMPLOYEES (EMPID)** |
| MOBILENUM | NUMBER(10,0) | **Primary Key** |
|  |  |  |
| **Employees** | Entity Class, to model employees |  |
| * EMPID | VARCHAR2(255 BYTE) | **Primary Key** |
| * FNAME | VARCHAR2(255 BYTE) |  |
| * LNAME | VARCHAR2(255 BYTE) |  |
| * SALARYGRADE | VARCHAR2(255 BYTE) | **Foreign Key SALARYGRADE references PAYROLL(SALARYGRADE)** |
| * DATEOFLEAVING | DATE |  |
| * SSN | NUMBER(9,0) |  |
| * DATEOFJOINING | DATE |  |
| * GENDER | VARCHAR2(255 BYTE) |  |
| * WORKLOCATION | VARCHAR2(255 BYTE) |  |
| * MANAGERID | VARCHAR2(255 BYTE) | **Foreign Key MANAGERID references EMPLOYEES (EMPID)** |
| * DEPARTMENTID | VARCHAR2(255 BYTE) |  |
| * AUTHID | VARCHAR2(255 BYTE) | **Foreign Key AUTHID references AUTHENTICATION(AUTHID)** |
| * EMAIL | VARCHAR2(255 BYTE) |  |
| * STREET | VARCHAR2(255 BYTE) |  |
| * CITY | VARCHAR2(255 BYTE) |  |
| * STATE | VARCHAR2(255 BYTE) |  |
| * ZIPCODE | NUMBER(5,0) |  |
| * DOB | DATE |  |
| * AGE | NUMBER |  |
| * ISMANAGER | CHAR(1 BYTE) |  |
| * ISONCALLSUPPORT | CHAR(1 BYTE) |  |
| * ISMECHANIC | CHAR(1 BYTE) |  |
|  |  |  |
| **Emp\_Emergency\_Contacts** | Weak Entity Class, to model emergency contacts for employees |  |
| * EMPID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key EMPID references EMPLOYEES (EMPID)** |
| * EMERGENCYCONTACTNAME | VARCHAR2(255 BYTE) |  |
| * EMERGENCYCONTACTNUM | NUMBER(10,0) |  |
|  |  |  |
| **Employee\_Authentication\_History** | Weak Entity Class, to model employee’s login / authentication history |  |
| * AUTHID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key AUTHID references AUTHENTICATION(AUTHID)** |
| * EMPID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key EMPID references EMPLOYEES (EMPID)** |
| * LOGGEDINTIME | TIMESTAMP(6) | **Primary Key** |
| * LOGGEDOUTTIME | TIMESTAMP(6) |  |
| * EMPAUTHLONGITUDE | VARCHAR2(255 BYTE) |  |
| * EMPAUTHLATITUDE | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **MONTHLY\_REVENUE** | Weak Entity Class, Relation created for procedure data |  |
| * REVID | VARCHAR2(255 BYTE) | **Primary Key** |
| * YEAR | NUMBER(4,0) |  |
| * MONTH | NUMBER(2,0) |  |
| * CUSTPAYMENTS | NUMBER(10,2) |  |
| * EMPSALARY | NUMBER(10,2) |  |
| * REFUNDAMOUNT | NUMBER(10,2) |  |
| * SPAREPARTSPURCHASES | NUMBER(10,2) |  |
| * BIKESPURCHASES | NUMBER(10,2) |  |
| * MONTHTOTAL | NUMBER(10,2) |  |
|  |  |  |
| **Payment** | Entity Class, to model the payments made by customers |  |
| * PAYMENTID | VARCHAR2(255 BYTE) | **Primary Key** |
| * AMOUNT | NUMBER |  |
| * PAYMENTTIME | DATE |  |
| * TYPEOFPAYMENT | CHAR(1 BYTE) |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Foreign key CUSTID references CUSTOMERS (CUSTID)** |
|  |  |  |
| **Payments\_Discounts** | Weak Entity Class, to model the payment discounts |  |
| DISCOUNTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key DISCOUNTID references DISCOUNTS (DISCOUNTID)** |
| PAYMENTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key PAYMENTID references PAYMENT (PAYMENTID)** |
|  |  |  |
| **Payroll\_History** | Weak Entity Class, to model payment history of employees |  |
| * PAYROLLID | VARCHAR2(255 BYTE) | **Primary Key** |
| * EMPID | VARCHAR2(255 BYTE) | **Primary Key, Foreign key EMPID references EMPLOYEES (EMPID)** |
| * PAYMENTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign key PAYROLLID references PAYROLLS (PAYROLLID)** |
| * YEAROFPAYMENT | NUMBER(4,0) |  |
| * MONTHOFPAYMENT | NUMBER(2,0) |  |
|  |  |  |
| **Payrolls** | Entity Class, to model salary grades |  |
| * PAYROLLID | VARCHAR2(255 BYTE) | **Primary Key** |
| * BASESALARY | NUMBER |  |
| * SALARYGRADE | VARCHAR2(255 BYTE) |  |
| * TAXPERCENT | NUMBER |  |
| * TAXAMOUNT | NUMBER |  |
| * INCENTIVEPERCENT | NUMBER |  |
| * INCENTIVEAMOUNT | NUMBER |  |
| * TOTALSALARY | NUMBER |  |
|  |  |  |
| **Purchase\_History** | Weak Entity Class, to model purchase history of spare parts |  |
| * VENDORID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key VENDORID references VENDORS (VENDORID)** |
| * SPARTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key SPARTID references SPARE\_PARTS (SPARTID)** |
| * PURCHASEID | VARCHAR2(255 BYTE) | **Primary Key** |
| * PDATE | DATE |  |
| * QUANTITY | NUMBER(38,0) |  |
|  |  |  |
| **Ratings** | Entity Class, to model ratings given by customers |  |
| * REVIEWID | VARCHAR2(255 BYTE) | **Primary Key** |
| * USERID | VARCHAR2(255 BYTE) |  |
| * DATEPOSTED | DATE |  |
| * BIKEID | VARCHAR2(255 BYTE) |  |
| * REVIEWTEXT | VARCHAR2(255 BYTE) |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Foreign Key CUSTID references CUSTOMERS (CUSTID)** |
|  |  |  |
| **Refunds** | Weak Entity Class, to model refunds related to payments made by customers |  |
| * REFUNDID | VARCHAR2(255 BYTE) | **Primary Key** |
| * PAYMENTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key PAYMENTID references PAYMENTS (PAYMENTID)** |
| * AMOUNT | NUMBER(10,2) |  |
|  |  |  |
| **Spare\_Parts** | Entity Class, to model spare parts required for maintenance |  |
| * SPARTID | VARCHAR2(255 BYTE) | **Primary Key** |
| * SPNAME | VARCHAR2(255 BYTE) |  |
| * SPWEIGHT | NUMBER(10,2) |  |
| * SPCOLOR | VARCHAR2(255 BYTE) |  |
| * SPBRAND | VARCHAR2(255 BYTE) |  |
| * PRICE | NUMBER(10,2) |  |
| * VENDORID | VARCHAR2(255 BYTE) | **Foreign Key VENDORID references VENDORS (VENDORID)** |
|  |  |  |
| **Station\_Maintenance** | Weak Entity Class, to model station maintenances |  |
| * MAINTENCEID | VARCHAR2(255 BYTE) | **Primary Key** |
| * STATIONID | VARCHAR2(255 BYTE) | **Primary Key, Foreign key STATIONID references STATIONS (STATIONID)** |
| * M\_START\_TIME | TIMESTAMP(6) |  |
| * M\_END\_TIME | TIMESTAMP(6) |  |
| * TYPEOFSERVICE | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Stations** | Entity Class, to model stations |  |
| * STATIONID | VARCHAR2(255 BYTE) | **Primary Key** |
| * SNAME | VARCHAR2(255 BYTE) |  |
| * CITY | VARCHAR2(255 BYTE) |  |
| * BIKECAPACITY | NUMBER(38,0) |  |
| * AVAILABLEBIKES | NUMBER(38,0) |  |
| * MAINTENANCEBIKES | NUMBER(38,0) |  |
| * FREEBIKESPACES | NUMBER(38,0) |  |
| * STATIONLATITUDE | VARCHAR2(255 BYTE) |  |
| * STATIONLONGITUDE | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Subscription\_Models** | Entity Class, to model subscription plans |  |
| * PLANID | VARCHAR2(255 BYTE) | **Primary Key** |
| * TITLE | VARCHAR2(255 BYTE) |  |
| * BASEPRICE | NUMBER |  |
| * DESCRIPTION | VARCHAR2(255 BYTE) |  |
|  |  |  |
| **Subscription\_Status** | Weak Entity Class, to model subscription history of a customer |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key CUSTID references CUSTOMERS (CUSTID)** |
| * PLANID | VARCHAR2(255 BYTE) | **Primary Key, Foreign Key PLANID references SUBSCRIPTION\_MODELS (PLANID)** |
| * JOINEDDATE | DATE | **Primary Key** |
| * ENDDATE | DATE |  |
|  |  |  |
| **Trips** | Entity Class, to model spare parts required for maintenance |  |
| * TRIPID | VARCHAR2(255 BYTE) | **Primary Key** |
| * STARTTIME | TIMESTAMP(6) |  |
| * ENDTIME | TIMESTAMP(6) |  |
| * CUSTID | VARCHAR2(255 BYTE) | **Foreign Key CUSTID references CUSTOMERS (CUSTID)** |
| * PICKUPSTATIONID | VARCHAR2(255 BYTE) | **Foreign Key PICKUPSTATIONID references STATIONS (STATIONID)** |
| * DROPOFFSTATIONID | VARCHAR2(255 BYTE) | **Foreign Key DROPOFFSTATIONID references STATIONS (STATIONID)** |
| * BIKEID | VARCHAR2(255 BYTE) | **Foreign Key BIKEID references BIKES (BIKEID)** |
| * EXCESSCHARGE | NUMBER(10,2) |  |
|  |  |  |
| **Vendors** | Entity Class, to model vendors |  |
| * VENDORID | VARCHAR2(255 BYTE) | **Primary Key** |
| * VPHONE | VARCHAR2(255 BYTE) |  |
| * VNAME | VARCHAR2(255 BYTE) |  |
| * VEMAIL | VARCHAR2(255 BYTE) |  |
| * STREET | VARCHAR2(255 BYTE) |  |
| * CITY | VARCHAR2(255 BYTE) |  |
| * STATE | VARCHAR2(255 BYTE) |  |
| * ZIPCODE | VARCHAR2(255 BYTE) |  |
| * COUNTRY | VARCHAR2(255 BYTE) |  |

## 3.4 SQL Create Statements

1. **AUTHENTICATION table**

CREATE TABLE "MIS531GROUPS1H"."AUTHENTICATION"

( "AUTHID" VARCHAR2(255 BYTE),

"PASSWORD" VARCHAR2(255 BYTE),

"EMAIL" VARCHAR2(255 BYTE),

"SECURITYQUESTION" VARCHAR2(255 BYTE),

"SECURITYANSWER" VARCHAR2(255 BYTE),

"TYPE" VARCHAR2(255 BYTE),

PRIMARY KEY ("AUTHID"),

CONSTRAINT "CHECK\_TYPE" CHECK ( type IN ( 'Customer','Employee' ) )

);

1. **BIKE\_MAINTENANCE table:**

CREATE TABLE "MIS531GROUPS1H"."BIKE\_MAINTENANCE"

( "MAINTENCEID" VARCHAR2(255 BYTE),

"BIKEID" VARCHAR2(255 BYTE),

"M\_START\_TIME" TIMESTAMP (6),

"M\_END\_TIME" TIMESTAMP (6),

"TYPEOFSERVICE" VARCHAR2(255 BYTE),

PRIMARY KEY ("MAINTENCEID", "BIKEID"),

CONSTRAINT "CHK\_BIKE\_SERVICE" CHECK (typeOfService IN ('Repair', 'Regular Maintenance')),

FOREIGN KEY ("BIKEID")

REFERENCES "MIS531GROUPS1H"."BIKES" ("BIKEID")

);

1. **BIKE MODELS Table**

CREATE TABLE "MIS531GROUPS1H"."BIKE\_MODELS"

( "MODELID" VARCHAR2(255 BYTE),

"MODELNAME" VARCHAR2(255 BYTE),

"TYPE" VARCHAR2(255 BYTE),

"DESCRIPTION" VARCHAR2(255 BYTE),

"MODELCOST" NUMBER,

"VENDORID" VARCHAR2(255 BYTE),

PRIMARY KEY ("MODELID"),

CONSTRAINT "FK\_VENDOR\_ID" FOREIGN KEY ("VENDORID")

REFERENCES "MIS531GROUPS1H"."VENDORS" ("VENDORID") ON DELETE SET NULL

);

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."BIKEMODEL\_ID\_GENERATOR" BEFORE

INSERT ON bike\_models

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(3);

v\_newbikemodelid bike\_models.modelid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(bikemodel\_seq.NEXTVAL)),

3,

'0')

INTO v\_nextval

FROM

dual;

v\_newbikemodelid := 'M' || v\_nextval;

:new.modelid := v\_newbikemodelid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."BIKEMODEL\_ID\_GENERATOR" ENABLE;

1. **BIKES tables:**

CREATE TABLE "MIS531GROUPS1H"."BIKES"

( "BIKEID" VARCHAR2(255 BYTE),

"BIKENAME" VARCHAR2(255 BYTE),

"YEAROFPURCHASE" NUMBER(4,0),

"MODELID" VARCHAR2(255 BYTE),

"STATIONID" VARCHAR2(255 BYTE),

"MONTHOFPURCHASE" NUMBER(2,0),

PRIMARY KEY ("BIKEID"),

CONSTRAINT "FK\_MODEL\_ID" FOREIGN KEY ("MODELID")

REFERENCES "MIS531GROUPS1H"."BIKE\_MODELS" ("MODELID") ON DELETE SET NULL,

CONSTRAINT "FK\_STATION\_ID" FOREIGN KEY ("STATIONID")

REFERENCES "MIS531GROUPS1H"."STATIONS" ("STATIONID"));

**5.COMPLAINTS table:**

CREATE TABLE "MIS531GROUPS1H"."COMPLAINTS"

( "COMPLAINTID" VARCHAR2(255 BYTE),

"TYPEOFCOMPLAINT" VARCHAR2(255 BYTE),

"STATUS" CHAR(1 BYTE),

"RAISEDTIME" TIMESTAMP (6),

"RESOLVETIME" TIMESTAMP (6),

"CUSTID" VARCHAR2(255 BYTE),

"EMPID" VARCHAR2(255 BYTE),

PRIMARY KEY ("COMPLAINTID"),

CONSTRAINT "CHECK\_TYPEOFCOMPLAINT" CHECK ( typeofcomplaint IN ( 'Bike', 'Station', 'Payment' ) ) ENABLE,

CONSTRAINT "CHECK\_STATUS" CHECK ( status IN ( 'N', 'P', 'R' ) ),

FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID"),

FOREIGN KEY ("EMPID")

REFERENCES "MIS531GROUPS1H"."EMPLOYEES" ("EMPID"));

1. **CUST\_REWARD**

CREATE TABLE "MIS531GROUPS1H"."CUST\_REWARDS"

( "CUSTID" VARCHAR2(255 BYTE),

"REWARDPOINTS" NUMBER,

PRIMARY KEY ("CUSTID"),

CONSTRAINT "FK\_CUSTID" FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID")

);

1. **CUSTOMER\_AUTHENTICATION\_HISTORY Table**

CREATE TABLE "MIS531GROUPS1H"."CUSTOMER\_AUTHENTICATION\_HISTORY"

( "AUTHID" VARCHAR2(255 BYTE),

"CUSTID" VARCHAR2(255 BYTE),

"LOGGEDINTIME" TIMESTAMP (6) NOT NULL ENABLE,

"LOGGEDOUTTIME" TIMESTAMP (6),

"CUSTAUTHLONGITUDE" VARCHAR2(255 BYTE),

"CUSTAUTHLATITUDE" VARCHAR2(255 BYTE),

CONSTRAINT "PK\_CUST\_AUTH\_HISTORY" PRIMARY KEY ("AUTHID", "CUSTID"),

CONSTRAINT "CUST\_UNIQUE" UNIQUE ("LOGGEDINTIME"),

CONSTRAINT "FK\_CUST\_AUTH\_ID" FOREIGN KEY ("AUTHID")

REFERENCES "MIS531GROUPS1H"."AUTHENTICATION" ("AUTHID") ENABLE,

CONSTRAINT "FK\_CUST\_AUTH\_HISTORY\_ID" FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID") ENABLE

);

1. **CUSTOMER\_CONTACTS tables**

CREATE TABLE "MIS531GROUPS1H"."CUSTOMER\_CONTACTS"

( "CUSTID" VARCHAR2(255 BYTE),

"MOBILENUM" NUMBER(10,0),

PRIMARY KEY ("CUSTID", "MOBILENUM"),

FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID") ON DELETE CASCADE ENABLE

);

1. **CUSTOMERS table:**

CREATE TABLE "MIS531GROUPS1H"."CUSTOMERS"

( "CUSTID" VARCHAR2(255 BYTE),

"FNAME" VARCHAR2(255 BYTE),

"LNAME" VARCHAR2(255 BYTE),

"AGE" NUMBER(\*,0),

"DOB" DATE,

"CUSTSINCE" DATE DEFAULT sysdate,

"STREET" VARCHAR2(255 BYTE),

"CITY" VARCHAR2(255 BYTE),

"STATE" VARCHAR2(255 BYTE),

"ZIPCODE" NUMBER(5,0),

"EMAIL" VARCHAR2(255 BYTE),

"ACTIVESTATUS" CHAR(1 BYTE),

"AUTHID" VARCHAR2(255 BYTE),

PRIMARY KEY ("CUSTID"),

CONSTRAINT "CHECK\_ISACTIVE" CHECK ( activestatus IN ( 'Y', 'N' ) ) ENABLE,

FOREIGN KEY ("AUTHID")

REFERENCES "MIS531GROUPS1H"."AUTHENTICATION" ("AUTHID") ON DELETE SET NULL ENABLE);

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."CUST\_ID\_GENERATOR" BEFORE

INSERT ON customers

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(6);

v\_newcustid customers.custid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(cust\_seq.NEXTVAL)),

6,

'0')

INTO v\_nextval

FROM

dual;

v\_newcustid := 'CUST' || v\_nextval;

:new.custid := v\_newcustid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."CUST\_ID\_GENERATOR" ENABLE;

1. **DEPARTMENTS table:**

CREATE TABLE "MIS531GROUPS1H"."DEPARTMENTS"

( "DEPTID" VARCHAR2(255 BYTE),

"DEPTNAME" VARCHAR2(255 BYTE),

PRIMARY KEY ("DEPTID")

);

1. **DISCOUNTS table:**

CREATE TABLE "MIS531GROUPS1H"."DISCOUNTS"

( "DISCOUNTID" VARCHAR2(255 BYTE),

"DISCODE" VARCHAR2(255 BYTE),

"CREDITPERCENT" NUMBER,

"DESCRIPTION" VARCHAR2(255 BYTE),

"VALIDFROM" DATE,

"VALIDTO" DATE,

PRIMARY KEY ("DISCOUNTID")

);

1. **EMP\_CONTACTS:**

CREATE TABLE "MIS531GROUPS1H"."EMP\_CONTACTS"

( "EMPID" VARCHAR2(255 BYTE),

"MOBILENUM" NUMBER(10,0),

CONSTRAINT "PK\_EMP\_CONTACTS" PRIMARY KEY ("EMPID", "MOBILENUM"),

CONSTRAINT "FK\_EMPLOYEE\_CONTACT" FOREIGN KEY ("EMPID")

REFERENCES "MIS531GROUPS1H"."EMPLOYEES" ("EMPID") ON DELETE CASCADE

)

;

1. **EMP\_EMERGENCY\_CONTACTS table:**

CREATE TABLE "MIS531GROUPS1H"."EMP\_EMERGENCY\_CONTACTS"

( "EMPID" VARCHAR2(255 BYTE),

"EMERGENCYCONTACTNAME" VARCHAR2(255 BYTE),

"EMERGENCYCONTACTNUM" NUMBER(10,0),

CONSTRAINT "PK\_EMP\_EMERGENCY\_CONTACTS" PRIMARY KEY ("EMPID"),

CONSTRAINT "FK\_EMPLOYEE\_ID" FOREIGN KEY ("EMPID")

REFERENCES "MIS531GROUPS1H"."EMPLOYEES" ("EMPID") ON DELETE CASCADE

)

;

1. **EMPLOYEE\_AUTHENTICATION\_HISTORY table**

CREATE TABLE "MIS531GROUPS1H"."EMPLOYEE\_AUTHENTICATION\_HISTORY"

( "AUTHID" VARCHAR2(255 BYTE),

"EMPID" VARCHAR2(255 BYTE),

"LOGGEDINTIME" TIMESTAMP (6) NOT NULL ENABLE,

"LOGGEDOUTTIME" TIMESTAMP (6),

"EMPAUTHLONGITUDE" VARCHAR2(255 BYTE),

"EMPAUTHLATITUDE" VARCHAR2(255 BYTE),

CONSTRAINT "PK\_EMP\_AUTH\_HISTORY" PRIMARY KEY ("AUTHID", "EMPID"),

CONSTRAINT "FK\_AUTH\_ID" FOREIGN KEY ("AUTHID")

REFERENCES "MIS531GROUPS1H"."AUTHENTICATION" ("AUTHID") ENABLE,

CONSTRAINT "FK\_EMP\_ID" FOREIGN KEY ("EMPID")

REFERENCES "MIS531GROUPS1H"."EMPLOYEES" ("EMPID") ENABLE

);

1. **EMPLOYEES table**

CREATE TABLE "MIS531GROUPS1H"."EMPLOYEES"

( "EMPID" VARCHAR2(255 BYTE),

"FNAME" VARCHAR2(255 BYTE),

"LNAME" VARCHAR2(255 BYTE),

"SALARYGRADE" VARCHAR2(255 BYTE),

"DATEOFLEAVING" DATE,

"SSN" NUMBER(9,0),

"DATEOFJOINING" DATE,

"GENDER" VARCHAR2(255 BYTE),

"WORKLOCATION" VARCHAR2(255 BYTE),

"MANAGERID" VARCHAR2(255 BYTE),

"DEPARTMENTID" VARCHAR2(255 BYTE),

"AUTHID" VARCHAR2(255 BYTE),

"EMAIL" VARCHAR2(255 BYTE),

"STREET" VARCHAR2(255 BYTE),

"CITY" VARCHAR2(255 BYTE),

"STATE" VARCHAR2(255 BYTE),

"ZIPCODE" NUMBER(5,0),

"DOB" DATE,

"AGE" NUMBER,

"ISMANAGER" CHAR(1 BYTE),

"ISONCALLSUPPORT" CHAR(1 BYTE),

"ISMECHANIC" CHAR(1 BYTE),

PRIMARY KEY ("EMPID"),

CONSTRAINT "CHECK\_IS\_MANAGER" CHECK ( ismanager IN ( 'Y', 'N' ) ) ENABLE,

CONSTRAINT "CHECK\_IS\_ONCALLSUPPORT" CHECK ( isoncallsupport IN ( 'Y', 'N' ) ),

CONSTRAINT "CHECK\_IS\_MECHANIC" CHECK ( ismechanic IN ( 'Y', 'N' ) ) ENABLE,

CONSTRAINT "FK\_AUTHENTICATION\_ID" FOREIGN KEY ("AUTHID")

REFERENCES "MIS531GROUPS1H"."AUTHENTICATION" ("AUTHID") ON DELETE SET NULL,

CONSTRAINT "EMP\_PAYROLL\_FK" FOREIGN KEY ("SALARYGRADE")

REFERENCES "MIS531GROUPS1H"."PAYROLLS" ("SALARYGRADE"),

CONSTRAINT "EMP\_FK3" FOREIGN KEY ("MANAGERID")

REFERENCES "MIS531GROUPS1H"."EMPLOYEES" ("EMPID")

)

;

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."EMP\_ID\_GENERATOR" BEFORE

INSERT ON employees

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(7);

v\_newempid employees.empid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(emp\_seq.NEXTVAL)),

7,

'0')

INTO v\_nextval

FROM

dual;

v\_newempid := 'EMP' || v\_nextval;

:new.empid := v\_newempid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."EMP\_ID\_GENERATOR" ENABLE;

1. **MONTHLY\_REVENUE table**

CREATE TABLE "MIS531GROUPS1H"."MONTHLY\_REVENUE"

( "REVID" VARCHAR2(255 BYTE),

"YEAR" NUMBER(4,0),

"MONTH" NUMBER(2,0),

"CUSTPAYMENTS" NUMBER(10,2),

"EMPSALARY" NUMBER(10,2),

"REFUNDAMOUNT" NUMBER(10,2),

"SPAREPARTSPURCHASES" NUMBER(10,2),

"BIKESPURCHASES" NUMBER(10,2),

"MONTHTOTAL" NUMBER(10,2),

PRIMARY KEY ("REVID"));

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."REVENUE\_ID\_GENERATOR" BEFORE

INSERT ON monthly\_revenue

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(4);

v\_newrevid monthly\_revenue.revid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(rev\_seq.NEXTVAL)),

4,

'0')

INTO v\_nextval

FROM

dual;

v\_newrevid := 'REV' || v\_nextval;

:new.revid := v\_newrevid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."REVENUE\_ID\_GENERATOR" ENABLE;

1. **PAYMENT table:**

CREATE TABLE "MIS531GROUPS1H"."PAYMENT"

( "PAYMENTID" VARCHAR2(255 BYTE),

"AMOUNT" NUMBER,

"PAYMENTTIME" DATE,

"TYPEOFPAYMENT" CHAR(1 BYTE),

"CUSTID" VARCHAR2(255 BYTE),

PRIMARY KEY ("PAYMENTID"),

CONSTRAINT "CHK\_PAYMENT\_TYPE" CHECK (typeOfPayment IN ('S', 'D')) ENABLE,

CONSTRAINT "CUSTOMER\_FK" FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID") ENABLE

)

;

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."PAYMENT\_ID\_GENERATOR" BEFORE

INSERT ON payment

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(3);

v\_newpaymentid payment.paymentid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(payment\_seq.NEXTVAL)),

3,

'0')

INTO v\_nextval

FROM

dual;

v\_newpaymentid := 'PM' || v\_nextval;

:new.paymentid := v\_newpaymentid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."PAYMENT\_ID\_GENERATOR" DISABLE;

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."CALCULATEMONTHLYREVENUE"

AFTER UPDATE ON payment

FOR EACH ROW DECLARE

v\_newmonth NUMBER;

v\_month NUMBER;

v\_paymenttime payment.paymenttime%TYPE;

BEGIN

Select paymenttime into v\_paymenttime

from payment

order by paymenttime desc

where rownum = 1;

SELECT

TO\_NUMBER(to\_char(EXTRACT(MONTH FROM v\_paymenttime)))

INTO v\_month

FROM

dual;

dbms\_output.put\_line(v\_month);

SELECT

TO\_NUMBER(to\_char(sysdate, 'MM'))

INTO v\_currentmonth

FROM

dual;

dbms\_output.put\_line(v\_currentmonth);

end;

/

ALTER TRIGGER "MIS531GROUPS1H"."CALCULATEMONTHLYREVENUE" ENABLE;

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."CHECK\_NEW\_MONTH\_TRIGGER" AFTER

UPDATE ON payment

FOR EACH ROW

DECLARE

CURSOR c1 IS

SELECT

paymenttime

FROM

payment

ORDER BY

paymenttime DESC

WHERE

ROWNUM = 1;

current\_month NUMBER;

new\_month NUMBER;

v\_cursor NUMBER;

BEGIN

-- Extracting the current month from the system date

-- Extracting the new month from the updated paymentTime

OPEN c1;

SELECT

TO\_NUMBER(to\_char(sysdate, 'MM'))

INTO current\_month

FROM

dual;

SELECT

TO\_NUMBER(to\_char(EXTRACT(MONTH FROM(:new.paymenttime))))

INTO new\_month

FROM

dual;

v\_cursor := extract(MONTH FROM ( c1.paymenttime ));

dbms\_output.put\_line(new\_month);

dbms\_output.put\_line(current\_month);

dbms\_output.put\_line(v\_cursor);

CLOSE c1;

-- Checking if a new month has started

-- Your logic here for actions to be taken when a new month starts

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."CHECK\_NEW\_MONTH\_TRIGGER" ENABLE;

1. **PAYMENTS\_DISCOUNTS table**

CREATE TABLE "MIS531GROUPS1H"."PAYMENTS\_DISCOUNTS"

( "DISCOUNTID" VARCHAR2(255 BYTE),

"PAYMENTID" VARCHAR2(255 BYTE),

CONSTRAINT "PK\_PAYMENTS\_DISCOUNTS" PRIMARY KEY ("DISCOUNTID", "PAYMENTID"),

CONSTRAINT "FK\_DISCOUNT\_ID" FOREIGN KEY ("DISCOUNTID")

REFERENCES "MIS531GROUPS1H"."DISCOUNTS" ("DISCOUNTID") ENABLE,

CONSTRAINT "FK\_PAYMENT\_ID" FOREIGN KEY ("PAYMENTID")

REFERENCES "MIS531GROUPS1H"."PAYMENT" ("PAYMENTID") ENABLE

) SEGMENT CREATION IMMEDIATE

;

1. **PAYROLL\_HISTORY table**

CREATE TABLE "MIS531GROUPS1H"."PAYROLL\_HISTORY"

( "PAYROLLID" VARCHAR2(255 BYTE),

"EMPID" VARCHAR2(255 BYTE),

"PAYMENTID" VARCHAR2(255 BYTE),

"YEAROFPAYMENT" NUMBER(4,0),

"MONTHOFPAYMENT" NUMBER(2,0),

PRIMARY KEY ("PAYROLLID", "EMPID", "PAYMENTID"),

FOREIGN KEY ("EMPID")

REFERENCES "MIS531GROUPS1H"."EMPLOYEES" ("EMPID") ENABLE,

FOREIGN KEY ("PAYROLLID")

REFERENCES "MIS531GROUPS1H"."PAYROLLS" ("PAYROLLID") ENABLE

)

;

1. **PAYROLLS table :**

CREATE TABLE "MIS531GROUPS1H"."PAYROLLS"

( "PAYROLLID" VARCHAR2(255 BYTE),

"BASESALARY" NUMBER,

"SALARYGRADE" VARCHAR2(255 BYTE),

"TAXPERCENT" NUMBER,

"TAXAMOUNT" NUMBER,

"INCENTIVEPERCENT" NUMBER,

"INCENTIVEAMOUNT" NUMBER,

"TOTALSALARY" NUMBER,

PRIMARY KEY ("PAYROLLID"),

CONSTRAINT "PAYROLL\_UNIQUE" UNIQUE ("SALARYGRADE");

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."CALCULATE\_TOTAL\_SALARY"

AFTER INSERT ON PAYROLLS

FOR EACH ROW

BEGIN

DECLARE emp\_id\_var INT;

DECLARE total\_salary DECIMAL(10, 2);

-- Get the employee ID from the new payroll record

SET emp\_id\_var = NEW.empID;

-- Declare and open a cursor to fetch all payroll records for the employee

DECLARE payroll\_cursor CURSOR FOR SELECT baseSalary, taxAmount, incentiveAmount FROM PAYROLLS WHERE empID = emp\_id\_var;

OPEN payroll\_cursor;

-- Initialize total salary

SET total\_salary = 0;

-- Loop through each payroll record

payroll\_loop: LOOP

FETCH payroll\_cursor INTO base\_salary\_var, tax\_amount\_var, incentive\_amount\_var;

-- Exit the loop if no more records

IF FETCH\_STATUS() = 0 THEN

LEAVE payroll\_loop;

END IF;

-- Calculate total salary

SET total\_salary = total\_salary + base\_salary\_var + tax\_amount\_var + incentive\_amount\_var;

END LOOP;

-- Close the cursor

CLOSE payroll\_cursor;

-- Update the total salary in the EMPLOYEES table

UPDATE EMPLOYEES SET totalSalary = total\_salary WHERE empID = emp\_id\_var;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."CALCULATE\_TOTAL\_SALARY" ENABLE;

1. **PURCHASE\_HISTORY table:**

CREATE TABLE "MIS531GROUPS1H"."PURCHASE\_HISTORY"

( "VENDORID" VARCHAR2(255 BYTE),

"SPARTID" VARCHAR2(255 BYTE),

"PURCHASEID" VARCHAR2(255 BYTE),

"PDATE" DATE,

"QUANTITY" NUMBER(\*,0),

"BILLINGAMT" NUMBER(10,2),

PRIMARY KEY ("VENDORID", "SPARTID", "PURCHASEID"),

FOREIGN KEY ("VENDORID")

REFERENCES "MIS531GROUPS1H"."VENDORS" ("VENDORID") ENABLE,

FOREIGN KEY ("SPARTID")

REFERENCES "MIS531GROUPS1H"."SPARE\_PARTS" ("SPARTID") ENABLE

)

;

1. **RATINGS table:**

CREATE TABLE "MIS531GROUPS1H"."RATINGS"

( "REVIEWID" VARCHAR2(255 BYTE),

"USERID" VARCHAR2(255 BYTE),

"DATEPOSTED" DATE,

"BIKEID" VARCHAR2(255 BYTE),

"REVIEWTEXT" VARCHAR2(255 BYTE),

"CUSTID" VARCHAR2(255 BYTE),

PRIMARY KEY ("REVIEWID"),

CONSTRAINT "FK\_CUSTOMER\_RATINGS" FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID") ENABLE

)

;

1. **REFUNDS Table:**

CREATE TABLE "MIS531GROUPS1H"."REFUNDS"

( "REFUNDID" VARCHAR2(255 BYTE),

"PAYMENTID" VARCHAR2(255 BYTE),

"AMOUNT" NUMBER(10,2),

PRIMARY KEY ("REFUNDID", "PAYMENTID"),

FOREIGN KEY ("PAYMENTID")

REFERENCES "MIS531GROUPS1H"."PAYMENT" ("PAYMENTID") ENABLE

)

;

1. **SPAREPARTS table:**

CREATE TABLE "MIS531GROUPS1H"."SPARE\_PARTS"

( "SPARTID" VARCHAR2(255 BYTE),

"SPNAME" VARCHAR2(255 BYTE),

"SPWEIGHT" NUMBER(10,2),

"SPCOLOR" VARCHAR2(255 BYTE),

"SPBRAND" VARCHAR2(255 BYTE),

"PRICE" NUMBER(10,2),

"VENDORID" VARCHAR2(255 BYTE),

PRIMARY KEY ("SPARTID"),

FOREIGN KEY ("VENDORID")

REFERENCES "MIS531GROUPS1H"."VENDORS" ("VENDORID") ON DELETE SET NULL ENABLE

)

;

1. **STATION\_MAINTENANCE table:**

CREATE TABLE "MIS531GROUPS1H"."STATION\_MAINTENANCE"

( "MAINTENCEID" VARCHAR2(255 BYTE),

"STATIONID" VARCHAR2(255 BYTE),

"M\_START\_TIME" TIMESTAMP (6),

"M\_END\_TIME" TIMESTAMP (6),

"TYPEOFSERVICE" VARCHAR2(255 BYTE),

PRIMARY KEY ("MAINTENCEID", "STATIONID"),

CONSTRAINT "CHK\_STATION\_SERVICE" CHECK (typeOfService IN ('Repair', 'Regular Maintenance')) ENABLE,

FOREIGN KEY ("STATIONID")

REFERENCES "MIS531GROUPS1H"."STATIONS" ("STATIONID") ENABLE

)

;

1. **STATIONS tables:**

CREATE TABLE "MIS531GROUPS1H"."STATIONS"

( "STATIONID" VARCHAR2(255 BYTE),

"SNAME" VARCHAR2(255 BYTE),

"CITY" VARCHAR2(255 BYTE),

"BIKECAPACITY" NUMBER(\*,0),

"AVAILABLEBIKES" NUMBER(\*,0),

"MAINTENANCEBIKES" NUMBER(\*,0),

"FREEBIKESPACES" NUMBER(\*,0),

"STATIONLATITUDE" VARCHAR2(255 BYTE),

"STATIONLONGITUDE" VARCHAR2(255 BYTE),

PRIMARY KEY ("STATIONID"),

CONSTRAINT "CHK\_BIKECAP" CHECK (bikeCapacity >= availableBikes AND bikeCapacity >= maintenanceBikes AND bikeCapacity >= freeBikeSpaces) ENABLE

);

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."STATION\_ID\_GENERATOR" BEFORE

INSERT ON stations

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(3);

v\_newstationid stations.stationid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(station\_seq.NEXTVAL)),

3,

'0')

INTO v\_nextval

FROM

dual;

v\_newstationid := 'S' || v\_nextval;

:new.stationid := v\_newstationid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."STATION\_ID\_GENERATOR" ENABLE;

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."CALCULATE\_FREE\_BIKESPACE" BEFORE

INSERT OR UPDATE OF bikecapacity, availablebikes, maintenancebikes ON stations

FOR EACH ROW

DECLARE

v\_bikecapacity stations.freebikespaces%TYPE;

BEGIN

v\_bikecapacity := :new.bikecapacity - ( :new.availablebikes + :new.maintenancebikes );

:new.freebikespaces := v\_bikecapacity;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."CALCULATE\_FREE\_BIKESPACE" ENABLE;

1. **SUBSCRIPTION\_MODELS table :**

CREATE TABLE "MIS531GROUPS1H"."SUBSCRIPTION\_MODELS"

( "PLANID" VARCHAR2(255 BYTE),

"TITLE" VARCHAR2(255 BYTE),

"BASEPRICE" NUMBER,

"DESCRIPTION" VARCHAR2(255 BYTE),

PRIMARY KEY ("PLANID")

);

1. **SUBSCRIPTION\_STATUS table**

CREATE TABLE "MIS531GROUPS1H"."SUBSCRIPTION\_STATUS"

( "CUSTID" VARCHAR2(255 BYTE),

"PLANID" VARCHAR2(255 BYTE),

"JOINEDDATE" DATE,

"ENDDATE" DATE,

PRIMARY KEY ("CUSTID", "PLANID", "JOINEDDATE"),

FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID") ENABLE,

FOREIGN KEY ("PLANID")

REFERENCES "MIS531GROUPS1H"."SUBSCRIPTION\_MODELS" ("PLANID") ENABLE

);

1. **TRIPS table:**

CREATE TABLE "MIS531GROUPS1H"."TRIPS"

( "TRIPID" VARCHAR2(255 BYTE),

"STARTTIME" TIMESTAMP (6),

"ENDTIME" TIMESTAMP (6),

"CUSTID" VARCHAR2(255 BYTE),

"PICKUPSTATIONID" VARCHAR2(255 BYTE),

"DROPOFFSTATIONID" VARCHAR2(255 BYTE),

"BIKEID" VARCHAR2(255 BYTE),

"EXCESSCHARGE" NUMBER(10,2),

PRIMARY KEY ("TRIPID"),

FOREIGN KEY ("PICKUPSTATIONID")

REFERENCES "MIS531GROUPS1H"."STATIONS" ("STATIONID") ENABLE,

FOREIGN KEY ("DROPOFFSTATIONID")

REFERENCES "MIS531GROUPS1H"."STATIONS" ("STATIONID") ENABLE,

FOREIGN KEY ("CUSTID")

REFERENCES "MIS531GROUPS1H"."CUSTOMERS" ("CUSTID") ENABLE,

FOREIGN KEY ("BIKEID")

REFERENCES "MIS531GROUPS1H"."BIKES" ("BIKEID") ENABLE

);

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."CALCULATE\_EXCESS\_CHARGE" BEFORE

INSERT OR UPDATE OF endtime ON trips

FOR EACH ROW

DECLARE

v\_timestampdiff INTERVAL DAY TO SECOND;

v\_total\_seconds NUMBER;

v\_total\_minutes NUMBER;

v\_chargeperminute NUMBER := 0.5;

v\_charge trips.excesscharge%TYPE;

v\_excesscharge trips.excesscharge%TYPE;

BEGIN

-- Calculate trip duration

v\_timestampdiff := :new.endtime - :new.starttime;

-- Convert the difference to total minutes

v\_total\_minutes := extract(DAY FROM v\_timestampdiff) \* 14400 + extract(HOUR FROM v\_timestampdiff) \* 60 + extract(MINUTE FROM v\_timestampdiff

) + extract(SECOND FROM v\_timestampdiff) / 60;

IF v\_total\_minutes > 30 THEN

v\_charge := ( v\_total\_minutes - 30 ) \* v\_chargeperminute;

:new.excesscharge := v\_charge;

ELSE

:new.excesscharge := 0.00;

END IF;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."CALCULATE\_EXCESS\_CHARGE" ENABLE;

1. **VENDORS table:**

CREATE TABLE "MIS531GROUPS1H"."VENDORS"

( "VENDORID" VARCHAR2(255 BYTE),

"VPHONE" VARCHAR2(255 BYTE),

"VNAME" VARCHAR2(255 BYTE),

"VEMAIL" VARCHAR2(255 BYTE),

"STREET" VARCHAR2(255 BYTE),

"CITY" VARCHAR2(255 BYTE),

"STATE" VARCHAR2(255 BYTE),

"ZIPCODE" VARCHAR2(255 BYTE),

"COUNTRY" VARCHAR2(255 BYTE),

PRIMARY KEY ("VENDORID")

);

CREATE OR REPLACE EDITIONABLE TRIGGER "MIS531GROUPS1H"."VENDOR\_ID\_GENERATOR" BEFORE

INSERT ON vendors

FOR EACH ROW

DECLARE

v\_nextval VARCHAR(6);

v\_newvendorid vendors.vendorid%TYPE;

BEGIN

SELECT

lpad(TRIM(to\_char(vendor\_seq.NEXTVAL)),

6,

'0')

INTO v\_nextval

FROM

dual;

v\_newvendorid := 'VEND' || v\_nextval;

:new.vendorid := v\_newvendorid;

END;

/

ALTER TRIGGER "MIS531GROUPS1H"."VENDOR\_ID\_GENERATOR" ENABLE;

## 3. 5 Sequences

1. CREATE SEQUENCE "MIS531GROUPS1H"."BIKEMODEL\_SEQ" MINVALUE 1 MAXVALUE 999 INCREMENT BY 1 START WITH 107 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;
2. CREATE SEQUENCE "MIS531GROUPS1H"."CUST\_SEQ" MINVALUE 1 MAXVALUE 999999 INCREMENT BY 1 START WITH 261 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;
3. CREATE SEQUENCE "MIS531GROUPS1H"."EMP\_SEQ" MINVALUE 1 MAXVALUE 9999999 INCREMENT BY 1 START WITH 161 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;
4. CREATE SEQUENCE "MIS531GROUPS1H"."PAYMENT\_SEQ" MINVALUE 1 MAXVALUE 999 INCREMENT BY 1 START WITH 1000 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;
5. CREATE SEQUENCE "MIS531GROUPS1H"."REV\_SEQ" MINVALUE 1 MAXVALUE 9999 INCREMENT BY 1 START WITH 181 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;
6. CREATE SEQUENCE "MIS531GROUPS1H"."STATION\_SEQ" MINVALUE 1 MAXVALUE 999 INCREMENT BY 1 START WITH 157 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;
7. CREATE SEQUENCE "MIS531GROUPS1H"."VENDOR\_SEQ" MINVALUE 1 MAXVALUE 999999 INCREMENT BY 1 START WITH 121 CACHE 20 NOORDER NOCYCLE NOKEEP NOSCALE GLOBAL ;

**CHAPTER 4**

# QUERIES

## 4.1. Finding out top customers based on maximum rental time

The SQL query retrieves and analyzes customer ride data by calculating the total ride duration for each customer.

Using a Common Table Expression (`rideTime`), the query computes ride durations in seconds, associates them with customer details, and then aggregates the results.

The final output presents a list of customers sorted by descending total ride duration, providing insights into customer engagement and preferences for strategic decision-making by Tugo Bikes.

**WITH ridetime AS (**

**SELECT**

**tripid,**

**( EXTRACT(HOUR FROM((endtime) -(starttime))) \* 3600 + EXTRACT(MINUTE FROM((endtime) -(starttime))) \* 60 + EXTRACT(SECOND FROM**

**((endtime) -(starttime))) ) AS time1,**

**age,**

**c.custid,**

**c.fname,**

**c.lname**

**FROM**

**customers c**

**INNER JOIN trips t ON c.custid = t.custid**

**)**

**SELECT**

**custid,**

**fname,**

**lname,**

**SUM(time1) AS sumtime**

**FROM**

**ridetime**

**GROUP BY**

**custid,**

**fname,**

**lname**

**ORDER BY**

**sumtime DESC;**

## 4.2. Analyze Net Income Performance for Consecutive Quarters

The SQL query analyzes quarterly revenue differences for Tugo Bikes by grouping monthly revenue data into quarters using a Common Table Expression (CTE). The main query then calculates and presents the year, previous and current quarter numbers, and the revenue difference between consecutive quarters. This allows Tugo Bikes to assess trends and changes in quarterly revenue for strategic decision-making.

**WITH quarters AS (**

**SELECT**

**year,**

**ceil(month / 3) AS quarter,**

**SUM(monthtotal) AS monthtotal**

**FROM**

**monthly\_revenue**

**GROUP BY**

**year,**

**ceil(month / 3)**

**)**

**SELECT**

**year,**

**LAG(quarter)**

**OVER(**

**ORDER BY**

**year, quarter**

**) AS previous\_quarter,**

**quarter AS current\_quarter,**

**monthtotal - LAG(monthtotal)**

**OVER(**

**ORDER BY**

**year, quarter**

**) AS revenue\_difference**

**FROM**

**quarters**

**ORDER BY**

**year,**

**quarter;**

## 4.3. Identify Bike Hotspots

The SQL query identifies hotspots for bike pickups by joining information from the trips, bikes, and stations tables. It counts the number of pickups at each station, associates the pickup station with its city, and presents the results grouped by pickup station and city. The output provides valuable insights into the popularity of bike pickups at various stations in different cities.

**WITH hotspotspick AS (**

**SELECT**

**\***

**FROM**

**trips t**

**LEFT JOIN bikes b USING ( bikeid )**

**LEFT JOIN stations s USING ( stationid )**

**)**

**SELECT**

**h.pickupstationid,**

**COUNT(h.pickupstationid) AS noofst,**

**ss.city**

**FROM**

**hotspotspick h**

**LEFT JOIN stations ss ON h.pickupstationid = ss.stationid**

**GROUP BY**

**h.pickupstationid,**

**ss.city**

**ORDER BY**

**noofst DESC;**

## 4.4. Calculate the time between consecutive bike rentals for each bike

The SQL query, utilizing a Common Table Expression (CTE) named **TripDetails**, identifies details for Tugo Bikes trips, including the start time, end time, and the subsequent start time for each bike, grouped by bike ID.

The main query then calculates the time between consecutive rentals by subtracting the end time of the current trip from the start time of the next trip for each bike.

The result set provides insights into the duration between successive bike rentals, aiding in understanding usage patterns and selecting bikes for maintenance.

**WITH tripdetails AS (**

**SELECT**

**tripid,**

**bikeid,**

**starttime,**

**endtime,**

**LEAD(starttime)**

**OVER(PARTITION BY bikeid**

**ORDER BY**

**starttime**

**) AS nextstarttime**

**FROM**

**trips**

**)**

**SELECT**

**tripid,**

**bikeid,**

**starttime,**

**endtime,**

**nextstarttime,**

**nextstarttime - endtime AS timebetweenrentals**

**FROM**

**tripdetails**

**ORDER BY**

**bikeid,**

**starttime;**

## 4.5. Analyzing the frequency of the most common type of service done on Bikes and Stations

The SQL query, utilizing a Common Table Expression (CTE) named `ServiceCounts`, analyzes Bikes'/Stations’ maintenance records by extracting the year from maintenance start times, categorizing maintenance types, and counting the occurrences of each type in a given year. The main query then presents the maintenance year, type of service, and the corresponding service count, providing a structured overview of the maintenance activities over different years.

1. BIKES:

**WITH servicecounts AS (**

**SELECT**

**EXTRACT(YEAR FROM m\_start\_time) AS maintenanceyear,**

**typeofservice,**

**COUNT(\*) AS servicecount**

**FROM**

**bike\_maintenance**

**GROUP BY**

**EXTRACT(YEAR FROM m\_start\_time),**

**typeofservice**

**)**

**SELECT**

**maintenanceyear,**

**typeofservice,**

**servicecount**

**FROM**

**servicecounts**

**ORDER BY**

**maintenanceyear,**

**typeofservice;**

1. STATIONS:

**WITH servicecounts AS (**

**SELECT**

**EXTRACT(YEAR FROM m\_start\_time) AS maintenanceyear,**

**typeofservice,**

**COUNT(\*) AS servicecount**

**FROM**

**station\_maintenance**

**GROUP BY**

**EXTRACT(YEAR FROM m\_start\_time),**

**typeofservice**

**)**

**SELECT**

**maintenanceyear,**

**typeofservice,**

**servicecount**

**FROM**

**servicecounts**

**ORDER BY**

**maintenanceyear,**

**typeofservice;**

## 4.6. Identify Top 5 customers who paid the highest Excess Charges for the Bike rentals

The SQL query, utilizing a Common Table Expression (CTE) named CustomerPayments, calculates the total payment amounts for customers who made excess charge payments ('D'), ranks them based on their payment amounts in descending order, and assigns a payment rank. The main query then selects the top 5 customers with their payment ranks, names, and total payment amounts, providing a concise list of the highest-paying customers.

**WITH customerpayments AS (**

**SELECT**

**c.custid,**

**c.fname**

**|| ' '**

**|| c.lname AS name,**

**SUM(p.amount) AS totalamount,**

**RANK()**

**OVER(**

**ORDER BY**

**SUM(p.amount) DESC**

**) AS paymentrank**

**FROM**

**customers c**

**JOIN payment p ON c.custid = p.custid**

**WHERE**

**p.typeofpayment = 'D'**

**GROUP BY**

**c.custid,**

**c.fname,**

**c.lname**

**)**

**SELECT**

**paymentrank,**

**name,**

**totalamount**

**FROM**

**customerpayments**

**WHERE**

**paymentrank <= 5;**

## 4.7. Customer Payment Categories Analysis

This SQL query classifies customers into payment categories (Platinum, Gold, Silver, or Regular) based on their payment frequency. Using a Common Table Expression, it computes the total number of payments per customer and assigns categories accordingly. The main query outputs customer names, payment counts, and categories, facilitating the identification of high-value customers. Results are ordered by payment count in descending order for quick insights into customer payment behavior.

**WITH CustomerCategories AS (**

**SELECT**

**c.custID,**

**c.fName || ' ' || c.lName AS name,**

**COUNT(p.paymentID) AS numberOfPayments,**

**CASE**

**WHEN COUNT(p.paymentID) >= 20 THEN 'Platinum'**

**WHEN COUNT(p.paymentID) >= 15 THEN 'Gold'**

**WHEN COUNT(p.paymentID) >= 10 THEN 'Silver'**

**ELSE 'Regular'**

**END AS category**

**FROM**

**CUSTOMERS c**

**LEFT JOIN**

**PAYMENT p ON c.custID = p.custid**

**GROUP BY**

**c.custID, c.fName, c.lName**

**)**

**SELECT**

**name,**

**numberOfPayments,**

**category**

**FROM**

**CustomerCategories**

**order by numberofpayments desc;**

## 4.8. Monthly Revenue Analysis

This SQL query calculates and presents the total revenue for the month of November 2023. Using the SUM function, it aggregates the 'amount' column from the 'payment' table. The specified date range filters payments made during this period. The result, labeled as "Total Revenue," provides a concise overview of the financial performance for the specified month, aiding in monitoring and assessing business transactions during November 2023.

**select sum(amount) as "Total Revenue" from payment where paymenttime between '01-NOV-23' and '30-NOV-23';**

## 4.9. Active Subscriber Percentage Analysis

This SQL query assesses the percentage of active subscribers among customers with active payment status. The "activeCust" Common Table Expression (CTE) filters customers with valid subscriptions, considering factors like active status, payment status, and future subscription end dates. The "totalCust" CTE calculates the overall customer count. The main query then computes the percentage of active subscribers and presents it as "Percent of Active Subscribers," offering insights into customer engagement and subscription retention.

**with activeCust as (select custid, fname, lname, age, city, planid**

**from customers**

**inner join subscription\_status using(custid)**

**where activestatus = 'Y' and haspayed='Y' and enddate > sysdate**

**order by custid),**

**totalCust as (select count(custid) as total from customers)**

**select (count(a.custid)/total \* 100) as "Percent of Active Subscribers"**

**from activeCust a ,totalCust t**

**group by total;**

## 4.10. Subscription Count by Plan Analysis

This SQL query analyzes the distribution of customers across different subscription plans. By grouping the data based on the 'planid' and counting the associated customers, it provides insights into plan popularity. The result, ordered by customer count in descending order, reveals the number of customers for each subscription plan, aiding in identifying the most and least subscribed plans within the dataset.

**select planid, count(custid) as CtCust**

**from subscription\_status**

**group by planid**

**order by CtCust desc;**

## 4.11. Top 3 Bike Models by Trip Count Analysis

This SQL query identifies the top three bike models based on the number of associated trips. The "BikeTripsCount" Common Table Expression (CTE) combines data from the 'BIKES,' 'TRIPS,' and 'BIKE\_MODELS' tables, counting the trips for each bike model. The main query then employs the RANK() window function to assign rankings based on trip counts, and finally, filters and presents the top three bike models, showcasing their popularity in terms of trip usage.

**WITH BikeTripsCount AS (**

**SELECT**

**b.modelID,**

**bm.modelName,**

**COUNT(t.tripID) AS tripCount**

**FROM**

**BIKES b**

**JOIN**

**TRIPS t ON b.bikeID = t.bikeID**

**JOIN**

**BIKE\_MODELS bm ON b.modelID = bm.modelID**

**GROUP BY**

**b.modelID, bm.modelName**

**)**

**SELECT**

**modelID,**

**modelName,**

**tripCount**

**FROM**

**(**

**SELECT**

**modelID,**

**modelName,**

**tripCount,**

**RANK() OVER (ORDER BY tripCount DESC) AS rnk**

**FROM**

**BikeTripsCount**

**)**

**WHERE**

**rnk <= 3;**

## 4.12. On-Call Employee with the Highest Complaint Count

This SQL query focuses on employees designated for on-call support, tracking the number of complaints each has received. The "OnCallEmployeeComplaints" Common Table Expression (CTE) calculates the complaint count for each on-call employee, sorting the results in descending order. The main query then selects and presents the on-call employee with the highest complaint count, aiding in identifying the most impactful employee in addressing customer concerns.

**WITH OnCallEmployeeComplaints AS (**

**SELECT**

**e.empID,**

**e.fName || ' ' || e.lName AS employeeName,**

**COUNT(c.complaintID) AS numberOfComplaints**

**FROM**

**EMPLOYEES e**

**JOIN**

**COMPLAINTS c ON e.empID = c.empID**

**WHERE**

**e.isOnCallSupport = 'Y'**

**GROUP BY**

**e.empID, e.fName, e.lName**

**order by numberOfComplaints desc**

**)**

**SELECT**

**employeeName,**

**numberOfComplaints**

**FROM**

**OnCallEmployeeComplaints**

**where rownum = 1**

**ORDER BY**

**numberOfComplaints DESC;**

## 4.13. Identify the top 5 revenue-generating vendors for Spare Parts

This SQL query evaluates the revenue generated from various vendors by summing the billing amounts associated with spare parts purchases. It utilizes joins between the 'VENDORS,' 'SPARE\_PARTS,' and 'PURCHASE\_HISTORY' tables to link vendor information with purchase records. The result presents each vendor's ID, name, and the total revenue generated from spare parts transactions. The list is ordered in descending order by total revenue, providing insights into vendor performance and contribution to overall revenue.

**SELECT v.vendorID, v.vName, SUM(ph.billingamt) AS TotalRevenue**

**FROM VENDORS v**

**JOIN SPARE\_PARTS sp**

**ON v.vendorID = sp.vendorID**

**JOIN PURCHASE\_HISTORY ph**

**ON sp.sPartID = ph.sPartID**

**GROUP BY v.vendorID, v.vName**

**ORDER BY TotalRevenue DESC;**

## 4.14. Customer Unresolved Complaints Analysis

This SQL query tallies unresolved complaints for each customer by joining the 'CUSTOMERS' and 'COMPLAINTS' tables. The results are grouped by customer ID and name, ordered by the count of unresolved complaints in descending order. This concise analysis offers insights into customer service concerns, facilitating prioritized resolution efforts.

**SELECT c.custID, c.fName || ' ' || c.lName AS CustomerName, COUNT(co.complaintID) AS UnresolvedComplaints**

**FROM CUSTOMERS c**

**inner JOIN COMPLAINTS co ON c.custID = co.custID AND co.status = 'P'**

**GROUP BY c.custID, c.fName, c.lName**

**ORDER BY UnresolvedComplaints DESC;**

**CHAPTER 6**

# TRIGGERS and PROCEDURES

## 5.1 Triggers

Triggers are event-driven actions that respond to changes in the database, and hence, they are widely used for automated tasks such as updating related tables when a change occurs in a specific table, sending notifications, or performing calculations based on certain events.

### 5.1.1. Trigger for calculating Excess Charge for a trip

In the Tugo Bikes model, customers are allowed a 30-minute usage window for each bike rental. If this duration is surpassed, a nominal charge per minute is applied for the extra time. To automate the calculation of the additional charge, a trigger has been implemented. This trigger activates when the end time of a trip is recorded in the trips table.

CREATE OR REPLACE TRIGGER calculate\_excess\_charge BEFORE

INSERT OR UPDATE OF endtime ON trips

FOR EACH ROW

DECLARE

v\_timestampdiff INTERVAL DAY TO SECOND;

v\_total\_seconds NUMBER;

v\_total\_minutes NUMBER;

v\_chargeperminute NUMBER := 0.5;

v\_charge trips.excesscharge%TYPE;

v\_excesscharge trips.excesscharge%TYPE;

BEGIN

-- Calculate trip duration

v\_timestampdiff := :new.endtime - :new.starttime;

-- Convert the difference to total minutes

v\_total\_minutes := extract(DAY FROM v\_timestampdiff) \* 14400 + extract(HOUR FROM v\_timestampdiff) \* 60 + extract(MINUTE FROM v\_timestampdiff

) + extract(SECOND FROM v\_timestampdiff) / 60;

IF v\_total\_minutes > 30 THEN

v\_charge := ( v\_total\_minutes - 30 ) \* v\_chargeperminute;

:new.excesscharge := v\_charge;

ELSE

:new.excesscharge := 0.00;

END IF;

END;

### 5.1.2. Trigger for calculating Free Bike Space at a station

For every station in the Tugo Bikes system, there exists a predetermined fixed total bike capacity. The system also keeps track of the current number of available bikes and bikes under maintenance. To ensure that customers are informed about the bike drop-off options at each station, an automatic calculation mechanism is in place. This mechanism operates whenever updates are made to attributes such as bike capacity, available bikes, and maintenance bikes. The result of this calculation yields the free bike space, providing users with real-time information on the station's capacity for bike returns.

CREATE OR REPLACE TRIGGER calculate\_free\_bikespace BEFORE

INSERT OR UPDATE OF bikecapacity, availablebikes, maintenancebikes ON stations

FOR EACH ROW

DECLARE

v\_bikecapacity stations.freebikespaces%TYPE;

BEGIN

v\_bikecapacity := :new.bikecapacity - ( :new.availablebikes + :new.maintenancebikes );

:new.freebikespaces := v\_bikecapacity;

END;

## 5.2 Procedures

Stored procedures in Oracle SQL offer a way to encapsulate a sequence of SQL and PL/SQL statements into a named program unit. Procedures can be employed for automating routine maintenance tasks or updates based on specific conditions. They do not run automatically like triggers and have to be executed either manually or can be called from inside a trigger.

### 5.2.1. Procedure for Calculating Monthly Net Income

The `CalculateMonthlyRevenue` procedure is a tool in the Tugo Bikes system that figures out how much money the company made in each month. It takes a date as input, looks at various financial transactions, and calculates the monthly revenue. Here's a simplified breakdown:

1. Date Setup: It starts by figuring out the month and year based on a date provided.
2. Financial Calculations: Then, it adds up the money Tugo Bikes received from customers, subtracts the money spent on employee salaries, bike-related costs, refunds, and spare parts.
3. Handling Missing Data: If there's no data for any of these categories, it assumes the value is zero to avoid errors.
4. Monthly Revenue: The result is the monthly revenue, showing how much profit or loss Tugo Bikes made that month.
5. Record Keeping: Finally, it records this information in a table called `monthly\_revenue` to keep a history of the company's financial performance over time.

CREATE OR REPLACE PROCEDURE calculatemonthlyrevenue (

v\_paramdate DATE

) AS

v\_date DATE;

v\_month NUMBER;

v\_year NUMBER;

v\_custpayments NUMBER;

v\_payroll NUMBER;

v\_bikecost NUMBER;

v\_refund NUMBER;

v\_spareparts NUMBER;

v\_monthlyrev NUMBER;

BEGIN

SELECT

v\_paramdate - 1

INTO v\_date

FROM

dual;

SELECT

EXTRACT(MONTH FROM v\_date),

EXTRACT(YEAR FROM v\_date)

INTO

v\_month,

v\_year

FROM

dual;

--Calculate amount from customer payments

SELECT

SUM(amount)

INTO v\_custpayments

FROM

payment

WHERE

( EXTRACT(MONTH FROM paymenttime) = v\_month

AND EXTRACT(YEAR FROM paymenttime) = v\_year );

--Calculate total salary paid to employees

SELECT

SUM(totalsalary)

INTO v\_payroll

FROM

payroll\_history p1

JOIN payrolls p2 ON p1.payrollid = p2.payrollid

WHERE

( monthofpayment = v\_month

AND yearofpayment = v\_year );

--Calculate money in refunds

SELECT

SUM(r.amount)

INTO v\_refund

FROM

refunds r

INNER JOIN payment p ON r.paymentid = p.paymentid

WHERE

( EXTRACT(MONTH FROM paymenttime) = v\_month

AND EXTRACT(YEAR FROM paymenttime) = v\_year );

--Calculate money spent on bikes

SELECT

SUM(modelcost)

INTO v\_bikecost

FROM

bike\_models bm

INNER JOIN bikes b ON bm.modelid = b.modelid

WHERE

yearofpurchase = v\_year

AND monthofpurchase = v\_month;

--Calculate money spent on spare parts

SELECT

SUM(price)

INTO v\_spareparts

FROM

spare\_parts sp

LEFT JOIN purchase\_history ph ON sp.spartid = ph.spartid

WHERE

( EXTRACT(YEAR FROM pdate) = v\_year

AND EXTRACT(MONTH FROM pdate) = v\_month );

IF v\_custpayments IS NULL THEN

v\_custpayments := 0;

END IF;

IF v\_payroll IS NULL THEN

v\_payroll := 0;

END IF;

IF v\_bikecost IS NULL THEN

v\_bikecost := 0;

END IF;

IF v\_refund IS NULL THEN

v\_refund := 0;

END IF;

IF v\_spareparts IS NULL THEN

v\_spareparts := 0;

END IF;

--Calculate monthly revenue

v\_monthlyrev := v\_custpayments - ( v\_payroll + v\_bikecost + v\_refund + v\_spareparts );

--Insert into monthly\_revenue table

INSERT INTO monthly\_revenue (

year,

month,

custpayments,

empsalary,

refundamount,

sparepartspurchases,

bikespurchases,

monthtotal

) VALUES (

v\_year,

v\_month,

v\_custpayments,

v\_payroll,

v\_refund,

v\_spareparts,

v\_bikecost,

v\_monthlyrev

);

END;

**Scheduled Job for running the procedure**

Since a stored has cannot automatically execute like a trigger, we have creates job named `MONTHLY\_REVENUE\_CALCULATION\_JOB` using the Oracle DBMS\_SCHEDULER package. This job is designed to automatically execute the `CALCULATEMONTHLYREVENUE` procedure on a monthly basis with specific parameters. Here's a description of the job:

1. Job Name: The job is named `MONTHLY\_REVENUE\_CALCULATION\_JOB` for easy identification.

2. Job Type: It is of type `PLSQL\_BLOCK`, indicating that the job executes a block of PL/SQL code.

3. Job Action: The job action is defined as a PL/SQL block that initiates the `CALCULATEMONTHLYREVENUE` procedure with the current date (SYSDATE) as a parameter.

4. Start Date: The job is set to start immediately upon creation, as indicated by `SYSTIMESTAMP`.

5. Repeat Interval: The job is configured to repeat monthly, specifically on the 1st day of each month, at midnight (00:00:00).

6. Enabled: The job is enabled (`TRUE`), ensuring that it is active and will run according to the specified schedule.

-- Create a job that runs monthly with parameters

BEGIN

DBMS\_SCHEDULER.create\_job (

job\_name => 'MONTHLY\_REVENUE\_CALCULATION\_JOB',

job\_type => 'PLSQL\_BLOCK',

job\_action => 'BEGIN CALCULATEMONTHLYREVENUE(SYSDATE); END;',

start\_date => SYSTIMESTAMP,

repeat\_interval => 'FREQ=MONTHLY; BYMONTHDAY=1; BYHOUR=0; BYMINUTE=0; BYSECOND=0',

enabled => TRUE

);

END;

/

### 5.2.2 Procedure for Rewarding Loyalty Points for all Customers

The **‘UPDATE\_REWARDS\_FOR\_ALL\_CUSTOMERS’** procedure iterates through all customer records, calculates their total spending based on payments, and then determines loyalty points. Depending on the total spending, a different number of additional points are assigned. The procedure calls another procedure **‘UPDATE\_OR\_INSERT\_REWARDS’** to update or insert the loyalty points for each customer in the cust\_rewards table. This ensures that customers receive loyalty points based on their spending behavior.

CREATE OR REPLACE PROCEDURE update\_rewards\_for\_all\_customers IS

-- Declare variables

cust\_id\_var customers.custid%TYPE;

total\_spending NUMBER(10, 2);

additional\_points\_param NUMBER;

-- Declare a cursor to fetch all customer IDs

CURSOR customer\_cursor IS

SELECT

custid

FROM

customers;

BEGIN

-- Loop through each customer

FOR customer\_rec IN customer\_cursor LOOP

-- Fetch customer ID into variable

cust\_id\_var := customer\_rec.custid;

-- Calculate total spending for the customer

SELECT

nvl(SUM(amount),

0)

INTO total\_spending

FROM

payment

WHERE

custid = cust\_id\_var;

-- Determine loyalty points based on spending

IF total\_spending >= 4000 THEN

additional\_points\_param := 50;

ELSE

additional\_points\_param := 10;

END IF;

-- Call the update\_or\_insert\_rewards procedure for each customer

update\_or\_insert\_rewards(cust\_id\_var, additional\_points\_param);

END LOOP;

END update\_rewards\_for\_all\_customers;

/

### 5.2.3 Procedure for Updating Reward Points for Customers

The **`UPDATE\_OR\_INSERT\_REWARDS`** procedure checks if a customer ID exists in the `cust\_rewards` table. It takes two parameters, custid & reward points. If the customer exists, it updates the existing reward points by adding the provided additional points. If the customer does not exist, a new row is inserted with the customer ID and the provided additional points. This procedure effectively manages the updating or inserting of loyalty points for customers in the `cust\_rewards` table based on the specified parameters.

CREATE OR REPLACE PROCEDURE update\_or\_insert\_rewards (

cust\_id\_param customers.custid%TYPE,

additional\_points\_param NUMBER

) IS

-- Declare variables to store existing reward points

existing\_reward\_points NUMBER;

-- Declare a cursor to fetch existing reward points

CURSOR reward\_cursor IS

SELECT

rewardpoints

FROM

cust\_rewards

WHERE

custid = cust\_id\_param;

BEGIN

-- Open the cursor

OPEN reward\_cursor;

-- Fetch existing reward points (if any)

FETCH reward\_cursor INTO existing\_reward\_points;

-- Close the cursor

CLOSE reward\_cursor;

-- Check if the customer ID exists in the REWARDS table

IF existing\_reward\_points IS NOT NULL THEN

-- Customer ID exists, update existing reward points

UPDATE cust\_rewards

SET

rewardpoints = existing\_reward\_points + additional\_points\_param

WHERE

custid = cust\_id\_param;

ELSE

-- Customer ID does not exist, insert a new row

INSERT INTO cust\_rewards (

custid,

rewardpoints

) VALUES (

cust\_id\_param,

additional\_points\_param

);

END IF;

END update\_or\_insert\_rewards;

/

**CHAPTER 6**

# USER INTERFACE

## 6.1. User Interface walk-through ([Recorded Video Walkthrough](https://drive.google.com/file/d/13fr0wdRHyaBM2l2quE4jLDJUXjHmDeaP/view?usp=sharing))

We have built a robust application using HTML, CSS, Bootstrap, JavaScript and Flask and hosted it on AWS EC2([Website-Link](http://ec2-18-236-172-125.us-west-2.compute.amazonaws.com:5000/)). There are several pages including landing page which provide detailed information about the Tugo Bikeshare and several forms to perform add, edit, delete rows functionality on multiple tables with certain exception handling mechanisms.

### 6.1.1 Home page or Landing page:

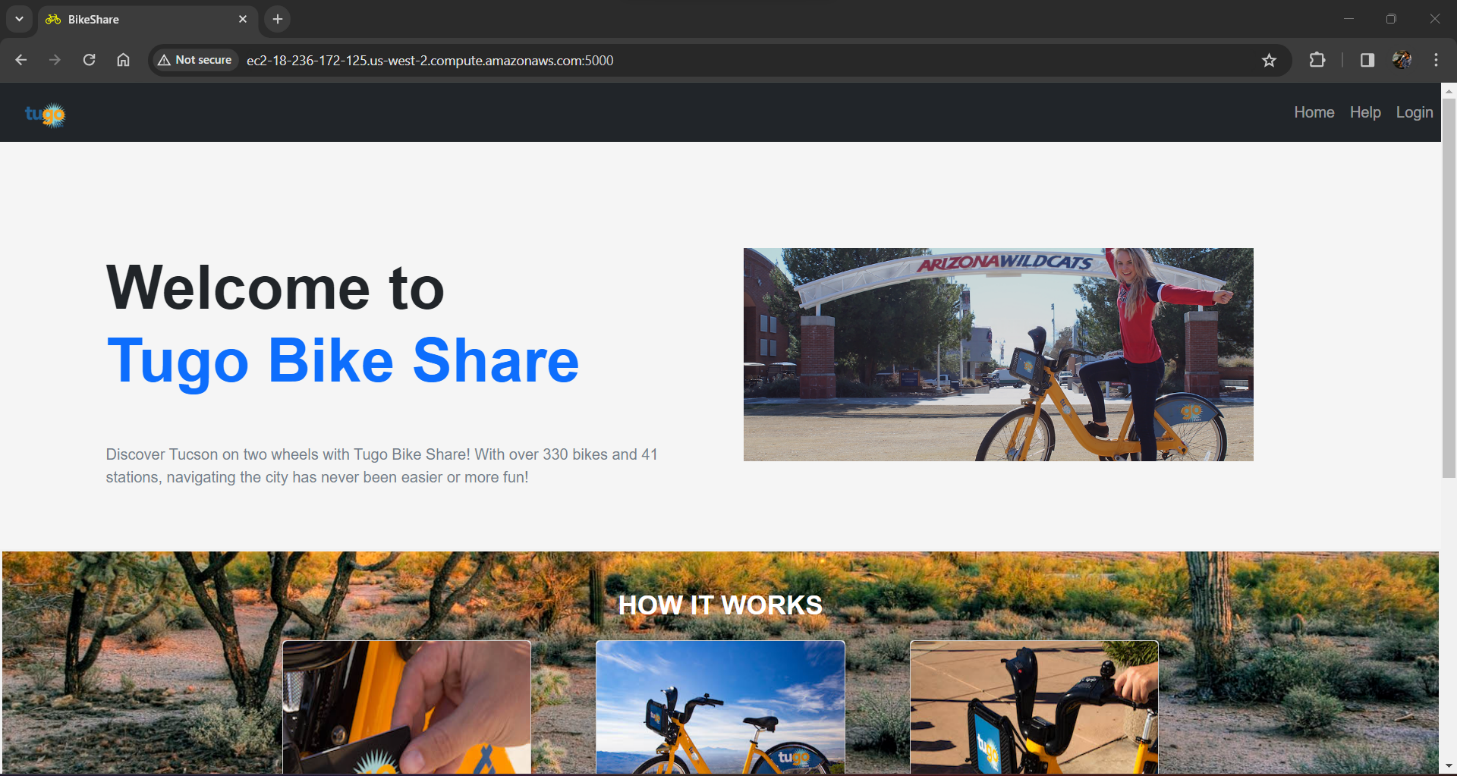
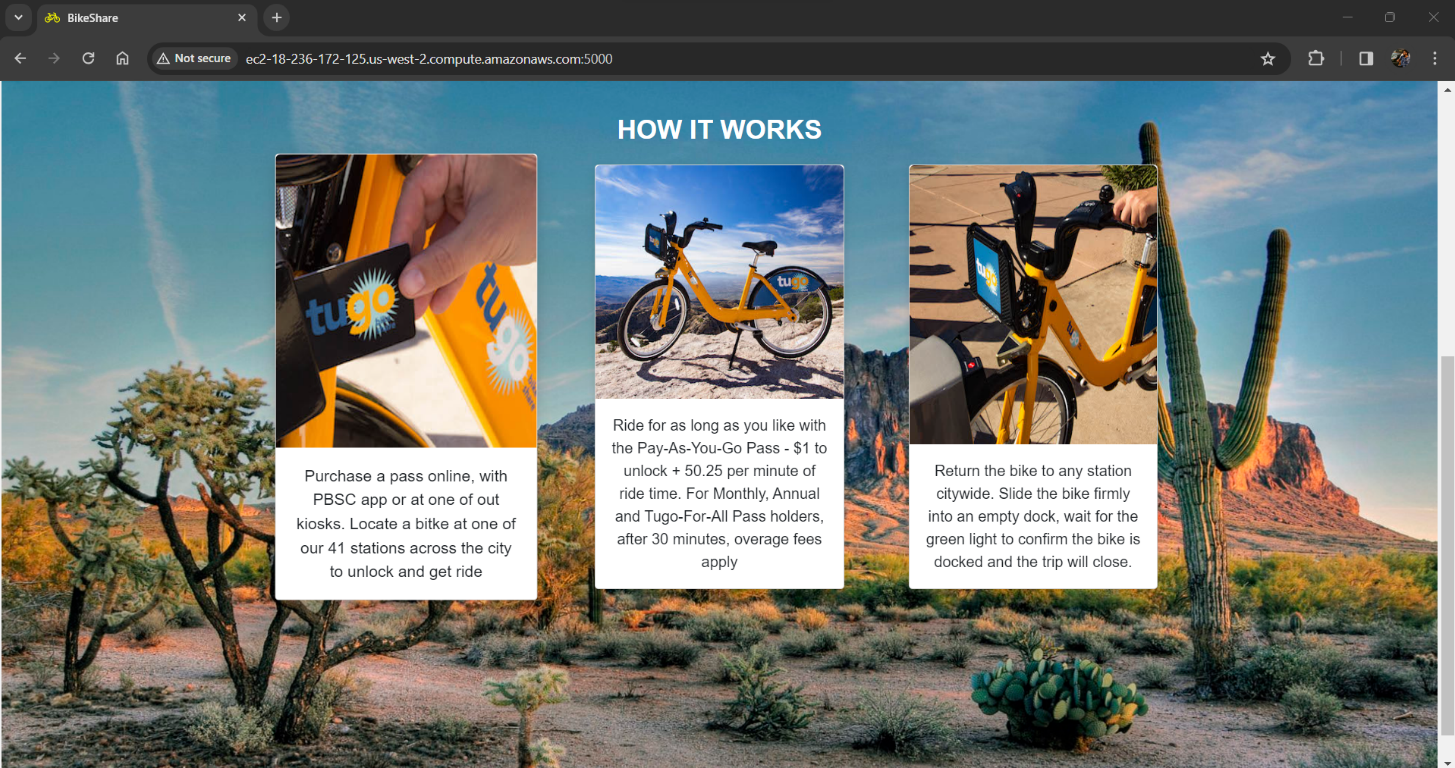
****

Figure 1: Home Page

The home page(Figure 1, Figure 2) provides a brief description of Tugo Bike Share and its features and how it works

Figure 2: Home Page continuation

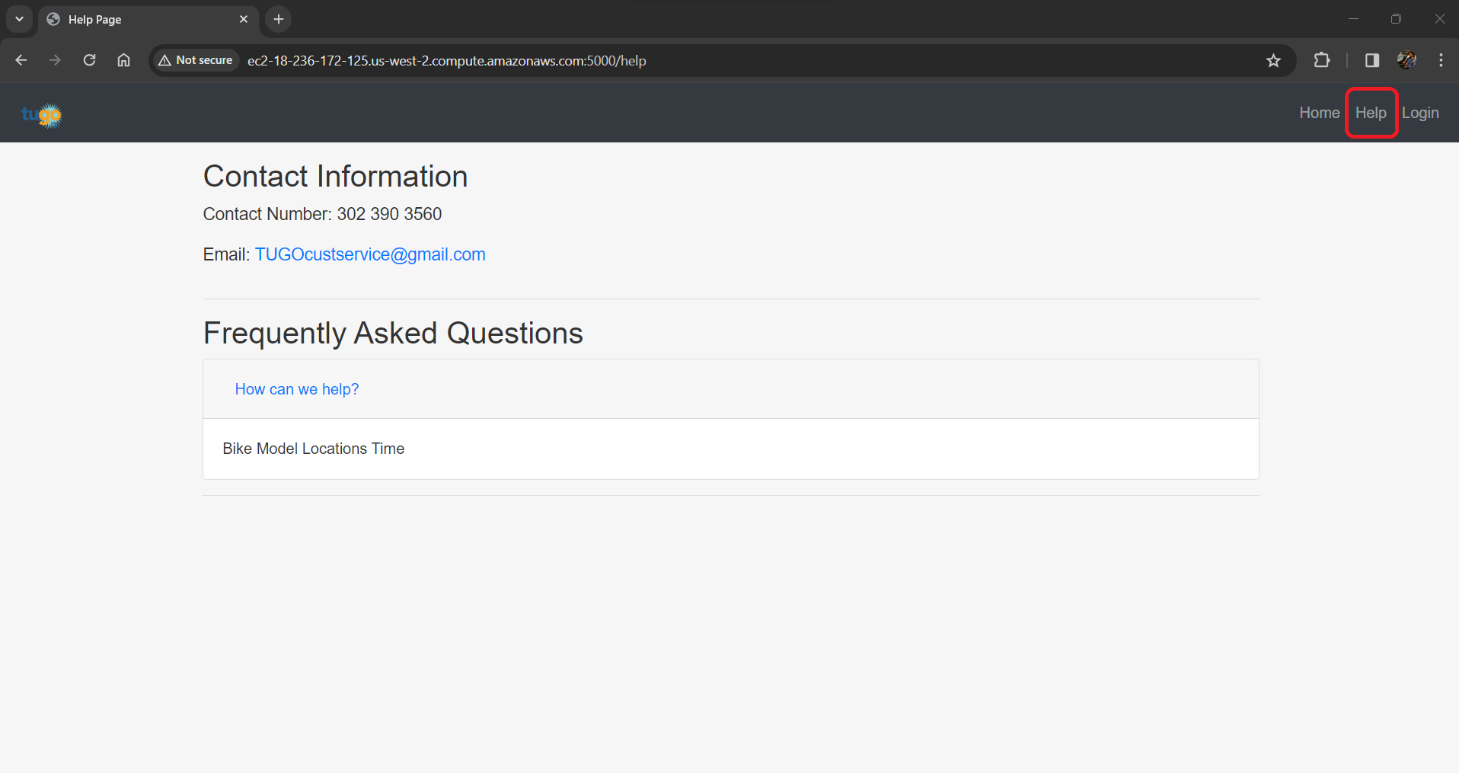
****

Figure 3: Help Page

A specified help page is included which can be accessed through the navigation bar (highlighted in red) which provides information about the customer service contact number and email address. They can be contacted for any queries regarding bikes, mobile app or any issues related to Tugo Bikeshare.

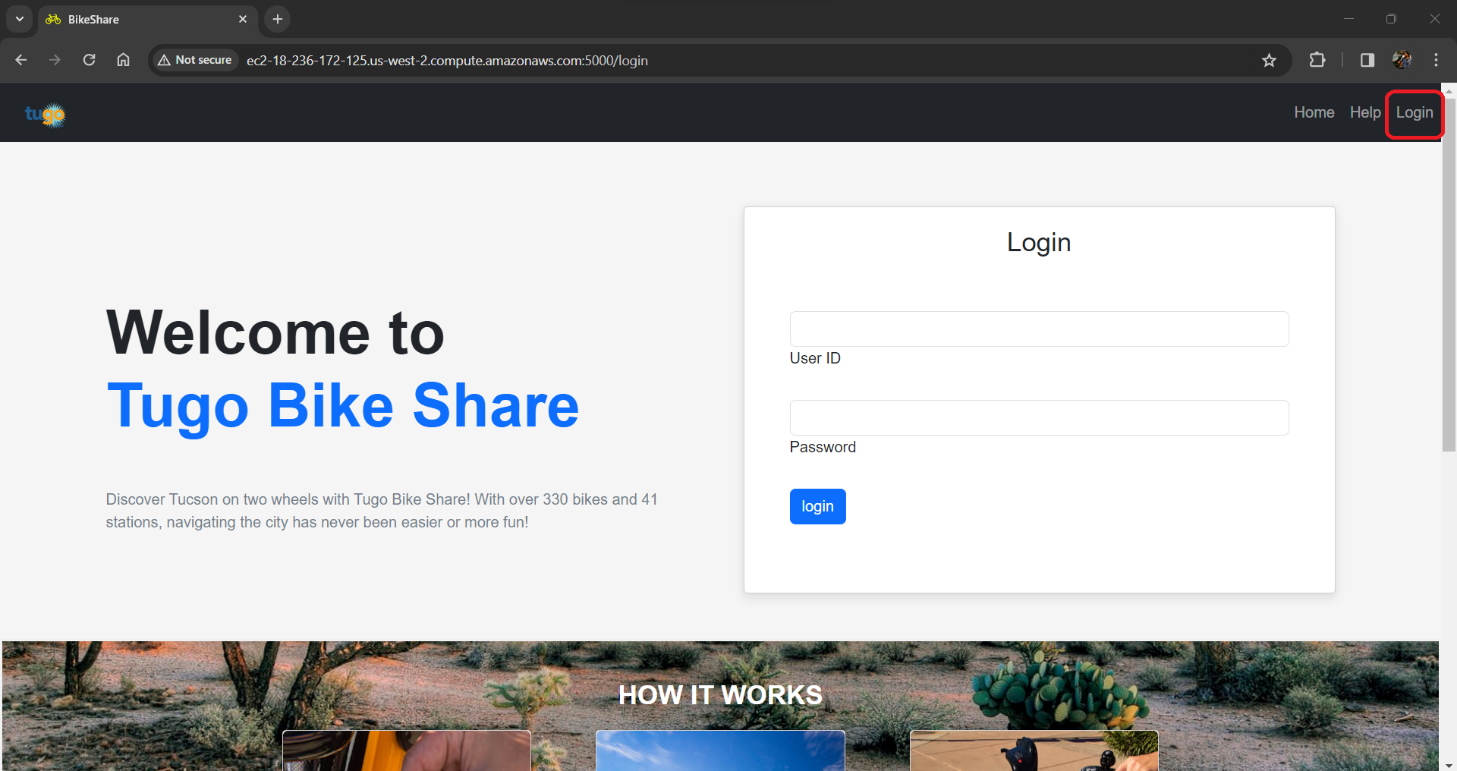
****

Figure 4: Login Page

A dedicated login page is provided for the administrative personnel which can be access through navigation bar (highlighted in red) to provide access to all the database tables, to perform add, update, delete operations and ability to query data and get business insights.

The key features of login page:

* User provide userID and password as input which will be validated with the data present in AUTHENTICATIONS table.
* If the username and password combination match with the record present in the database, user will be redirected to the home page (Figure 5) and all the options will be enabled in the navigation bar to access certain database table and perform operations.
* Successful Login Credentials:
* Username: laura\_brown
* Password: pass004
* If the user authentication fails (Figure 6), an alert with invalid credentials will be displayed and the user will be redirected to the login page.

A screenshot of a video

Description automatically generated

Figure5: User Authentication Failed

A screenshot of a computer

Description automatically generated

Figure6: Login Successful

## 6.2 Perform Add, Update and Delete Operations on Table

### 6.2.1. CUSTOMERS Table

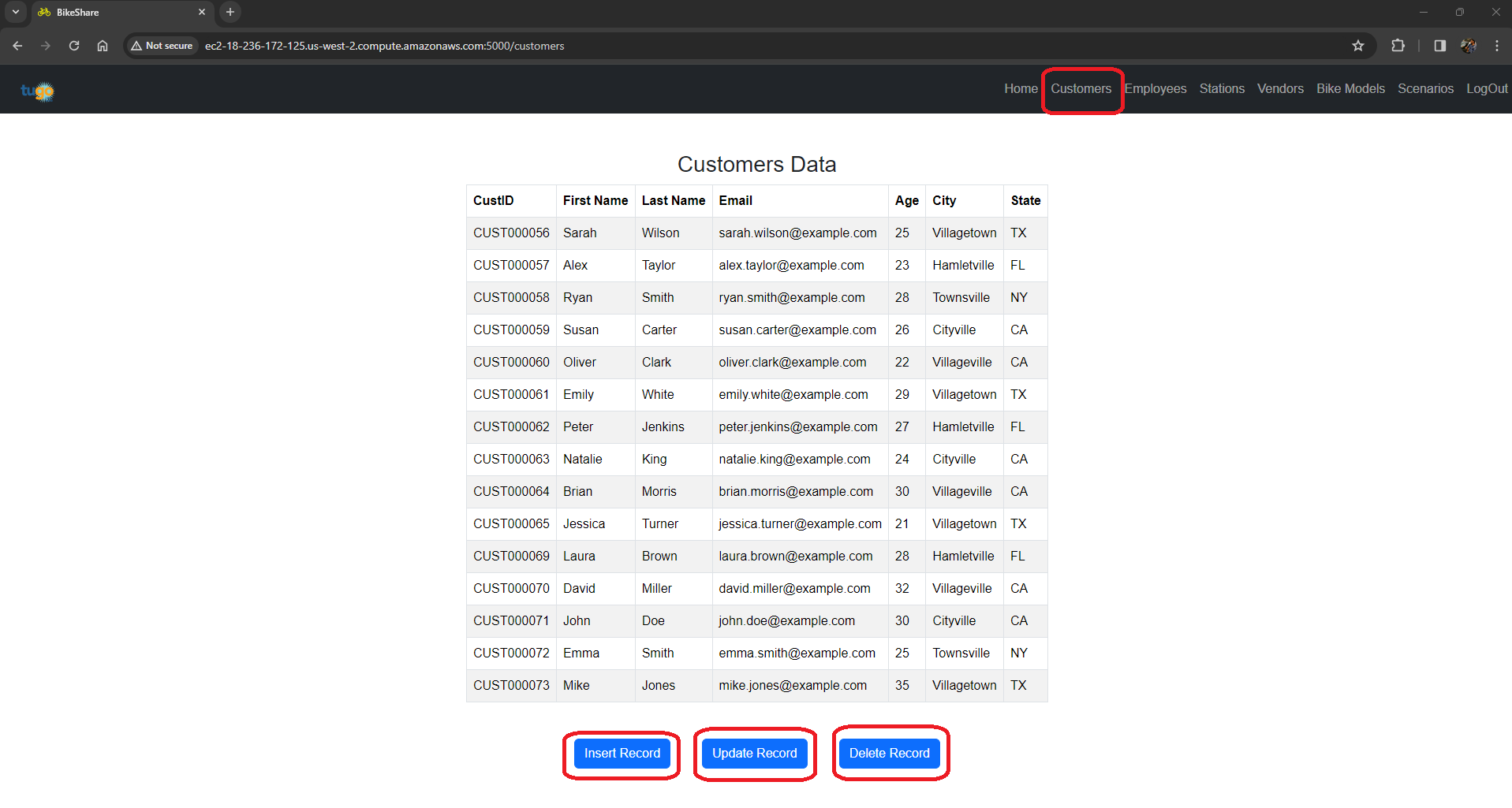


Figure 7: Customers Data

When the Customers option is chosen from the top navigation bar, the system fetches data from the customers table and displays it on the front end. We also offer options to insert, update and delete records into the table by using three blue buttons at the bottom of the table.

Important thing to note while filling form to insert, update is that the customer identifier (CustID) is incremented automatically using a trigger-sequence combination, Age should be a numeric value. During insertion, all the fields in the form are mandatory. To update a record using update record option, the respective CustID must be provided in the form. Similarly, to delete a record the respective CustID must be provided. The walkthrough of all the forms and actions are clearly explained in the attached [Front-End Walkthrough video.](https://drive.google.com/file/d/13fr0wdRHyaBM2l2quE4jLDJUXjHmDeaP/view?usp=sharing)

### 6.2.2. Other Data Tables

All the data tables like Employees, Stations, Vendors, Bike Models allow user to add records, update record and delete record functionality.

Some important features in each of the tables are given below

* **Employees Table:** The Employee ID is automatically incremented using trigger-sequence combination while inserting.
* **Stations Table:** The Station ID is automatically incremented using a trigger-sequence combination while inserting and the Free Spaces is automatically calculated using a trigger (refer video walkthrough for clarity).
* **Vendors Table:** The Vendor ID is automatically incremented using trigger-sequence combination while inserting.
* **Bike Models Table:** The Model ID is automatically incremented using trigger-sequence combination while inserting. The respective vendor id must be there in the parent Vendors table. An alert with invalid vendor details will be provided if user tried to provide a Vendor ID that is not provided in the Vendors table.

### 6.2.3: Business Insights using SQL Queries.

The Scenarios page provides access to all the SQL queries that were written to retrieve specific business insight. A Run Query button is provided to run the query and it redirects the user to separate page where output for the query is displayed.

For example: when user selects to run query for scenario 2(figure 8) the user is redirected to a new page where the query output is displayed (Figure 9) and the business insight is inferred from it.

A screenshot of a computer

Description automatically generated

Figure 8: Insights/ Scenarios PageA screenshot of a graph

Description automatically generated

Figure 9: Output for Scenario 2

**CHAPTER 7**

# IMPLEMENTATION PLAN

**Step 1: Pre-Implementation Preparation Plan**

Resource Allocation:

* Assign tasks to the Database Administrator (DBA) and Software
* Developers/Engineers based on their roles.
* Allocate 50 hours for the DBA and 100 hours for the Developers.

Budget Estimation:

* Calculate the cost estimate based on hourly rates:
* DBA hourly rate: $50
* Developer hourly rate: $80

Task Breakdown:

* DBA Tasks (50 hours):
  + - Review current database structure and requirements.
    - Plan database optimizations and improvements.
    - Create a backup plan and disaster recovery strategy.
    - Collaborate with Developers for database-related technical specifications.
* Developer Tasks (100 hours):
  + - Review system requirements and technical specifications.
    - Design software architecture compatible with the database.
    - Develop necessary components/modules with database integration.
    - Conduct testing and debugging in coordination with the DBA.

Timeline:

* Allocate timeframes for each task based on estimated hours.
* Ensure synchronization between the DBA and Developer tasks for seamless integration.

Communication and Coordination:

* Establish regular meetings between the DBA and Developers to discuss progress, challenges, and adjustments needed.
* Document discussions and decisions for reference and future planning.

Quality Assurance:

* Implement quality checks and validations at each stage of development.
* Set up a review process to ensure compliance with specifications and standards.

Risk Assessment:

* Identify potential risks related to database changes or software development.
* Develop contingency plans to mitigate any unexpected issues.

Documentation:

* Maintain detailed documentation of the pre-implementation phase, including plans, decisions, and configurations.

Approval Process:

* Define the process for approval of the pre-implementation phase before moving forward to the next steps.

Review and Adjustment:

* Conduct a final review of the pre-implementation phase.
* Make necessary adjustments based on feedback or unforeseen challenges encountered during this stage.

**Step 2: Cloud Provider Selection and Setup**

Cloud Provider: AWS (Amazon Web Services)

Person-Hours Estimate:

Research and Selection: 20 hours

* + Evaluating various cloud providers based on their features, pricing, and suitability for the project requirements
  + Analyzing the performance, scalability, and security aspects of each cloud platform
  + Consulting with experts and considering existing experience with cloud platforms

Setup and Configuration: 30 hours

* + Creating an AWS account and setting up billing and access control mechanisms
  + Provisioning the required AWS resources, including virtual machines, storage, and networking components
  + Configuring the cloud environment to meet the specific needs of the DBMS project
  + Integrating the DBMS with other cloud services, such as Amazon RDS (Relational Database Service) or Amazon Aurora

Cost Estimate:

AWS Subscription Cost: Depends on chosen services (breakdown per service)

Amazon RDS:

* + Shared Instances: Starting from $0.005 per hour
  + Dedicated Instances: Starting from $0.062 per hour

Amazon Aurora:

* + Single-AZ: Starting from $0.055 per hour
  + Multi-AZ: Starting from $0.065 per hour

Amazon Elastic Block Store (EBS):

* + Standard Storage: $0.005 per GB per month
  + Cold Storage: $0.0012 per GB per month

Data Transfer:

* + Outbound data transfer to the internet: Varies depending on the region
  + Inbound data transfer from the internet: Free for the first 100 GB per month, then varies depending on the region

**Step 3: Database Design Implementation:** Including detailed tasks, estimates, and costs

Technical Overview: Step 3 focuses on translating the finalized design into a database schema and implementing it. It involves 40 hours for schema translation, mapping the design into tables, relationships, and constraints. Following this, 60 hours are dedicated to physically setting up the database according to the schema, ensuring data integrity and performance through rigorous testing. Estimated costs include around $500 for database design tools and approximately $200/year for developer tool subscriptions, facilitating efficient schema translation and implementation for a robust foundation for TUGO.

Tools and Software:

Database Design Tools:

* Tool: ER/Studio Data Architect
* Estimated Cost: Approximately $1,500 (one-time purchase)

Developer Tools Subscription:

* + Tool: SQL Server Management Studio (SSMS)
  + Estimated Cost: Approximately $200/year (for subscription renewal)

**Breakdown of Implementation:**

Translating Design to Database Schema (40 hours):

* + Use ER/Studio Data Architect to translate the logical design into a detailed database schema.
  + Define tables, relationships, constraints, and data types according to the design specifications.
  + Collaborate with the development team to ensure alignment with software requirements.

Implementing Schema (60 hours):

* + Utilize SQL Server Management Studio (SSMS) for physical implementation of the schema.
  + Set up tables, fields, indexes, primary/foreign keys based on the generated schema.
  + Conduct extensive testing to ensure data integrity, performance, and adherence to specifications.

**Cost Estimates:**

ER/Studio Data Architect:

* + Cost: Approximately $1,500 (one-time purchase)

SQL Server Management Studio (SSMS) Subscription:

* + Cost: Approximately $200/year (for subscription renewal)

**Resource Allocation:**

Person-Hours Estimate:

* 100 hours (40 hours for schema translation, 60 hours for schema implementation)

Tools and Software Costs:

* Approximately $1,700 (including one-time purchase and yearly subscription)

Timeline and Quality Assurance:

* + Allocate time for translation and implementation, allowing for testing and refinement.
  + Ensure rigorous testing to validate functionality, performance, and adherence to specifications.

This plan outlines the tasks, technical tools, estimated costs, and resource allocation required for efficiently translating the design into a database schema using ER/Studio Data Architect and implementing it with SQL Server Management Studio (SSMS), ensuring a robust foundation for subsequent development phases.

**Step 4: Application Integration and Testing**

Technical Overview:

Application integration and testing play a crucial role in ensuring the seamless interaction between the DBMS and the applications that rely on it. This step involves establishing connections between the DBMS and the applications, implementing data transfer mechanisms, and thoroughly testing the integrated system to ensure data integrity and functionality for TUGO.

Person-Hours Estimate:

Integration with Applications: 80 hours

* + Data Mapping: 20 hours
    - Identifying and mapping data elements between the DBMS and the applications
    - Defining data transformation rules to ensure compatibility between different data formats
  + Connection Establishment: 20 hours
    - Establishing secure and reliable connections between the DBMS and the applications
    - Configuring connection parameters, such as connection strings, authentication credentials, and connection pool settings
  + Data Transfer Implementation: 40 hours
    - Implementing data exchange mechanisms between the DBMS and the applications
    - Utilizing appropriate protocols, such as JDBC, ODBC, or web services, to facilitate data transfer
    - Handling data synchronization and conflict resolution issues

Testing and Debugging: 100 hours

* + Functional Testing: 40 hours
    - Validating the functionality of the integrated system
    - Ensuring that data is accurately transferred between the DBMS and the applications
    - Verifying that the applications can correctly read, modify, and delete data in the DBMS
  + Performance Testing: 30 hours
    - Evaluating the performance of the integrated system under load
    - Identifying and addressing performance bottlenecks
    - Optimizing data transfer and processing to ensure scalability
  + Security Testing: 30 hours
    - Assessing the security vulnerabilities of the integrated system
    - Implementing security measures to protect data from unauthorized access, modification, or deletion
    - Conducting penetration testing to identify and remediate security weaknesses

**Cost Estimate:**

Testing Tools/Subscriptions: Approx. $300

* + Subscription to automated testing tools: $100 per month
  + Purchase of specialized testing software: $200 per license

**Technical Considerations:**

Integration Approach:

* + Point-to-Point Integration: Establishing direct connections between the DBMS and each individual application
  + Enterprise Service Bus (ESB): Implementing a centralized messaging platform to facilitate data exchange between the DBMS and multiple applications
  + Application Programming Interfaces (APIs): Utilizing APIs to expose DBMS functionality to applications

Data Transfer Mechanisms:

* + Batch Data Transfer: Transferring large volumes of data in bulk at specific intervals
  + Real-time Data Streaming: Transferring data immediately as it is generated or modified
  + Event-driven Data Transfer: Triggering data transfer based on specific events or changes in the DBMS

Testing Framework:

* + Unit Testing: Testing individual components of the integrated system in isolation
  + Integration Testing: Testing the interactions between different components of the integrated system
  + System Testing: Testing the entire integrated system as a whole

By carefully considering the technical aspects of application integration and testing, Tugo can ensure that the DBMS is seamlessly integrated with the applications, providing a reliable and secure data management solution.

**Step 5: Deployment and Maintenance**

Technical Overview:

Deploying and maintaining the DBMS is crucial for ensuring its long-term operation and continuous availability. This step involves planning the deployment process, configuring the DBMS environment, monitoring system performance, and implementing regular maintenance tasks to ensure data integrity, security, and performance optimization.

Person-Hours Estimate:

Deployment Planning: 20 hours

* + Environment Preparation: 10 hours
    - Configuring the target environment, including the operating system, network settings, and security measures
    - Provisioning necessary hardware resources, such as servers, storage devices, and network infrastructure
  + Deployment Strategy: 5 hours
    - Determining the appropriate deployment strategy, such as in-place upgrade, rolling upgrade, or new deployment
    - Developing a detailed deployment plan that outlines the steps, tasks, and dependencies
  + Rollback Plan: 5 hours
    - Defining a rollback plan in case of unforeseen issues during deployment
    - Documenting the steps required to revert to the previous state of the DBMS

Ongoing Maintenance (monthly): 40 hours/month

* + Performance Monitoring: 10 hours/month
    - Continuously monitoring system performance metrics, such as CPU usage, memory consumption, and I/O operations
    - Identifying and addressing performance bottlenecks to maintain optimal system responsiveness
  + Data Integrity Checks: 5 hours/month
    - Verifying the integrity of data stored in the DBMS
    - Implementing data validation procedures to ensure data accuracy and consistency
  + Security Patching: 5 hours/month
    - Regularly applying security patches and updates to the DBMS and its components
    - Reviewing security vulnerabilities and implementing appropriate mitigation measures
  + Change Management: 10 hours/month
    - Establishing a change management process to control and track modifications to the DBMS environment
    - Ensuring that changes are properly documented, tested, and implemented with minimal disruption to system operations
  + Backups and Disaster Recovery: 10 hours/month
    - Implementing a robust backup strategy to protect data in case of hardware failures or other disruptions
    - Maintaining a disaster recovery plan to quickly restore the DBMS in the event of a major outage

**Cost Estimate**:

Hardware/Server Costs (if applicable): Varies based on chosen specifications

* + Server Hardware: Cost depends on factors such as CPU, memory, storage capacity, and network bandwidth
  + Storage Components: Cost depends on storage type (HDD, SSD), capacity, and performance requirements
  + Networking Infrastructure: Cost depends on network speed, scalability, and security requirements
  + We came up with following costs:
    - Server Hardware: $1,000 - $5,000
    - Storage: $500 - $2,000
    - Network Infrastructure: $200 - $1,000

By carefully considering the technical aspects of deployment and maintenance, TUGO can ensure that the DBMS is deployed and maintained in a way that optimizes performance, ensures data integrity, and protects against security threats, providing a reliable and secure data management solution for the long term.

**Visual Cost Summary**:

|  |  |  |  |
| --- | --- | --- | --- |
| Cost Category | Person/Hour | Total Labor Amount | Monetary Costs ( in $) |
| Pre-Implementation | 40 hours | 160 hours | 10400 |
| Requirements Gathering and Analysis | 10 | 40 | 2600 |
| Project Planning and Design | 15 | 90 | 5850 |
| Vendor Selection and Evaluation | 5 | 25 | 1625 |
| Data Modeling and Design | 10 | 35 | 2275 |
| Implementation | 60 hours | 240 hours | 15600 |
| Cloud Provider Setup | 8 | 32 | 2080 |
| DBMS Installation and Configuration | 15 | 90 | 5850 |
| Data Migration and Loading | 10 | 60 | 3900 |
| Application Integration and Testing | 20 | 160 | 10400 |
| Deployment and Go-Live | 7 | 48 | 3120 |
| Maintenance (per month) | 10 hours | 40 hours | 2600 |
| Performance Monitoring | 2.5 | 10 | 650 |
| Data Integrity Checks | 1.25 | 5 | 325 |
| Security Patching | 1.25 | 5 | 325 |
| Change Management | 2.5 | 10 | 650 |
| Backups and Disaster Recovery | 2.5 | 10 | 650 |
| Hardware |  |  |  |
| On-premises Hardware |  |  | $1,000 - $5,000 |
| Cloud Computing Hardware |  |  | $500 - $1,000 per month |
| Software/Tools |  |  | Approx. $1000 (initial setup) |
| Personnel Costs |  |  | Calculated based on hourly rates and estimated hours |

**References:**

* AWS Pricing Calculator: [AWS Pricing](<https://aws.amazon.com/pricing/)>- Database Design Tools: (e.g., MySQL Workbench, ER/Studio)  
  - Developer Tools: (e.g., Visual Studio Code, IntelliJ IDEA)  
  - Testing Tools: (e.g., Selenium, JUnit)  
  - Personnel Hourly Rates: Industry Standards and Local Employment Lawss

# APPENDIX A

Undertaking the Enterprise Data Management course has been a multifaceted learning journey, encompassing both theoretical concepts and hands-on applications. The culmination of this educational experience materialized in the development of a project for Tugo Bikeshare, where various aspects of data management and web development were applied. The project not only served as a practical application of acquired knowledge but also provided a platform to implement twelve complex queries aimed at offering crucial business insights to the administrative team.

The foundational lessons learned spanned diverse areas of data management, starting with the creation of Entity-Relationship (ER) diagrams and their conversion into relational schemas. Understanding the intricate relationships between entities and tables was crucial in establishing a robust and scalable database structure. The application of normalization techniques further underscored the importance of minimizing redundancy and ensuring optimal database performance.

A pivotal aspect of the project involved database optimization, where the delicate balance between normalization and performance enhancement became evident. Creating clustered and non-clustered indexes significantly accelerated query performance, showcasing the practical implications of theoretical concepts. The development of stored procedures and triggers emerged as valuable skills, contributing not only to enhanced data integrity but also to the security of the system.

The utilization of a diverse technological stack, including HTML, CSS, JavaScript, Python Flask, and Oracle Database, provided a comprehensive understanding of web development and database connectivity. Hosting the project on AWS EC2 introduced the practical aspects of cloud computing, emphasizing the significance of scalable and secure cloud infrastructure.

Jinja2 templating for rendering dynamic content added sophistication to the project, enhancing the user experience through efficient integration of server-side logic with HTML templates. Beyond individual project components, collaborative learning from presentations by other groups enriched the overall experience. Exposure to alternative approaches, challenges faced, and innovative solutions broadened perspectives and fostered a collaborative learning environment.

In addition to these foundational aspects, a noteworthy accomplishment was the development of twelve complex queries aimed at providing essential business insights. These queries, designed to extract meaningful information from the database, cover various business metrics such as user activity patterns, popular bike stations, and peak usage times. This achievement not only demonstrates proficiency in SQL but also positions the project as a valuable tool for the administration to make informed, data-driven decisions.

In conclusion, the Enterprise Data Management project, with its diverse facets, serves as a comprehensive showcase of acquired knowledge and practical skills. From conceptualizing ER diagrams to deploying a functional web application on AWS and implementing complex queries, the project not only contributes to technical proficiency but also positions itself as a valuable resource for strategic decision-making within the Tugo Bikeshare administration. The collaborative learning environment, combined with the practical applications, has laid a robust foundation for navigating the dynamic landscape of enterprise data management.