A Song of Ice and Fire: Back-Office Retail System



**DESIGNED BY: GREGORY ABBENE**

**Professor Labouseur**

**CMPT-308- Database Systems (Design Project)**

Table of Contents

**Executive Summary4**

**Entity-Relationship Diagram5-8**

People and Addresses ERD5

Vendor with Item ERD6

Item with Price, Cost, and Sales ERD7

Full ERD8

**Tables9-22**

Item Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)9

Categories Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)10

Departments Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)10

People Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)11

PeoplePhone Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)11

Vendors Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)12

VendorPhone Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)12

Customers Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)13

Employees Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)14

Addresses Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)15

PeopleAddresses Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)15

VendorAddresses Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)15

Regions Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)16

ZipCode Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)16

CategoryItems Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)17

DepartmentCategories Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)17

DepartmentItems Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)18

VendorItems Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)18

ItemsPurchased Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)19

ItemPrice Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)20

VendorCost Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)20

DailySales Table (Brief Description, Create Statements, Functional Dependencies, and Sample Data)21-22

**View 23-25**

Vendor Contact View (Brief Description, Create Statements, and Sample Data)23-24

Customer Contact View (Brief Description, Create Statements, and Sample Data)23-24

Daily Accounting View (Brief Description, Create Statements, and Sample Data)25

Marketing Areas View (Brief Description, Create Statements, and Sample Data)25

**Reports26-29**

Category Sales History Report (Brief Description, Create Statements, and Sample Data)26

Department Sales History Report (Brief Description, Create Statements, and Sample Data)26

Item Margin Report (Brief Description, Create Statements, and Sample Data)27

Daily Sales UPC Report (Brief Description, Create Statements, and Sample Data)27

Item Lookup Report (Brief Description, Create Statements, and Sample Data)28

Zero Sales UPC Report (Brief Description, Create Statements, and Sample Data)29

**Stored Procedures & Triggers30-33**

Customer Points Check Stored Procedure (Definition, Create/Query, and Sample Output!)30-31

Customer Points Fix Stored Procedure (Definition, Create/Query, and Sample Output!)32

Customer Points Fix with Triggers33

**Security34**

**Implementation Notes35**

**Know Problems35**

**Future Enhancements35**

# EXECUTIVE SUMMARY:

***I would like to ask for this information not to be disclosed, as some of the information is based off some proprietary knowledge and architecture from my previous job—I received permission to loosely use their architecture, but under the condition that it only sees your eyes (Professor Labouseur). Thanks, and sorry for not bring this up earlier.***

This database design is meant fit the scope and basic objectives for a back-office retail system (mainly focused on super-markets) that has the twist of using data referencing the great book series by George R.R Martin, *A Song of Ice and Fire*. The primary objective of this database design is to implement some of the modern-day supermarket system best-practices into a relational database model, as the general scope is far too complex and robust for this assignment. This system will attempt to meet the needs of the back-end users to access product/item data with respect to its various functional uses. It will also look into customers and employee data. This data will provide information for back-end’s to optimize their retail system strategy.

The following project uses an Entity-Relationship Diagram(s) to layout the design/architecture of the database. In order to describe the entities of the database, table definitions and descriptions are provided—along with their “create” statements, their functional dependencies, and sample data outputs. In addition, this project includes some sample views and reports that could be helpful for the use of the database. Following these resources is the inclusion of a few store procedures and triggers that may be helpful in the effective use of the database. Then the project will provide some security features with the inclusion of future notes—denoting implementation notes, some know problems, and potential future enhancements.

# People and Addresses Entity-Relationship Diagram:

# 

# Vendor with Item Entity-Relationship Diagram: [*Note: Removed LastUpdated field in VendorCost]*

# Item with Price, Cost, and Sales Entity-Relationship Diagram:

# Full Entity-Relationship Diagram: [*Note: Removed ‘LastUpdated’ fields]*

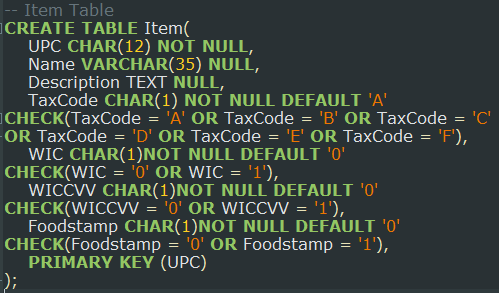
# Tables:

## Item Table:

The *Item* table is one of the core elements of this database as it stores all data about the products/items that are part of the retailer’s product portfolio. The Universal Pricing Code, UPC, is the unique identifier as it is typically the code used in modern day retail systems in the United States, Canada, the United Kingdom, Australia, New Zealand and in other countries for tracking trade items in stores. The “TaxCode” field represents the fact that some items and regions have different tax applications. The “WIC,” “WICCVV,” and “Foodstamp” fields are all item specific fields that are necessary to apply based on government policy for applications towards social development programs.

**Functional Dependency:** UPC 🡪 Name, Description, TaxCode, WIC, WICCVV, Foodstamp

**Create Statement: Sample Data Output:**



## Categories and Department Table:

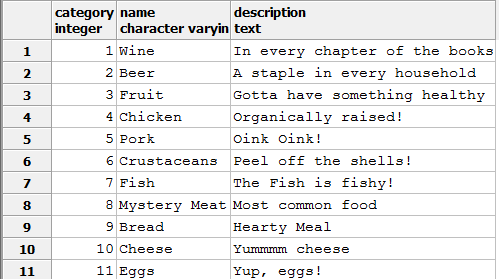
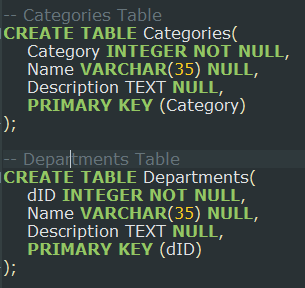
The *Categories* table is meant to store data for grouping together products/items based on similar characteristics, and it used to determine performance of certain item types compared to others. The use of category code (category) as the unique identifier for category application is meant to reflect retail best-practices. Typically, this is represented a “char” field because a usual category name would be say C0002, but to fit other primary keys methodologies it is an integer.

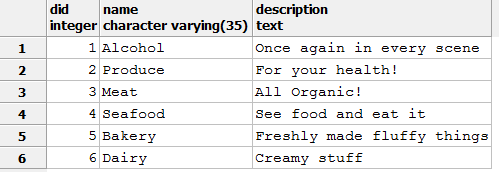
The *Departments* table is an additional grouping method of placing certain categories into a broader group for storing and relative performance means—departments (dID).

**Functional Dependencies:** Category 🡪 Name, Description

dID 🡪 Name, Description

**Create Statement(s): Sample Data Output(s):**





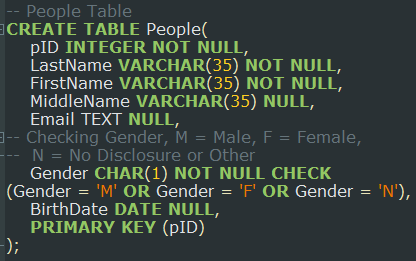
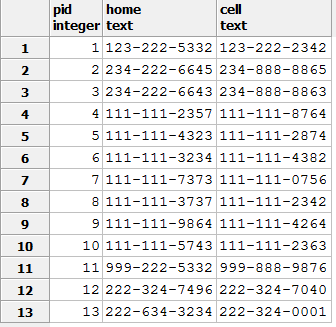
## People and People Phone Table(s):

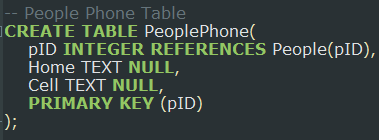
The *People* table stores all data from the people that are involved with the retail system (market), such as employees or customers.

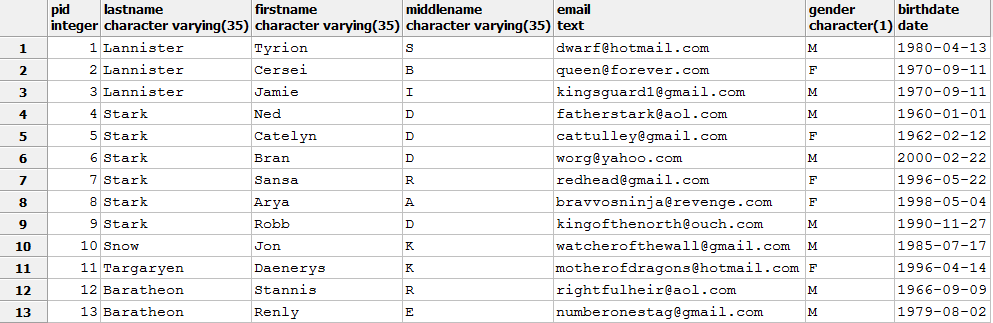
The *People Phone* table stores the home and cell phone numbers for each person in as a separate table.

**Functional Dependency:** pID 🡪 LastName, FirstName, MiddleName, Email, Gender, Birthdate

pID 🡪 Home, Cell

**Create Statement: Sample Data Output:**





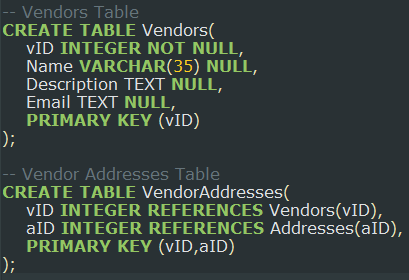
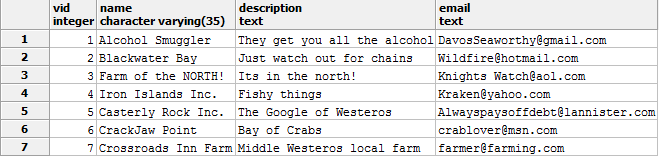
## Vendors and Vendor Phone Table(s):

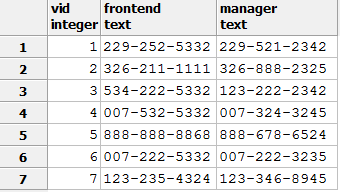
The *Vendors* table stores all data for Vendors who supply the products/items to the retail system (market). Vendors set costs for the items, which are incurred by the market to purchase the products with the goal to resell the given products at a premium of the cost. As of now, each item requires an external vendor, as this relatively small market does not produce any of their own products. Also, please note that an “email” field has been appended to the table that is not represented in the ER Diagram.

The *Vendor Phone* table stores the frontend (overall employee contact number) and manager’s phone number for each vendor.

**Functional Dependencies:** vID 🡪 Name, Description,Email

vID 🡪 Frontend, Manager

**Create Statement(s): Sample Data Output(s):**

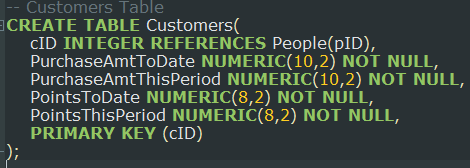


## Customers Table:

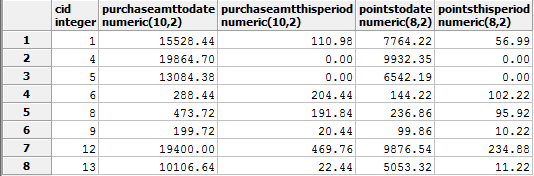
The *Customers* table is a subset of the *People* table that stores all the central data for customers of the market. As of now, it is assumed that all of the people who shop at the store are automatically “rewards” members that are tracked by the database—the problem will be mended in future iterations of the database. This table includes a unique identifier as well as “rewards” data for history and promotional perks. The rewards data will most likely need to be populated by an external application that follow specific business rules and logic. As of now, the points system is just set as have of total paid, although this is rather simplistic to the actual calculation of a rewards system.

**Functional Dependency:** cID (pID) 🡪 PurchaseAmtToDate, PurchaseAmtThisPeriod, PointsToDate, PointsThisPeriod

**Create Statement:**



**Sample Data Output:**

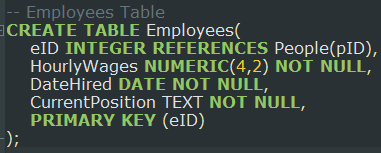


## Employees Table:

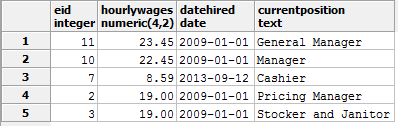
The *Employees* table is a subset of the *People* table that stores all the central data for the employees. This also assumes that all of the employees are still currently working there, and that there are no other prior employees—who would be stored in say an “employee achieve” table. This table stores a unique identifier, wage data, and hire data, as well as a “current position” field which in the future should be referenced to an additional table as say *Roles* or *Jobs* to store all possible positions.

**Functional Dependency:** eID (pID) 🡪 HourlyWage, DateHired, CurrentPosition

**Create Statement:**



**Sample Data Output:**



## Addresses, PeopleAddresses, and VendorAddresses Table(s):

The *Addresses* table stores all the central data for applying addresses to a certain table—people or vendors. This address table uses two fields, “ZipCode” and “RegionID” that are referenced to additional tables to aid in creating BCNF characteristics and limit address data inputs.

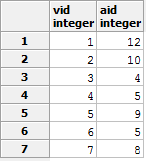
The *PeopleAddresses* table applies a certain address identification to a person by cross referencing “pID” and “aID”

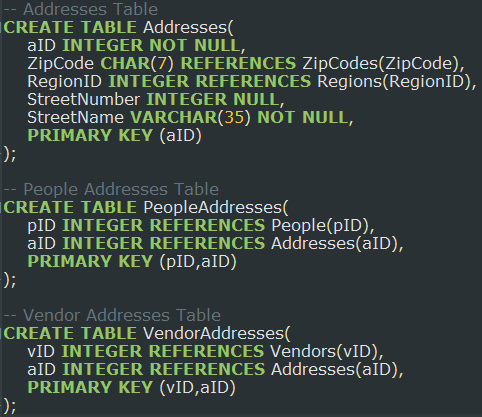
The *VendorAddresses* table applies a certain address identification to a vendor by cross referencing “vID” and “aID”

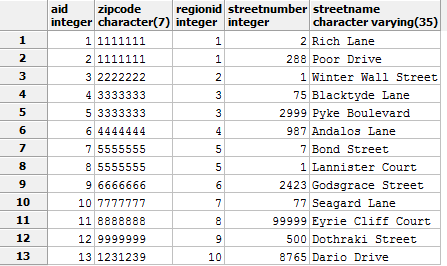
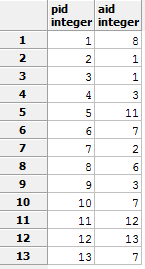
**Functional Dependencies:** aID 🡪 ZipCode, RegionID, StreetNumber, SteetName

pID,aID 🡪

vID,aID 🡪

**Create Statement(s): Sample Data Output(s):**





## Regions, and ZipCodes Table(s):

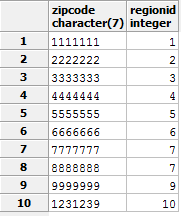
The *Regions* table stores data for applying regions to a certain zip code for an address. This table is replaceing a “State” table given the scope of the data. Also, it is assumed that the limited amount of regions are all of the possible regions that could be addressed in a given aID row.

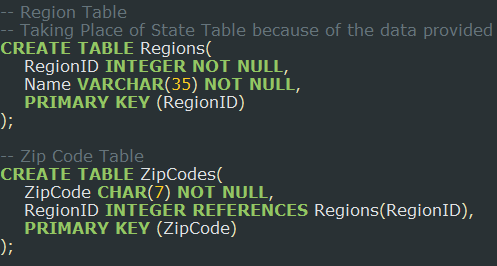
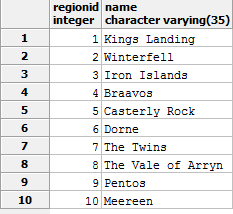
The *ZipCodes* table stores all of the possible zip codes for all of the regions, and it also follows the same assumption as the *Regions* table in which this table includes all possible zip codes to be correctly referenced in an aID row.

The *VendorAddresses* table applies a certain address identification to a vendor by cross referencing “vID” and “aID”

**Functional Dependencies:** RegionID 🡪 Name

ZipCode 🡪 RegionID

**Create Statement(s): Sample Data Output(s):**



## CategoryItems, and DepartmentCategories Table(s):

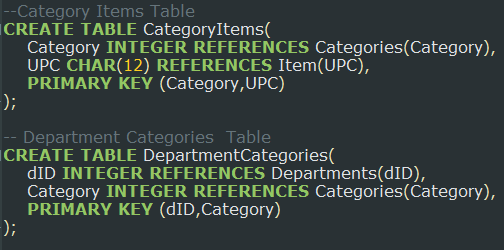
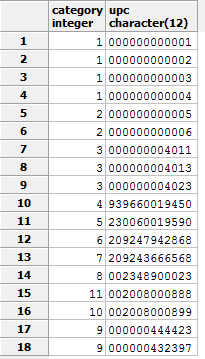
The *CategoryItems* table links together which items/products belong to which category. There is a cross reference between “Category” and “UPC” to determine which category data is applied to a particular item.

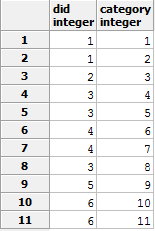
The *DepartmentCategories* table links together which categories are stored into what departments. There is a cross reference between “Department” and “Category” to determine which category data is applied to the particular department. It provides data to come to the informative conclusion of which categories belong to a particular department—and vice versa

**Functional Dependencies:** Category,UPC 🡪

dID,Category 🡪

**Create Statement(s): Sample Data Output(s):**





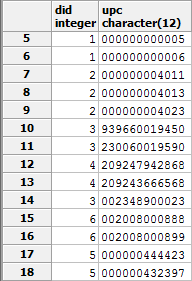
## DepartmentItems, and VendorItems Table(s):

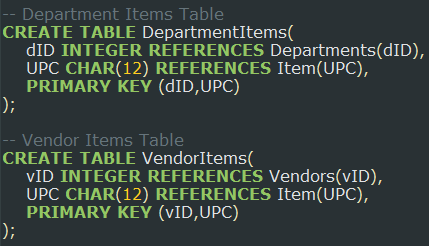
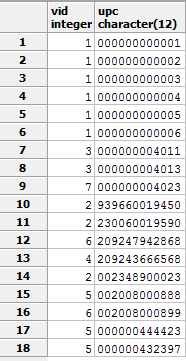
The *DepartmentItems* table links together which items/products belong to a particular department. There is a cross reference between “dID” and “UPC” to determine which department data is applied to a particular item.

The *VendorItems* table links together which vendor supplies a particular item/product. There is a cross reference between “vID” and “UPC” to determine which vendor data is applied to the particular item. This table logic assumes that each item requires a vendor (as an “in-house vID” is not setup in the particular system). Also, in this given system each item only has one vendor, which may not be the case and it is not limited by the architecture to do so.

**Functional Dependencies:** dID,UPC 🡪

vID,UPC 🡪

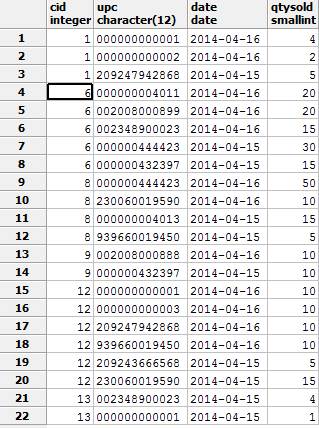
**Create Statement(s): Sample Data Output(s):**

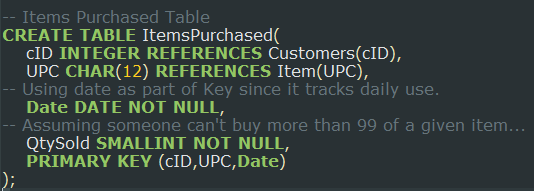


## ItemsPurchased Table:

The *ItemsPurchased* table links together the *Customers* entity to the *Items* entity. Presumably, an *Orders* or *Transactions* table should be created to link these tables; however by using a date in the primary key, it enables identification to particular orders. This could be more precise using a timestamp to assure that multiple transactions could be made by a customer for a particular item, or a given stored procedure or trigger could be used update the primary key row to update the “QtySold” field.

**Functional Dependency:** cID,UPC,Date 🡪 QtySold

**Create Statement: Sample Data Output:**



## ItemPrice and VendorCost Table:

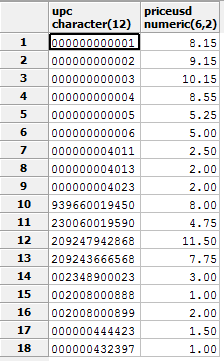
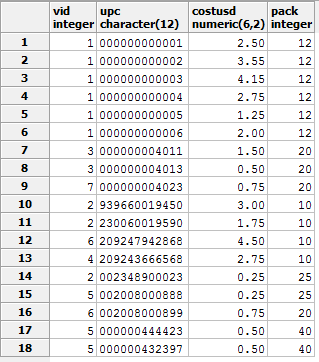
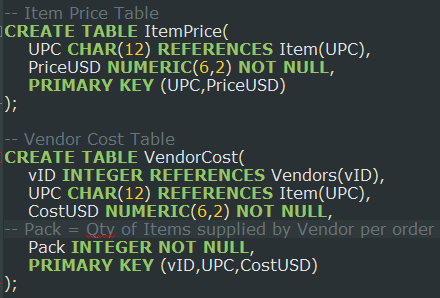
The *ItemPrice* table associates a price, in U.S dollars, that customers pay for each item. It is in USD because it is easier to conceptualize relative product prices that way.

The *VendorCost*  table associates a cost, in U.S dollars, that company must pay the vendors for each item. The “costUSD” is associated at a per item cost, and the given order is applied as multiple items per purchase, in the form of “pack.” The table gives the “vID,” “UPC,” “costUSD,” and “pack” for the given supply.

**Functional Dependencies:** UPC,PriceUSD 🡪

vID,UPC,CostUSD 🡪Pack

**Create Statement: Sample Data Output:**

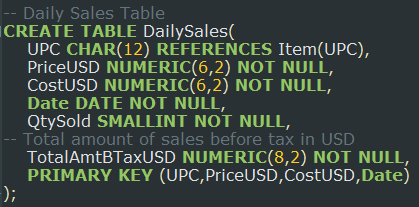


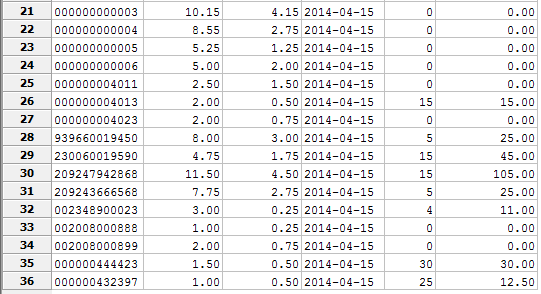
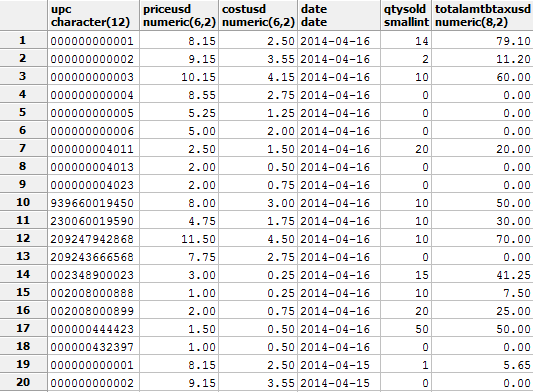
## DailySales Table:

The *DailySales* table takes an item/product and stores its “PriceUSD” and “CostUSD” on a given date. This will also provide the total quantity of the sales for the given item on a day. With this data, the specific total amount that the item generates, before tax, will be part of the table. As of now, this is all hardcoded and there is no after-tax figure. This could potentially be calculated in a *Profit* table for each item for each day. The goal of this table is to provide sales data, for further information to determine the given pricing strategy and product portfolio for the retailer.

**Functional Dependency:** UPC, PriceUSD, CostUSD, Date 🡪 QtySold, TotalAmtBTaxUSD

**Create Statement:**





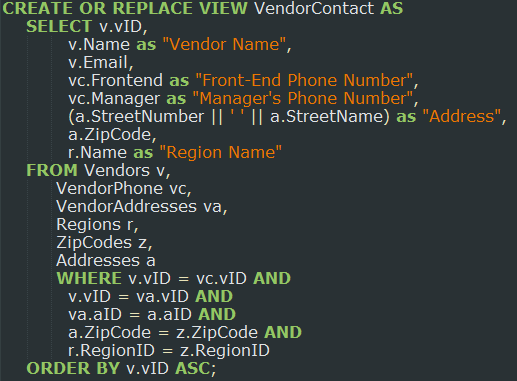
**Sample Data Output:**

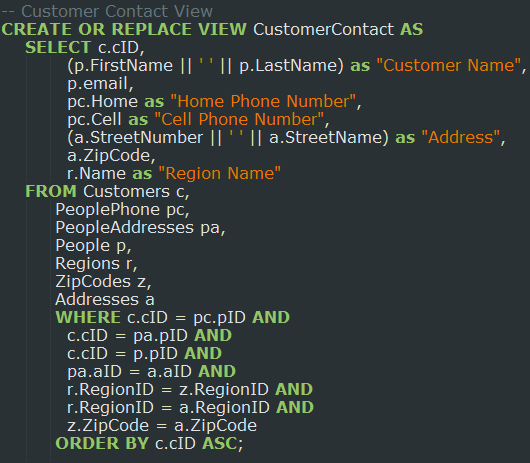
# Views:

## VendorContact and CustomerContact View(s):

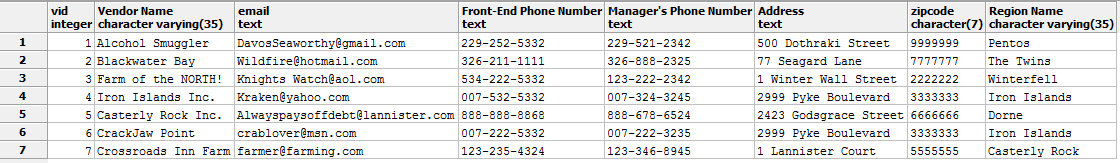
The *VendorContact* view gives the managers the ability to bring up all of the necessary contact information for all of the stored vendors. This will enable a user-friendly concatenation of several tables into a single view for managers to possibly order additional inventories, or just contact the vendors for any other reason.

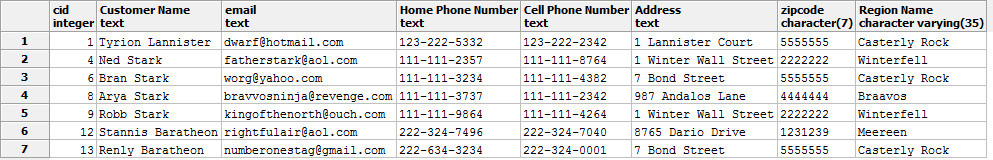
The *CustomerContact* view does essentially the same thing as the *VendorContact* view, except it gives the manager access to the customers contact information. This could enable the ability for the managers to target specific customers via marketing techniques and potentially send promotional incentives to the customers.

**Create Statement(s):**



**Sample Data Outputs for *VendorContact* and *CustomerContact* View(s):**

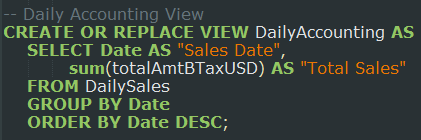


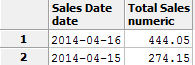


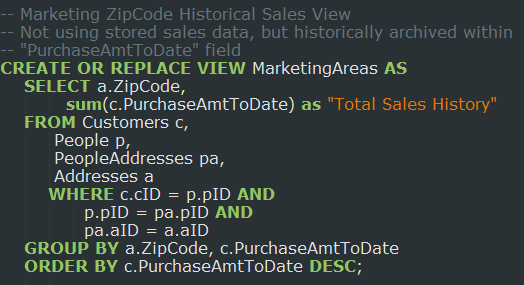
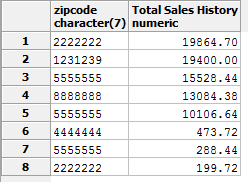
## DailyAccounting and MarketingAreas View(s):

The *DailyAccounting* view gives the managers the ability view daily sales data. This will give management, or the accounting department if the retailer has one, the ability to efficiently and effectively report before-tax sales figures. It is a fairly simplistic view; however, it provides valuable functionality for the average user— who may not have simple querying abilities—who wants the daily sales without UPC specifics

The *MarketingAreas* view is intended for marketing, if the retailer participates in marketing, in the sense that it shows certain zip codes that produce the most sales. By looking at this view, the marketing department, or managers, the ability to find areas that produce sales in order to focus on that market or to improve marketing conditions in the lacking areas. It is also important to note that this area sensitive information is built off of the data provided in the *Customers* table which tracks total purchase anounts per customer. This data is assumed to be arcived from the database, making it not available in the *DailySales* table—to narrow the sample data for the project.

**Create Statement(s): Sample Data Output(s):**



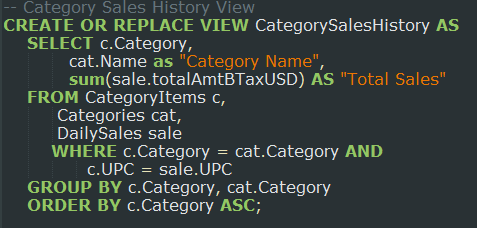


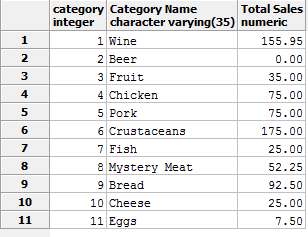
# Reports:

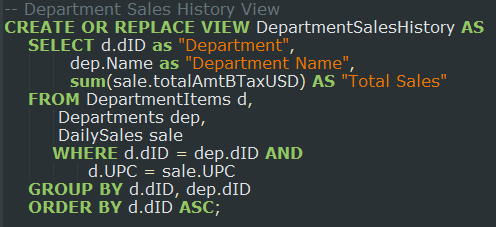
## CategorySalesHistory and DepartmentSalesHistory Report(s):

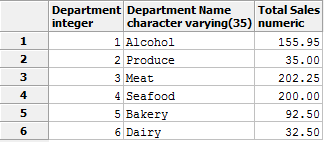
The *CategorySalesHistory* report creates a view that gives the managers, most applicably to the pricing manager, the ability to see the total sales history of a given category in their product matrix. This could help the overall strategy of how the retailer could strategies their product matrix.

The *DepartmentSalesHistory* report creates a view does essentially the same thing as the *CategorySalesHistory* report, except it gives the manager access to sales history of each department. This is a broader look at the retailer’s product matrix, as it helps track overall department performance relative to the other departments.

**Create Statement(s): Sample Data Output(s):**





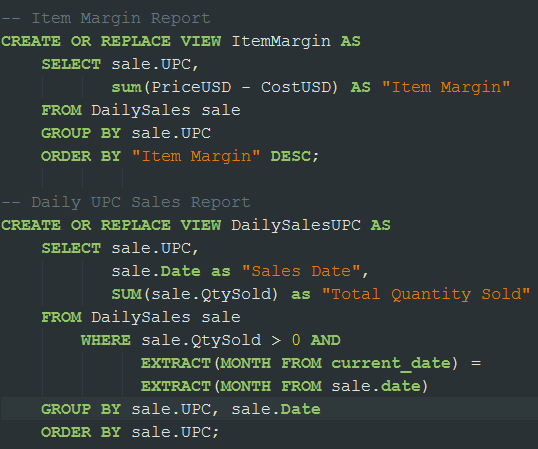
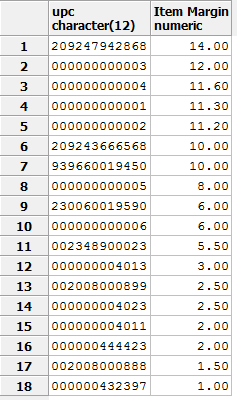


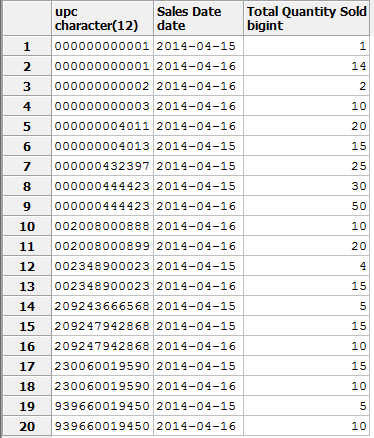
## ItemMargin and DailySalesUPC Report(s):

The *ItemMargin* report creates a view that gives the managers, most applicably to the pricing manager, the ability to look at the margins between the price they are offering the item and the cost they are paying from the vendor. With this data, they can see the highest marginally grossing items in order to push those through marketing or adjust their pricing strategy. They could extrapolate to adjust prices to induce more customer to buy the item, they could realize that the vendor’s cost are too high, they could broaden a certain variety of an advantageous item, etc…

The *DailySaleUPCs* report creates a view that will allow for a Pricing Manager, or another Manager, quickly look at the *DailySales* query as shown by per UPC per date. This will simplify shifting through this table, and will have the data storing the information necessary to see what sold on what days, and it will also eliminate all of the items that did not sell on a given date. In addition, this could enable inventory functionality as it could track what sold, and it could lead to figuring out how to replenish inventory. It can also show frequency of sales, to assure correct supply to meet customer.

**Create Statement(s): Sample Data Output(s):**

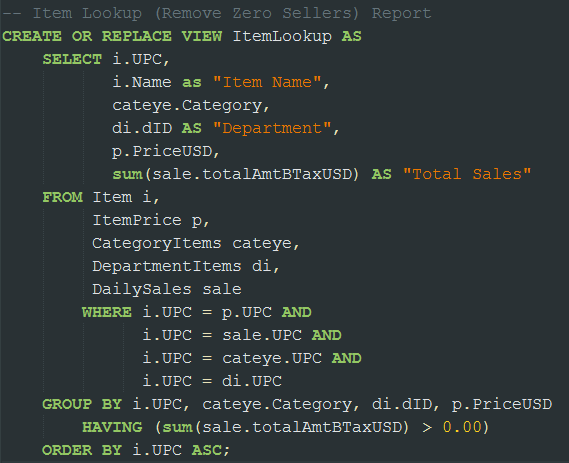
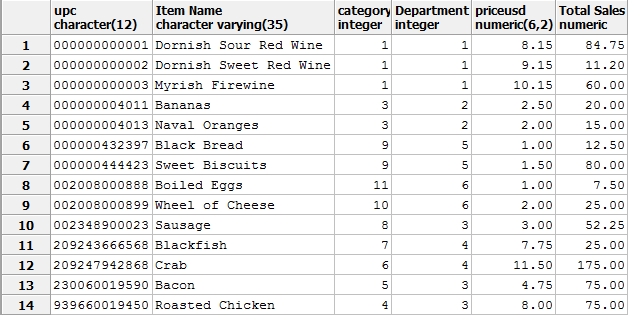




## ItemLookup Report:

The *ItemLookup* report creates a view that gives the managers, most applicably to the pricing manager, the ability to quickly access the UPC, name, price, category, department, and Total Sales from the product portfolio. It is yet another view for the managers to have a clean look at item data that is ordered in ascending order with various characteristics of the item. This view will also eliminate complexity by not displaying items that do not have any active sales figures.

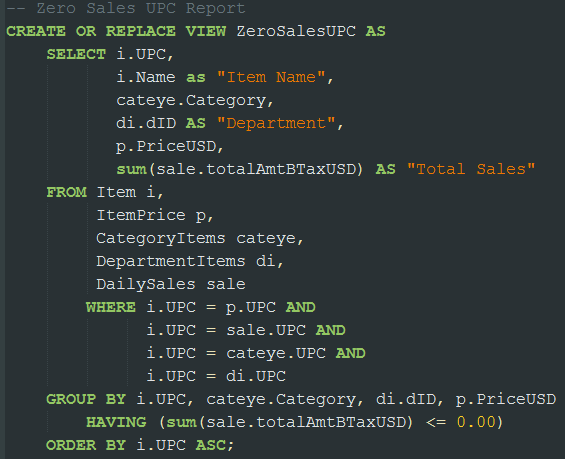
**Create Statement(s): Sample Data Output(s):**

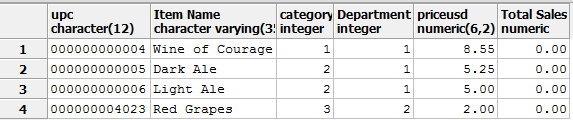


## ZeroSalesUPC Report:

The *ItemLookup* report creates a view that gives the managers, most applicably to the pricing manager, the inverse of the *ItemLookup* report, in which it gives the same data but for all the UPCs that do not have active sales figures. It seems as if everyone is getting their beer and courage at the locale inn instead of the locale market. Also, the only times people are eating grapes are when is fermented into liver poison! Maybe they should divest those items from their product portfolio and inventory.

**Create Statement(s):**





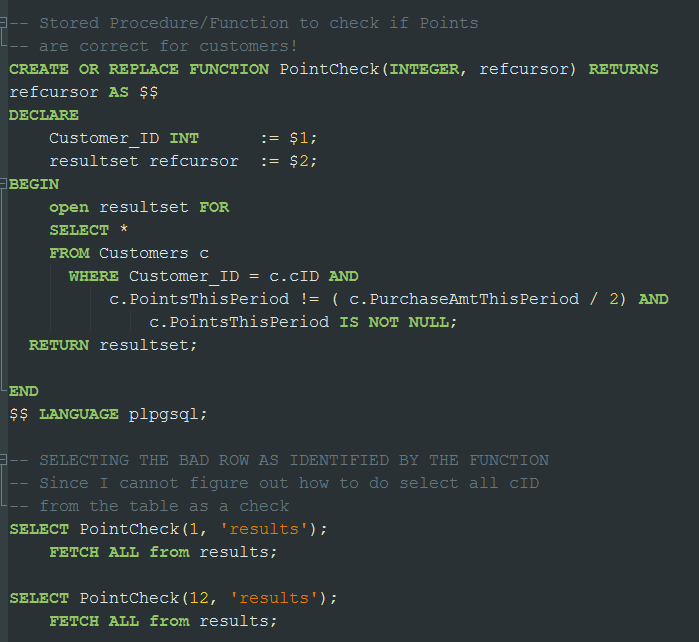
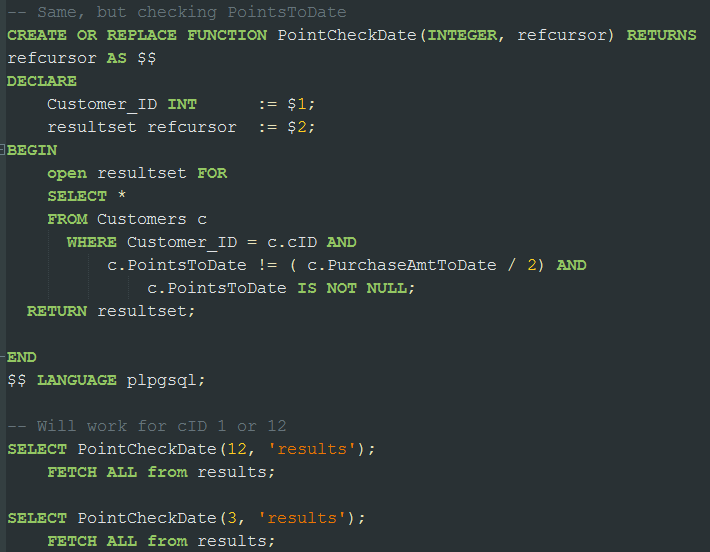
**Sample Data Output(s):**

# Store Procedures and Triggers:

## Customer Points Check Store Procedure(s):

These function, *PointCheck()* and *PointCheckDate()*, will return if the customer has a correct calculation for their “rewards point” data. The points are simply calculated as (Purchase Amount / 2). With the input of a “cID” this function can fetch if the data is incorrect. If it is correct their will be no output, but if the row is incorrect it will store all of the data in the *Customers* table regarding that given “cID.” This stored procedure is simplistic, yet if the function is run it will lead to the necessity of using other functions.

**Create Statement(s):**



**Sample Output(s):**

Running “SELECT PointCheck(1, ‘results);

FETCH ALL FROM results;”



Running “SELECT PointCheck(12, ‘results);

FETCH ALL FROM results;”



Running “SELECT PointCheckDate(12, ‘results);

FETCH ALL FROM results;”



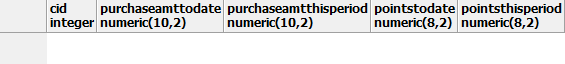
Running “SELECT PointCheckDate(1, ‘results);

FETCH ALL FROM results;”



Running “SELECT PointCheckDate(13, ‘results);

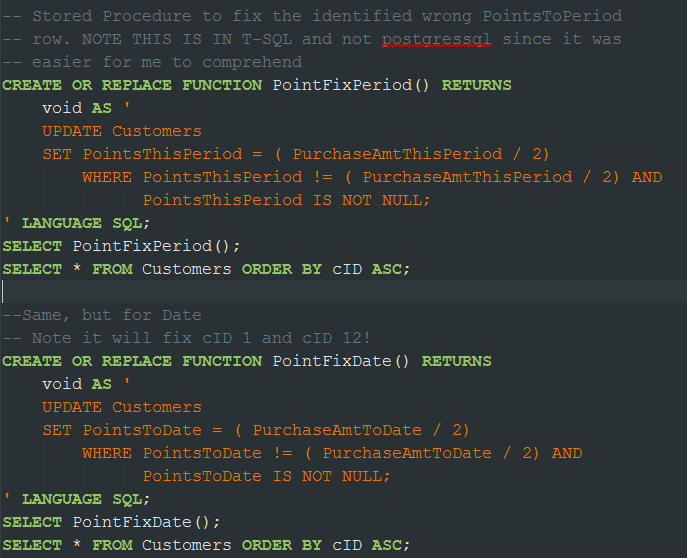
FETCH ALL FROM results;”



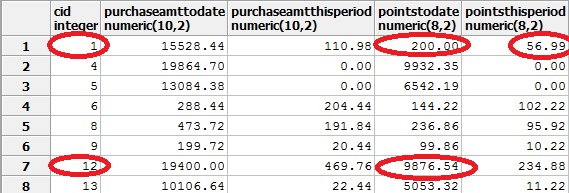
## Customer Points Fix Store Procedure(s):

These function, *PointFixPeriod()* and *PointFixDate()*, will fix the“rewards point” data that has been identified by the previous functions.

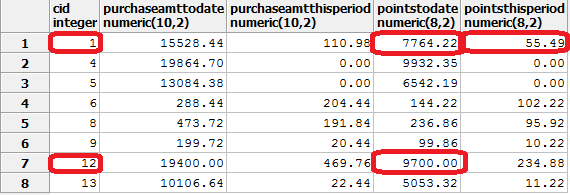
**Create Statement(s):** **Sample Data Output(s):**



Before Running Both Functions

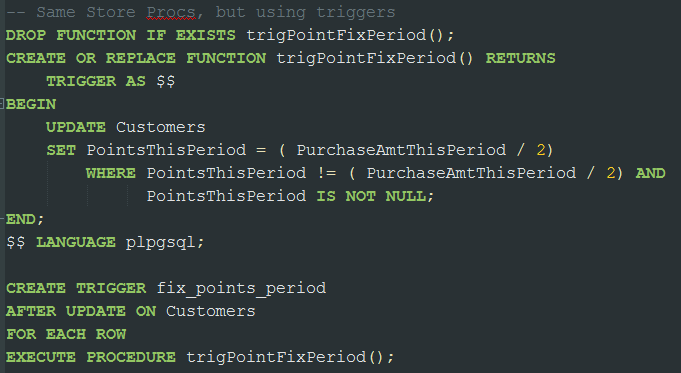


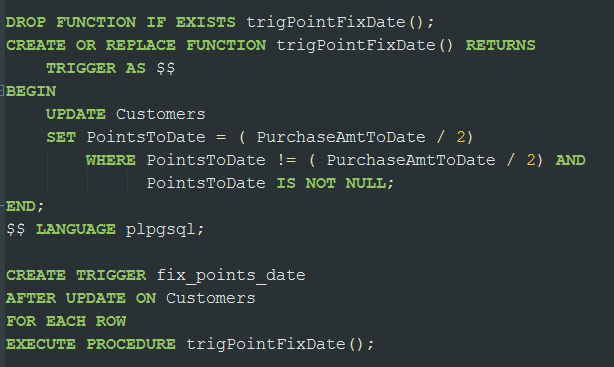
After Running Both Functions:



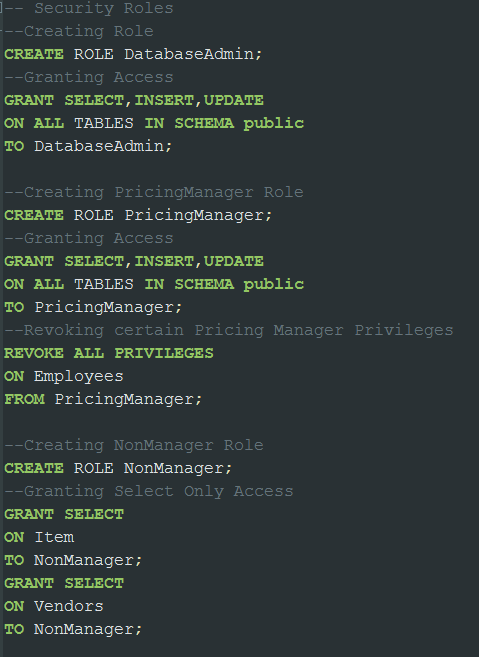
## Triggers:

The following queries are the “reward” data fixes with the use of triggers.





## Security:



These use of secrity within a database can restrict the access, and overall CRUD cycle capbilites of the user. In the case of this system, it restricitings could be religated to different employees—as they are the end-users of the database system. In the case of the provided sample data, the “dasebase administrator” could be considered the general manager and manager of the store. They have all the privlages to select, insert, update, or delete on all tables in the database. The “pricing manager” will have most of the same rights, except they wouldn’t have employee sensitive data. The other roles may be given brief selection rights from a few tables, as most of their privlages are revoked while interacting with the system.

# Implementation Notes:

The user should create a new database, under name any schema name of their choice for application for the goal of overall avoidance of commonality in other tables, procedures, triggers etc. that arestored on the server. This database system was also written and tested for a PostgresSQL server, version 9.3.4. The used of this version is highly recommended, as other versions may have restrictions to the functionality used within the database architecture and interactions.

# Known Problems:

* Need to have solution for Non-rewards customers to track sales.
* Currently this is only built for single store system.
* There is a lack of transactional support.
* The architecture could be a bit more intuitive to fit BCNF; however, this was limited by the functionality chosen to limit the scope of the project.
* Built for single Vendor per Item, many vendors should support a particular item and all of them should essentially bid for lowest prices by offering “allowances” or temporary vendor discounts.
* Possible need for adjustment of UPC changes to CHAR(15) or just an integer to support 15 numbers; since this is the industry trend as more UPCs are becoming globalized in the continually globally traded market.
* Lack of tracking inventory and minimal auditing support.

# Future Enhancements:

* Add default person and Customer for all none-rewarded customers to track sales as say pID (cID) = 0.
* Add Multiple Store Support for Scalability; this will drastically change the architecture as it would be integrated into the primary keys of most tables. Also, the complexity of the system would exponentially grow with this support.
* Add many other store procedures and triggers, such as updating DailySales on transactions
* Add transactional support with ACID property compliance at an isolation level of “Repeatable Read” to maintain speed and accuracy balance.
* Add Multiple Vendors per Item, this may also include functionality to order from lowest cost/margin
* Add Tables that track the future prices and costs of items that triggers amd updated when reach specified date
* Add support for Monthly Sales, as well as maybe weekly. Also, arhiece sales data to reduce spacing.
* Add Inventory Support to track the shelf, Aisle, and qty on hand of items, and what not.