Fetch Assessment

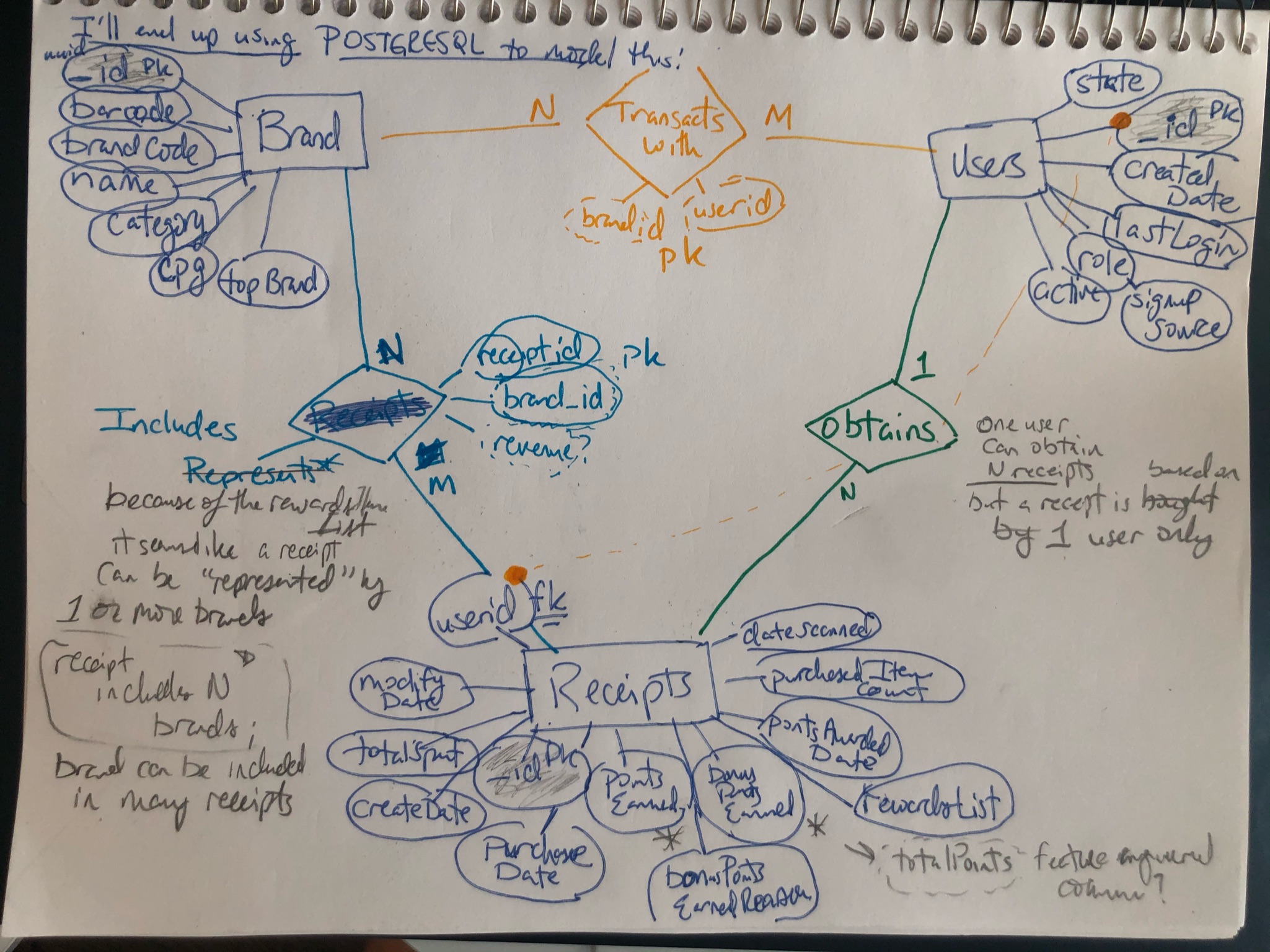
Josiah Brady

5/5/2021

#### Chose to use PostgreSQL since I use pgadmin4 on my home computer a lot

## 1) Reviewing the Unstructured Data & Diagramming a New Structured Relationship!

I think there are some additional tables that can be created because of the relationships between the users & receipts, brands & users, and receipts & brands.



ERD!

A user can obtain a bunch of receipts. Multiple users can transacts with multiple brands. Multiple receipts can include multiple brands (because of that rewardsList).

#### Setting up simple Postgresql tables that just accept json structure

CREATE TABLE receipts ( id serial NOT NULL PRIMARY KEY, info json NOT NULL );

CREATE TABLE brand ( id serial NOT NULL PRIMARY KEY, info json NOT NULL );

CREATE TABLE users ( id serial NOT NULL PRIMARY KEY, info json NOT NULL );

#### Wrote a program in order to fill these (dummy) tables with Json data quickly (C#) [this should also be included in the github submission]

using Npgsql; // package for Postgresql execution in C#, in Python I use psycopg2  
using System;  
using System.Collections.Generic;  
using System.Configuration;  
using System.Data;  
using System.Diagnostics;  
using System.IO;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace FetchAssessment  
{  
 class Program  
 {  
 static void Main(string[] args)  
 {  
 // stuff  
 string user = ConfigurationManager.AppSettings["user"];  
 string password = ConfigurationManager.AppSettings["password"];  
 string host = ConfigurationManager.AppSettings["host"];  
 string database = ConfigurationManager.AppSettings["database"];  
  
 writeToTbl(user, password, host, database, "brand", @"C:\Users\josia\source\repos\FetchAssessment\FetchAssessment\brands.json");  
 writeToTbl(user, password, host, database, "receipts", @"C:\Users\josia\source\repos\FetchAssessment\FetchAssessment\receipts.json");  
 writeToTbl(user, password, host, database, "users", @"C:\Users\josia\source\repos\FetchAssessment\FetchAssessment\users.json");  
  
 }  
  
  
 static void writeToTbl(string user, string password, string host, string database, string sqlTable, string filePath)  
 {  
 var cs = $"Host={host};Username={user};Password={password};Database={database}";  
  
 using var con = new NpgsqlConnection(cs);  
  
 con.Open();  
  
  
 if (File.Exists(filePath))  
 {  
 using (var reader = new StreamReader(filePath))  
 {  
 string[] params\_ = { };  
 while (!reader.EndOfStream)  
 {  
 string sql = $"INSERT INTO \"public\".{sqlTable} (info) VALUES (@info);";  
  
 using var cmd = new NpgsqlCommand(sql, con);  
  
 cmd.CommandText = sql;  
  
 var line = reader.ReadLine();  
  
 // parameter  
 var param = cmd.CreateParameter();  
 param.ParameterName = "info";  
 param.Value = line;  
 param.NpgsqlDbType = NpgsqlTypes.NpgsqlDbType.Json;  
 cmd.Parameters.Add(param);  
  
 // insert statement send  
 cmd.ExecuteNonQuery();  
 }  
 }  
  
 con.Close();  
 }  
  
 }  
 }

#### Refactoring the tables to be multi-column– with the PostgreSQL Json handling syntax, the way I handled data types here isn’t perfect, for example there are a lot of variables that are text types, but this was a quick way to configure what we’re looking for.

CREATE TABLE \_brand AS  
 SELECT info -> '\_id' ->> '$oid' brand\_id, --this will be the primary key  
 cast(info ->> 'barcode' as bigint) barcode,  
 info ->> 'brandCode' brandCode,  
 info ->> 'category' category,   
 info ->> 'categoryCode' categoryCode,  
 info -> 'cpg' -> '$id' ->> '$oid' cpg,  
 cast(info ->> 'topBrand' as boolean) topBrand,  
 info ->> 'name' \_name  
 FROM brand;

Handling the array elements that belong to rewardsReceiptItemList is probably the trickiest part, I wasn’t able to simply extract just the barcodes (or just the itemPrice, or finalPrice) of a receipt list into their own array object, in a way that could be easily part of a table schema.

CREATE TABLE \_receipts AS  
 SELECT   
 info -> '\_id' ->> '$oid' receipt\_id, -- primary key  
 info ->> 'bonusPointsEarned' bonusPointsEarned,  
 info ->> 'bonusPointsEarnedReason' bonusPointsEarnedReason,  
 info -> 'createDate' ->> '$date' createDate,   
 info -> 'dateScanned' ->> '$date' dateScanned,  
 info -> 'finishedDate' ->> '$date' finishedDate,  
 info -> 'modifyDate' ->> '$date' modifyDate,  
 info -> 'pointsAwardedDate' ->> '$date' pointsAwardedDate,  
 info ->> 'pointsEarned' pointsEarned,  
 info -> 'purchaseDate' ->> '$date' purchaseDate,   
 info ->> 'purchasedItemCount' purchasedItemCount,  
 info ->> 'rewardsReceiptItemList' rewardsReceiptItemList,  
 info ->> 'rewardsReceiptStatus' rewardsReceiptStatus,  
 info ->> 'totalSpent' totalSpent,  
 info ->> 'userId' user\_id -- foreign\_key because of 1 to N relationship, but it's native to the json file? so I added it as a primary key  
 FROM receipts

I think we could join the following query to the first query on the receipt\_id column to get a more granular representation of the messy data from the list objects, but I think there could be an issue with the performance of that join (especially as the database grows)… But I’m pretty sure this rewardsList disambiguation couldn’t be a part of the actual main \_receipts table construction because receipt\_id would no longer be a unique constraint [ since a receipt\_id can have multiple bought items & an item bought doesn’t have to be unique, i.e. you can buy Doritos over and over.]

select r.info -> '\_id' ->> '$oid' receipt\_id,   
elem->>'barcode' barcode,  
 elem->>'finalPrice' finalPrice,  
 elem->>'itemPrice' itemPrice,  
 elem->> 'quantityPurchased' quantityPurchased,  
 elem->> 'description' description,  
 elem->> 'needsFetchReview' needsFetchReview,  
 elem->> 'partnerItemId' partnerItemId,  
 elem->> 'pointsNotAwaredReason' pointsNotAwaredReason,  
 elem->> 'pointsPayerId' pointsPayerId,  
 elem->> 'preventTargetGapPoints' preventTargetGapPoints,  
 elem->> 'quantityPurchased' quantityPurchased,  
 elem->> 'rewardsGroup' rewardsGroup,  
 elem->> 'rewardsProductPartnerId' rewardsProductPartnerId,  
 elem->> 'targetPrice' targetPrice,  
 elem->> 'userFlaggedBarcode' userFlaggedBarcode,  
 elem->> 'userFlaggedNewItem' userFlaggedNewItem,  
 elem->> 'userFlaggedPrice' userFlaggedPrice,  
 elem->> 'userFlaggedQuantity' userFlaggedQuantity  
 from receipts r,  
 json\_array\_elements(info->'rewardsReceiptItemList') elem

Then, the user table…. there’s an incredible amount of duplication in this json file (across user\_ids, createddates, lastlogins, etc.)

CREATE TABLE \_users AS  
 SELECT DISTINCT info -> '\_id' ->> '$oid' user\_id, -- priamry key  
 info ->> 'state' state,  
 info -> 'createdDate' ->> '$date' createdDate,   
 info -> 'lastLogin' ->> '$date' lastLogin,   
 info ->> 'role' \_role,  
 info ->> 'active' active,  
 info ->> 'signUpSource' signUpSource  
 FROM users;

Relational database tables: I was a bit less confident about how I would incorporate this, especially the brand/receipt relationship.

CREATE Table \_transacts\_with (  
 brand\_id text,  
 user\_id text,  
 transaction\_count int,  
 constraint tw\_pk primary key (brand\_id, user\_id)  
 );  
   
CREATE Table \_represented (  
 brand\_id text,  
 receipt\_id text,  
 revenue double precision,  
 constraint rep\_pk primary key (brand\_id, receipt\_id)  
 );

### 2) Write a query that directly answers the following:

#### When considering average spend from receipts with ‘rewardsReceiptStatus’ of ‘Accepted’ or ‘Rejected’, which is greater?

select rewardsreceiptstatus, avg(cast(totalspent as double precision)) avgSpendIfRewards  
from "public".\_receipts  
where totalspent is not null  
group by rewardsreceiptstatus;

4 records

|  |  |
| --- | --- |
| rewardsreceiptstatus | avgspendifrewards |
| REJECTED | 23.32606 |
| FLAGGED | 180.45174 |
| PENDING | 28.03245 |
| FINISHED | 80.85430 |

I’m assuming that Finished and Accepted are equivalent here. Accepted correlates with greater spending than rejected, which is pretty intuitive.

#### Answering “When considering total number of items purchased from receipts with ‘rewardsReceiptStatus’ of ‘Accepted’ or ‘Rejected’, which is greater?” shouldn’t be too difficult from here either.

select rewardsreceiptstatus, count(cast(purchasedItemCount as int)) numItemsPurchased  
from "public".\_receipts  
where totalspent is not null  
group by rewardsreceiptstatus;

4 records

|  |  |
| --- | --- |
| rewardsreceiptstatus | numitemspurchased |
| REJECTED | 71 |
| FLAGGED | 46 |
| PENDING | 0 |
| FINISHED | 518 |

### 3) Data Quality Issues

My primary issues are with the un-intuitive datetime style & the heavily nested lists in the RewardsList of the receipts table.

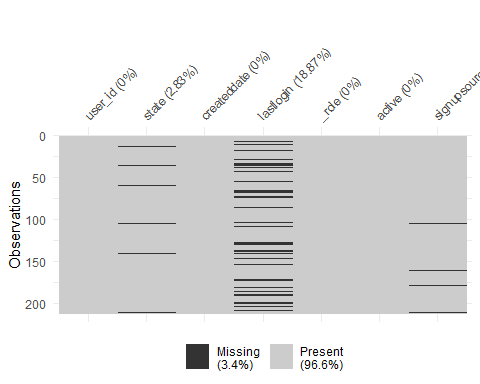
The rewards list issue can be obviated, like I eluded to early with this:

select r.info -> '\_id' ->> '$oid' receipt\_id, elem->>'barcode' barcode,  
 elem->>'finalPrice' finalPrice,  
 elem->>'itemPrice' itemPrice,  
 elem->> 'quantityPurchased' quantityPurchased,  
 elem->> 'description' description  
 from receipts r,  
 json\_array\_elements(info->'rewardsReceiptItemList') elem

But as I was saying earlier, I think it would take extra processing time to join these unnested tables with the main \_receipts table with all of the other relevant information, especially as this grows.

#### I wanted to look for null data in here since the first time I saw the json files. This could help us understand what data we like, why & perhaps what questions to ask those who are more knowledgeable about the data.

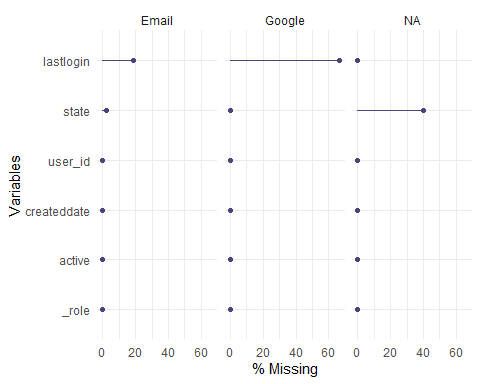
users <- dbGetQuery(db, 'SELECT \* from "public".\_users')  
  
vis\_miss(users)



For users, there seems to be no issue finding the active status, roles, created\_account dates… but last\_login information is regularly not there. A solution could be to find the median days between a user’s last login and today’s date and just impute the median to the rest of these missing values in last\_login, though.

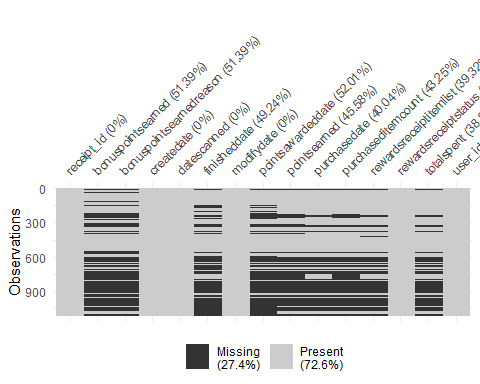
Maybe there’s an issue with tracking users that initially signed up on google…

gg\_miss\_var(users, signupsource, show\_pct = T)



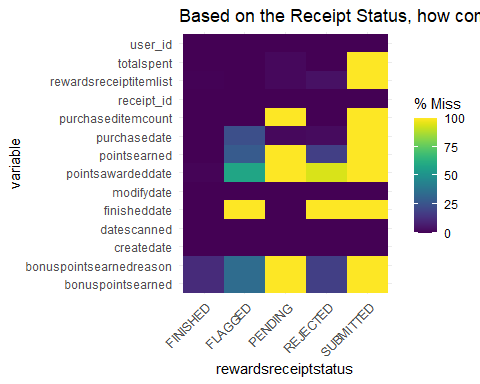
Google had a 60% missing rate for last login.

receipts <- dbGetQuery(db, 'SELECT \* from "public".\_receipts')  
  
vis\_miss(receipts)



We’re struggling to keep track of points earned (about bonus points), as well as several dates within the json data.

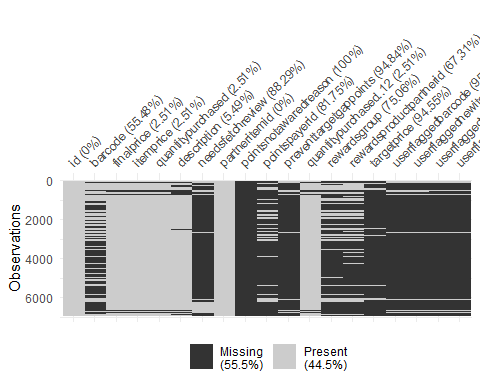
gg\_miss\_fct(x = receipts, fct = rewardsreceiptstatus) + labs(title = "Based on the Receipt Status, how complete does the data become?")



Pending & Submitted Data might be hard to account for.

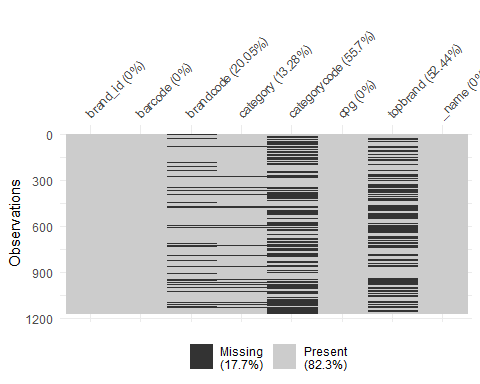
Additionally, I wanted to take a closer look at the rewardsReceiptItemList as well.

receiptList <- dbGetQuery(db, "select r.info -> '\_id' ->> '$oid' id,  
 elem->>'barcode' barcode,  
 elem->>'finalPrice' finalPrice,  
 elem->>'itemPrice' itemPrice,  
 elem->> 'quantityPurchased' quantityPurchased,  
 elem->> 'description' description,  
 elem->> 'needsFetchReview' needsFetchReview,  
 elem->> 'partnerItemId' partnerItemId,  
 elem->> 'pointsNotAwaredReason' pointsNotAwaredReason,  
 elem->> 'pointsPayerId' pointsPayerId,  
 elem->> 'preventTargetGapPoints' preventTargetGapPoints,  
 elem->> 'quantityPurchased' quantityPurchased,  
 elem->> 'rewardsGroup' rewardsGroup,  
 elem->> 'rewardsProductPartnerId' rewardsProductPartnerId,  
 elem->> 'targetPrice' targetPrice,  
 elem->> 'userFlaggedBarcode' userFlaggedBarcode,  
 elem->> 'userFlaggedNewItem' userFlaggedNewItem,  
 elem->> 'userFlaggedPrice' userFlaggedPrice,  
 elem->> 'userFlaggedQuantity' userFlaggedQuantity  
 from receipts r,  
 json\_array\_elements(info->'rewardsReceiptItemList') elem")  
  
vis\_miss(receiptList)



We may actually be able to get rid of the pointsnotawardedreason & the user flagged data.

brands <- dbGetQuery(db, 'SELECT \* from "public".\_brand')  
  
vis\_miss(brands)



Whether or not something is a top brand, I think could be something we work on re-engineering, but missing brand codes and category codes may hurt the upper analysis when examining user’s transaction data.

## 4. Email with Stakeholders to express concerns/ask questions about data.

Hello all, We’ve been able to transform json data into a relational database pretty reliably at this point, but after examining the data, we have some further questions and would love to know whether anyone can point us in the right direction. While creating the structure and importing the data, we noticed some data quality issues trying to cast certain variables and unnest some of the more convoluted json structures in the data we received.

1. How can we parse the datetime variables from the json file? 1609632000000 is an example of one date, but by what method should we handle these large integers? 1b) And is there a process that you all would recommend for handling missing dates? For example, we’ve got a great deal of columns like pointsawardeddate, finisheddate, modifydate, etc. that have not been given? Is there a specific methodology that you’d recommend for closing these gaps?
2. How can we come to know category codes? There’s a bit of a discrepancy in the amount of categories we’ve got (87%) and the codes (only 44%). Is there an independent source that we could impute codes based on?
3. Is there anyone we can contact about the higher probability of missing last\_login timestamps from users that initially signed up via Google?
4. How should we address the issues with missing spending data & datetime information for Pending & Submitted rewards receipt statuses? Should we expect that this data be backfilled by another process at a later date?