



College of Engineering

University of Missouri

CMP'SC 4380/7380
Database Management Systems - I
Spring 2018

FINAL PROJECT REPORT

Lawn Management Executive Dashboard

GROUP 1

<i>Author(s):</i>	<i>Pawprint(s):</i>	<i>Student-ID(s):</i>
Benjavicha Hotrabhavannada	bhv39	14170746
Nijaporn Hotrabhavananda	nhqp8	14149399
Aaron Thomas	athc5	14232199
John Umstead	jbup7c	12435519

TABLE OF CONTENTS

1	INTRODUCTION.....	2
1.1	PROJECT URL	2
1.2	USERNAMES AND PASSWORDS	2
2	ENTITY RELATIONSHIP DIAGRAM.....	3
2.1	ERD.....	3
3	QUERIES	16
3.1	USER MANAGEMENT AND LOGIN SYSTEMS	16
3.2	DASHBOARD DISPLAY	20
3.3	COMPANY PERFORMANCE AND GROSS PROFIT	21
3.4	LEADS (POTENTIAL CUSTOMERS)	23
3.5	EXPENSES.....	25
3.6	TICKET.....	28
3.7	PURCHASE ORDERS.....	30
3.8	ISSUES	32
3.9	EMPLOYEES.....	35
3.10	CLIENTS	35
4	ANALYTICS:.....	36
	COMPANY PERFORMANCE.....	36
4.1	FIND THE CURRENT COMPANY PROFIT MARGIN	36
4.2	RETRIEVE THE TOTAL REVENUE AND EXPENSE ON EACH YEAR.....	36
4.3	CONTRIBUTION MARGIN OVER YEARS.....	37
4.4	ACCURACY OF ALL BRANCHES.....	38
4.5	OPEN-ISSUE-PRIORITY LEVELS.....	39
5	NORMALIZATION:.....	40
5.1	NORMALIZATION	40
6	INDEXING SELECTION:.....	41
7	OPTIMIZATION AND TUNING:	43
8	SECURITY SETTING:.....	45
8.1	MANDATORY ACCESS CONTROL IN THE FRONT-END.....	45
8.2	LIMITED VIEW OF CLIENT INFORMATION FOR PARTICULAR TYPE OF USER.....	45
9	OTHER TOPICS:	47
9.1	TRIGGER	47
10	USER'S MANUAL:	48

1 Introduction

Our client for this final project is The Integra Group, a software development firm based out of Chesterfield, Missouri. Their main product that they have created and continually update is their BOSS system. The project they gave us is to create a prototype of an executive dashboard for their clients that will allow them to view various analytics about gross margin, profitability, sales, etc. This prototype is to help Integra visualize a final dashboard that they will use across all of their industries; we will be creating the dashboard based off of a lawn management company's data.

To build our database, Integra is sending us csv files for issues, purchase orders, tickets, contracts, and leads pulled from their program. We will be able to showcase what statistics, for each of these areas, are more prominent by service, property, branch, industry, etc. along with internal statistics about gross margin, profits/expenses, sales, etc.

1.1 Project URL

<https://web.dsa.missouri.edu/~bhv39/Executive/index.php>

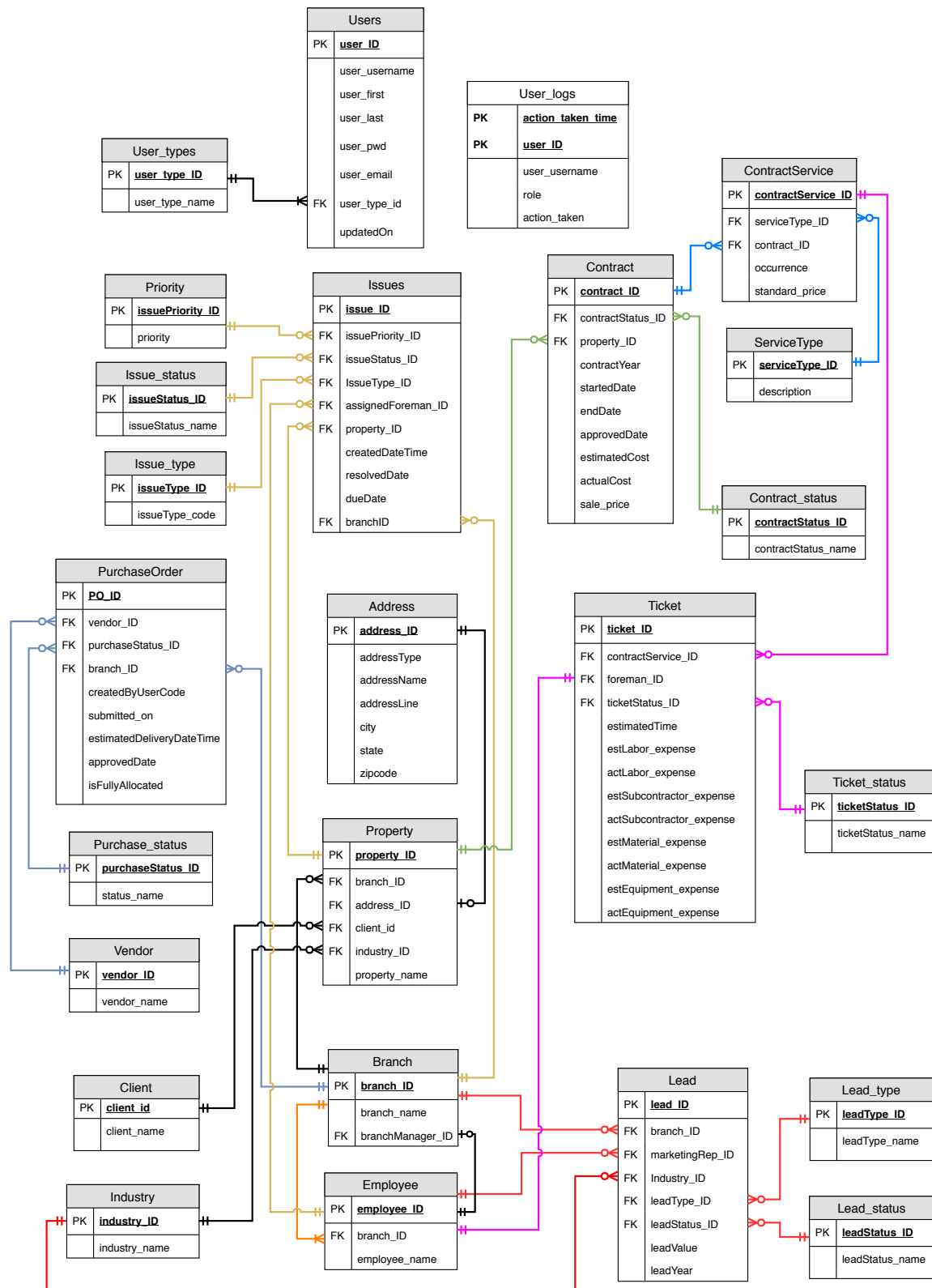
1.2 Usernames and Passwords

User Type	Administrator	Executive Manager	Account Manager
Username:	admin	executive	account
Password	pwd	pwd	pwd

*Notice: username and password are in all lowercase characters.

2 Entity Relationship Diagram

2.1 ERD



2.2 Data Definition Language (DDL)

2.2.1 User Types

```
CREATE TABLE s18group01.User_types (  
    user_type_ID SERIAL,  
    user_type_name VARCHAR(40) NOT NULL,  
    PRIMARY KEY (user_type_ID)  
);
```

- **Description:**

- I. The *User_types* table is created first to store the distinct type of users. Furthermore, it would provide ability to database administrator to give a certain right access to a specific type of user for the security concern. The table is considered as an independent entity and it contains the fields as follows.
- II. *user_type_ID* is a unique identifier for the type of users.
- III. *user_type_name* contains type name of user, e.g., admin, executive officer, account manager

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on *user_type_ID* to uniquely identify each type of user in the table
 - II. *user_type_name* is NOT NULL to enforce the column to specify what the user type name is.
-

2.2.2 Users

```
CREATE TABLE s18group01.Users (  
    user_ID SERIAL,  
    user_username VARCHAR(25) NOT NULL,  
    user_first VARCHAR(60) NOT NULL,  
    user_last VARCHAR(60) NOT NULL,  
    user_pwd VARCHAR(70) NOT NULL,  
    user_email VARCHAR(256) NOT NULL,  
    user_type_ID INT NOT NULL,  
    updatedOn TIMESTAMP DEFAULT (CURRENT_TIMESTAMP::timestamp(0)),  
    PRIMARY KEY (user_ID),  
    UNIQUE(user_username),  
    FOREIGN KEY (user_type_ID) REFERENCES User_types (user_type_ID) ON  
    DELETE CASCADE  
);
```

- **Description:**

- I. The *Users* table contains a set of all users who can access the web application system along with their basic information such as followings.
- II. *user_ID* is a unique identifier of users. The column is defined with SERIAL data type. Thus, new created user will get id value assigned in a sequence of integers.
- III. *user_username* is defined as unique of users.
- IV. *user_first* stores each user's first name.

- V. *user_last* stores each user's last name.
- VI. *user_pwd* stores each user's secure password and defined with NOT NULL constraint to ensure that every user has their password set.
- VII. *user_email* stores each user's email address.
- VIII. *user_type_ID* indicates the identifier of role of users.
- IX. *updatedOn* stores the current timestamp when the data is being either inserted or modified.

- **Constraint(s):**

- I. PRIMARY KEY is defined on *user_ID* column to prevent duplicate values from being inserted.
- II. UNIQUE KEY constraint is defined on *user_username* to uniquely identify each user with their record in the table.
- III. NOT NULL is defined on *user_username*, *user_first*, *user_last*, *user_pwd*, *user_email* and *user_type_ID*.
- IV. FOREIGN KEY constraint defined on *user_type_ID* to reference the *user_type_ID* in the *user_type* table.
- V. Users can only have exactly one user type.

2.2.3 User Logs

```
CREATE TABLE sl8group01.User_logs (
    user_ID INT NOT NULL,
    action_taken_time TIMESTAMP NOT NULL DEFAULT
(CURRENT_TIMESTAMP::timestamp(0)),
    user_username varchar(100) NOT NULL,
    role varchar(15),
    action_taken TEXT NOT NULL,
    PRIMARY KEY (user_ID, action_taken_time)
);
```

- **Description:**

- I. The *User_logs* table record a set of all logs of action of from every users who have committed any changes, e.g., insert, update, delete, etc. in the database. Consequently, it allows an admin to monitor such events including himself/herself log activity.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on *user_ID* and *action_taken_time* as a composite key to uniquely identify each log record.
 - II. NOT NULL is defined on
-

2.2.4 Address

```
CREATE TABLE s18group01.Address (  
    address_ID INT,  
    addressType VARCHAR(20),  
    addressName VARCHAR(60) NOT NULL,  
    city VARCHAR(30) NOT NULL,  
    state VARCHAR(14) NOT NULL,  
    zipcode VARCHAR(12) NOT NULL,  
    addressLine varchar(200) NOT NULL,  
    PRIMARY KEY (address_ID)  
);
```

- **Description:**

II. This table contains information about the address for each client's property.

- **Constraint(s):**

III. The PRIMARY KEY constraint is defined on address_ID to uniquely identify each address in the table

IV. NOT NULL is defined on addressName, city, state, zipcode, addressLine

2.2.5 Branch

```
CREATE TABLE s18group01.Branch (  
    branch_ID SERIAL,  
    branch_name VARCHAR(60) NOT NULL,  
    branchManager_ID INT NOT NULL,  
    PRIMARY KEY (branch_ID),  
    FOREIGN KEY (branchManager_ID) REFERENCES Employee(employee_ID)  
);
```

- **Description:**

I. The Branch table stores the list of branches of a lawn company has, including its branch's name and manager who manages the branch.

- **Constraint(s):**

I. The PRIMARY KEY constraint is defined on branch_ID to uniquely identify each branch in the table

II. NOT NULL is defined on branch_name and branchManager_ID.

III. FOREIGN KEY constraint defined on branchManager_ID to reference the employee_ID in the Employee table.

IV. A branch could only have one manager managed by and an employee could work as a manager at most at one branch.

2.2.6 Employee

```
CREATE TABLE s18group01.Employee (  
    employee_ID SERIAL,  
    employee_name VARCHAR(60) NOT NULL,  
    branch_ID INT NOT NULL,  
    PRIMARY KEY (employee_ID),  
    FOREIGN KEY (branch_ID) REFERENCES Branch(branch_ID)  
);
```

- **Description:**

- I. The table contains list of employees who work for the lawn company, his/her name and the branch_ID he/she works in.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on employee_ID to uniquely identify each employee.
 - II. NOT NULL is defined on employee_name and branch_ID.
 - III. FOREIGN KEY constraint defined on branch_ID to reference the branch_ID in the Branch table.
 - IV. Employee can work for exactly one branch and one branch should have a least one of employee working in.
-

2.2.7 Client

```
CREATE TABLE s18group01.Client (  
    client_ID SERIAL,  
    client_name VARCHAR(60) NOT NULL,  
    PRIMARY KEY (client_ID)  
);
```

- **Description:**

- I. This table contains a list of clients of the lawn company including the client's name.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on client_ID to uniquely identify each client in the table.
 - II. NOT NULL is defined on client_name.
-

2.2.8 Industry

```
CREATE TABLE s18group01.Industry (  
    industry_ID SERIAL,  
    industry_name VARCHAR(60) NOT NULL,  
    PRIMARY KEY (industry_ID)  
);
```

- **Description:**

- I. This table contains a list of industry which each property or lead is in, such as residential, commercial, hospital, restaurant, municipality, etc.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on industry_ID to uniquely identify each type industry in the table.
 - II. NOT NULL is defined on industry_name.
-

2.2.9 Property

```
CREATE TABLE s18group01.Property (  
    property_ID SERIAL,  
    branch_ID INT NOT NULL,  
    address_ID INT NOT NULL,  
    client_ID INT NOT NULL,  
    industry_ID INT NOT NULL,  
    property_name VARCHAR(60),  
    PRIMARY KEY (property_ID),  
    FOREIGN KEY (branch_ID) REFERENCES Branch(branch_ID),  
    FOREIGN KEY (address_ID) REFERENCES Address(address_ID),  
    FOREIGN KEY (client_ID) REFERENCES Client(client_ID),  
    FOREIGN KEY (industry_ID) REFERENCES Industry(industry_ID)  
);
```

- **Description:**

- I. This table contains the list of property of clients of lawn company with other necessary information.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on property_ID to uniquely identify each property in the table
 - II. NOT NULL is defined on branch_ID, address_ID, client_ID, industry_ID.
 - III. FOREIGN KEY constraint is defined on branch_ID, address_ID, client_ID, industry_ID.
 - IV. One property can only be serviced by exactly one branch. One branch could have zero or many properties in control.
 - V. Each property must have exactly one address.
 - VI. Each property must own by exactly one client. A client can own zero or multiple properties.
 - VII. Each property must be categorized in one type of industry.
-

2.2.10 Vendor

```
CREATE TABLE s18group01.Vendor (  
    vendor_ID SERIAL,  
    vendor_name VARCHAR(40) NOT NULL,  
    PRIMARY KEY (vendor_ID)  
);
```

- **Description:**

- I. This table contains a list of local vendors where the lawn company has purchased material from.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on vendor_ID to uniquely identify each vendor.
 - II. NOT NULL is defined on vendor_name.
-

2.2.11 Purchase Status

```
CREATE TABLE s18group01.Purchase_status (  
    purchaseStatus_ID SERIAL,  
    status VARCHAR(40) NOT NULL,  
    PRIMARY KEY (purchaseStatus_ID)  
);
```

- **Description:**

- I. This table is created to list the purchase status of the purchase order whether the order is new, approved, complete, void, etc.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on purchaseStatus_ID to uniquely identify each purchase_status
 - II. NOT NULL is defined on status name.
-

2.2.12 Purchase Order

```
CREATE TABLE s18group01.PurchaseOrder (  
    PO_ID INT,  
    vendor_ID INT NOT NULL,  
    purchaseStatus_ID INT NOT NULL,  
    created_by_user_code CHAR(4) NOT NULL,  
    submitted_on TIMESTAMP NOT NULL,  
    estimated_delivery_date TIMESTAMP,  
    approved_on TIMESTAMP,  
    branch_ID INT NOT NULL,  
    is_fully_allocated BOOLEAN NOT NULL,  
    PRIMARY KEY (PO_ID),  
    FOREIGN KEY (vendor_ID) REFERENCES Vendor (vendor_ID),  
    FOREIGN KEY (purchaseStatus_ID) REFERENCES Purchase_status  
(purchaseStatus_ID),  
    FOREIGN KEY (branch_ID) REFERENCES Branch (branch_ID)  
);
```

- **Description:**

- I. This table stores a list of purchase order that has been added to the database.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on POID to uniquely identify each type of purchase order in the table.
 - II. NOT NULL is defined on vendor_ID, purchaseStatus_ID, created_by_user_code, submitted_on, branch_ID, is_fully_allocated.
 - III. Each purchase order must have exactly one vendor. A vendor could supply many purchase order.
 - IV. Each purchase order must have exactly one status specified.
 - V. Each purchase order must have exactly one branch specified.
-

2.2.13 Priority (Issue)

```
CREATE TABLE s18group01.Priority (  
    issuePriority_ID SERIAL,  
    priority VARCHAR(40) NOT NULL,  
    PRIMARY KEY (issuePriority_ID)  
);
```

- **Description:**

- I. This table contains a list of issue priority, i.e., urgent, high, medium, low.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `issuePriority_ID` to uniquely identify each type of issue priority.
 - II. NOT NULL is defined on `priority`.
-

2.2.14 Issue Status

```
CREATE TABLE s18group01.Issue_status (  
    issueStatus_ID SERIAL,  
    issueStatus_name VARCHAR(40) NOT NULL,  
    PRIMARY KEY (issueStatus_ID)  
);
```

- **Description:**

- I. This table contains a list of issue statuses: open, past due and closed.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `issueStatus_ID` to uniquely identify each status of issue.
 - II. NOT NULL is defined on `issueStatus_name`.
-

2.2.15 Issue Type

```
CREATE TABLE s18group01.Issue_type (  
    issueType_ID SERIAL,  
    issueType_code VARCHAR(40) NOT NULL,  
    PRIMARY KEY (issueType_ID)  
);
```

- **Description:**

- I. This table stores a list of issue type such as: internal, customer, audit.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `issueType_ID` to uniquely identify each issue type in the table
 - II. NOT NULL is defined on `issueType_code`.
-

2.2.16 Issues

```
CREATE TABLE s18group01.Issues (  
    issue_ID SERIAL,  
    issuePriority_ID INT NOT NULL,  
    issueStatus_ID INT NOT NULL,  
    issueType_ID INT NOT NULL,  
    assignedForeman_ID INT,  
    property_id INT NOT NULL,  
    createdDateTime TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,  
    resolvedDate TIMESTAMP,  
    dueDate TIMESTAMP,  
    branch_ID INT NOT NULL,  
    PRIMARY KEY (issue_ID),  
    FOREIGN KEY (issuePriority_ID) REFERENCES Priority  
(issuePriority_ID),  
    FOREIGN KEY (issueStatus_ID) REFERENCES Issue_status  
(issueStatus_ID),  
    FOREIGN KEY (issueType_ID) REFERENCES Issue_type (issueType_ID),  
    FOREIGN KEY (assignedForeman_ID) REFERENCES Employee (employee_ID),  
    FOREIGN KEY (property_ID) REFERENCES Property (property_ID),  
    FOREIGN KEY (branch_ID) REFERENCES Branch (branch_ID)  
);
```

- **Description:**

- I. This table contains a list of issues that have to be solved with its priority, status, and type of issue.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on issue_ID to uniquely identify each type issue in the table
 - II. NOT NULL is defined on the attributes shown above.
 - III. FOREIGN KEY is defined on the attributes shown above.
 - IV. Each issue is referred to exactly one branch.
 - V. Each issue can only have one exactly priority specified.
 - VI. Each issue can only have one exactly issue status specified.
 - VII. Each issue can only have one exactly issue type specified.
 - VIII. Each issue can only be assigned by one exactly employee. Each employee can work on zero or many issues.
 - IX. Each issue can only refer to exactly one property. But one property could have zero or many issues reported.
-

2.2.17 Service Type

```
CREATE TABLE s18group01.ServiceType (  
    serviceType_ID SERIAL,  
    description VARCHAR(250) NOT NULL,  
    PRIMARY KEY (serviceType_ID)  
);
```

- **Description:**

- I. This table contains a list of 228 services the lawn company can provide such as salting driveways & Sidewalks, shoveling sidewalks, and etc.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on serviceType_ID to uniquely identify each type service in the table.
 - II. NOT NULL is defined on description.
-

2.2.18 Ticket Status

```
CREATE TABLE s18group01.Ticket_status (
    ticketStatus_ID SERIAL,
    ticketStatus_name VARCHAR(30) NOT NULL,
    PRIMARY KEY (ticketStatus_ID)
);
```

- **Description:**

- I. This table contains a list of ticket status such as: new, open, billed, and etc.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on ticketStatus_ID to uniquely identify each ticket status.
 - II. NOT NULL is defined on ticketStatus_name.
-

2.2.19 Ticket

```
CREATE TABLE s18group01.Ticket (
    ticket_ID SERIAL,
    contractService_ID INT NOT NULL,
    foreman_ID INT NOT NULL,
    ticketStatus_ID INT NOT NULL,
    estHours NUMERIC(8,2),
    estLabor_expense NUMERIC(12,2),
    actLabor_expense NUMERIC(12,2),
    estSubcontractor_expense NUMERIC(12,2),
    actSubcontractor_expense NUMERIC(12,2),
    estMaterial_expense NUMERIC(12,2),
    actMaterial_expense NUMERIC(12,2),
    estEquipment_expense NUMERIC(12,2),
    actEquipment_expense NUMERIC(12,2),
    PRIMARY KEY (ticket_ID),
    FOREIGN KEY (contractService_ID) REFERENCES ContractService
(contractService_ID),
    FOREIGN KEY (foreman_ID) REFERENCES Employee (employee_ID),
    FOREIGN KEY (ticketStatus_ID) REFERENCES Ticket_status
(ticketStatus_ID)
);
```

- **Description:**

- I. This table is storing every tickets or receipt of the customer to show how everything allocated.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on ticket_ID to uniquely identify each type of ... in the table

- II. NOT NULL is defined on `contractService_ID`, `foreman_ID`, `ticketStatus_ID`.
 - III. FOREIGN KEY is defined on the attributes shown above.
 - IV. Each ticket must have exactly one contract service.
 - V. Each ticket must have exactly one foreman or employee assigned to be responsible for.
 - VI. Each ticket must have exactly one ticket status.
-

2.2.20 Contract Status

```
CREATE TABLE s18group01.Contract_status (
    contractStatus_ID SERIAL,
    contractStatus_name VARCHAR(40) NOT NULL,
    PRIMARY KEY (contractStatus_ID)
);
```

- **Description:**

- I. This table contains a list of contract status such as approved, complete, signed, and etc.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `contractStatus_ID` to uniquely identify each type of contract status in the table.
 - II. NOT NULL is defined on `contractStatus_name`.
-

2.2.21 Contract

```
CREATE TABLE s18group01.Contract (
    contract_ID SERIAL,
    contractStatus_ID INT NOT NULL,
    property_ID INT NOT NULL,
    contractYear INT,
    startedDate DATE,
    endDate DATE,
    approvalDate DATE,
    estimatedCost NUMERIC(12,2),
    actualCost NUMERIC(12,2),
    sale_price NUMERIC(12,2),
    PRIMARY KEY (contract_ID),
    FOREIGN KEY (contractStatus_ID) REFERENCES Contract_status
(contractStatus_ID),
    FOREIGN KEY (property_ID) REFERENCES Property (property_ID)
);
```

- **Description:**

- I. This table contains a list of contracts the lawn company has been signed with clients along with other information shown in the above schema.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `contract_ID` to uniquely identify each contract in the table.
- II. NOT NULL is defined on `contractStatus_ID`, `property_ID`.
- III. Each contract must have exactly one contract status.

- IV. Each contract must have exactly one property assigned to it.
-

2.2.22 Contract Service

```
CREATE TABLE s18group01.ContractService (  
    contractService_ID SERIAL,  
    serviceType_ID INT NOT NULL,  
    contract_ID INT NOT NULL,  
    occurrence INT,  
    standard_price NUMERIC(12,2),  
    PRIMARY KEY (contractService_ID),  
    FOREIGN KEY (serviceType_ID) REFERENCES ServiceType  
    (serviceType_ID),  
    FOREIGN KEY (contract_ID) REFERENCES Contract (contract_ID)  
);
```

- **Description:**

- I. This table contains the list of all services the company should be provided to every contracts.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `contractService_ID` to uniquely identify each service.
- II. NOT NULL is defined on `serviceType_ID`, `contract_ID`.
- III. Each contract service must have exactly one contract assigned to.
- IV. Each contract service must have exactly one service type.
-

2.2.23 Lead Type

```
CREATE TABLE s18group01.Lead_type (  
    leadType_ID SERIAL,  
    leadType_name VARCHAR(30) NOT NULL,  
    PRIMARY KEY (leadType_ID)  
);
```

- **Description:**

- I. This table contains the list of the types of lead such as lawn maintenance, retaining walls, residential maintenance, tree cares, and etc.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `leadType_ID` to uniquely identify each lead type.
- II. NOT NULL is defined on `leadType_name`.
-

2.2.24 Lead Status

```
CREATE TABLE s18group01.Lead_status (  
    leadStatus_ID SERIAL,  
    leadStatus_name VARCHAR(30) NOT NULL,  
    PRIMARY KEY (leadStatus_ID)  
);
```

- **Description:**

- I. This table stores the lead status type such as complete, new, disqualified, qualified.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `leadStatus_ID` to uniquely identify each lead status.
 - II. NOT NULL is defined on `leadStatus_name`.
-

2.2.25 Lead

```
CREATE TABLE s18group01.Lead (  
    lead_ID SERIAL,  
    branch_ID INT NOT NULL,  
    marketingRep_id INT NOT NULL,  
    industry_ID INT NOT NULL,  
    leadType_ID INT NOT NULL,  
    leadStatus_ID INT NOT NULL,  
    leadValue NUMERIC(12,2),  
    leadYear int,  
    PRIMARY KEY (lead_ID),  
    FOREIGN KEY (branch_ID) REFERENCES Branch (branch_ID),  
    FOREIGN KEY (marketingRep_id) REFERENCES Employee (employee_ID),  
    FOREIGN KEY (industry_ID) REFERENCES Industry (industry_ID),  
    FOREIGN KEY (leadType_ID) REFERENCES Lead_type (leadType_ID),  
    FOREIGN KEY (leadStatus_ID) REFERENCES Lead_status (leadStatus_ID)  
);
```

- **Description:**

- I. This table contains a list of contacts with potential customers (business) or leads.

- **Constraint(s):**

- I. The PRIMARY KEY constraint is defined on `lead_ID` to uniquely identify each lead in the table.
 - II. NOT NULL is defined on `branch_ID`, `marketingRep_id`, `industry_ID`, `leadType_ID`, `leadStatus_ID`.
 - III. Each lead must contain exactly one lead type.
 - IV. Each lead must contain exactly one lead status.
 - V. Each lead must be in exactly one industry.
 - VI. Each lead must have exactly one marketing representative or employee assigned to.
 - VII. Each lead must contain exactly one branch.
-

3 Queries

List at least 20 useful queries (with justification) for your system in natural language and SQL.

3.1 User Management and Login Systems

Query 1: Check the existence of the username or email

```
SELECT *
FROM users U JOIN User_types UT ON U.user_type_ID = UT.user_type_ID
WHERE U.user_username = '$username' OR U.user_email = '$username';
```

Justification: If the username or email does not exist, then the login will not be allowed.

Query 2: Insert the log in activity into the user logs table

```
INSERT INTO User_logs (user_ID, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].",
'".$_SESSION['u_type_name']."', '".$_SESSION['u_username']."', 'Logged
in');
```

Justification: Keep a record of each login for potential security problem solving in the future.

Query 3: Users update their own profile with password change

```
UPDATE Users
SET user_first = '$user_first', user_last = '$user_last',
user_email = '$user_email',
user_username = '$user_username', user_pwd = '$hashedPwd'
WHERE user_ID = '$user_ID';
```

Justification: Users can change their own password for better security. Most companies require a password change every 90-120 days.

Query 4: Log user update activity

```
INSERT INTO User_logs (user_ID, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].",
'".$_SESSION['u_type_name']."', '".$_SESSION['u_username']."',
'Update profile and password');
```

Justification: Recording update activity for tracking purposes that may help solve issues in the future.

Query 5: Users update their own profile without password change

```
UPDATE Users
SET user_first = '$user_first', user_last = '$user_last', user_email =
'$user_email',
user_username = '$user_username'
WHERE user_ID = '$user_ID';
```

Justification: Users can update their information, just no change in password at this time.

Query 6: Log user update activity

```
INSERT INTO User_logs (user_ID, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].",
"'.$_SESSION['u_type_name'].'", "'.$_SESSION['u_username'].'",
'Update profile');
```

Justification: Recording update activity for tracking purposes that may help solve issues in the future.

Query 7: display-user-table

```
SELECT *
FROM Users U, User_types Ut WHERE U.user_type_id = Ut.user_type_id;
```

Justification: Show current users of this dashboard.

Query 8: footer - Updated time

```
SELECT max(updatedon::timestamp(0)) FROM Users;
```

Justification: Show current timestamp as an aesthetic attribute.

Query 9: Update User w/ password

```
UPDATE Users
SET user_first = '$user_first', user_last = '$user_last', user_email =
'$user_email', user_type_id = '$user_type_id', user_username =
'$user_username', user_pwd = '$hashedPwd'
WHERE user_ID = '$user_ID';
```

Justification: Update the user login information and hash the password for more protection.

Query 10: log admin update user

```
INSERT INTO User_logs (user_ID, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].", '".$_SESSION['u_type_name'].",
'".$_SESSION['u_username'].", 'Update user ' . $user_ID . "' and
password');
```

Justification: Recording update activity for tracking purposes that may help solve issues in the future.

Query 11: Update User w/out password

```
UPDATE Users SET user_first = '$user_first', user_last = '$user_last',
user_email = '$user_email', user_type_id = '$user_type_id', user_username
= '$user_username'
WHERE user_ID = '$user_ID';
```

Justification: Update the user login information without the password.

Query 12: Log admin update user without password

```
INSERT INTO User_logs (user_ID, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].", '".$_SESSION['u_type_name'].",
'".$_SESSION['u_username'].", 'Update user ' . $user_ID . "' '');
```

Justification: Recording update activity for tracking purposes that may help solve issues in the future.

Query 13: Delete user

```
DELETE FROM Users WHERE user_id = " . $delete_user_id . ";
```

Justification: Removing a user from the website to prevent that user from further accessing the website in the future.

Query 14: log the admin delete user activity

```
INSERT INTO User_logs (user_id, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].", '".$_SESSION['u_type_name'].",
'".$_SESSION['u_username'].", 'Delete user =
' . $delete_user_username . "' '');
```

Justification: Recording delete activity for tracking purposes that may help solve issues in the future.

Query 15: Create a new user

```
INSERT INTO Users (user_username, user_first, user_last, user_pwd,
user_email, user_type_id)
VALUES ('$user_username', '$user_first', '$user_last', '$hashedPwdCheck',
'$user_email', '$user_type_id');
```

Justification: Add new user to the website.

Query 16: Log admin create new user activity

```
INSERT INTO User_logs (user_ID, role, user_username, action_taken)
VALUES (".$_SESSION['u_id'].", "$_SESSION['u_type_name'].",
'".$_SESSION['u_username'].", 'Create a new user:  username = '" .
$user_username . "' '');
```

Justification: Recording create activity for tracking purposes that may help solve issues in the future.

Query 17: display activity log table

```
SELECT *
FROM User_logs
ORDER BY action_taken_time DESC;
```

Justification: Show all records of the activity log for potential problem solving matters.

3.2 Dashboard Display

Query 18: open ticket

```
SELECT count(*) FROM Ticket WHERE ticketStatus_ID = 2;
```

Justification: Initial display to see a glance of the operational tickets of the company.

Query 19: new purchase order

```
SELECT count(*) FROM purchaseOrder WHERE purchaseStatus_ID = 1;
```

Justification: Initial display to see a glance of the new purchase orders that have been created.

Query 20: Open Issue

```
SELECT count(*) FROM Issues WHERE issueStatus_ID = 1;
```

Justification: Initial display to see a glance at the operational open issues created.

Query 21: new lead

```
SELECT count(*) FROM lead WHERE leadstatus_id = 4;
```

Justification: Initial display to see a glance at new potential leads for the company.

Query 22: display Geography chart (location) of the company branches

```
SELECT b.branch_name, count(p.property_ID) AS properties
      FROM branch b, Property P
      WHERE b.branch_ID = p.branch_ID
      GROUP BY b.branch_name;
```

Justification: Show the company where each of their branches are located. For this company, there branches are in missouri, but could be located across state borders.

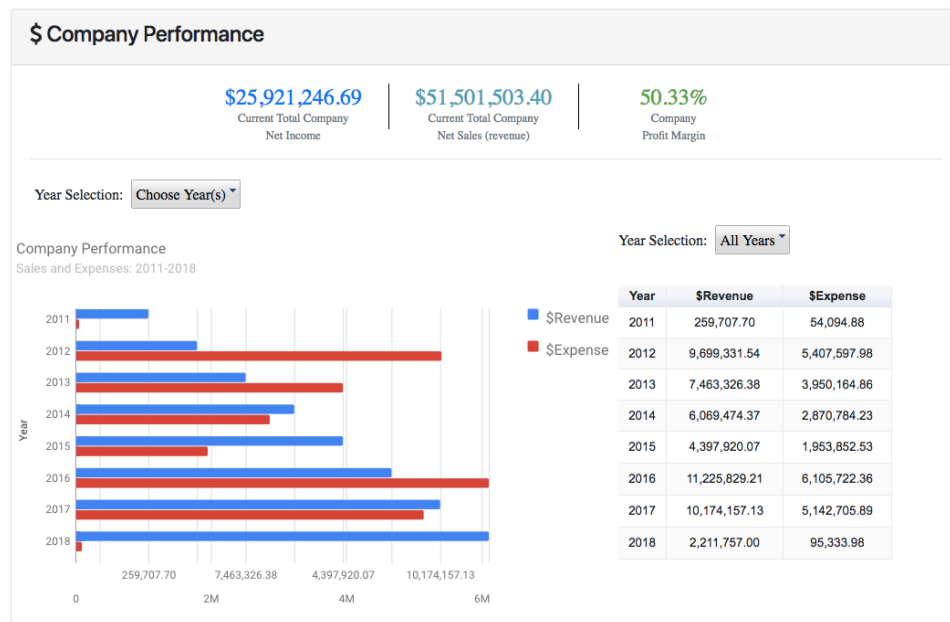
3.3 Company Performance and Gross Profit

Query 23: Find the over all company profit margin

```
SELECT (SUM(sale_price) - SUM(actualCost)), SUM(sale_price),  
       (SUM(sale_price) - SUM(actualCost)) / SUM(sale_price)  
FROM Contract;
```

Justification: Give an overview of the success of the company's history.

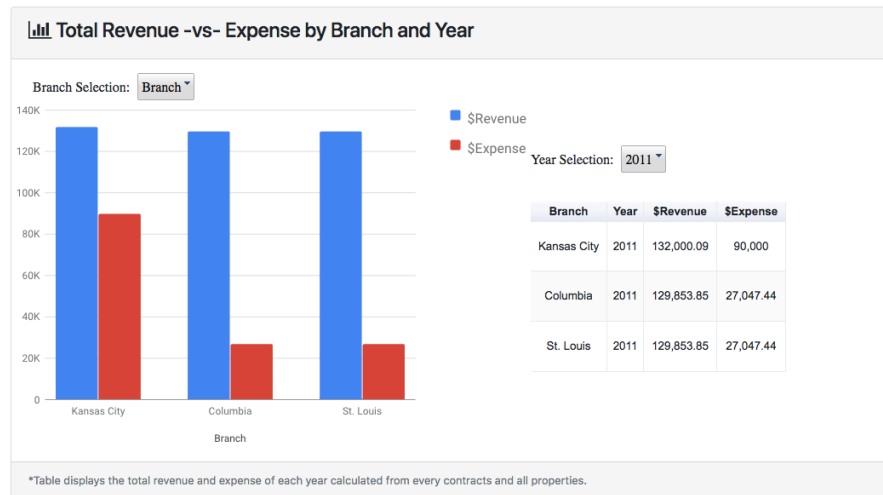
Query 24: Total revenue and expense on each year



```
SELECT contractYear, SUM(sale_price), SUM(actualCost)  
FROM Contract  
GROUP BY contractYear  
ORDER BY ContractYear ASC;
```

Justification: Show how their revenues are accounting for the expenses they are accumulating annually.

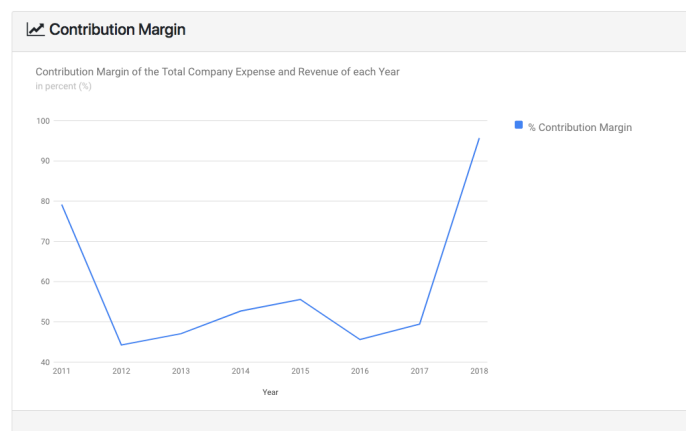
Query 25: Total revenue and expense of each year on different branch branches



```
SELECT B.Branch_name, C.contractYear, SUM(sale_price), SUM(actualCost)
FROM Contract C, Property P, Branch B
WHERE C.property_ID = P.property_ID AND P.branch_ID = B.branch_ID
GROUP BY B.Branch_ID, C.contractYear;
```

Justification: Break down the revenues and expenses for each branch to show how the revenues are accounting for the expenses accumulated annually.

Query 26: Contribution Margin

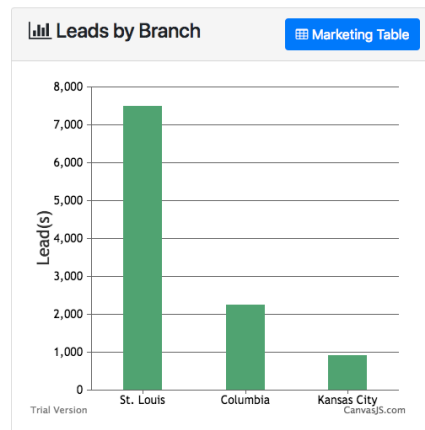


```
SELECT contractyear, (1 - (sum(actualCost)/sum(sale_price))) * 100 AS
Contribution_Margin
FROM Contract
GROUP BY contractyear
ORDER BY contractyear;
```

Justification: Show a yearly contribution margin for yearly results, not overall as above, to see the success of the company each year.

3.4 Leads (Potential Customers)

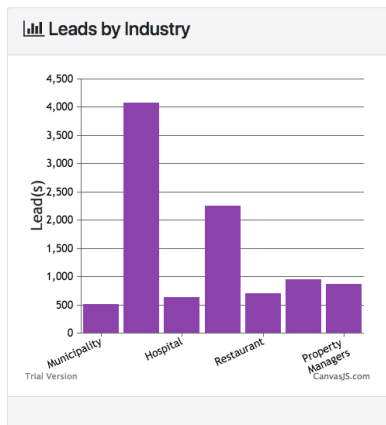
Query 27: Leads by Branch



```
SELECT b.branch_name, count(l.lead_ID)
FROM Lead l, branch b
WHERE l.branch_id = b.branch_id
GROUP BY b.branch_id;
```

Justification: Show how well the company is doing in their sales department with generating new business.

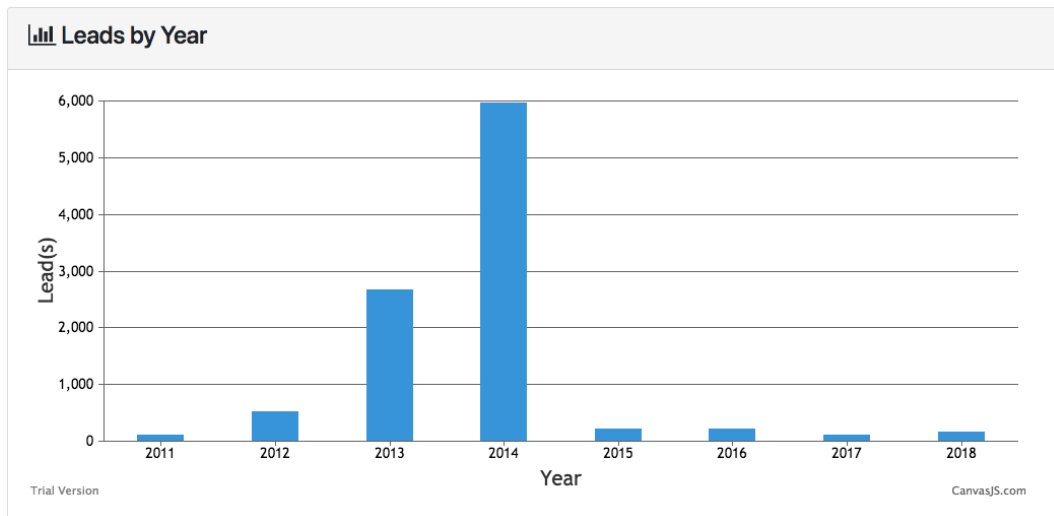
Query 28: Leads by Industry



```
SELECT i.industry_name, count(l.lead_ID)
FROM Lead l, industry i
WHERE l.industry_id = i.industry_id AND l.leadStatus_ID IN (2,4)
GROUP BY i.industry_id;
```

Justification: Show which industries are most lucrative in terms of bringing in new clients.

Query 29: Leads by Year



```
SELECT leadYear, count(lead_ID)
FROM lead
WHERE leadStatus_ID IN (2,4)
GROUP BY leadYear
ORDER BY leadYear ASC;
```

Justification: Give an overview of the success in bringing in new clients annually.

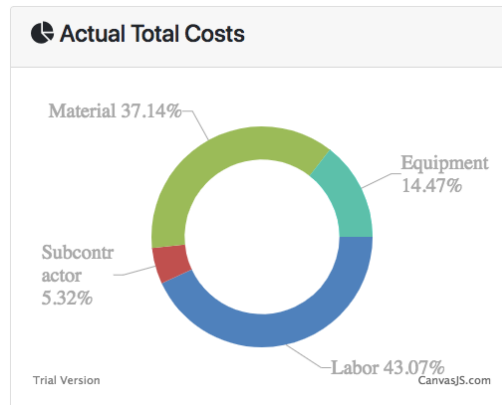
Query 30: Marketing Table

```
SELECT l.lead_ID, b.branch_name, e.employee_name, i.industry_name,
lt.leadType_name, ls.leadStatus_name, l.leadValue, l.leadYear
FROM Lead l
JOIN Branch b ON l.branch_ID = b.branch_ID
JOIN Employee e ON l.marketingRep_ID = e.employee_ID
JOIN Industry i ON l.industry_ID = i.industry_ID
JOIN Lead_type lt ON l.leadType_ID = lt.leadType_ID
JOIN Lead_Status ls ON l.leadStatus_ID = ls.leadStatus_ID
;
```

Justification: For executives, show all information regarding leads and their respective statuses.

3.5 Expenses

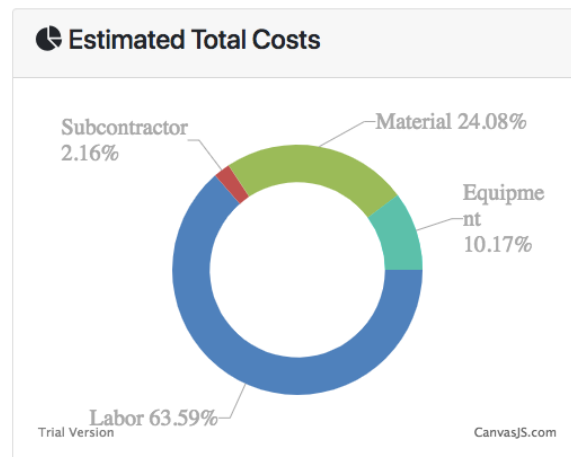
Query 31: Actual Total Costs



```
SELECT SUM(actLabor_expense) AS Labor, SUM(actSubcontractor_expense) AS  
Subcontractor, SUM(actMaterial_expense) AS Material,  
SUM(actEquipment_expense) AS Equipment  
FROM ticket
```

Justification: Show actual costs for each type of expense.

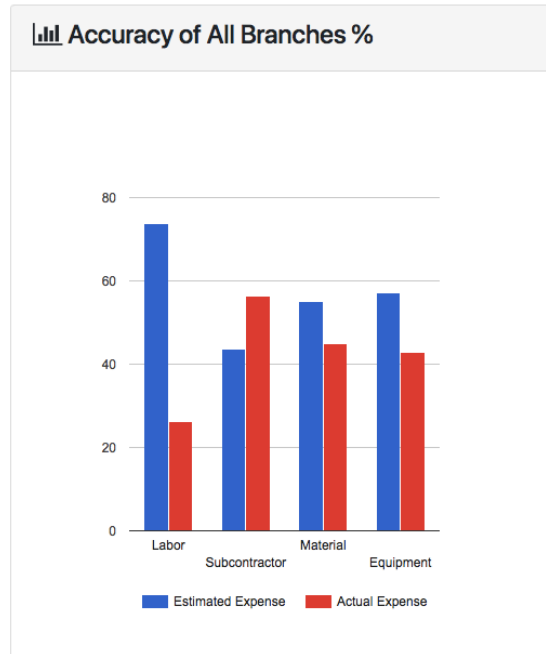
Query 32: Estimated Total Costs



```
SELECT SUM(estLabor_expense) AS Labor, SUM(estSubcontractor_expense) AS  
Subcontractor, SUM(estMaterial_expense) AS Material,  
SUM(estEquipment_expense) AS Equipment  
FROM ticket
```

Justification: Show estimated costs for each type of expense.

Query 33: Accuracy of all branches



```
SELECT SUM(actLabor_expense) AS act_labor, SUM(actSubcontractor_expense) AS  
act_subcontractor,  
SUM(actMaterial_expense) AS act_material, SUM(actEquipment_expense) AS  
act_equipment,  
SUM(estLabor_expense) AS est_labor, SUM(estSubcontractor_expense) AS  
est_subcontractor,  
SUM(estMaterial_expense) AS est_material, SUM(estEquipment_expense) AS  
est_equipment  
FROM ticket;
```

Justification: Compare estimated versus actual costs to show accuracy in the company's predictive capacity by each branch. If its close, nothing needs to change. If the difference is large, there may need to be some changes.

Query 34: Accuracy by Property %



```
SELECT c.property_id, SUM(actLabor_expense) AS act_labor,  
SUM(actSubcontractor_expense) AS act_subcontractor, SUM(actMaterial_expense)  
AS act_material, SUM(actEquipment_expense) AS act_equipment,  
SUM(estLabor_expense) AS est_labor, SUM(estSubcontractor_expense) AS  
est_subcontractor, SUM(estMaterial_expense) AS est_material,  
SUM(estEquipment_expense) AS est_equipment  
FROM ticket t  
JOIN contractservice cs ON t.contractService_ID = cs.contractService_ID  
JOIN contract c ON cs.contract_id = c.contract_id  
GROUP BY c.property_id  
ORDER BY c.property_id ASC;
```

Justification: Compare estimated versus actual costs to show accuracy in the company's predictive capacity by each property. This will show accuracy in their predictions for each property.

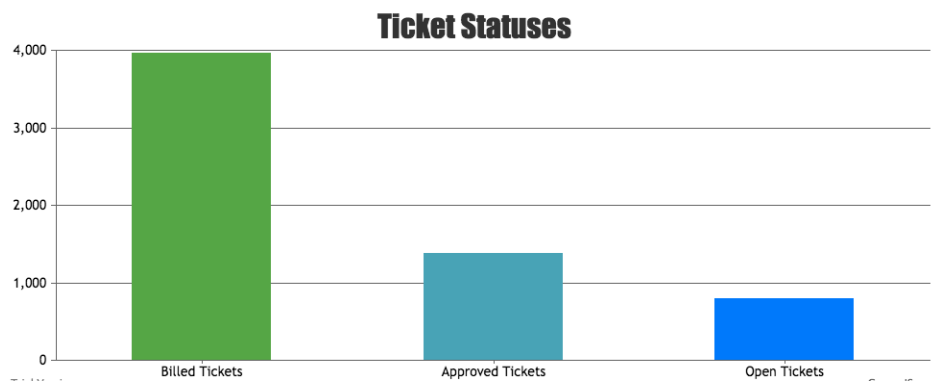
3.6 Ticket

Query 35:

```
$sql_open_ticket = "SELECT count(*) FROM Ticket WHERE ticketStatus_ID = 2;  
$sql_approved_ticket = "SELECT count(*) FROM Ticket WHERE ticketStatus_ID = 5;  
$sql_billed_ticket = "SELECT count(*) FROM Ticket WHERE ticketStatus_ID = 7;
```

Justification: Count the open, billed, and approved tickets to place in a visual to show the user where the ticket is in the pipeline.

Query 36: Bar Graph



```
SELECT ts.ticketStatus_name AS status, COUNT(*)  
FROM Ticket t  
JOIN Ticket_status ts ON t.ticketStatus_ID = ts.ticketStatus_ID  
GROUP BY (ts.ticketStatus_name);
```

Justification: Give a visual for tickets in the pipeline, only showing billed, approved, and open tickets to track tickets that will be most successful in completing.

Query 37: Billed Ticket

```
SELECT t.ticket_id, st.description, e.employee_name, cl.client_name,  
b.branch_name, co.contractYear  
FROM Ticket t  
JOIN Contractservice cs ON t.contractService_ID =  
cs.contractservice_id  
JOIN Servicetype st ON cs.servicetype_ID = st.servicetype_ID  
JOIN Employee e ON t.foreman_ID = e.employee_ID  
JOIN Contract co ON cs.contract_ID = co.contract_ID  
JOIN Property p ON co.property_ID = p.property_ID  
JOIN Client cl ON p.client_ID = cl.client_ID  
JOIN Branch b ON p.branch_ID = b.branch_ID  
JOIN Ticket_status ts ON t.ticketStatus_ID = ts.ticketStatus_ID  
WHERE ts.ticketStatus_name = 'Billed';
```

Justification: Show all information for billed tickets to review as needed.

Query 38: Approved Ticket Table

```
SELECT t.ticket_id, st.description, e.employee_name, cl.client_name,  
b.branch_name, co.contractYear  
  FROM Ticket t  
    JOIN Contractservice cs ON t.contractService_ID =  
cs.contractservice_id  
    JOIN Servicetype st ON cs.servicetype_ID = st.servicetype_ID  
    JOIN Employee e ON t.foreman_ID = e.employee_ID  
    JOIN Contract co ON cs.contract_ID = co.contract_ID  
    JOIN Property p ON co.property_ID = p.property_ID  
    JOIN Client cl ON p.client_ID = cl.client_ID  
    JOIN Branch b ON p.branch_ID = b.branch_ID  
    JOIN Ticket_status ts ON t.ticketStatus_ID = ts.ticketStatus_ID  
WHERE ts.ticketStatus_name = 'Approved';
```

Justification: Show all information for approved tickets to review as needed.

Query 39: Open Ticket

```
SELECT t.ticket_id, st.description, e.employee_name, cl.client_name,  
b.branch_name, co.contractYear  
  FROM Ticket t  
    JOIN Contractservice cs ON t.contractService_ID =  
cs.contractservice_id  
    JOIN Servicetype st ON cs.servicetype_ID = st.servicetype_ID  
    JOIN Employee e ON t.foreman_ID = e.employee_ID  
    JOIN Contract co ON cs.contract_ID = co.contract_ID  
    JOIN Property p ON co.property_ID = p.property_ID  
    JOIN Client cl ON p.client_ID = cl.client_ID  
    JOIN Branch b ON p.branch_ID = b.branch_ID  
    JOIN Ticket_status ts ON t.ticketStatus_ID = ts.ticketStatus_ID  
WHERE ts.ticketStatus_name = 'Open';
```

Justification: Show all information for open tickets to review as needed.

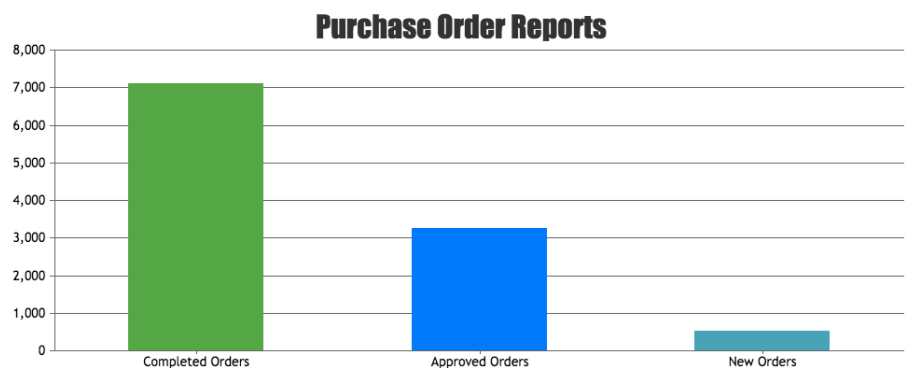
3.7 Purchase Orders

Query 40:

```
$sql_new_purchase = "SELECT count(*) FROM purchaseOrder WHERE  
purchaseStatus_ID = 1;";  
$sql_approved_purchase = "SELECT count(*) FROM purchaseOrder WHERE  
purchaseStatus_ID = 5;";  
$sql_complete_purchase = "SELECT count(*) FROM purchaseOrder WHERE  
purchaseStatus_ID = 6;";
```

Justification: Count the new, approved, and completed purchase orders to place in a visual to show the user where the ticket is in the pipeline.

Query 41: Bar graph



```
SELECT ps.status, COUNT(*)  
FROM PurchaseOrder po  
JOIN Purchase_status ps ON po.purchaseStatus_ID = ps.purchaseStatus_ID  
GROUP BY (ps.status);
```

Justification: Give a visual for the purchase orders and where they are in the pipeline.

Query 42: New order

```
SELECT po.po_id, v.vendor_name, ps.status, b.branch_name,  
po.created_by_user_code,  
po.submitted_on, po.estimated_delivery_date, po.is_fully_allocated  
FROM PurchaseOrder po  
JOIN Vendor v ON po.vendor_ID = v.vendor_ID  
JOIN Purchase_status ps ON po.purchaseStatus_ID = ps.purchaseStatus_ID  
JOIN Branch b ON po.branch_ID = b.branch_ID  
WHERE ps.status = 'New';
```

Justification: Show all information for new purchase orders to review as needed.

Query 43: Approved order

```
SELECT po.po_id, v.vendor_name, ps.status, b.branch_name,  
po.created_by_user_code,  
    po.submitted_on, po.estimated_delivery_date, po.is_fully_allocated  
FROM PurchaseOrder po  
JOIN Vendor v ON po.vendor_ID = v.vendor_ID  
JOIN Purchase_status ps ON po.purchaseStatus_ID = ps.purchaseStatus_ID  
JOIN Branch b ON po.branch_ID = b.branch_ID  
WHERE ps.status = 'Approved';
```

Justification: Show all information for approved purchase orders to review as needed.

Query 44: Complete Order

```
SELECT po.po_id, v.vendor_name, ps.status, b.branch_name,  
po.created_by_user_code,  
    po.submitted_on, po.estimated_delivery_date, po.is_fully_allocated  
FROM PurchaseOrder po  
JOIN Vendor v ON po.vendor_ID = v.vendor_ID  
JOIN Purchase_status ps ON po.purchaseStatus_ID = ps.purchaseStatus_ID  
JOIN Branch b ON po.branch_ID = b.branch_ID  
WHERE ps.status = 'Complete';
```

Justification: Show all information for completed purchase orders to review as needed.

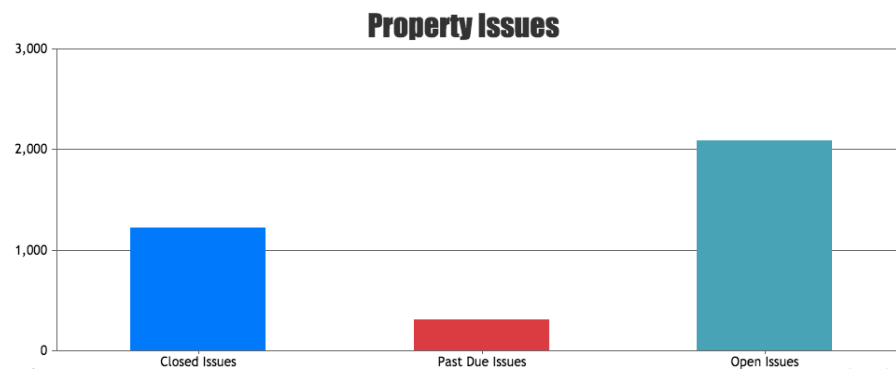
3.8 Issues

Query 45:

```
$sql_open_issue = "SELECT count(*) FROM Issues WHERE issueStatus_ID = 1;";  
$sql_closed_issue = "SELECT count(*) FROM Issues WHERE issueStatus_ID = 2;";  
$sql_past_issue = "SELECT count(*) FROM Issues WHERE issueStatus_ID = 3;";
```

Justification: Count the open, closed, and past due issues to place in a visual to show the user where the ticket is in the pipeline.

Query 46: Bar Graph



```
SELECT iss.issueStatus_name AS status, COUNT(*) AS count  
FROM Issues i  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
GROUP BY (iss.issueStatus_name);
```

Justification: Give a visual for where issues are in the pipeline.

Query 47: Closed issues

```
SELECT i.issue_id, iss.issueStatus_name, e.employee_name, p.property_name,  
it.issueType_code, py.priority, i.dueDate  
FROM Issues i  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
JOIN Employee e ON i.assignedForeman_ID = e.employee_ID  
JOIN Property p ON i.property_ID = p.property_ID  
JOIN Issue_type it ON i.issueType_ID = it.IssueType_ID  
JOIN Priority py ON i.issuePriority_ID = py.issuePriority_ID  
WHERE iss.issueStatus_name = 'Closed';
```

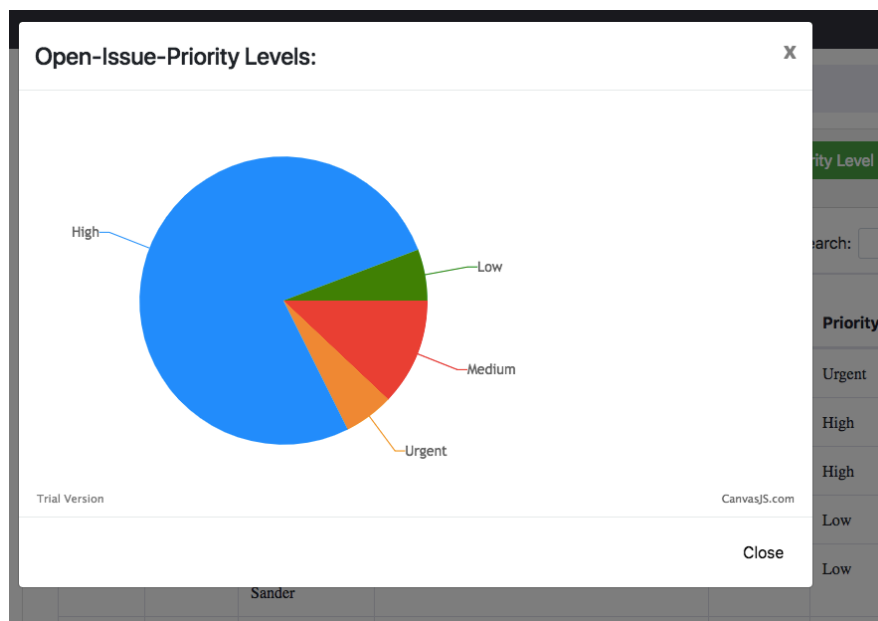
Justification: Show all information for closed issues to review as needed.

Query 48: Open issues

```
SELECT i.issue_id, iss.issueStatus_name, e.employee_name, p.property_name,  
it.issueType_code, py.priority, i.dueDate  
FROM Issues i  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
JOIN Employee e ON i.assignedForeman_ID = e.employee_ID  
JOIN Property p ON i.property_ID = p.property_ID  
JOIN Issue_type it ON i.issueType_ID = it.IssueType_ID  
JOIN Priority py ON i.issuePriority_ID = py.issuePriority_ID  
WHERE iss.issueStatus_name = 'Open';
```

Justification: Show all information for open issues to review as needed.

Query 49: Open-Issue-Priority Levels



```
SELECT p.priority, COUNT(*)  
FROM Issues i  
JOIN Priority p ON i.issuePriority_ID = p.issuePriority_ID  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
WHERE iss.issueStatus_name = 'Open'  
GROUP BY p.priority;
```

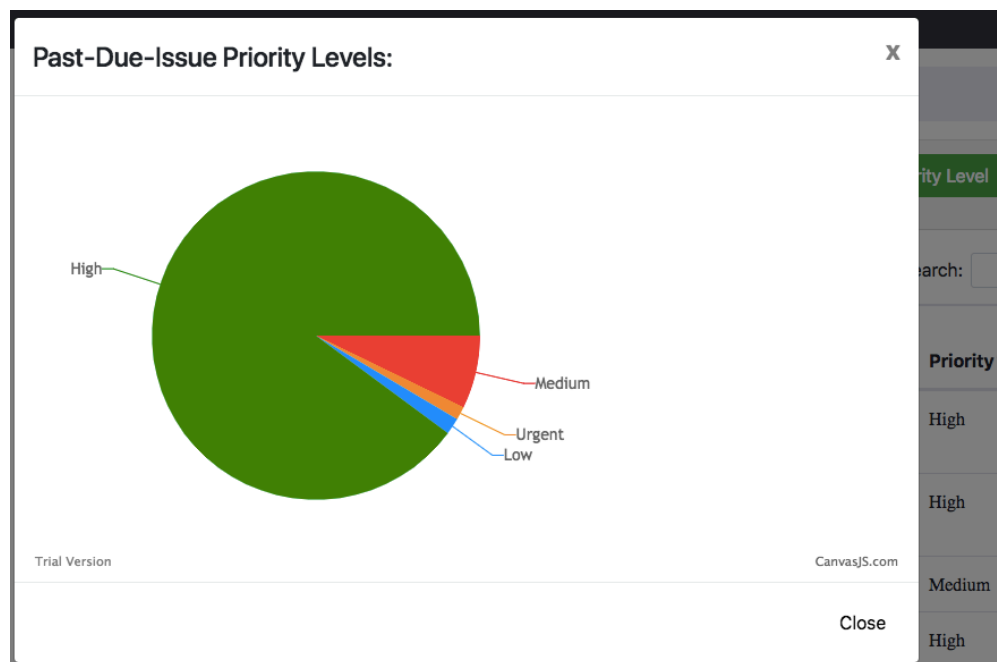
Justification: Give a visual of the amount of issues in each respective priority levels.

Query 50: past-due table

```
SELECT i.issue_id, iss.issueStatus_name, e.employee_name, p.property_name,  
it.issueType_code, py.priority, i.dueDate  
FROM Issues i  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
JOIN Employee e ON i.assignedForeman_ID = e.employee_ID  
JOIN Property p ON i.property_ID = p.property_ID  
JOIN Issue_type it ON i.issueType_ID = it.IssueType_ID  
JOIN Priority py ON i.issuePriority_ID = py.issuePriority_ID  
WHERE iss.issueStatus_name = 'Past_Due';
```

Justification: Show all information for issues that are past their due dates.

Query 51:



```
SELECT p.priority, COUNT(*)  
FROM Issues i  
JOIN Priority p ON i.issuePriority_ID = p.issuePriority_ID  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
WHERE iss.issueStatus_name = 'Past_Due'  
GROUP BY p.priority;
```

Justification: Give a visual of the amount of past due issues in each respective priority levels.

3.9 Employees

Query 52: Find a manager name of each branch

```
SELECT B.branch_id, B.branch_name, E.employee_name
FROM Branch B, Employee E
WHERE E.employee_ID = B.branchmanager_id
ORDER BY B.branch_id;
```

Justification: Locate branch managers in each branch, useful for companies with multiple branches.

Query 53: List all the employee

```
SELECT E.employee_id, E.employee_name, B.branch_name
FROM Employee E, Branch B
WHERE E.branch_id = B.branch_id
ORDER BY E.employee_id;
```

Justification: Show all employees and their respective branches.

3.10 Clients

Query 54: Client information (Admin and Executive Manager View)

```
SELECT P.property_id, P.property_name, B.branch_name, C.client_name,
I.industry_name, A.addresstype,
      A.addressline, A.city, A.state, A.zipcode
FROM Property P, Branch B, Client C, Industry I, Address A
WHERE P.branch_id = B.branch_id AND P.client_id =
C.client_id AND P.industry_id = I.industry_id AND P.address_id
= A.address_id
ORDER BY P.property_id;
```

Justification: Show client information for admins and executives for review.

Query 55: Client summary view for Account Manager

```
SELECT * FROM client_summary;
```

Justification: Once log in to the web system as an account manager user, the user can only see the certain information except, e.g., client's address, client's ID, etc.

4 Analytics:

Company Performance

4.1 Find the current company profit margin

The following query will return:

- I. the current total net income (total revenue – total cost)
- II. the current total revenue
- III. the result of division which yield the current company profit margin.

```
SELECT (SUM(sale_price) - SUM(actualCost)), SUM(sale_price),  
       (SUM(sale_price) - SUM(actualCost)) /  
SUM(sale_price)  
FROM Contract;
```

4.2 Retrieve the total revenue and expense on each year

The following query will only return the sum of revenue and expense of company on each year since the first contract had signed.

```
SELECT contractYear, SUM(sale_price), SUM(actualCost)  
FROM Contract  
GROUP BY contractYear  
ORDER BY ContractYear ASC;
```

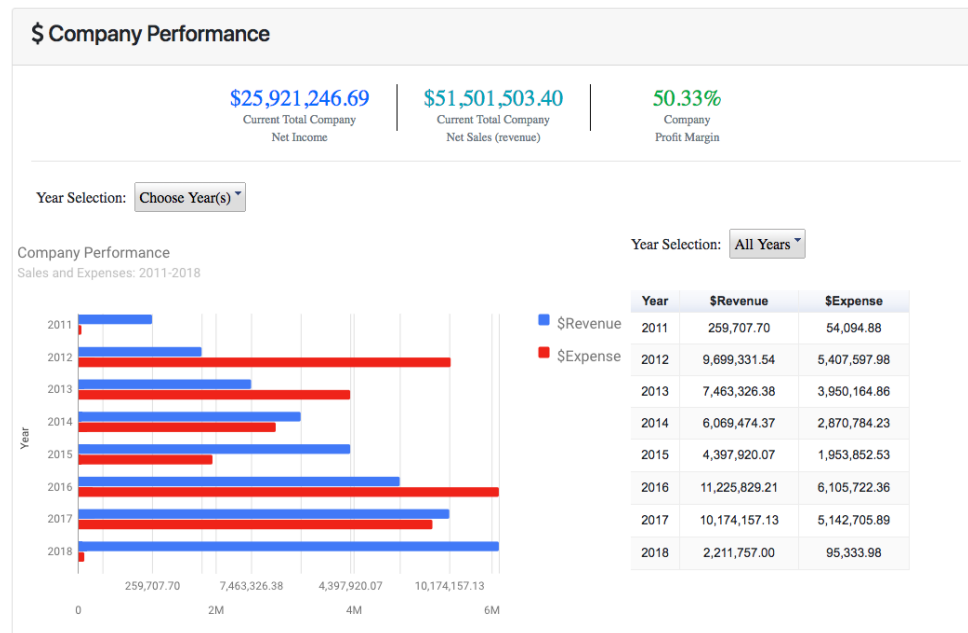


Figure 4.1: illustrates a display numbers of company performance, and the bar chart including the table comparing the total revenue and expense since 2011 – 2018.

4.3 Contribution margin over years

The following query will only return years and contribution margin on each year which is calculated by $(1 - (\text{total cost} / \text{total sale})) * 100$.

```
SELECT contractyear, (1 - (sum(actualCost)/sum(sale_price))) * 100 AS  
Contribution_Margin  
FROM Contract  
GROUP BY contractyear  
ORDER BY contractyear;
```

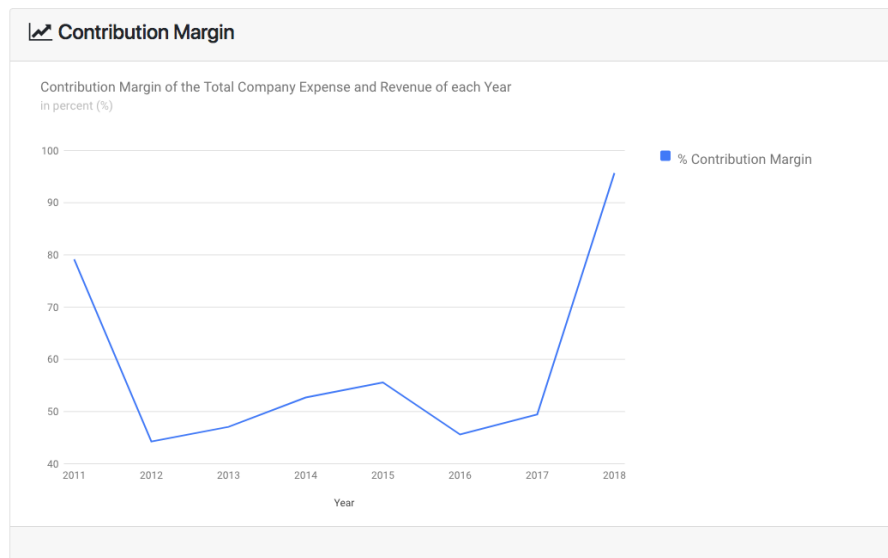


Figure 4.2: illustrates the line chart of contribution margin between 2011 – 2018.

4.4 Accuracy of all branches

The following query will return the company's estimated versus actual expense over labor, subcontractor, material, and equipment.

```
SELECT SUM(actLabor_expense) AS act_labor, SUM(actSubcontractor_expense) AS  
act_subcontractor,  
SUM(actMaterial_expense) AS act_material, SUM(actEquipment_expense) AS  
act_equipment,  
SUM(estLabor_expense) AS est_labor, SUM(estSubcontractor_expense) AS  
est_subcontractor,  
SUM(estMaterial_expense) AS est_material, SUM(estEquipment_expense) AS  
est_equipment  
FROM ticket;
```

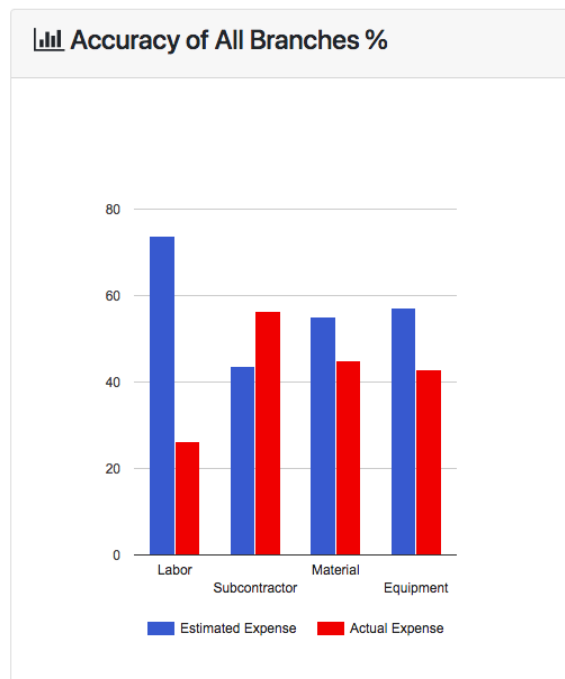


Figure 4.3: illustrates the bar chart comparing estimated versus actual costs to show accuracy in the company's predictive capacity by each branch.

4.5 Open-Issue-Priority Levels

The following query will only return number of issues in each priority where the issue status is 'open.'

```
SELECT p.priority, COUNT(*)  
FROM Issues i  
JOIN Priority p ON i.issuePriority_ID = p.issuePriority_ID  
JOIN Issue_status iss ON i.issueStatus_ID = iss.issueStatus_ID  
WHERE iss.issueStatus_name = 'Open'  
GROUP BY p.priority;
```

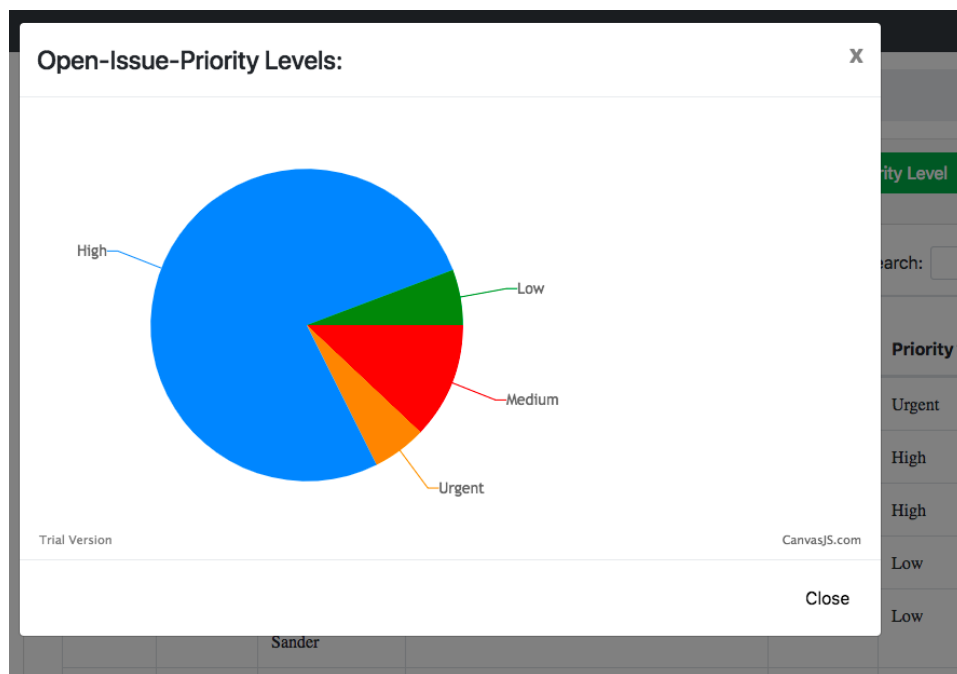


Figure 4.4: illustrates the pie chart comparing the numbers of issue in different priority level, e.g., urgent, high, medium, low.

5 Normalization:

5.1 Normalization

During our designing process of the database, we found many information to be redundant and waste of memory resources. For example, the original table we designed was to store `user_type_name` attribute in the User table (see figure 5.1). The Keys of this relation are `U_ID` and `U_username`. To add a new functional dependency and decompose this relation into BCNF it would help us eliminate redundant storage, which also help preventing insert, delete, and update anomalies.

Doing so it will allows to easily remove, add, or update the type user without messing with the Users table. Thus, we came up with new attribute called `user_type_id` in the users table and created a new table called `user_type`. The `user_type_ID` and `user_type_name` now introduced a new functional dependency (FD: $UT_id \rightarrow UT_name$).

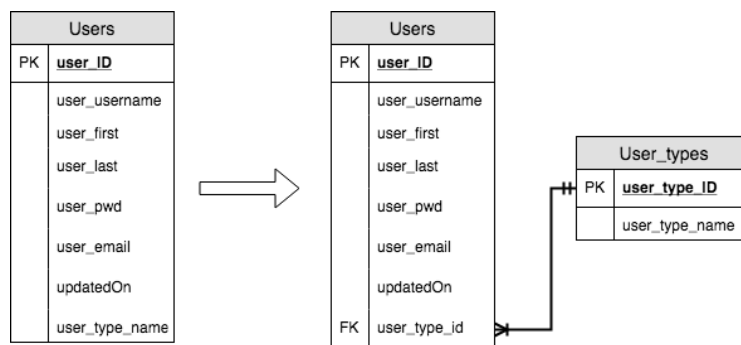


Figure 5.1

Furthermore, we have done the same process of normalization repetitively on the following tables:

- I. issue - priority
- II. issue - issue_status
- III. issue - issue_type
- IV. contract - contract_status
- V. purchaseOrder - purchase_status
- VI. ticket - ticket_status
- VII. lead - lead_type
- VIII. lead - lead_status

6 Indexing Selection:

The creation of index is important for our project because we are dealing with a lot numbers and many large amounts of record in many tables. However, we decided to use the front-end technique in sorting and scoping data. Thus, to show the potential of implementation of the index on our project we have created the new query and display the result shown in figure 6.1. The query is to list the contracts that have a revenue/price more than \$10,000 and then calculated the profit percentage. We simulated the indexing to search a ranged selection of our contract prices to pull the expensive contracts. We created the index on sale price column in Contract table by using a B-Tree method. This allows the searching time to be faster. As shown in figures 6.2 and 6.3, the execution time has dropped almost 100% after creating the index.

```
SELECT Contract_ID, contractYear, sale_price AS Revenue, actualCost AS
Cost,
        round((( (sale_price - actualCost)/(actualCost+1)) *100), 2) AS
Profit_percentage
FROM Contract
Where sale_price > 10000
Order BY Contract_ID ASC;
```

887 rows affected.

contract_id	contractYear	revenue	cost	profit_percentage
134	2011	13050.00	1156.24	1027.77
164	2011	32440.00	2125.11	1425.84
196	2011	12041.00	2000.00	501.80
233	2012	13680.00	4534.12	201.67
258	2012	18648.69	5611.52	232.29
490	2012	12440.00	4821.36	157.99
562	2012	14305.96	3857.36	270.80
578	2012	10015.00	4961.20	101.85
617	2012	43140.00	28238.73	52.77
767	2012	35880.00	18944.61	89.39
794	2012	25015.00	10279.70	143.33

Figure 6.1

Before creating index:

```
explain analyze
SELECT Contract_ID, contractYear, sale_price AS Revenue, actualCost AS
Cost,
        round((( (sale_price - actualCost)/(actualCost+1)) *100), 2) AS
Profit_percentage
FROM Contract
Where sale_price > 10000
Order BY Contract_ID ASC;
```

QUERY PLAN
Sort (cost=551.96..554.23 rows=906 width=19) (actual time=5.624..5.993 rows=887 loops=1)
Sort Key: contract_id
Sort Method: quicksort Memory: 94kB
-> Seq Scan on contract (cost=0.00..507.46 rows=906 width=19) (actual time=0.196..5.079 rows=887 loops=1)
Filter: (sale_price > '10000'::numeric)
Rows Removed by Filter: 16244
Planning time: 0.121 ms
Execution time: 6.369 ms

Figure 6.2

Create index on Contract Table (sale price):

```
CREATE INDEX contract_index on contract USING Btree(sale_price);
```

```

--sql
explain analyze
SELECT Contract_ID, contractYear, sale_price AS Revenue, actualCost AS Cost,
       round((( sale_price - actualCost)/(actualCost+1)) *100), 2) AS Profit_percentage
FROM Contract
Where sale_price > 10000
Order BY Contract_ID ASC;

```

10 rows affected.

QUERY PLAN
Sort (cost=368.46..370.72 rows=906 width=19) (actual time=2.628..2.969 rows=887 loops=1)
Sort Key: contract_id
Sort Method: quicksort Memory: 94kB
-> Bitmap Heap Scan on contract (cost=19.31..323.96 rows=906 width=19) (actual time=0.208..2.047 rows=887 loops=1)
Recheck Cond: (sale_price > '10000'::numeric)
Heap Blocks: exact=161
-> Bitmap Index Scan on contract_index (cost=0.00..19.08 rows=906 width=0) (actual time=0.173..0.173 rows=887 loops=1)
Index Cond: (sale_price > '10000'::numeric)
Planning time: 0.516 ms
Execution time: 3.350 ms

Figure 6.3

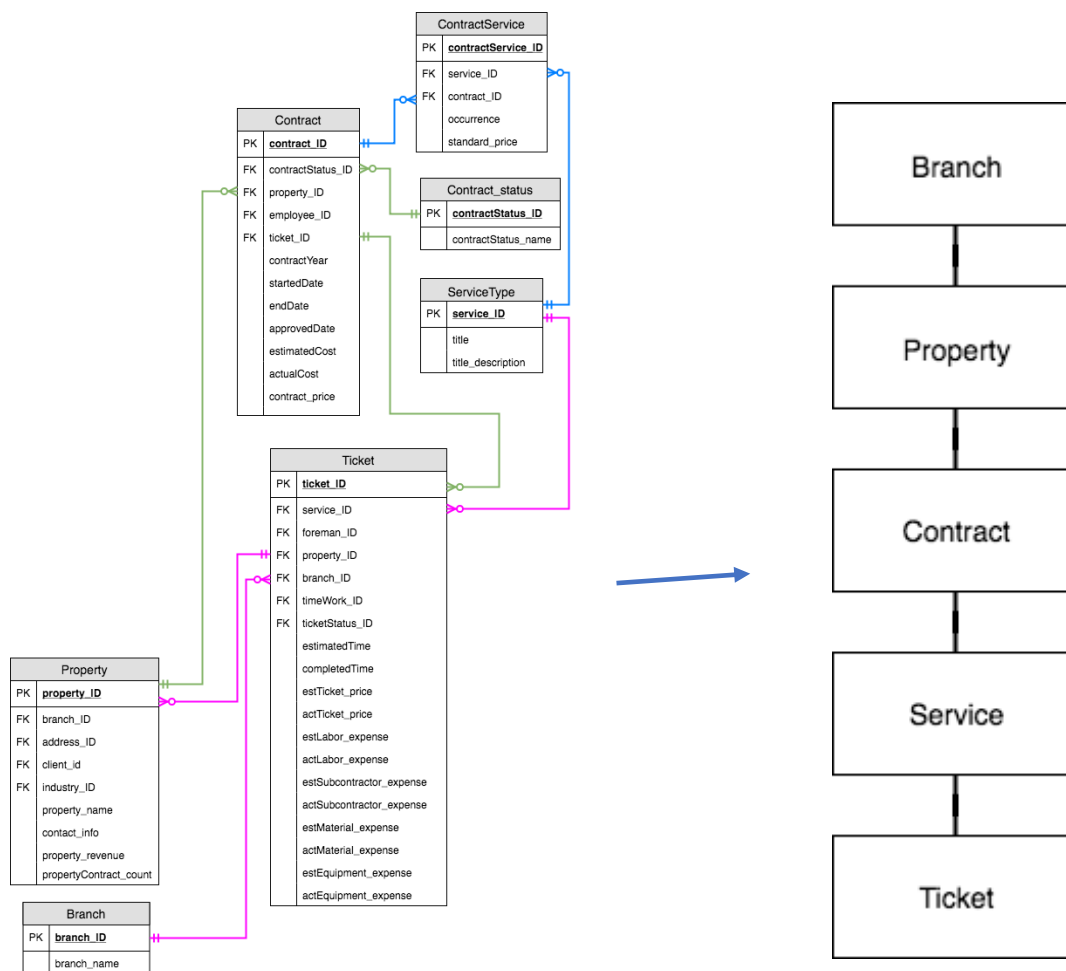
The example above also shows that the “sorted” file organizations is the most efficient for the query above.

7 Optimization and Tuning:

Optimization and Tuning: Discuss the topics covered in class, specifically how they impacted the design of your final project.

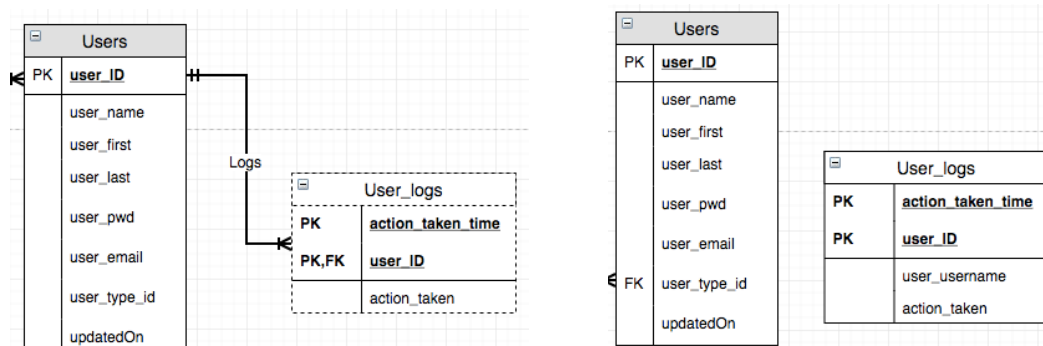
7.1 Tables refinement

We changed our original ERD diagram from being very clustered into a more organized hierarchical structure to simplify our querying paths. Before the change, each and every table was interconnected. After the change, we have a straight path to each tuple of data from any given table. This allows our queries to be more organized and efficient.



7.2 Constraint conflict/error

In the beginning, we created a composite key of `user_ID` and `action_taken_time` to identify each activity log data in the `user_logs` table. We experienced the error when an admin was trying to delete the user because of FOREIGN KEY constraint in user logs table. Since we desire to keep `user_username` data of a particular user even he/she is removed from the `users` table as well, we decided to create an independent table that contains the attributes we would like to keep. Also, we do not want to use Reference Constraint in the `user_ID` from `users` table with ON DELETE CASCADE because that way we will lose all the activity logs from that user who is being deleted from the system entirely.



8 Security Setting:

8.1 Mandatory Access Control in the front-end

In this project, most the security setting has been done through PHP API where we wrote the query to check the existence of the user and their password before allowing them to access to our web application. The admin is a superuser who, theoretically, can grant or revoke the privileges of all users.

8.2 Limited view of client information for particular type of user

Although, in this project we did not perform any database security and authorization, we had implemented the View of client's summary to show the limited access of a certain type of user, i.e., account manager. Below is the implementation of our view on client table where limited the account manager to see only what he/she should see.


```
DROP VIEW IF EXISTS s18group01.client_summary;

CREATE VIEW s18group01.client_summary AS
SELECT P.property_name, B.branch_name, C.client_name, I.industry_name,
A.addresstype, A.city, A.state
      FROM Property P, Branch B, Client C, Industry I, Address A
      WHERE P.branch_id = B.branch_id AND P.client_id = C.client_id AND
P.industry_id = I.industry_id AND P.address_id
      = A.address_id
      ORDER BY P.property_id;
```

- Account Manager View Result:

Client Information						
Search Property: All						
Search Client: Search						
Branch Selection: All Industry Selection: All						
Property Name	Branch	Client Name	Industry	Address Type	City	State
Crandall Residence	St. Louis	John Smith	Residential	Work	Chesterfield	MO
Harfinger Residence	St. Louis	John Smith	Residential	Work	Chesterfield	MO
Balian Residence	St. Louis	Alex Smith	Residential	Work	Lake in the Hills	IL
Nyenhuis Residence - OLD	St. Louis	Jack Black	Residential	Physical	Algonquin	IL
Patel, Shashikant Residence	St. Louis	John Smith	Residential	Mailing	Algonquin	IL
Wokoun Residence	St. Louis	Jack Black	Residential	Physical	Spring Grove	IL
Wong Residence	St. Louis	John Smith	Residential	Mailing	Spring Grove	IL
George Residence	St. Louis	John Smith	Residential	Physical	Crystal Lake	IL
Ramanna Residence	St. Louis	Alex Smith	Residential	Mailing	Crystal Lake	IL
Yang Residence	St. Louis	John Smith	Residential	Physical	Lake Geneva	WI
Casaccio Residence	St. Louis	Tom Cruise	Residential	Mailing	Huntley	IL
More Residence	St. Louis	John Smith	Residential	Physical	Glenview	IL
Puppala Residence	St. Louis	John Smith	Residential	Mailing	Glenview	IL
Behal Residence	St. Louis	Tom Cruise	Residential	Physical	North Barrington	IL

- Administrator and Executive Manager Display:

 **Client Information**

Search Property:

Search Client:

Branch Selection: Industry Selection:

Property Name	Branch	Client Name	Industry	Address Type	Address	City	State	zipcode
Crandall Residence	St. Louis	John Smith	Residential	Work	1234 Main St.	Chesterfield	MO	63005
Harfinger Residence	St. Louis	John Smith	Residential	Work	1234 Main St.	Chesterfield	MO	63005
Balian Residence	St. Louis	Alex Smith	Residential	Work	8595 Pyott Road, Suite C	Lake in the Hills	IL	60156

9 Other Topics:

9.1 Trigger

This below trigger was created to maintain the database without inserting in multiple files. This trigger will allow the user to input a ticket tuple and the database will automatically update the occurrence in ContractService table to account for the ticket. With this updating table, more analytics could be run by seeing if the contract has been fulfilled yet, etc.

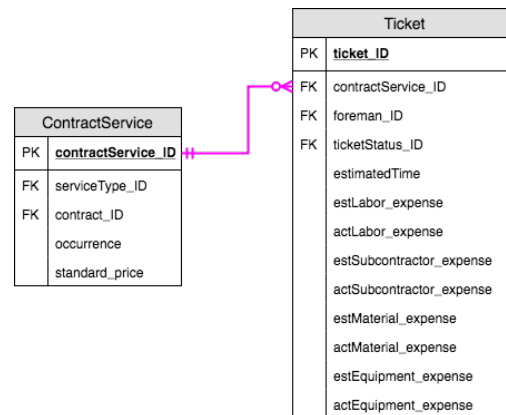


Figure 9.1

* Function to update the occurrence by incrementing the previous value by one when the function is being called.

```
DROP TRIGGER IF EXISTS update_ticket_occurrence ON s18group01.Ticket;
DROP FUNCTION IF EXISTS increment_occurrence_contractService();

CREATE FUNCTION increment_occurrence_contractService()
    RETURNS trigger AS
$$ BEGIN
    UPDATE s18group01.ContractService
    SET occurrence = occurrence+1
    WHERE NEW.contractservice_id = contractservice_id;
    RETURN NEW;
END;
$$
LANGUAGE plpgsql;
```

*After inserting a new row into ticket table, trigger will be fired the update_ticket_occurrence function.

```
CREATE TRIGGER update_ticket_occurrence
    AFTER INSERT ON Ticket
    FOR EACH ROW
    EXECUTE PROCEDURE increment_occurrence_contractService();
```


10 User's Manual:

For the user's manual, please the see the attachment 'User Manuel Executive Board.pdf'