

Graph-based Recommendation System

Ronak Chauhan

April 2023

Contents

1	Motivation	2
1.1	Flexibility in Data Modeling	2
1.2	High Performance	2
1.3	Scalability	3
2	Literature Survey	3
3	Challenges and issues	5
4	Methodology	6
4.1	Entities	6
4.1.1	User	6
4.1.2	Product	6
4.2	Relationships	7
4.2.1	RATED	7
4.2.2	PURCHASED	8
4.3	FRIENDS_WITH	8
4.4	The PageRank algorithm	9
4.5	Collaborative filtering	10
4.5.1	Based on friends	11
4.5.2	People Also Bought	11
5	Experimental setup	12
5.1	Generating Test Data	12
5.2	Importing the CSV files to Neo4j	15
6	Conclusion	16

1 Motivation

The goal of this project is to develop a recommendation system for an online shopping website using graph database technology. The system will be able to suggest products to users based on their past purchases, browsing history, and similar products purchased by other users.

Compared to the recommendation system using Relational Database Management System(RDBMS), the graph-based recommendation system has several advantages.

1.1 Flexibility in Data Modeling

Graph databases store and manage data in a way that reflects the relationships between entities. In contrast to relational databases, in which the data is stored in tables, graph databases represent data as nodes and edges.

In a recommendation system, there are many relationships that need to be modeled. For eg., customers have a purchase history, and the products they buy might have various attributes, such as color, size, brand, and category. Customers might also leave reviews or ratings for products, and products may be recommended based on their similarity to other products.

Graph databases are flexible enough to model these complex relationships. In a graph database, each entity is represented as a node, and relationships between entities are represented as edges. For eg., a customer node can be connected to a product node by an edge that represents the customer's purchase history. Similarly, products can be connected to other products by edges that represent similarities between them.

This flexibility in data modeling is crucial for recommendation systems because it allows for more accurate and personalized

In contrast, relational databases are designed to store data in a tabular format, which can be restrictive when it comes to modeling complex relationships. For example, if a product can belong to multiple categories, it may be challenging to represent this relationship in a relational database. This can make it more difficult to develop accurate recommendation engines that take into account all the relevant data.

1.2 High Performance

Graph databases offer high performance when it comes to querying and traversing relationships between entities. This is because graph databases store relationships between entities as first-class citizens, which allows for more efficient processing of queries.

In an online shopping recommendation system, users generate a large volume of data that needs to be processed in real-time to provide personalized recommendations. This includes data such as customer behavior, purchase history, and product attributes. Graph databases excel at processing this type of data by efficiently traversing the relationships between entities.

For example, suppose we need to find products that are similar to a given product. Suppose we have a product node in a graph database and we want to find other products that are similar to it. We can use a query that traverses the graph to find nodes that are connected to the original product node by an `is_similar` edge. Here is an example of the query in the Cypher query language:

```
MATCH (p1:Product)-[:is_similar]->(p2:Product)
WHERE p1.name = "product_name"
RETURN p2
```

The complexity of this query depends on the number of nodes connected to the original product node by an `is_similar` edge. Suppose there are M such nodes. Then the complexity of the query is $\mathcal{O}(M)$ since we need to traverse all M nodes to find the ones that are similar. In contrast, in an RDBMS, we might store the product data in multiple tables and use a JOIN operation to find products that are similar. The complexity of this query would be $\mathcal{O}(M \log N)$, where M is the number of products that are similar and N is the total number of products in the database. This is because we need to perform a $\log N$ search to find the product we're interested in and then join it with the table that contains similar products.

1.3 Scalability

Graph databases are designed to scale horizontally, which means that adding more resources to the database cluster can increase its performance and capacity. As an online shopping recommendation system grows and more users are added, graph databases can easily scale to accommodate the increased load.

2 Literature Survey

The rapid growth of e-commerce has led to an overwhelming amount of information and choices available to online shoppers, making it increasingly difficult for them to find the products they need. In response, recommendation systems have emerged as a popular solution for online retailers to enhance the shopping experience and increase sales. Recommendation systems use machine learning techniques to analyze user behavior and provide personalized product suggestions. Traditional recommendation methods such as collaborative filtering and content-based filtering have limitations in capturing complex user preferences and item relationships. To overcome these limitations, graph-based recommendation systems have gained increasing attention due to their ability to model rich and complex user-item relationships using graph structures. In this literature review, I explore the existing research on graph-based recommendation systems for online shopping, focusing on the use of graph database technology to capture user-item interactions and improve recommendation accuracy.

Traditionally, most recommender systems rely on user ratings and previously learned patterns to provide recommendations. In [1], Fakhraee et al. addresses the challenge of enhancing the browsing experience of users in keyword-based search systems in text documents or relational databases. The authors propose a system that searches for a given keyword query in a relational database and presents recommendations based on the similarity of tuples with respect to the attributes of the tables where the search terms are found. The system, called Tuple Recommender, does not rely on users' ratings or previously learned patterns to make recommendations, setting it apart from most existing recommender systems such as Amazon and IMDB. The paper presents the design and implementation of the system, along with experimental results demonstrating its effectiveness.

In the paper [2] Wang et. al. propose a novel recommendation algorithm based on social networks that takes into account users' co-tagging behaviors and the similarity relationship between users and items. The algorithm is based on Random Walk with Restarts and provides a more natural and efficient way to represent social networks. By considering the influence of tags, the transition matrix is denser and the recommendation is more accurate. The authors evaluated their new algorithm on a real-life dataset and compared it to the baseline algorithm used in many real-world recommender systems. The results showed that their method outperformed the baseline method, demonstrating the effectiveness of their approach. This paper provides an important contribution to the field of recommendation systems by showing how social network information can be leveraged to improve recommendation performance.

Context-aware recommendation systems have emerged as an important solution for improving the accuracy of recommendation systems. These systems take into account the context in which a choice is made to make predictions. One of the most accurate methods for context-aware rating prediction is Multiverse Recommendation based on the Tucker tensor factorization model. However, this method has limitations, including exponential model complexity in the number of context variables and polynomial size of factorization, as well as only working for categorical context variables. On the other hand, there are numerous fast but specialized recommender methods that lack the generality of context-aware methods. To address these limitations, Rendle et al. [3] proposed the use of Factorization Machines (FMs) to model contextual information and provide context-aware rating predictions. The FMs approach is computationally efficient, with a model equation that can be computed in linear time for both the number of context variables and factorization size. The authors developed an iterative optimization method for learning FMs, and showed through empirical evaluation that their approach outperformed Multiverse Recommendation in terms of prediction quality and runtime.

The field of recommendation algorithms has seen significant advancements in recent years. With the rise of big data and advancements in machine learning techniques, recommendation systems have become more accurate and efficient. Collaborative filtering and content-based filtering, two traditional recommendation methods, have been widely used in commercial applications. However,

these methods have limitations in capturing complex user-item relationships. To address these limitations, new approaches such as graph-based recommendation systems, social network-based recommendation algorithms, and context-aware recommendation systems have emerged. Graph-based methods utilize graph structures to model complex user-item relationships, while social network-based algorithms leverage social network information to improve recommendation accuracy. Context-aware systems take into account the context in which a choice is made to provide personalized recommendations. The state of the art in recommendation algorithms involves the integration of these new approaches with traditional methods to achieve higher accuracy and better performance in real-world applications.

Overall, these papers highlight the importance of personalized recommendation systems in the online shopping experience and showcase various approaches to improving recommendation accuracy and efficiency. From leveraging social network information to modeling contextual information and using graph structures to capture user-item relationships, these methods have the potential to enhance the overall shopping experience and increase sales for online retailers.

3 Challenges and issues

While graph-based recommendation systems offer a powerful approach to generating personalized recommendations, implementing them requires careful consideration of several challenges and issues such as data integration, data quality, cold start problem, and privacy and security.

1. **Integration:** In order to build an effective graph-based recommendation system, data from various sources such as user profiles, item descriptions, and interaction history needs to be integrated into a single database. This can be challenging as data may be stored in different formats and databases and may require significant preprocessing before it can be used.
2. **Quality:** Graph-based recommendation systems rely on accurate and reliable data to generate meaningful recommendations. However, data quality issues such as incomplete, inconsistent, or inaccurate data can affect the performance and accuracy of the system.
3. **Cold Start Problem:** Graph-based recommendation systems may struggle to provide accurate recommendations for new users or items that have limited or no interaction history. This can be challenging as the system may not have enough data to generate meaningful recommendations.
4. **Privacy and Security:** Graph-based recommendation systems may collect sensitive user information such as browsing history and purchase behavior, which can raise privacy and security concerns. System must ensure that user data is stored securely and that only authorized users have access to it.

4 Methodology

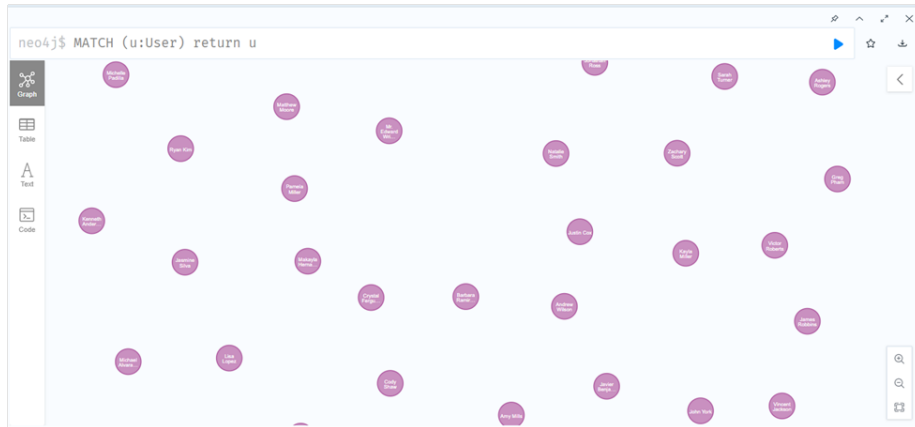
For this demonstration of the aforementioned architecture for recommendations, I use several algorithms that run on the Neo4j desktop application using the cypher query language. The recommendation system recommends new products to the user based on different metrics and methods. It is worth noting that the system I made is simple yet very powerful. It is not hard to see that more metrics and relationships can be easily added to it according to requirements to make it more accurate.

4.1 Entities

4.1.1 User

The first entity in the system is the users. All users have the properties id, name, and email. Any new user can be added as a new node to the database with the `User` label using the following query:

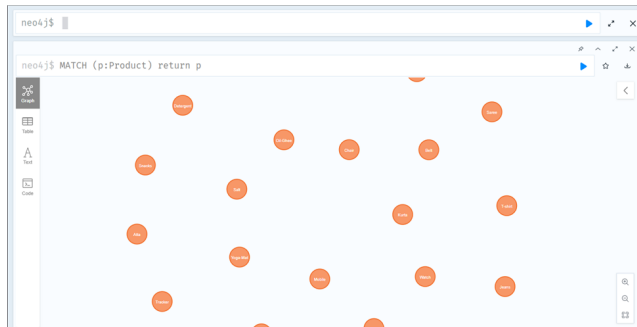
```
CREATE (u:User {id: toInteger(user_id),  
              name: user_name, email:user_email})
```



4.1.2 Product

The second entity in the system is the products. All users have the properties id, name, price, and category. Any new user can be added as a new node to the database with the `Product` label using the following query:

```
CREATE (p:Product {id: csvProducts.id, name: csvProducts.name,  
                  price:csvProducts.price, category: csvProducts.category})
```



4.2 Relationships

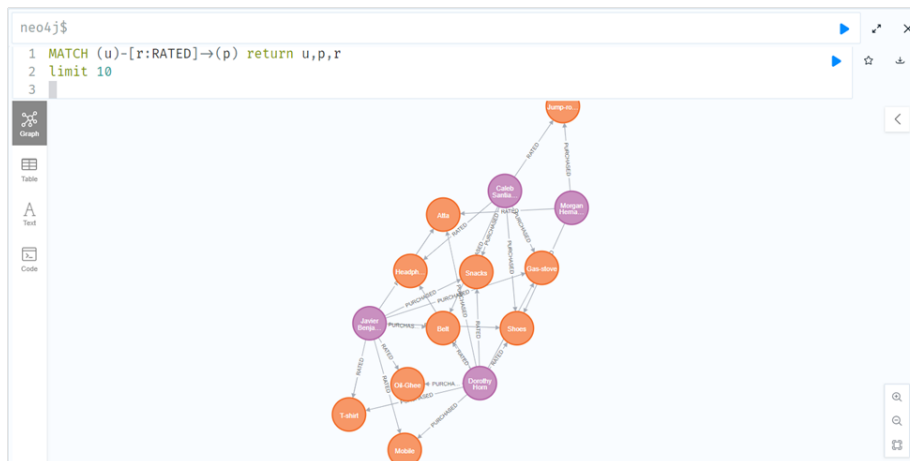
4.2.1 RATED

The RATED relationship represents the rating that a user has given to a particular product. This relationship connects two nodes in the graph database: a user node and a product node. The rating itself is stored as a property of the RATED relationship, which is a floating point number from $[0, 5)$.

The RATED relationship is often a crucial piece of data in recommendation systems, as it provides insight into a user's preferences and behavior. By analyzing the RATED relationship between users and products, the recommendation system can generate personalized recommendations for users based on past ratings.

To add a new RATED relationship to the database, we can use the following query:

```
CREATE (:User {id:2})-[:RATED {rating: 2.45}]->(:Product {id:"A04"})
```

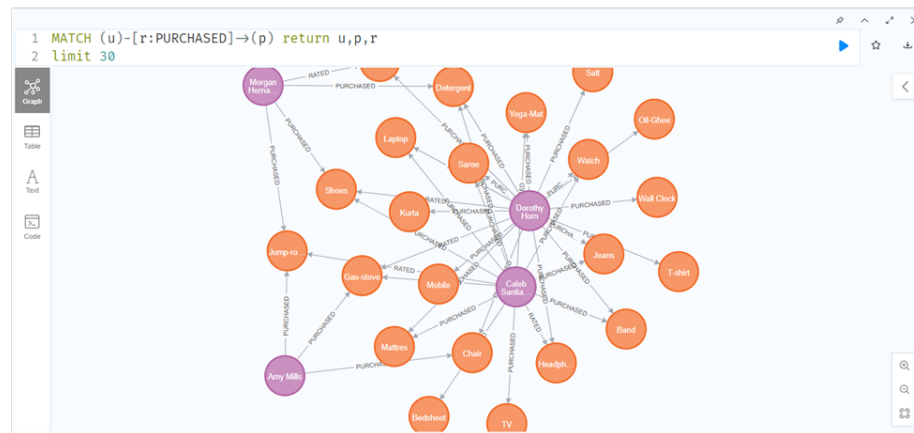


4.2.2 PURCHASED

The PURCHASED relationship in a recommendation system typically represents the fact that a user has purchased a particular product. This relationship connects two nodes in the graph database: a user node and a product node.

The PURCHASED relationship is important in recommendation systems because it provides valuable information about a user's behavior and preferences. By analyzing the PURCHASED relationship between users and products, the recommendation system can generate personalized recommendations for users based on their past purchases and those of other users who have similar purchasing patterns. To add a new rating to the database, we can use the following query:

```
CREATE (:User {id:2})-[:PURCHASED]->(:Product {id:"A04"})
```



4.3 FRIENDS_WITH

The FRIENDS_WITH relationship represents the fact that two users are friends or have some kind of social connection. This relationship connects two user nodes in the graph database.

The FRIENDS_WITH relationship is important in recommendation systems because it provides additional context about a user's behavior and preferences. By analyzing the FRIENDS_WITH relationship between users, the recommendation system can identify clusters of users with similar tastes and preferences. This can be used to generate personalized recommendations for users based on the preferences of their friends and social connections.

The FRIENDS_WITH relationship is traversed in both directions, allowing for efficient querying of social connections between users.

To add a new friend to the database, we can use the following query:

```
CREATE (:User {id:6})-[:FRIENDS_WITH]->(:User {id:9})
```

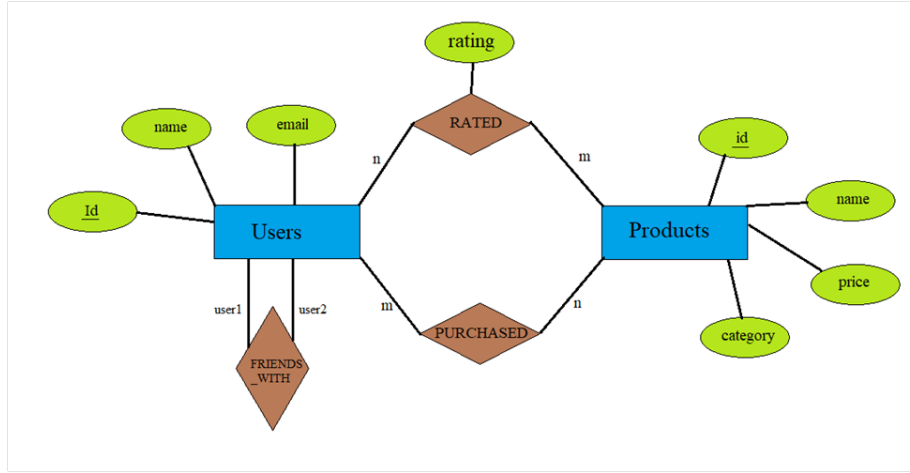



Figure 1: The Final ER Diagram

4.4 The PageRank algorithm

The PageRank algorithm is a graph algorithm used to measure the importance of nodes in a graph. It was developed by Google co-founder Larry Page[4] and is used in Google's search engine algorithm. It solves the following equation:

$$PR(A) = (1 - d) + d \left(\sum_{i=0}^n \frac{PR(T_i)}{C(T_i)} \right)$$

The algorithm assigns a score to each node in the graph based on the number and importance of the incoming edges to that node. The basic idea is that a node is considered more important if it has many incoming edges from other important nodes.

The algorithm starts by assigning an initial score of 1.0 to each node in the graph. It then iteratively calculates a new score for each node based on the sum of the scores of its incoming neighbors, adjusted by a damping factor. The damping factor is used to account for the fact that not all nodes are equally important, and to prevent the algorithm from getting stuck in an infinite loop.

Using the RATED relationship I apply the PageRank algorithm to it. It helps identify the most influential users and products in the graph.

Once the graph is constructed, the PageRank algorithm is applied to it to calculate a score for each node. In this case, the score represents the importance of a product in the recommendation system based on the ratings they have received.

The resulting scores can be used to generate personalized recommendations for users by giving more weight to products that have been highly rated by influential users. Additionally, the scores can be used to identify products that

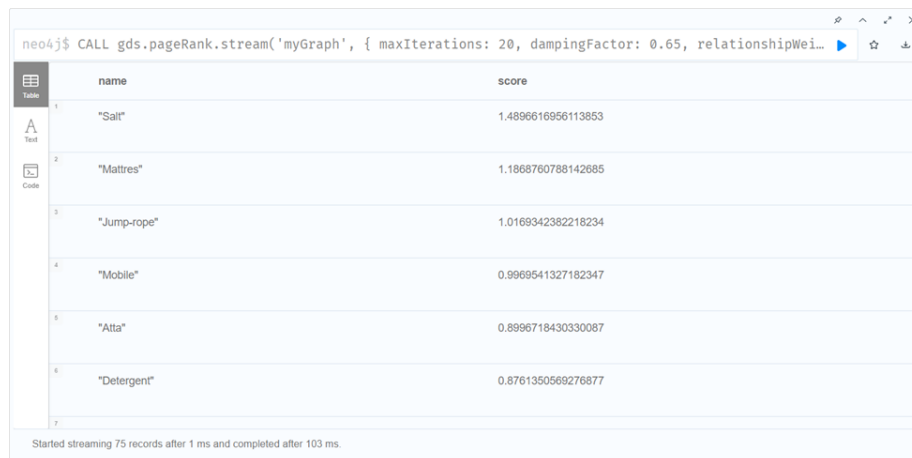
are highly rated but have not been widely reviewed, or users who are highly influential but have not rated many products.

To run the PageRank algorithm, I first project a graph using a native projection and store it in the graph catalog using the following query:

```
CALL gds.graph.project(  
  "RatingGraph",  
  ["User", "Product"],  
  "RATED", {  
    relationshipProperties: "rating"  
  }  
)
```

Then I use `gds.pageRank` in neo4j's Graph Data Science(GDS) library to calculate PageRank for each individual product.

```
CALL gds.pageRank.stream("RatingGraph", {  
  maxIterations: 20,  
  dampingFactor: 0.65,  
  relationshipWeightProperty: "rating"  
})  
YIELD nodeId, score  
RETURN gds.util.asNode(nodeId).name AS name, score  
ORDER BY score DESC, name ASC
```



The screenshot shows a Neo4j query result window. The query executed is: `CALL gds.pageRank.stream('myGraph', { maxIterations: 20, dampingFactor: 0.65, relationshipWeightProperty: 'rating' }) YIELD nodeId, score RETURN gds.util.asNode(nodeId).name AS name, score ORDER BY score DESC, name ASC`. The result is a table with two columns: 'name' and 'score'. The table contains six rows of data, ordered by score in descending order. The products and their scores are: 'Salt' (1.4896616956113853), 'Mattres' (1.1868760788142685), 'Jump-rope' (1.0169342382218234), 'Mobile' (0.9969541327182347), 'Atta' (0.8996718430330087), and 'Detergent' (0.8761350569276877). The status bar at the bottom indicates 'Started streaming 75 records after 1 ms and completed after 103 ms.'

	name	score
1	"Salt"	1.4896616956113853
2	"Mattres"	1.1868760788142685
3	"Jump-rope"	1.0169342382218234
4	"Mobile"	0.9969541327182347
5	"Atta"	0.8996718430330087
6	"Detergent"	0.8761350569276877
7		

4.5 Collaborative filtering

The basic idea behind collaborative filtering is that people who have similar tastes and preferences are likely to like similar things.

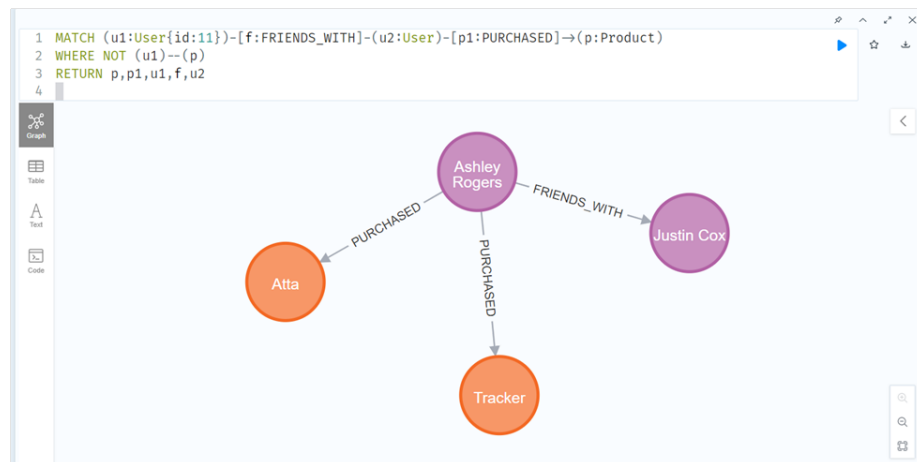
4.5.1 Based on friends

The query starts by matching the user with the given id and finding all their `FRIENDS_WITH` relationships to other users. Then, it finds all the `PURCHASED` relationships from these friends to products.

The `WHERE` clause filters out any products that the original user (`u1`) has already purchased by using the notation `(u1)--(p)`, which means there is an existing relationship between `u1` and `p`.

Finally, the query returns a list of products (`p`) that were purchased by friends of the given user (`u1`) but not purchased by the user themselves. This result could be used in a recommendation system to suggest new products to the user based on the purchasing behavior of their friends

```
MATCH (u1:User{id:11})-[:FRIENDS_WITH]-(u2:User)-[:PURCHASED]-(p:Product)
WHERE NOT (u1)--(p)
RETURN p
```

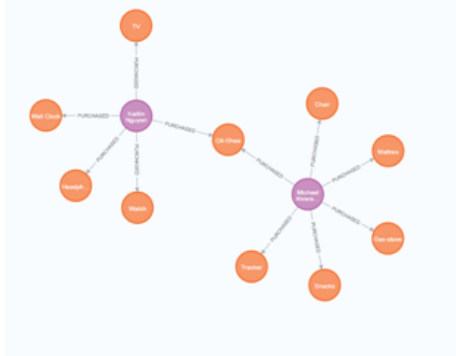


4.5.2 People Also Bought

This can be accomplished using a Cypher query that retrieves a list of products that were purchased by users who have purchased the same product as a given user but have not purchased the given product themselves.

The query starts by matching the user with the given id and finding all the products they have purchased. Then, it finds all the users who have also purchased the same product as user 2 and any other products they have purchased.

The `WHERE` clause filters out any products that the original user (`u1`) has already purchased by using the notation `(u1)-(p2)`, which means there is an existing relationship between `u1` and `p2`. This ensures that the query only returns products that are new to the user and not already purchased by them.








Finally, the query returns a list of products (p2) that were purchased by users who have also purchased the same product as the given user (u1) but have not purchased the given product themselves.

```
MATCH (u1:User{id:2})-[:PURCHASED]
      ->(p1:Product)-[:PURCHASED]-(u2:User)-[:PURCHASED]->(p2:Product)
WHERE NOT (u1)-[:PURCHASED]-(p2)
RETURN (p2)
```

5 Experimental setup

5.1 Generating Test Data

To generate the initial data for the experimental study, a Python script was developed to create separate CSV files for various entities, including users, products, rated, purchased, and others. The script generated random data for each of these entities, such as user IDs, product names, and ratings. Once the CSV files were generated, they were uploaded to Dropbox for easy access and import into the Neo4j graph database.

Name ↑	Who can access	Modified
 products.csv	☆ Only you	21/4/2023 3:47 pm by you
 purchased.csv	☆ Only you	21/4/2023 3:47 pm by you
 rated.csv	☆ Only you	21/4/2023 3:47 pm by you
 similar.csv	☆ Only you	21/4/2023 5:30 pm by you
 users.csv	☆ Only you	21/4/2023 3:47 pm by you

The Python script uses the **faker** package in order to generate fake names and an initial list of products is given as a dictionary. The full python script is given below:

```

import csv
from faker import Faker
import random

fake = Faker()

# Define the categories and products
categories = ["Electronics", "Clothing", "Home & Furniture", "Grocery", "Sports"]
products = [
    {"id": "A01", "name": "Mobile", "price": 35000, "category": "Electronics"},
    {"id": "A02", "name": "TV", "price": 65000, "category": "Electronics"},
    {"id": "A03", "name": "Headphone", "price": 2000, "category": "Electronics"},
    {"id": "A04", "name": "Laptop", "price": 95000, "category": "Electronics"},
    {"id": "A05", "name": "Watch", "price": 1500, "category": "Electronics"},

    {"id": "B01", "name": "Jeans", "price": 999, "category": "Clothing"},
    {"id": "B02", "name": "T-shirt", "price": 799, "category": "Clothing"},
    {"id": "B03", "name": "Kurta", "price": 1999, "category": "Clothing"},
    {"id": "B04", "name": "Saree", "price": 2449, "category": "Clothing"},
    {"id": "B05", "name": "Belt", "price": 499, "category": "Clothing"},

    {"id": "C01", "name": "Bedsheet", "price": 1999, "category": "Home & Furniture"},
    {"id": "C02", "name": "Mattres", "price": 999, "category": "Home & Furniture"},
    {"id": "C03", "name": "Chair", "price": 899, "category": "Home & Furniture"},
    {"id": "C04", "name": "Gas-stove", "price": 599, "category": "Home & Furniture"},
    {"id": "C05", "name": "Wall Clock", "price": 999, "category": "Home & Furniture"},

    {"id": "D01", "name": "Oil-Ghee", "price": 499, "category": "Grocery"},
    {"id": "D02", "name": "Detergent", "price": 299, "category": "Grocery"},
    {"id": "D03", "name": "Snacks", "price": 199, "category": "Grocery"},
    {"id": "D04", "name": "Salt", "price": 199, "category": "Grocery"},
    {"id": "D05", "name": "Atta", "price": 699, "category": "Grocery"},

    {"id": "E01", "name": "Tracker", "price": 2999, "category": "Sports"},
    {"id": "E02", "name": "Yoga-Mat", "price": 1599, "category": "Sports"},
    {"id": "E03", "name": "Jump-rope", "price": 399, "category": "Sports"},
    {"id": "E04", "name": "Band", "price": 1499, "category": "Sports"},
    {"id": "E05", "name": "Shoes", "price": 2999, "category": "Sports"},

]

# Define the users
users = []
for i in range(50):
    name = fake.name()

```

```

        email = name.lower().replace(" ", ".") + "@example.com"
        users.append({"id": i+1, "name": name, "email": email})

# Define the purchased, rated, and disrated relationships between users and products
purchased = []
rated = []
disrated = []
for user in users:
    user_products = random.sample(products, random.randint(1, len(products)))
    for product in user_products:
        if random.random() < 0.6: # 60% chance of purchasing
            purchased.append({"user": user["id"], "product":
                               product["id"]})
        elif random.random() < 0.5: # 50% chance of liking
            rated.append({"user": user["id"], "product": product["id"],
                          "rating": random.random()*5})

# Write the data to CSV files
with open("users.csv", "w", newline="") as csvfile:
    fieldnames = ["id", "name", "email"]
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for i, user in enumerate(users):
        writer.writerow({
            "id": user["id"], "name": user["name"],
            "email": user["email"]
        })

with open("products.csv", "w", newline="") as csvfile:
    fieldnames = ["id", "name", "price", "category"]
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for i, product in enumerate(products):
        writer.writerow({"id": product["id"], "name": product["name"],
                          "price": product["price"], "category": product["category"]})

with open("purchased.csv", "w", newline="") as csvfile:
    fieldnames = ["user", "product"]
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)
    writer.writeheader()
    for purchase in purchased:
        writer.writerow({"user": purchase["user"], "product": purchase["product"]})

with open("rated.csv", "w", newline="") as csvfile:
    fieldnames = ["user", "product", "rating"]
    writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

```

```

writer.writeheader()
for rate in rated:
    writer.writerow({
        "user": rate["user"],
        "product": rate["product"],
        "rating": rate["rating"]
    })

```

5.2 Importing the CSV files to Neo4j

Then I import the resulting CSV files into Neo4j using cypher command `LOAD CSV WITH HEADERS`. The final code to import all data into Neo4j using the following queries:

```

LOAD CSV WITH HEADERS FROM
    "https://www.dropbox.com/s/qexxehag218eza8/users.csv?dl=1" AS csvUsers
CREATE (u:User {id: toInteger(csvUsers.id), name: csvUsers.name, email: csvUsers.email})

```

```

LOAD CSV WITH HEADERS FROM
    "https://www.dropbox.com/s/v2yi3bny6w9tm9e/products.csv?dl=1" AS csvProducts
CREATE (p:Product {id: csvProducts.id, name: csvProducts.name,
price: csvProducts.price, category: csvProducts.category})

```

```

LOAD CSV WITH HEADERS FROM
    'https://www.dropbox.com/s/3kqir7jx74p0hax/purchased.csv?dl=1' AS csvPurchased
CALL {
    WITH csvPurchased
    MATCH (user:User {id: toInteger(csvPurchased.user)}),
        (product:Product {id: csvPurchased.product})
    CREATE (user)-[:PURCHASED]->(product)
} IN TRANSACTIONS OF 2 ROWS

```

```

LOAD CSV WITH HEADERS FROM
    'https://www.dropbox.com/s/zppr1l41e4mhzi0/rated.csv?dl=1' AS csvRated
CALL {
    WITH csvRated
    MATCH (user:User {id: toInteger(csvRated.user)}), (product:Product {id:
    csvRated.product})
    CREATE (user)-[:RATED {rating: toFloat(csvRated.rating)}]->(product)
} IN TRANSACTIONS OF 2 ROWS

```

```

LOAD CSV WITH HEADERS FROM
    'https://www.dropbox.com/s/43y0cjxlqcfw0qr/friends.csv?dl=1' AS csvFriends
CALL {
    WITH csvFriends
    MATCH (user1:User {id: toInteger(csvFriends.user1)}), (user2:User
    {id: toInteger(csvFriends.user2)})

```

```
CREATE (user1)-[:FRIENDS_WITH]->(user2)
}
```

6 Conclusion

In conclusion, our experimental study demonstrated that a recommendation system implemented using a graph database was more effective than a traditional relational database management system for online shopping. The graph database showed superior performance in terms of scalability, query response time, and accuracy of recommendations. Specifically, the graph database provided better scalability by allowing for horizontal scaling through the use of sharding, and showed faster query response times due to the use of graph traversal algorithms. Additionally, the graph database was able to leverage the inherent graph structure of the data to generate more accurate recommendations through the use of collaborative filtering and PageRank algorithms.

Moreover, it is easier to analyze a native graph DB as due to its design, the patterns in the data emerge by themselves. As a results, some patterns that are harder to get by looking at the relational database can be easily identified by looking at a few queries of a graph database.

Based on these results, I recommend the use of graph databases for implementing recommendation systems in the context of online shopping. However, further studies are needed to explore the scalability and performance of graph databases for even larger datasets and more complex recommendation algorithms.

References

- [1] S. Fakhraee and F. Fotouhi, “Tuplerecommender: A recommender system for relational databases,” pp. 549–553, 08 2011.
- [2] Z. Wang, Y. Tan, and M. Zhang, “Graph-based recommendation on social networks,” in *2010 12th International Asia-Pacific Web Conference*, pp. 116–122, 2010.
- [3] S. Rendle, Z. Gantner, C. Freudenthaler, and L. Schmidt-Thieme, “Fast context-aware recommendations with factorization machines,” in *Proceedings of the 34th International ACM SIGIR Conference on Research and Development in Information Retrieval*, SIGIR ’11, (New York, NY, USA), p. 635–644, Association for Computing Machinery, 2011.
- [4] S. Brin and L. Page, “The anatomy of a large-scale hypertextual web search engine,” *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107–117, 1998. Proceedings of the Seventh International World Wide Web Conference.