Exploratory Analysis Report

Introduction:

This report presents a data quality audit and exploratory analysis of three CSV files — User_Takehome, Transaction_Takehome, And Products_Takehome. The objective is to identify any quality issues or ambiguous fields that may hinder analysis and to support all findings with Python code.

The data reflects transactional activity, user demographics, and product metadata — aligned with systems like Fetch Rewards. All conclusions are based on the latest version of the dataset provided.

Important Context on Data Types:

Upon review, all three files originate from Excel and use the "General" column format, which does not enforce strict typing. As a result:

- Every column across the datasets (except for Barcode) is interpreted by Python as object (i.e., string), even when the ER diagram defines them as numeric, datetime, varchar or integer types.
- Notably, fields like Final_Quantity, Final_Sale, User_Id, And Created_Date are all typed as object, which prevents direct numerical or date-based operations without prior cleaning.
- The only exception is Barcode, which was interpreted as float64, likely due to its long numeric values.

This data type inconsistency is considered a core quality issue, as it impacts validation, aggregation, and analytics reliability. Each section of this report will address how such issues manifest across each individual file.

User Take Home Dataset:

This dataset has 100,000 rows and contains demographic data including Id, Created_Date, Birth_Date, State, Language, And Gender.

Datatypes:

The datatypes of each column are presented below:

```
print(user_df.dtypes)  # Show data types of each column
```

```
ID              object
CREATED_DATE    object
BIRTH_DATE      object
STATE           object
LANGUAGE        object
GENDER          object
dtype: object
```

It can be seen that all of the columns have the object datatype which is not accurate. This can cause quite a few issues. First, date fields cannot be used in time-based calculations. Second, no type enforcement would mean that Python does not check for invalid formats and forces manual cleaning. Fields like these should be explicitly converted at the beginning so that they do not cause problems further along.

Missing Values:

The total number of missing values for each column are presented below:

```
user_missing_count = user_df.isna().sum() #Sum all the missing values in each column
print(user_missing_count) #Print the missing values in each column
```

```
ID                 0
CREATED_DATE       0
BIRTH_DATE      3675
STATE           4812
LANGUAGE       30508
GENDER          5892
dtype: int64
```

A significant number of records are missing key user demographics. Missing Language values may affect regional marketing efforts. Gender and State gaps can hinder segmentation and geographic targeting. Birth_Date missing values prevent age-based analysis or cohort modeling.

Empty Strings:

No empty strings detected (i.e., values like '' or ' '):

```
user_empty_string = (user_df.astype(str).apply(lambda col: col.str.strip() == '')).sum() #Sum
print(user_empty_string) #Print the empty strings in each column
```

```
ID              0
CREATED_DATE    0
BIRTH_DATE      0
STATE           0
LANGUAGE        0
GENDER          0
dtype: int64
```

This means blanks are represented as proper NaN, not hidden as strings.

Duplicate Values:

No duplicate IDs detected:

```
user_duplicate_ids = user_df['ID'].duplicated().sum() #Sum all the duplicate IDs
print(f"Number of duplicate IDs: {user_duplicate_ids}") #Print the duplicate IDs
```

```
Number of duplicate IDs: 0
```

No duplicate rows detected:

```
user_duplicate_rows = user_df.duplicated().sum() #Sum all the duplicate rows
print(f"Number of duplicate rows: {user_duplicate_rows}") #Print all the duplicate rows
```

```
Number of duplicate rows: 0
```

No structural duplication observed. Good data hygiene in terms of primary key uniqueness.

Unusual Birth Years:

Unusual entries were detected in the Birth_Date column:

```
#Validating the entries in the BIRTH_DATE column to see if they make sense
user_df['BIRTH_DATE'] = pd.to_datetime(user_df['BIRTH_DATE'], errors='coerce')
print("Earliest:", user_df['BIRTH_DATE'].min()) #Check the earliest year
print("Latest:", user_df['BIRTH_DATE'].max()) #Check the latest year
```

```
Earliest: 1900-01-01 00:00:00+00:00
Latest: 2022-04-03 07:00:00+00:00
```

The number of entries with earliest and latest values in the birth year column are shown below:

```
#Earliest BIRTH_DATE column value 1900 and latest value 2022 need further business analysis to check whether it is an error or a placeholder value

birth_year_1900 = user_df[user_df['BIRTH_DATE'].dt.year == 1900] #Find rows with birth year 1900
print(f"Rows with birth year 1900: {birth_year_1900.shape[0]}") # Count the number of such rows
print(birth_year_1900[['ID', 'BIRTH_DATE']].head()) # Show the first few rows
```

```
Rows with birth year 1900: 4
```

```
birth_year_2022 = user_df[user_df['BIRTH_DATE'].dt.year == 2022] #Find rows with birth year 2022
print(f"Rows with birth year 2022: {birth_year_2022.shape[0]}") # Count the number of such rows
print(birth_year_2022[['ID', 'BIRTH_DATE', 'CREATED_DATE', 'STATE', 'LANGUAGE', 'GENDER']].head()) # Show the first few rows
```

```
Rows with birth year 2022: 2
```

The year 1900 may represent a placeholder value and the year 2022 may indicate test data. These values should be flagged for manual review and further business analysis.

State Format:

All values in the State column (where present) follow the correct two-letter uppercase format

```
state_format_check = user_df["STATE"].str.fullmatch(r"[A-Z]{2}").value_counts()
print(state_format_check) #Print the count of matches
```

```
STATE
True    95188
Name: count, dtype: int64
```

No format issues detected.

Language Values:

No spelling inconsistencies:

```
print(user_df['LANGUAGE'].unique())
```

```
['es-419' 'en' nan]
```

Missing values:

```
print(user_df['LANGUAGE'].value_counts(dropna=False))
```

```
LANGUAGE
en        63403
NaN       30508
es-419     6089
Name: count, dtype: int64
```

The language es-419 represents Latin American Spanish, but might require clarification or mapping if other variants appear and the high number of missing values (~30% of users) limits localization or multilingual analysis.

Gender Values:

The number of unique entries in the Gender column is shown below:

```
print(user_df['GENDER'].unique()) #Find unique values to check for any spelling inconsistencies or formatting issues
```

```
['female' nan 'male' 'non_binary' 'transgender' 'prefer_not_to_say'
 'not_listed' 'Non-Binary' 'unknown' 'not_specified'
 "My gender isn't listed" 'Prefer not to say']
```

Diverse and inclusive set of gender values can be seen. However, case inconsistencies (Non-Binary, non_binary) and redundant expressions (prefer_not_to_say, Prefer not to say) mean missed normalization opportunities.

The number of entries for each gender type is shown below:

```
print(user_df['GENDER'].value_counts(dropna=False)) #Find the count of each occurence to check for any outliers
```

```
GENDER
female                   64240
male                     25829
NaN                       5892
transgender               1772
prefer_not_to_say         1350
non_binary                 473
unknown                    196
not_listed                 180
Non-Binary                  34
not_specified               28
My gender isn't listed       5
Prefer not to say            1
Name: count, dtype: int64
```

Categories like "My gender isn't listed" indicate free-text responses where a dropdown was likely expected. Standardizing these entries is crucial for any demographic segmentation or diversity reporting.

Summary:

While the dataset is structurally sound (no duplicate IDs or rows), the dataset contains high missingness, semantic inconsistencies and potential placeholder dates.

Challenging fields to understand could include Birth_Date and Created_Date are in string format with time zone suffixes (Z) — require parsing to datetime along with language values like es-419 (Latin American Spanish) may not be immediately intuitive and might need decoding or grouping.

Transaction Take Home Dataset:

This dataset contains 50,000 rows and captures receipt-level transaction data including receipt identifiers, purchase dates, scanned dates, store names, user links, barcodes, quantities, and final sale amounts.

Datatypes:

The datatypes of each column are presented below:

```
print(transaction_df.dtypes) # Show data types of each column
```

```
RECEIPT_ID         object
PURCHASE_DATE      object
SCAN_DATE          object
STORE_NAME         object
USER_ID            object
BARCODE           float64
FINAL_QUANTITY     object
FINAL_SALE         object
dtype: object
```

All columns except BARCODE are interpreted as object types in Python. This includes fields that should be numeric or datetime based on the ER diagram — such as Final_Quantity, Final_Sale, Purchase_Date, and Scan_Date while Barcode is interpreted as float64 which is also incorrect.

These issues prevent direct numerical calculations (e.g., summing total quantities or sale values), require manual conversion and validation before any time-based or financial analysis and increases the risk of misinterpretation or incorrect aggregations.

Specifically for the Barcode column interpreting them as floats causes issues such as precision Loss where long barcodes may lose digits or be rounded, leading to mismatches during joins with the product table and format confusion where barcodes displayed in scientific notation are not human-readable and can lead to incorrect reporting or manual handling errors.

Missing Values:

The total number of missing values for each column are presented below:

```
transaction_missing_values= transaction_df.isna().sum() #Sum all the missing values in each column
print(transaction_missing_values) #Print the missing values in each column
```

```
RECEIPT_ID           0
PURCHASE_DATE        0
SCAN_DATE            0
STORE_NAME           0
USER_ID              0
BARCODE           5762
FINAL_QUANTITY       0
FINAL_SALE           0
dtype: int64
```

The Barcode column has 5,762 missing values and may indicate system-generated receipts or partially scanned data. Additionally, without these values, it is not possible to tie these transactions to specific product details.

Empty Strings:

The total number of empty strings for each column are presented below:

```
transaction_empty_string = (transaction_df.astype(str).apply(lambda col: col.str.strip() == '')).sum() # #Sum all the empty strings in each column
print(transaction_empty_string) #Print the empty strings in each column
```

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

The Final_Sale column 12,500 rows contain blank strings (''), which are not the same as NaN. This prevents aggregation of revenue or analysis of average sales and must be cleaned and converted to NaN or 0.0 (if appropriate) to proceed with numerical operations.

Duplicate Values:

The total number of duplicate IDs are presented below:

```
transaction_duplicate_receipt_ids = transaction_df['RECEIPT_ID'].duplicated().sum() #Sum all the duplicate IDs
print(f"Number of duplicate IDs: {transaction_duplicate_receipt_ids}") #Print all the duplicate RECEIPT_IDs
```

```
Number of duplicate IDs: 25560
```

The total number of duplicate rows are presented below:

```
transaction_duplicate_rows= transaction_df.duplicated().sum() #Sum all the duplicate rows
print(f"Number of duplicate rows: {transaction_duplicate_rows}") #Print all the duplicate rows
```

```
Number of duplicate rows: 171
```

While Receipt_Id duplication may appear concerning at first, it is expected in this style of data, as one receipt often contains multiple products (one row per line item). This follows a one-to-many structure where Receipt_Id is repeated across rows with unique combinations of Barcode, Final_Quantity, and Final_Sale.

However, fully duplicated rows (same receipt, same product, same quantity, same sale) may point to ingestion errors or users rescanning receipts. These should be deduplicated in production systems to ensure accurate analytics and avoid double-counting rewards or revenue.

Unexpected String Values in Numeric Fields:

The number of unique values in the Final_Quantity column is presented below:

```
print(transaction_df['FINAL_QUANTITY'].unique()) #Find the unique values to check for format inconsistency
```

```
['1' 'zero' '2' '3' '4' '4.55' '2.83' '2.34' '0.46' '7' '18' '12' '5'
 '2.17' '0.23' '8' '1.35' '0.09' '2.58' '1.47' '16' '0.62' '1.24' '1.4'
 '0.51' '0.53' '1.69' '6' '2.39' '2.6' '10' '0.86' '1.54' '1.88' '2.93'
 '1.28' '0.65' '2.89' '1.44' '2.75' '1.81' '276' '0.87' '2.1' '3.33'
 '2.54' '2.2' '1.93' '1.34' '1.13' '2.19' '0.83' '2.61' '0.28' '1.5'
 '0.97' '0.24' '1.18' '6.22' '1.22' '1.23' '2.57' '1.07' '2.11' '0.48' '9'
 '3.11' '1.08' '5.53' '1.89' '0.01' '2.18' '1.99' '0.04' '2.25' '1.37'
 '3.02' '0.35' '0.99' '1.8' '3.24' '0.94' '2.04' '3.69' '0.7' '2.52'
 '2.27']
```

The Final_Quantity column contains the string value "zero" in several rows while the Final_Sale column contains many blank entries and this can cause significant issues when performing

numerical calculations. These should be cast to a numeric datatype to enable calculations while "zero" in the Final_Quantity column needs to be normalized to 0 or treated as an error depending on context.

Issue with Purchase_Date Column:

The first few and last few rows of the purchase date column are represented below:

```
print(transaction_df['PURCHASE_DATE']) #Print the rows of PURCHASE_DATE column
```

```
0       2024-08-21
1       2024-07-20
2       2024-08-18
3       2024-06-18
4       2024-07-04
           ...
49995   2024-08-21
49996   2024-08-11
49997   2024-07-11
49998   2024-06-18
49999   2024-08-07
```

The Purchase_Date column is defined as a datetime field in the ER diagram, but the Excel file only contains date-level granularity (i.e., no time of day). While this may be sufficient for basic analysis, it introduces a schema mismatch and limits use cases involving intra-day behavior, scan-to-purchase interval analysis, or fraud detection based on unusual purchase times. If time data is not available at source, the schema should reflect that it's a date field to avoid downstream confusion or default time imputation.

Summary:

The Transaction_Takehome.csv dataset captures item-level purchase details, with each row representing a product tied to a receipt. While duplicate Receipt_Ids are expected in this style of data (since one receipt can contain multiple products), the presence of fully duplicated rows suggests potential ingestion errors or re-submissions that may require deduplication. The purchase_date field, while defined as a datetime in the ER diagram, contains only date-level precision in the dataset, limiting time-based behavioral analysis.

Challenging fields to understand could include Final_Quantity and Final_Sale which are stored as strings due to Excel's general formatting, and contain inconsistent values such as "zero" and blank strings. These must be cleaned and converted to numeric types for accurate aggregation and analysis.

Additional quality concerns include the barcode field being typed as float64, which can lead to failed joins due to scientific notation or rounding. Overall, the dataset is structurally aligned with transactional systems but requires significant cleaning and normalization before it can be reliably used for analysis or modeling.

Products Take Home Dataset:

This dataset contains 100,000 rows with product-level metadata including hierarchical category fields, manufacturer, brand, and barcode. It is used to enrich transaction-level data with descriptive product attributes.

Data Types:

The datatypes of each column are represented below:

```
print(products_df.dtypes) # Show data types of each column
```

```
CATEGORY_1       object
CATEGORY_2       object
CATEGORY_3       object
CATEGORY_4       object
MANUFACTURER     object
BRAND            object
BARCODE         float64
dtype: object
```

All columns are interpreted as object (string) in Python, **except** the BARCODE column, which is read as float64.

As barcodes are identifiers, not numeric values representing them as floats leads to scientific notation, precision loss, and rounding, which can break joins between this dataset and transactional records. Although the ERD defines Barcode as an integer, using **object (string)** in practice is safer to preserve formatting exactly, avoid mathematical operations on identifiers and ensure compatibility with system-generated alphanumeric barcodes.

This column should be explicitly cast to string format upon reading the file to preserve its full value and formatting.

Missing values:

The missing values of each column are represented below:

```
products_missing_values= products_df.isna().sum() #Sum all the missing values in each column
print(products_missing_values) #Print the missing values in each column
```

```
CATEGORY_1         111
CATEGORY_2        1424
CATEGORY_3       60566
CATEGORY_4      778093
MANUFACTURER    226474
BRAND           226472
BARCODE           4025
```

Missing values in the category hierarchy indicate incomplete product classification, especially beyond Category_2. This makes it difficult to perform segmentation or analysis at the subcategory level. Additionally, a large number of products lack manufacturer and brand information, reducing the ability to analyze brand performance or manufacturer trends along with missing barcodes make it impossible to join some products with transactions, which can impact reward attribution and analytics coverage.

A viable solution is to backfill missing categories by leveraging barcode matches across more complete records and to impute missing manufacturer and brand fields using internal similarity rules. Collaboration with upstream data sources may also be necessary to improve future data completeness.

Empty Strings:

The number of empty strings in each column are represented below:

```
products_empty_string = (products_df.astype(str).apply(lambda col: col.str.strip() == '')).sum() #Sum all the empty strings in each column
print(products_empty_string) #Print the empty strings in each column
```

```
CATEGORY_1      0
CATEGORY_2      0
CATEGORY_3      0
CATEGORY_4      0
MANUFACTURER    0
BRAND           0
BARCODE         0
```

No empty string values were detected. This suggests that missing values are properly encoded as NaN, which simplifies cleaning.

Duplicate Values:

The number of duplicate barcodes is represented below:

```
products_duplicate_barcode= products_df['BARCODE'].duplicated().sum() #Sum all the duplicate barcodes
print(f"Number of duplicate barcodes: {products_duplicate_barcode}") #Print the duplicate barcodes
```

```
Number of duplicate rows: 4209
```

The number of duplicate rows is represented below:

```
products_duplicate_rows= products_df.duplicated().sum() #Sum all the duplicate rows
print(f"Number of duplicate rows: {products_duplicate_rows}") #Print all the duplicate rows
```

```
Number of duplicate rows: 215
```

A high number of rows **(**4,209) have the same BARCODE value along with 215 rows that are fully duplicate. Since BARCODE is the unique product identifier, duplication introduces ambiguity in joining product details to receipts. Exact duplicates suggest redundancy in the dataset and should be deduplicated during preprocessing to ensure consistency.

Category Hierarchy Validation:

No category hierarchical violations were found:

```
#Four category columns appear to be in a hierarchy.
#This code checks for hierarchical violations such as category 2 existing without category 1 and other similar scenarios

cat2_missing_cat1 = products_df[
    products_df["CATEGORY_2"].notna() & products_df["CATEGORY_1"].isna()
].shape[0]

cat3_missing_cat2 = products_df[
    products_df["CATEGORY_3"].notna() & products_df["CATEGORY_2"].isna()
].shape[0]

cat4_missing_cat3 = products_df[
    products_df["CATEGORY_4"].notna() & products_df["CATEGORY_3"].isna()
].shape[0]

print(f"CATEGORY_2 without CATEGORY_1: {cat2_missing_cat1}")
print(f"CATEGORY_3 without CATEGORY_2: {cat3_missing_cat2}")
print(f"CATEGORY_4 without CATEGORY_3: {cat4_missing_cat3}")
```

```
CATEGORY_2 without CATEGORY_1: 0
CATEGORY_3 without CATEGORY_2: 0
CATEGORY_4 without CATEGORY_3: 0
```

No instances were found where Category_2 exists without Category_1, or Category_4 exists without Category_3.

This indicates the structural hierarchy is preserved, but the depth of categorization is inconsistent due to missing values, particularly in Category_3 and Category_4. But analysts relying on full hierarchy (e.g., for drill-down dashboards) may find results skewed or sparse at deeper levels. Business rules may be needed to roll products with partial hierarchy into higher-level categories.

Summary:

The Products_Takehome.csv file contains metadata about products referenced in user transactions. Several key fields, including Category_3, Category_4, Manufacturer, and Brand, exhibit a high volume of missing values. While the category hierarchy is structurally intact (i.e., no subcategory appears without its parent), the sparsity in deeper category levels limits fine-grained product analysis. Additionally, the dataset includes over 4,200 duplicate barcodes and 215 fully duplicated rows, which could impact product-level aggregations or joins if not resolved.

The Barcode field is stored as float64 instead of a string, which introduces format issues—particularly when barcodes are displayed in scientific notation or rounded. This can lead to failed joins with transaction data. Overall, the product metadata offers essential contextual value for analytics but requires deduplication, missing value handling, and type correction to ensure integrity across linked datasets.

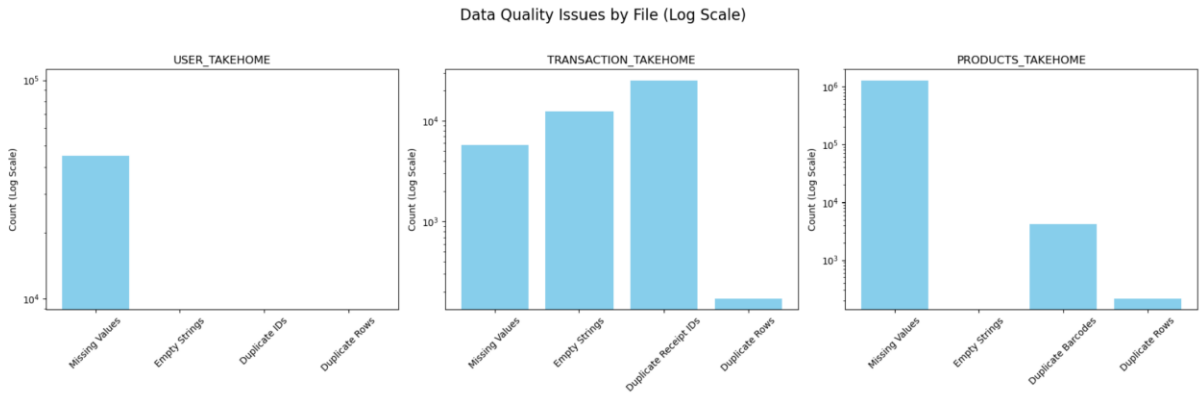
Visual Summary of data quality issues across the three datasets:

The visualization below provides a comparative overview of the most prominent data quality issues across the three datasets: User_Takehome, Transaction_Takehome, and Products_Takehome. The chart highlights four common quality dimensions for each file: missing values, empty strings, duplicate identifiers, and fully duplicated rows.

A bar chart was selected for this purpose due to its clarity and effectiveness in comparing discrete categories. It enables both technical and non-technical audiences to quickly assess which datasets are most affected by specific types of quality issues.

A logarithmic scale was applied to the y-axis to address the large variation in counts across the datasets. For instance, the number of missing values in Products_Takehome exceeds One million, whereas duplicate rows in User_Takehome are negligible. Without a log scale, smaller but still meaningful values would be visually minimized and potentially overlooked.

By using a base-10 log scale, the chart preserves the proportional differences while ensuring that all categories remain visible and comparable. This approach enables a more balanced and interpretable view of the overall data quality landscape.



Recommendations:

Enforce explicit data type conversion during ingestion to align with the ER diagram. Key fields such as Final_Quantity, Final_Sale, Purchase_Date, and Birth_Date should be cast to their intended numeric or datetime formats to enable reliable analysis and prevent logic errors.

Convert all barcode values to string format to avoid precision loss and rounding issues caused by float64 interpretation. This ensures successful joins between transactional and product datasets.

Standardize categorical fields, particularly Gender, by mapping similar or inconsistent entries (e.g., "non_binary", "Non-Binary", "Prefer not to say") into a unified set of categories to improve demographic reporting accuracy.

Deduplicate exact rows in both the transaction and product datasets to avoid metric inflation and redundancy in analysis. Retain multiple rows per receipt only when they represent distinct products.

Impute or enrich missing metadata in the Products_Takehome.csv file using internal cross-references (e.g., matching barcodes) or external product catalogs. Priority should be given to populating Category_3, Category_4, Manufacturer, and Brand.

Update the schema definition where necessary to reflect actual data granularity — for instance, if Purchase_Date consistently lacks timestamps, it may be more appropriate to define it as a date field instead of a datetime.

Implement quality checks upstream, especially for fields frequently affected by Excel's "General" format (e.g., dates and numeric fields), to minimize manual cleaning during analysis.

Conclusion:

This data quality audit highlights several issues across the three datasets, ranging from incorrect data types and missing values to formatting inconsistencies and duplication. While each dataset reflects core components of the data system — users, transactions, and product metadata — they require thoughtful cleaning, standardization, and validation before being used for analytics, reporting, or modeling.

By addressing these issues through consistent data typing, deduplication, and metadata enrichment, the datasets can serve as a reliable foundation for insights into user behavior, product performance, and reward dynamics. Implementing the recommended remediations will enhance data integrity and ensure that future analyses are both accurate and actionable.