DSC 202 Project Report

Team member:

Shuyu Wang (A16357263),
Yuan Zhang (A 69037582),
Zhoutianning Pan (A16263806)

1. Problem Statement

1.1. Application Overview

The goal of this project is to develop a hybrid recommendation system that enhances product recommendations by leveraging SQL and Neo4j together. Traditional recommendation approaches often struggle with either data relation complexity (SQL) or lack of structured storage (Neo4j), so we combine the strengths of both to build a system that is fast, scalable, and personalized. A customer visits the AdventureWorks website and views a product page.The system retrieves similar products based on the customer's purchasing history and recommends goods he/she may be interested in. The customer can also set filters for the recommendation such as product category and expected shipping time.

1.2. Importance of Our Recommendation System

1.2.1. **Enhanced User Experience**: Customers receive more relevant and personalized product recommendations which improves users' shopping experience and increases the company's sales.

1.2.2. **Efficient Data Handling**: SQL stores structured data (product details such as keywords of product description and the estimated shipping time of certain product) while Neo4j optimizes relationship-based queries (collaborative filtering).

1.2.3. **Scalability & Performance** : SQL handles large-scale structured data, while Neo4j provides real-time traversal for recommendations without expensive joins.

1.2.4. **Applicable Across Industries**: Advanced system with similar structure is used for real industries:
E-commerce platforms (e.g., Amazon, eBay)
Streaming services (e.g., Netflix, Spotify)
Social networks (e.g., LinkedIn, Facebook)

1.3. Data Source and tables needed for the application:

We use data from the **AdventureWorks Database**, which consists of multiple tables that store customer transactions, product details, and relationships. AdventureWorks is a bicycle manufacturing industry, it has a global presence with various departments such as sales and production. We select only offline sales transactions (OnlineOrderFlag = 0) which provides **60,919** recorded purchases.

Here are the tables we mainly use to build our recommendation system.

Table Name	Description	Usage
SalesOrderHeader & SalesOrderDetailWithDuration	Tracks sales orders and their details.	Used for customer purchase history and co-purchase analysis in collaborative filtering.
Product	Contains all available products, including bikes, accessories, and components.	Serves as the reference for recommended products.
ProductDescription	Contains textual product descriptions in multiple languages.	Helps in text-based filtering and finding similar products based on descriptions.
ProductCategory & ProductSubcategory	Defines hierarchical product categories (e.g., "Bikes" → "Mountain Bikes").	Enables category-based filtering and diverse recommendations.
ProductDescriptionCulture	Links product descriptions to specific languages (e.g., English).	Ensures text-based recommendations are language-specific.
Customer	Stores customer details and purchase histories.	Helps in collaborative filtering by linking customers to past purchases.

2. Detailed Coverage

2.1. Data Preparation

2.1.1. Full-Text Indexing:

- Created ProductDescriptionIndex tables to store tsvector representations for product descriptions and special offers.
- Applied GIN indexes to enhance full-text search performance.

```sql
-- Create Full-Text Search Indexes
-- Store precomputed tsvector values to optimize search performance.
CREATE TABLE Production.ProductDescriptionIndex AS
SELECT
    pmdc.ProductDescriptionID,
    pd.Description,
    to_tsvector('english', pd.Description) AS tsv
FROM Production.ProductModelProductDescriptionCulture pmdc
JOIN Production.ProductDescription pd
    1..n<->1: ON pmdc.ProductDescriptionID = pd.ProductDescriptionID
WHERE pmdc.CultureID = 'en';  -- Only English descriptions

-- Create indexes to speed up full-text search
CREATE INDEX idx_productdescription_tsv ON Production.ProductDescriptionIndex USING GIN(tsv);
```

2.1.2. Keyword Extraction from Product Descriptions:

Split long descriptions into individual keywords for better searchability and kept the top 10 keywords per product.

```sql
CREATE TABLE Production.ProductIDKeywords_split AS
WITH exploded_keywords AS (
    -- split keywords into multiple rows
    SELECT
        p.ProductID,
        pd.ProductDescriptionID,
        unnest(string_to_array(pd.Keywords, ', ')) AS keyword
    FROM Production.ProductDescriptionKeywords pd
    JOIN Production.ProductModelProductDescriptionCulture pmdc
        ON pd.ProductDescriptionID = pmdc.ProductDescriptionID
    JOIN Production.ProductModel pm
        1..n<->1: ON pmdc.ProductModelID = pm.ProductModelID
    JOIN Production.Product p
        1<->1..n: ON pm.ProductModelID = p.ProductModelID
    WHERE pmdc.CultureID = 'en'
),
ranked_keywords AS (
    SELECT
        ProductID,
        ProductDescriptionID,
        keyword,
        ROW_NUMBER() OVER (PARTITION BY ProductDescriptionID ORDER BY keyword) AS rn
    FROM exploded_keywords
    WHERE keyword IS NOT NULL
)
SELECT
    ProductID,
    ProductDescriptionID,
    keyword
FROM ranked_keywords
WHERE rn <= 10  -- keep 10 keywords
ORDER BY ProductID, ProductDescriptionID, rn;
```

2.1.3. Order Duration Calculation:

Duration (ShipDate - OrderDate) measures the days between order and shipping, helping track delivery speed.

```
CREATE TABLE --- AS
SELECT
    SalesOrderID,
    RevisionNumber,
    OrderDate,
    DueDate,
    ShipDate,
    Status,
    OnlineOrderFlag,
    SalesOrderNumber,
    PurchaseOrderNumber,
    AccountNumber,
    CustomerID,
    .
    (ShipDate - OrderDate) AS Duration  -- Calculate duration in days
FROM ;
```

2.1.4. Output table used as input for Neo4j

All the output table in the data preparation step will be used as input file and put into the Neo4j input folder for building the recommendation system

2.2. Neo4j Recommender

The recommender is implemented using Neo4j, a graph database. The system generates product recommendations for a target customer based on collaborative filtering, product categories, and keyword matching. The code integrates with SQL databases and Python to extract relevant data and enhance recommendation quality by considering various factors such as product categories and description keywords.

2.2.1. Finding Products Purchased by the Target Customer

```
MATCH (targetCustomer:SalesCustomer {CustomerID:
$customer_id})-[:PLACED_ORDER]->(order:SalesFilteredSalesOrderHeaderWithDur
ation)
WHERE order.Duration <= $max_duration
MATCH
(order)-[:HAS_Detail]->(:SalesFilteredSalesOrderDetail)-[:HAS_PRODUCT]->(pu
rchased:ProductionProduct)
WITH targetCustomer, COLLECT(DISTINCT purchased.ProductID) AS
targetProducts
```

The first step in generating product recommendations is to retrieve the list of products that the target customer has purchased within the specified time duration. This is done using a Cypher query that matches the target customer (`targetCustomer`) to their past orders (`SalesFilteredSalesOrderHeaderWithDuration`). The `WHERE order.Duration <= $max_duration` condition ensures that only orders placed within the specified time frame are considered. The purchased products are then collected as a list of distinct `ProductID`s.

2.2.2. Finding Similar Customers Based on Purchase History

```
MATCH
(similarCustomer:SalesCustomer)-[:PLACED_ORDER]->(similarOrder:SalesFiltere
dSalesOrderHeaderWithDuration)
WHERE similarCustomer <> targetCustomer AND similarOrder.Duration <=
$max_duration
MATCH
(similarOrder)-[:HAS_Detail]->(:SalesFilteredSalesOrderDetail)-[:HAS_PRODUC
T]->(sharedProduct:ProductionProduct)
WHERE sharedProduct.ProductID IN targetProducts
WITH targetCustomer, similarCustomer, COLLECT(DISTINCT
sharedProduct.ProductID) AS sharedProducts, targetProducts
```

The system identifies other customers who have purchased similar products. It matches all customers (`similarCustomer`) who have placed an order within the same duration and have purchased at least one product that overlaps with the target customer's purchase list (`targetProducts`). This allows the system to find customers with similar purchasing patterns, which is essential for collaborative filtering-based recommendations.

2.2.3. Computing Similarity Between Customers

```
WITH targetCustomer, similarCustomer, sharedProducts, targetProducts,
    size(sharedProducts) AS sharedProductCount,
    size(targetProducts) AS targetProductCount,
    (size(sharedProducts) * 1.0 / size(targetProducts)) AS similarityScore
```

After identifying similar customers, the system calculates a similarity score based on the overlap of purchased products. The formula `size(sharedProducts) * 1.0 / size(targetProducts)` determines the degree of similarity by dividing the count of shared products by the total number of products purchased by the target customer. A higher similarity score indicates a greater overlap in purchasing behavior.

2.2.4. Identifying Un-purchased Products by the Target Customer

```
MATCH
(similarCustomer)-[:PLACED_ORDER]->(similarOrder)-[:HAS_Detail]->(:SalesFil
teredSalesOrderDetail)-[:HAS_PRODUCT]->(recommended:ProductionProduct)
WHERE NOT recommended.ProductID IN targetProducts
```

Once similar customers are identified, the system extracts the products they have purchased that the target customer has not yet bought. This ensures that recommendations include only new products that might be of interest to the customer. The query filters out any product IDs already present in `targetProducts`.

2.2.5. Fetching Product Category Details

```
OPTIONAL MATCH
(recommended)-[:PART_OF_SUBCAT]->(subCategory:ProductionProductSubcategory)
OPTIONAL MATCH
(subCategory)-[:BELONGS_TO]->(category:ProductionProductCategory)
```

To enhance the recommendations, the system retrieves category and subcategory details for each recommended product. This information is optional but useful for filtering and improving recommendation relevance. The query matches products to their respective subcategories (`ProductionProductSubcategory`) and categories (`ProductionProductCategory`).

2.2.6. Calculating Recommendation Scores

```
WITH DISTINCT recommended.ProductID AS ProductID, recommended.Name AS
ProductName,
        category.ProductCategoryID AS CategoryID, category.Name AS
CategoryName,
        subCategory.ProductSubcategoryID AS SubcategoryID, subCategory.Name
AS SubcategoryName,
        COALESCE(COUNT(DISTINCT similarCustomer), 0) AS collaborativeScore,
        COALESCE(COUNT(DISTINCT subCategory), 0) AS categoryMatchScore,
recommended, targetProducts

    // Compute keyword overlap count as keywordSimilarityScore
    OPTIONAL MATCH
(recommended)-[:HAS_KEYWORD]->(rk:ProductionProductKeywords)
    OPTIONAL MATCH
(targetProduct:ProductionProduct)-[:HAS_KEYWORD]->(pk:ProductionProductKeyw
ords)
    WHERE targetProduct.ProductID IN targetProducts AND rk.Keyword =
pk.Keyword
```

```
    WITH ProductID, ProductName, CategoryID, CategoryName, SubcategoryID,
SubcategoryName,
         collaborativeScore,
         COALESCE(categoryMatchScore, 0) AS categoryMatchScore,
         COALESCE(COUNT(DISTINCT rk.Keyword), 0) AS keywordSimilarityScore

    // If CategoryName is provided, filter by CategoryName
    WHERE ($category_name IS NULL OR CategoryName = $category_name)

    // Return top 10 recommended products sorted by scores
    RETURN ProductID, ProductName, CategoryID, CategoryName, SubcategoryID,
SubcategoryName,
        collaborativeScore, categoryMatchScore, keywordSimilarityScore,
        (collaborativeScore * 2 + categoryMatchScore +
keywordSimilarityScore) AS RecommendationScore
    ORDER BY RecommendationScore DESC, collaborativeScore DESC,
keywordSimilarityScore DESC
    LIMIT 10;
```

A composite recommendation score is calculated based on multiple factors:

- **Collaborative Score**: The number of similar customers who purchased the product.
- **Category Match Score**: The count of matching product categories.
- **Keyword Similarity Score**: A measure of keyword overlap between recommended products and the target customer's purchased products.

These values are combined in a weighted manner: `collaborativeScore * 2 + categoryMatchScore + keywordSimilarityScore` to determine the final recommendation ranking.

If a `CategoryName` is provided as an input parameter, the system filters the recommendations accordingly. This ensures that customers receive suggestions relevant to their preferred categories.

2.3. Demonstration of the Recommender System

Example input: CustomerID: 29715.0, CategoryName: ‘Clothing’, MaxDuration: 9

	ProductID	ProductName	CategoryName	SubcategoryName	collaborativeScore	categoryMatchScore	keywordSimilarityScore	RecommendationScore
0	715.0	Long-Sleeve Logo Jersey, L	Clothing	Jerseys	1068	3	2	2141
1	714.0	Long-Sleeve Logo Jersey, M	Clothing	Jerseys	808	3	3	1622
2	859.0	Half-Finger Gloves, M	Clothing	Gloves	748	3	4	1503
3	884.0	Short-Sleeve Classic Jersey, XL	Clothing	Jerseys	628	3	3	1262
4	864.0	Classic Vest, S	Clothing	Vests	616	3	3	1238
5	856.0	Men's Bib-Shorts, M	Clothing	Bib-Shorts	604	3	4	1215
6	854.0	Women's Tights, L	Clothing	Tights	576	3	3	1158
7	881.0	Short-Sleeve Classic Jersey, S	Clothing	Jerseys	500	3	2	1005
8	849.0	Men's Sports Shorts, M	Clothing	Shorts	384	3	3	774
9	861.0	Full-Finger Gloves, S	Clothing	Gloves	352	3	4	711

The final recommendation table is saved as a pd dataframe, this is the first few rows of the output. These are the recommended products for the consumer with CustomerID: 29715.0 who is interested in Clothing products and expected to receive the product in 9 days. The recommendation is determined by our recommendation ranking.

3. Things learned from the project:

3.1. **SQL & Neo4j work best together**: SQL is efficient for structured storage, while Neo4j excels at relationship-based recommendations.

3.2. **Indexing significantly improves performance**: Optimizing indexes in both SQL & Neo4j reduced query times a lot which makes fast recommendations with large data feasible.

3.3. **Graph-based recommendations outperform SQL joins**: Relationships in Neo4j enables faster query execution. Neo4j directly traverses relationships between nodes unlike SQL, which requires expensive joins. Neo4j supports multi-level recommendations (e.g., "Users who bought a bike also bought helmets, Users who bought helmets also bought gloves").

3.4. **Hybrid filtering is the most effective**: Combining collaborative (user-based) filtering with content-based filtering leads to better recommendations.

3.5. **Efficient data pipelines are essential**: Using Pandas to transform original and SQL preprocessed data into Neo4j queries ensures smooth integration and faster recommendation processing.

4. Future Work:

4.1. Real-Time Recommendation System:

Current Limitation: Our system processes recommendations in batch mode, meaning recommendations are generated based on past data rather than dynamically adapting to user behavior in real time.

Future Enhancement: Implement real-time recommendation updates using streaming data pipelines like Apache Kafka or Google Pub/Sub to update

recommendations instantly as users browse and interact. Additionally, use event-driven architectures to trigger recommendations on-the-fly, ensuring that users get dynamic and fresh suggestions based on their live activity. For example: a customer adds a bike to their cart, and the system instantly suggests matching accessories (helmet, water bottle, gloves) instead of waiting for periodic batch updates.

4.2. Improved Personalization with Machine Learning:

Current Limitation: The system relies solely on past purchase behavior and product attributes for recommendations.
Future Enhancement: Integrate machine learning (ML) models to analyze user preferences such as supplier and price preference, purchase frequency, and browsing history to make personalized recommendations. Train a collaborative filtering ML model (e.g.Neural Networks) to refine recommendations based on implicit user behavior. For example, a customer who frequently buys premium cycling gear may be recommended higher-end products, whereas a beginner cyclist may get budget-friendly suggestions.

4.3. Expanding to Multi-Language & Multi-Region Support:

Current Limitation: The system currently supports only one language(English) and does not adapt recommendations to different cultural preferences.
Future Enhancement: Incorporate multi-language support using ProductDescriptionCulture from the AdventureWorks dataset. Then we can customize recommendations by region, considering cultural preferences, seasonal trends, and regional popularity. We can also Introduce currency conversions and localized pricing adjustments in SQL queries to tailor recommendations for global audiences. For example, a customer in Japan may see different bicycle models or brands recommended compared to a customer in the U.S., based on region-specific sales trends.