Third: Evaluate Data Quality Issues in the Data Provided

1.) Data Issues with Raw data

After inspecting the data visually and attempting to parse the JSON file, I identified several issues:

- The data is not in a proper JSON format. Both receipts.json and brands.json contain multiple separate JSON objects for some fields, making it difficult to read the entire file at once.
- There are inconsistent data types. For example, the "total spent" field contains both strings and integers.
- Some data appears to be corrupted.
- Dates are stored in milliseconds and need to be converted to a standard time format for better readability.

I used the below python codes to sparse the data to get into a desired format to load into a database. This can also be found in the github directory as parse_1.py

```
import json
import pandas as pd
from datetime import datetime

def flatten_json(nested_json):
  flat_data = {}

  def flatten(obj, key=''):
      if isinstance(obj, dict):
          for k, v in obj.items():
              new_key = f"{key}.{k}" if key else k
              flatten(v, new_key)
      elif isinstance(obj, list):
          flat_data[key] = json.dumps(obj)
      else:
          flat_data[key] = obj

  flatten(nested_json)

  for k, v in list(flat_data.items()):
      if isinstance(v, (int, float)) and 'date' in k.lower():
          try:
              flat_data[k] = datetime.utcfromtimestamp(v /
1000).strftime('%Y-%m-%d %H:%M:%S')
          except ValueError:
              pass

  return flat_data
```

```
def load_json_file(file_path):
  data = []
  with open(file_path, 'r', encoding='utf-8') as file:
      for line in file:
          try:
              data.append(json.loads(line.strip()))
          except json.JSONDecodeError:
              print("Skipping invalid JSON line")
  return data

file_path = "receipts.json"
data = load_json_file(file_path)
flattened_data = [flatten_json(record) for record in data]

df = pd.DataFrame(flattened_data)
df.to_csv("Receipts_beta.csv", index=False)
```

The above code was used to parse all the json files. For Receipts I observed that rewardReceiptItemList had multiple objects which needed to be separated into different columns. I used the below python codes to sparse the data to get into a desired format to load into a database. This can also be found in the github directory as parse_2.py

```
import pandas as pd
import json

file_path = "Receipts_beta.csv"
df = pd.read_csv(file_path)
df["rewardsReceiptItemList"] = df["rewardsReceiptItemList"].apply(lambda x:
json.loads(x) if isinstance(x, str) else [])
df_exploded = df.explode("rewardsReceiptItemList")
df_expanded =
df_exploded.join(pd.json_normalize(df_exploded["rewardsReceiptItemList"],
sep="_"), rsuffix="_item")
df_expanded.drop(columns=["rewardsReceiptItemList"], inplace=True)
output_file = "Receipts_final.csv"
df_expanded.to_csv(output_file, index=False)
```

2.) Data issues with the structured data

1.) Identifying Columns with High Null or Blank Values

To identify columns with a high number of Null or blank values, we used the following query

```
SELECT bounus_point_earned, COUNT(*) AS null_count
FROM fact_Receipts
WHERE bounus_point_earned IS NULL OR bounus_point_earned = ' '
```

```
GROUP BY bounus_point_earned
```

By applying similar queries across various columns, we got significant insights into the data quality. We found that many Brand fields contain Null or blank values. Additionally, the Receipts data has numerous empty fields related to points and rewards. Similarly, ReceiptItems contains many empty fields. These missing or blank values need to be investigated to determine whether they were not captured or if they should be replaced with 0 or an appropriate placeholder.

<u>2.) Check for duplicate records</u>

To identify duplicate records, we ran the following query:

```
SELECT id, COUNT(*) AS duplicate_count
FROM dim_Users
GROUP BY id
HAVING COUNT(*) > 1
```

This query flagged any id that appeared multiple times. We found that while Receipts and Brands did not have duplicate records, the Users table contained a significant number of duplicates. This needs further investigation to determine if they are valid duplicates or require deduplication.

<u>3.) Validating dates</u>

To ensure date integrity, we checked for cases where the purchase_date was later than the modify_date using:

```
SELECT *
FROM dim_Receipts
WHERE purchase_date > modify_date
```

This query helped identify records with logically incorrect dates. We found some cases where the Purchase Date was after the Modify Date, which should not be possible. Similar checks can be applied to other date fields to ensure logical consistency.

<u>4.) Logical consistency check</u>

To verify logical consistency, we checked whether the Final Price was greater than the Item Price in the factReceiptsItems table

```
SELECT receipts_item_id, final_price, item_price
```

```
FROM factReceiptsItems
WHERE final_price > item_price
```

This query highlighted cases where the final price exceeded the item price, which is illogical unless discounts or adjustments were applied incorrectly. This needs further validation to ensure accurate pricing data.

By conducting these data validation checks, we identified key areas requiring attention, including missing values, duplicate records, incorrect date sequences, and logical inconsistencies.