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

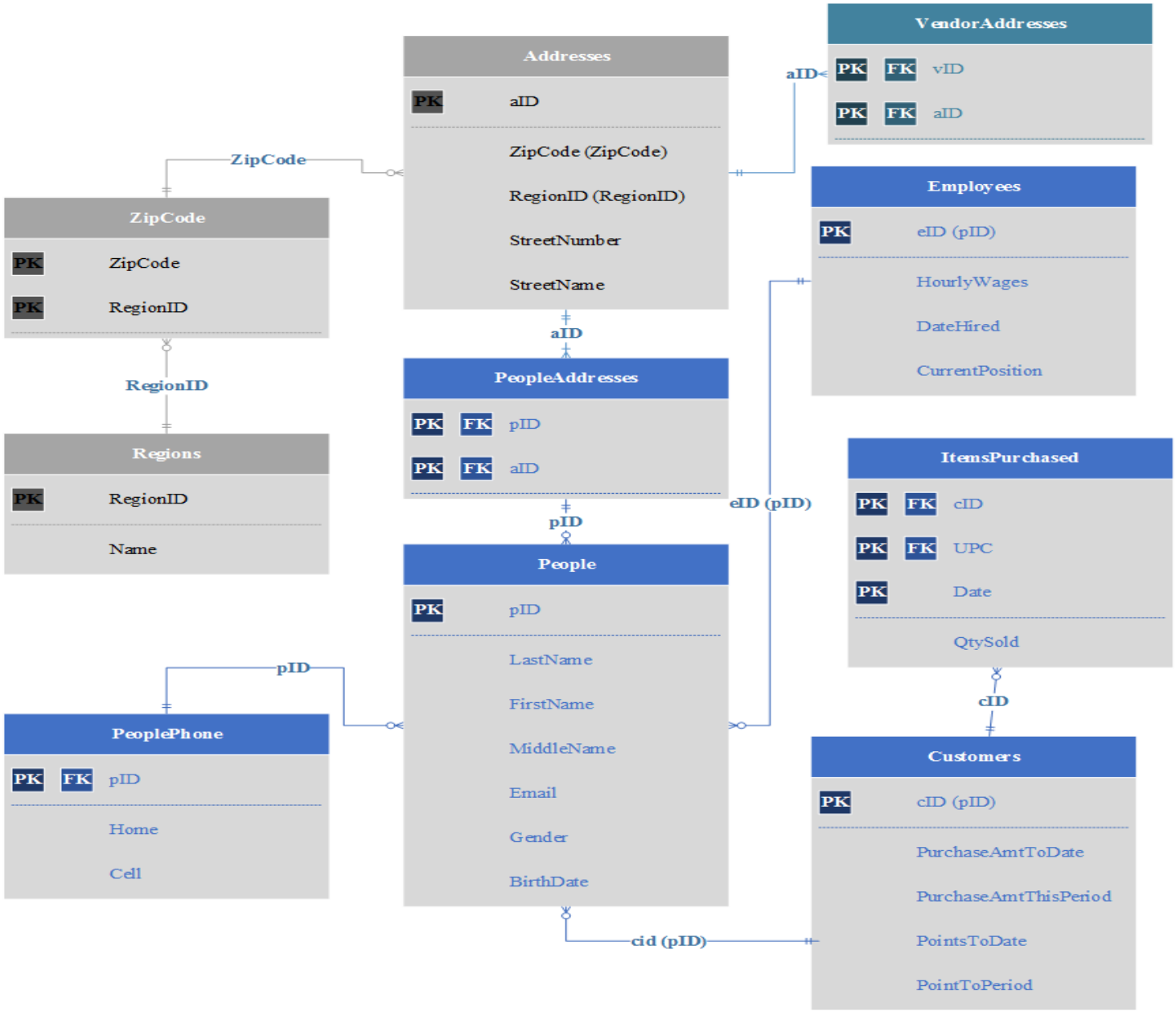
EXECUTIVE SUMMARY:

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.

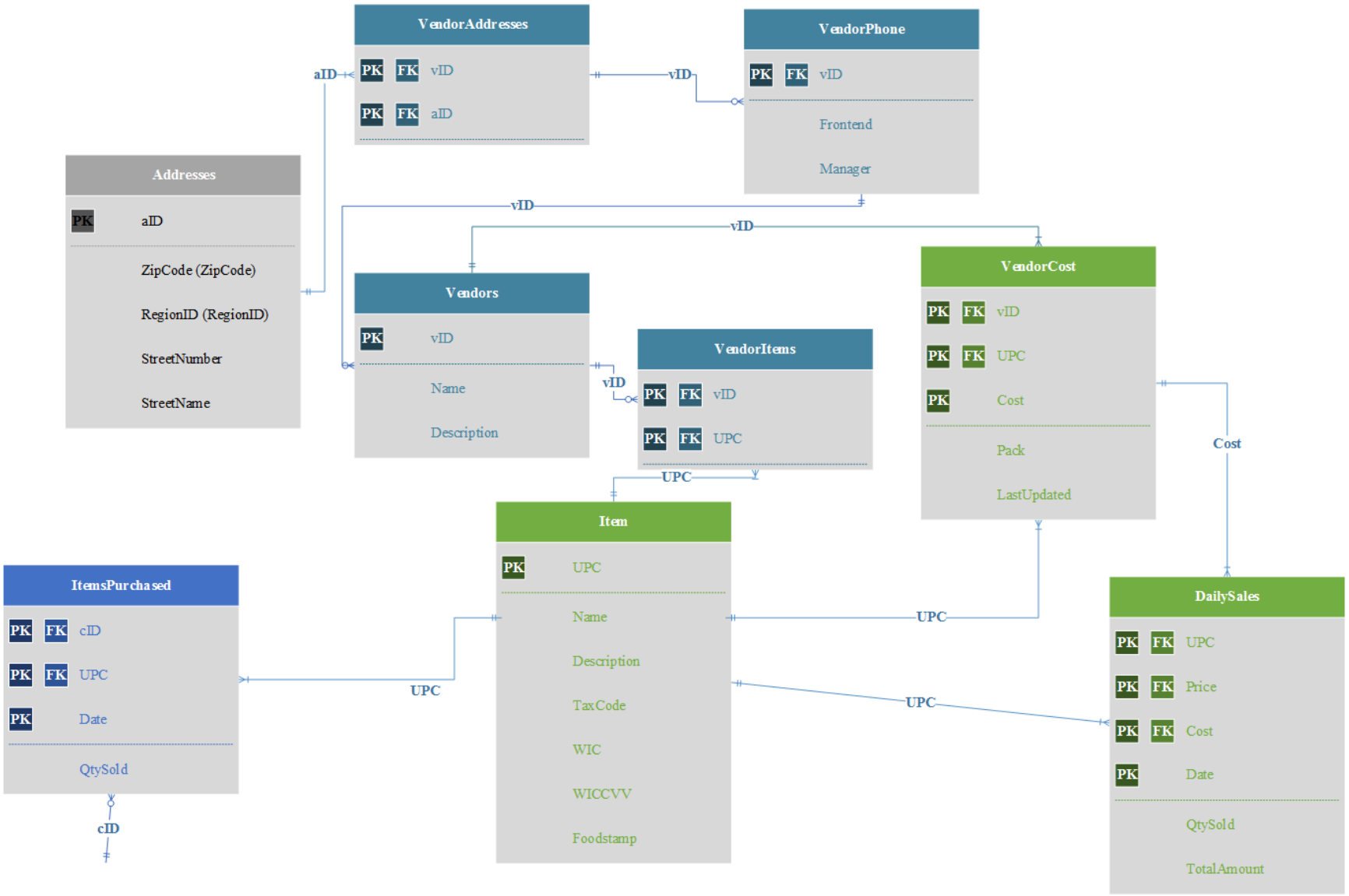
This database design is meant fit the scope and basic objectives for a back-office retail system (mainly focused on super-markets) that has the twist of using data referencing the great book series by George R.R Martin, *A Song of Ice and Fire*. The primary objective of this database design is to implement 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.

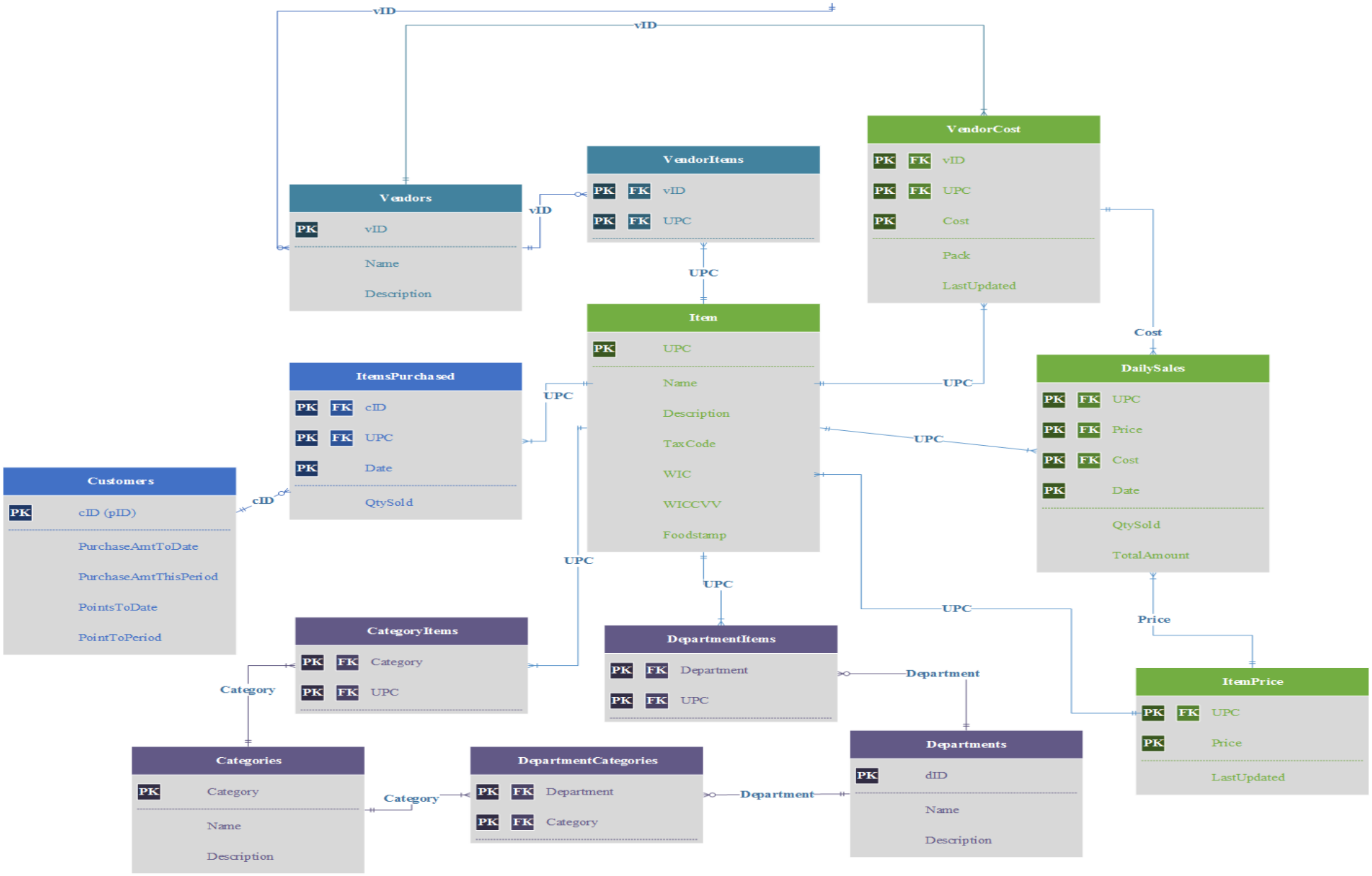
People and Addresses Entity-Relationship Diagram:



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

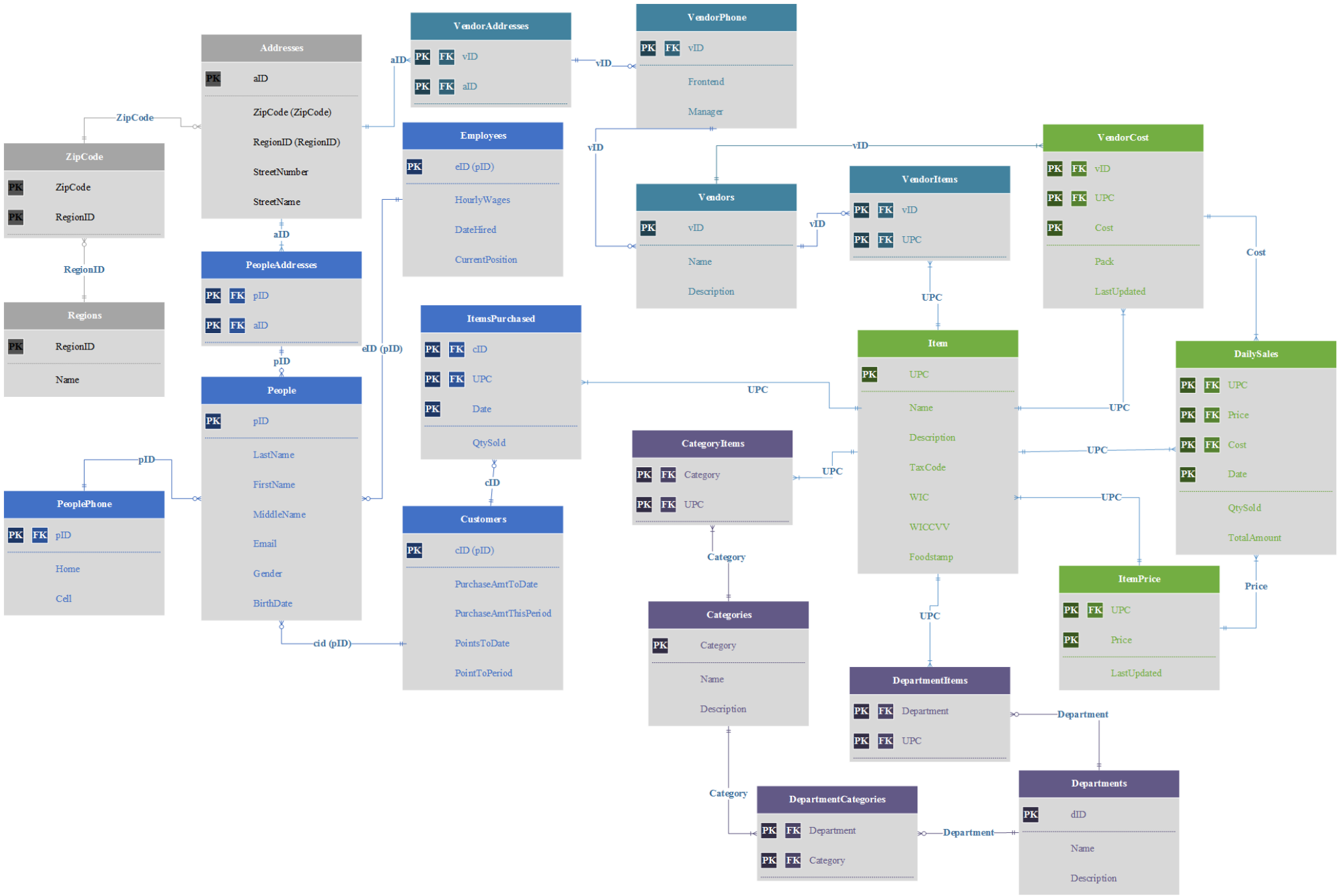


Item with Price, Cost, and Sales Entity-Relationship Diagram:



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

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:

```
-- Item Table
CREATE TABLE Item(
    UPC CHAR(12) NOT NULL,
    Name VARCHAR(35) NULL,
    Description TEXT NULL,
    TaxCode CHAR(1) NOT NULL DEFAULT 'A'
CHECK(TaxCode = 'A' OR TaxCode = 'B' OR TaxCode = 'C'
OR TaxCode = 'D' OR TaxCode = 'E' OR TaxCode = 'F'),
    WIC CHAR(1)NOT NULL DEFAULT '0'
CHECK(WIC = '0' OR WIC = '1'),
    WICCVV CHAR(1)NOT NULL DEFAULT '0'
CHECK(WICCVV = '0' OR WICCVV = '1'),
    Foodstamp CHAR(1)NOT NULL DEFAULT '0'
CHECK(Foodstamp = '0' OR Foodstamp = '1'),
    PRIMARY KEY (UPC)
);
```

Sample Data Output:

	upc character(12)	name character varying(35)	description text	taxcode character(1)	wic character(1)	wiccvv character(1)	foodstamp character(1)
1	000000000001	Dornish Sour Red Wine	The finest sour red wine in all	A	1	1	1
2	000000000002	Dornish Sweet Red Wine	The finest Sweet red wine in all	A	1	1	1
3	000000000003	Myrish Firewine	The finest red wine from the spi	A	1	1	1
4	000000000004	Wine of Courage	The wine of the Unsullied	A	1	1	1
5	000000000005	Dark Ale	Dark beer for the commons	B	1	1	1
6	000000000006	Light Ale	Light beer for the commons	B	1	1	1
7	000000004011	Bananas	Plain ole bananas	C	0	0	1
8	000000004013	Naval Oranges	ORANGES!!!	C	0	0	1
9	000000004023	Red Grapes	Grapes(for wine)	C	0	0	1
10	939660019450	Roasted Chicken	From the local Inn	D	1	1	0
11	230060019590	Bacon	Cant have breakfast without baco	D	1	1	0
12	209247942868	Crab	Freshly spiked from the Iron Isl	D	1	1	0
13	209243666568	Blackfish	Not the house but the food!	D	1	1	0
14	002348900023	Sausage	Mystery meat ground up	D	1	1	0
15	002008000888	Boiled Eggs	Common food	F	1	1	1
16	002008000899	Wheel of Cheese	Compliments every meal	F	1	1	1
17	000000444423	Sweet Biscuits	YUM!	E	0	0	1
18	000000432397	Black Bread	Sounds gross, tastes great!	E	0	0	1

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):

```
-- Categories Table
CREATE TABLE Categories(
    Category INTEGER NOT NULL,
    Name VARCHAR(35) NULL,
    Description TEXT NULL,
    PRIMARY KEY (Category)
);

-- Departments Table
CREATE TABLE Departments(
    dID INTEGER NOT NULL,
    Name VARCHAR(35) NULL,
    Description TEXT NULL,
    PRIMARY KEY (dID)
);
```

Sample Data Output(s):

	category integer	name character varyin	description text
1	1	Wine	In every chapter of the books
2	2	Beer	A staple in every household
3	3	Fruit	Gotta have something healthy
4	4	Chicken	Organically raised!
5	5	Pork	Oink Oink!
6	6	Crustaceans	Peel off the shells!
7	7	Fish	The Fish is fishy!
8	8	Mystery Meat	Most common food
9	9	Bread	Hearty Meal
10	10	Cheese	Yummmm cheese
11	11	Eggs	Yup, eggs!

	did integer	name character varying(35)	description text
1	1	Alcohol	Once again in every scene
2	2	Produce	For your health!
3	3	Meat	All Organic!
4	4	Seafood	See food and eat it
5	5	Bakery	Freshly made fluffy things
6	6	Dairy	Creamy stuff

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:

```
-- People Table
CREATE TABLE People(
    pID INTEGER NOT NULL,
    LastName VARCHAR(35) NOT NULL,
    FirstName VARCHAR(35) NOT NULL,
    MiddleName VARCHAR(35) NULL,
    Email TEXT NULL,
-- Checking Gender, M = Male, F = Female,
--  N = No Disclosure or Other
    Gender CHAR(1) NOT NULL CHECK
(Gender = 'M' OR Gender = 'F' OR Gender = 'N'),
    BirthDate DATE NULL,
    PRIMARY KEY (pID)
);
```

```
-- People Phone Table
CREATE TABLE PeoplePhone(
    pID INTEGER REFERENCES People(pID),
    Home TEXT NULL,
    Cell TEXT NULL,
    PRIMARY KEY (pID)
);
```

Sample Data Output:

	pid integer	home text	cell text
1	1	123-222-5332	123-222-2342
2	2	234-222-6645	234-888-8865
3	3	234-222-6643	234-888-8863
4	4	111-111-2357	111-111-8764
5	5	111-111-4323	111-111-2874
6	6	111-111-3234	111-111-4382
7	7	111-111-7373	111-111-0756
8	8	111-111-3737	111-111-2342
9	9	111-111-9864	111-111-4264
10	10	111-111-5743	111-111-2363
11	11	999-222-5332	999-888-9876
12	12	222-324-7496	222-324-7040
13	13	222-634-3234	222-324-0001

	pid integer	lastname character varying(35)	firstname character varying(35)	middlename character varying(35)	email text	gender character(1)	birthdate date
1	1	Lannister	Tyrion	S	dwarf@hotmail.com	M	1980-04-13
2	2	Lannister	Cersei	B	queen@forever.com	F	1970-09-11
3	3	Lannister	Jamie	I	kingsguard1@gmail.com	M	1970-09-11
4	4	Stark	Ned	D	fatherstark@aol.com	M	1960-01-01
5	5	Stark	Catelyn	D	cattulley@gmail.com	F	1962-02-12
6	6	Stark	Bran	D	worg@yahoo.com	M	2000-02-22
7	7	Stark	Sansa	R	redhead@gmail.com	F	1996-05-22
8	8	Stark	Arya	A	bravvosninja@revenge.com	F	1998-05-04
9	9	Stark	Robb	D	kingofthenorth@ouch.com	M	1990-11-27
10	10	Snow	Jon	K	watcherofthewall@gmail.com	M	1985-07-17
11	11	Targaryen	Daenerys	K	motherofdragons@hotmail.com	F	1996-04-14
12	12	Baratheon	Stannis	R	rightfulheir@aol.com	M	1966-09-09
13	13	Baratheon	Renly	E	numberonestag@gmail.com	M	1979-08-02

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):

```
-- Vendors Table
CREATE TABLE Vendors(
    vID INTEGER NOT NULL,
    Name VARCHAR(35) NULL,
    Description TEXT NULL,
    Email TEXT NULL,
    PRIMARY KEY (vID)
);

-- Vendor Phone Table
CREATE TABLE VendorPhone(
    vID INTEGER REFERENCES Vendors(vID),
    Frontend TEXT NULL,
    Manager TEXT NULL,
    PRIMARY KEY (vID)
);
```

Sample Data Output(s):

	vid integer	name character varying(35)	description text	email text
1	1	Alcohol Smuggler	They get you all the alcohol	DavosSeaworthy@gmail.com
2	2	Blackwater Bay	Just watch out for chains	Wildfire@hotmail.com
3	3	Farm of the NORTH!	Its in the north!	Knights Watch@aol.com
4	4	Iron Islands Inc.	Fishy things	Kraken@yahoo.com
5	5	Casterly Rock Inc.	The Google of Westeros	Alwayspaysoffdebt@lannister.com
6	6	CrackJaw Point	Bay of Crabs	crablover@msn.com
7	7	Crossroads Inn Farm	Middle Westeros local farm	farmer@farming.com

	vid integer	frontend text	manager text
1	1	229-252-5332	229-521-2342
2	2	326-211-1111	326-888-2325
3	3	534-222-5332	123-222-2342
4	4	007-532-5332	007-324-3245
5	5	888-888-8868	888-678-6524
6	6	007-222-5332	007-222-3235
7	7	123-235-4324	123-346-8945

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:

```
-- Customers Table
CREATE TABLE Customers(
    cID INTEGER REFERENCES People(pID),
    PurchaseAmtToDate NUMERIC(10,2) NOT NULL,
    PurchaseAmtThisPeriod NUMERIC(10,2) NOT NULL,
    PointsToDate NUMERIC(8,2) NOT NULL,
    PointsThisPeriod NUMERIC(8,2) NOT NULL,
    PRIMARY KEY (cID)
);
```

Sample Data Output:

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)
1	1	15528.44	110.98	7764.22	56.99
2	4	19864.70	0.00	9932.35	0.00
3	5	13084.38	0.00	6542.19	0.00
4	6	288.44	204.44	144.22	102.22
5	8	473.72	191.84	236.86	95.92
6	9	199.72	20.44	99.86	10.22
7	12	19400.00	469.76	9876.54	234.88
8	13	10106.64	22.44	5053.32	11.22

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:

```
-- Employees Table
CREATE TABLE Employees(
    eID INTEGER REFERENCES People(pID),
    HourlyWages NUMERIC(4,2) NOT NULL,
    DateHired DATE NOT NULL,
    CurrentPosition TEXT NOT NULL,
    PRIMARY KEY (eID)
);
```

Sample Data Output:

	eid integer	hourlywages numeric(4,2)	datehired date	currentposition text
1	11	23.45	2009-01-01	General Manager
2	10	22.45	2009-01-01	Manager
3	7	8.59	2013-09-12	Cashier
4	2	19.00	2009-01-01	Pricing Manager
5	3	19.00	2009-01-01	Stocker and Janitor

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):

```
-- Addresses Table
CREATE TABLE Addresses(
    aID INTEGER NOT NULL,
    ZipCode CHAR(7) REFERENCES ZipCodes(ZipCode),
    RegionID INTEGER REFERENCES Regions(RegionID),
    StreetNumber INTEGER NULL,
    StreetName VARCHAR(35) NOT NULL,
    PRIMARY KEY (aID)
);

-- People Addresses Table
CREATE TABLE PeopleAddresses(
    pID INTEGER REFERENCES People(pID),
    aID INTEGER REFERENCES Addresses(aID),
    PRIMARY KEY (pID,aID)
);

-- Vendor Addresses Table
CREATE TABLE VendorAddresses(
    vID INTEGER REFERENCES Vendors(vID),
    aID INTEGER REFERENCES Addresses(aID),
    PRIMARY KEY (vID,aID)
);
```

Sample Data Output(s):

	aid integer	zipcode character(7)	regionid integer	streetnumber integer	streetname character varying(35)
1	1	1111111	1	2	Rich Lane
2	2	1111111	1	288	Poor Drive
3	3	2222222	2	1	Winter Wall Street
4	4	3333333	3	75	Blacktyde Lane
5	5	3333333	3	2999	Pyke Boulevard
6	6	4444444	4	987	Andalos Lane
7	7	5555555	5	7	Bond Street
8	8	5555555	5	1	Lannister Court
9	9	6666666	6	2423	Godsgrace Street
10	10	7777777	7	77	Seagard Lane
11	11	8888888	8	99999	Eyrie Cliff Court
12	12	9999999	9	500	Dothraki Street
13	13	1231239	10	8765	Dario Drive

	vid integer	aid integer
1	1	12
2	2	10
3	3	4
4	4	5
5	5	9
6	6	5
7	7	8

	pid integer	aid integer
1	1	8
2	2	1
3	3	1
4	4	3
5	5	11
6	6	7
7	7	2
8	8	6
9	9	3
10	10	7
11	11	12
12	12	13
13	13	7

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):

```
-- Region Table
-- Taking Place of State Table because of the data provided
CREATE TABLE Regions(
    RegionID INTEGER NOT NULL,
    Name VARCHAR(35) NOT NULL,
    PRIMARY KEY (RegionID)
);

-- Zip Code Table
CREATE TABLE ZipCodes(
    ZipCode CHAR(7) NOT NULL,
    RegionID INTEGER REFERENCES Regions(RegionID),
    PRIMARY KEY (ZipCode)
);
```

Sample Data Output(s):

	regionid integer	name character varying(35)
1	1	Kings Landing
2	2	Winterfell
3	3	Iron Islands
4	4	Braavos
5	5	Casterly Rock
6	6	Dorne
7	7	The Twins
8	8	The Vale of Arryn
9	9	Pentos
10	10	Meereen

	zipcode character(7)	regionid integer
1	1111111	1
2	2222222	2
3	3333333	3
4	4444444	4
5	5555555	5
6	6666666	6
7	7777777	7
8	8888888	7
9	9999999	9
10	1231239	10

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):

```
--Category Items Table
CREATE TABLE CategoryItems(
    Category INTEGER REFERENCES Categories(Category),
    UPC CHAR(12) REFERENCES Item(UPC),
    PRIMARY KEY (Category,UPC)
);

-- Department Categories  Table
CREATE TABLE DepartmentCategories(
    dID INTEGER REFERENCES Departments(dID),
    Category INTEGER REFERENCES Categories(Category),
    PRIMARY KEY (dID,Category)
);
```

Sample Data Output(s):

	category integer	upc character(12)
1	1	000000000001
2	1	000000000002
3	1	000000000003
4	1	000000000004
5	2	000000000005
6	2	000000000006
7	3	000000004011
8	3	000000004013
9	3	000000004023
10	4	939660019450
11	5	230060019590
12	6	209247942868
13	7	209243666568
14	8	002348900023
15	11	002008000888
16	10	002008000899
17	9	000000444423
18	9	000000432397

	did integer	category integer
1	1	1
2	1	2
3	2	3
4	3	4
5	3	5
6	4	6
7	4	7
8	3	8
9	5	9
10	6	10
11	6	11

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):

```
-- Department Items Table
CREATE TABLE DepartmentItems(
    dID INTEGER REFERENCES Departments(dID),
    UPC CHAR(12) REFERENCES Item(UPC),
    PRIMARY KEY (dID,UPC)
);

-- Vendor Items Table
CREATE TABLE VendorItems(
    vID INTEGER REFERENCES Vendors(vID),
    UPC CHAR(12) REFERENCES Item(UPC),
    PRIMARY KEY (vID,UPC)
);
```

Sample Data Output(s):

	did integer	upc character(12)
5	1	000000000005
6	1	000000000006
7	2	000000004011
8	2	000000004013
9	2	000000004023
10	3	939660019450
11	3	230060019590
12	4	209247942868
13	4	209243666568
14	3	002348900023
15	6	002008000888
16	6	002008000899
17	5	000000444423
18	5	000000432397

	vid integer	upc character(12)
1	1	000000000001
2	1	000000000002
3	1	000000000003
4	1	000000000004
5	1	000000000005
6	1	000000000006
7	3	000000004011
8	3	000000004013
9	7	000000004023
10	2	939660019450
11	2	230060019590
12	6	209247942868
13	4	209243666568
14	2	002348900023
15	5	002008000888
16	6	002008000899
17	5	000000444423
18	5	000000432397

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:

```
-- Items Purchased Table
CREATE TABLE ItemsPurchased(
    cID INTEGER REFERENCES Customers(cID),
    UPC CHAR(12) REFERENCES Item(UPC),
-- Using date as part of Key since it tracks daily use.
    Date DATE NOT NULL,
-- Assuming someone can't buy more than 99 of a given item...
    QtySold SMALLINT NOT NULL,
    PRIMARY KEY (cID,UPC,Date)
);
```

Sample Data Output:

	cid integer	upc character(12)	date date	qtysold smallint
1	1	000000000001	2014-04-16	4
2	1	000000000002	2014-04-16	2
3	1	209247942868	2014-04-15	5
4	6	000000004011	2014-04-16	20
5	6	002008000899	2014-04-16	20
6	6	002348900023	2014-04-16	15
7	6	000000444423	2014-04-15	30
8	6	000000432397	2014-04-15	15
9	8	000000444423	2014-04-16	50
10	8	230060019590	2014-04-16	10
11	8	000000004013	2014-04-15	15
12	8	939660019450	2014-04-15	5
13	9	002008000888	2014-04-16	10
14	9	000000432397	2014-04-15	10
15	12	000000000001	2014-04-16	10
16	12	000000000003	2014-04-16	10
17	12	209247942868	2014-04-16	10
18	12	939660019450	2014-04-16	10
19	12	209243666568	2014-04-15	5
20	12	230060019590	2014-04-15	15
21	13	002348900023	2014-04-15	4
22	13	000000000001	2014-04-15	1

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:

```
-- Item Price Table
CREATE TABLE ItemPrice(
    UPC CHAR(12) REFERENCES Item(UPC),
    PriceUSD NUMERIC(6,2) NOT NULL,
    PRIMARY KEY (UPC,PriceUSD)
);

-- Vendor Cost Table
CREATE TABLE VendorCost(
    vID INTEGER REFERENCES Vendors(vID),
    UPC CHAR(12) REFERENCES Item(UPC),
    CostUSD NUMERIC(6,2) NOT NULL,
-- Pack = Qty of Items supplied by Vendor per order
    Pack INTEGER NOT NULL,
    PRIMARY KEY (vID,UPC,CostUSD)
);
```

Sample Data Output:

	upc character(12)	priceusd numeric(6,2)
1	000000000001	8.15
2	000000000002	9.15
3	000000000003	10.15
4	000000000004	8.55
5	000000000005	5.25
6	000000000006	5.00
7	000000004011	2.50
8	000000004013	2.00
9	000000004023	2.00
10	939660019450	8.00
11	230060019590	4.75
12	209247942868	11.50
13	209243666568	7.75
14	002348900023	3.00
15	002008000888	1.00
16	002008000899	2.00
17	000000444423	1.50
18	000000432397	1.00

	vid integer	upc character(12)	costusd numeric(6,2)	pack integer
1	1	000000000001	2.50	12
2	1	000000000002	3.55	12
3	1	000000000003	4.15	12
4	1	000000000004	2.75	12
5	1	000000000005	1.25	12
6	1	000000000006	2.00	12
7	3	000000004011	1.50	20
8	3	000000004013	0.50	20
9	7	000000004023	0.75	20
10	2	939660019450	3.00	10
11	2	230060019590	1.75	10
12	6	209247942868	4.50	10
13	4	209243666568	2.75	10
14	2	002348900023	0.25	25
15	5	002008000888	0.25	25
16	6	002008000899	0.75	20
17	5	000000444423	0.50	40
18	5	000000432397	0.50	40

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:

```
-- Daily Sales Table
CREATE TABLE DailySales(
    UPC CHAR(12) REFERENCES Item(UPC),
    PriceUSD NUMERIC(6,2) NOT NULL,
    CostUSD NUMERIC(6,2) NOT NULL,
    Date DATE NOT NULL,
    QtySold SMALLINT NOT NULL,
-- Total amount of sales before tax in USD
    TotalAmtBTaxUSD NUMERIC(8,2) NOT NULL,
    PRIMARY KEY (UPC,PriceUSD,CostUSD,Date)
);
```

Sample Data Output:

	upc character(12)	priceusd numeric(6,2)	costusd numeric(6,2)	date date	qtysold smallint	totalamtbtaxusd numeric(8,2)
1	000000000001	8.15	2.50	2014-04-16	14	79.10
2	000000000002	9.15	3.55	2014-04-16	2	11.20
3	000000000003	10.15	4.15	2014-04-16	10	60.00
4	000000000004	8.55	2.75	2014-04-16	0	0.00
5	000000000005	5.25	1.25	2014-04-16	0	0.00
6	000000000006	5.00	2.00	2014-04-16	0	0.00
7	000000004011	2.50	1.50	2014-04-16	20	20.00
8	000000004013	2.00	0.50	2014-04-16	0	0.00
9	000000004023	2.00	0.75	2014-04-16	0	0.00
10	939660019450	8.00	3.00	2014-04-16	10	50.00
11	230060019590	4.75	1.75	2014-04-16	10	30.00
12	209247942868	11.50	4.50	2014-04-16	10	70.00
13	209243666568	7.75	2.75	2014-04-16	0	0.00
14	002348900023	3.00	0.25	2014-04-16	15	41.25
15	002008000888	1.00	0.25	2014-04-16	10	7.50
16	002008000899	2.00	0.75	2014-04-16	20	25.00
17	000000444423	1.50	0.50	2014-04-16	50	50.00
18	000000432397	1.00	0.50	2014-04-16	0	0.00
19	000000000001	8.15	2.50	2014-04-15	1	5.65
20	000000000002	9.15	3.55	2014-04-15	0	0.00
21	000000000003	10.15	4.15	2014-04-15	0	0.00
22	000000000004	8.55	2.75	2014-04-15	0	0.00
23	000000000005	5.25	1.25	2014-04-15	0	0.00
24	000000000006	5.00	2.00	2014-04-15	0	0.00
25	000000004011	2.50	1.50	2014-04-15	0	0.00
26	000000004013	2.00	0.50	2014-04-15	15	15.00
27	000000004023	2.00	0.75	2014-04-15	0	0.00
28	939660019450	8.00	3.00	2014-04-15	5	25.00
29	230060019590	4.75	1.75	2014-04-15	15	45.00
30	209247942868	11.50	4.50	2014-04-15	15	105.00
31	209243666568	7.75	2.75	2014-04-15	5	25.00
32	002348900023	3.00	0.25	2014-04-15	4	11.00
33	002008000888	1.00	0.25	2014-04-15	0	0.00
34	002008000899	2.00	0.75	2014-04-15	0	0.00
35	000000444423	1.50	0.50	2014-04-15	30	30.00
36	000000432397	1.00	0.50	2014-04-15	25	12.50

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):

```
CREATE OR REPLACE VIEW VendorContact AS
   SELECT v.vID,
          v.Name as "Vendor Name",
          v.Email,
          vc.Frontend as "Front-End Phone Number",
          vc.Manager as "Manager's Phone Number",
          (a.StreetNumber || ' ' || a.StreetName) as "Address",
          a.ZipCode,
          r.Name as "Region Name"
   FROM Vendors v,
        VendorPhone vc,
        VendorAddresses va,
        Regions r,
        ZipCodes z,
        Addresses a
        WHERE v.vID = vc.vID AND
          v.vID = va.vID AND
          va.aID = a.aID AND
          a.ZipCode = z.ZipCode AND
          r.RegionID = z.RegionID
   ORDER BY v.vID ASC;
```

```
-- Customer Contact View
CREATE OR REPLACE VIEW CustomerContact AS
   SELECT c.cID,
          (p.FirstName || ' ' || p.LastName) as "Customer Name",
          p.email,
          pc.Home as "Home Phone Number",
          pc.Cell as "Cell Phone Number",
          (a.StreetNumber || ' ' || a.StreetName) as "Address",
          a.ZipCode,
          r.Name as "Region Name"
   FROM Customers c,
        PeoplePhone pc,
        PeopleAddresses pa,
        People p,
        Regions r,
        ZipCodes z,
        Addresses a
        WHERE c.cID = pc.pID AND
          c.cID = pa.pID AND
          c.cID = p.pID AND
          pa.aID = a.aID AND
          r.RegionID = z.RegionID AND
          r.RegionID = a.RegionID AND
          z.ZipCode = a.ZipCode
        ORDER BY c.cID ASC;
```

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

	vid integer	Vendor Name character varying(35)	email text	Front-End Phone Number text	Manager's Phone Number text	Address text	zipcode character(7)	Region Name character varying(35)
1	1	Alcohol Smuggler	DavosSeaworthy@gmail.com	229-252-5332	229-521-2342	500 Dothraki Street	9999999	Pentos
2	2	Blackwater Bay	Wildfire@hotmail.com	326-211-1111	326-888-2325	77 Seagard Lane	7777777	The Twins
3	3	Farm of the NORTH!	Knights Watch@aol.com	534-222-5332	123-222-2342	1 Winter Wall Street	2222222	Winterfell
4	4	Iron Islands Inc.	Kraken@yahoo.com	007-532-5332	007-324-3245	2999 Pyke Boulevard	3333333	Iron Islands
5	5	Casterly Rock Inc.	Alwayspaysoffdebt@lannister.com	888-888-8868	888-678-6524	2423 Godsgrace Street	6666666	Dorne
6	6	CrackJaw Point	crablover@msn.com	007-222-5332	007-222-3235	2999 Pyke Boulevard	3333333	Iron Islands
7	7	Crossroads Inn Farm	farmer@farming.com	123-235-4324	123-346-8945	1 Lannister Court	5555555	Casterly Rock

	cid integer	Customer Name text	email text	Home Phone Number text	Cell Phone Number text	Address text	zipcode character(7)	Region Name character varying(35)
1	1	Tyrion Lannister	dwarf@hotmail.com	123-222-5332	123-222-2342	1 Lannister Court	5555555	Casterly Rock
2	4	Ned Stark	fatherstark@aol.com	111-111-2357	111-111-8764	1 Winter Wall Street	2222222	Winterfell
3	6	Bran Stark	worg@yahoo.com	111-111-3234	111-111-4382	7 Bond Street	5555555	Casterly Rock
4	8	Arya Stark	bravvosninja@revenge.com	111-111-3737	111-111-2342	987 Andalos Lane	4444444	Braavos
5	9	Robb Stark	kingofthenorth@ouch.com	111-111-9864	111-111-4264	1 Winter Wall Street	2222222	Winterfell
6	12	Stannis Baratheon	rightfulair@aol.com	222-324-7496	222-324-7040	8765 Dario Drive	1231239	Meereen
7	13	Renly Baratheon	numberonestag@gmail.com	222-634-3234	222-324-0001	7 Bond Street	5555555	Casterly Rock

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):

```
-- Daily Accounting View
CREATE OR REPLACE VIEW DailyAccounting AS
   SELECT Date AS "Sales Date",
         sum(totalAmtBTaxUSD) AS "Total Sales"
   FROM DailySales
   GROUP BY Date
   ORDER BY Date DESC;
```

```
-- Marketing ZipCode Historical Sales View
-- Not using stored sales data, but historically archived within
-- "PurchaseAmtToDate" field
CREATE OR REPLACE VIEW MarketingAreas AS
   SELECT a.ZipCode,
         sum(c.PurchaseAmtToDate) as "Total Sales History"
   FROM Customers c,
      People p,
      PeopleAddresses pa,
      Addresses a
      WHERE c.cID = p.pID AND
         p.pID = pa.pID AND
         pa.aID = a.aID
   GROUP BY a.ZipCode, c.PurchaseAmtToDate
   ORDER BY c.PurchaseAmtToDate DESC;
```

Sample Data Output(s):

	Sales Date date	Total Sales numeric
1	2014-04-16	444.05
2	2014-04-15	274.15

	zipcode character(7)	Total Sales History numeric
1	2222222	19864.70
2	1231239	19400.00
3	5555555	15528.44
4	8888888	13084.38
5	5555555	10106.64
6	4444444	473.72
7	5555555	288.44
8	2222222	199.72

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):

```
-- Category Sales History View
CREATE OR REPLACE VIEW CategorySalesHistory AS
    SELECT c.Category,
           cat.Name as "Category Name",
           sum(sale.totalAmtBTaxUSD) AS "Total Sales"
    FROM CategoryItems c,
         Categories cat,
         DailySales sale
        WHERE c.Category = cat.Category AND
              c.UPC = sale.UPC
    GROUP BY c.Category, cat.Category
    ORDER BY c.Category ASC;

-- Department Sales History View
CREATE OR REPLACE VIEW DepartmentSalesHistory AS
    SELECT d.dID as "Department",
           dep.Name as "Department Name",
           sum(sale.totalAmtBTaxUSD) AS "Total Sales"
    FROM DepartmentItems d,
         Departments dep,
         DailySales sale
        WHERE d.dID = dep.dID AND
              d.UPC = sale.UPC
    GROUP BY d.dID, dep.dID
    ORDER BY d.dID ASC;
```

Sample Data Output(s):

	category integer	Category Name character varying(35)	Total Sales numeric
1	1	Wine	155.95
2	2	Beer	0.00
3	3	Fruit	35.00
4	4	Chicken	75.00
5	5	Pork	75.00
6	6	Crustaceans	175.00
7	7	Fish	25.00
8	8	Mystery Meat	52.25
9	9	Bread	92.50
10	10	Cheese	25.00
11	11	Eggs	7.50

	Department integer	Department Name character varying(35)	Total Sales numeric
1	1	Alcohol	155.95
2	2	Produce	35.00
3	3	Meat	202.25
4	4	Seafood	200.00
5	5	Bakery	92.50
6	6	Dairy	32.50

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):

```
-- Item Margin Report
CREATE OR REPLACE VIEW ItemMargin AS
    SELECT sale.UPC,
           sum(PriceUSD - CostUSD) AS "Item Margin"
    FROM DailySales sale
    GROUP BY sale.UPC
    ORDER BY "Item Margin" DESC;

-- Daily UPC Sales Report
CREATE OR REPLACE VIEW DailySalesUPC AS
    SELECT sale.UPC,
           sale.Date as "Sales Date",
           SUM(sale.QtySold) as "Total Quantity Sold"
    FROM DailySales sale
        WHERE sale.QtySold > 0 AND
              EXTRACT(MONTH FROM current_date) =
              EXTRACT(MONTH FROM sale.date)
    GROUP BY sale.UPC, sale.Date
    ORDER BY sale.UPC;
```

Sample Data Output(s):

	upc character(12)	Item Margin numeric
1	209247942868	14.00
2	000000000003	12.00
3	000000000004	11.60
4	000000000001	11.30
5	000000000002	11.20
6	209243666568	10.00
7	939660019450	10.00
8	000000000005	8.00
9	230060019590	6.00
10	000000000006	6.00
11	002348900023	5.50
12	000000004013	3.00
13	002008000899	2.50
14	000000004023	2.50
15	000000004011	2.00
16	000000444423	2.00
17	002008000888	1.50
18	000000432397	1.00

	upc character(12)	Sales Date date	Total Quantity Sold bigint
1	000000000001	2014-04-15	1
2	000000000001	2014-04-16	14
3	000000000002	2014-04-16	2
4	000000000003	2014-04-16	10
5	000000004011	2014-04-16	20
6	000000004013	2014-04-15	15
7	000000432397	2014-04-15	25
8	000000444423	2014-04-15	30
9	000000444423	2014-04-16	50
10	002008000888	2014-04-16	10
11	002008000899	2014-04-16	20
12	002348900023	2014-04-15	4
13	002348900023	2014-04-16	15
14	209243666568	2014-04-15	5
15	209247942868	2014-04-15	15
16	209247942868	2014-04-16	10
17	230060019590	2014-04-15	15
18	230060019590	2014-04-16	10
19	939660019450	2014-04-15	5
20	939660019450	2014-04-16	10

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):

```
-- Item Lookup (Remove Zero Sellers) Report
CREATE OR REPLACE VIEW ItemLookup AS
    SELECT i.UPC,
           i.Name as "Item Name",
           cateye.Category,
           di.dID AS "Department",
           p.PriceUSD,
           sum(sale.totalAmtBTaxUSD) AS "Total Sales"
    FROM Item i,
         ItemPrice p,
         CategoryItems cateye,
         DepartmentItems di,
         DailySales sale
        WHERE i.UPC = p.UPC AND
              i.UPC = sale.UPC AND
              i.UPC = cateye.UPC AND
              i.UPC = di.UPC
    GROUP BY i.UPC, cateye.Category, di.dID, p.PriceUSD
        HAVING (sum(sale.totalAmtBTaxUSD) > 0.00)
    ORDER BY i.UPC ASC;
```

Sample Data Output(s):

	upc character(12)	Item Name character varying(35)	category integer	Department integer	priceusd numeric(6,2)	Total Sales numeric
1	000000000001	Dornish Sour Red Wine	1	1	8.15	84.75
2	000000000002	Dornish Sweet Red Wine	1	1	9.15	11.20
3	000000000003	Myrish Firewine	1	1	10.15	60.00
4	000000004011	Bananas	3	2	2.50	20.00
5	000000004013	Naval Oranges	3	2	2.00	15.00
6	000000432397	Black Bread	9	5	1.00	12.50
7	000000444423	Sweet Biscuits	9	5	1.50	80.00
8	002008000888	Boiled Eggs	11	6	1.00	7.50
9	002008000899	Wheel of Cheese	10	6	2.00	25.00
10	002348900023	Sausage	8	3	3.00	52.25
11	209243666568	Blackfish	7	4	7.75	25.00
12	209247942868	Crab	6	4	11.50	175.00
13	230060019590	Bacon	5	3	4.75	75.00
14	939660019450	Roasted Chicken	4	3	8.00	75.00

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):

```
-- Zero Sales UPC Report
CREATE OR REPLACE VIEW ZeroSalesUPC AS
    SELECT i.UPC,
           i.Name as "Item Name",
           cateye.Category,
           di.dID AS "Department",
           p.PriceUSD,
           sum(sale.totalAmtBTaxUSD) AS "Total Sales"
    FROM Item i,
         ItemPrice p,
         CategoryItems cateye,
         DepartmentItems di,
         DailySales sale
        WHERE i.UPC = p.UPC AND
              i.UPC = sale.UPC AND
              i.UPC = cateye.UPC AND
              i.UPC = di.UPC
    GROUP BY i.UPC, cateye.Category, di.dID, p.PriceUSD
        HAVING (sum(sale.totalAmtBTaxUSD) <= 0.00)
    ORDER BY i.UPC ASC;
```

Sample Data Output(s):

	upc character(12)	Item Name character varying(3	category integer	Department integer	priceusd numeric(6,2)	Total Sales numeric
1	000000000004	Wine of Courage	1	1	8.55	0.00
2	000000000005	Dark Ale	2	1	5.25	0.00
3	000000000006	Light Ale	2	1	5.00	0.00
4	000000004023	Red Grapes	3	2	2.00	0.00

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):

```
-- Stored Procedure/Function to check if Points
-- are correct for customers!
CREATE OR REPLACE FUNCTION PointCheck(INTEGER, refcursor) RETURNS
refcursor AS $$
DECLARE
    Customer_ID INT      := $1;
    resultset refcursor  := $2;
BEGIN
    open resultset FOR
    SELECT *
    FROM Customers c
      WHERE Customer_ID = c.cID AND
          c.PointsThisPeriod != ( c.PurchaseAmtThisPeriod / 2) AND
              c.PointsThisPeriod IS NOT NULL;
  RETURN resultset;

END
$$ LANGUAGE plpgsql;

-- SELECTING THE BAD ROW AS IDENTIFIED BY THE FUNCTION
-- Since I cannot figure out how to do select all cID
-- from the table as a check
SELECT PointCheck(1, 'results');
    FETCH ALL from results;

SELECT PointCheck(12, 'results');
    FETCH ALL from results;
```

```
-- Same, but checking PointsToDate
CREATE OR REPLACE FUNCTION PointCheckDate(INTEGER, refcursor) RETURNS
refcursor AS $$
DECLARE
    Customer_ID INT      := $1;
    resultset refcursor  := $2;
BEGIN
    open resultset FOR
    SELECT *
    FROM Customers c
      WHERE Customer_ID = c.cID AND
          c.PointsToDate != ( c.PurchaseAmtToDate / 2) AND
              c.PointsToDate IS NOT NULL;
  RETURN resultset;

END
$$ LANGUAGE plpgsql;

-- Will work for cID 1 or 12
SELECT PointCheckDate(12, 'results');
    FETCH ALL from results;

SELECT PointCheckDate(3, 'results');
    FETCH ALL from results;
```

Sample Output(s):

Running "SELECT PointCheck(1, 'results);
FETCH ALL FROM results;"

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)
1	1	15528.44	110.98	200.00	56.99

Running "SELECT PointCheck(12, 'results);
FETCH ALL FROM results;"

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)

Running "SELECT PointCheckDate(12, 'results);
FETCH ALL FROM results;"

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)
1	12	19400.00	469.76	9876.54	234.88

Running "SELECT PointCheckDate(1, 'results);
FETCH ALL FROM results;"

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)
1	1	15528.44	110.98	200.00	56.99

Running "SELECT PointCheckDate(13, 'results);
FETCH ALL FROM results;"

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)

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):

```
-- Stored Procedure to fix the identified wrong PointsToPeriod
-- row. NOTE THIS IS IN T-SQL and not postgressql since it was
-- easier for me to comprehend
CREATE OR REPLACE FUNCTION PointFixPeriod() RETURNS
    void AS '
    UPDATE Customers
    SET PointsThisPeriod = ( PurchaseAmtThisPeriod / 2)
        WHERE PointsThisPeriod != ( PurchaseAmtThisPeriod / 2) AND
              PointsThisPeriod IS NOT NULL;
' LANGUAGE SQL;
SELECT PointFixPeriod();
SELECT * FROM Customers ORDER BY cID ASC;

--Same, but for Date
-- Note it will fix cID 1 and cID 12!
CREATE OR REPLACE FUNCTION PointFixDate() RETURNS
    void AS '
    UPDATE Customers
    SET PointsToDate = ( PurchaseAmtToDate / 2)
        WHERE PointsToDate != ( PurchaseAmtToDate / 2) AND
              PointsToDate IS NOT NULL;
' LANGUAGE SQL;
SELECT PointFixDate();
SELECT * FROM Customers ORDER BY cID ASC;
```

Sample Data Output(s):

	pointfixperiod void		pointfixdate void
1		1	

Before Running Both Functions

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)
1	1	15528.44	110.98	200.00	56.99
2	4	19864.70	0.00	9932.35	0.00
3	5	13084.38	0.00	6542.19	0.00
4	6	288.44	204.44	144.22	102.22
5	8	473.72	191.84	236.86	95.92
6	9	199.72	20.44	99.86	10.22
7	12	19400.00	469.76	9876.54	234.88
8	13	10106.64	22.44	5053.32	11.22

After Running Both Functions:

	cid integer	purchaseamttodate numeric(10,2)	purchaseamtthisperiod numeric(10,2)	pointstodate numeric(8,2)	pointsthisperiod numeric(8,2)
1	1	15528.44	110.98	7764.22	55.49
2	4	19864.70	0.00	9932.35	0.00
3	5	13084.38	0.00	6542.19	0.00
4	6	288.44	204.44	144.22	102.22
5	8	473.72	191.84	236.86	95.92
6	9	199.72	20.44	99.86	10.22
7	12	19400.00	469.76	9700.00	234.88
8	13	10106.64	22.44	5053.32	11.22

Triggers:

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

```
-- Same Store Procs, but using triggers
DROP FUNCTION IF EXISTS trigPointFixPeriod();
CREATE OR REPLACE FUNCTION trigPointFixPeriod() RETURNS
    TRIGGER AS $$
BEGIN
    UPDATE Customers
    SET PointsThisPeriod = ( PurchaseAmtThisPeriod / 2)
        WHERE PointsThisPeriod != ( PurchaseAmtThisPeriod / 2) AND
              PointsThisPeriod IS NOT NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER fix_points_period
AFTER UPDATE ON Customers
FOR EACH ROW
EXECUTE PROCEDURE trigPointFixPeriod();
```

```
DROP FUNCTION IF EXISTS trigPointFixDate();
CREATE OR REPLACE FUNCTION trigPointFixDate() RETURNS
    TRIGGER AS $$
BEGIN
    UPDATE Customers
    SET PointsToDate = ( PurchaseAmtToDate / 2)
        WHERE PointsToDate != ( PurchaseAmtToDate / 2) AND
              PointsToDate IS NOT NULL;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER fix_points_date
AFTER UPDATE ON Customers
FOR EACH ROW
EXECUTE PROCEDURE trigPointFixDate();
```

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.

```
-- Security Roles
--Creating Role
CREATE ROLE DatabaseAdmin;
--Granting Access
GRANT SELECT,INSERT,UPDATE
ON ALL TABLES IN SCHEMA public
TO DatabaseAdmin;

--Creating PricingManager Role
CREATE ROLE PricingManager;
--Granting Access
GRANT SELECT,INSERT,UPDATE
ON ALL TABLES IN SCHEMA public
TO PricingManager;
--Revoking certain Pricing Manager Privileges
REVOKE ALL PRIVILEGES
ON Employees
FROM PricingManager;

--Creating NonManager Role
CREATE ROLE NonManager;
--Granting Select Only Access
GRANT SELECT
ON Item
TO NonManager;
GRANT SELECT
ON Vendors
TO NonManager;
```

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.