NOSQL PROJECT

REVIEW- 2

PRODUCT RECOMMENDATION USING NEO4J

20MIA1046 Vishakhaa S

20MIA1153 Yashwanth Sarathi S P

Datasets:

Orders.csv
Employees.csv
Categories.csv
Customers.csv
Products.csv
Suppliers.csv

Creating Orders:

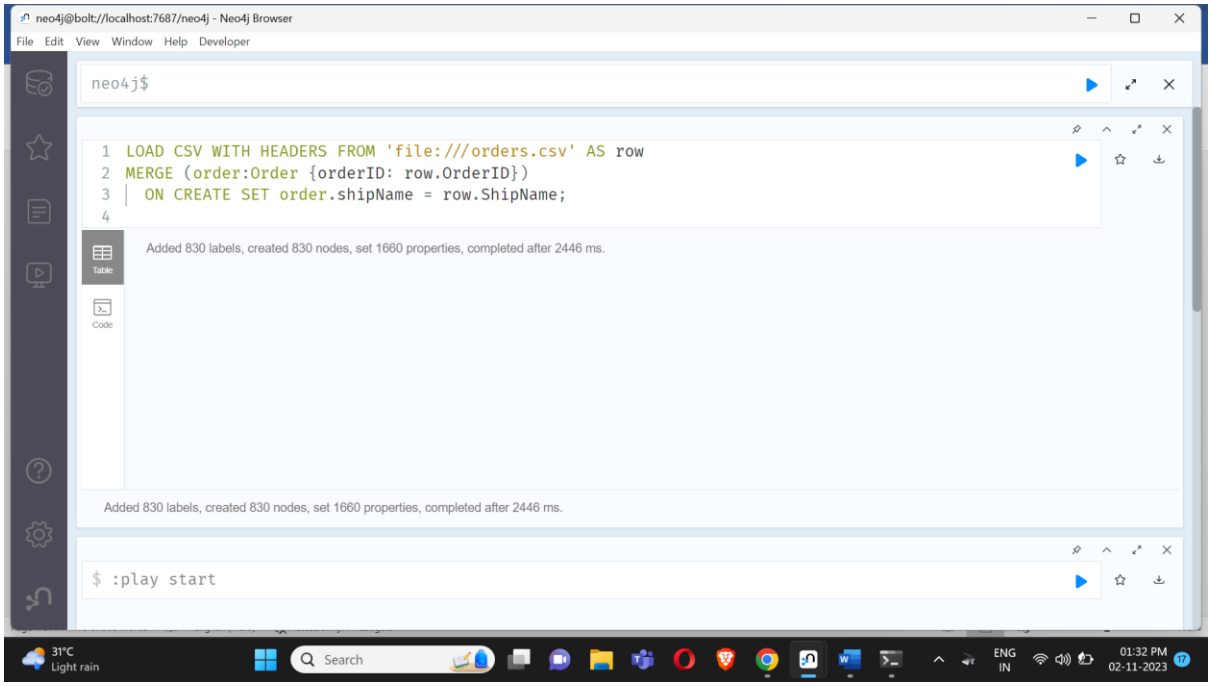
In this section, data from a CSV file containing order information is loaded into the Neo4j database. Each row in the CSV file corresponds to an order, and a node labeled as Order is created for each order. The orderID property is set based on the data in the CSV file.

// Create orders

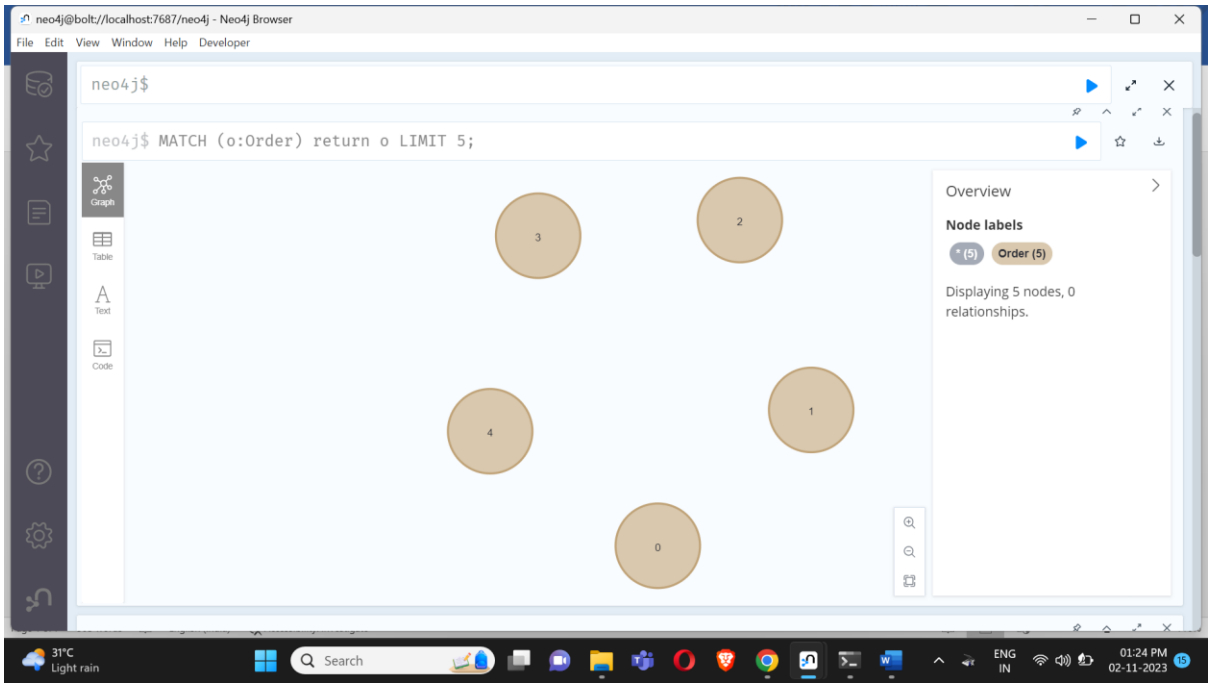
```
LOAD CSV WITH HEADERS FROM file:///orders.csv' AS row

MERGE (order:Order {orderID: row.OrderID})

ON CREATE SET order.shipName = row.ShipName;
```



MATCH (o:Order) return o LIMIT 5;



Creating Products:

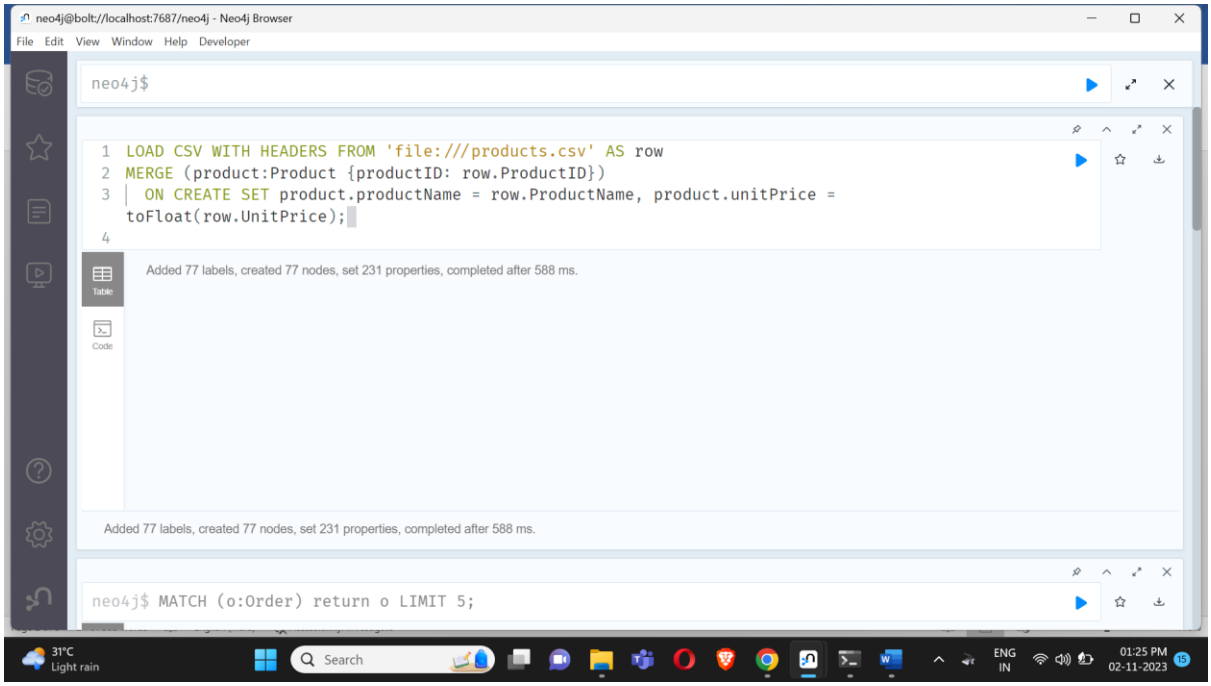
Similar to the previous section, this part loads data from a CSV file containing product information. For each product in the CSV file, a node labeled as Product is created in the Neo4j database. The product's name and unit price are set as properties of the node.

// Create products

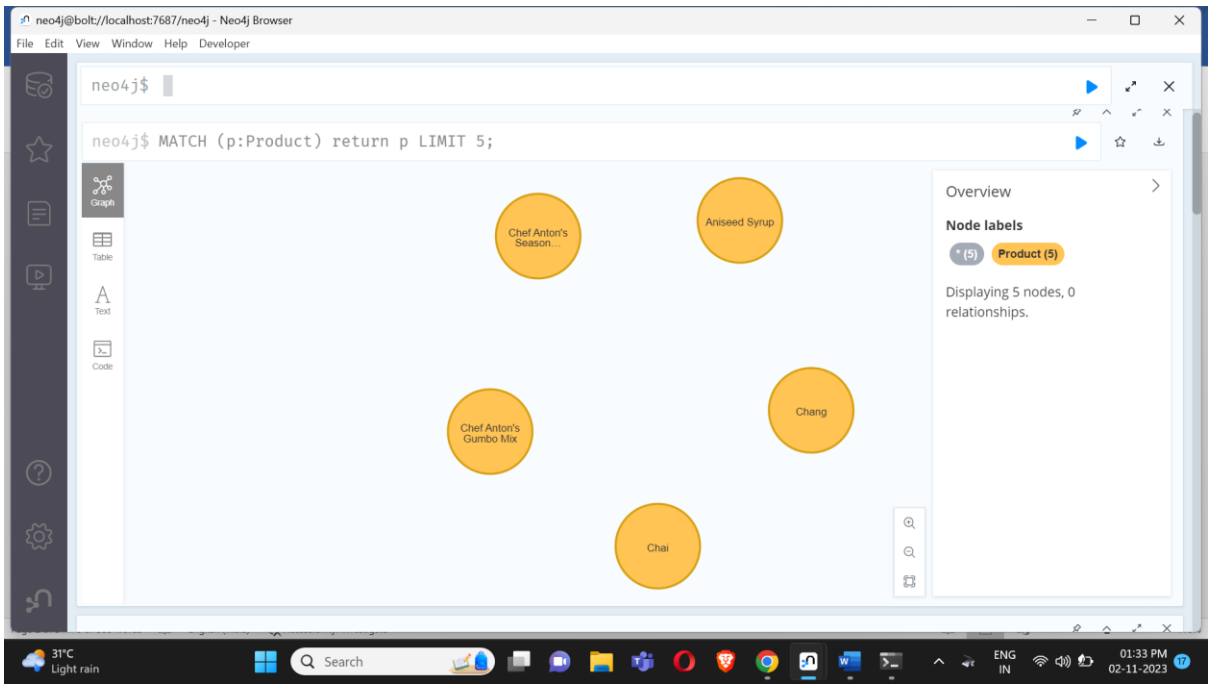
LOAD CSV WITH HEADERS FROM file:///products.csv' AS row

MERGE (product:Product {productID: row.ProductID})

ON CREATE SET product.productName = row.ProductName, product.unitPrice = toFloat(row.UnitPrice);



MATCH (p:Product) return p LIMIT 5;



Creating Suppliers, Employees, and Categories:

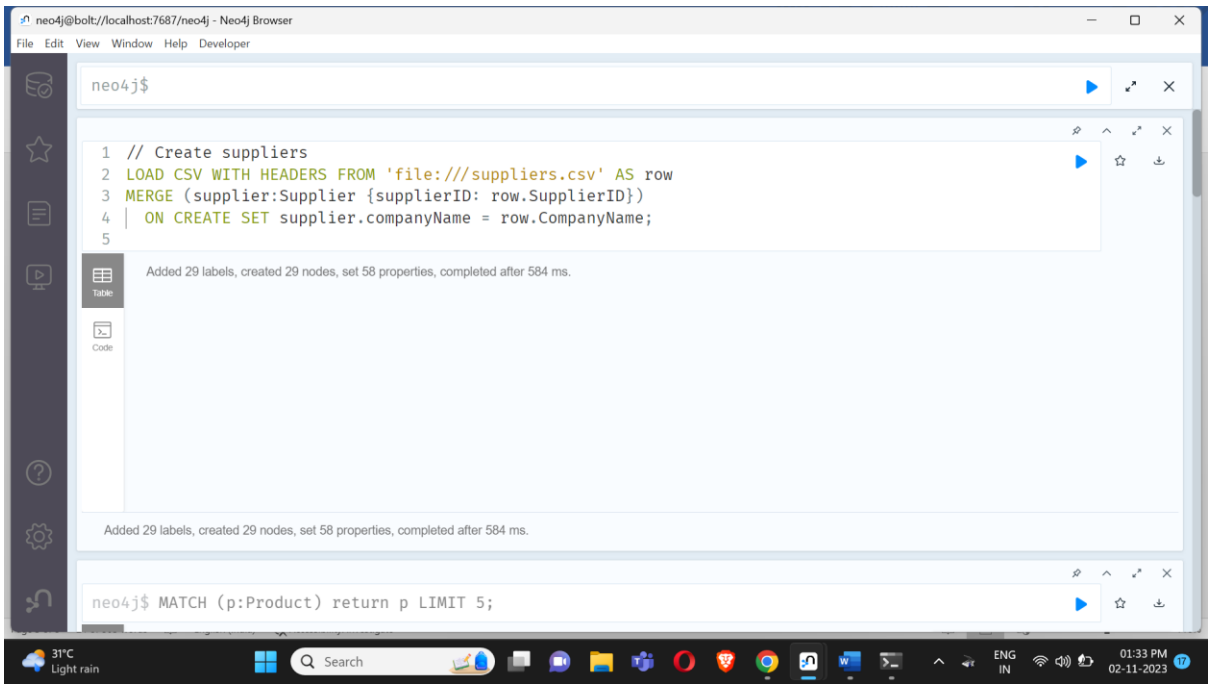
These sections follow a similar pattern. They load data from CSV files related to suppliers, employees, and product categories. Nodes labeled as Supplier, Employee, and Category are created for each entry in the respective CSV files. The properties of these nodes, such as supplierID, employeeID, companyName, etc., are set based on the CSV data.

// Create suppliers

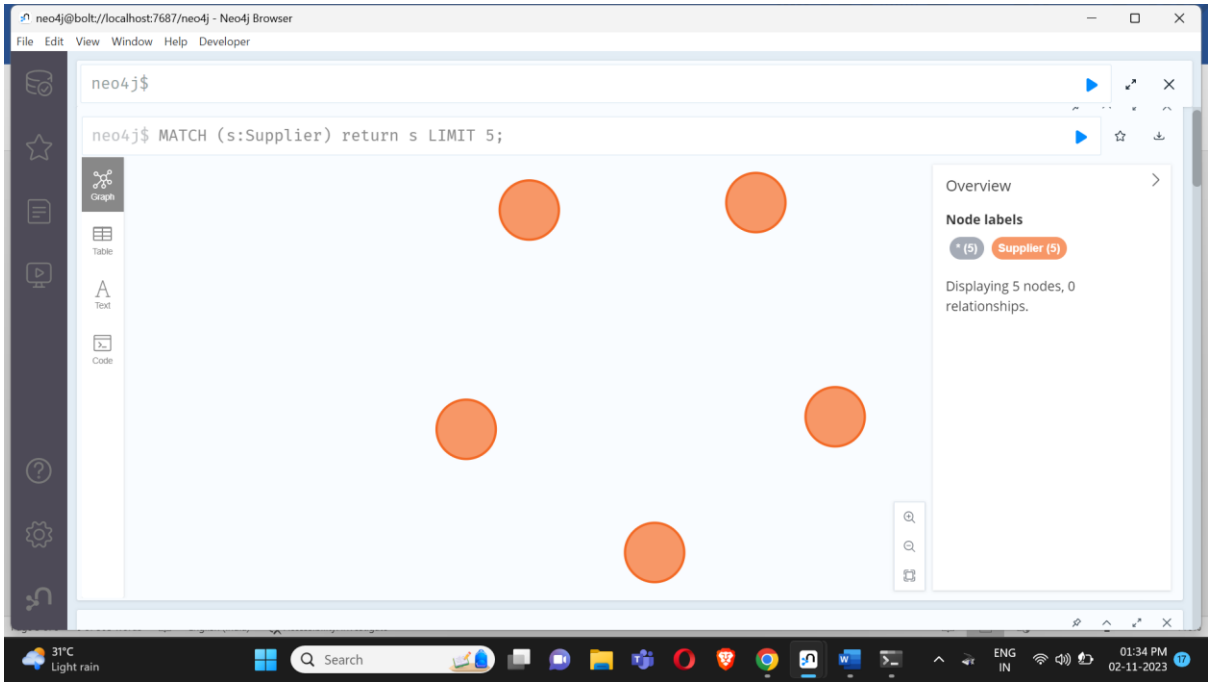
LOAD CSV WITH HEADERS FROM file:///suppliers.csv' AS row

MERGE (supplier:Supplier {supplierID: row.SupplierID})

ON CREATE SET supplier.companyName = row.CompanyName;



MATCH (s:Supplier) return s LIMIT 5;

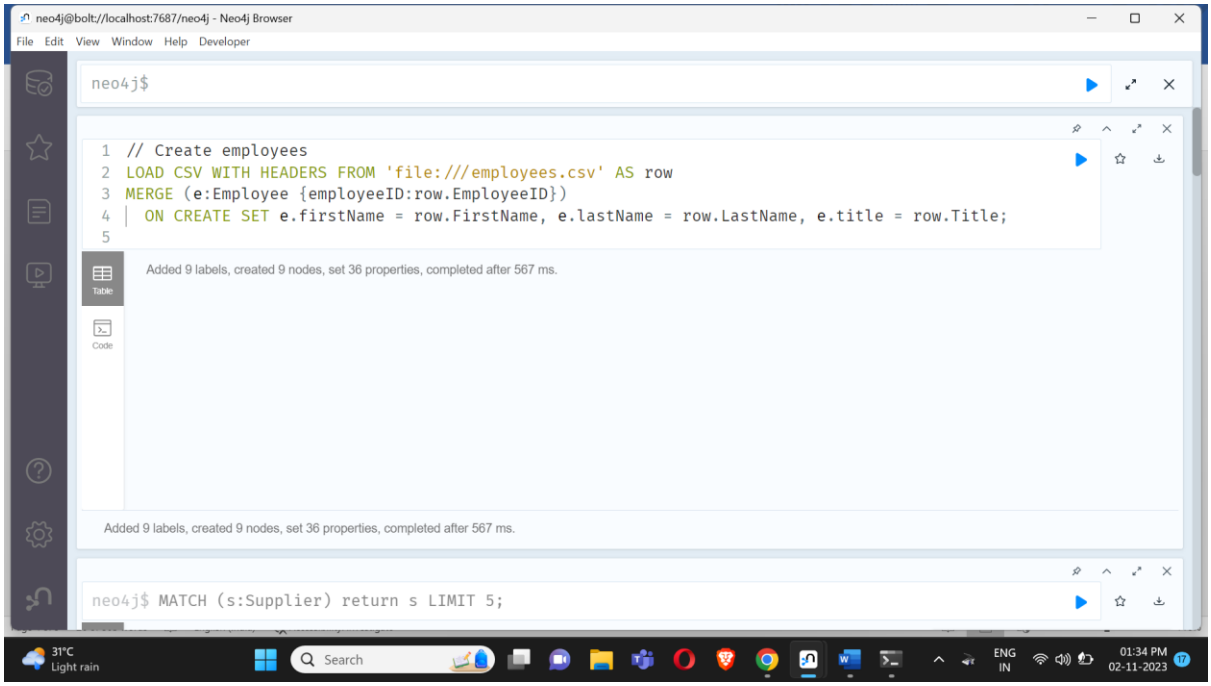


// Create employees

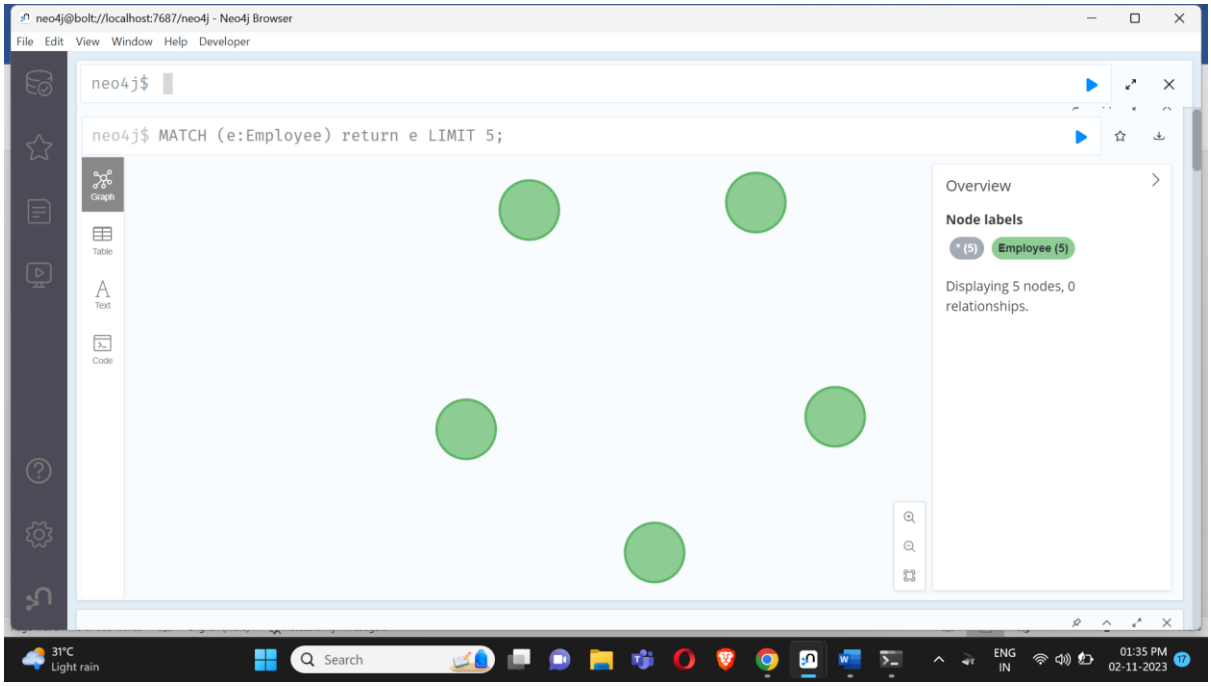
LOAD CSV WITH HEADERS FROM file:///employees.csv' AS row

MERGE (e:Employee {employeeID:row.EmployeeID})

ON CREATE SET e.firstName = row.FirstName, e.lastName = row.LastName, e.title = row.Title;



MATCH (e:Employee) return e LIMIT 5;

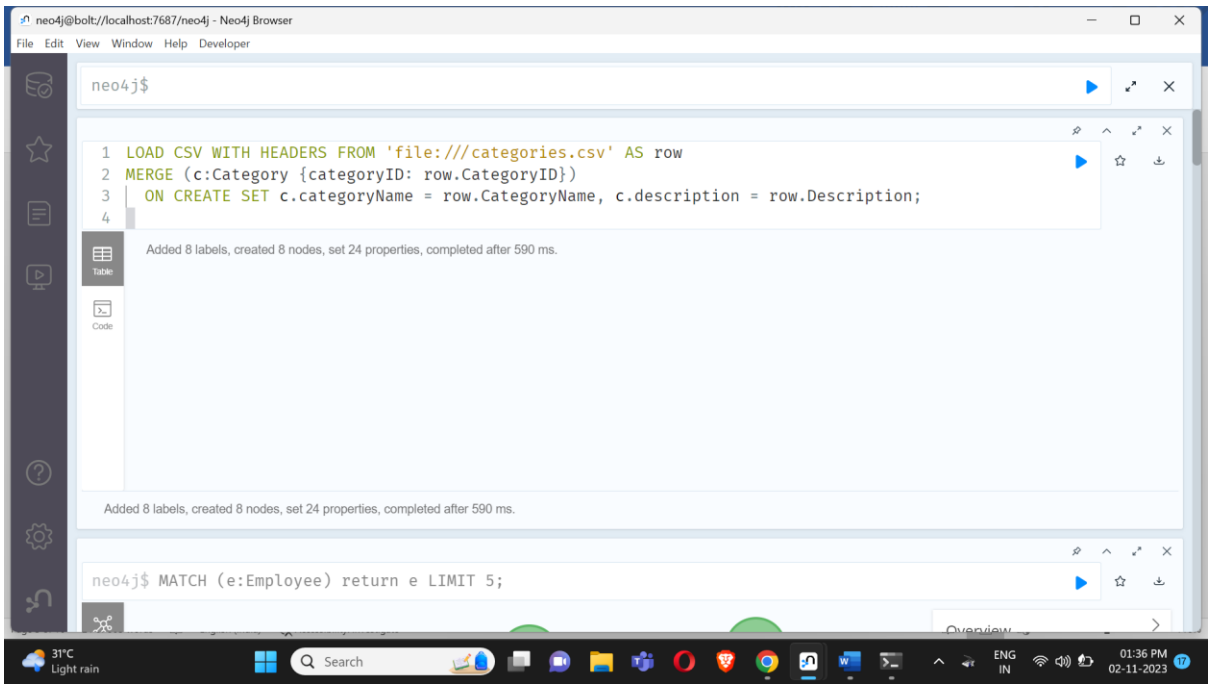


// Create categories

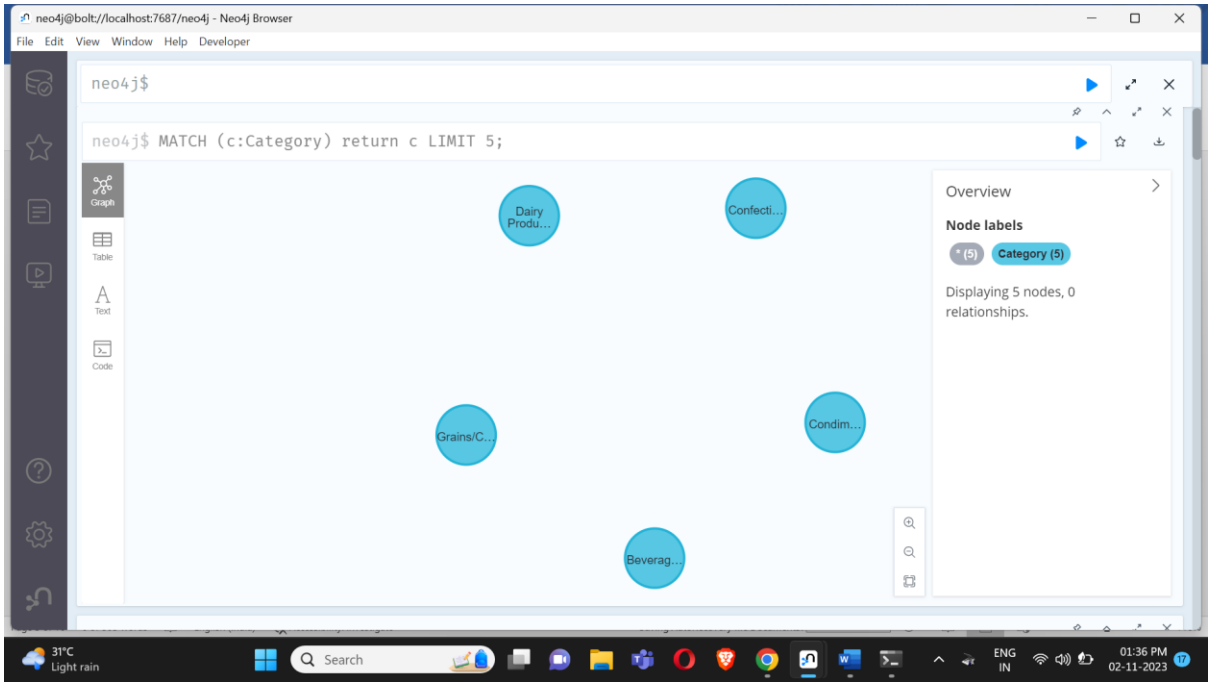
LOAD CSV WITH HEADERS FROM file:///categories.csv' AS row

MERGE (c:Category {categoryID: row.CategoryID})

ON CREATE SET c.categoryName = row.CategoryName, c.description = row.Description;



MATCH (c:Category) return c LIMIT 5;

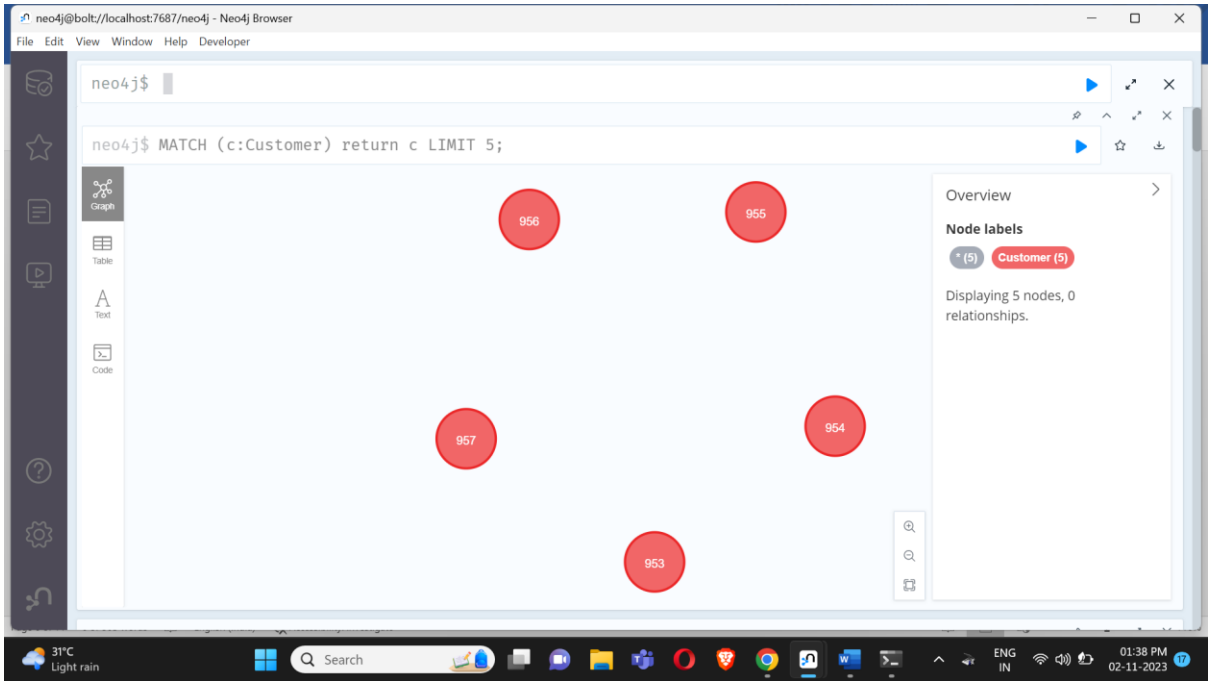


Creating Customers:

This section loads customer data from a CSV file and creates nodes labeled as Customer in the Neo4j database. The properties of these nodes, such as customerID, companyName, fax, and phone, are set based on the CSV data.

//create customers
LOAD CSV WITH HEADERS FROM "file:///customers.csv" AS row
CREATE (:Customer {companyName: row.CompanyName, customerID: row.CustomerID, fax: row.Fax, phone: row.Phone});

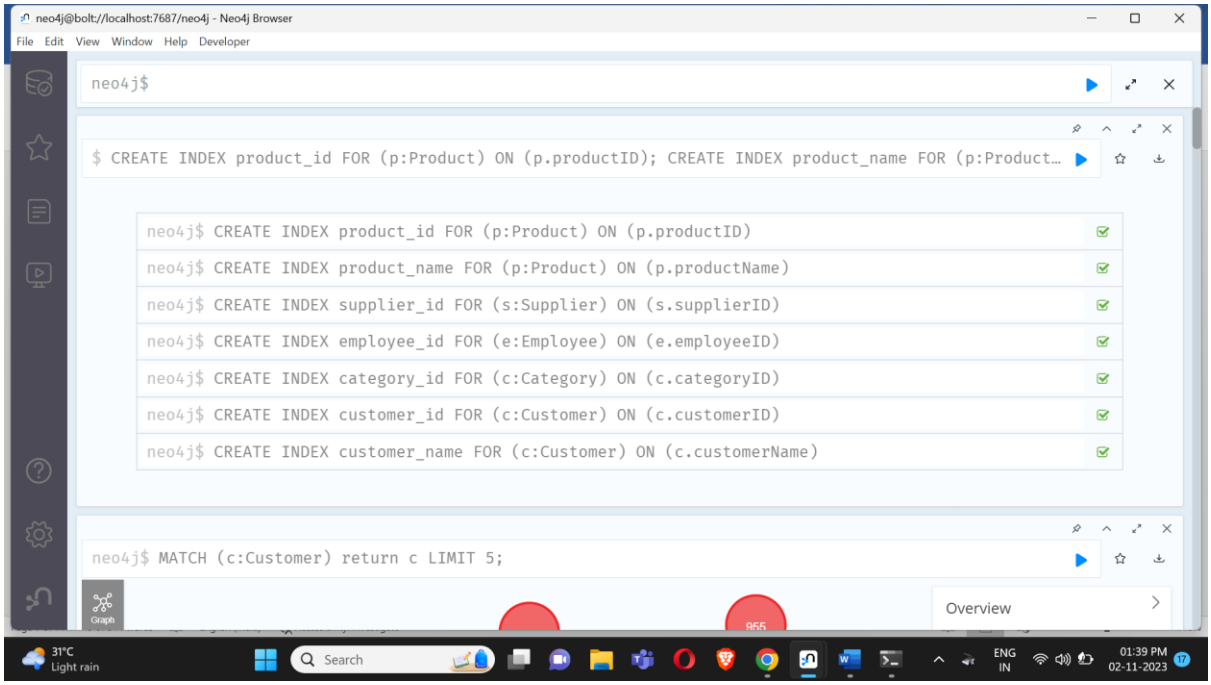
MATCH (c:Customer) return c LIMIT 5;



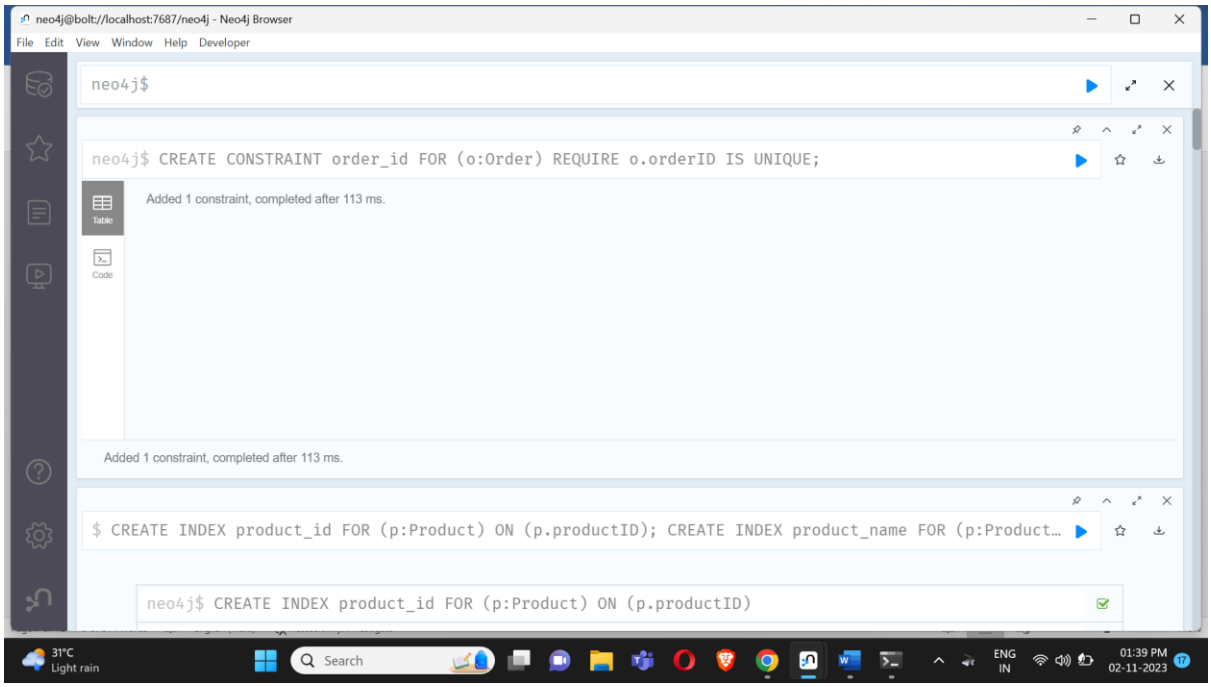
Creating Indexes and Constraints:

Indexes and constraints are essential for optimizing database performance and ensuring data integrity. In this section, various indexes are created on properties of nodes to speed up data retrieval. Additionally, a constraint is set on the orderID property to ensure that order IDs are unique.

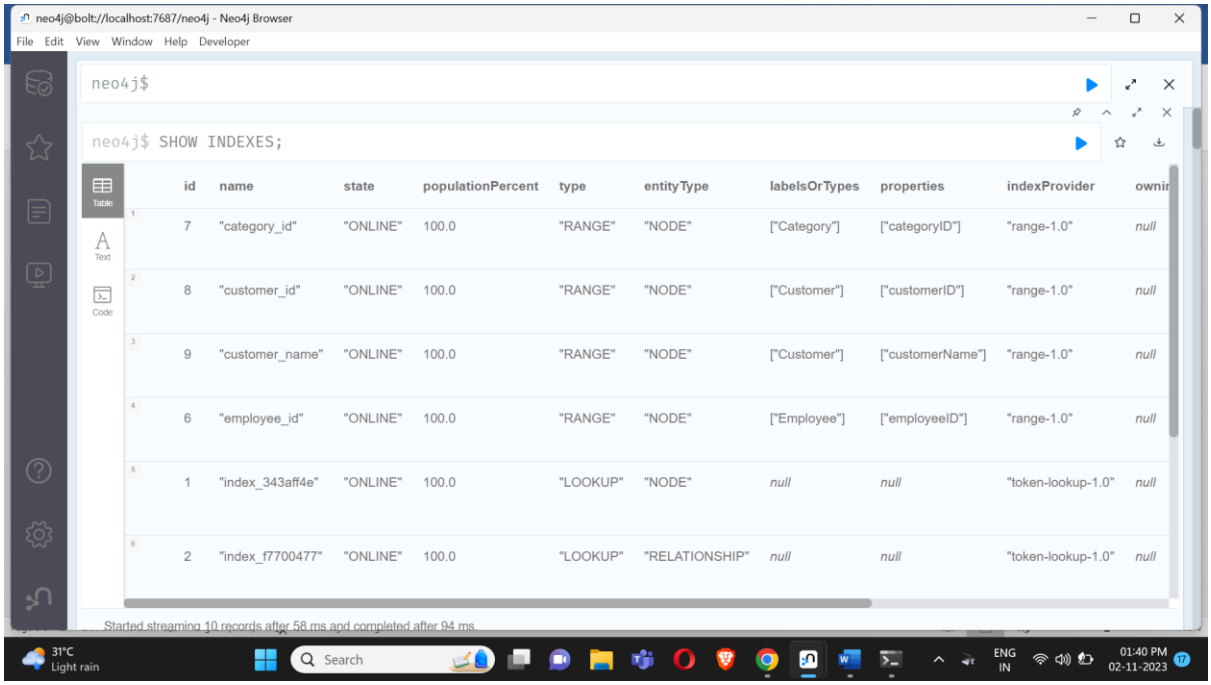
```
CREATE INDEX product_id FOR (p:Product) ON (p.productID);
CREATE INDEX product_name FOR (p:Product) ON (p.productName);
CREATE INDEX supplier_id FOR (s:Supplier) ON (s.supplierID);
CREATE INDEX employee_id FOR (e:Employee) ON (e.employeeID);
CREATE INDEX category_id FOR (c:Category) ON (c.categoryID);
CREATE INDEX customer_id FOR (c:Customer) ON (c.customerID);
CREATE INDEX customer_name FOR (c:Customer) ON (c.customerName);
```



CREATE CONSTRAINT order_id FOR (o:Order) REQUIRE o.orderID IS UNIQUE;



SHOW INDEXES;



Creating Relationships between Orders and Products:

This part establishes relationships between Order nodes and Product nodes. It represents that an order contains one or more products. Information about the unit price and quantity of each product within an order is also stored as properties on the relationship.

// Create relationships between orders and products

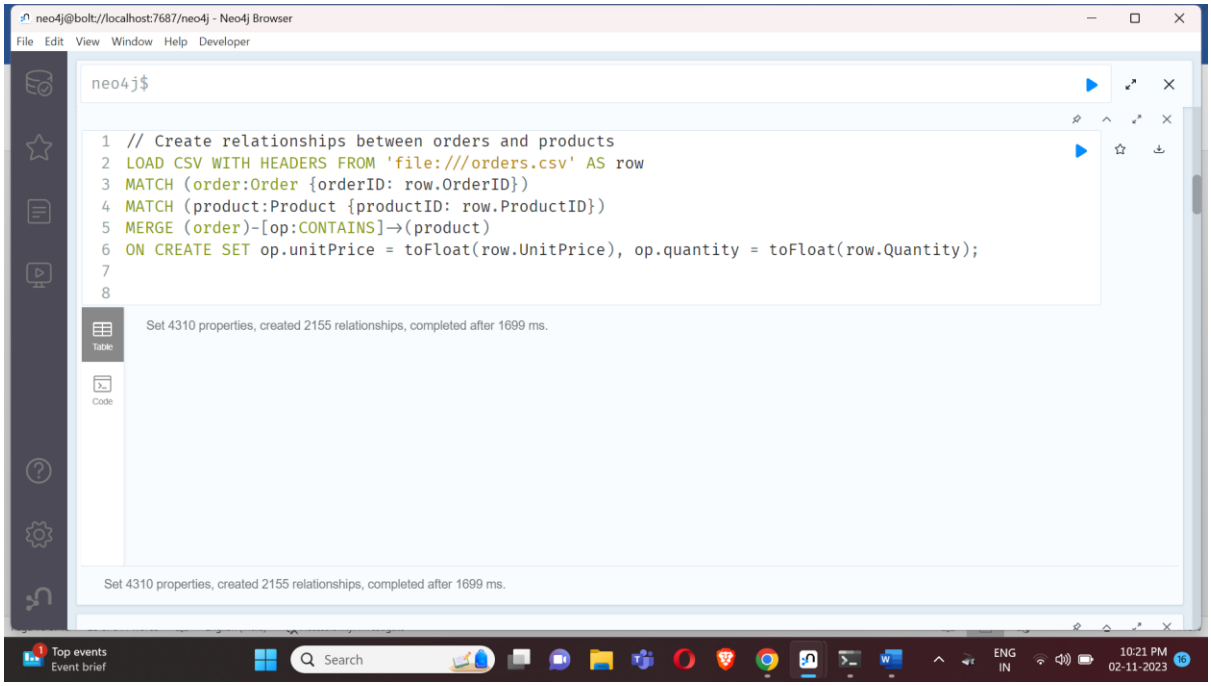
LOAD CSV WITH HEADERS FROM file:///orders.csv' AS row

MATCH (order:Order {orderID: row.OrderID})

MATCH (product:Product {productID: row.ProductID})

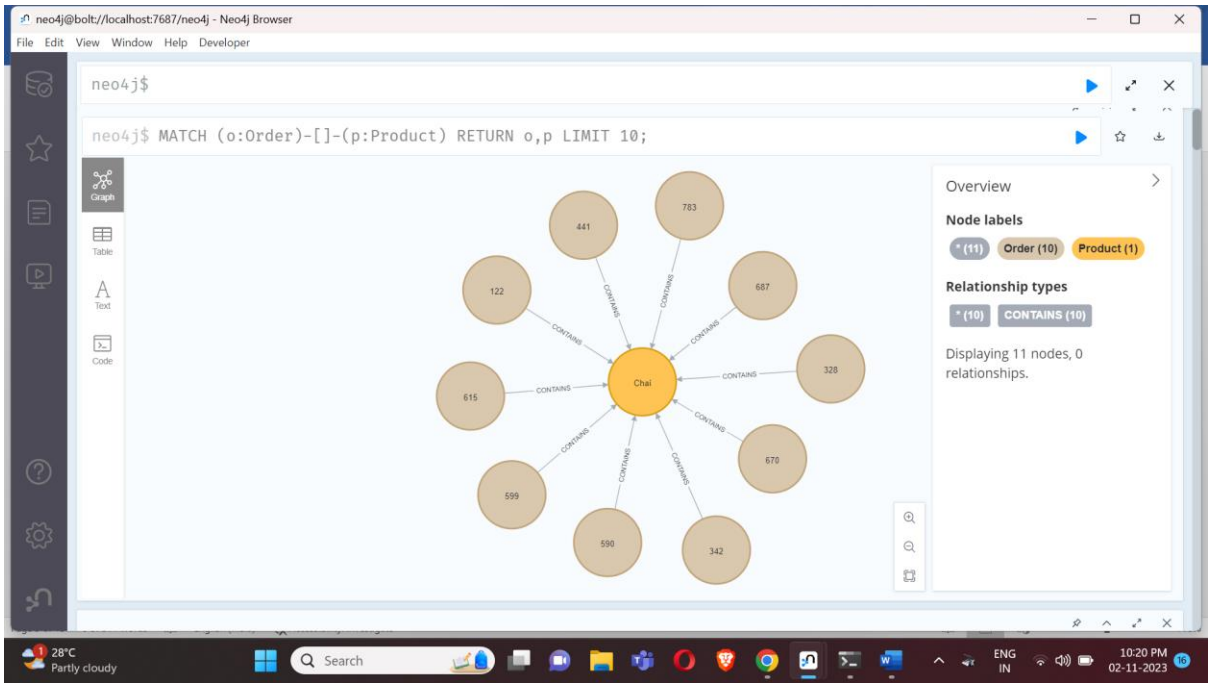
MERGE (order)-[op:CONTAINS]->(product)

ON CREATE SET op.unitPrice = toFloat(row.UnitPrice), op.quantity = toFloat(row.Quantity);



MATCH (o:Order)-[]-(p:Product)

RETURN o,p LIMIT 10;



Creating Relationships between Orders and Employees:

Relationships are created between Order nodes and Employee nodes, indicating which employee is responsible for selling a particular order. This information can be used to analyze the performance of employees in terms of sales.

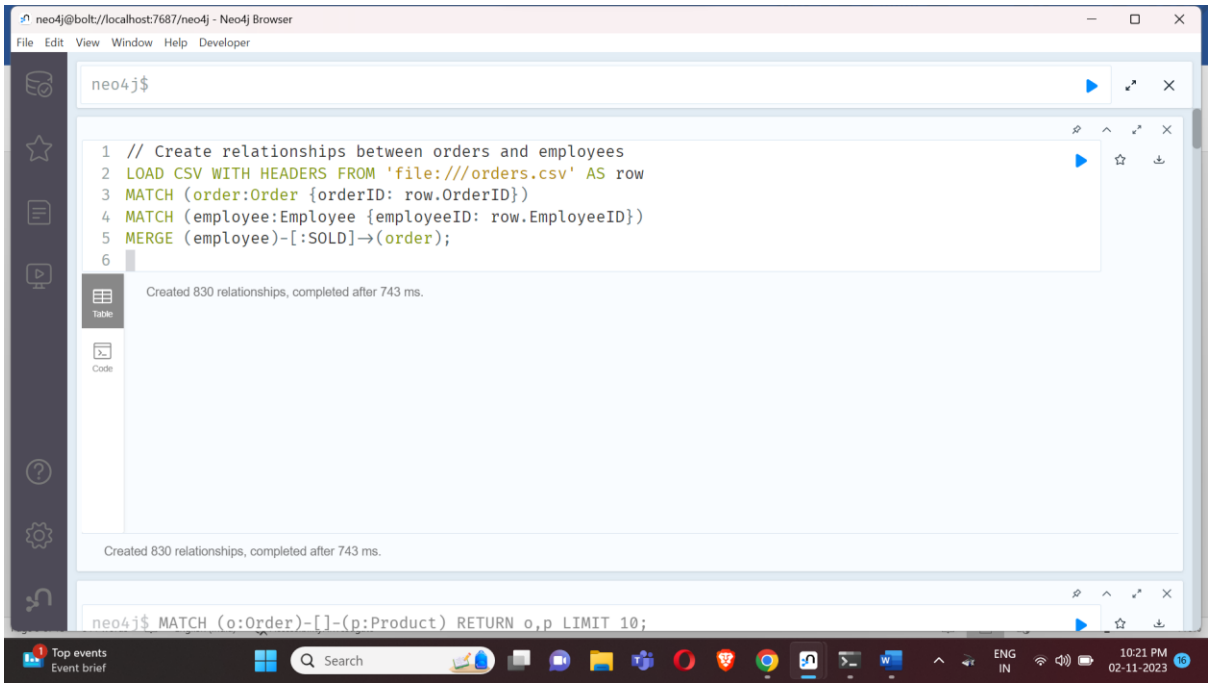
// Create relationships between orders and employees

LOAD CSV WITH HEADERS FROM file:///orders.csv' AS row

MATCH (order:Order {orderID: row.OrderID})

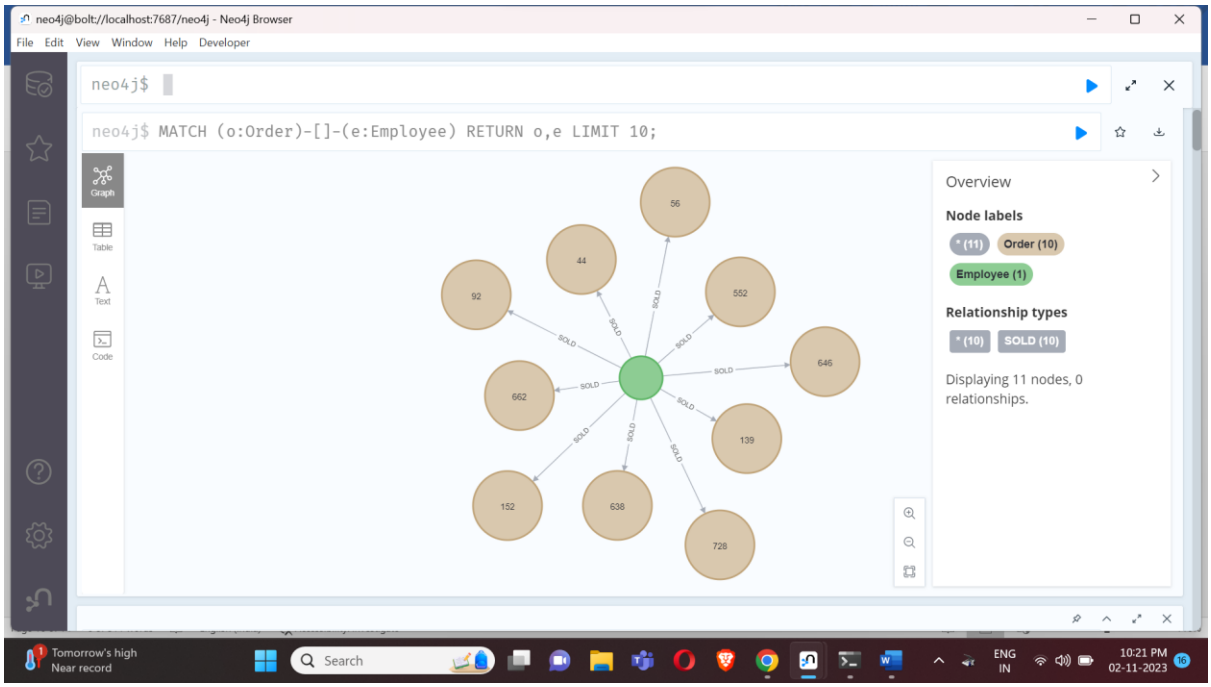
MATCH (employee:Employee {employeeID: row.EmployeeID})

MERGE (employee)-[:SOLD]->(order);



MATCH (o:Order)-[]-(e:Employee)

RETURN o,e LIMIT 10;

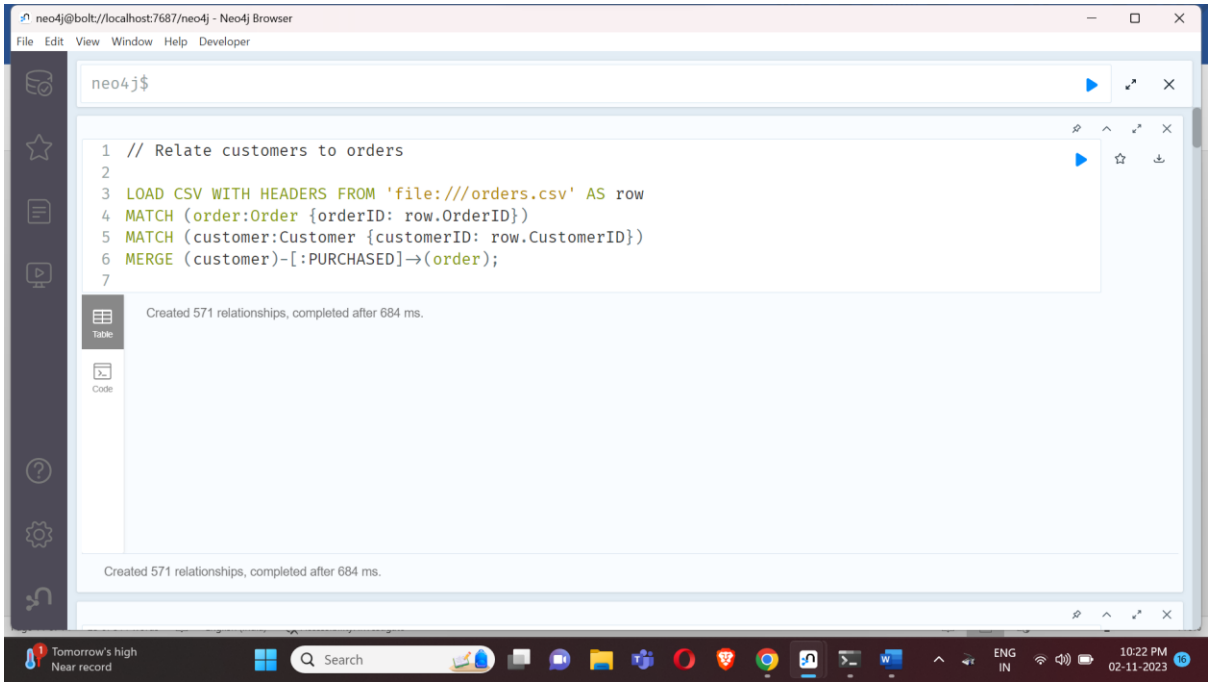


Relating Customers to Orders:

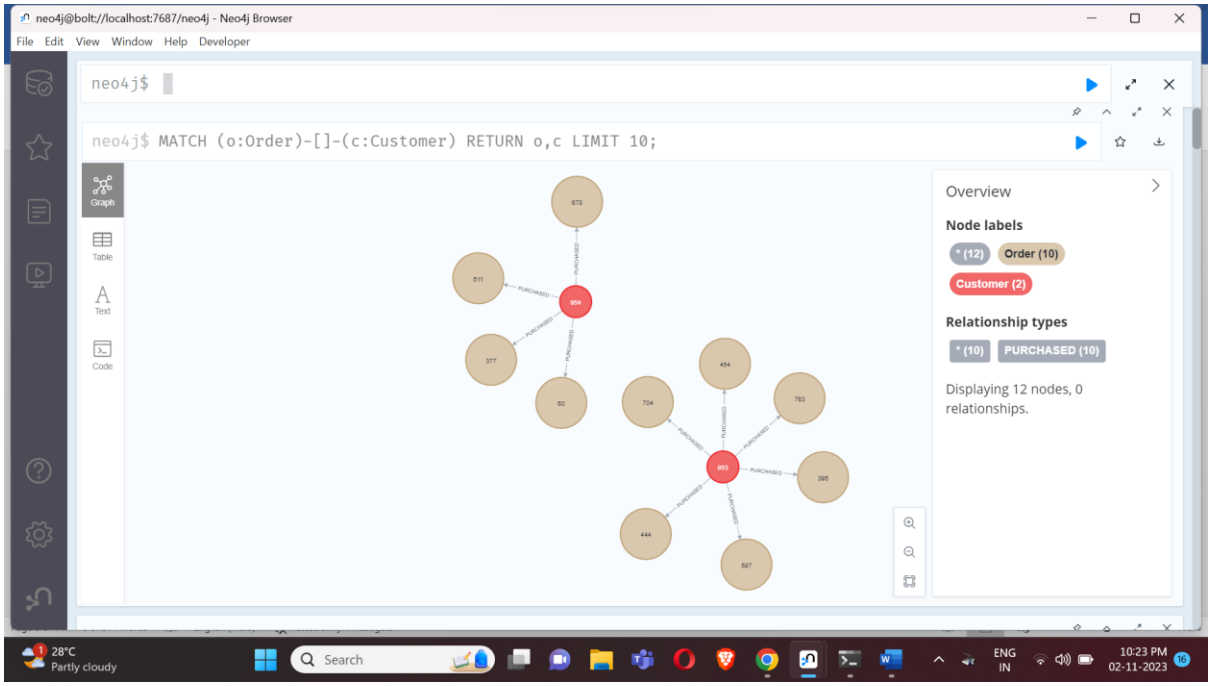
This section links Customer nodes to their respective Order nodes. It signifies that a customer made a purchase by placing an order. This information is useful for understanding customer behavior and preferences.

// Relate customers to orders

LOAD CSV WITH HEADERS FROM file:///orders.csv' AS row
MATCH (order:Order {orderID: row.OrderID})
MATCH (customer:Customer {customerID: row.CustomerID})
MERGE (customer)-[:PURCHASED]->(order);



MATCH (o:Order)-[]-(c:Customer)
RETURN o,c LIMIT 10;



Creating Relationships between Products and Suppliers:

Relationships are established between Product nodes and Supplier nodes, indicating which supplier supplies which products. This information is valuable for tracking the sources of products.

// Create relationships between products and suppliers

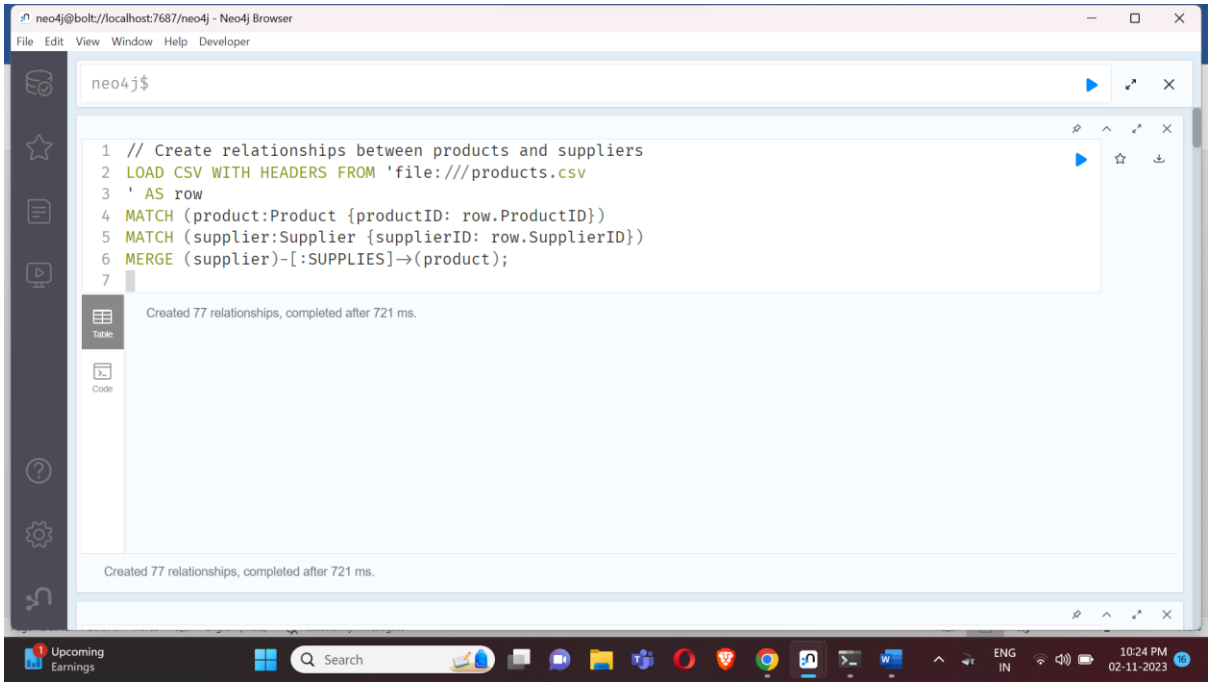
LOAD CSV WITH HEADERS FROM file:///products.csv

' AS row

MATCH (product:Product {productID: row.ProductID})

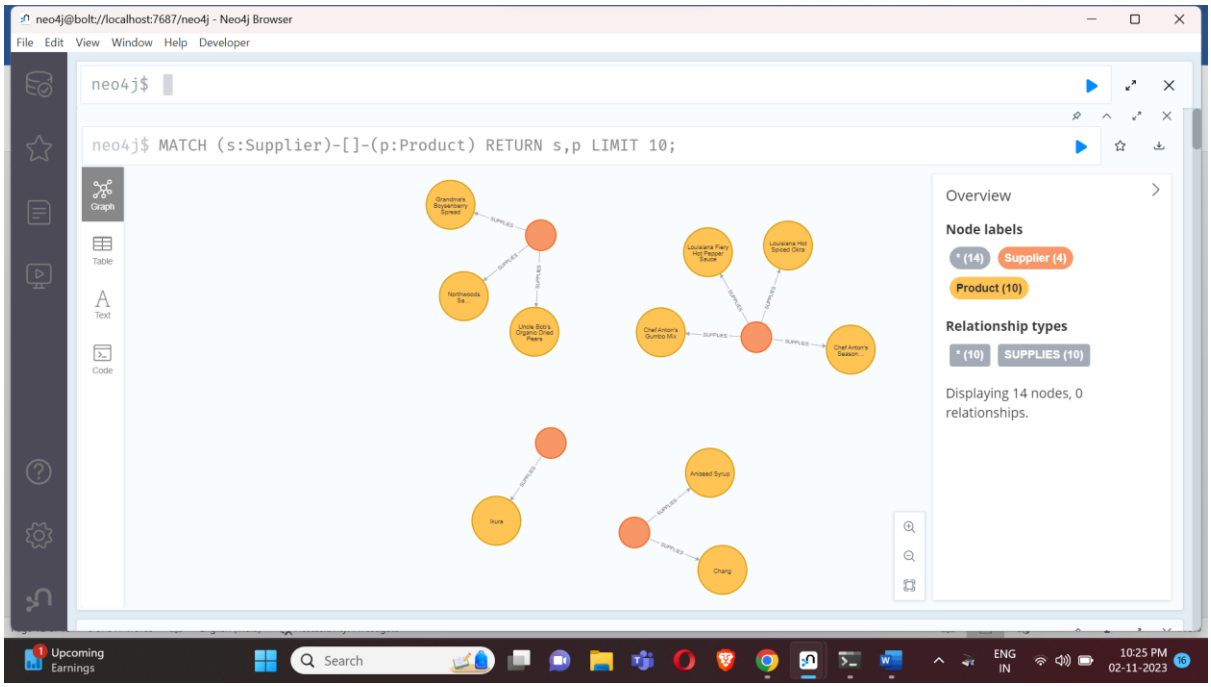
MATCH (supplier:Supplier {supplierID: row.SupplierID})

MERGE (supplier)-[:SUPPLIES]->(product);



MATCH (s:Supplier)-[]-(p:Product)

RETURN s,p LIMIT 10;



Creating Relationships between Products and Categories:

This section associates Product nodes with Category nodes, representing that a product belongs to a particular category. This categorization is useful for organizing and querying products.

// Create relationships between products and categories

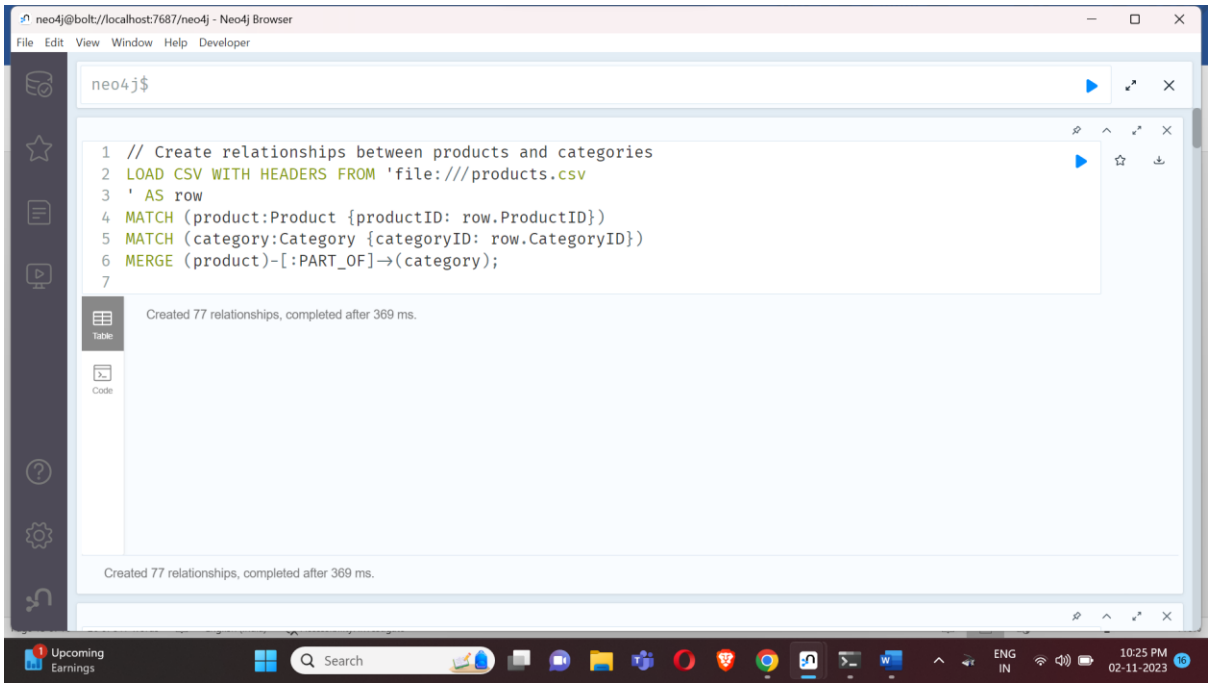
LOAD CSV WITH HEADERS FROM file:///products.csv

' AS row

MATCH (product:Product {productID: row.ProductID})

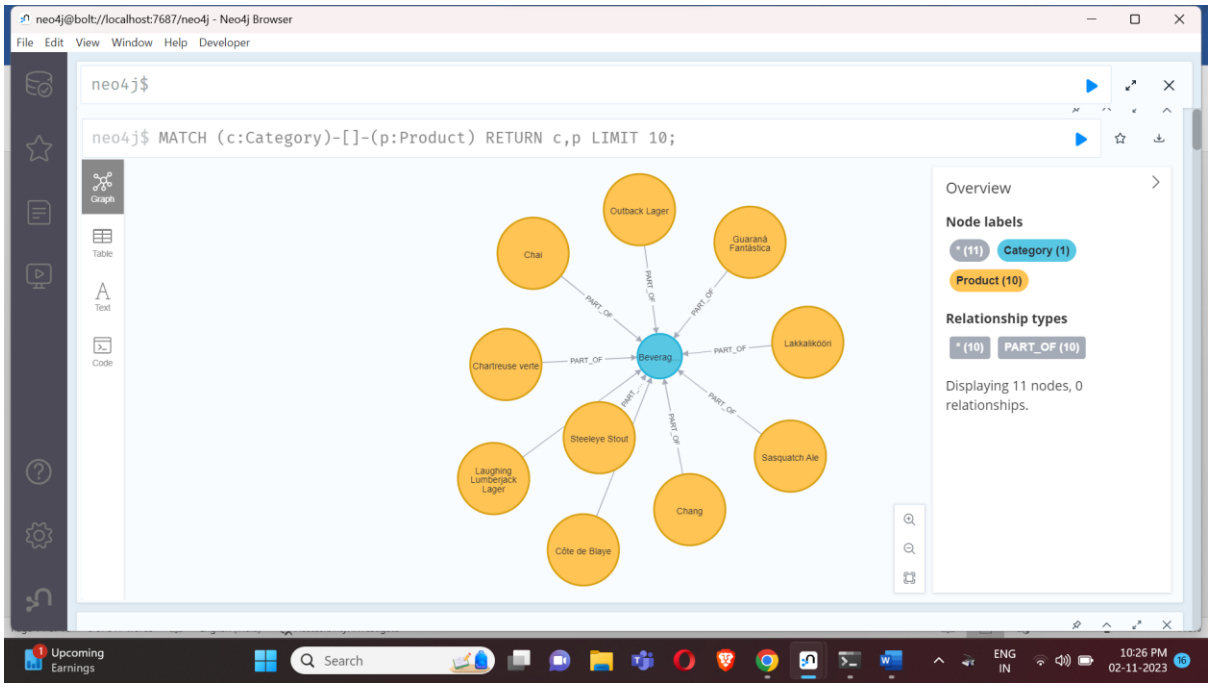
MATCH (category:Category {categoryID: row.CategoryID})

MERGE (product)-[:PART_OF]->(category);



MATCH (c:Category)-[]-(p:Product)

RETURN c,p LIMIT 10;



Creating Relationships between Employees (Reporting Hierarchy):

Relationships are created between Employee nodes to represent the reporting hierarchy within the organization. An Employee node is linked to their respective manager using the REPORTS_TO relationship. This hierarchy is essential for understanding the organization's structure.

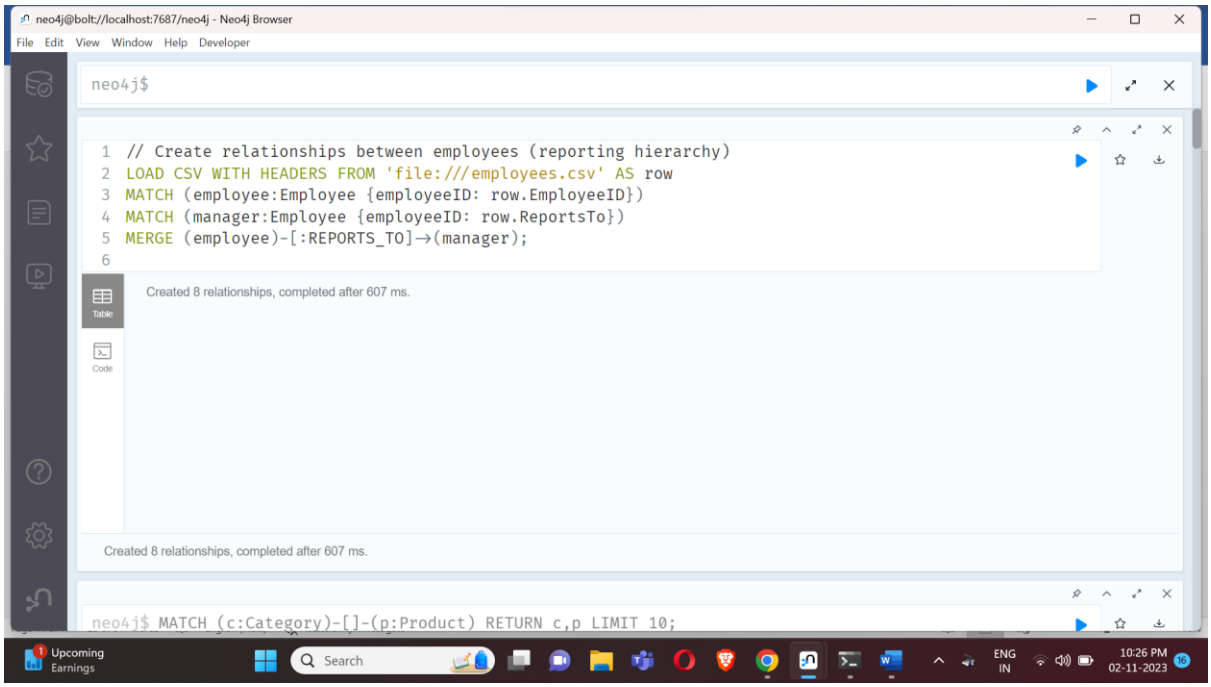
// Create relationships between employees (reporting hierarchy)

LOAD CSV WITH HEADERS FROM file:///employees.csv' AS row

MATCH (employee:Employee {employeeID: row.EmployeeID})

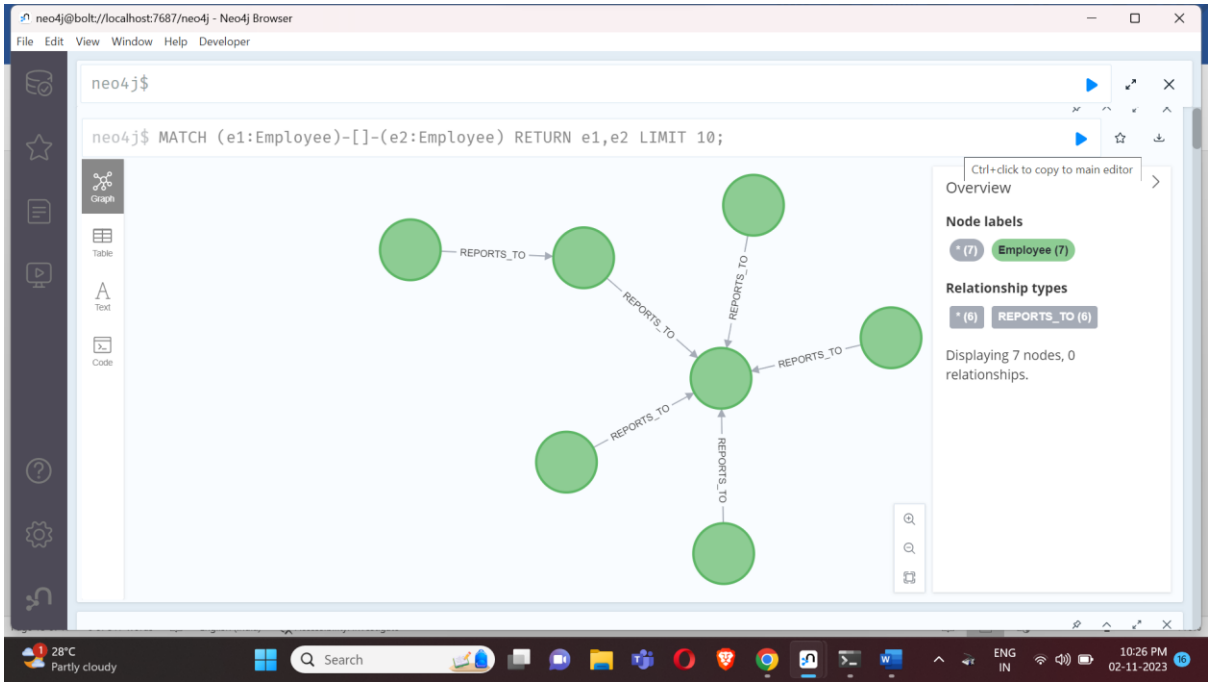
MATCH (manager:Employee {employeeID: row.ReportsTo})

MERGE (employee)-[:REPORTS_TO]->(manager);



MATCH (e1:Employee)-[]-(e2:Employee)

RETURN e1,e2 LIMIT 10;



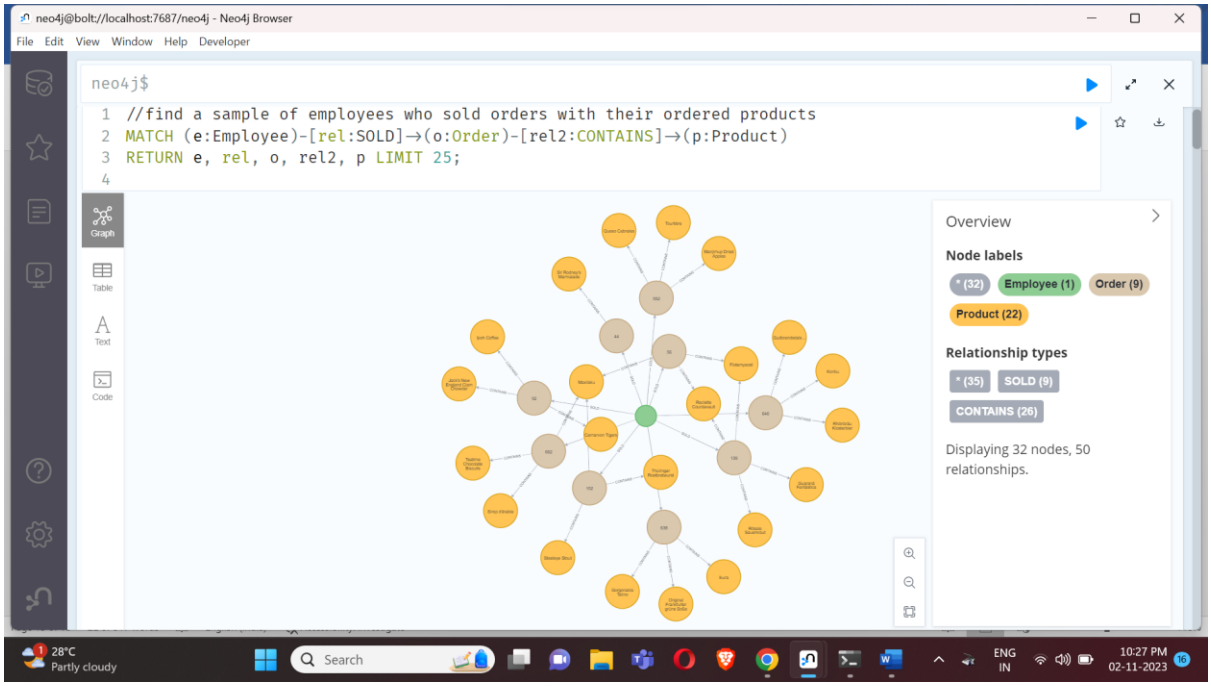
ANALYSIS:

This section contains various Cypher queries for analyzing the data in the Neo4j graph database. The queries provide insights into employee performance, product information, organizational structure, and order statistics.

//find a sample of employees who sold orders with their ordered products

MATCH (e:Employee)-[rel:SOLD]->(o:Order)-[rel2:CONTAINS]->(p:Product)

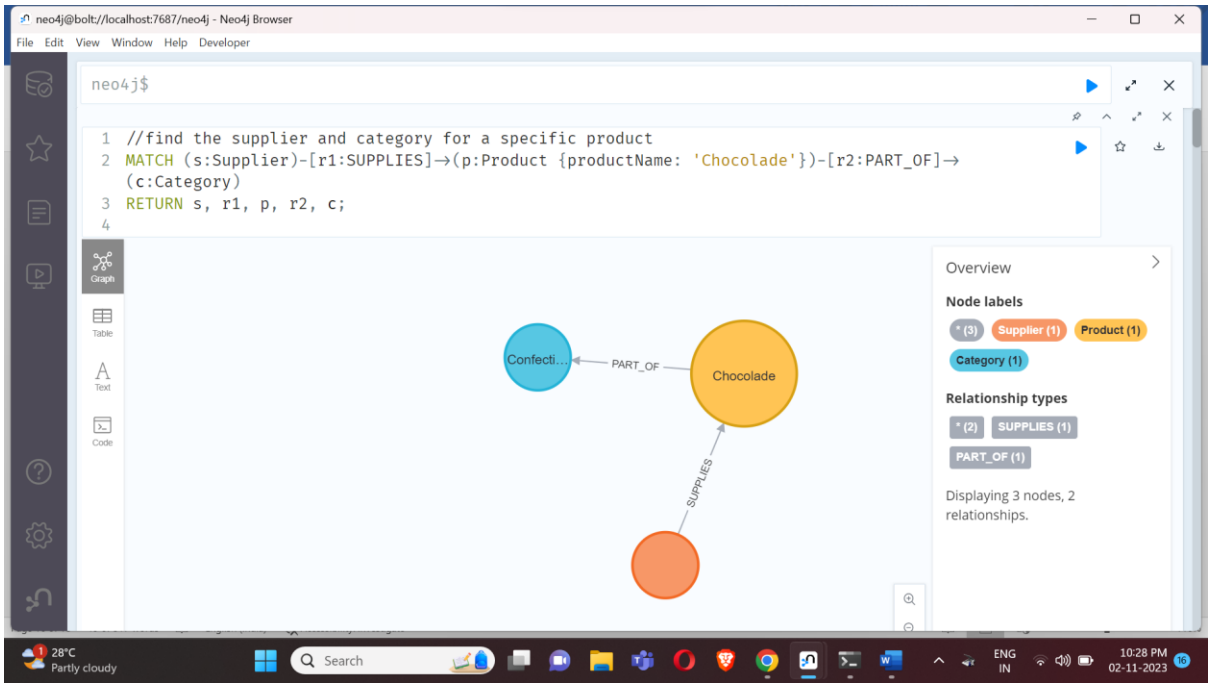
RETURN e, rel, o, rel2, p LIMIT 25;



//find the supplier and category for a specific product

MATCH (s:Supplier)-[r1:SUPPLIES]->(p:Product {productName: 'Chocolade'})-[r2:PART_OF]->(c:Category)

RETURN s, r1, p, r2, c;



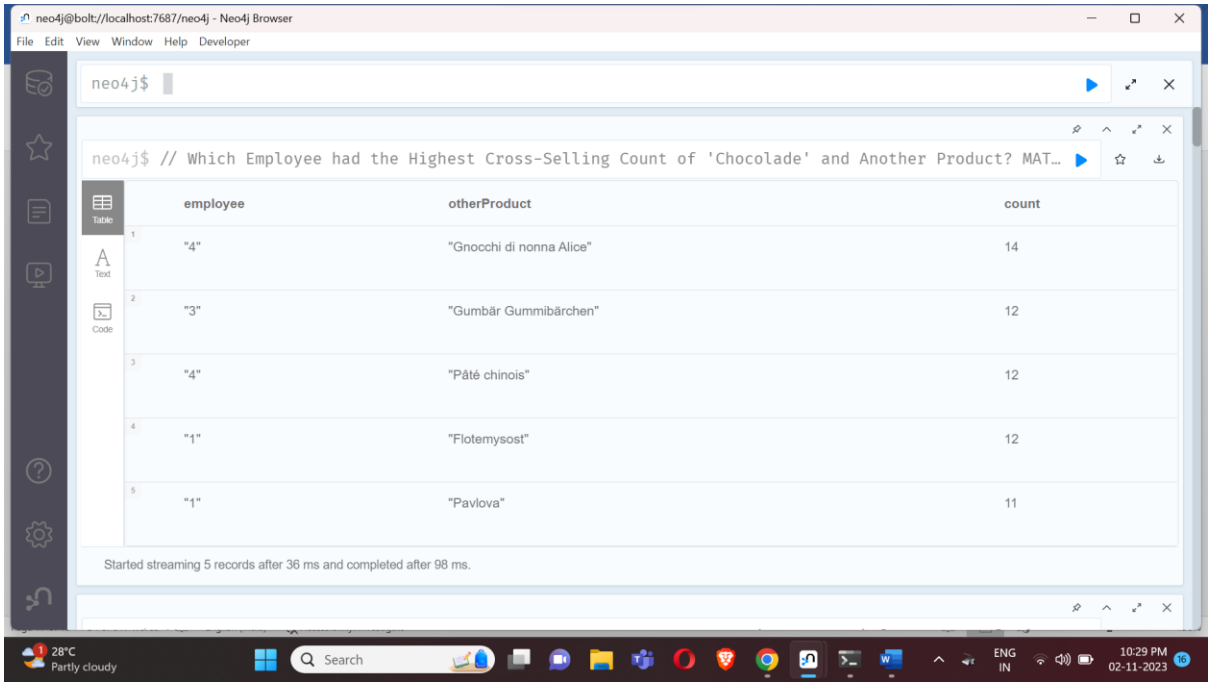
// Which Employee had the Highest Cross-Selling Count of 'Chocolade' and Another Product?

MATCH (choc:Product {productName:'Chocolade'})<-[:CONTAINS]-(:Order)<-[:SOLD]-(employee), (employee)-[:SOLD]->(o2)-[:CONTAINS]->(other:Product)

RETURN employee.employeeID as employee, other.productName as otherProduct, count(distinct o2) as count

ORDER BY count DESC

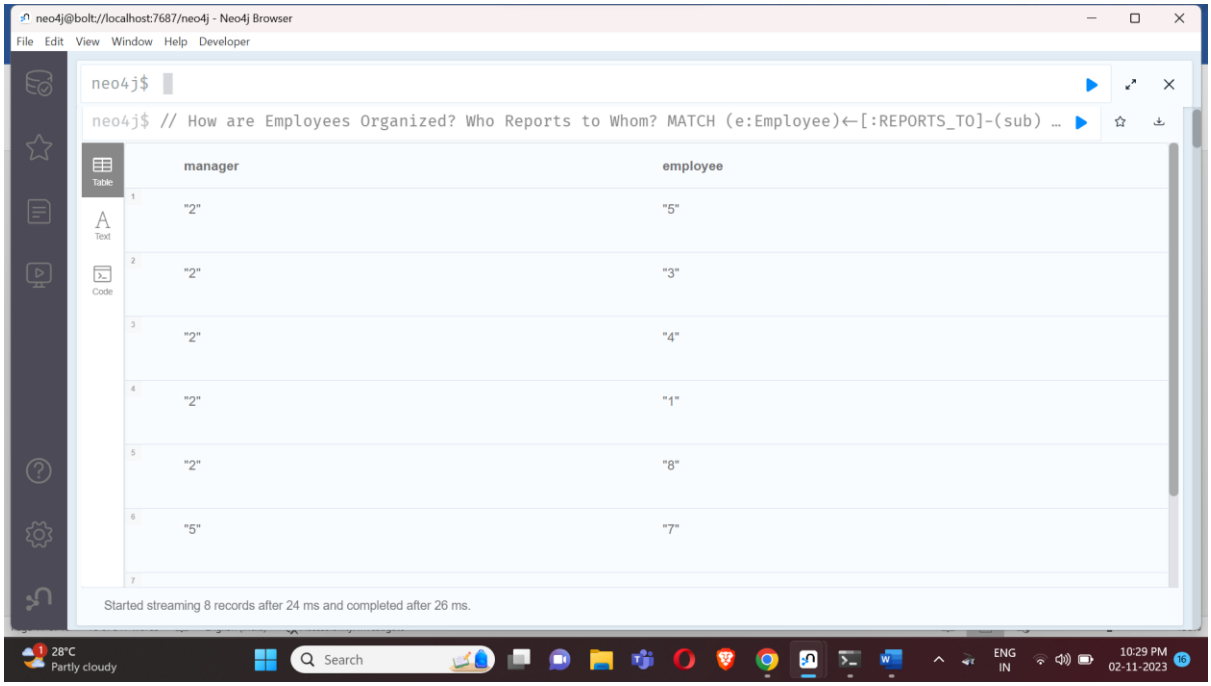
LIMIT 5;



// How are Employees Organized? Who Reports to Whom?

MATCH (e:Employee)<-[:REPORTS_TO]-(sub)

RETURN e.employeeID AS manager, sub.employeeID AS employee;



// Which Employees Report to Each Other Indirectly?

MATCH path = (e:Employee)<-[:REPORTS_TO*]-(sub)
WITH e, sub, [person in NODES(path) | person.employeeID][1..-1] AS path
RETURN e.employeeID AS manager, path as middleManager, sub.employeeID AS employee
ORDER BY size(path);

// How Many Orders were Made by Each Part of the Hierarchy?

MATCH (e:Employee)

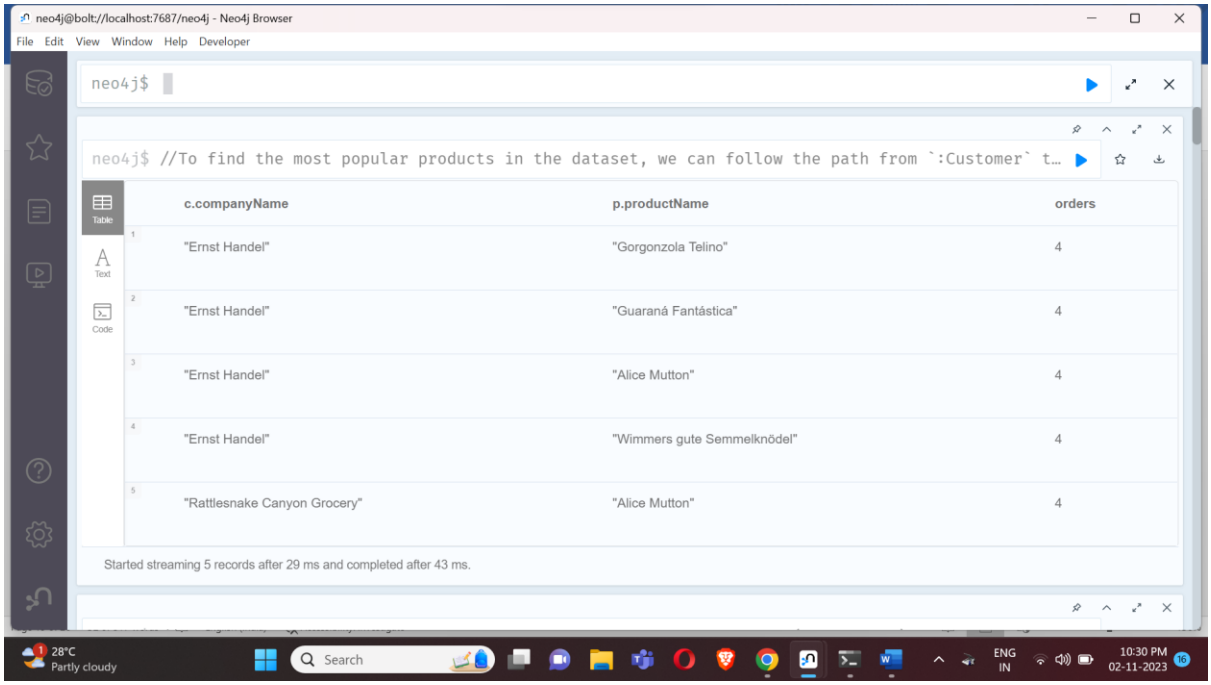
OPTIONAL MATCH (e)<-[:REPORTS_TO*0..]-(sub)-[:SOLD]->(order)

RETURN e.employeeID as employee, [x IN COLLECT(DISTINCT sub.employeeID)
WHERE x <> e.employeeID] AS reportsTo, COUNT(distinct order) AS totalOrders

ORDER BY totalOrders DESC;

//To find the most popular products in the dataset, we can follow the path from `:Customer` to `:Product`

MATCH (c:Customer)-[:PURCHASED]->(o:Order)-[:CONTAINS]->(p:Product)
RETURN c.companyName, p.productName, count(o) as orders
ORDER BY orders DESC
LIMIT 5

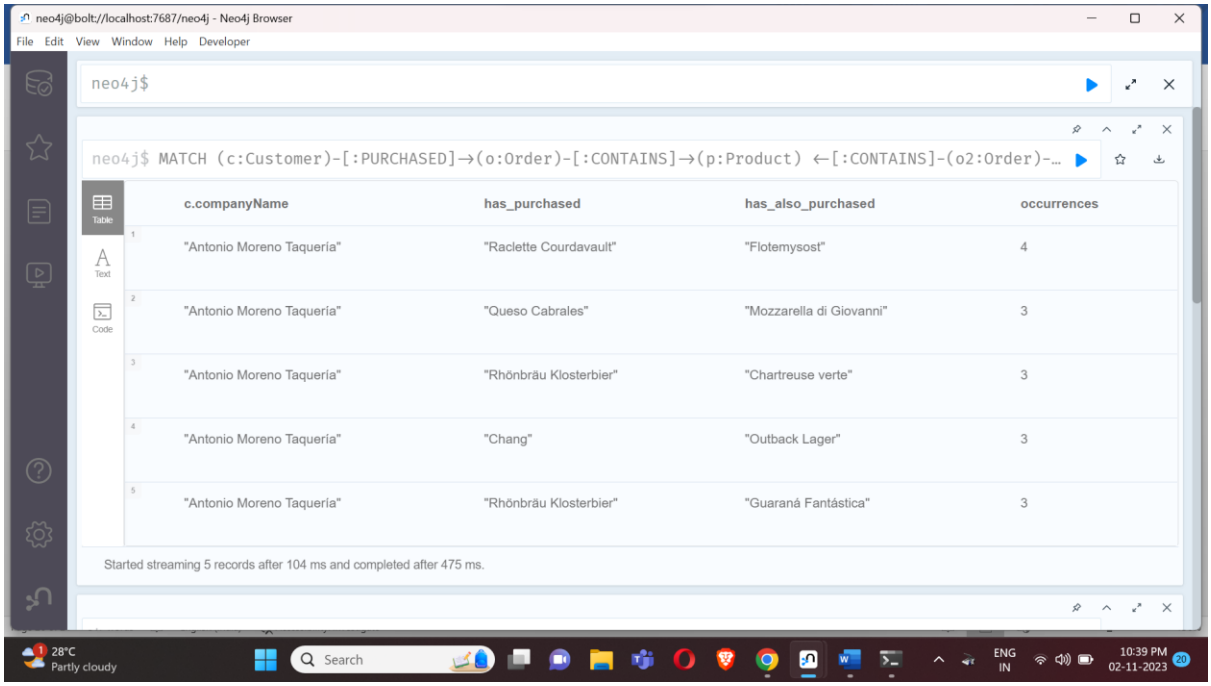


Content-Based Recommendations:

Content-based recommendations are based on products that a specific customer has purchased. The query finds products that the customer has bought and suggests other products within the same category that the customer hasn't purchased. This is a basic form of recommendation.

//Content Based Recommendations

```
MATCH (c:Customer)-[:PURCHASED]->(o:Order)-[:CONTAINS]->(p:Product)
<-[:CONTAINS]-(o2:Order)-[:CONTAINS]->(p2:Product)-[:PART_OF]->(:Category)<-
[:PART_OF]-(p)
WHERE c.customerID = 'ANTON' and NOT( (c)-[:PURCHASED]->(:Order)-
[:CONTAINS]->(p2) )
return c.companyName, p.productName as has_purchased, p2.productName as has_also_purc
hased, count(DISTINCT o2) as occurrences
order by occurrences desc
limit 5
```

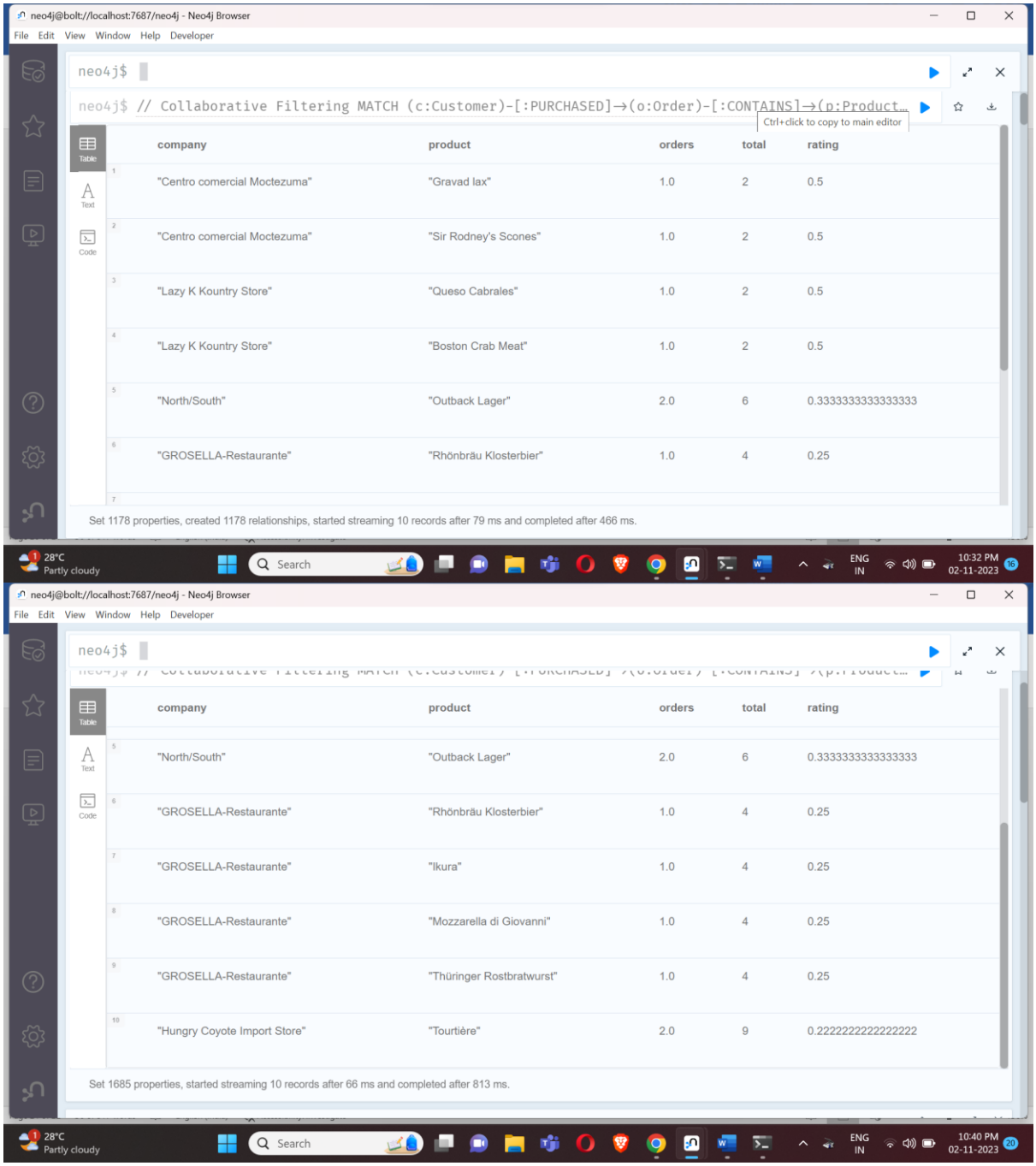


Collaborative Filtering

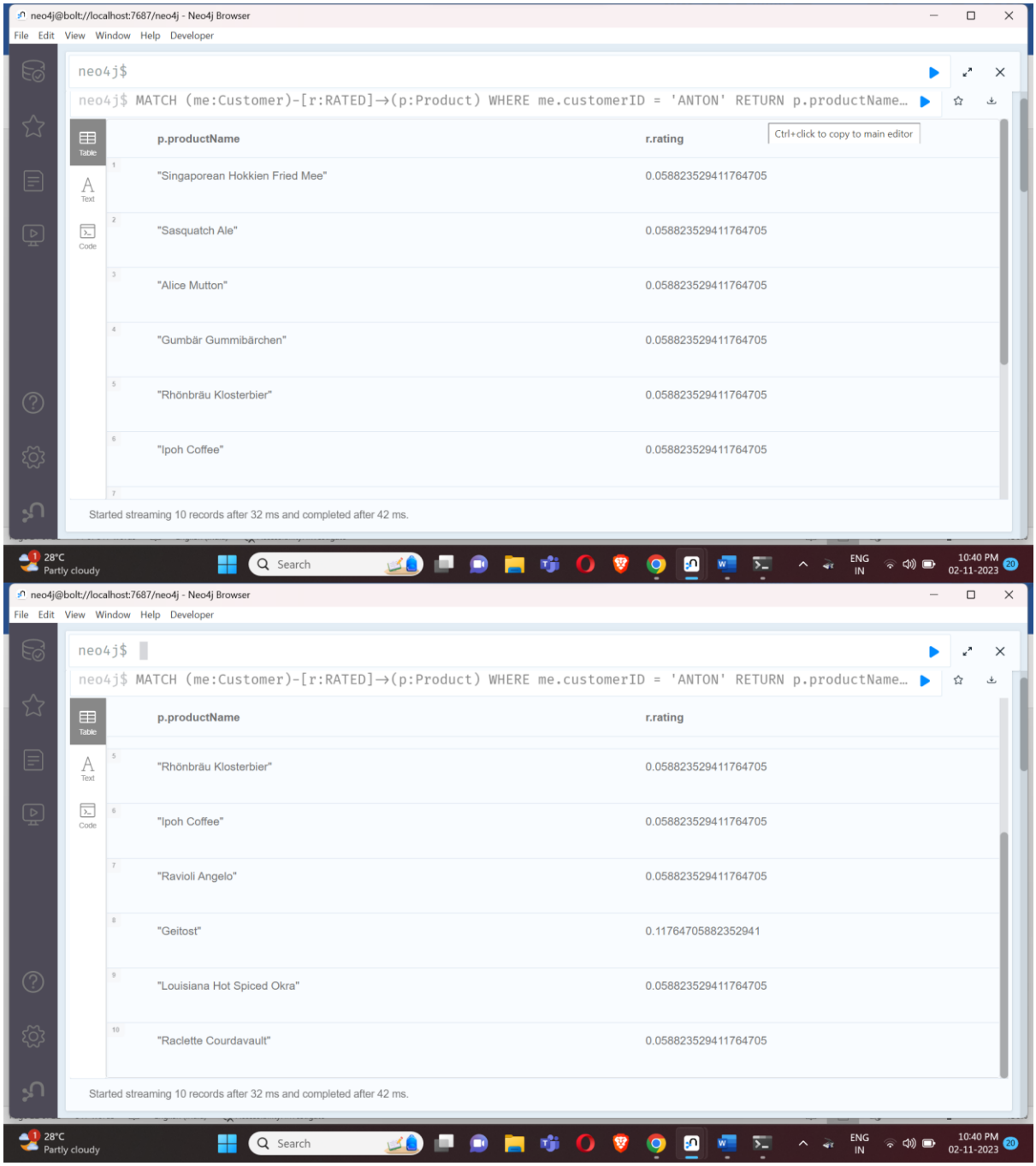
Collaborative filtering is a recommendation technique that identifies similarities between customers based on their product preferences. It calculates similarity scores between customers and products they have rated. These scores are used to make recommendations to customers based on the preferences of similar customers.

// Collaborative Filtering

```
MATCH (c:Customer)-[:PURCHASED]->(o:Order)-[:CONTAINS]->(p:Product)
WITH c, count(p) as total
MATCH (c)-[:PURCHASED]->(o:Order)-[:CONTAINS]->(p:Product)
WITH c, total,p, count(o)*1.0 as orders
MERGE (c)-[rated:RATED]->(p)
ON CREATE SET rated.rating = orders/total
ON MATCH SET rated.rating = orders/total
WITH c.companyName as company, p.productName as product, orders, total, rated.rating as rating
ORDER BY rating DESC
RETURN company, product, orders, total, rating LIMIT 10
```

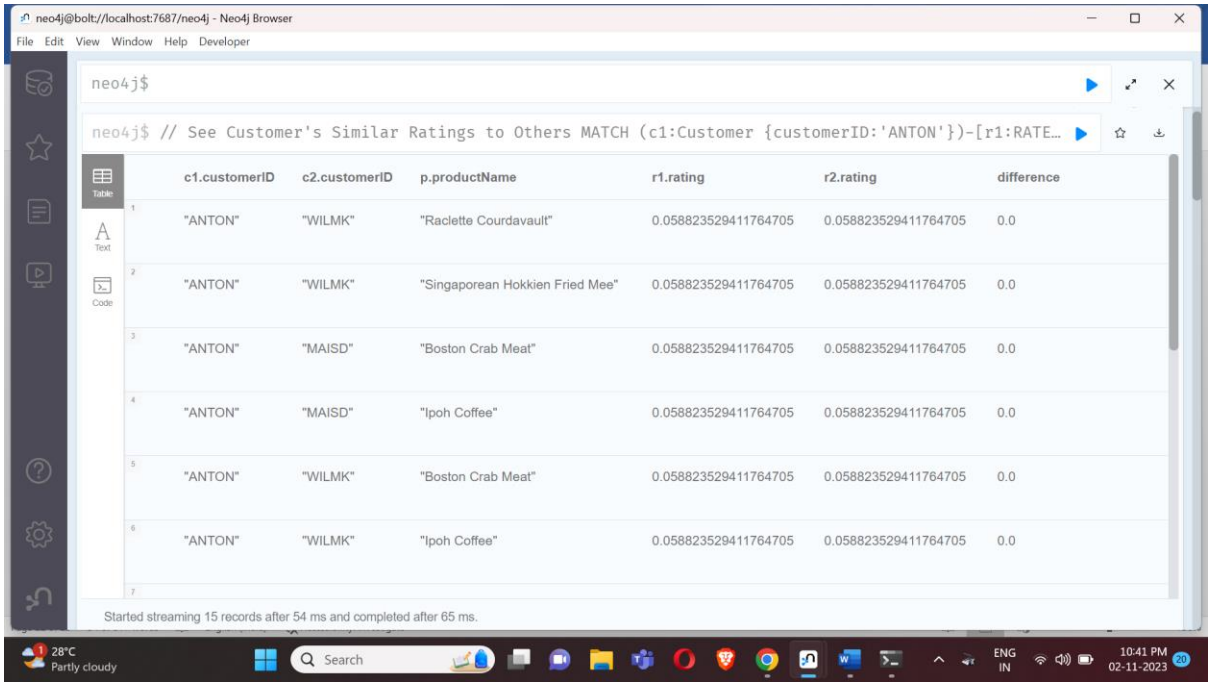


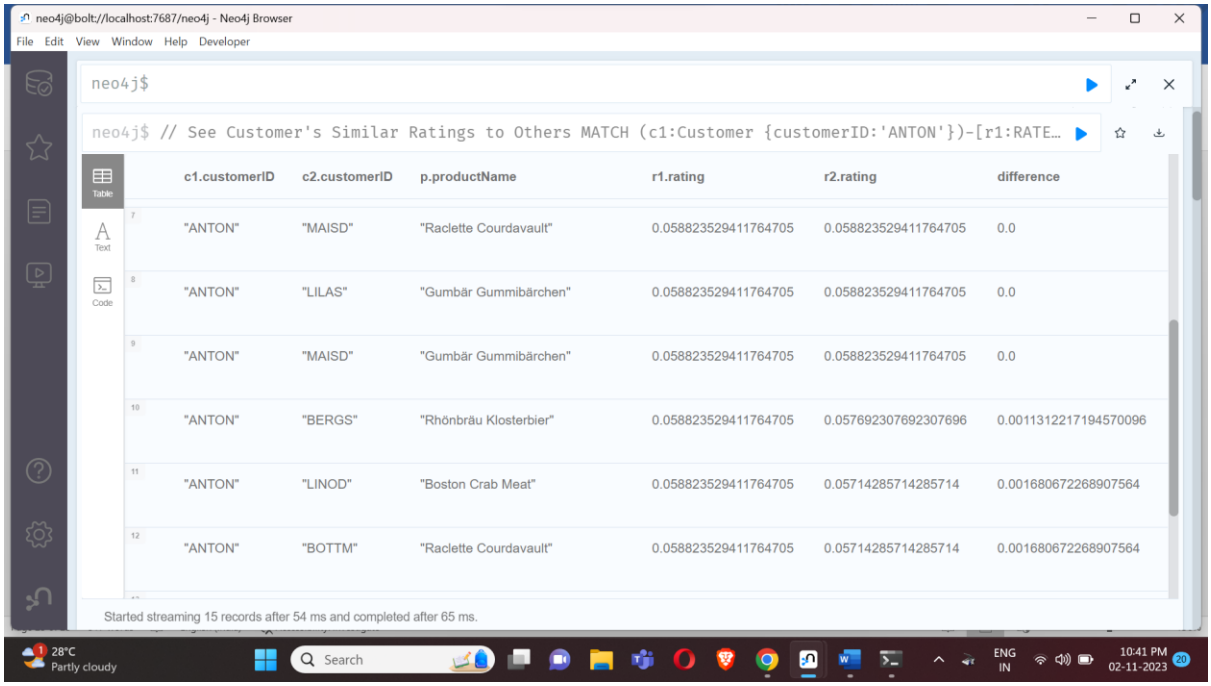
MATCH (me:Customer)-[r:RATED]->(p:Product)
WHERE me.customerID = 'ANTON'
RETURN p.productName, r.rating limit 10

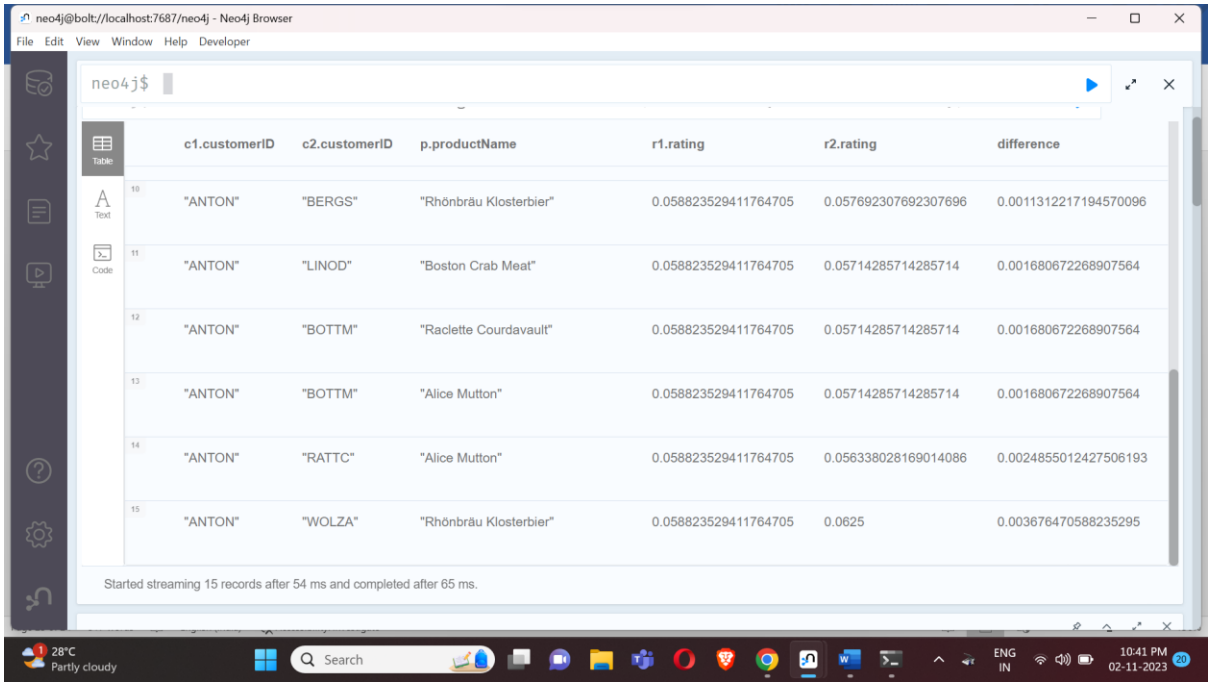


// See Customer's Similar Ratings to Others

```
MATCH (c1:Customer {customerID:'ANTON'})-[r1:RATED]->(p:Product)<-[r2:RATED]-(c2:Customer)
RETURN c1.customerID, c2.customerID, p.productName, r1.rating, r2.rating,
CASE WHEN r1.rating-r2.rating < 0 THEN -(r1.rating-r2.rating) ELSE r1.rating-r2.rating END as difference
ORDER BY difference ASC
LIMIT 15
```

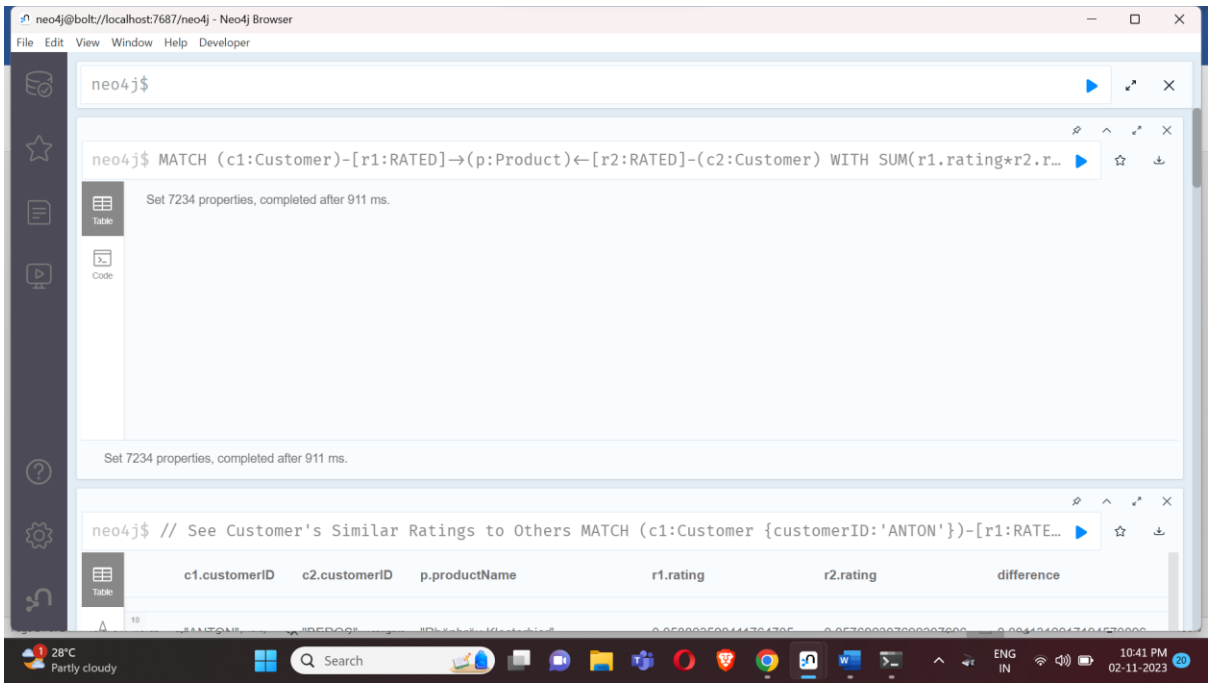

neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j$
neo4j$ // See Customer's Similar Ratings to Others MATCH (c1:Customer {customerID:'ANTON'})-[r1:RATE…
Table
Text
Code
c1.customerID c2.customerID p.productName r1.rating r2.rating difference
1 "ANTON" "WILMK" "Raclette Courdavault" 0.058823529411764705 0.058823529411764705 0.0
2 "ANTON" "WILMK" "Singaporean Hokkien Fried Mee" 0.058823529411764705 0.058823529411764705 0.0
3 "ANTON" "MAISD" "Boston Crab Meat" 0.058823529411764705 0.058823529411764705 0.0
4 "ANTON" "MAISD" "Ipoh Coffee" 0.058823529411764705 0.058823529411764705 0.0
5 "ANTON" "WILMK" "Boston Crab Meat" 0.058823529411764705 0.058823529411764705 0.0
6 "ANTON" "WILMK" "Ipoh Coffee" 0.058823529411764705 0.058823529411764705 0.0
7
Started streaming 15 records after 54 ms and completed after 65 ms.
28°C Partly cloudy
Search
ENG IN
10:41 PM 02-11-2023


neo4j@bolt://localhost:7687/neo4j - Neo4j Browser
File Edit View Window Help Developer
neo4j$
neo4j$ // See Customer's Similar Ratings to Others MATCH (c1:Customer {customerID:'ANTON'})-[r1:RATE…
Table
Text
Code
c1.customerID c2.customerID p.productName r1.rating r2.rating difference
7 "ANTON" "MAISD" "Raclette Courdavault" 0.058823529411764705 0.058823529411764705 0.0
8 "ANTON" "LILAS" "Gumbär Gummibärchen" 0.058823529411764705 0.058823529411764705 0.0
9 "ANTON" "MAISD" "Gumbär Gummibärchen" 0.058823529411764705 0.058823529411764705 0.0
10 "ANTON" "BERGS" "Rhönbräu Klosterbier" 0.058823529411764705 0.057692307692307696 0.0011312217194570096
11 "ANTON" "LINOD" "Boston Crab Meat" 0.058823529411764705 0.05714285714285714 0.001680672268907564
12 "ANTON" "BOTTM" "Raclette Courdavault" 0.058823529411764705 0.05714285714285714 0.001680672268907564
Started streaming 15 records after 54 ms and completed after 65 ms.
28°C Partly cloudy
Search
ENG IN
10:41 PM 02-11-2023



// create a similarity score between two Customers using Cosine Similarity

```
MATCH (c1:Customer)-[r1:RATED]->(p:Product)<-[r2:RATED]-(c2:Customer)
WITH
SUM(r1.rating*r2.rating) as dot_product,SQRT( REDUCE(x=0.0, a IN COLLECT(r1.rating)
 | x + a^2) ) as r1_length,
SQRT( REDUCE(y=0.0, b IN COLLECT(r2.rating) | y + b^2) ) as r2_length,
c1,c2
MERGE (c1)-[s:SIMILARITY]-(c2)
SET s.similarity = dot_product / (r1_length * r2_length)
```



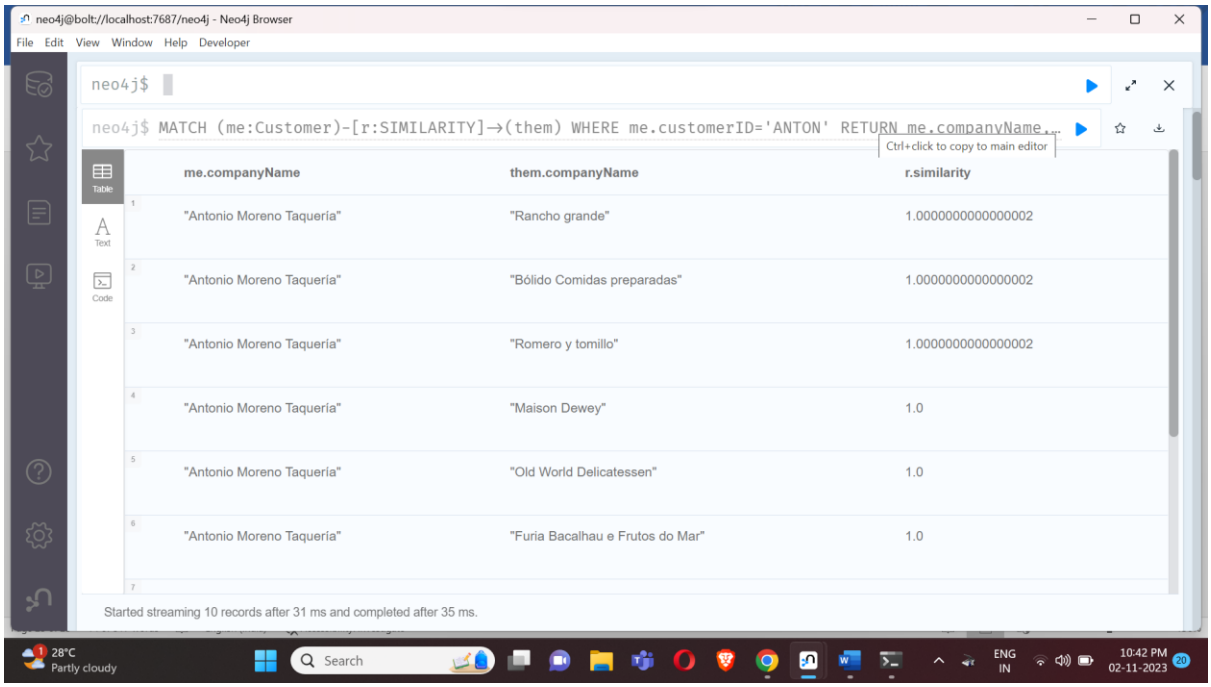
```
MATCH (me:Customer)-[r:SIMILARITY]->(them)
WHERE me.customerID='ANTON'
RETURN me.companyName, them.companyName, r.similarity
ORDER BY r.similarity DESC limit 10
```

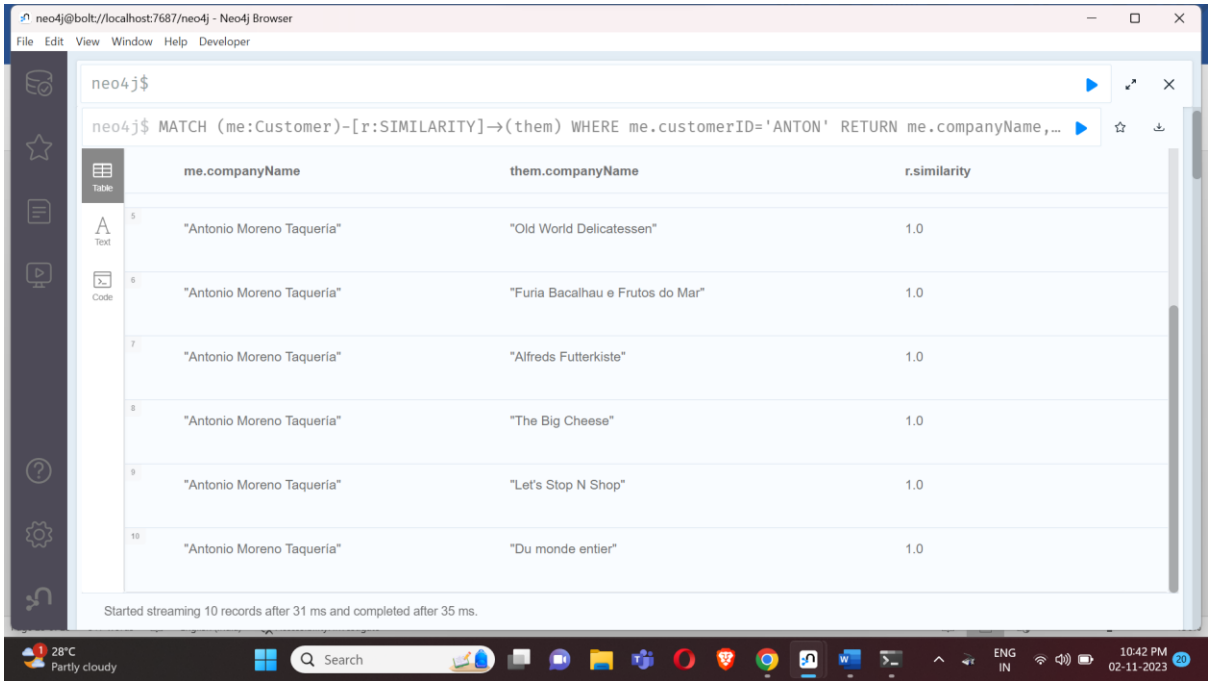
Recommendation based on Similarity Scores:

Building upon the collaborative filtering approach, this section generates recommendations for a specific customer. It identifies similar customers and suggests products that these similar customers have purchased but the specific customer has not. These recommendations are based on the calculated similarity scores.

// recommendation based on these similarity scores.

```
WITH 1 as neighbours
MATCH (me:Customer)-[:SIMILARITY]->(c:Customer)-[r:RATED]->(p:Product)
WHERE me.customerID = 'ANTON' and NOT ( (me)-[:RATED|PRODUCT|ORDER*1..2]-
>(p:Product) )
WITH p, COLLECT(r.rating)[0..neighbours] as ratings,
collect(c.companyName)[0..neighbours] as customers
WITH p, customers, REDUCE(s=0,i in ratings | s+i) / LENGTH(ratings)  as recommendation
ORDER BY recommendation DESC
RETURN p.productName, customers, recommendation LIMIT 10
```





Database Information

Use database

neo4j

Node labels

*(1,044) Category Customer Employee Order Product Supplier

Relationship types

*(9,279) CONTAINS PART_OF PURCHASED RATED REPORTS_TO SIMILARITY SOLD SUPPLIES

Property keys

categoryID categoryName companyName customerID customerName description employeeID fax firstName lastName orderID phone productID productName quantity rating shipName similarity supplierID title unitPrice