Third: Evaluate Data Quality Issues in the Data Provided

In Part 1, I discussed the process of flattening the JSON files and applying compliance rules, resulting in the data being split into four tables:

i) Users
ii) Brands
iii) Receipts
iv) Rewards_Receipts.

This document outlines the **Data Quality Checks** applied to these tables and identifies any potential issues that may impact the integrity of the data.

Data Quality Assessment – Users Table:

Overview of data quality checks for the Users table:

1) **Schema verification:** Ensuring the Schema of the table (possibly new data) be consistent with the already established schema for the Users table.
2) **Checking for Duplicates:** Ensure there are no duplicate records in the **Users** table, particularly for columns like *user_id*, which should uniquely identify each user.
3) **Verifying Uniqueness of Primary Key:** Confirm that the assigned primary key (*user_id*) is unique across all records, ensuring each user is represented only once in the table.
4) **Missing Values:** Identify any columns with missing values and calculate the percentage of missing data. This helps assess the completeness of the dataset and ensures that key fields like *user_id* and *created_date_time* are fully populated.
5) **Timestamp Validation:** Verify that the *lastLogin_date_time* field occurs after the *created_date_time*, ensuring chronological consistency in the data.
6) **Data Type Validation:** Check that all columns have the appropriate data types. For example, the *active* field should be a Boolean, and all date-related fields (e.g., *created_date_time*, *lastLogin_date_time*) should be properly formatted as date/time. Any missing dates should be filled with NaT (Not a Time) instead of NaN to avoid inconsistencies in date handling.
7) **Acceptable Range of values:** Ensuring the values that are to be entered would follow in the range of the values. This ensures flagging unwanted data.

1) Schema verification

It is expected that the table would have the following columns: *active, signUpSource, created_date_time, user_id, lastLogin_date_time, state*. From the jupyter notebook, this is verified. The test also ensures if there are any spelling mistakes in the column names to maintain consistency.

2) Checking for Duplicates

It can be found that **57.17 %** of the data entries are duplicate. This is a **Data Quality Issue** that needs to be addressed. It can be resolved by deleting the duplicate records.

```
## Checking for duplicates

users_duplicates = users_df[users_df.duplicated()]
print(f'Percentage of Duplicate user records :{round(100*users_duplicates.shape[0]/users_df.shape[0],4)}')
```

✓ 0.0s

```
Percentage of Duplicate user records :57.1717
```

3) Verifying Uniqueness of Primary Key (*user_id*)

The primary key of the table – user_id needs to be unique and not missing in the table.

```
## Checking if user_id is unique
users_df_no_duplicates['user_id'].isna().sum()
```

```
0
```

This shows that the primary key *user_id* indeed is unique and not missing in the table.

4) Missing Values:

Here is the information about the missing values in Users Table:

Some notable issues are with the following columns:

a) signUpSource
b) state
c) lastLogin_date_time

```
## percentage of missing values

users_df_no_duplicates.isna().sum()*100/users_df_no_duplicates.shape[0]
```

```
active                 0.000000
role                   0.000000
signUpSource           2.358491
state                  2.830189
user_id                0.000000
created_date_time      0.000000
lastLogin_date_time    18.867925
dtype: float64
```

Analyzing the missing values in *SignupSource* and *state*:

```
## Analyzing missing values in 'signUpSource'
users_df_no_duplicates[users_df_no_duplicates['signUpSource'].isna()]
```

	active	role	signUpSource	state	user_id	created_date_time	lastLogin_date_time
388	True	consumer	NaN	WI	55308179e4b0eabd8f99caa2	2015-04-16 22:43:53.186	2018-05-07 12:23:40.003
395	True	fetch-staff	NaN	WI	59c124bae4b0299e55b0f330	2017-09-19 09:07:54.302	2021-02-08 10:42:58.117
422	True	consumer	NaN	NaN	5a43c08fe4b014fd6b6a0612	2017-12-27 09:47:27.059	2021-02-12 10:22:37.155
462	True	fetch-staff	NaN	IL	5964eb07e4b03efd0c0f267b	2017-07-11 10:13:11.771	2021-03-04 13:07:49.770
475	True	fetch-staff	NaN	NaN	54943462e4b07e684157a532	2014-12-19 08:21:22.381	2021-03-05 10:52:23.204

```
## Unique values of signUpSource
users_df_no_duplicates['signUpSource'].unique()
```

```
array(['Email', 'Google', nan], dtype=object)
```

3 out of 6 *signUpSource* missing values occur for the user who is a fetch-staff (Mild Data Quality Issue). The unique values for signup source are Email and Google. It is very much possible that fetch-staff can sign up using an **internal source** which is not email or google. The resources list could be expanded including possible valid sources such as **App Store**, **Referral**, and other relevant platforms, to guarantee that this field is complete for all users. If the field is still missing, it could be imputed as 'others'.

State missing values could be imputed with 'DNF' (Did not fill) or 'N/A' or 'None' appropriately (Mild Data Quality Issue).

Analyzing the missing values in *lastLogin_date_time*:

The missing last login data indicates that users are not immediately logging in after signing up. This suggests a potential bottleneck in the sign-up process, where users may be unable to log in right after signing up. One possible cause could be permission issues or other technical restrictions that prevent the app from opening or proceeding to the login page after email registration. Further investigation is needed to identify and address this issue to streamline the sign-up and login experience, especially for email users.

I'm curious to understand if the users who never logged in, made a purchase. One would expect that it would not.

```
## Checking if the user_id present in the missing last login date if they made any orders.
missing_last_login_merge_receipts = missing_last_login\
    .merge(receipts_df, how = 'inner',left_on = 'user_id',right_on = 'userId')\
    [['user_id','created_date_time_x','lastLogin_date_time',\
      'bonusPointsEarned','receipt_id','scanned_date_time']]
missing_last_login_merge_receipts.info()
```

✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 57 entries, 0 to 56
Data columns (total 6 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   user_id              57 non-null     object 
 1   created_date_time_x  57 non-null     object 
 2   lastLogin_date_time  0 non-null      object 
 3   bonusPointsEarned    32 non-null     float64
 4   receipt_id           57 non-null     object 
 5   scanned_date_time    57 non-null     object 
```

Data Quality Issue spotted

From the above figure,

It is interesting to see that the data suggests that **57** users with no recorded last login date are still able to scan receipts and earn bonus points. This is unexpected behavior, as we would typically expect a last login date to be present once a user interacts with the app to scan receipts. This anomaly needs to be investigated further, as it could indicate a logging issue or a flaw in the system where login events are not properly captured before allowing user actions such as receipt scanning.

5) Timestamp validation

It is expected that the last login date and time would always be after the creation of the account/user.

```
##Checking if last login date is after creation date

users_df_no_duplicates[users_df_no_duplicates['lastLogin_date_time'] \
                       < users_df_no_duplicates['created_date_time']].shape
```

✓ 0.0s

```
(0, 8)
```

The above figure validates that the last login datetime is always after creation of the account.

6) Appropriate Data formatting

Columns are expected to have the correct data types, such as *datetime* for date fields instead of *object* types. It is also important to ensure consistency in object-type columns to avoid

discrepancies like ‘Google’ vs. ‘GOOGLE’.

```
## checking for discrepancies of the 'object' columns
for columns in ['active','role','signUpSource','state']:
    print(users_df_no_duplicates[columns].unique())
```

✓ 0.0s

```
[ True False]
['consumer' 'fetch-staff']
['Email' 'Google' nan]
['WI' 'KY' 'AL' 'CO' 'IL' nan 'OH' 'SC' 'NH']
```

- No object discrepancies or errors were found.
- Datetime columns were successfully converted from object to datetime types.
- Boolean columns were verified to ensure they contain valid boolean values.

7) Range type checks

The *state* column was checked to ensure all entries are valid U.S. state abbreviations, adhering to the official two-letter codes for U.S. states. The test results from the Jupyter notebook confirm that this check passed successfully.

For the date columns, validations were performed to ensure that the values fall within a developer-defined period (Example: Year 2000 to 2100). This step is essential to prevent unreasonable or erroneous dates. The notebook results show that the date range checks also passed without issues.

```
Created counts by month
 2014-12      1
2015-04      1
2017-07      1
2017-09      1
2017-12      1
2020-01      1
2020-07      1
2020-11      4
2020-12      1
2021-01    170
2021-02     30
```

Fig: 3.

However, while reviewing the distribution of the *created_date_time* variable, a concerning pattern was observed. There are significant gaps in the data, with entire years such as 2016, 2018, and 2019 missing. This sporadic distribution of user creation dates seems unusual and could indicate a **Data Quality Issue**. Further investigation is recommended to understand the cause of these gaps.

Data Quality Assessment – Brands Table:

This section outlines the data quality checks conducted on the **Brands** table, following a structure like that used for the **Users** table. A new additional check has been added (which discusses relationships between the columns). These checks are designed to ensure data consistency, integrity, and overall quality.

1) Schema Verification: The schema of the **Brands** table was validated to ensure consistency with the established schema, ensuring that any new data conforms to the expected structure. This ensures that the **Brands** table is aligned with other related tables, such as the **Users** table, and prevents schema drift.
2) Duplicate Records Check: The table was examined for any duplicate records, as duplicate entries can distort analysis and lead to inaccurate insights. The data passed this check, ensuring that all records in the **Brands** table are unique.
3) Primary Key Uniqueness: To maintain data integrity, it is critical that the primary key (*brand_id*) remains unique across all records. The **Brands** was validated for primary key uniqueness, confirming that no duplicate *brand_id* exist.
4) One-to-One Relationship checks: To maintain data consistency, it is important to take an educated guess if the columns are maintaining one-to-one relationship and check if there is possible explanation of inconsistency if exists. This check explores possible human errors or data entry errors to some level that are discussed below.
5) Missing Values Analysis: A detailed analysis was conducted to identify any missing values across columns. While missing values are expected in some cases, understanding which fields are incomplete can help guide future imputation strategies or data collection improvements.
6) Data Type Validation: Each column in the **Brands** table was assessed to ensure it is stored in the correct data type. For instance, categorical fields like *name* and *brandCode* should be stored as strings, while fields like *topBrand* should be Boolean. Ensuring appropriate data types prevents downstream errors in analysis and reporting.

Schema Validation, Duplicates Records Check, Primary Key Uniqueness Check:

```
##brand_df columns
brands_df.columns
```

✓ 0.0s

```
Index(['barcode', 'category', 'categoryCode', 'name', 'topBrand', 'brand_id',
       'cpg_id', 'cpg_ref', 'brandCode'],
      dtype='object')
```

```python
## Checking for duplicate entries in brands table
if brands_df[brands_df.duplicated()].shape[0] == 0:
    print('No Duplicate entries in Brands table')
else:
    print('Duplicate entries in Brands table')

## Checking if brand_id is unique
if brands_df['brand_id'].is_unique:
    print('brand_id is unique')
else:
    print('brand_id is not unique')

## Checking if there are any missing values in brands_id
if brands_df['brand_id'].isna().sum() == 0:
    print('brand_id is not missing')
else:
    print('brand_id is missing')
```

✓ 0.0s

```
No Duplicate entries in Brands table
brand_id is unique
brand_id is not missing
```

From the above figures, it can be shown that Data Quality in terms of Schema Validation, Primary Key Uniqueness, Duplicate records are verified.

One-to-One Relationship Checks:

I'll start with an example. It's reasonable to make an educated guess that brand name (*name*) and brand code (*brandCode*) should maintain a one-to-one relationship. If they don't, and the variance is significant, the initial assumption may be incorrect. However, if the variance is minimal, it might indicate small errors in data entry. The notebook code below explores this further:

```
##checking if brand name and brand code have one-one to relationship (which is expected)

inconsistent = brands_df.groupby('name')['brandCode'].nunique().reset_index()
inconsistent = inconsistent[inconsistent['brandCode'] > 1]
inconsistent
```

✓ 0.0s

	name	brandCode
73	Baken-Ets	2
129	Caleb's Kola	2
223	Dippin Dots® Cereal	2
313	Health Magazine	2
335	I CAN'T BELIEVE IT'S NOT BUTTER!	2
504	ONE A DAY® WOMENS	2
564	Pull-Ups	2

The table above shows the variance I mentioned, and it appears to be quite small. For instance, the brand name 'Baken-Ets' has two unique brand codes. Let's dive into possible reasons for this occurrence by analyzing some of the brand names highlighted in the table:

```
## Deep analysis of why the name and brandCode are inconsistent

print(brands_df[brands_df['name'] == 'Baken-Ets'][['name','brandCode']])
print(brands_df[brands_df['name'] == 'Caleb\'s Kola'][['name','brandCode']])
print(brands_df[brands_df['name'] == 'Dippin Dots® Cereal'][['name','brandCode']])
print(brands_df[brands_df['name'] == 'I CAN\'T BELIEVE IT\'S NOT BUTTER!'][['name','brandCode']])
```

✓ 0.0s

```
         name  brandCode
574  Baken-Ets  BAKEN ETS
848  Baken-Ets  BAKEN-ETS
             name     brandCode
140  Caleb's Kola  CALEB'S KOLA
740  Caleb's Kola   CALEBS KOLA
                    name           brandCode
1081  Dippin Dots® Cereal         DIPPIN DOTS
1163  Dippin Dots® Cereal  DIPPIN DOTS CEREAL
                                name                          brandCode
176  I CAN'T BELIEVE IT'S NOT BUTTER!  I CAN'T BELIEVE IT'S NOT BUTTER!
846  I CAN'T BELIEVE IT'S NOT BUTTER!   I CAN'T BELIEVE IT'S NOT BUTTER
```

This is a **Data Quality Issue.** The spelling and punctuation mistakes in the brandCode highlight the inconsistency in the relationship between brandCode and brand name. This data quality issue needs to be addressed, as brandCode is a crucial parameter for identifying a brand. It is a serious problem when trying to understand and analyze brand related KPIs. Additionally, brandCode serves as a foreign key in the **Rewards_Receipts** table, as discussed in the first part of this project. To ensure data integrity, it is essential to have a consistent, error-free one-to-one mapping between brandCode and brand name.

The same approach can be applied to other column pairs where one might reasonably assume a one-to-one relationship. This has been checked for the following columns: ['name', 'brandCode', 'topBrand', 'brand_id']

```
## Automation code to check if there are any one-one relatipnship between non primary keys in a dataframe.

df = brands_df # one of the four tables: Brands, Users, Receipts and Rewards_Receipts

columns_might_have_one_one_relationship = ['name','brandCode','topBrand','brand_id']

for columni in columns_might_have_one_one_relationship:
    for columnj in columns_might_have_one_one_relationship:

        if columni != columnj:
            inconsistent = df.groupby(columni)[columnj].nunique().reset_index()
            inconsistent = inconsistent[inconsistent[columnj] > 1]
            if  not inconsistent.empty:
                print(f'one-one relationship does not exist between {columni} and {columnj}')
```

✓ 0.0s

```
one-one relationship does not exist between name and brandCode
one-one relationship does not exist between name and topBrand
one-one relationship does not exist between name and brand_id
one-one relationship does not exist between brandCode and name
one-one relationship does not exist between brandCode and topBrand
one-one relationship does not exist between brandCode and brand_id
one-one relationship does not exist between topBrand and name
one-one relationship does not exist between topBrand and brandCode
one-one relationship does not exist between topBrand and brand_id
```

It could be unnecessary to go into full detail here, but a couple of inconsistencies are highlighted. for example:

```
## Deep analysis of why the name and topbrand are inconsistent

print(brands_df[brands_df['name'] == 'Huggies'][['name','topBrand']])

print(brands_df[brands_df['name'] == 'Pull-Ups'][['name','topBrand']])
```

✓ 0.0s

```
        name topBrand
628   Huggies    False
1074  Huggies     True
        name topBrand
126  Pull-Ups    False
978  Pull-Ups     True
```

This shows that some of the brands like ‘Huggies’ are flagged as both ‘top brand’ and ‘not a top brand’ which is a **Data Quality Issue**. This raises the need to reconsider how *topBrand* is being assigned to each brand.

Missing Value Checks:

```
## Missing value percentages in brands_df
brands_df.isna().sum()*100/brands_df.shape[0]
```

✓ 0.0s

```
barcode          0.000000
category        13.281919
categoryCode    55.698372
name             0.000000
topBrand        52.442159
brand_id         0.000000
cpg_id           0.000000
cpg_ref          0.000000
brandCode       23.050557
```

13% of the category data is missing, which is a **Data Quality Issue**. Can the category field be imputed based on the brand name? Maybe, but there are cases like the brand 'Nike', which sells both shoes and apparel, making the category ambiguous. So, imputing the category just by brand name isn't straightforward.

Exploring methods like automating data entry with reference tables that map barcode or brandCode to specific *category* and *categoryCode* could be a viable solution.

We could verify if the missing *categoryCode* is consistent across the *category* column which are present.

```
## Understanding why categoryCode is missing given category.
## Later checking if missing categoryCode values are missing for all the respective categories

categoryCode_missing_brands = brands_df[(~brands_df['category'].isna()) & (brands_df['categoryCode'].isna())]\
    [['category','categoryCode']].drop_duplicates().sort_values(by='category')
categoryCode_missing_brands

categoryCode_present_brands = brands_df[(~brands_df['category'].isna()) & (~brands_df['categoryCode'].isna())]\
    [['category','categoryCode']].drop_duplicates().sort_values(by='category')

categoryCode_missing_brands.merge(categoryCode_present_brands, \
                                  how = 'left', suffixes=('_missing', '_present'),\
                                  on = 'category').sort_values(by = 'categoryCode_present',ascending=False).head(9)
```

0.0s

category	categoryCode_missing	categoryCode_present
Personal Care	NaN	PERSONAL_CARE
Magazines	NaN	MAGAZINES
Health & Wellness	NaN	HEALTHY_AND_WELLNESS
Grocery	NaN	GROCERY
Frozen	NaN	FROZEN
Beverages	NaN	BEVERAGES
Beer Wine Spirits	NaN	BEER_WINE_SPIRITS
Baking	NaN	BAKING

The table above shows that for a given category (e.g., 'Grocery'), some *categoryCode* values are missing, while others are correctly filled with 'GROCERY'.

Data Quality Issue has been spotted. The table above reveals that for certain categories, the *categoryCode* is missing in some instances, while it is properly filled for the same category in other records. This inconsistency highlights a data quality issue where *categoryCode* has not been filled in appropriately, despite having clear information on how it should be completed.

A similar logic could be applied for brandCode information given the brand name. Firstly, 23% of the brandCodes are missing. This is a huge **Data Quality issue** as *brandCode* is going to play a key role in connecting tables (connecting **Rewards_Receipts** table in this project). The missing values need to be imputed appropriately. Coming back to the logic, here is the table that shows if the missing brandCodes is consistent everywhere given a brand name.

brandCode_missing	name	brandCode_present
NaN	V8 Hydrate	V8 HYDRATE
NaN	Sierra Mist	SIERRA MIST
NaN	Diabetic Living Magazine	511111805298

the brandCode is missing in some instances, while it is properly filled for the same brand in other records. This inconsistency highlights a data quality issue where brandCode has not been filled appropriately, despite having clear information on how it should be completed.

Missing top_brand information can be flagged as a mild Data Quality Issue. A third category, such as 'undecided,' could be introduced to handle these cases.

Consistent Data Format Checks:

1) TopBrand Column needs to be Boolean. In the notebook, *topBrand* column is converted to Boolean type.
2) Barcodes need to follow a consistent format (For example, GS1 barcodes should be 12 digits)

```
## Verifying if the barcode is in consistent format -> 12 digit
if brands_df[brands_df['barcode'].apply(lambda x: len(str(x))) != 12].empty:
    print('All the barcodes are in consistent format -> 12 digits')
else:
    print('Barcodes is not in consistent format')
    print(brands_df[brands_df['barcode'].apply(lambda x: len(str(x))) != 12]['barcode'])
```

✓ 0.0s

```
All the barcodes are in consistent format -> 12 digits
```

Data Quality Assessment – Receipts table:

This section outlines the data quality checks conducted on the **Receipts** table, following a structure like that used for the **Users** and **Brands** table. These checks are designed to ensure data consistency, integrity, and overall quality.

Standard Data Quality Checks (Similar to Users and Brands tables):

The **Receipts** table underwent key data quality checks, following a similar structure used for the **Users** and **Brands** tables. This included schema verification, duplicate detection, primary key uniqueness validation, assessment of missing values, and verifying if the dates variables would make sense. Data types were reviewed to confirm proper formatting for key fields, such as *receipt_id* and date columns.

Additional Checks:

1) **Chronology of Date variables:** Given date variables like modified, purchased, and scanned, it's generally expected that events occur in a chronological order.
2) **Foreign Key *userId* Integrity check:** It is important to ensure that the *user_id* column, which links the **Receipts** and **Users** tables, contains only *user_id* values present in the **Users** table. This check guarantees proper data linkage between the two tables.
3) **Range of the Numerical Data checks:** The data can be analyzed to assess the distribution, and a deeper investigation can be carried out if anything unusual is flagged.

Standard Data Quality Checks:

From the code given in the Jupyter notebook, it can be verified that the check is passed.

```python
## Checking for duplicate entries in receipts table
if receipts_df[receipts_df.duplicated()].shape[0] == 0:
    print('No Duplicate entries in Receipts table')
else:
    print('Duplicate entries in Brands table')
## Checking if receipt_id is unique
if receipts_df['receipt_id'].is_unique:
    print('receipt_id is unique')
else:
    print('receipt_id is not unique')
## Checking if there are any missing values in brands_id
if receipts_df['receipt_id'].isna().sum() == 0:
    print('receipt_id is not missing')
else:
    print('receipt_id is missing')
```

✓ 0.0s

```
No Duplicate entries in Receipts table
receipt_id is unique
receipt_id is not missing
```

Missing Value Checks:

Here is the Missing Values percentage for the **Receipts** table:

bonusPointsEarned	51.385165
bonusPointsEarnedReason	51.385165
pointsEarned	45.576408
purchasedItemCount	43.252904
rewardsReceiptItemList	39.320822
rewardsReceiptStatus	0.000000
totalSpent	38.873995
userId	0.000000
receipt_id	0.000000
created_date_time	0.000000
scanned_date_time	0.000000
finished_date_time	49.240393
modify_date_time	0.000000
pointsAwarded_date_time	52.010724
purchased_date_time	40.035746

Understanding the reason for missing points related variables:

In the table below, all missing points statuses are either submitted, pending, flagged, or rejected (essentially not finished), which makes sense since the receipt might be awaiting points or a decision not to reward, and hence the missing values.

rewardsReceiptStatus	Missing_Percentage
SUBMITTED	85.098039
PENDING	9.803922
FLAGGED	2.549020
REJECTED	2.549020

```
✓ ## Unique values of the Status ···
```

```
array(['FINISHED', 'REJECTED', 'FLAGGED', 'SUBMITTED', 'PENDING']
```

Given the above information, I would not flag the missing points related values, or the values could be imputed to 0.

Understanding the reason for missing items list:

The output snippet of missing values shows that apart from Receipt ID, Rewards Status, User ID, and the created-scanned-modified date and time, most other fields are significantly missing. This suggests that while a registered user was able to create and scan a receipt, no contents were captured. It could be that the receipt scanning algorithm is not capturing the contents of some receipts. This could be due to illegible receipt content or technical issues with the computer vision scanning algorithm. While I wouldn't flag this as a major data quality issue, I would store this information and use it to notify users that their receipts couldn't be scanned, prompting them to try again later.

```
## Understanding how the missing values distributed across other columns when items are missing
missing_items.info()
```
✓ 0.0s

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 440 entries, 71 to 1118
Data columns (total 15 columns):
 #   Column                  Non-Null Count  Dtype  
---  ------                  --------------  -----  
 0   bonusPointsEarned       2 non-null      float64
 1   bonusPointsEarnedReason 2 non-null      object 
 2   pointsEarned            2 non-null      float64
 3   purchasedItemCount      5 non-null      float64
 4   rewardsReceiptItemList  0 non-null      object 
 5   rewardsReceiptStatus    440 non-null    object 
 6   totalSpent              5 non-null      float64
 7   userId                  440 non-null    object 
 8   receipt_id              440 non-null    object 
 9   created_date_time       440 non-null    object 
 10  scanned_date_time       440 non-null    object 
 11  finished_date_time      3 non-null      object 
 12  modify_date_time        440 non-null    object 
 13  pointsAwarded_date_time 2 non-null      object 
 14  purchased_date_time     3 non-null      object
```

Chronology of Date variables check:

Here is the rough chronological order the process of scanning a receipt would follow: first, the items are purchased (1) -> then the instance is created (2) -> followed by the contents being scanned (3)-> then finished (4)-> and finally modified (5). Here is the code that checks the order:

```
# Check if chronology is maintained across multiple columns
chronology_issues = receipts_df[
    (receipts_df['purchased_date_time'] > receipts_df['created_date_time']) |
    (receipts_df['created_date_time'] > receipts_df['scanned_date_time']) |
    (receipts_df['scanned_date_time'] > receipts_df['finished_date_time'])
]
# Display the Data percentage where chronology is not maintained
print(f'Percentage of the Data not following the above mentioned chronological order \
= {round(100*chronology_issues.shape[0]/receipts_df.shape[0],4)}')
```
✓ 0.0s

```
Percentage of the Data not following the above mentioned chronological order = 1.1618
```

1.6% of the data violates the discussed chronology. Since the proportion is less, I would flag this as Mild Data Quality Issue. But I would deep dive into understanding if there is an appropriate reason for the anomaly.

Foreign Key Integrity Check (userId):

Since userId is the foreign key, it is expected that the userId is a subset of the **Users** table. However, from the code snippet below, it shows that **45.3%** of the User ids are not present in the **Users** table. This is a serious **Data Quality Issue**. This suggests that the **Users** table is not being refreshed in a timely manner, and there is a need to ensure consistency.

```
## checking for presence of the users present in the receipts table but not in the Users table
user_id_in_receipts_df = list(receipts_df['userId'].unique())
user_id_in_users_df = list(users_df_no_duplicates['user_id'].unique())

missing_user_id = [x for x in user_id_in_receipts_df if x not in user_id_in_users_df]
percentage_of_users_not_in_users_table = 100*len(missing_user_id)/len(user_id_in_receipts_df)

print(f'Percentage of the Users not present in the Users table = {round(percentage_of_users_not_in_users_table,4)}%')
```

✓ 0.0s

```
Percentage of the Users not present in the Users table = 45.3488%
```

Range of Numerical Data Check:

A preliminary check could be applied to the numerical values to understand the basic distribution of the variables. Here is the snippet of the output.

```
## Description of numerical variables in receipts table
receipts_df.describe()
```

✓ 0.0s

	bonusPointsEarned	pointsEarned	purchasedItemCount	totalSpent
count	544.000000	609.000000	635.00000	684.000000
mean	238.893382	585.962890	14.75748	77.796857
std	299.091731	1357.166947	61.13424	347.110349
min	5.000000	0.000000	0.00000	0.000000
25%	5.000000	5.000000	1.00000	1.000000
50%	45.000000	150.000000	2.00000	18.200000
75%	500.000000	750.000000	5.00000	34.960000
max	750.000000	10199.800000	689.00000	4721.950000

From the table above, we can see that the median is much smaller than the average, indicating a right-skewed distribution. This suggests the presence of outliers in the data. While outliers don't necessarily signal a major data quality issue, they could be flagged as a Mild Data Quality Concern. These outliers should be monitored closely for any signs of fraudulent scans or unusual patterns, but they don't indicate a serious data quality problem on their own.

Range of Date variables check:

A date range from **2000-01-01** to **2100-12-31** was selected, and a check was implemented to ensure all date variables fall within this range. The output from the notebook confirms the success of this check.

```
Check passed for created_date_time
Check passed for scanned_date_time
Check passed for finished_date_time
Check passed for modify_date_time
Check passed for pointsAwarded_date_time
Check passed for purchased_date_time
```

Data Quality Assessment - Rewards_Receipts table:

This section outlines the data quality checks conducted on the **Rewards_Receipts** table, following a structure like that used for the previous tables. These checks are designed to ensure data consistency, integrity, and overall quality.

Standard Data Quality Checks (Similar to Users and Brands tables):

The **Rewards_Receipts** table underwent key data quality checks, following a similar structure used for the **Users** and **Brands** tables. This included schema verification, duplicate detection, primary key uniqueness validation, assessment of missing values, and verifying if the dates variables would make sense. Data types were reviewed to confirm proper formatting for key fields.

Additional Checks:

1) **Missing Foreign Key values and a possible solution:** The variables related to the brand are **brandCode** and **barcode**, but there are many missing values. A potential method for imputing missing brandCode is discussed.
2) **Foreign Key *userId* Integrity check:** It is important to ensure that the *brandCode* column, which links the **Rewards_Receipts** and **Brands** tables, contains only *brandCode* values present in the **Brands** table. This check guarantees proper data linkage between the two tables.
3) **Data Format for Barcodes check:** The data can be analyzed to assess the distribution, and a deeper investigation can be carried out if anything unusual is flagged.

Standard Data Quality Checks:

Here is the code and output snippet that conducted the standard data quality checks. The checks are a clean pass.

```python
## Checking for duplicate entries in rewards_receipts table
if rewards_receipts_df[rewards_receipts_df.duplicated()].shape[0] == 0:
    print('No Duplicate entries in Rewards_Receipts table')
else:
    print('Duplicate entries in rewards_receipts table')
## Checking if receipt_id is unique
if rewards_receipts_df[['receipt_id','partnerItemId']].drop_duplicates().shape[0] == rewards_receipts_df.shape[0]
    print('receipt_id and partnerItemId pair is unique')
else:
    print('receipt_id and partnerItemId pair is not unique')
## Checking if there are any missing values in brands_id
if rewards_receipts_df[['receipt_id','partnerItemId']].isna().sum().sum() == 0:
    print('receipt_id and partnerItemId pair is not missing')
else:
    print('receipt_id and partnerItemId pair is missing')
```

✓ 0.0s

```
No Duplicate entries in Rewards_Receipts table
receipt_id and partnerItemId pair is unique
receipt_id and partnerItemId pair is not missing
```

Missing Value Checks:

A significant portion of the data is missing, with some occurring in groups, such as the *'userflagged_'* variables, which have many missing values. A reasonable assumption is that these missing values correspond to instances where the user did not flag anything. Similarly, missing values in *pointsNotAwardedReason* might indicate that points were awarded. Another similar group is *OriginalMetaBrite*. It should be reviewed whether it's valuable to retain such largely missing data, depending on the business use case. Further investigation is needed to determine the reason for these missing values. Since the foreign key used is *brandCode*, we can dig deeper to explore potential improvements.

Missing brandCode and Imputation techniques:

Out of many variables present in the table, only two of the columns seemed to connect the **Brands** table: *brandCode* and *barcode*. I have reservations about barcode being the connector since barcode is specific to a product not a brand. We also observed that there are many missing values in the 2 variables. We can do a small comparison to check which column has the most number of intersections between **Rewards_Receipts** table and **Brand** table. From the figure below (FILL FIG NUMBER), even though the intersection number is less, brandCode has most intersections. It reinforces my idea of choosing brandCode to be the foreign key. Can this number be improved? I propose a union of two imputation techniques:

1) The description field often contains brand information, particularly in the first word. Extracting that could serve as a potential imputation method for missing *brandCode* in the rewards_receipts table.

2) Impute missing *brandCode* in the brands table using the brand's name.

```python
## checking for presence of the barcodes present in the rewards_receipts table and in the brands table
def convert_barcode(barcode):
    try:
        return int(barcode)
    except:
        return barcode

barcode_in_rewards_receipts_df = list(rewards_receipts_df['barcode'].apply(convert_barcode).unique())
barcode_in_brands_df = list(brands_df['barcode'].unique())

barcode_present = [x for x in barcode_in_rewards_receipts_df if x in barcode_in_brands_df]
print(f'Number of the barcodes present in both the tables = {len(barcode_present)}')
```

✓ 0.5s

```
Number of the barcodes present in both the tables = 16
```

```python
## checking for presence of the brandCodes present in the rewards_receipts table and in the brands table
brandCode_in_rewards_receipts_df = list(rewards_receipts_df['brandCode'].unique())
brandCode_in_brands_df = list(brands_df['brandCode'].unique())

brandCode_present = [x for x in brandCode_in_rewards_receipts_df if x in brandCode_in_brands_df]
print(f'Number of the brandCodes present in both the tables = {len(brandCode_present)}')
```

✓ 0.0s

```
Number of the brandCodes present in both the tables = 42
```

Low intersection numbers suggests that there is **Data Quality Issue.**

My observations pertaining to the first word of the description for missing brandCode values:

```
## Rationale for the above chosen imputer for missing brandCode
rewards_receipts_df[rewards_receipts_df['brandCode'].isna()][['description_first_word','description']].drop_duplicates().sample(5)
```

✓ 0.0s

	description_first_word	description
3732	USDA	USDA Choice Boneless Beef Loin Top Sirloin Steak
120	KRAFT	KRAFT BACK TO NATURE CHEDDAR CHEESE SHELL MACA...
4881	SIG	SIG ICE CRM MOOSE
758	LUIGES	LUIGES DELUXE P1ZZA
648	KLARBRUNN	KLARBRUNN 12PK 12 SL OZ

description_first_word	count
ITEM	173
PC	138
KLARBRUNN	128
HUGGIES	93
MILLER	90
HYV	84
COMP	74
OSCAR	73
LEGO	56
PLAY	55
EMIL'	55
GREEN	51
FLIPBELT	50
SIG	47
THINDUST	44
MUELLER	44

Because of the above imputation method, brands like KLARBRUNN (128 entries), HUGGIES (93 entries), MUELLER (44 entries) etc. are coming into picture. Of course, the imputation is not perfect (for example: ITEM NOT FOUND would become ITEM). Therefore, when performing brand related queries, one needs to keep in mind about this imputation method.

Similarly, we can see in brands table, how Brand name be used to impute brandCode value:

name_first_word	name
KICKSTART	Kickstart
TEST BRAND @1601939064752	test brand @1601939064752
QUAKER	Quaker Rice Cakes
TEST BRAND @1595968944654	test brand @1595968944654
BUDWEISER	Budweiser

Comparing if there are any improvements in terms of intersections:

```
## checking for presence of the brandCodes present in the rewards_receipts table and in the brands table
brandCode_in_rewards_receipts_df = list(rewards_copy['brandCode'].unique())
brandCode_in_brands_df = list(brands_copy_df['brandCode'].unique())

present_brandCode_new = [x for x in brandCode_in_rewards_receipts_df if x in brandCode_in_brands_df]
print(f'Number of the brandCodes present in both the Brands table after new imputation technique = {len(present_brandCode_new)}')
```

✓ 0.0s

```
Number of the brandCodes present in both the Brands table after new imputation technique = 88
```

```
## checking for presence of the brandCodes present in the rewards_receipts table and in the brands table
brandCode_in_rewards_receipts_df = list(rewards_receipts_df['brandCode'].unique())
brandCode_in_brands_df = list(brands_df['brandCode'].unique())

brandCode_present = [x for x in brandCode_in_rewards_receipts_df if x in brandCode_in_brands_df]
print(f'Number of the brandCodes present in both the tables before new imputation technique = {len(brandCode_present)}')
```

✓ 0.0s

```
Number of the brandCodes present in both the tables before new imputation technique = 42
```

There has been an increase of over 200% due to the new imputation techniques. But is this the best approach? Not necessarily, though it is better than leaving the brandCodes missing. Other potential methods to explore include extracting brand information directly from the barcodes, especially if there is a pattern, such as the first four digits indicating something relevant. An efficient way of pairing barcodes with brand information is quite necessary. The proper approach would be to map the barcodes to their respective brands, store that information, and then use it to accurately extract brand details.

Foreign Key Integrity Check:

Let's understand about the brandCodes which are present in rewards_receipts table but not present in the Brands table, verifying foreign key integrity. Note that the follow test is done before the proposed imputation.

Here is a sample list of Brand Codes that are present in **Rewards_Receipts** table but not in **Brands** table.

```
##checking if unique brandcodes exist in receipt but not in brands

brandcodes_rewards_receipt = list(rewards_receipts_df['brandCode'].unique())
brandcodes_brands = list(brands_df['brandCode'].unique())
len(brandcodes_rewards_receipt)
missing_bar_codes = [x for x in brandcodes_rewards_receipt if x not in brandcodes_brands]
sorted(missing_bar_codes)[:8]
```

✓ 0.0s

```
['7UP',
 'ADVIL',
 'AMERICAN BEAUTY',
 'ARROWHEAD',
 'AZTECA',
 'BANZA',
 'BEAR CREEK COUNTRY KITCHENS',
 'BEN AND JERRYS']
```

Are the Brand Codes actually missing?
Answer: No.

After some initial eyeball checks, a few observations were made:

1) **'7UP'** in the rewards_receipts table and **'7 up'** in brands_df refer to the same brand but lack consistency in structure.
2) **"BEN AND JERRYS"** in **Rewards_Receipts** table and **"BEN & JERRY'S"** in **Brands** table are also inconsistent in spelling and punctuation.
3) These are just a few examples of errors, whether in spelling or punctuation, which impact the consistency of brandCodes.

Given that brandCode is used as a foreign key, this poses a significant **Data Quality Issue.** Creating a data pipeline to directly replace eyeball checks for grammatical errors would be challenging. The core issue, as mentioned earlier, lies in the lack of a proper mapping between barcode and brand information. Establishing this mapping would greatly improve data quality.

Boolean format consistency check:

There are some variables in the **Rewards_Receipts** table which are expected to be *Boolean*. Here is the list:
['needsFetchReview','preventTargetGapPoints','userFlaggedNewItem','competitiveProduct','deleted']

To ensure the best Data Quality, these columns are tested to see if they are 'Boolean' type. If not, they are converted to 'Boolean'.

Barcode format Consistency check:

Given the presence of barcodes, it is essential that they follow a consistent format:
1) numeric
2) 12 digits long.

From the code from Jupyter notebook, it can be shown that the datatype is an object which is a **Data Quality Issue.**

Also the 12 digit consistency has failed as well.

```
## Verifying if the barcode is in consistent format -> 12 digit
if rewards_receipt_df_bc_present[rewards_receipt_df_bc_present['barcode'].apply(lambda x: len(str(x))) != 12].empty:
    print('All the barcodes are in consistent format -> 12 digits')
else:
    print('Barcodes is not in consistent format \n Examples:\n')
    print(rewards_receipt_df_bc_present[rewards_receipt_df_bc_present['barcode'].apply(lambda x: len(str(x))) != 12]['barcode'].sample(5))
```

✓ 0.0s

```
Barcodes is not in consistent format
 Examples:

6866    B07BRRLSVC
1619          4069
1204          4060
419           1234
373           4011
```

Some of the barcodes contain only 4 digits or 11 digits. Some of the barcodes contain letters which violate the check and shows **Data Quality Issue.**

Automating the Data Quality Checks:

What we discussed were some important data quality checks. It's useful to write Python pandas code in a Jupyter notebook and carefully perform these checks one by one. However, in practice, especially in high-volume data environments, manually writing checks like this might not be scalable or efficient. To address this, building automated data quality pipelines becomes essential. These pipelines ensure that incoming data is automatically validated, and any errors or anomalies are flagged for further investigation. Tools like **Great Expectations** provide robust frameworks for defining, validating, and documenting data quality rules, while **Airflow DAGs** can orchestrate these pipelines, ensuring checks run on schedule or in response to new data.

Here is a sample Great Expectation code to check the uniqueness of the Primary key *user_id* in **Users** table. The expectation that is used is **"ge.expectations.ExpectColumnValuesToBeUnique"**

```
## Unique test:

context = ge.get_context()
data_source = context.data_sources.add_pandas("pandas")
data_asset = data_source.add_dataframe_asset(name="pd dataframe asset")

batch_definition = data_asset.add_batch_definition_whole_dataframe("batch definition")
batch = batch_definition.get_batch(batch_parameters={"dataframe": users_df_no_duplicates})

#expect user_id to exist:
unique_user_expectation = ge.expectations.ExpectColumnValuesToBeUnique(column = 'user_id')
validation_result = batch.validate(unique_user_expectation)
print(validation_result)
```

✓ 0.1s

Calculating Metrics: 100% 10/10 [00:00<00:00, 159.85it/s]


```
{
  "success": true,
  "expectation_config": {
    "kwargs": {
      "batch_id": "pandas-pd dataframe asset",
      "column": "user_id"
    },
    "type": "expect_column_values_to_be_unique",
    "meta": {}
```


Here is a list of other useful ge.expectations

"ExpectColumnDistinctValuesToBeInSet"; "ExpectColumnValuesToNotBeNull"; "ExpectColumnDistinctValuesToBeInSet"; "ExpectColumnMaxToBeBetween" etc.

Data Engineering practices for Best Data Quality:

One of the best Data engineering practices out there to ensure the best data quality for downstream, a technique called **WAP (Write-Audit-Publish)** is widely used. As the name suggests, it consists of three key phases: Write, Audit and Push. **Write** is where raw data is ingested into a staging area or storage system. **Audit** is where the data is thoroughly checked for completeness, accuracy, and consistency, using automated checks or validation rules. This is very similar to what I discussed in this document. It could be either by using custom scripts or tools like **great expectations**. Publish is where the clean, verified data is made available for downstream analysis or reporting.

Adopting the **WAP** framework ensures data quality is rigorously maintained by incorporating an Audit phase between ingestion and publishing, catching issues like missing values or inconsistencies that we discussed before they affect downstream processes. This approach reduces the risk of errors and builds trust in the data's accuracy, helping organizations make informed, data-driven decisions. Automating these checks, using **GE** or **Apache Airflow DAGs**, provides ongoing validation, making **WAP** especially effective in high-volume or complex environments.