Chenault Analysis
February 13, 2025

Requirement 1: Inspect the provided datasets and run the provided Extract_Amazon_Appliances_Reviews_data.ipynb.script. Comment on the different fields (data types, NaN values, etc.) in the created CSV files (Amazon_Appliances_Metadata.csv and Amazon_Appliances_Reviews.csv) and what data preprocessing is needed to prepare the data to be loaded into a database. The asin - ID of the product is the common field between the two tables, do you think that is sufficient to cross-reference the data entries/rows in the two datasets? Explain.

The Amazon_Appliances_Metadata.csv file includes asin (the product ID), price (a float), description (which has some missing values), and numerous other fields. The missing values in key attributes like price and title need data cleaning, like filling missing prices with a category-based median/mean or dropping rows with inflated null values. The categories column needs normalization because stowing hierarchical lists immediately in a relational database is inefficient. Instead, categories should be pulled into distinct columns or maintained in a separate category mapping table.

The Amazon_Appliances_Reviews.csv dataset possesses customer review data, with critical fields such as reviewerID (a string uniquely identifying users), asin (a foreign key linking reviews to products), reviewerName, helpful, reviewText (needing standardization to terminate special characters and HTML tags), and other fields. It is imperative to validate that all asin values in the reviews table exist in the metadata table and to remove or individually handle products that lack metadata entries to guarantee referential integrity. Structuring this data into a well-normalized relational database, with individual tables for products, reviews, categories, and related products, delivers efficient repository, querying, and examination while preserving data consistency and integrity.

Requirement 2: Create a table for a sample list of 3 queries (NOT listed in Requirements Specification Document) for every category listed below:
o Quantitative Queries
o Fuzzy/Exact-Search Queries
o Hybrid-Search Queries

Type	Query
Quantitative	How many products have an average review rating below 2.5?
Quantitative	How many reviewers have written at least 3

	reviews in the last 12 months?
Quantitative	What is the average number of reviews with a 4.5-star rating per product category?
Fuzzy/Exact-Search	Find reviews that mention “easy” but do not include “again.”
Fuzzy/Exact-Search	Retrieve reviews where users misspelled “Refrigerator” and rated with 3.5 stars or below.
Fuzzy/Exact-Search	Get a list of reviews where the reviewer mentioned “bad quality” within five words of “expensive”.
Hybrid-Search	Find reviews most similar to “The disposal worked and was installed in less time than expected.”
Hybrid-Search	Identify reviews that are similar to “After a couple years the unit started freezing up,” but only for products with more than 50 reviews.
Hybrid-Search	Retrieve reviews that are semantically related to “The oven heats unevenly” and were written in 2020 or later.

Requirement 3: Use ChatGPT and enter the CoT text (Section 5, Page 18 of Final Project - Requirements Specification.pdf) into ChatGPT prompt to get the recommended database that best meets the needs of every query you identified above. Create a table that has two columns: the query and the recommended database for it as printed by ChatGPT.

Here is the recommended database system for each query based on the steps you provided:

Query	Recommended Database
How many products have an average review rating below 2.5?	PostgreSQL
How many reviewers have written at least 3 reviews in the last year?	PostgreSQL
What is the average number of reviews with a 4.5-star rating per product category?	PostgreSQL
Find reviews that mention “easy” but do not include “again.”	Elasticsearch
Retrieve reviews where users misspelled “Refrigerator Parts” and rated with 3.5 stars or below.	Elasticsearch
Get a list of reviews where the reviewer mentioned “bad quality” within five words of “expensive.”	Elasticsearch
Find reviews most similar to “The disposal worked and was installed in less time than expected.”	Milvus
Identify reviews that are similar to “After a couple of years, the unit started freezing up,” but only for products with more than 50 reviews.	Milvus
Retrieve reviews that are semantically related to “The oven heats unevenly” and were written in 2020 or later.	Milvus

Requirement 4: Use ChatGPT to generate code for every query you identified above using the recommended database. Please see sample prompts and output in Appendix A of this document.

My input:

- I have an Amazon product reviews dataset. Use the PostgreSQL database system and create code for the query: How many products have an average review rating below 2.5?
- I have an Amazon product reviews dataset. Use the PostgreSQL database system and create code for the query: How many reviewers have written at least 3 reviews in the last year?
- I have an Amazon product reviews dataset. Use the PostgreSQL database system and create code for the query: What is the average number of reviews with a 4.5-star rating per product category?
- I have an Amazon product reviews dataset. Use the Elasticsearch database system and create code for the query: Find reviews that mention “easy” but do not include “again.”
- I have an Amazon product reviews dataset. Use the Elasticsearch database system and create code for the query: Retrieve reviews where users misspelled “Refrigerator Parts” and rated with 3.5 stars or below.

- I have an Amazon product reviews dataset. Use the Elasticsearch database system and create code for the query: Get a list of reviews where the reviewer mentioned “bad quality” within five words of “expensive.”
- I have an Amazon product reviews dataset. Use the Milvus database system and create code for the query: Find reviews most similar to “The disposal worked and was installed in less time than expected.”
- I have an Amazon product reviews dataset. Use the Milvus database system and create code for the query: Identify reviews that are similar to “After a couple of years, the unit started freezing up,” but only for products with more than 50 reviews.
- I have an Amazon product reviews dataset. Use the Milvus database system and create code for the query: Retrieve reviews that are semantically related to “The oven heats unevenly” and were written in 2020 or later.

ChatGPT Output:

I'll provide the SQL queries for PostgreSQL, the Elasticsearch DSL queries, and the Milvus queries accordingly.

PostgreSQL Queries

1. How many products have an average review rating below 2.5?

```
SELECT COUNT(product_id)
FROM (
   SELECT product_id, AVG(rating) AS avg_rating
   FROM reviews
   GROUP BY product_id
   HAVING AVG(rating) < 2.5
) subquery;
```

2. How many reviewers have written at least 3 reviews in the last year?

```
SELECT COUNT(reviewer_id)
FROM (
   SELECT reviewer_id, COUNT(*) AS review_count
   FROM reviews
   WHERE review_date >= NOW() - INTERVAL '1 year'
   GROUP BY reviewer_id
   HAVING COUNT(*) >= 3
) subquery;
```

3. What is the average number of reviews with a 4.5-star rating per product category?

```
SELECT category, AVG(review_count) AS avg_reviews
FROM (
   SELECT category, product_id, COUNT(*) AS review_count
   FROM reviews r
   JOIN products p ON r.product_id = p.product_id
   WHERE rating = 4.5
   GROUP BY category, product_id
) subquery
GROUP BY category;
```

Elasticsearch Queries

4. Find reviews that mention “easy” but do not include “again.”

```
{
 "query": {
   "bool": {
     "must": {
       "match": {
         "review_text": "easy"
       }
     },
     "must_not": {
       "match": {
         "review_text": "again"
       }
     }
   }
 }
}
```

5. Retrieve reviews where users misspelled “Refrigerator Parts” and rated with 3.5 stars or below.

```
{
 "query": {
   "bool": {
```

```
      "must": [
        {
          "match": {
            "review_text": {
              "query": "Refrigerator Parts",
              "fuzziness": "AUTO"
            }
          }
        },
        {
          "range": {
            "rating": {
              "lte": 3.5
            }
          }
        }
      ]
    }
  }
}
```

6. Get a list of reviews where the reviewer mentioned “bad quality” within five words of “expensive.”

```
{
  "query": {
    "span_near": {
      "clauses": [
        { "span_term": { "review_text": "bad quality" } },
        { "span_term": { "review_text": "expensive" } }
      ],
      "slop": 5,
      "in_order": false
    }
  }
}
```

Milvus Queries

7. Find reviews most similar to “The disposal worked and was installed in less time than expected.”

```
from pymilvus import Collection
import numpy as np

collection = Collection("reviews")
query_vector = embed("The disposal worked and was installed in less time than expected.")

search_params = {"metric_type": "IP", "params": {"nprobe": 10}}

results = collection.search(
    data=[query_vector],
    anns_field="review_embedding",
    param=search_params,
    limit=10,
    output_fields=["review_text"]
)

for result in results[0]:
    print(result.entity.get("review_text"))
```

8. Identify reviews that are similar to “After a couple of years, the unit started freezing up,” but only for products with more than 50 reviews.

```
query_vector = embed("After a couple of years, the unit started freezing up.")

search_results = collection.search(
    data=[query_vector],
    anns_field="review_embedding",
    param={"metric_type": "IP", "params": {"nprobe": 10}},
    limit=10,
    output_fields=["review_text", "product_id"]
)

filtered_results = []
for result in search_results[0]:
    product_id = result.entity.get("product_id")

    # Check if the product has more than 50 reviews
    review_count = collection.query(
```

```
        expr=f"product_id == '{product_id}'",
        output_fields=["count(*)"]
    )[0]["count(*)"]

    if review_count > 50:
        filtered_results.append(result.entity.get("review_text"))

print(filtered_results)
```

9. Retrieve reviews that are semantically related to “The oven heats unevenly” and were written in 2020 or later.

```
query_vector = embed("The oven heats unevenly.")

search_results = collection.search(
    data=[query_vector],
    anns_field="review_embedding",
    param={"metric_type": "IP", "params": {"nprobe": 10}},
    limit=10,
    output_fields=["review_text", "review_date"]
)

filtered_reviews = [
    result.entity.get("review_text")
    for result in search_results[0]
    if result.entity.get("review_date") >= "2020-01-01"
]

print(filtered_reviews)
```

This covers the required queries for PostgreSQL, Elasticsearch, and Milvus. Let me know if you need modifications or explanations! 🚀

Requirement 5: (10 Points) Analyze, synthesize, and comment on the code (Postgres SQL statements) generated by ChatGPT for every Quantitative Query you identified above for completeness, correctness, and consistency.

The PostgreSQL queries developed by ChatGPT for the quantitative queries effectively grasp the intended data retrieval tasks, but they have some areas for improvement concerning totality, correctness, and consistency. The first query, which calculates the number of products with an average review rating below 2.5, is structured well, using a subquery to calculate the average rating per product before filtering results in the outer query. However, COUNT(DISTINCT product_id) should be explicitly used in the outer query to bypass counting duplicate product IDs when using a subquery.

The second query, which specifies how many reviewers have written at least three reviews in the last year, correctly applies NOW() - INTERVAL '1 year' to filter reviews from the past year. The use of GROUP BY reviewer_id and HAVING COUNT(*) >= 3 is suitable, but COUNT(DISTINCT reviewer_id) should be used to guarantee unique reviewer IDs are measured. Further, if duplicate review entries exist for the identical reviewer, adding DISTINCT in the subquery enhances precision.

The third query calculates the average number of 4.5-star reviews per product category using a JOIN between the reviews and product tables. This approach is promising for retrieving product categories. However, the nested subquery approach adds unnecessary complexity. It would be more efficient to compute the category-level aggregation directly. Also, guaranteeing that the category is indexed and handled correctly when grouping improves performance. Altogether, the queries align with the most valuable practices for relational database operations. These modifications would guarantee the database functions optimally while maintaining query accuracy.