Updated Fetch Takehome_Jiatong Song (1)-Copy1

October 21, 2024

0.1 Fetch Customer Behavior and Transaction Analysis (June 12, 2024 – September 8, 2024)

Fetch is a rewards app where users earn points by scanning grocery receipts.

Assumptions:

1 1. Explore the Data

[46]:
```
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt

# Load CSV files into dataframes
users_df = pd.read_csv(r'C:\Users\kathy\Downloads\USER_TAKEHOME.csv')
products_df = pd.read_csv(r'C:\Users\kathy\Downloads\PRODUCTS_TAKEHOME.csv')
transactions_df = pd.read_csv(r'C:\Users\kathy\Downloads\TRANSACTION_TAKEHOME.
↪csv')

# Inspect the data
print("Users DataFrame Overview:")
users_df.head()
```

```
Users DataFrame Overview:
```

[46]:
```
                         ID               CREATED_DATE  \
0  5ef3b4f17053ab141787697d  2020-06-24 20:17:54.000 Z
1  5ff220d383fcfc12622b96bc  2021-01-03 19:53:55.000 Z
2  6477950aa55bb77a0e27ee10  2023-05-31 18:42:18.000 Z
3  658a306e99b40f103b63ccf8  2023-12-26 01:46:22.000 Z
4  653cf5d6a225ea102b7ecdc2  2023-10-28 11:51:50.000 Z

                  BIRTH_DATE STATE LANGUAGE  GENDER
0  2000-08-11 00:00:00.000 Z    CA   es-419  female
1  2001-09-24 04:00:00.000 Z    PA       en  female
2  1994-10-28 00:00:00.000 Z    FL   es-419  female
3                        NaN    NC       en     NaN
```

```
4  1972-03-19 00:00:00.000 Z    PA       en  female
```

[47]:
```
pip install nbconvert
```

```
Requirement already satisfied: nbconvert in c:\users\kathy\anaconda3\lib\site-
packages (7.10.0)
Requirement already satisfied: beautifulsoup4 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (4.12.2)
Requirement already satisfied: bleach!=5.0.0 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (4.1.0)
Requirement already satisfied: defusedxml in c:\users\kathy\anaconda3\lib\site-
packages (from nbconvert) (0.7.1)
Requirement already satisfied: jinja2>=3.0 in c:\users\kathy\anaconda3\lib\site-
packages (from nbconvert) (3.1.3)
Requirement already satisfied: jupyter-core>=4.7 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (5.5.0)
Requirement already satisfied: jupyterlab-pygments in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (0.1.2)
Requirement already satisfied: markupsafe>=2.0 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (2.1.3)
Requirement already satisfied: mistune<4,>=2.0.3 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (2.0.4)
Requirement already satisfied: nbclient>=0.5.0 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (0.8.0)
Requirement already satisfied: nbformat>=5.7 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (5.9.2)
Requirement already satisfied: packaging in c:\users\kathy\anaconda3\lib\site-
packages (from nbconvert) (23.1)
Requirement already satisfied: pandocfilters>=1.4.1 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (1.5.0)
Requirement already satisfied: pygments>=2.4.1 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (2.15.1)
Requirement already satisfied: tinycss2 in c:\users\kathy\anaconda3\lib\site-
packages (from nbconvert) (1.2.1)
Requirement already satisfied: traitlets>=5.1 in
c:\users\kathy\anaconda3\lib\site-packages (from nbconvert) (5.7.1)
Requirement already satisfied: six>=1.9.0 in c:\users\kathy\anaconda3\lib\site-
packages (from bleach!=5.0.0->nbconvert) (1.16.0)
Requirement already satisfied: webencodings in
c:\users\kathy\anaconda3\lib\site-packages (from bleach!=5.0.0->nbconvert)
(0.5.1)
Requirement already satisfied: platformdirs>=2.5 in
c:\users\kathy\anaconda3\lib\site-packages (from jupyter-core>=4.7->nbconvert)
(3.10.0)
Requirement already satisfied: pywin32>=300 in
c:\users\kathy\anaconda3\lib\site-packages (from jupyter-core>=4.7->nbconvert)
(305.1)
Requirement already satisfied: jupyter-client>=6.1.12 in
```

```
c:\users\kathy\anaconda3\lib\site-packages (from nbclient>=0.5.0->nbconvert)
(8.6.0)
Requirement already satisfied: fastjsonschema in
c:\users\kathy\anaconda3\lib\site-packages (from nbformat>=5.7->nbconvert)
(2.16.2)
Requirement already satisfied: jsonschema>=2.6 in
c:\users\kathy\anaconda3\lib\site-packages (from nbformat>=5.7->nbconvert)
(4.19.2)
Requirement already satisfied: soupsieve>1.2 in
c:\users\kathy\anaconda3\lib\site-packages (from beautifulsoup4->nbconvert)
(2.5)
Requirement already satisfied: attrs>=22.2.0 in
c:\users\kathy\anaconda3\lib\site-packages (from
jsonschema>=2.6->nbformat>=5.7->nbconvert) (23.1.0)
Requirement already satisfied: jsonschema-specifications>=2023.03.6 in
c:\users\kathy\anaconda3\lib\site-packages (from
jsonschema>=2.6->nbformat>=5.7->nbconvert) (2023.7.1)
Requirement already satisfied: referencing>=0.28.4 in
c:\users\kathy\anaconda3\lib\site-packages (from
jsonschema>=2.6->nbformat>=5.7->nbconvert) (0.30.2)
Requirement already satisfied: rpds-py>=0.7.1 in
c:\users\kathy\anaconda3\lib\site-packages (from
jsonschema>=2.6->nbformat>=5.7->nbconvert) (0.10.6)
Requirement already satisfied: python-dateutil>=2.8.2 in
c:\users\kathy\anaconda3\lib\site-packages (from jupyter-
client>=6.1.12->nbclient>=0.5.0->nbconvert) (2.8.2)
Requirement already satisfied: pyzmq>=23.0 in c:\users\kathy\anaconda3\lib\site-
packages (from jupyter-client>=6.1.12->nbclient>=0.5.0->nbconvert) (25.1.2)
Requirement already satisfied: tornado>=6.2 in
c:\users\kathy\anaconda3\lib\site-packages (from jupyter-
client>=6.1.12->nbclient>=0.5.0->nbconvert) (6.3.3)
Note: you may need to restart the kernel to use updated packages.
```

```
DEPRECATION: Loading egg at c:\users\kathy\anaconda3\lib\site-
packages\huggingface_hub-0.24.0rc0-py3.8.egg is deprecated. pip 24.3 will
enforce this behaviour change. A possible replacement is to use pip for package
installation.. Discussion can be found at
https://github.com/pypa/pip/issues/12330
```

[48]:
```python
print("\nProducts DataFrame Overview:")
products_df.head()
```

```
Products DataFrame Overview:
```

[48]:

	CATEGORY_1	CATEGORY_2	CATEGORY_3 \
0	Health & Wellness	Sexual Health	Conductivity Gels & Lotions
1	Snacks	Puffed Snacks	Cheese Curls & Puffs

```
2  Health & Wellness               Hair Care        Hair Care Accessories
3  Health & Wellness               Oral Care                   Toothpaste
4  Health & Wellness  Medicines & Treatments               Essential Oils

  CATEGORY_4                                       MANUFACTURER  \
0        NaN                                                NaN
1        NaN                                                NaN
2        NaN                           PLACEHOLDER MANUFACTURER
3        NaN                                  COLGATE-PALMOLIVE
4        NaN  MAPLE HOLISTICS AND HONEYDEW PRODUCTS INTERCHA…

             BRAND       BARCODE
0              NaN  7.964944e+11
1              NaN  2.327801e+10
2          ELECSOP  4.618178e+11
3          COLGATE  3.500047e+10
4  MAPLE HOLISTICS  8.068109e+11
```

[49]:
```python
print("\nTransactions DataFrame Overview:")
transactions_df.head()
```

```
Transactions DataFrame Overview:
```

[49]:
```
                             RECEIPT_ID PURCHASE_DATE  \
0  0000d256-4041-4a3e-adc4-5623fb6e0c99    2024-08-21
1  0001455d-7a92-4a7b-a1d2-c747af1c8fd3    2024-07-20
2  00017e0a-7851-42fb-bfab-0baa96e23586    2024-08-18
3  000239aa-3478-453d-801e-66a82e39c8af    2024-06-18
4  00026b4c-dfe8-49dd-b026-4c2f0fd5c6a1    2024-07-04

                 SCAN_DATE STORE_NAME                   USER_ID  \
0  2024-08-21 14:19:06.539 Z    WALMART  63b73a7f3d310dceeabd4758
1  2024-07-20 09:50:24.206 Z       ALDI  62c08877baa38d1a1f6c211a
2  2024-08-19 15:38:56.813 Z    WALMART  60842f207ac8b7729e472020
3  2024-06-19 11:03:37.468 Z  FOOD LION  63fcd7cea4f8442c3386b589
4  2024-07-05 15:56:43.549 Z   RANDALLS  6193231ae9b3d75037b0f928

       BARCODE FINAL_QUANTITY FINAL_SALE
0  1.530001e+10           1.00
1           NaN           zero       1.49
2  7.874223e+10           1.00
3  7.833997e+11           zero       3.49
4  4.790050e+10           1.00
```

1.0.1 a. Check Data Types

[50]:
```
users_df.info()
products_df.info()
transactions_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 100000 entries, 0 to 99999
Data columns (total 6 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   ID            100000 non-null  object
 1   CREATED_DATE  100000 non-null  object
 2   BIRTH_DATE    96325 non-null   object
 3   STATE         95188 non-null   object
 4   LANGUAGE      69492 non-null   object
 5   GENDER        94108 non-null   object
dtypes: object(6)
memory usage: 4.6+ MB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 845552 entries, 0 to 845551
Data columns (total 7 columns):
 #   Column        Non-Null Count   Dtype
---  ------        --------------   -----
 0   CATEGORY_1    845441 non-null  object
 1   CATEGORY_2    844128 non-null  object
 2   CATEGORY_3    784986 non-null  object
 3   CATEGORY_4    67459 non-null   object
 4   MANUFACTURER  619078 non-null  object
 5   BRAND         619080 non-null  object
 6   BARCODE       841527 non-null  float64
dtypes: float64(1), object(6)
memory usage: 45.2+ MB
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 50000 entries, 0 to 49999
Data columns (total 8 columns):
 #   Column          Non-Null Count  Dtype
---  ------          --------------  -----
 0   RECEIPT_ID      50000 non-null  object
 1   PURCHASE_DATE   50000 non-null  object
 2   SCAN_DATE       50000 non-null  object
 3   STORE_NAME      50000 non-null  object
 4   USER_ID         50000 non-null  object
 5   BARCODE         44238 non-null  float64
 6   FINAL_QUANTITY  50000 non-null  object
 7   FINAL_SALE      50000 non-null  object
dtypes: float64(1), object(7)
memory usage: 3.1+ MB
```

1.0.2 Convert Data Types

[51]:
```
# Convert date columns to datetime
users_df['CREATED_DATE'] = pd.to_datetime(users_df['CREATED_DATE'],␣
 ↪errors='coerce')
users_df['BIRTH_DATE'] = pd.to_datetime(users_df['BIRTH_DATE'], errors='coerce')

transactions_df['PURCHASE_DATE'] = pd.
 ↪to_datetime(transactions_df['PURCHASE_DATE'], errors='coerce')
transactions_df['SCAN_DATE'] = pd.to_datetime(transactions_df['SCAN_DATE'],␣
 ↪errors='coerce')

# Convert ID fields to string (object) type
users_df['ID'] = users_df['ID'].astype(str)
transactions_df['USER_ID'] = transactions_df['USER_ID'].astype(str)
transactions_df['RECEIPT_ID'] = transactions_df['RECEIPT_ID'].astype(str)
products_df['BARCODE'] = products_df['BARCODE'].astype(str)
transactions_df['BARCODE'] = transactions_df['BARCODE'].astype(str)

# Convert categorical columns to 'category' datatype
users_df['STATE'] = users_df['STATE'].astype('category')
users_df['LANGUAGE'] = users_df['LANGUAGE'].astype('category')
users_df['GENDER'] = users_df['GENDER'].astype('category')

products_df['CATEGORY_1'] = products_df['CATEGORY_1'].astype('category')
products_df['CATEGORY_2'] = products_df['CATEGORY_2'].astype('category')
products_df['CATEGORY_3'] = products_df['CATEGORY_3'].astype('category')
products_df['CATEGORY_4'] = products_df['CATEGORY_4'].astype('category')
products_df['MANUFACTURER'] = products_df['MANUFACTURER'].astype('category')
products_df['BRAND'] = products_df['BRAND'].astype('category')

# Convert FINAL_QUANTITY and FINAL_SALE in transactions_df to numeric
transactions_df['FINAL_QUANTITY'] = pd.
 ↪to_numeric(transactions_df['FINAL_QUANTITY'], errors='coerce')
transactions_df['FINAL_SALE'] = pd.to_numeric(transactions_df['FINAL_SALE'],␣
 ↪errors='coerce')

# Checking the final datatypes after conversion
print(users_df.dtypes)
print(products_df.dtypes)
print(transactions_df.dtypes)
```

```
ID                           object
CREATED_DATE    datetime64[ns, UTC]
BIRTH_DATE      datetime64[ns, UTC]
STATE                      category
LANGUAGE                   category
GENDER                     category
```

```
dtype: object
CATEGORY_1      category
CATEGORY_2      category
CATEGORY_3      category
CATEGORY_4      category
MANUFACTURER    category
BRAND           category
BARCODE           object
dtype: object
RECEIPT_ID                       object
PURCHASE_DATE            datetime64[ns]
SCAN_DATE           datetime64[ns, UTC]
STORE_NAME                       object
USER_ID                          object
BARCODE                          object
FINAL_QUANTITY                  float64
FINAL_SALE                      float64
dtype: object
```

1.0.3 b. Check for Missing Value

Users Table: Major missing data is in the **LANGUAGE** and GENDER columns.

Products Table: **CATEGORY_4** has significant missing data, while CATEGORY_3, MANUFACTURER, and BRAND have some missing data.

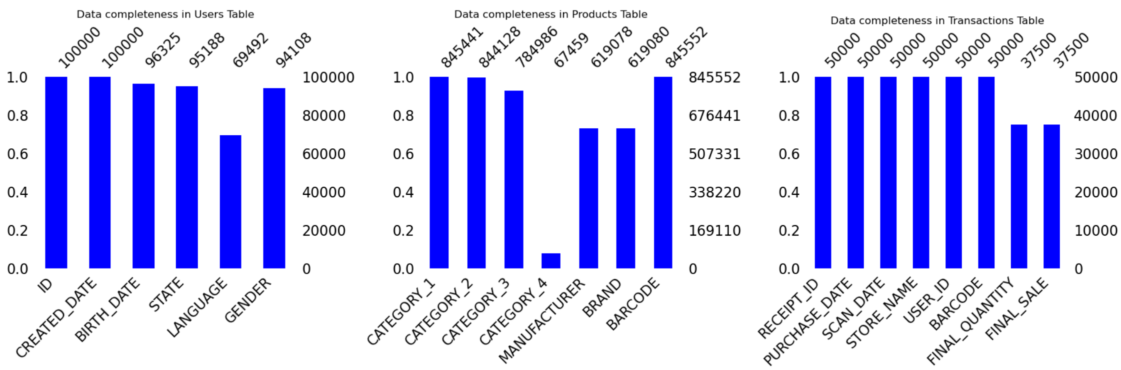
Transactions Table: Fields are complete, but there are many missing values in both **FINAL_QUANTITY** and **FINAL_SALE**.

[52]:
```
!pip install missingno
import missingno as msno
null_df = pd.DataFrame({
    "Users Null Values": users_df.isnull().sum(),
    "Users Percentage Null": (users_df.isnull().sum() / users_df.shape[0]) *␣
 ↪100,
    "Products Null Values": products_df.isnull().sum(),
    "Products Percentage Null": (products_df.isnull().sum() / products_df.
 ↪shape[0]) * 100,
     "Transactions Null Values": transactions_df.isnull().sum(),
    "Transactions Percentage Null": (transactions_df.isnull().sum() /␣
 ↪products_df.shape[0]) * 100
})

fig, axs = plt.subplots(1, 3, figsize=(18, 6))

# Plot the missing data bar chart for users_df
msno.bar(users_df, ax=axs[0], color="blue")
axs[0].set_title("Data completeness in Users Table")
```

```python
# Plot the missing data bar chart for products_df
msno.bar(products_df, ax=axs[1], color="blue")
axs[1].set_title("Data completeness in Products Table")

# Plot the missing data bar chart for transactions_df
msno.bar(transactions_df, ax=axs[2], color="blue")
axs[2].set_title("Data completeness in Transactions Table")

# Adjust layout to prevent overlap
plt.tight_layout()
plt.show()
```

```
Requirement already satisfied: missingno in c:\users\kathy\anaconda3\lib\site-
packages (0.5.2)
Requirement already satisfied: numpy in c:\users\kathy\anaconda3\lib\site-
packages (from missingno) (1.26.4)
Requirement already satisfied: matplotlib in c:\users\kathy\anaconda3\lib\site-
packages (from missingno) (3.8.0)
Requirement already satisfied: scipy in c:\users\kathy\anaconda3\lib\site-
packages (from missingno) (1.11.4)
Requirement already satisfied: seaborn in c:\users\kathy\anaconda3\lib\site-
packages (from missingno) (0.12.2)
Requirement already satisfied: contourpy>=1.0.1 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (1.2.0)
Requirement already satisfied: cycler>=0.10 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (0.11.0)
Requirement already satisfied: fonttools>=4.22.0 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (4.25.0)
Requirement already satisfied: kiwisolver>=1.0.1 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (1.4.4)
Requirement already satisfied: packaging>=20.0 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (23.1)
Requirement already satisfied: pillow>=6.2.0 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (10.2.0)
Requirement already satisfied: pyparsing>=2.3.1 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (3.0.9)
Requirement already satisfied: python-dateutil>=2.7 in
c:\users\kathy\anaconda3\lib\site-packages (from matplotlib->missingno) (2.8.2)
Requirement already satisfied: pandas>=0.25 in
c:\users\kathy\anaconda3\lib\site-packages (from seaborn->missingno) (2.1.4)
Requirement already satisfied: pytz>=2020.1 in
c:\users\kathy\anaconda3\lib\site-packages (from
pandas>=0.25->seaborn->missingno) (2023.3.post1)
Requirement already satisfied: tzdata>=2022.1 in
c:\users\kathy\anaconda3\lib\site-packages (from
pandas>=0.25->seaborn->missingno) (2023.3)
Requirement already satisfied: six>=1.5 in c:\users\kathy\anaconda3\lib\site-
packages (from python-dateutil>=2.7->matplotlib->missingno) (1.16.0)
```

```
DEPRECATION: Loading egg at c:\users\kathy\anaconda3\lib\site-
packages\huggingface_hub-0.24.0rc0-py3.8.egg is deprecated. pip 24.3 will
enforce this behaviour change. A possible replacement is to use pip for package
installation.. Discussion can be found at
https://github.com/pypa/pip/issues/12330
```



[53]:
```python
print("\nMissing Values in Users DataFrame:")
print(users_df.isnull().sum())
print("\nMissing Values in Products DataFrame:")
print(products_df.isnull().sum())
print("\nMissing Values in Transactions DataFrame:")
print(transactions_df.isnull().sum())
```

```
Missing Values in Users DataFrame:
ID                  0
CREATED_DATE        0
BIRTH_DATE       3675
STATE            4812
LANGUAGE        30508
GENDER           5892
dtype: int64

Missing Values in Products DataFrame:
CATEGORY_1          111
CATEGORY_2         1424
CATEGORY_3        60566
CATEGORY_4       778093
MANUFACTURER     226474
BRAND            226472
BARCODE               0
dtype: int64

Missing Values in Transactions DataFrame:
```

```
RECEIPT_ID            0
PURCHASE_DATE         0
SCAN_DATE             0
STORE_NAME            0
USER_ID               0
BARCODE               0
FINAL_QUANTITY    12500
FINAL_SALE        12500
dtype: int64
```

1.0.4 3. Handle Data Quality Issues & Duplicate Entries

3.1 Transactions table: duplicate entries for the same RECEIPT_ID appears twice with the same PURCHASE_DATE, SCAN_DATE, and USER_ID, but only one of the duplicated rows has complete quantity and sale information. The other duplicated row contains missing values. This indicates data entry errors.

[54]:
```python
# Show duplicates based on specific columns
df_sorted = transactions_df.sort_values(by=['RECEIPT_ID'])
duplicate_entries = df_sorted[df_sorted.duplicated(subset=['RECEIPT_ID',␣
 ↪'PURCHASE_DATE', 'STORE_NAME', 'USER_ID', 'BARCODE'], keep=False)]
duplicate_entries
```

[54]:
```
                                 RECEIPT_ID PURCHASE_DATE  \
0      0000d256-4041-4a3e-adc4-5623fb6e0c99    2024-08-21
41567  0000d256-4041-4a3e-adc4-5623fb6e0c99    2024-08-21
1      0001455d-7a92-4a7b-a1d2-c747af1c8fd3    2024-07-20
39291  0001455d-7a92-4a7b-a1d2-c747af1c8fd3    2024-07-20
2      00017e0a-7851-42fb-bfab-0baa96e23586    2024-08-18
…                                       …           …
28152  fffbb112-3cc5-47c2-b014-08db2f87e0c7    2024-07-30
24998  fffbfb2a-7c1f-41c9-a5da-628fa7fcc746    2024-07-28
31602  fffbfb2a-7c1f-41c9-a5da-628fa7fcc746    2024-07-28
25233  fffe8012-7dcf-4d84-b6c6-feaacab5074a    2024-09-07
24999  fffe8012-7dcf-4d84-b6c6-feaacab5074a    2024-09-07

                              SCAN_DATE STORE_NAME                   USER_ID  \
0      2024-08-21 14:19:06.539000+00:00    WALMART  63b73a7f3d310dceeabd4758
41567  2024-08-21 14:19:06.539000+00:00    WALMART  63b73a7f3d310dceeabd4758
1      2024-07-20 09:50:24.206000+00:00       ALDI  62c08877baa38d1a1f6c211a
39291  2024-07-20 09:50:24.206000+00:00       ALDI  62c08877baa38d1a1f6c211a
2      2024-08-19 15:38:56.813000+00:00    WALMART  60842f207ac8b7729e472020
…                                     …          …                         …
28152  2024-08-04 11:43:31.474000+00:00    WALMART  5eb59d6be7012d13941af5e2
24998  2024-07-28 11:47:34.180000+00:00    WALMART  62a0c8f7d966665570351bb8
31602  2024-07-28 11:47:34.180000+00:00    WALMART  62a0c8f7d966665570351bb8
25233  2024-09-08 08:21:25.648000+00:00  WALGREENS  5f53c62bd683c715b9991b20
24999  2024-09-08 08:21:25.648000+00:00  WALGREENS  5f53c62bd683c715b9991b20
```

```
              BARCODE  FINAL_QUANTITY  FINAL_SALE
0       15300014978.0             1.0         NaN
41567   15300014978.0             1.0        1.54
1                 nan             NaN        1.49
39291             nan             1.0        1.49
2       78742229751.0             1.0         NaN
…                 …               …          …
28152  818000020115.0             1.0        4.88
24998   13000009546.0             1.0         NaN
31602   13000009546.0             1.0        3.48
25233   74323095777.0             2.0        2.98
24999   74323095777.0             NaN        2.98

[50000 rows x 8 columns]
```

[55]:
```
unique_rows = transactions_df.drop_duplicates(subset=['RECEIPT_ID',␣
↪'PURCHASE_DATE', 'STORE_NAME', 'USER_ID', 'BARCODE'])

num_unique_rows = unique_rows.shape[0]

print(f"Number of unique rows based on specific columns: {num_unique_rows}")
```

```
Number of unique rows based on specific columns: 24795
```

Clean duplicate rows in the transaction dataset by replacing invalid or non-numeric values, ensuring that only valid entries with non-zero quantities and sales are kept.

[56]:
```
# Step 1: Identify rows with duplicates based on specific columns
duplicate_rows = transactions_df[transactions_df.
↪duplicated(subset=['RECEIPT_ID', 'PURCHASE_DATE', 'SCAN_DATE', 'STORE_NAME',␣
↪'USER_ID', 'BARCODE'], keep=False)]

# Step 2: Replace 'zero' with NaN for easier handling of non-numeric data
duplicate_rows['FINAL_QUANTITY'].replace('zero', pd.NA, inplace=True)

# Step 3: Convert FINAL_QUANTITY and FINAL_SALE to numeric values, coercing␣
↪errors
duplicate_rows['FINAL_QUANTITY'] = pd.
↪to_numeric(duplicate_rows['FINAL_QUANTITY'], errors='coerce')
duplicate_rows['FINAL_SALE'] = pd.to_numeric(duplicate_rows['FINAL_SALE'],␣
↪errors='coerce')

# Step 4: Keep only the rows where both FINAL_QUANTITY and FINAL_SALE are valid␣
↪(not zero or NaN)
valid_rows = duplicate_rows.dropna(subset=['FINAL_QUANTITY', 'FINAL_SALE'])
valid_rows = valid_rows[(valid_rows['FINAL_QUANTITY'] > 0) &␣
↪(valid_rows['FINAL_SALE'] > 0)]
```

```
# Step 5: Drop the duplicate entries from the original DataFrame and add the
↪cleaned valid rows
transactions_df = pd.concat([transactions_df.drop(duplicate_rows.index),
↪valid_rows])

# Display the cleaned data
transactions_df.head(10)
```

[56]:

```
                                 RECEIPT_ID PURCHASE_DATE  \
25000  7b3ec72d-9d30-40b8-b185-0bfb638942a9    2024-08-20
25001  04869b68-29e3-4e8d-9bdb-950046fc3473    2024-08-05
25002  f1a96308-24a5-46a8-8d8c-285cf9dce1ba    2024-09-03
25003  7ee1798e-fd2e-4278-838b-f417fdcafe08    2024-08-30
25004  21feab39-49f2-42e9-ae69-10371e2fc0a9    2024-08-23
25005  30977cbc-1d29-4f2d-851c-1104432769d0    2024-09-01
25006  48c7720b-7097-4cee-995e-721e52c623bd    2024-06-25
25007  d542a912-30a7-4f73-89a8-365f8de17409    2024-08-12
25008  c70b5591-92a5-4d9f-8d82-5525cf91cfaf    2024-06-20
25009  21a0945c-09ec-4b76-92e4-f2e590062470    2024-07-29

                             SCAN_DATE           STORE_NAME  \
25000 2024-08-20 11:17:29.633000+00:00  DOLLAR GENERAL STORE
25001 2024-08-09 16:06:00.570000+00:00  DOLLAR GENERAL STORE
25002 2024-09-03 11:28:25.264000+00:00               WALMART
25003 2024-09-04 12:53:31.478000+00:00  DOLLAR GENERAL STORE
25004 2024-08-27 10:45:00.125000+00:00                TARGET
25005 2024-09-01 09:40:16.103000+00:00               WALMART
25006 2024-06-25 17:56:43.654000+00:00                COSTCO
25007 2024-08-15 18:34:31.745000+00:00             FOOD LION
25008 2024-06-21 11:32:23.957000+00:00               WALMART
25009 2024-08-02 13:27:25.284000+00:00         MARKET BASKET

                        USER_ID         BARCODE  FINAL_QUANTITY  FINAL_SALE
25000  60fc1e6deb7585430ff52ee7  745527114884.0             1.0        1.65
25001  654cf234a225ea102b81072e  745527114884.0             1.0        1.65
25002  63c1cb6d3d310dceeac55487   37000828761.0             1.0       28.22
25003  65c29b137050d0a6206cd24f   12000504051.0             1.0        5.25
25004  61a58ac49c135b462ccddd1c   24000393429.0             1.0        2.59
25005  5baf733455206419c416c3be   37000779704.0             1.0        2.20
25006  65c5b9a416cc39173210ae15       9697867.0             1.0        9.69
25007  6567a084bc6a13d85a5cf0dd  752798149286.0             2.0        1.58
25008  62f069014e73e2db30ecab93   17000132556.0             1.0        8.76
25009  6318f67ab2906b770ead6e92   70200504318.0             1.0        3.49
```

[57]:

```
num_columns = transactions_df.shape[0]
```

```python
print(f'After removing duplicates, the number of rows for transactions table is:
 ↪ {num_columns}')
```

```
After removing duplicates, the number of rows for transactions table is: 24679
```

[58]:
```python
print("\nMissing Values in Transactions DataFrame:")
print(transactions_df.isnull().sum())
```

```
Missing Values in Transactions DataFrame:
RECEIPT_ID        0
PURCHASE_DATE     0
SCAN_DATE         0
STORE_NAME        0
USER_ID           0
BARCODE           0
FINAL_QUANTITY    0
FINAL_SALE        0
dtype: int64
```

3.2 Products table: duplicate barcodes

The total count of barcodes is greater than the number of distinct barcodes. This means there are 4,209 duplicated entries. The same product might have slightly different attributes, causing it to be listed multiple times but with the same barcode.

[138]:
```python
# Step 1: Create an in-memory SQLite database
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
products_df.to_sql('products', conn, index=False)

# Step 3: Write the SQL query to find customers in the past year, using all␣
 ↪three tables
query = """
select count(barcode)
, count(distinct barcode)
from products
"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
   count(barcode)  count(distinct barcode)
0          845552                   841343
```

Ideally, a BARCODE should uniquely identify a product, so there shouldn't be variation in its categories or manufacturer details. This could mean that either the data was entered incorrectly, products were misclassified, or there's a need for further cleaning to ensure consistent information.

There are records without valid barcodes. It's essential to investigate why these entries are missing barcodes and determine if they need to be corrected or removed.

[144]:
```python
# Step 1: Create an in-memory SQLite database
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
products_df.to_sql('products', conn, index=False)

# Step 3: Write the SQL query to find customers in the past year, using all␣
 ↪three tables
query = """
SELECT
    barcode,
    COUNT(DISTINCT CATEGORY_1) AS distinct_category_1,
    COUNT(DISTINCT CATEGORY_2) AS distinct_category_2,
    COUNT(DISTINCT MANUFACTURER) AS distinct_manufacturers
FROM
    products
GROUP BY
    barcode
HAVING
    COUNT(DISTINCT CATEGORY_1) > 1
    OR COUNT(DISTINCT CATEGORY_2) > 1
    OR COUNT(DISTINCT MANUFACTURER) > 1;
"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
       BARCODE  distinct_category_1  distinct_category_2  \
0    1018158.0                    1                    1
1   20522445.0                    1                    2
2   20733056.0                    1                    1
3   20733254.0                    1                    2
```

```
4    3454503.0                    1                    1
5    3484708.0                    1                    1
6    4003207.0                    1                    2
7   40111216.0                    1                    1
8     404310.0                    1                    1
9    5265169.0                    1                    2
10    701983.0                    1                    2
11  80310167.0                    1                    1
12         nan                    9                   44

    distinct_manufacturers
0                        2
1                        1
2                        2
3                        1
4                        2
5                        2
6                        1
7                        2
8                        2
9                        1
10                       2
11                       2
12                     126
```

1.0.5 d. Check Distribution of FINAL_QUANTITY and FINAL_SALE

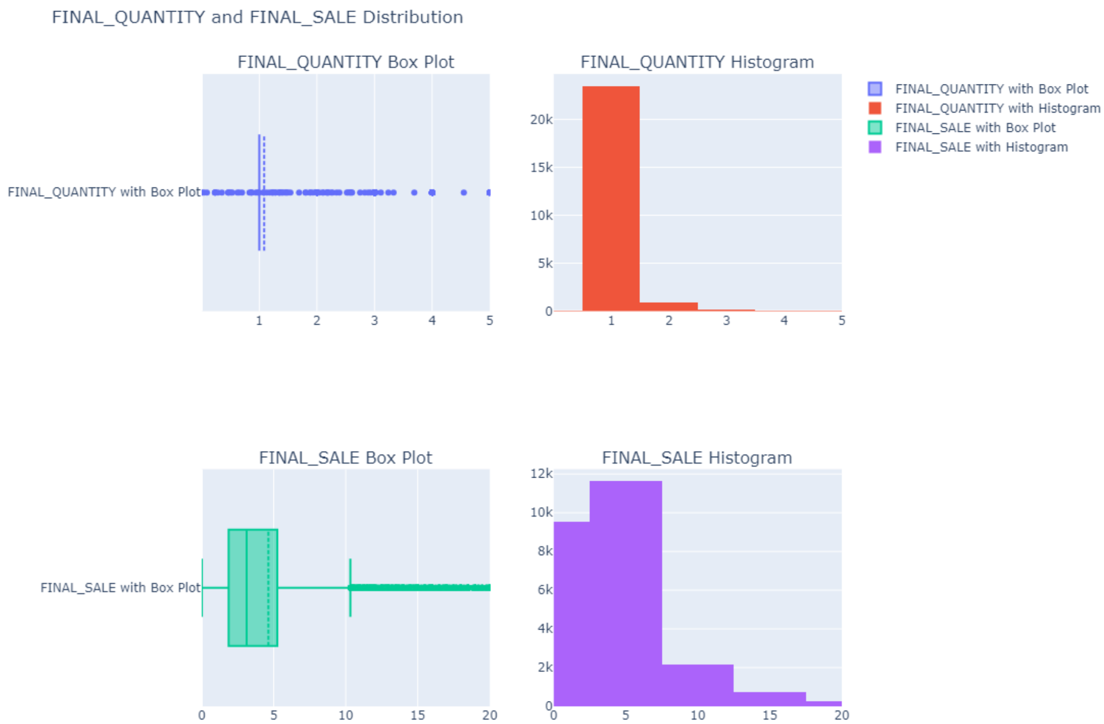
The transactions cover a period from June 12, 2024 (min) to September 8, 2024 (max), indicating a roughly 3-month period of data.

The average quantity purchased is 1.08, with most transactions involving around 1 unit of a product. The average sale amount is 4.63. The median sale is 3.12, indifacting that the majority of transactions are for relatively low-value items.

The variation of final quantity is relatively low(1.8), showing that most transactions involve small quantities The minimum quantity is 0.01, which might suggest very small or weight-based purchases. The minimum sale amount is 0.01, *whichcouldindicateeitherextremelylow − valueitemsorpotentialpromotions/giveaways.Themaximumquantityis*276, *indicatingpotentialbulkpurchasesorc* which likely due to either bulk purchases or high-ticket items.

[59]:
```python
import plotly.graph_objects as go
from plotly.subplots import make_subplots

# Create subplots: two rows, two columns
fig = make_subplots(
    rows=2, cols=2,
    subplot_titles=("FINAL_QUANTITY Box Plot", "FINAL_QUANTITY Histogram",␣
 ↪"FINAL_SALE Box Plot", "FINAL_SALE Histogram")
```

```
)

# Box plot for FINAL_QUANTITY (limit x-axis to a reasonable range, e.g., [0,␣
↪20])
trace1 = go.Box(x=transactions_df['FINAL_QUANTITY'], name='FINAL_QUANTITY with␣
↪Box Plot', boxmean=True)
fig.add_trace(trace1, row=1, col=1)
fig.update_xaxes(range=[0, 5], tickvals=[1, 2, 3, 4, 5], row=1, col=1)  #␣
↪Adjust range for FINAL_QUANTITY

# Histogram for FINAL_QUANTITY (limit x-axis range and set bin size to 1)
trace2 = go.Histogram(x=transactions_df['FINAL_QUANTITY'], name='FINAL_QUANTITY␣
↪with Histogram', xbins=dict(size=1))
fig.add_trace(trace2, row=1, col=2)
fig.update_xaxes(range=[0, 5], tickvals=[1, 2, 3, 4, 5], row=1, col=2)  #␣
↪Adjust range for FINAL_QUANTITY

# Box plot for FINAL_SALE (limit x-axis to a reasonable range, e.g., [0, 100])
trace3 = go.Box(x=transactions_df['FINAL_SALE'], name='FINAL_SALE with Box␣
↪Plot', boxmean=True)
fig.add_trace(trace3, row=2, col=1)
fig.update_xaxes(range=[0, 20], row=2, col=1)  # Adjust range for FINAL_SALE

# Histogram for FINAL_SALE (limit x-axis range and set bin size to a reasonable␣
↪value)
trace4 = go.Histogram(x=transactions_df['FINAL_SALE'], name='FINAL_SALE with␣
↪Histogram', xbins=dict(size=5))
fig.add_trace(trace4, row=2, col=2)
fig.update_xaxes(range=[0, 20], row=2, col=2)  # Adjust range for FINAL_SALE

# Update layout for better readability
fig.update_layout(height=800, width=1000, title_text="FINAL_QUANTITY and␣
↪FINAL_SALE Distribution")
fig.show()
```



[60]:

```
transactions_df.describe()
```

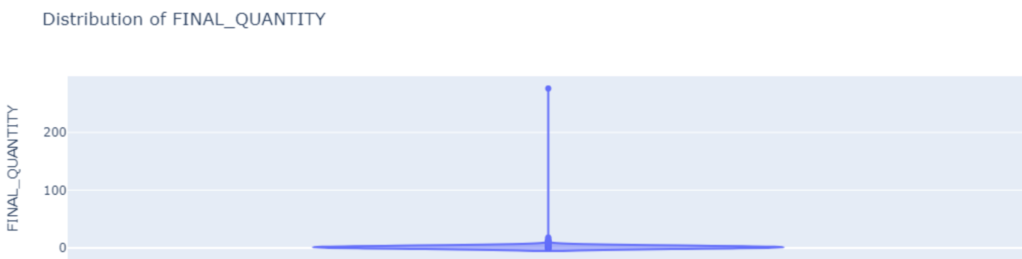
[60]:

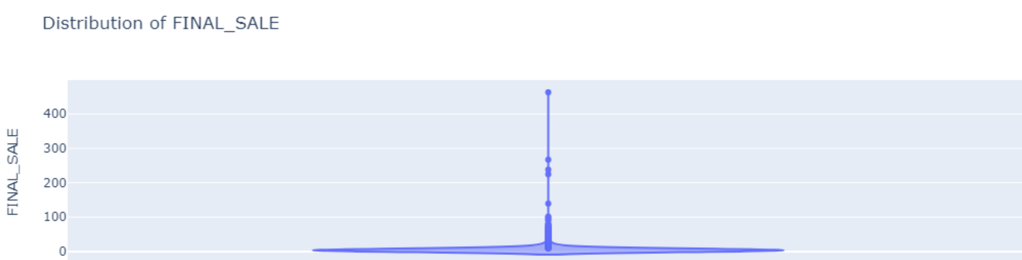
```
                      PURCHASE_DATE  FINAL_QUANTITY    FINAL_SALE
count                         24679    24679.000000  24679.000000
mean  2024-07-24 09:39:59.465132288        1.084343      4.629107
min             2024-06-12 00:00:00        0.010000      0.010000
25%             2024-07-03 00:00:00        1.000000      1.870000
50%             2024-07-24 00:00:00        1.000000      3.120000
75%             2024-08-15 00:00:00        1.000000      5.250000
max             2024-09-08 00:00:00      276.000000    462.820000
std                             NaN        1.806564      6.556336
```

[61]:

```
import plotly.express as px

# Convert FINAL_QUANTITY and FINAL_SALE to numeric if needed
transactions_df['FINAL_QUANTITY'] = pd.
 ↪to_numeric(transactions_df['FINAL_QUANTITY'], errors='coerce')
transactions_df['FINAL_SALE'] = pd.to_numeric(transactions_df['FINAL_SALE'],␣
 ↪errors='coerce')

# Create a violin plot for FINAL_QUANTITY distribution
```

```
fig_quantity = px.violin(transactions_df, y='FINAL_QUANTITY',␣
 ↪title="Distribution of FINAL_QUANTITY")
fig_quantity.show()

# Create a violin plot for FINAL_SALE distribution
fig_sale = px.violin(transactions_df, y='FINAL_SALE', title="Distribution of␣
 ↪FINAL_SALE")
fig_sale.show()
```





1.0.6 e. Time Series Analysis

From the chart, we can observe that Snacks and Health & Wellness categories have consistently higher sales and number of receipt scanned compared to other categories, with notable sales peaks around June 30th and August 25th.

Base on the insight, Fetch can take the following actions:

1. Conduct deeper analysis to understand what exactly is driving the peaks around June 30th and August 25th. If specific events (e.g., sales events, holidays) or external factors (e.g., seasonal trends) lead to more receipt scanning can help Fetch replicate and enhance these behaviors in future campaigns.

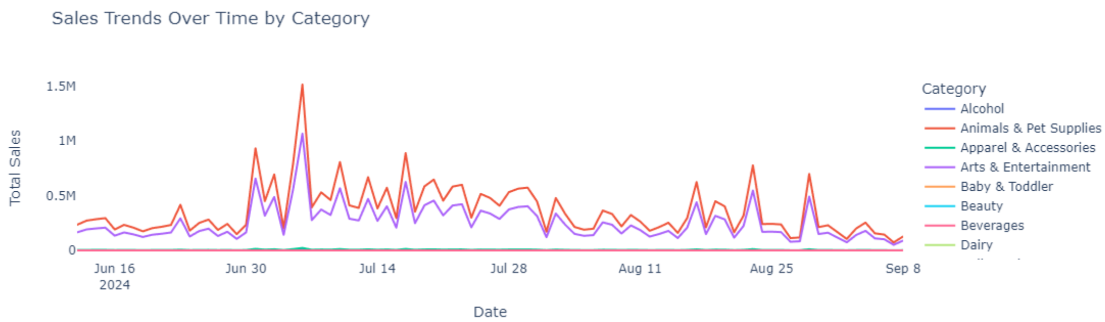
2. Send targeted push notifications to remind users of special offers or opportunities to earn extra points on snacks and health products, especially leading up to and during peak periods. For example, messages like "Earn double points this weekend when you scan receipts for your favorite snacks!".

3. Increase marketing efforts and increase ad spend ahead of peak times. Focus on promoting how users can earn extra points or unlock exclusive rewards by scanning receipts for popular products in these categories, driving greater engagement and boosting customer participation.

[62]:
```
# Step 1: Merge Transactions and Products DataFrames on 'BARCODE'
merged_df = pd.merge(transactions_df, products_df, on='BARCODE', how='left')

# Convert 'PURCHASE_DATE' to datetime
merged_df['purchase_date'] = pd.to_datetime(merged_df['PURCHASE_DATE'])

# Step 3: Group by date and category_1, summing the sales
sales_over_time_by_category = merged_df.groupby([merged_df['purchase_date'].dt.
 ↪date, 'CATEGORY_1'])['FINAL_SALE'].sum().reset_index()

# Step 4: Create a line chart using Plotly Express
fig = px.line(sales_over_time_by_category,
              x='purchase_date',
              y='FINAL_SALE',
              color='CATEGORY_1',  # Different lines for each category
              title='Sales Trends Over Time by Category',
              labels={'purchase_date': 'Date', 'FINAL_SALE': 'Total Sales',␣
 ↪'CATEGORY_1': 'Category'})

# Customize the layout for better aesthetics
fig.update_layout(
    xaxis_title='Date',
    yaxis_title='Total Sales',
    plot_bgcolor='rgba(0,0,0,0)',  # Transparent background
    xaxis=dict(showgrid=True),  # Show x-axis gridlines
    yaxis=dict(showgrid=True)   # Show y-axis gridlines
)

# Step 5: Show the chart
fig.show()
```

```
C:\Users\kathy\AppData\Local\Temp\ipykernel_29196\3672214416.py:8:
FutureWarning:

The default of observed=False is deprecated and will be changed to True in a
future version of pandas. Pass observed=False to retain current behavior or
observed=True to adopt the future default and silence this warning.

C:\Users\kathy\anaconda3\Lib\site-packages\plotly\express\_core.py:1958:
```

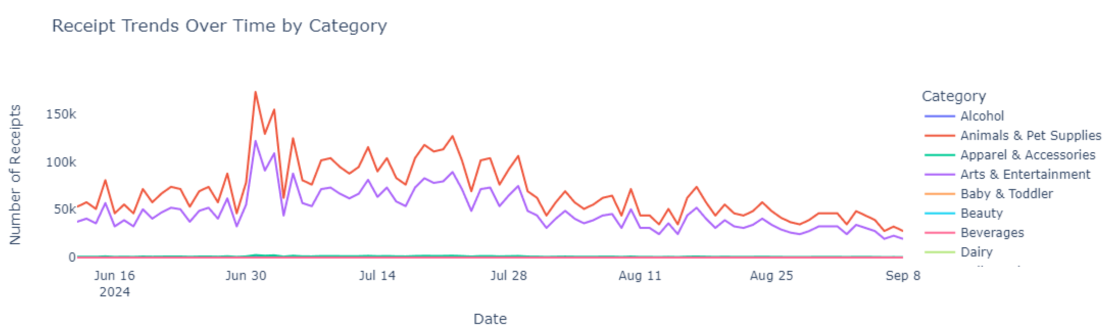
```
FutureWarning:

The default of observed=False is deprecated and will be changed to True in a
future version of pandas. Pass observed=False to retain current behavior or
observed=True to adopt the future default and silence this warning.
```



[112]:
```python
# Step 3: Group by date and category_1, counting the number of 'RECEIPT_ID'
receipts_over_time_by_category = merged_df.groupby([merged_df['purchase_date'].
    ↪dt.date, 'CATEGORY_1'])['RECEIPT_ID'].count().reset_index()

# Rename columns for clarity
receipts_over_time_by_category.rename(columns={'RECEIPT_ID': 'Receipt_Count'},␣
    ↪inplace=True)

# Step 4: Create a line chart using Plotly Express
fig = px.line(receipts_over_time_by_category,
              x='purchase_date',
              y='Receipt_Count',
              color='CATEGORY_1',  # Different lines for each category
              title='Receipt Trends Over Time by Category',
              labels={'purchase_date': 'Date', 'Receipt_Count': 'Number of␣
    ↪Receipts', 'CATEGORY_1': 'Category'})

# Customize the layout for better aesthetics
fig.update_layout(
    xaxis_title='Date',
    yaxis_title='Number of Receipts',
    plot_bgcolor='rgba(0,0,0,0)',  # Transparent background
    xaxis=dict(showgrid=True),  # Show x-axis gridlines
    yaxis=dict(showgrid=True)   # Show y-axis gridlines
)

# Step 5: Show the chart
```

```
fig.show()
```

```
C:\Users\kathy\AppData\Local\Temp\ipykernel_29196\1277788291.py:2:
FutureWarning:

The default of observed=False is deprecated and will be changed to True in a
future version of pandas. Pass observed=False to retain current behavior or
observed=True to adopt the future default and silence this warning.

C:\Users\kathy\anaconda3\Lib\site-packages\plotly\express\_core.py:1958:
FutureWarning:

The default of observed=False is deprecated and will be changed to True in a
future version of pandas. Pass observed=False to retain current behavior or
observed=True to adopt the future default and silence this warning.
```



2 2. Answer the questions

[65]:
```python
import sqlite3
```

[66]:
```python
import pandas as pd
from sqlalchemy import create_engine


# Step 1: Inspect columns
print("Original Columns:", transactions_df.columns.tolist())

# Step 2: Handle duplicate 'purchase_date' columns
if 'purchase_date' in transactions_df.columns and 'PURCHASE_DATE' in␣
 ↪transactions_df.columns:
    # Option B: Drop the duplicate
    transactions_df = transactions_df.drop(columns=['purchase_date'])
```

```
    print("Columns after dropping duplicate:", transactions_df.columns.tolist())

# Alternatively, rename if both are needed
# transactions_df = transactions_df.rename(columns={'purchase_date':␣
↪'PURCHASE_DATE_2'})

# Step 3: Standardize column names
transactions_df.columns = transactions_df.columns.str.upper()

# Verify uniqueness
print("Standardized Columns:", transactions_df.columns.tolist())
print("Number of unique columns:", len(transactions_df.columns.unique()))

# Step 4: Write to SQL
engine = create_engine('sqlite:///fetch.db')  # Replace with your actual␣
↪database URI
transactions_df.to_sql('transactions', engine, index=False, if_exists='replace')
```

```
Original Columns: ['RECEIPT_ID', 'PURCHASE_DATE', 'SCAN_DATE', 'STORE_NAME',
'USER_ID', 'BARCODE', 'FINAL_QUANTITY', 'FINAL_SALE', 'purchase_date']
Columns after dropping duplicate: ['RECEIPT_ID', 'PURCHASE_DATE', 'SCAN_DATE',
'STORE_NAME', 'USER_ID', 'BARCODE', 'FINAL_QUANTITY', 'FINAL_SALE']
Standardized Columns: ['RECEIPT_ID', 'PURCHASE_DATE', 'SCAN_DATE', 'STORE_NAME',
'USER_ID', 'BARCODE', 'FINAL_QUANTITY', 'FINAL_SALE']
Number of unique columns: 8
```

[66]: 24679

Question 1: Top 5 Brands by Receipts Scanned Among Users 21 and Over

[67]:
```
# Step 1: Create an in-memory SQLite database
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3: Write the SQL query with JOINs to merge tables and calculate age
query = """
WITH Age_Calculation AS (
    SELECT
        t.RECEIPT_ID,
        p.BRAND,
        CAST(strftime('%Y', t.SCAN_DATE) AS INTEGER) - CAST(strftime('%Y', u.
↪BIRTH_DATE) AS INTEGER) AS AGE
    FROM transactions t
    JOIN users u ON t.USER_ID = u.ID
```

```
    JOIN products p ON t.BARCODE = p.BARCODE
)
SELECT
    BRAND,
    COUNT(RECEIPT_ID) AS Receipt_Count
FROM Age_Calculation
WHERE AGE >= 21 and brand is not null
GROUP BY BRAND
ORDER BY Receipt_Count DESC
LIMIT 5;
"""

# Step 4: Execute the query and display the result
result_df = pd.read_sql_query(query, conn)
print(result_df)

# Step 5: Close the connection
conn.close()
```

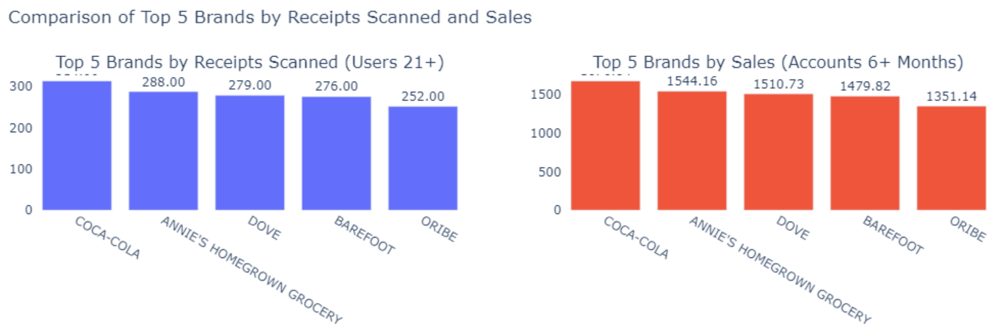
```
                      BRAND  Receipt_Count
0                 COCA-COLA            314
1  ANNIE'S HOMEGROWN GROCERY            288
2                      DOVE            279
3                  BAREFOOT            276
4                     ORIBE            252
```

Question 2: Top 5 Brands by Sales Among Users with Accounts for at Least Six Months

The graph compares the top 5 brands by receipts scanned among users aged 21 and over, with Coca-Cola leading, and the top 5 brands by sales among users with accounts for at least six months, where Coca-Cola also dominates in total sales.

[189]:
```
import plotly.express as px
import pandas as pd
from plotly.subplots import make_subplots
import plotly.graph_objects as go

# Data for first chart (Top 5 brands by receipts scanned among users 21 and␣
↪over)
data1 = {
    'BRAND': ["COCA-COLA", "ANNIE'S HOMEGROWN GROCERY", "DOVE", "BAREFOOT",␣
↪"ORIBE"],
    'Total_Receipts': [314, 288, 279, 276, 252]
}

# Data for second chart (Top 5 brands by sales among users that have had their␣
↪account for at least six months)
```

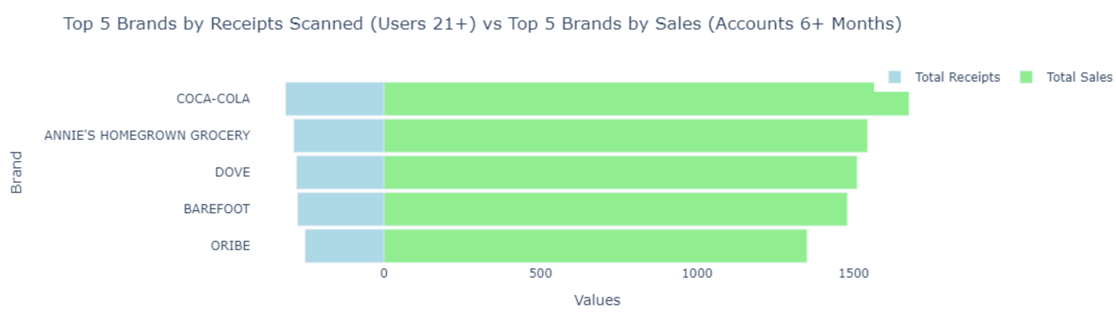
```python
data2 = {
    'BRAND': ["COCA-COLA", "ANNIE'S HOMEGROWN GROCERY", "DOVE", "BAREFOOT",
 ↪"ORIBE"],
    'Total_Sales': [1676.84, 1544.16, 1510.73, 1479.82, 1351.14]
}

# Create DataFrames
df1 = pd.DataFrame(data1)
df2 = pd.DataFrame(data2)

# Create subplots: 1 row, 2 columns
fig = make_subplots(
    rows=1, cols=2,  # 1 row and 2 columns
    subplot_titles=("Top 5 Brands by Receipts Scanned (Users 21+)", "Top 5
 ↪Brands by Sales (Accounts 6+ Months)")
)

# Add first chart (Receipts Scanned)
fig.add_trace(
    go.Bar(x=df1['BRAND'], y=df1['Total_Receipts'], text=df1['Total_Receipts'],
 ↪name="Total Receipts"),
    row=1, col=1
)

# Add second chart (Total Sales)
fig.add_trace(
    go.Bar(x=df2['BRAND'], y=df2['Total_Sales'], text=df2['Total_Sales'],
 ↪name="Total Sales"),
    row=1, col=2
)

# Update traces (formatting text for both charts)
fig.update_traces(texttemplate='%{text:.2f}', textposition='outside')

# Update layout for better aesthetics
fig.update_layout(
    title_text="Comparison of Top 5 Brands by Receipts Scanned and Sales",
    showlegend=False,
    plot_bgcolor='rgba(0,0,0,0)',  # Transparent background
    yaxis=dict(showgrid=False),  # Remove gridlines
)

# Show the combined plot
fig.show()
```



[74]:

```python
import plotly.graph_objects as go
import pandas as pd

# Data for the chart (Receipts Scanned and Total Sales by Brand)
data = {
    'BRAND': ["COCA-COLA", "ANNIE'S HOMEGROWN GROCERY", "DOVE", "BAREFOOT", "ORIBE"],
    'Total_Receipts': [314, 288, 279, 276, 252],
    'Total_Sales': [1676.84, 1544.16, 1510.73, 1479.82, 1351.14]
}

# Create a DataFrame
df = pd.DataFrame(data)

# Sort the data by Total_Receipts and Total_Sales in descending order
df = df.sort_values(by=['Total_Receipts', 'Total_Sales'], ascending=True)

# Create the butterfly chart using mirrored bars
fig = go.Figure()

# Add the first bar for Total Receipts (negative to mirror)
fig.add_trace(go.Bar(
    y=df['BRAND'],
    x=-df['Total_Receipts'],  # Negative values for mirroring
    name='Total Receipts',
    orientation='h',  # Horizontal bars
    marker=dict(color='lightblue')
))

# Add the second bar for Total Sales
fig.add_trace(go.Bar(
    y=df['BRAND'],
    x=df['Total_Sales'],  # Positive values
```

```
    name='Total Sales',
    orientation='h',  # Horizontal bars
    marker=dict(color='lightgreen')
))

# Customize layout for a better appearance
fig.update_layout(
    title='Top 5 Brands by Receipts Scanned (Users 21+) vs Top 5 Brands by␣
 ↪Sales (Accounts 6+ Months)',
    xaxis_title='Values',
    yaxis_title='Brand',
    barmode='overlay',  # Overlay the bars
    bargap=0.1,  # Small gap between bars
    xaxis=dict(showgrid=False),  # Remove gridlines
    plot_bgcolor='rgba(0,0,0,0)',  # Transparent background
    legend=dict(x=0.9, y=1.1, orientation='h')  # Horizontal legend
)

# Show the butterfly chart
fig.show()
```



[184]:
```
# Step 1: Create an in-memory SQLite database
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3: Write the SQL query to calculate account age and find top brands by␣
 ↪sales
query = """
WITH Account_Age_Calculation AS (
```

```
    SELECT
        t.RECEIPT_ID,
        p.BRAND,
        t.FINAL_SALE,
        (CAST(strftime('%Y', t.SCAN_DATE) AS INTEGER) - CAST(strftime('%Y', u.
↪CREATED_DATE) AS INTEGER)) * 12 +
        (CAST(strftime('%m', t.SCAN_DATE) AS INTEGER) - CAST(strftime('%m', u.
↪CREATED_DATE) AS INTEGER)) AS Account_Age_Months
    FROM transactions t
    JOIN users u ON t.USER_ID = u.ID
    JOIN products p ON t.BARCODE = p.BARCODE
)
SELECT
    BRAND,
    SUM(FINAL_SALE) AS Total_Sales
FROM Account_Age_Calculation
WHERE Account_Age_Months >= 6 and Brand is not null
GROUP BY BRAND
ORDER BY Total_Sales DESC
LIMIT 5;
"""

# Step 4: Execute the query and display the result
result_df = pd.read_sql_query(query, conn)
print(result_df)

# Step 5: Close the connection
conn.close()
```

```
                       BRAND  Total_Sales
0                  COCA-COLA      1676.84
1  ANNIE'S HOMEGROWN GROCERY      1544.16
2                       DOVE      1510.73
3                   BAREFOOT      1479.82
4                      ORIBE      1351.14
```

Question 3: Percentage of Sales in the Health & Wellness Category by Generation

Millennials (ages 25 to 40) dominate Health & Wellness sales, driven by trends in fitness, self-care, and proactive health management, followed by Baby Boomers at 28%, Gen X at 19%, and a minimal contribution from the Silent Generation.

[167]:
```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)
```

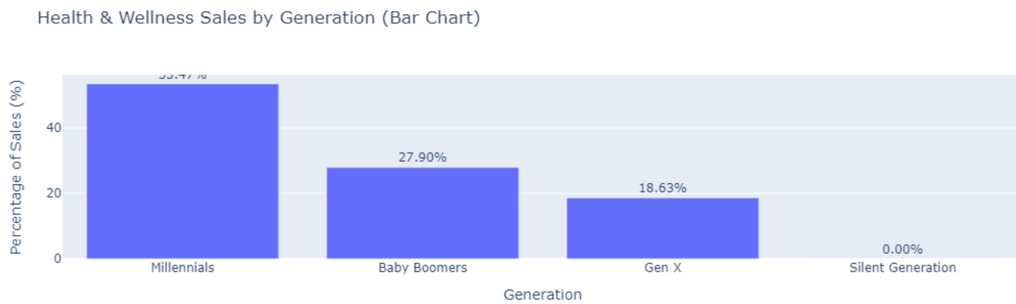
```
# Step 3: Write the SQL query to calculate percentage of Health & Wellness␣
↪sales by generation
query = """
WITH Age_Calculation AS (
    SELECT
        t.RECEIPT_ID,
        p.CATEGORY_1,
        t.FINAL_SALE,
        CAST(strftime('%Y', t.SCAN_DATE) AS INTEGER) - CAST(strftime('%Y', u.
↪BIRTH_DATE) AS INTEGER) AS AGE
    FROM transactions t
    JOIN users u ON t.USER_ID = u.ID
    JOIN products p ON t.BARCODE = p.BARCODE
),
Generations AS (
    SELECT
        CASE
            WHEN AGE BETWEEN 9 AND 24 THEN 'Gen Z'
            WHEN AGE BETWEEN 25 AND 40 THEN 'Millennials'
            WHEN AGE BETWEEN 41 AND 56 THEN 'Gen X'
            WHEN AGE BETWEEN 57 AND 75 THEN 'Baby Boomers'
            WHEN AGE > 75 THEN 'Silent Generation'
            ELSE 'Other'
        END AS Generation,
        FINAL_SALE,
        CATEGORY_1
    FROM Age_Calculation
),
Total_Health_Wellness_Sales AS (
    -- Calculate total sales in the Health & Wellness category across all␣
↪generations
    SELECT
        SUM(FINAL_SALE) AS Total_Health_Wellness_Sales
    FROM Generations
    WHERE CATEGORY_1 = 'Health & Wellness'
)
-- Final query to calculate the percentage of Health & Wellness sales by␣
↪generation
SELECT
    Generation,
    SUM(FINAL_SALE) * 100.0 / (SELECT Total_Health_Wellness_Sales FROM␣
↪Total_Health_Wellness_Sales) AS Percentage_Health_Wellness_Sales
FROM Generations
WHERE CATEGORY_1 = 'Health & Wellness'
GROUP BY Generation
ORDER BY Percentage_Health_Wellness_Sales DESC;
```

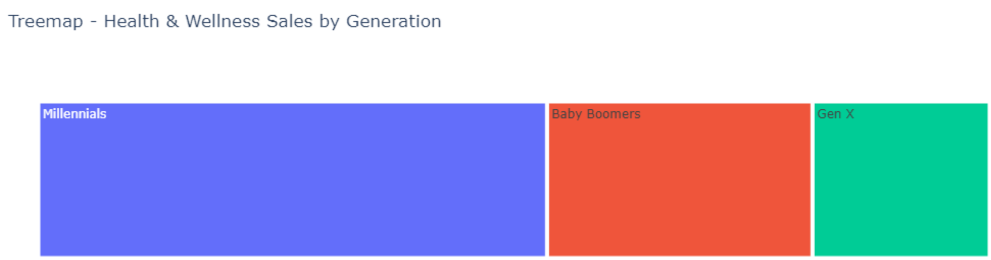
```
"""

# Step 4: Execute the query and display the result
result_df = pd.read_sql_query(query, conn)
print(result_df)

# Step 5: Close the connection
conn.close()
```

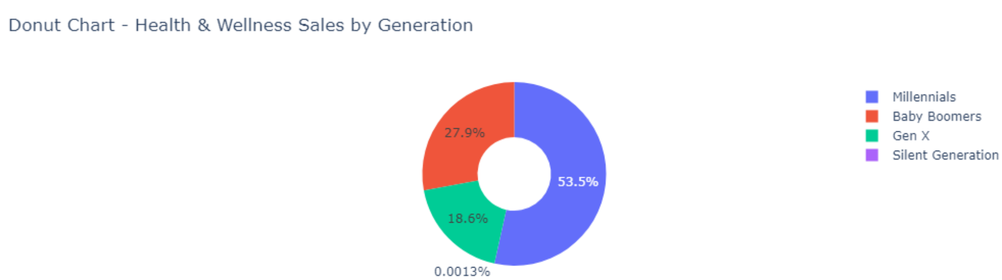
```
           Generation  Percentage_Health_Wellness_Sales
0         Millennials                         53.468716
1        Baby Boomers                         27.897526
2               Gen X                         18.632432
3   Silent Generation                          0.001326
```

[86]:
```
# Data for the generation health and wellness sales
result_df = pd.DataFrame({
    'Generation': ['Millennials', 'Baby Boomers', 'Gen X', 'Silent Generation'],
    'Percentage_Health_Wellness_Sales': [53.47, 27.90, 18.63, 0.0013]
})



# Create 100% stacked bar chart
result_df['Total'] = result_df['Percentage_Health_Wellness_Sales']  # A trick␣
 ↪to use for 100% stacked
stacked_bar_chart = px.bar(result_df, x='Generation', y='Total',
                           title='100% Stacked Bar Chart - Health & Wellness␣
 ↪Sales by Generation',
                           labels={'Total': 'Percentage of Sales (%)',␣
 ↪'Generation': 'Generation'},
                           text='Total')

# Create treemap chart
treemap_chart = px.treemap(result_df, path=['Generation'],␣
 ↪values='Percentage_Health_Wellness_Sales',
                           title='Treemap - Health & Wellness Sales by␣
 ↪Generation')

# Create donut chart (pie chart with a hole)
donut_chart = px.pie(result_df, names='Generation',␣
 ↪values='Percentage_Health_Wellness_Sales', hole=0.4,
                     title='Donut Chart - Health & Wellness Sales by␣
 ↪Generation')



# Display all charts
bar_chart.show()
```

```
treemap_chart.show()
donut_chart.show()
```







Question 4: Who are Fetch's Power Users?

We want to focus on frequent customers who have been active within the past 3 months and consistently scan receipts, as these users represent our most engaged and valuable

customer base.

[131]:

```
# Step 1: Create an in-memory SQLite database
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3: Write the SQL query to find customers in the past year, using all␣
↪three tables
query = """
WITH Merged_Data AS (
    SELECT
        t.USER_ID,
        t.SCAN_DATE,
        t.FINAL_SALE,
        t.RECEIPT_ID,
        u.ID AS User_ID,
        p.BRAND
    FROM transactions t
    LEFT JOIN users u ON t.USER_ID = u.ID
    LEFT JOIN products p ON t.BARCODE = p.BARCODE
    WHERE t.SCAN_DATE >= date('now', '-1 year')
)
SELECT
    USER_ID,
    COUNT(RECEIPT_ID) AS Total_Receipts,
    SUM(FINAL_SALE) AS Total_Sales
FROM Merged_Data
GROUP BY USER_ID
ORDER BY Total_Receipts DESC, Total_Sales DESC
LIMIT 10;
"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print('Top 10 Power User in the past 3 months')
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
Top 10 Power User in the past 3 months
                    USER_ID  Total_Receipts  Total_Sales
0  64e62de5ca929250373e6cf5           40251    105742.73
```

1	604278958fe03212b47e657b	36226	64853.22
2	64023fa080552327896edb23	20126	56150.97
3	66390784b7b24d45d93a0e6a	20126	33130.54
4	63ae0dc29f3fc9c7546ef080	16102	55444.23
5	5d8661a736d69e65e99233af	16101	34139.59
6	61aea787e9b3d75037b5ea45	12078	25490.99
7	6456eac19f7c516a13f471f4	12077	57846.21
8	634aee03305e373439460ac3	12077	45205.05
9	63d97d69b425eb11a4709d56	12077	45170.28

3 3. Communicate with stakeholders

Email:

Dear Brandon,

I am pleased to share a summary of the key findings from the analysis of user and transaction data collected between June 12, 2024, and September 8, 2024. Below, I've highlighted the main insights, addressed data quality concerns, and outlined requests for clarification as well as proposed next steps.

First, I noticed several data quality issues that need to be addressed:

1. Duplicate Entries in the transaction table for the same receipt IDs and barcodes: Only the entries with valid quantity and sale information for each transaction are kept.
2. Duplicate Barcodes in the products table: Duplicate barcodes can lead to inflated sales counts and other metrics, as the same transaction may be counted multiple times during joins. To address this, we need to standardize product information and ensure that barcode values in the products table are unique.
3. Missing Values: Several fields, including product categories, have missing or zero values.
4. Extreme values: Very small quantities and sales values (e.g., 0.01) may be outliers. We need to determine if these represent products sold by weight, extremely low-value items, promotions/giveaways, or if they might be due to data entry errors.

Then, several key trends are identified along with recommendations to help drive future strategy:

1. The top brands by receipts scanned among users aged 21+ and by sales for accounts active for at least six months include prominent names like COCA-COLA, ANNIE'S HOMEGROWN GROCERY, and DOVE. I would recommend leverage these top-performing brands in partnership or loyalty campaigns to increase Fetch's engagement. Offering exclusive promotions for these brands can help further drive sales and user interaction.
2. Snacks and Health & Wellness categories have consistently higher sales and number of receipt scanned compared to other categories, with notable sales peaks around June 30th and August 25th. Fetch can collaborate with brands to launch promotions, limited-time deals, or bonus points offers that incentivize users to buy more during these periods. By showcasing more offers for products in Snacks and Health & Wellness, Fetch can drive more sales and scans, as users are already inclined to buy these products.
3. A small subset of users is responsible for the majority of receipts and sales, with only 0.1% of users making transactions in the past three months. We need to priortize user rentention and engagement strategies.

Lastly, to move forward, I would like to confirm the following:

1. It's a concern that we only have 50k rows of tranaction data, as this may limit the accuracy of our insights. To ensure more reliable results, we need additional data points to validate our findings.
2. We need to work with the data team to validate the cause of the data entry errors and discuss solutions to fix them.
3. It would be helpful to get clarification on the missing product categories and brands for certain transactions. Are there alternative sources where we could pull this information?
4. To improve our customer segmentation, it would be helpful to have additional behavioral data and demographic data (e.g., income level, family size). This would allow us to conduct more refined customer segmentation and better tailor our marketing strategies.

Please let me know how we can proceed on these points. I look forward to discussing these insights further.

Thank you,

Jiatong Song

4 Appendix

4.0.1 i. User activity analysis

Only 0.091% users have transactions in the past 3 months (2024-06-12 to 2024-09-08)

[137]:
```
# Step 1: Create an in-memory SQLite database
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3: Write the SQL query to find customers in the past year, using all␣
↪three tables
query = """
WITH users_without_transactions AS (
    SELECT u.ID
    FROM Users u
    LEFT JOIN Transactions t
    ON u.ID = t.USER_ID
    WHERE t.USER_ID IS NULL
),
total_users AS (
    SELECT COUNT(*) AS total_user_count
    FROM Users
),
users_no_transaction_count AS (
    SELECT COUNT(*) AS no_transaction_count
    FROM users_without_transactions
```

```
)
SELECT
    (u.no_transaction_count * 100.0 / t.total_user_count) AS␣
 ↪percentage_no_transaction
FROM
    users_no_transaction_count u,
    total_users t;
"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
   percentage_no_transaction
0                     99.909
```

Churn rate analysis on customers who joined in different years

Definition of Churned Customers: Customers who do not have any transactions between 2024-06-12 and 2024-09-08

Given the churn rate for customers across all cohorts is extremely high, here are some potential strategies Fetch could consider to address this issue:

a) Marketing efforts

1. Push Notifications for Inactivity: Automatically send a gentle nudge via push notification or SMS to users who haven't engaged with the app in the last two weeks. This could include reminders about available offers, new promotions, or unclaimed rewards.

2. Weekly Emails: Send a weekly roundup email summarizing the latest deals, top brands to earn rewards with, and a reminder to scan receipts.

3. Highlight special deals that are personalized based on users' past behaviors.

4. Gamified Challenges: Create engagement challenges, such as "Log in to the app and earn coins/points for consecutive daily check-ins" to reinforce more frequent use of the app.

b) Data-Driven techniques

1. Predictive Analysis: Use machine learning to predict which customers are most likely to churn and intervene early. If a user shows signs of disengagement (e.g., hasn't scanned a receipt in a month), proactively reach out to them with incentives or reminders.

2. Segment Analysis: Perform deeper analysis to determine if churn rates vary significantly by demographics (e.g., state, language, age). This information could guide more personalized and effective engagement strategies for each segment.

c) Analyze customer behavior and feedback

It would be essential to understand why customers are not returning through surveys. Deep-dive into the data to identify patterns in customer behavior. For instance, analyze the engagement of customers who do not churn to understand what keeps them active, and try to replicate those engagement patterns across other customer segments.

[113]:
```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
query = """
-- Step 1: Define Customer Cohorts Based on Join Year Using strftime()
WITH customer_cohorts AS (
    SELECT
        c.ID,
        CAST(strftime('%Y', c.CREATED_DATE) AS INTEGER) AS join_year
    FROM
        users c
),

-- Step 2: Identify Active Customers (Those with Transactions in the Specified␣
 ↪Period)
active_customers AS (
    SELECT
        DISTINCT t.USER_ID
    FROM
        transactions t
    WHERE
        t.PURCHASE_DATE BETWEEN '2024-06-12' AND '2024-09-08'
),

-- Step 3: Combine Cohorts with Active Status to Determine Churned Customers
churn_analysis AS (
```

```
    SELECT
        cc.join_year,
        COUNT(*) AS total_customers,
        SUM(CASE
                WHEN ac.USER_ID IS NULL THEN 1
                ELSE 0
            END) AS churned_customers
    FROM
        customer_cohorts cc
    LEFT JOIN
        active_customers ac ON cc.ID = ac.USER_ID
    GROUP BY
        cc.join_year
)

-- Step 4: Calculate Churn Rate Percentage
SELECT
    join_year,
    total_customers,
    churned_customers,
    ROUND((churned_customers * 100.0) / total_customers, 2) AS␣
 ↪churn_rate_percentage
FROM
    churn_analysis
ORDER BY
    join_year;

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
   join_year  total_customers  churned_customers  churn_rate_percentage
0       2014               30                 30                 100.00
1       2015               51                 51                 100.00
2       2016               70                 70                 100.00
3       2017              644                642                  99.69
4       2018             2168               2166                  99.91
5       2019             7093               7087                  99.92
6       2020            16883              16867                  99.91
7       2021            19159              19150                  99.95
8       2022            26807              26784                  99.91
```

9	2023	15464	15444	99.87
10	2024	11631	11619	99.90

Monthly Active Users (MAU): number of unique users who were active (scanned at least one receipt) for each month

[119]:
```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
query = """
SELECT
    strftime('%Y-%m', SCAN_DATE) AS month_start,
    COUNT(DISTINCT USER_ID) AS monthly_active_users
FROM Transactions
GROUP BY strftime('%Y-%m', SCAN_DATE)
ORDER BY month_start;

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
  month_start  monthly_active_users
0     2024-06                  4296
1     2024-07                  7959
2     2024-08                  7349
3     2024-09                  2074
```

Users Active Every Week

[120]:
```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
```

```
query = """
WITH weekly_activity AS (
    SELECT
        USER_ID,
        strftime('%Y-%W', SCAN_DATE) AS week_start
    FROM Transactions
    GROUP BY USER_ID, strftime('%Y-%W', SCAN_DATE)
),
active_users_all_weeks AS (
    SELECT
        USER_ID
    FROM weekly_activity
    GROUP BY USER_ID
    HAVING COUNT(DISTINCT week_start) = (SELECT COUNT(DISTINCT
 ↪strftime('%Y-%W', SCAN_DATE)) FROM Transactions)
)
SELECT COUNT(DISTINCT USER_ID) AS users_active_every_week
FROM active_users_all_weeks;

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
   users_active_every_week
0                        0
```

Users Active Every Month

[121]:
```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
query = """
WITH monthly_activity AS (
    SELECT
        USER_ID,
        strftime('%Y-%m', SCAN_DATE) AS month_start
```

```
    FROM Transactions
    GROUP BY USER_ID, strftime('%Y-%m', SCAN_DATE)
),
active_users_all_months AS (
    SELECT
        USER_ID
    FROM monthly_activity
    GROUP BY USER_ID
    HAVING COUNT(DISTINCT month_start) = (SELECT COUNT(DISTINCT␣
↪strftime('%Y-%m', SCAN_DATE)) FROM Transactions)
)
SELECT COUNT(DISTINCT USER_ID) AS users_active_every_month
FROM active_users_all_months;

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
   users_active_every_month
0                        29
```

[122]:

```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
query = """
SELECT
    USER_ID,
    COUNT(DISTINCT RECEIPT_ID) AS receipts_scanned
FROM Transactions
GROUP BY USER_ID
ORDER BY receipts_scanned DESC;

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
```

```python
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
                        USER_ID  receipts_scanned
0      64e62de5ca929250373e6cf5                10
1      62925c1be942f00613f7365e                10
2      64063c8880552327897186a5                 9
3      6327a07aca87b39d76e03864                 7
4      624dca0770c07012cd5e6c03                 7
…                           …                 …
17513  5748f001e4b03a732e4ecdc0                 1
17514  5640f111e4b0a905f487f861                 1
17515  56242219e4b07364e3e0bef4                 1
17516  548e5dfae4b096ae8875dfec                 1
17517  53ce6404e4b0459d949f33e9                 1

[17518 rows x 2 columns]
```

Average Receipts Scanned per User per Month

[128]:
```python
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
query = """
WITH monthly_scans AS (
    SELECT
        USER_ID,
        strftime('%Y-%m', SCAN_DATE) AS month_start,
        COUNT(RECEIPT_ID) AS receipts_scanned
    FROM Transactions
    GROUP BY USER_ID, month_start
)
SELECT
    month_start,
    AVG(receipts_scanned) AS avg_receipts_per_user
FROM monthly_scans
GROUP BY month_start
ORDER BY month_start;
```

```
"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
  month_start  avg_receipts_per_user
0     2024-06               1.090317
1     2024-07               1.160824
2     2024-08               1.159069
3     2024-09               1.079074
```

4.0.2 ii. Store Performance Analysis

Top 10 stores based on total receipts scanned

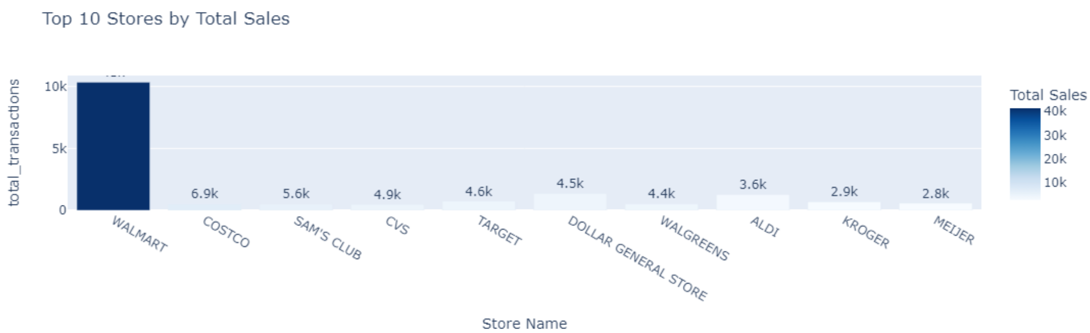
Given Walmart has the most receipts scanned, with Health & Wellness and Snacks being the top categories, Fetch can Understand what makes Walmart successful (e.g., pricing, product availability, customer loyalty) and apply these learnings to help other partner stores improve their engagement and sales. Additionally, Fetch can create a dedicated section within the Fetch app that highlights Walmart deals, user promotions, and rewards specifically for shopping at Walmart.

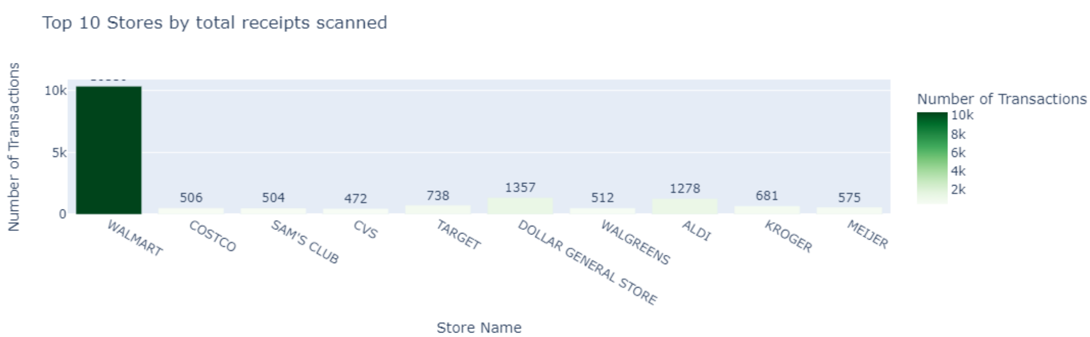
[97]:
```
import pandas as pd
import plotly.express as px
from sqlalchemy import create_engine

# Example using SQLite (adjust as needed)
engine = create_engine('sqlite:///fetch.db')  # Replace with your database

# Step 2: Define the SQL Query
sql_query = """
SELECT
    STORE_NAME,
    COUNT(distinct RECEIPT_ID) AS total_transactions,
    SUM(FINAL_SALE) AS total_sales
FROM
    transactions
GROUP BY
    STORE_NAME
ORDER BY
    total_sales DESC
LIMIT 10;
"""
```

```python
# Step 3: Execute the Query and Load the Data into a DataFrame
try:
    top_stores_df = pd.read_sql_query(sql_query, engine)
    print("Top Stores Data:")
    print(top_stores_df)
except Exception as e:
    print("An error occurred while executing the SQL query:")
    print(e)

# Step 4: Visualize the Results with Plotly Express
# Bar Chart: Total Sales per Store
fig_sales = px.bar(
    top_stores_df,
    x='STORE_NAME',
    y='total_transactions',
    title='Top 10 Stores by Total Sales',
    labels={'STORE_NAME': 'Store Name', 'total_sales': 'Total Sales'},
    text='total_sales',
    color='total_sales',
    color_continuous_scale='Blues'
)

fig_sales.update_traces(texttemplate='%{text:.2s}', textposition='outside')
fig_sales.update_layout(uniformtext_minsize=8, uniformtext_mode='hide')
fig_sales.show()

# Bar Chart: Total Transactions per Store
fig_transactions = px.bar(
    top_stores_df,
    x='STORE_NAME',
    y='total_transactions',
    title='Top 10 Stores by total receipts scanned',
    labels={'STORE_NAME': 'Store Name', 'total_transactions': 'Number of␣
↪Transactions'},
    text='total_transactions',
    color='total_transactions',
    color_continuous_scale='Greens'
)

fig_transactions.update_traces(texttemplate='%{text}', textposition='outside')
fig_transactions.update_layout(uniformtext_minsize=8, uniformtext_mode='hide')
fig_transactions.show()
```

```
Top Stores Data:
             STORE_NAME  total_transactions  total_sales
0               WALMART               10339     41300.06
1                COSTCO                 506      6919.78
```

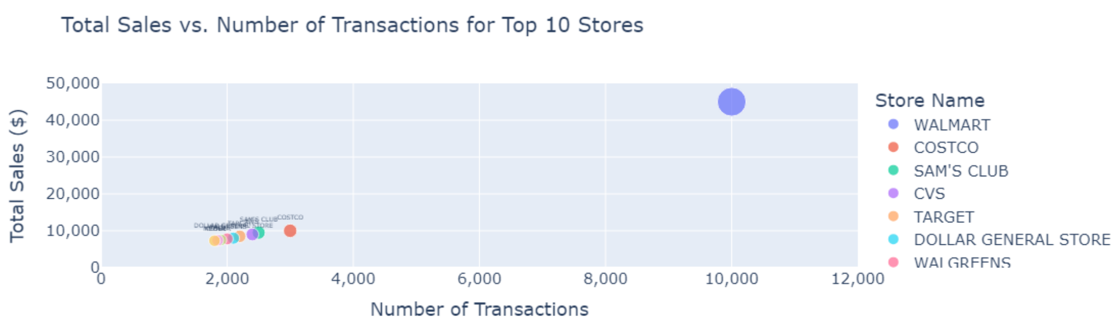
```
2            SAM'S CLUB                504     5633.92
3                   CVS                472     4949.93
4                TARGET                738     4624.60
5  DOLLAR GENERAL STORE               1357     4461.34
6             WALGREENS                512     4420.86
7                  ALDI               1278     3611.49
8                KROGER                681     2908.76
9                MEIJER                575     2766.04
```





[94]:
```python
fig = px.scatter(
    top_stores_df,
    x='total_transactions',
    y='total_sales',
    size='total_sales',  # Bubble size based on total sales
    color='STORE_NAME',
    text='STORE_NAME',  # Add store names as labels
    title='Total Sales vs. Number of Transactions for Top 10 Stores',
    labels={'total_transactions': 'Number of Transactions', 'total_sales':
 ↪'Total Sales ($)'}
)
```

```python
# Adjust layout for better readability
fig.update_traces(
    textposition='top center',
    textfont=dict(size=6),  # Make the label text smaller
    marker=dict(opacity=0.7, sizemode='area')
)

fig.update_layout(
    xaxis=dict(title='Number of Transactions', tickformat=',', range=[0, 12000]),  # Adjust x-axis range if needed
    yaxis=dict(title='Total Sales ($)', tickformat=',', range=[0, 50000]),  # Adjust y-axis range if needed
    title_font_size=20,
    legend_title_text='Store Name',
    font=dict(size=15)
)


# Show the plot
fig.show()
```



Top 10 Primary Product Categories by Quantity Sold and Receipt Count at Walmart

[102]:
```python
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
query = """
SELECT
    p.CATEGORY_1 AS primary_category,
```

```
    COUNT(t.RECEIPT_ID) AS total_purchases,
    SUM(t.FINAL_QUANTITY) AS total_quantity_sold
FROM
    transactions t
JOIN
    products p ON t.BARCODE = p.BARCODE
JOIN
    users c ON t.USER_ID = c.ID
WHERE
    t.STORE_NAME = 'WALMART'
GROUP BY
    p.CATEGORY_1
ORDER BY
    total_quantity_sold DESC, total_purchases DESC
LIMIT 10;

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
        primary_category  total_purchases  total_quantity_sold
0       Health & Wellness             4619               4619.0
1                  Snacks             3265               3265.0
2              Restaurant               68                 68.0
3                 Alcohol               57                 57.0
4               Beverages               25                 25.0
5                   Dairy               21                 21.0
6   Apparel & Accessories               12                 12.0
7                  Pantry                8                  8.0
8            Deli & Bakery                6                  6.0
```

4.0.3 iii. Records with a final sale amount of 0.01

[70]:
```
conn = sqlite3.connect(':memory:')

# Step 2: Write each dataframe to the SQLite database
users_df.to_sql('users', conn, index=False)
products_df.to_sql('products', conn, index=False)
transactions_df.to_sql('transactions', conn, index=False)

# Step 3
```

```python
query = """
SELECT
    p.CATEGORY_1,
    p.CATEGORY_2,
    p.MANUFACTURER,
    p.BRAND,
    p.BARCODE,
    t.STORE_NAME,
    Final_quantity,
    Final_sale
FROM
    transactions t
LEFT JOIN
    products p
    ON t.BARCODE = p.BARCODE
WHERE
    t.receipt_id IN (
        SELECT t_inner.receipt_id
        FROM transactions t_inner
        ORDER BY t_inner.final_sale asc
        LIMIT 1
    );

"""

# Step 4: Execute the query and load the result into a pandas DataFrame
result_df = pd.read_sql_query(query, conn)

# Step 5: Display the result
print(result_df)

# Step 6: Close the connection
conn.close()
```

```
     CATEGORY_1            CATEGORY_2             MANUFACTURER  \
0       Alcohol                  None                    GALLO
1       Alcohol                  None              MOLSONCOORS
2       Alcohol                  Beer              MOLSONCOORS
3       Alcohol                  Beer              MOLSONCOORS
4       Alcohol                  Beer              MOLSONCOORS
…          …                   …                      …
4020     Snacks             Trail Mix             HORMEL FOODS
4021     Snacks             Trail Mix                  PEPSICO
4022     Snacks             Trail Mix                  PEPSICO
4023     Snacks  Variety Snack Packs                      None
4024     Snacks  Variety Snack Packs  MONDELĒZ INTERNATIONAL

                                BRAND BARCODE   STORE_NAME  FINAL_QUANTITY  \
```

```
0                           BAREFOOT     nan  GIANT EAGLE             1.0
1                        COORS LIGHT     nan  GIANT EAGLE             1.0
2                        COORS LIGHT     nan  GIANT EAGLE             1.0
3                        COORS LIGHT     nan  GIANT EAGLE             1.0
4                        COORS LIGHT     nan  GIANT EAGLE             1.0
…                                  …     …           …              …
4020                        PLANTERS     nan  GIANT EAGLE             1.0
4021                     NUT HARVEST     nan  GIANT EAGLE             1.0
4022  OCEAN SPRAY SINGLE SERVE JUICES     nan  GIANT EAGLE             1.0
4023                            None     nan  GIANT EAGLE             1.0
4024                         NABISCO     nan  GIANT EAGLE             1.0

      FINAL_SALE
0           0.01
1           0.01
2           0.01
3           0.01
4           0.01
…            …
4020        0.01
4021        0.01
4022        0.01
4023        0.01
4024        0.01

[4025 rows x 8 columns]
```