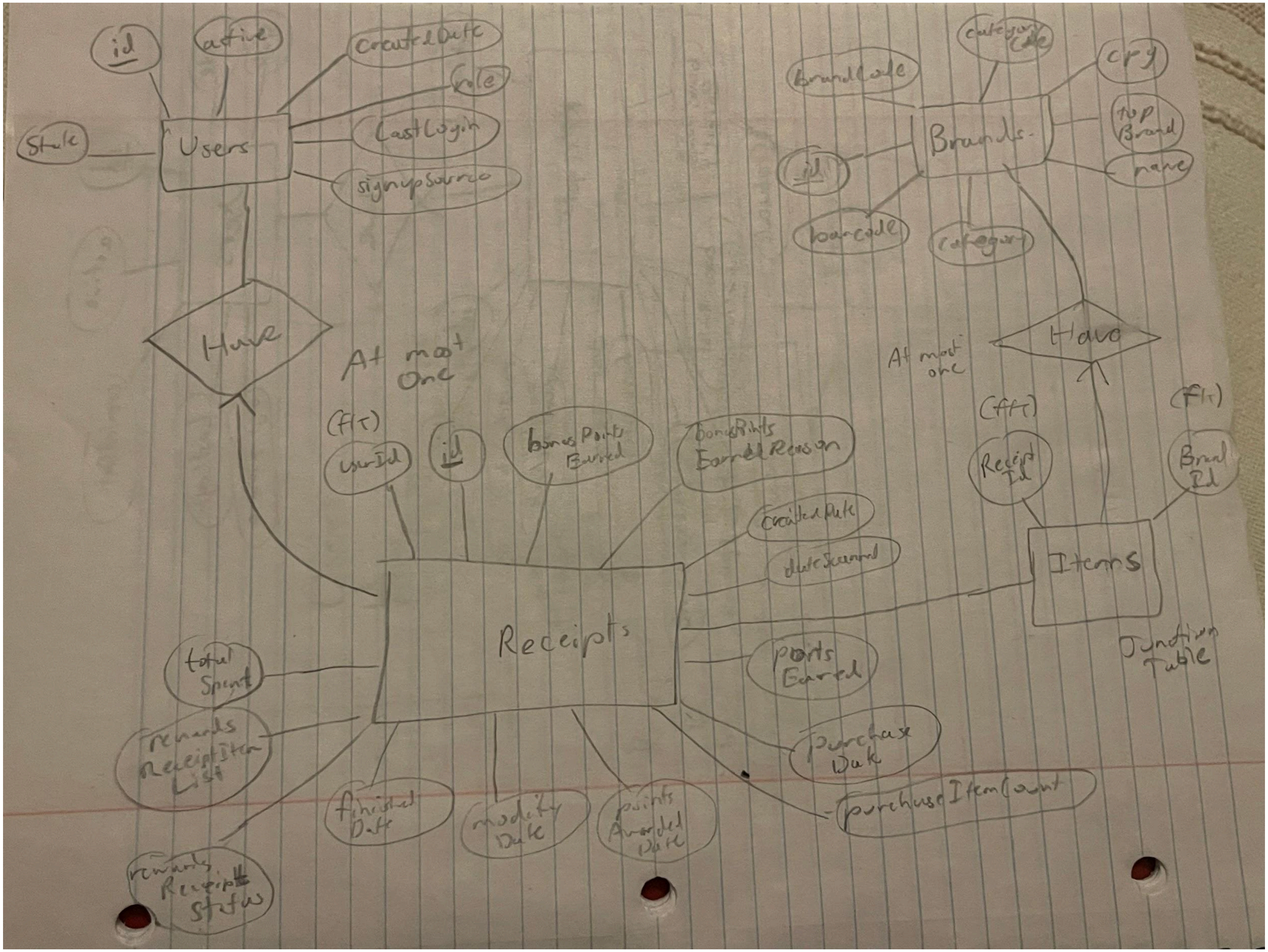
1) Used Mysql


Users
id
active
CreatedDate
role
LastLogin
signupSource
State
Have
At most One
(FK)
userId
id
bonusPointsEarned
bonusPointsEarnedReason
createdDate
dateScanned
Receipts
pointsEarned
purchaseDate
purchasedItemCount
totalSpent
rewardsReceiptItemList
rewardsReceiptStatus
finishedDate
modifyDate
pointsAwardedDate
Brands
brandCode
categoryCode
cpg
topBrand
name
id
barcode
category
Have
At most one
(FK)
ReceiptId
(FK)
BrandId
Items
Junction Table

```
CREATE TABLE Receipts (
    id VARCHAR(36) PRIMARY KEY,
    purchaseDate DATETIME NOT NULL
);

CREATE TABLE Brands (
    id VARCHAR(36) PRIMARY KEY,
    name VARCHAR(255) NOT NULL
);

CREATE TABLE ReceiptItems (
    receiptId VARCHAR(36) NOT NULL,
    brandId VARCHAR(36) NOT NULL,
    itemId INT PRIMARY KEY,
    quantity INT NOT NULL,
    FOREIGN KEY (receiptId) REFERENCES Receipts(id),
    FOREIGN KEY (brandId) REFERENCES Brands(id)
);

CREATE TABLE Users (
    _id VARCHAR(24) PRIMARY KEY,
    state CHAR(2) NOT NULL,
    createdDate TIMESTAMP NOT NULL,
    lastLogin TIMESTAMP,
    role VARCHAR(20) NOT NULL DEFAULT 'CONSUMER',
    active BOOLEAN NOT NULL DEFAULT TRUE
);

CREATE TABLE Brands (
    _id UUID PRIMARY KEY,
 barcode VARCHAR(50) NOT NULL,
  brandCode VARCHAR(50) NOT NULL,
    category VARCHAR(100) NOT NULL,
    categoryCode VARCHAR(50) NOT NULL,
    cpg UUID,
    topBrand BOOLEAN NOT NULL DEFAULT FALSE,
    name VARCHAR(255) NOT NULL
);
```

- What are the top 5 brands by receipts scanned for most recent month?

```
SELECT
   b.name AS brandName, COUNT(DISTINCT ri.receiptId) AS receiptCount,
   RANK() OVER (ORDER BY COUNT(DISTINCT ri.receiptId) DESC) AS rank
FROM Receipts r
JOIN ReceiptItems ri ON r.id = ri.receiptId
JOIN Brands b ON ri.brandId = b.id
WHERE
   r.purchaseDate >= '2025-01-01'    AND r.purchaseDate < '2025-02-01'
GROUP BY
   b.name
ORDER BY
   Rank
LIMIT 5;
```

Using with, I created a table called recent receipts where we filter the most recent month through the WHERE statement, so current date is greater than start of most recent month and less and end of most recent month. Then I used an Inner join for ReceiptItems and RecentReceipts on id. I then used a INNER JOIN for Brands to match brand id with Receipt Items id. Then I Groups by name and intended to count distinct receipt IDs (r.id) for each brand.

3)

- When considering *average spend* from receipts with 'rewardsReceiptStatus' of 'Accepted' or 'Rejected', which is greater?

```
SELECT
   rewardsReceiptStatus,
   AVG(totalSpent) AS avgSpend
FROM Receipts
WHERE rewardsReceiptStatus IN ('Accepted', 'Rejected')
GROUP BY rewardsReceiptStatus;
```

Used Aggregate Function Average for totalSpent along with the rewardsReceipt Status, then filtered on Accepted and Rejected and Group by the rewardsReceiptStatus. Compare the results

4)

- When considering *total number of items purchased* from receipts with 'rewardsReceiptStatus' of 'Accepted' or 'Rejected', which is greater?

```
SELECT
    rewardsReceiptStatus, SUM(purchasedItemCount) AS totalItems
FROM  Receipts
WHERE
    rewardsReceiptStatus IN ('Accepted', 'Rejected')
GROUP BY
    rewardsReceiptStatus
ORDER BY
    totalItems DESC;
```

Using same logic as about, this time I calculated for aggregate function Sum instead of average, then compare the results between accepted and rejected.

```
5)WITH sixmonthusers AS (
    SELECT id
    FROM Users
    WHERE createdDate >= DATE_ADD(CURDATE(), INTERVAL -6 MONTH)
)
SELECT
    b.name AS brandName,
    SUM(r.totalSpent) AS totalSpend
FROM
    RecentUsers u
INNER JOIN
    Receipts r ON u.id = r.userId
INNER JOIN
    ReceiptItems ri ON r.id = ri.receiptId
INNER JOIN
    Brands b ON ri.brandId = b.id
GROUP BY
    b.name
ORDER BY
    totalSpend DESC
LIMIT 1;
```

This query finds the top brand by total spending among users who created their accounts in the last 6 months. It initially finds users from the Users who joined within the last 6 months through the 6 month interval and based on user id in the sixmonthusers table(sixmonthusers CTE). After that, we do an inner Join on users id and Receipts id then on ReceiptsItems and Receipts based on id, and it selects the aggregate function sum for totalspent.Finally it arranges the results in descending order, and returns the brand with the highest total spending.

3)

Invalid JSON formatting in the rewardsReceiptItemList column was one of the issues I ran across when loading data into SQL. Due to a syntax issue, PostgreSQL rejected the insert operation because the data contained unescaped double quotes within values, such as "rewardsGroup": "CAPRI SUN ROARIN" WATERS BEVERAGE DRINK." WATERS BEVERAGE DRINK is an invalid token since the database considered the quote that follows ROARIN to be the end of the string. Using re.sub(r'(?<=\w)"(?=\w)', r'\"', text), I created a Python script to address this, making sure that double quotes within values were appropriately escaped. Before performing the SQL import, I also used json.loads() to validate each record and identify any formatting errors. I was able to sanitize the JSON data while maintaining its original structure thanks to this method.

Another issue was another problem I ran into when transforming the dataset before putting it into SQL. In particular, I first wrote receipts['rewardsReceiptItemList'] = receipts['rewardsReceiptItemList'] while attempting to change specific values in rewardsReceiptItemList.resulting in a syntax error: apply(lambda x: if x == 'N' then 'NULL' else x). Python lambda functions do not use the if-then syntax; instead, they should be written as lambda x: 'NULL' if x == 'N' else x. After resolving this, I ran into another problem where some fields had NaN values. This resulted in type mismatches that generated issues while inserting into PostgreSQL. To fix this, I made sure that missing values were handled explicitly as NULL in SQL by using df.fillna('NULL') before to exporting the cleaned data.

When I tried to write the cleaned data to a new CSV file, I also ran into issues with erroneous JSON escaping. I first tried using json.dumps(data) to escape double quotes.replace('"', '""'), making the JSON that was inserted into PostgreSQL over-escaped. My method was to convert all double quotes to doubled quotes (""), which is correct for CSV but not for JSONB columns in PostgreSQL. The database required appropriately escaped quotes (\" inside JSON values). When PostgreSQL tried to parse the JSON data, this resulted in problems. In order to correct this, I changed the script to apply json.dumps(data) to escape just internal quotes within JSON values while maintaining the outside structure, making sure that JSON serialization was done correctly before publishing to the CSV.

When exporting to CSV using csv.QUOTE_ALL, over-quoting fields caused another problem. When inserting into PostgreSQL, this resulted in unanticipated behavior where some columns had excessive quotations, producing syntax errors, even though it guaranteed that every value was encased in quotes. The CSV writing rules resulted in extra quotes even if the JSON fields were already structured correctly. This was particularly troublesome because PostgreSQL's JSONB columns require correct JSON input, and any more quotes would cause the processing to fail. I changed the script to only conditionally apply json.dumps(data) when required (for JSON-like fields) and kept other fields unchanged in order to fix this. Before importing, I also checked the output with json.loads() to make sure each row had properly structured JSON. This helped me successfully load data into the database.

A comprehensive Excel cleanup was carried out for the users and brand schemas, with an emphasis on improving the data's integrity and structure for analysis. In order to ensure that text and numerical fields were appropriately categorized and prevent potential misinterpretations during analysis, the method started with rectifying data types across different columns. To make the dataset more manageable, unnecessary columns were eliminated, and missing values were carefully filled in or eliminated to maintain the analysis's correctness.

To further prevent hassles from format conversions, the date columns' original text format was kept, and problems with text encoding were fixed to guarantee that all data was correctly presented and read. To ensure the dataset's dependability, a last stage of data validation was carried out to look for anomalies or unexpected numbers. The result of these efforts was a consistent and cleansed CSV file that was prepared for accurate and perceptive data analysis.

4)
To Fetch Team,

We ran into a number of data quality issues that affected accuracy and ingestion as we were putting the rewards receipt data into our database. The rewardsReceiptItemList column has a significant problem with improper JSON formatting. PostgreSQL rejected the data because of misplaced double quotes (e.g., "ROARIN" WATERS BEVERAGE DRINK"). In order to fix this, we used json.loads() and regex-based adjustments to automate JSON validation, guaranteeing correct escaping before to ingestion. Furthermore, when altering award receipt elements, we stumbled into transformation errors—more especially, improper syntax when substituting values with Python lambda functions. In order to avoid type incompatibilities, we fixed the transformation logic and explicitly handled missing values as NULL in SQL.This was fixed by utilizing json.dumps() to ensure correct JSON serialization, maintaining a legitimate JSON structure for seamless data intake.

We discovered discrepancies in CSV formatting and data integrity across the Users and Brands datasets in addition to JSON-related problems. We improved the export procedure to selectively apply JSON escaping only where required because using csv.QUOTE_ALL resulted in excessive quotation marks when exporting to CSV, which interfered with PostgreSQL's JSON parsing. A thorough data validation and cleansing was also necessary due to missing numbers, inconsistent data types, and text encoding problems. To increase data integrity and analytical readiness, we standardized column formats, eliminated superfluous fields, and made sure missing values were appropriately categorized. We were able to produce a clean, organized dataset that is dependable for reporting and business insights thanks to these enhancements.

We would like to agree on a common format for rewardsReceiptItemList JSON fields in order to guarantee long-term data quality and avoid discrepancies in subsequent data loads. It would also be beneficial to verify whether our changes need to incorporate any business-specific guidelines for dealing with NULL values or missing brand information. We predict that querying JSON fields in PostgreSQL will have performance limitations as the dataset grows. To increase retrieval speed, we advise implementing optimal indexing on frequently searched variables (such as purchaseDate and brandId). In the future, automating JSON formatting and ETL validation prior to intake would improve scalability and decrease human error, enabling us to process bigger datasets more effectively. If you would want to talk about these findings in greater detail, please let us know.

Best,

Anubhav Darisi