UC San Diego

Amazon Product Co-purchasing Analysis

Hao Wang, Tianxiang Wang, Tianhao Zhou

Mar 17, 2025

Introduction

- The rise of e-commerce platforms like Amazon has created vast product networks with complex co-purchasing relationships. Traditional databases struggle with these interactions.
- Our project integrates the Amazon Meta Dataset into a hybrid SQL + Graph + NoSQL system to analyze product influence, enhance recommendations, and leverage graph-based approaches for deeper insights.

Amazon Product Recommendation System

UC San Diego

- **What is the Amazon product recommendation system?**
  - The Amazon recommendation system suggests products based on customer behavior, past purchases, and purchasing history.
- **Why do we need the Amazon recommendation system?**
  - It helps customers find products easily, improves their shopping experience, and increases sales.
- **How does the Amazon recommendation system work?**
  - Product data is stored in SQL, co-purchasing patterns are analyzed in graphs, and fast recommendations are cached in NoSQL.

Dataset-amazon products

Amazon Product Co-Purchasing Network Metadata

- **Source:**
  - Data crawled from the Amazon website during the summer of 2006.
- **Structure & Content:**
  - **Product Information:** Title, ASIN, and product group (e.g., Books, Music CDs, DVDs, Videos).
  - **Sales Data:** Amazon sales rank.
  - **Network Connections:** List of similar products (i.e., products that are co-purchased).
  - **Categorization:** Detailed hierarchical product categories with corresponding category IDs.
  - **Review Information:** Includes review date, customer identifier, rating, number of votes, and helpfulness count.
- **Role in Application:**
  - Establishes a co-purchasing network among products.
  - Supports market analysis and evaluation of product performance.

Relational Database - Pre Processing

Explored relational dataset(PostgreSQL): Identified columns (**ASIN, Title, Product Group, Sales Rank, Categories, Similar Products**).

- **Data Cleaning:**
  - Removed **duplicates**.
  - Handled **missing values** (e.g., filtering out `NULL` titles, categories).
  - **Formatted categories** for easier querying.
  - **Excluded invalid rankings (-1)** in queries(Discontinued Products).
- **Schema Design in PostgreSQL:**
  - Created **`amazon_products`** table for product metadata.
  - **Indexed `ASIN`** for faster lookups.
- **Imported data using `COPY` command:**
  - Loaded CSVs into PostgreSQL (`\COPY` used for permission issues).

Relational Database

UC San Diego

- **Key Queries to gain insights of the data:**

🔍 **Find top-selling book:**

```sql
--find best selling products--
SELECT title, sales_rank
FROM amazon_products
WHERE sales_rank > 0  -- Excludes -1 (invalid rankings)
ORDER BY sales_rank ASC
LIMIT 10;
```

	title	sales_rank
1	The War of the Worlds	1
2	Shirley Valentine	2
3	Leslie Sansone - Walk Away the Pounds - Super Fat Burning	6
4	Robin Hood - Men in Tights	7
5	Richard Simmons - Sweatin' to the Oldies	8
6	Howard the Duck	12

🔍 **Find products with many similar co-purchases:**

```sql
--Find products with high purchasing rate
SELECT asin, title,
       LENGTH(similar_products) - LENGTH(REPLACE(similar_products, '|', '')) + 1 AS similar_count
FROM amazon_products
WHERE similar_products IS NOT NULL AND similar_products <> ''
ORDER BY similar_count DESC
LIMIT 10;
```

	asin	title	similar_count
1	1893115542	Data Mining & Statistical Analysis Using SQL	5
2	0691020841	The Camphor Flame	5
3	091912366X	The Mysterium Lectures: A Journey Through C.G. Ju…	5
4	0312981252	Bad Boy: The Murderous Life of Kenneth Allen McDu…	5
5	0064420280	Stanley and the Magic Lamp	5

🔍 **Find products with low co-purchase rate:**

```sql
--Find Products with Low Co-Purchases--
SELECT asin, COUNT(*) AS co_purchase_count
FROM amazon_products
WHERE similar_products IS NOT NULL
GROUP BY asin
ORDER BY co_purchase_count ASC;
```

	asin	co_purchase_count
1	1569715076	1
2	0963290614	1
3	0802773613	1
4	0806522178	1
5	0715307975	1

Graph Database- Data Preprocessing

UC San Diego

Relationships

Product – CO_PURCHASED_WITH – Another Product

Product – BELONGS_TO – Category

```sql
SELECT asin AS product_asin,
       UNNEST(string_to_array(similar_products, '|')) AS similar_asin
FROM amazon_products
WHERE similar_products IS NOT NULL
  AND similar_products <> '';
```

(asin, similar_asin)
(<that product>, 0804215715)
(<that product>, 156101074X)
(<that product>, 0687023955)

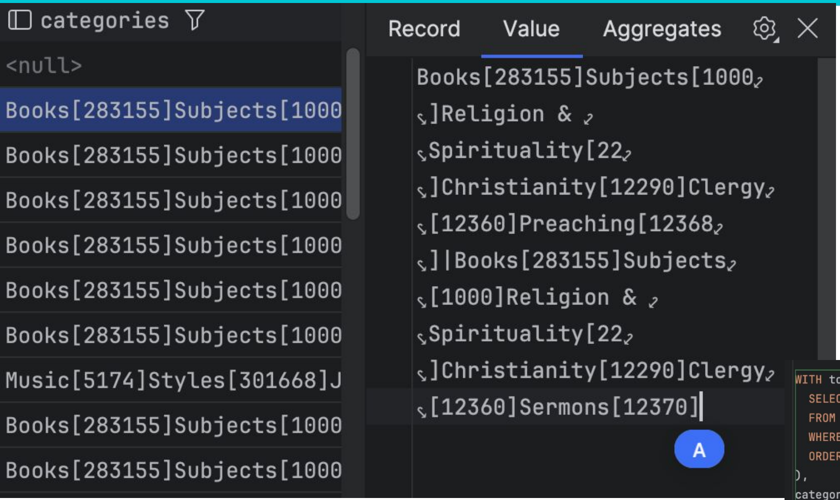
similar_products
1559360968|1559361247|1559360828|155936101…
1585741485|0140246967|1557504288|037420518…
0071410546|1580531784|1578200326|B0000A2W55
B000059QC1|B00000JQIE|B00029J1X6|B0006TR06…
0785114572|078511078X|0785114033|078511404…
0865778973|0071343547|0721662854|072168173…
B00007JMD8|6305350221|B00004RF9B|B00005JKF…
B00000JCDS|B000004CSZ|B00016XN6Q|B00005LLY…

Belongs-To

UC San Diego

–Redundancy

–Missing a clear delimiter



–Multiple IDs for the same category name

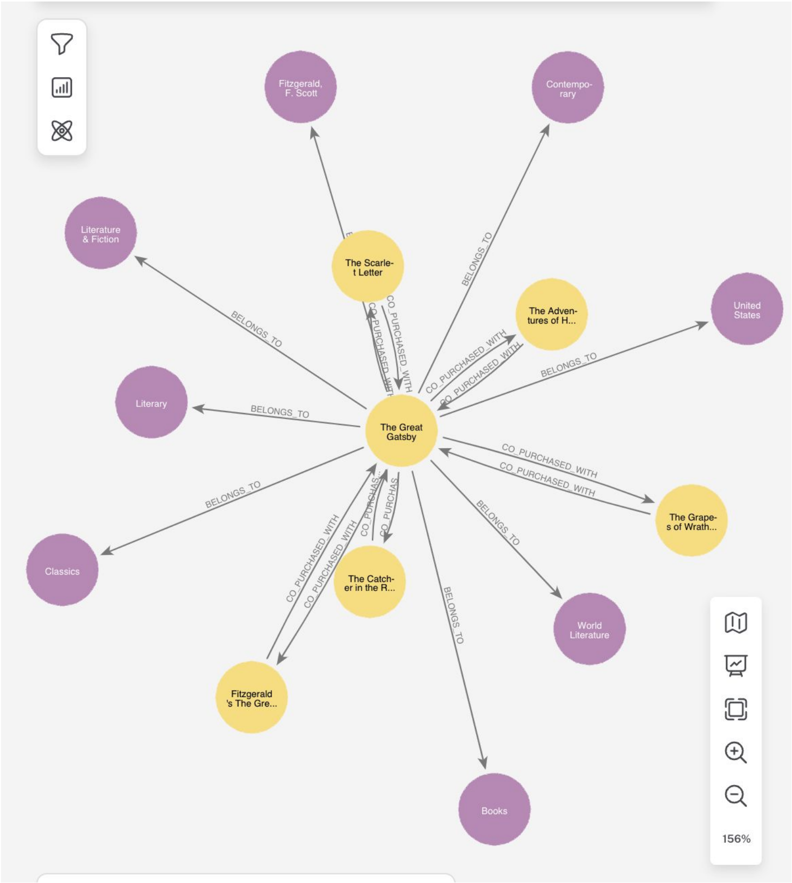
–Overly generalized categories

```sql
WITH top_products AS (
  SELECT asin
  FROM amazon_products
  WHERE sales_rank IS NOT NULL AND sales_rank > 0
  ORDER BY sales_rank ASC
),
category_names AS (
  SELECT DISTINCT
    sub.asin AS product_asin,
    TRIM(BOTH ' ' FROM regexp_replace(sub.cat_chunk, '\[.*$', '')) AS category_name
  FROM (
    SELECT
      p.asin,
      UNNEST(
        string_to_array(
          regexp_replace(p.categories, '\]', ']|', 'g'),
          '|'
        )
      ) AS cat_chunk
    FROM amazon_products p
    JOIN top_products tp ON p.asin = tp.asin
    WHERE p.categories IS NOT NULL
      AND p.categories <> ''
  ) sub
  WHERE sub.cat_chunk <> ''
    AND TRIM(BOTH ' ' FROM regexp_replace(sub.cat_chunk, '\[.*$', '')) <> ''
    AND TRIM(BOTH ' ' FROM regexp_replace(sub.cat_chunk, '\[.*$', '')) NOT IN ('General', 'Genres', 'Subjects', 'Categories', 'Reference', 'Formats', 'Authors, A-Z')
    AND TRIM(BOTH ' ' FROM regexp_replace(sub.cat_chunk, '\[.*$', '')) !~ '^\(\s*[A-Z]\s*\)$'
)
SELECT
  product_asin,
  category_name
FROM category_names;
```

Graph Database

PageRank

Louvain Community Detection



Key-value Database

- **Objective**:
  - Store frequently accessed product attributes in Redis for rapid lookups, reducing MongoDB load.

- **Implementation**:
  - **Cached Attributes**: Product metadata (ASIN, title, category, sales rank, reviews).
  - **Key Patterns**:
    - top_products:{category}:limit_{limit}: Top-selling products (e.g., 10 best by sales rank).
    - product:{asin}: Individual product details (e.g., "0738700797").
    - most_reviewed:limit_{limit}: Products with most reviews (e.g., Harry Potter editions).
    - promoted:{asin}: Promoted products (e.g., "Candlemas: Feast of Flames").

- **Results**:
  - **Sample Data**:
    - Top Sales: "Vehicular Technology..." (ASIN: 0780357213).
    - Most Reviewed: "Harry Potter..." (ASIN: 043936213X, 4995 reviews).
    - Promoted: "Candlemas..." (ASIN: 0738700797).

```python
### 8️⃣ Promoted Products Handling ###
def add_promoted_product(asin, promotion_reason):
    """Add a product to the promoted list"""
    try:
        product = products_collection.find_one({"asin": asin})
        if not product:
            print(f"❌ Product {asin} not found!")
            return
        promoted_data = {
            "asin": asin,
            "title": product["title"],
            "reason": promotion_reason,
            "added_date": "2025-03-18"  # Use datetime.now().isoformat() in production
        }
        promoted_collection.update_one({"asin": asin}, {"$set": promoted_data}, upsert=True)
        r.set(f"promoted:{asin}", json.dumps(promoted_data), ex=7*24*3600)
        print(f"✅ Promoted product added: {asin}")
    except Exception as e:
        print(f"❌ Error adding promoted product: {e}")
```

Combine Databases

UC San Diego

- **Goal:**

  - Use **PostgreSQL (structured metadata)** + **Neo4j (graph-based recommendations)** + **MongoDB (fast lookup for promotions)**
  - Build an **intelligent hybrid recommendation system.**

- **Connection Setup**

  Python integration with:

  - **PostgreSQL** (psycopg2) for structured data.
  - **Neo4j** (neo4j-driver) for graph-based recommendations.
  - **MongoDB** (pymongo) for quick promotional product retrieval.
  - **Figure on the right** shows the connection setup.

- **How It Works:**

Step 1: Query **Neo4j** to find **co-purchased items** (graph relationships).

Step 2: Query **PostgreSQL** to get **product details and user ratings**.

Step 3: Query **MongoDB** to retrieve **promotional products for fast lookup**.

Step 4: Combine results, rank by **sales rank & average rating**, and return recommendations.

Database Connection and Check

```python
# PostgreSQL (change to your own database/user/password/host/port)
pg_conn = psycopg2.connect(
    database="amazon_db",
    user="postgres",
    password="101123",
    host="localhost",
    port="5432"
)
pg_cursor = pg_conn.cursor()

# Test PostgreSQL
pg_cursor.execute("SELECT COUNT(*) FROM amazon_products;")
result = pg_cursor.fetchone()
print("PostgreSQL is connected, total products:", result[0])

# Neo4j
neo4j_driver = GraphDatabase.driver("bolt://localhost:7687", auth=("neo4j", "20001011"))

# Test Neo4j
def test_neo4j_connection():
    query = "MATCH (p:Product) RETURN count(p) AS total_products"
    with neo4j_driver.session() as session:
        result = session.run(query)
        total_products = result.single()["total_products"]
        print("Neo4j is connected, total products:", total_products)

test_neo4j_connection()

# MongoDB
mongo_client = pymongo.MongoClient("mongodb://localhost:27017/")
db = mongo_client["amazon"]
promoted_collection = db["promoted_products"]

# Redis
r = redis.Redis(host='localhost', port=6379, db=0)
```

```
PostgreSQL is connected, total products: 548552
```

DEMO

Lessons Learned

- **Version 1: Exact Matching Issues**
  - Only returned products with **exact title match**, missing relevant results.
  - Solution: Switched to **partial title search**.

- **Version 2: Basic Co-Purchase Recommendations**
  - Recommended **products directly linked** in Neo4j.
  - Problem: Lacked **flexibility and ranking** for better results.

- **Version 3: Smarter Recommendations with Multiple Factors**
  - Introduced **ranking by rating** (from PostgreSQL).
  - Improved Neo4j recommendations with **community-based product associations**.

- **Version 4: Fully Integrated Hybrid System**
  - Combined **SQL, Graph, and NoSQL** for a **multi-source recommendation**.
  - Ensured **fast lookup**, **category-based**, and **community-based** recommendations.
  - Balanced **speed, relevance, and scalability**.

- **Key Takeaways**
  - **Data preprocessing** is critical for large datasets.
  - **Hybrid systems** provide more accurate and diverse recommendations.
  - **Database integration** requires careful optimization to avoid performance bottlenecks.
  - **Real-world recommendation systems** need flexibility in **search queries and ranking factors**.

Conclusion

Contributions
- Successfully integrated **PostgreSQL, Neo4j, and MongoDB** into a unified recommendation system.
- Built a **hybrid recommendation model** combining relational, graph, and key-value store techniques.
- Optimized data processing by **preprocessing categories** and reducing redundant data.

Challenges Overcome:

- **Large-scale data issues** (memory constraints in Neo4j).
- **Balancing accuracy vs. efficiency** in recommendation ranking.

Future Directions:

- **Improve recommendation logic** (e.g., collaborative filtering, deep learning-based ranking).
- **Enhance user personalization** (e.g., user behavior-based recommendations).
- **Optimize query performance** (e.g., indexing, caching for faster lookups).
- **Detects fake reviews and suspicious activities,** ensuring trust and better recommendations on the platform.

Thank you!