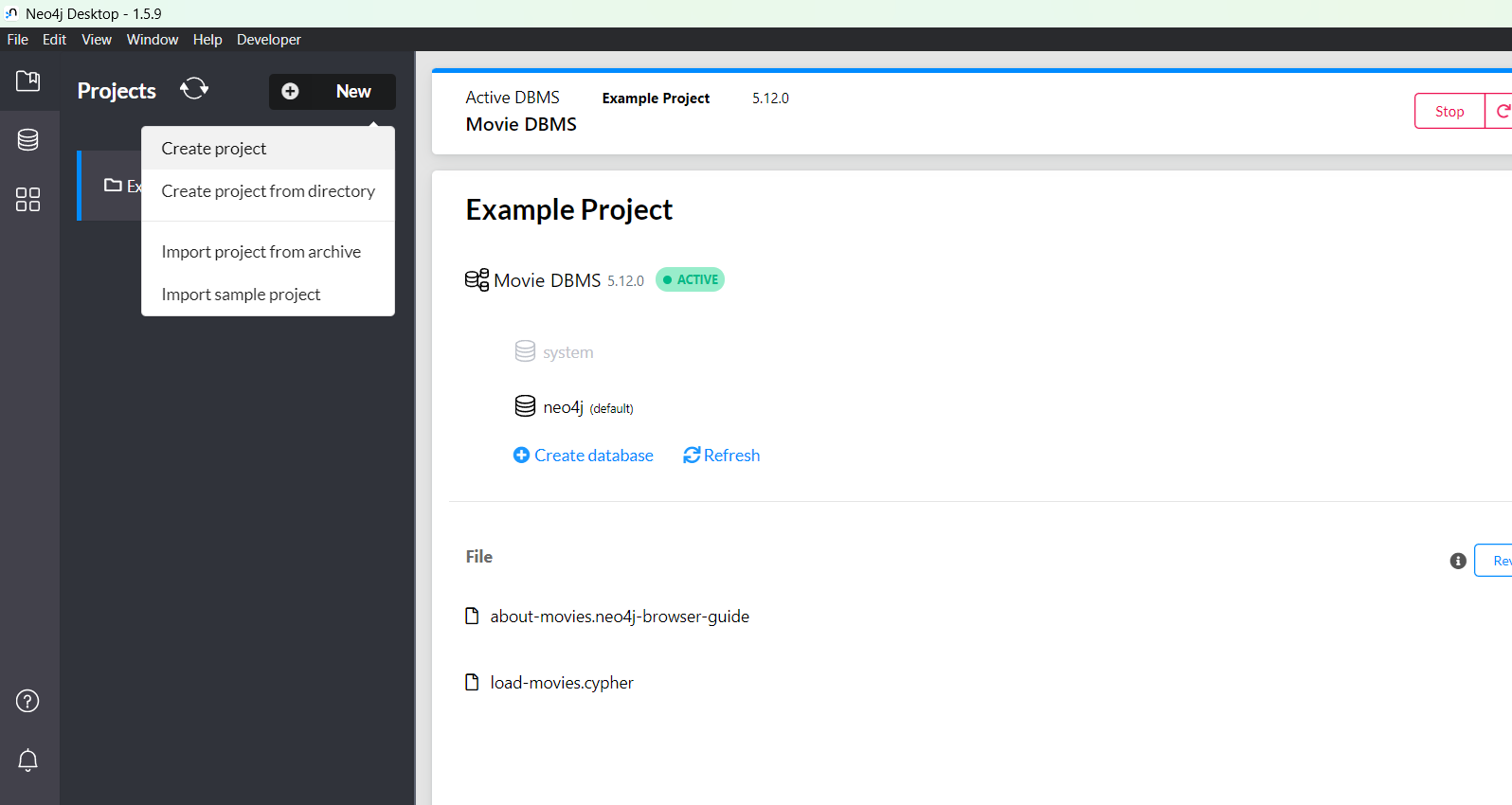
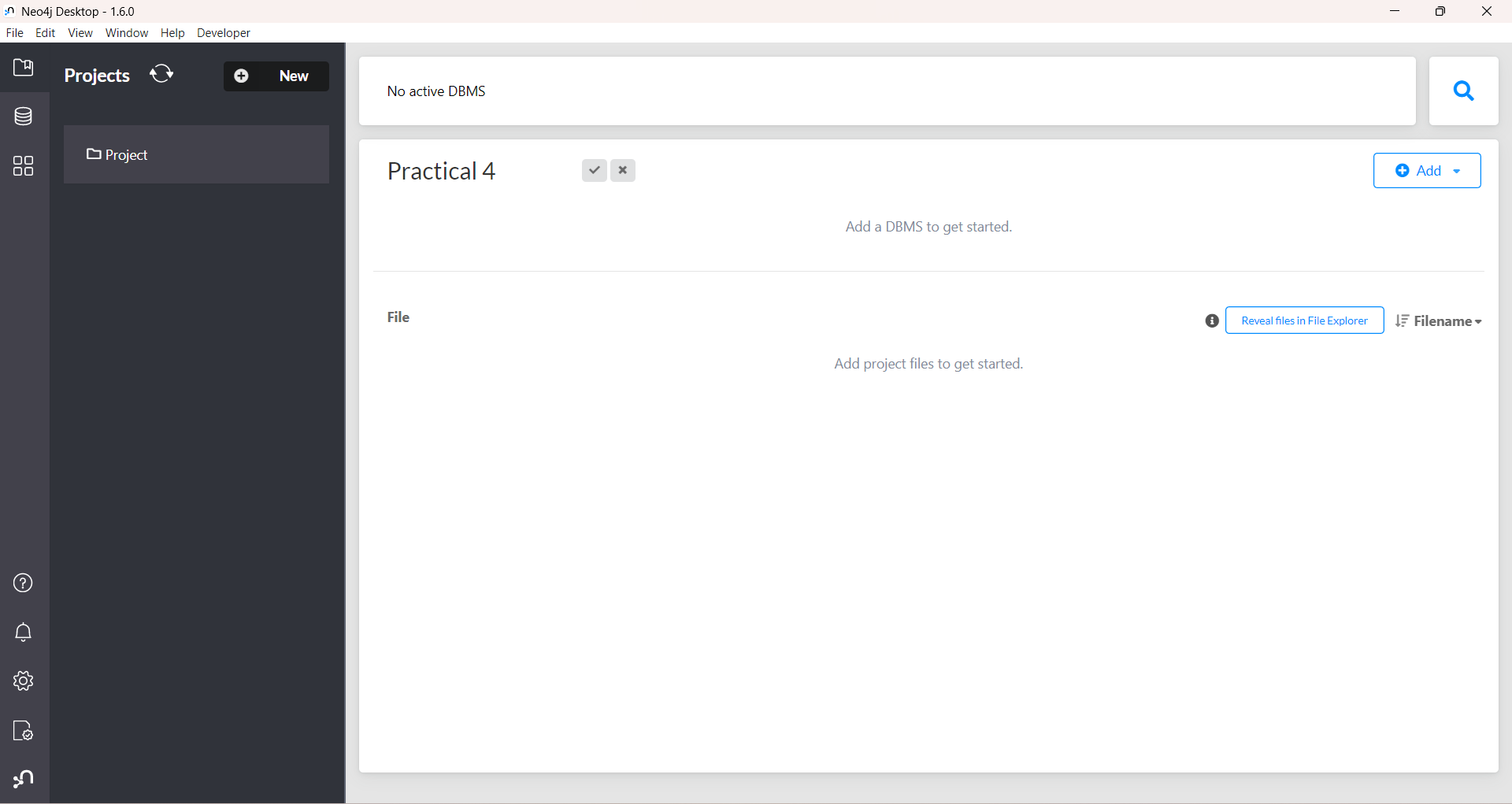
**PRACTICAL 4**

**EXPLORING FRAUD DETECTION WITH GRAPH DATA SCIENCE**

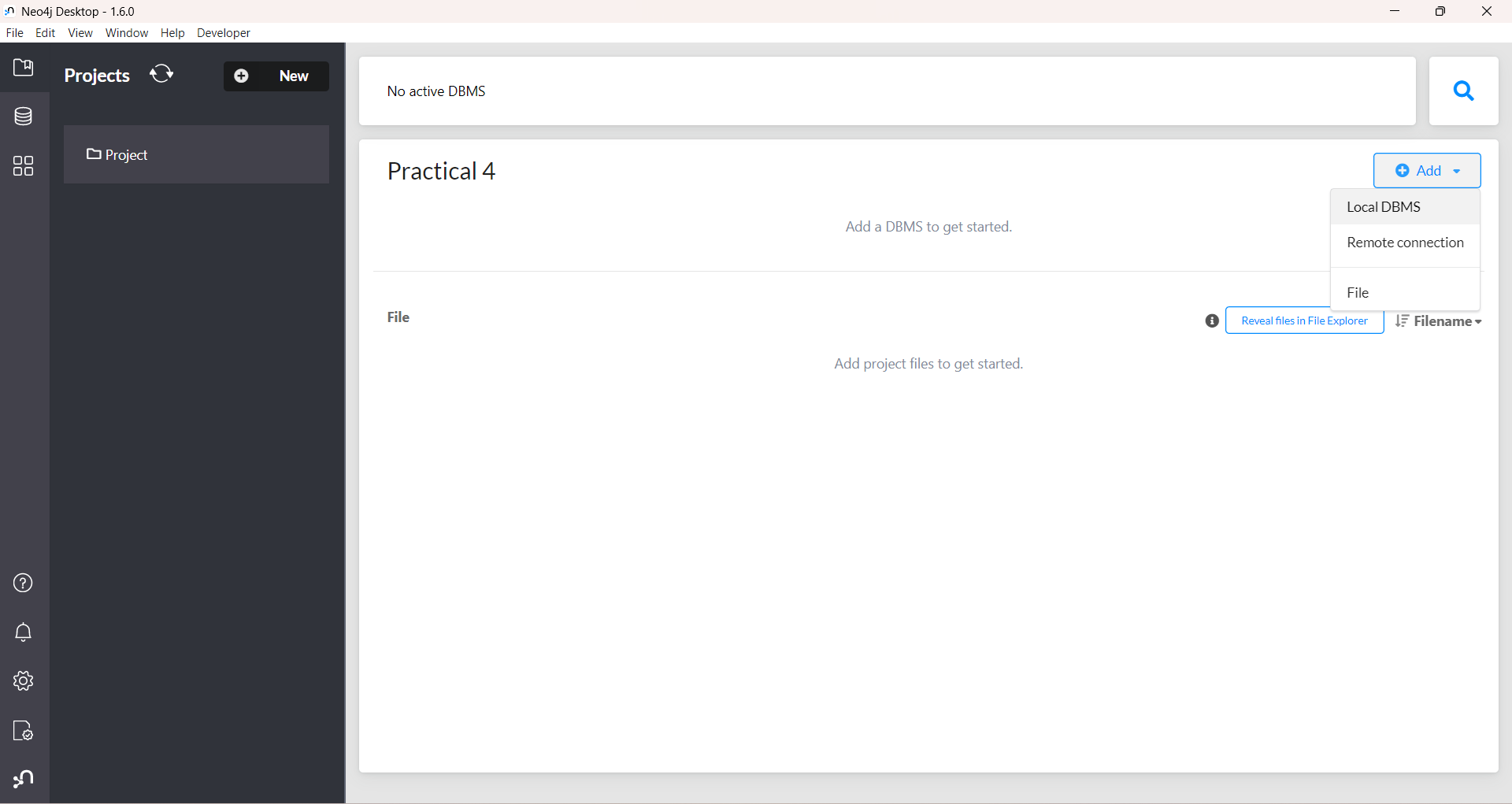
**Click on the New panel and create a new project**



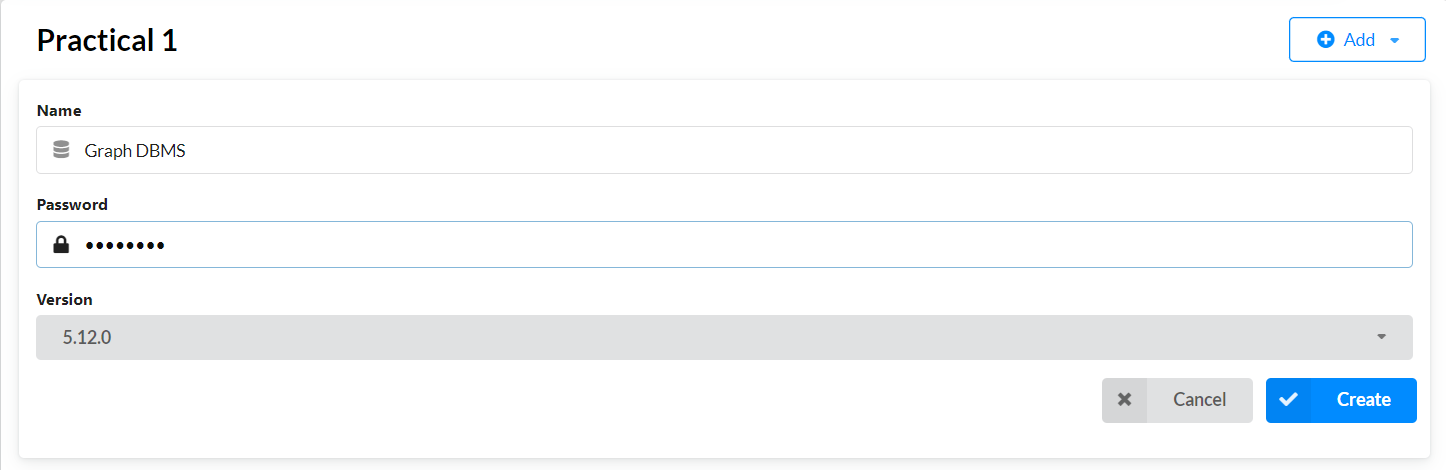
**Rename the project**



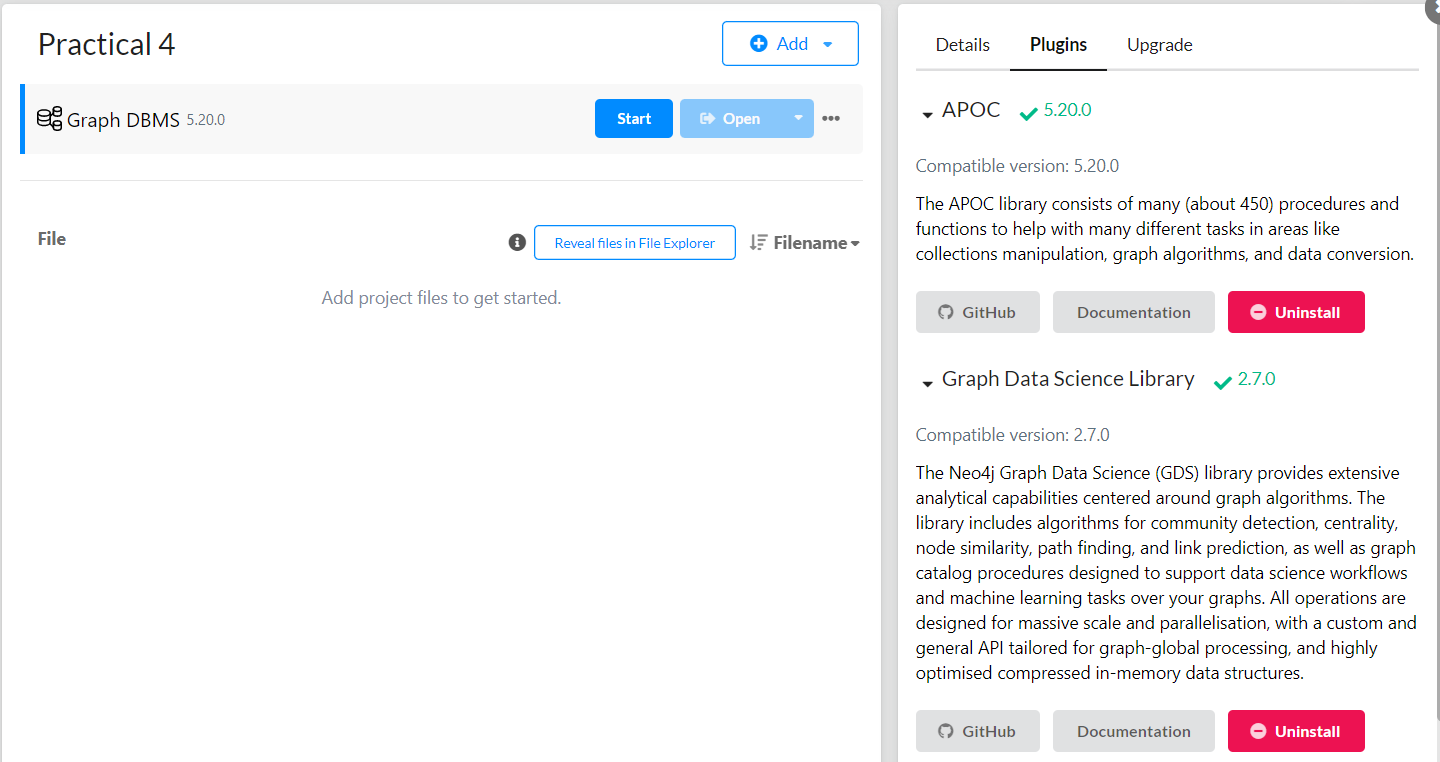
**In the right side click on add panel and select local dbms**



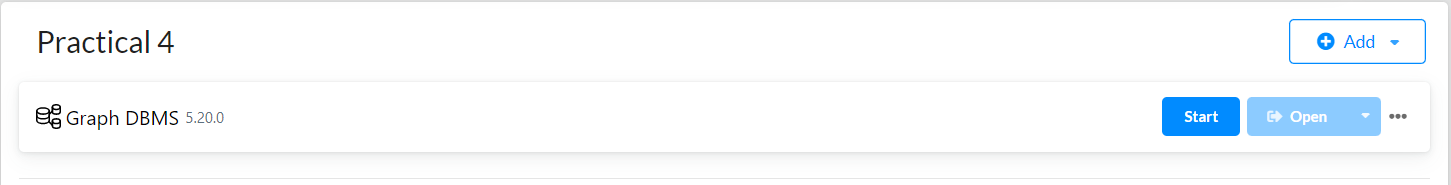
**Set a password with eight characters long & click on create and wait till the dbms gets created**



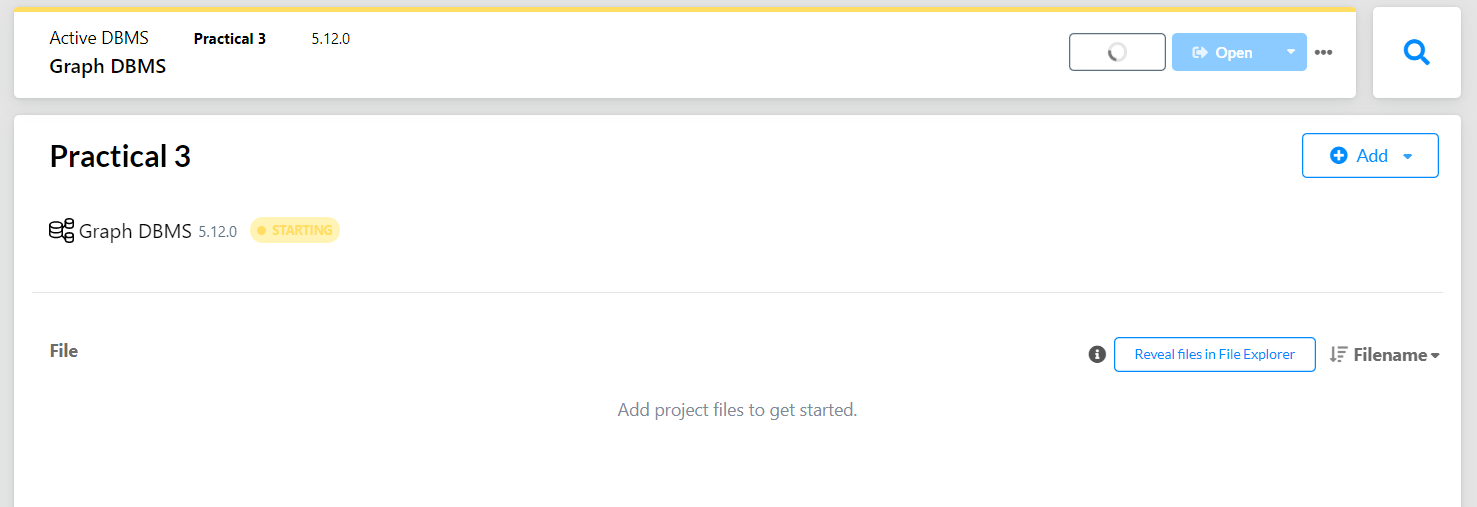
**Click on the empty space of the dbms and open the plugins tab and install “APOC” & “Graph Data Science Library”**



**Now click on start button to start the dbms**

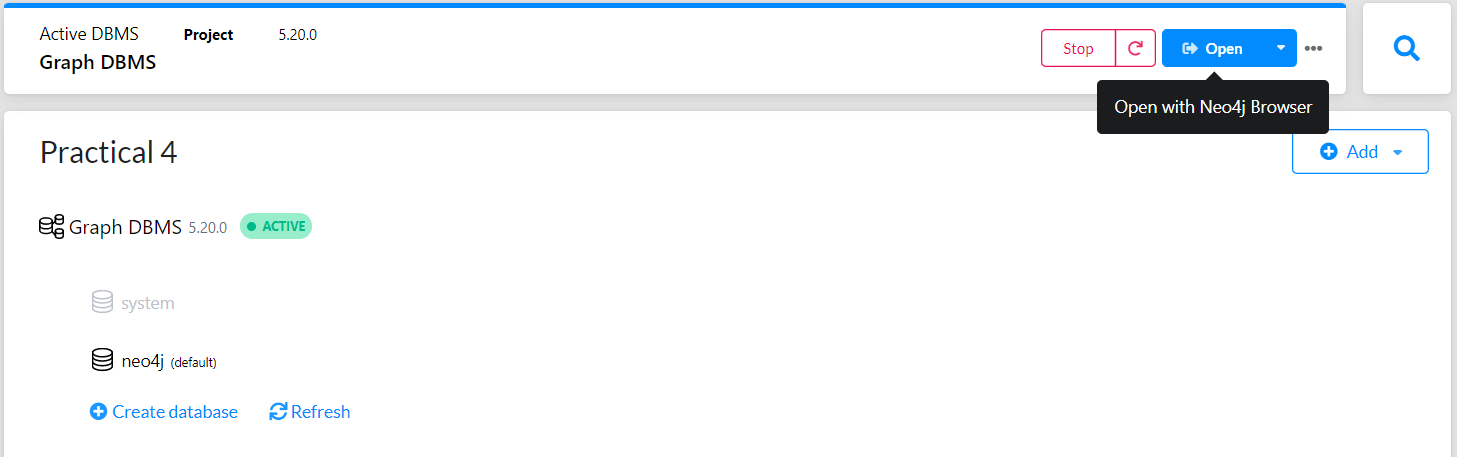


**A terminal will open do not close it otherwise dbms will get stopped**



**(Note: if while starting it shows** DBMS failed to start **then ignore it and click on start it will run properly)**

**If the dbms gets started Click on open to start the neo4j browser on the localhost**



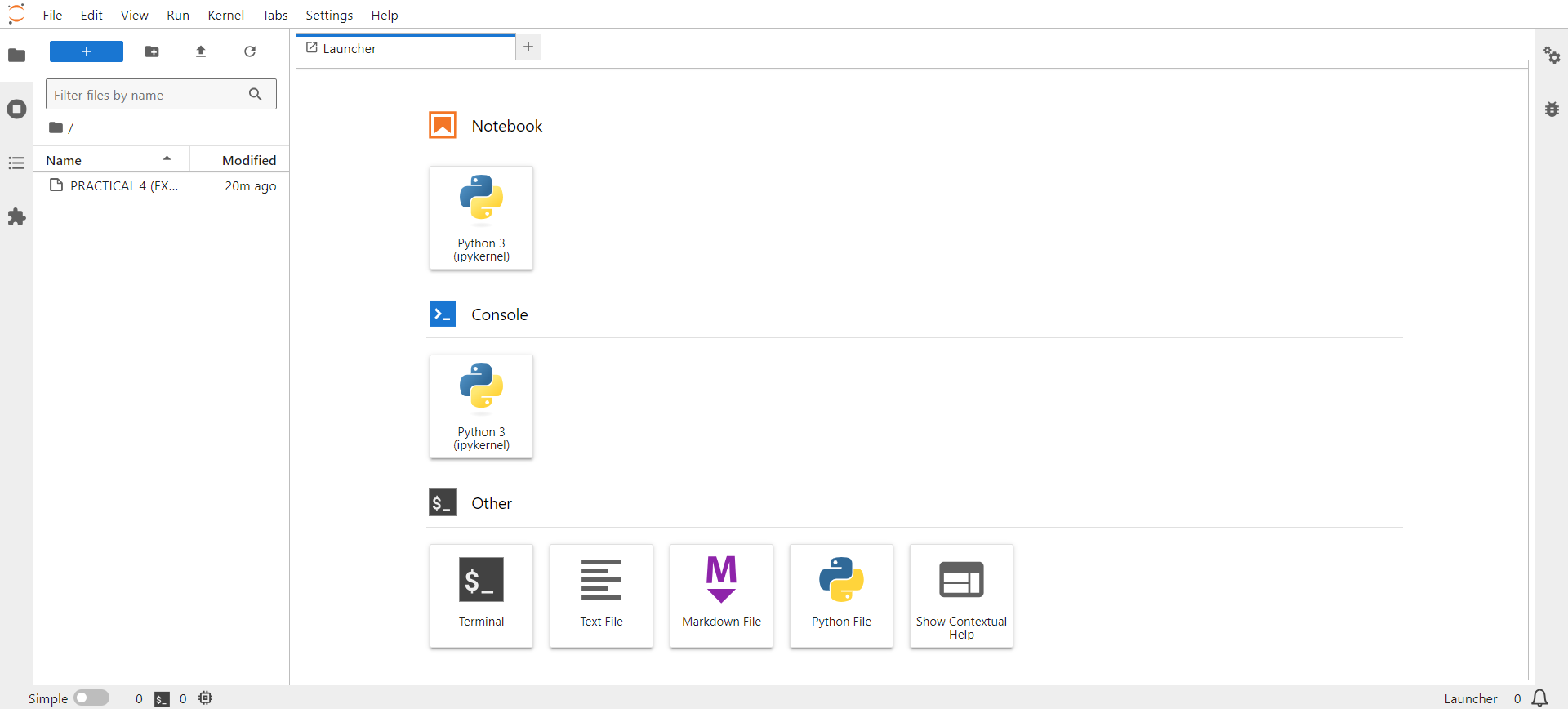
**Install JupyterLab via pip**

pip install jupyterlab

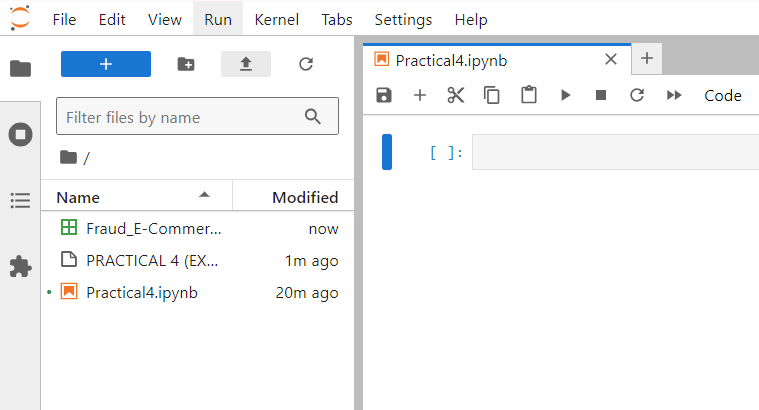
**Once installed, launch JupyterLab with:**

jupyter lab

**Click the first option in "Notebook (python3 (ipykernel))"**

**`**

**A new Notebook will open upload the dataset "Fraud\_E-Commerce\_Transactions" into the folders**

****

Install Dependencies

%%capture

%pip install graphdatascience neo4j pandas

Import Libraries

import pandas as pd

from graphdatascience import GraphDataScience

from neo4j import GraphDatabase

Neo4j Connection Setup

gds = GraphDataScience("bolt://localhost:7687", auth=("neo4j", "12345678"))

print(f"GDS version: {gds.version()}")

Helper Function to clear a graph by its name

def clear\_graph\_by\_name(g\_name):

    if gds.graph.exists(g\_name).exists:

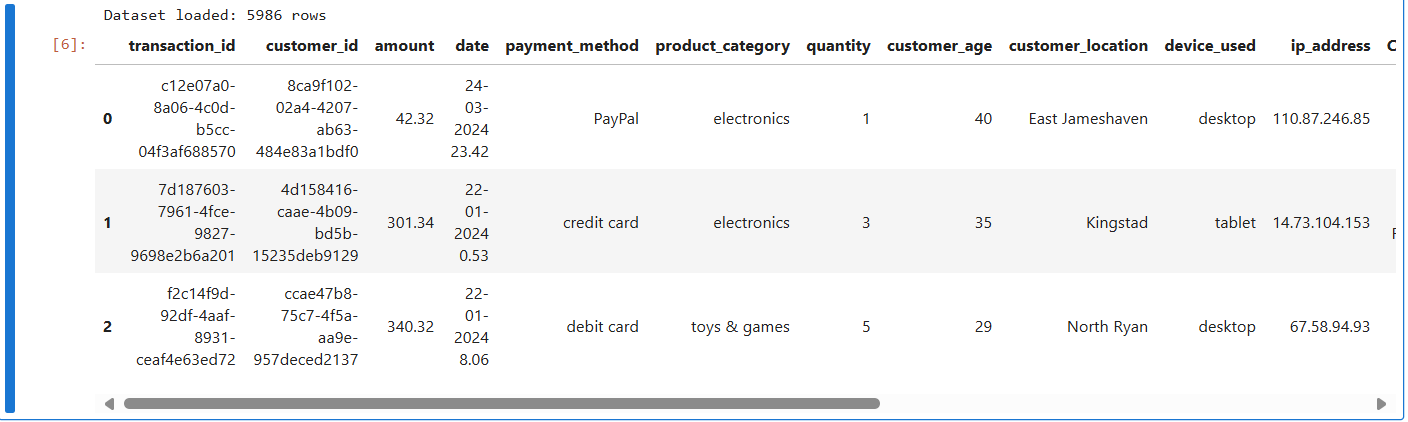
        gds.graph.drop(gds.graph.get(g\_name))

Dataset

df = pd.read\_csv('Fraud\_E-Commerce\_Transactions.csv')

print(f"Dataset loaded: {len(df)} rows")

df.head(3)

****

|  |
| --- |
| Create Driver to load data into Neo4j |

|  |
| --- |
| uri = "bolt://localhost:7687"  driver = GraphDatabase.driver(uri, auth=("neo4j", "12345678")) |

|  |
| --- |
| Create constraints and load data into neo4j |

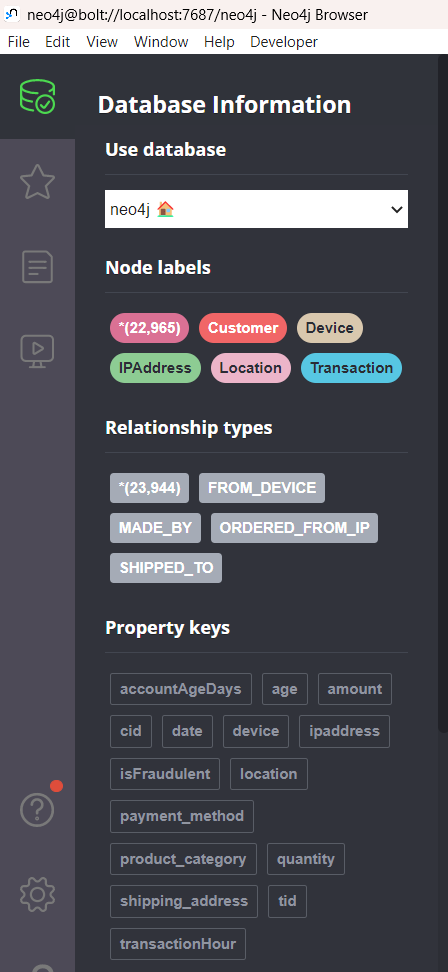
|  |
| --- |
| def create\_graph\_schema(tx):      constraints = [          "CREATE CONSTRAINT unique\_transaction IF NOT EXISTS FOR (t:Transaction) REQUIRE t.tid IS UNIQUE",          "CREATE CONSTRAINT unique\_customer IF NOT EXISTS FOR (c:Customer) REQUIRE c.cid IS UNIQUE",          "CREATE CONSTRAINT unique\_device IF NOT EXISTS FOR (d:Device) REQUIRE d.device IS UNIQUE",          "CREATE CONSTRAINT unique\_ip IF NOT EXISTS FOR (ip:IPAddress) REQUIRE ip.ipaddress IS UNIQUE",          "CREATE CONSTRAINT unique\_location IF NOT EXISTS FOR (l:Location) REQUIRE l.location IS UNIQUE"      ]      for constraint in constraints:          tx.run(constraint) |

|  |
| --- |
| def ingest\_data(tx, df):      for \_, row in df.iterrows():          tx.run("""                 MERGE (t:Transaction {tid: $transaction\_id})                 SET t.amount = $amount, t.date = $date, t.payment\_method = $payment\_method,                     t.product\_category = $product\_category, t.quantity = $quantity,                     t.isFraudulent = $is\_fraudulent, t.transactionHour = $transaction\_hour, t.device = $device\_used                 MERGE (c:Customer {cid: $customer\_id})                 SET c.age = $customer\_age, c.accountAgeDays = $account\_age\_days,                     c.ipaddress = $ip\_address, c.location = $customer\_location,                     c.device = $device\_used, c.shipping\_address = $Customer\_Shipping\_Address, c.isFraudulent = $is\_fraudulent                 MERGE (d:Device {device: $device\_used})                 MERGE (ip:IPAddress {ipaddress: $ip\_address})                 MERGE (l:Location {location: $customer\_location})                 MERGE (t)-[:MADE\_BY]->(c)                 MERGE (t)-[:FROM\_DEVICE]->(d)                 MERGE (t)-[:SHIPPED\_TO]->(l)                 MERGE (t)-[:ORDERED\_FROM\_IP]->(ip)                 """,                 dict(row)              ) |

|  |
| --- |
| with driver.session() as session:      session.execute\_write(create\_graph\_schema)      session.execute\_write(ingest\_data, df)  print("Data ingestion completed") |

|  |
| --- |
|  |

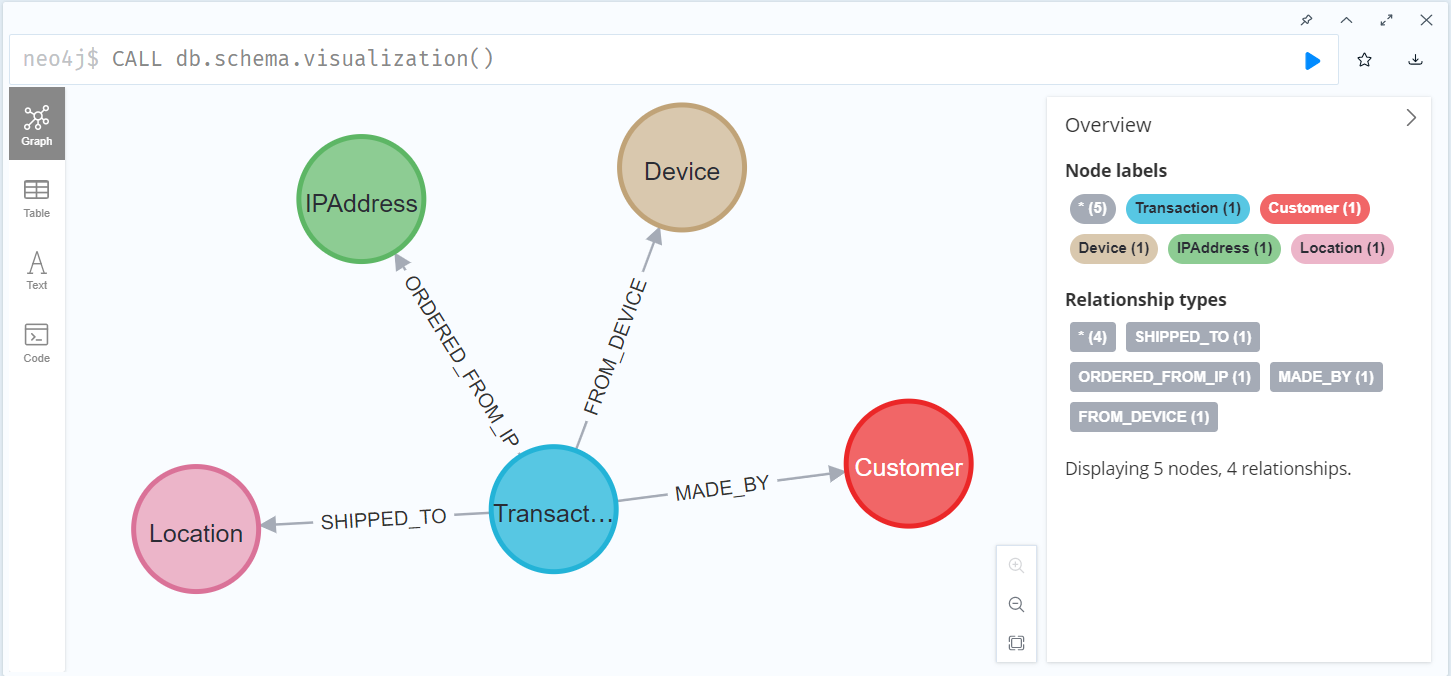
**Check into neo4j browser whether constraints, relationships & its properties got properly inserted**

****

**[A] BASIC VISUALIZATION**

**Visualize the graph schema in Neo4j Browser**

CALL db.schema.visualization()

****

**Node Counts by Labels**

**(You can run these type of codes in neo4j browser by only using the cypher query inside ''' ''' )**

print("Node Counts by Labels")

gds.run\_cypher('''

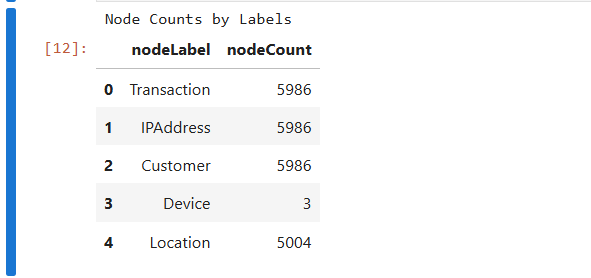
    CALL apoc.meta.stats()

    YIELD labels

    UNWIND keys(labels) AS nodeLabel

    RETURN nodeLabel, labels[nodeLabel] AS nodeCount

''')

****

**Relationship Counts by Type**

print("Relationship Counts by Type")

gds.run\_cypher('''

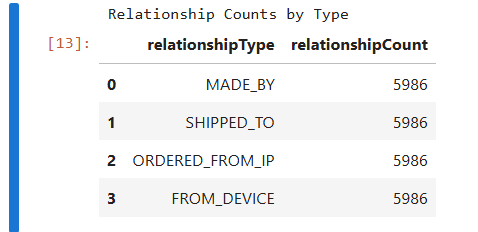
    CALL apoc.meta.stats()

    YIELD relTypesCount

    UNWIND keys(relTypesCount) AS relationshipType

    RETURN relationshipType, relTypesCount[relationshipType] AS relationshipCount

''')

****

**Fraudulent Transactions**

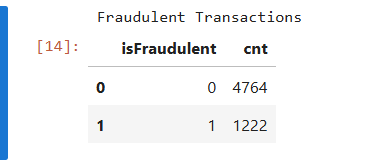
print("Fraudulent Transactions")

gds.run\_cypher('''

    MATCH (t:Transaction)

    RETURN t.isFraudulent AS isFraudulent, count(t) AS cnt

''')

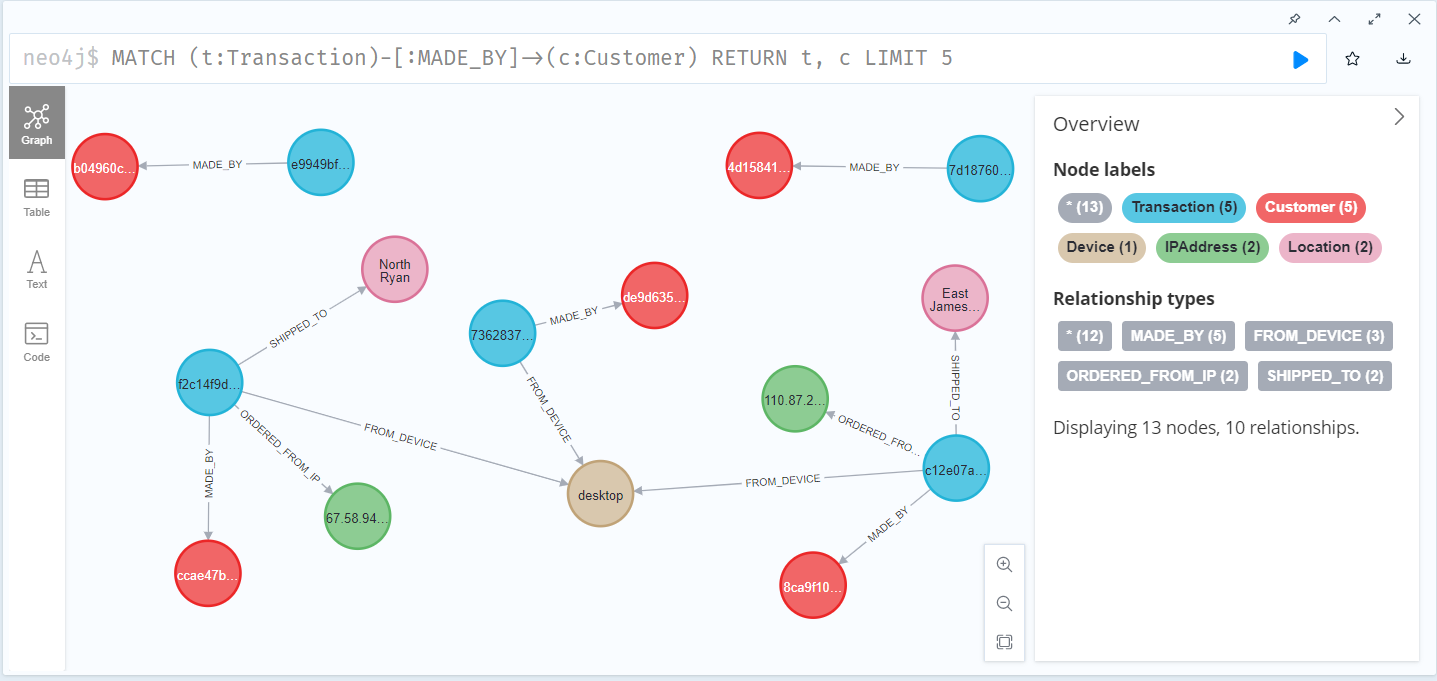
****

**Display all transactions and their associated customers in neo4j browser**

MATCH (t:Transaction)-[:MADE\_BY]->(c:Customer)

RETURN t, c LIMIT 5

**(I have modified the visualization by expanding the specific transaction nodes to know its connections)**

****

**Distribution of Transaction Amounts**

print("Show distribution of transaction amounts")

gds.run\_cypher('''

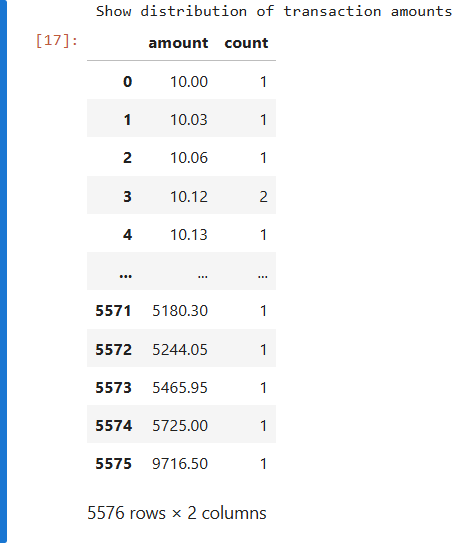
        MATCH (t:Transaction)

        WITH t.amount AS amount, count(\*) AS count

        RETURN amount, count

        ORDER BY amount

''')

****

**Customer Age Distribution**

print("Customer age distribution")

gds.run\_cypher('''

    MATCH (c:Customer)

    WITH CASE

        WHEN c.age >= 0 AND c.age <= 17 THEN '0-17'

        WHEN c.age >= 18 AND c.age <= 24 THEN '18-24'

        WHEN c.age >= 25 AND c.age <= 34 THEN '25-34'

        WHEN c.age >= 35 AND c.age <= 44 THEN '35-44'

        WHEN c.age >= 45 AND c.age <= 54 THEN '45-54'

        WHEN c.age >= 55 AND c.age <= 64 THEN '55-64'

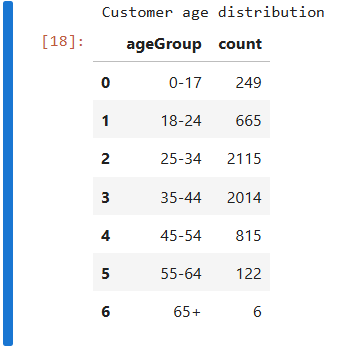
        ELSE '65+' END AS ageGroup,

        count(\*) AS count

    RETURN ageGroup, count

    ORDER BY ageGroup

''')

****

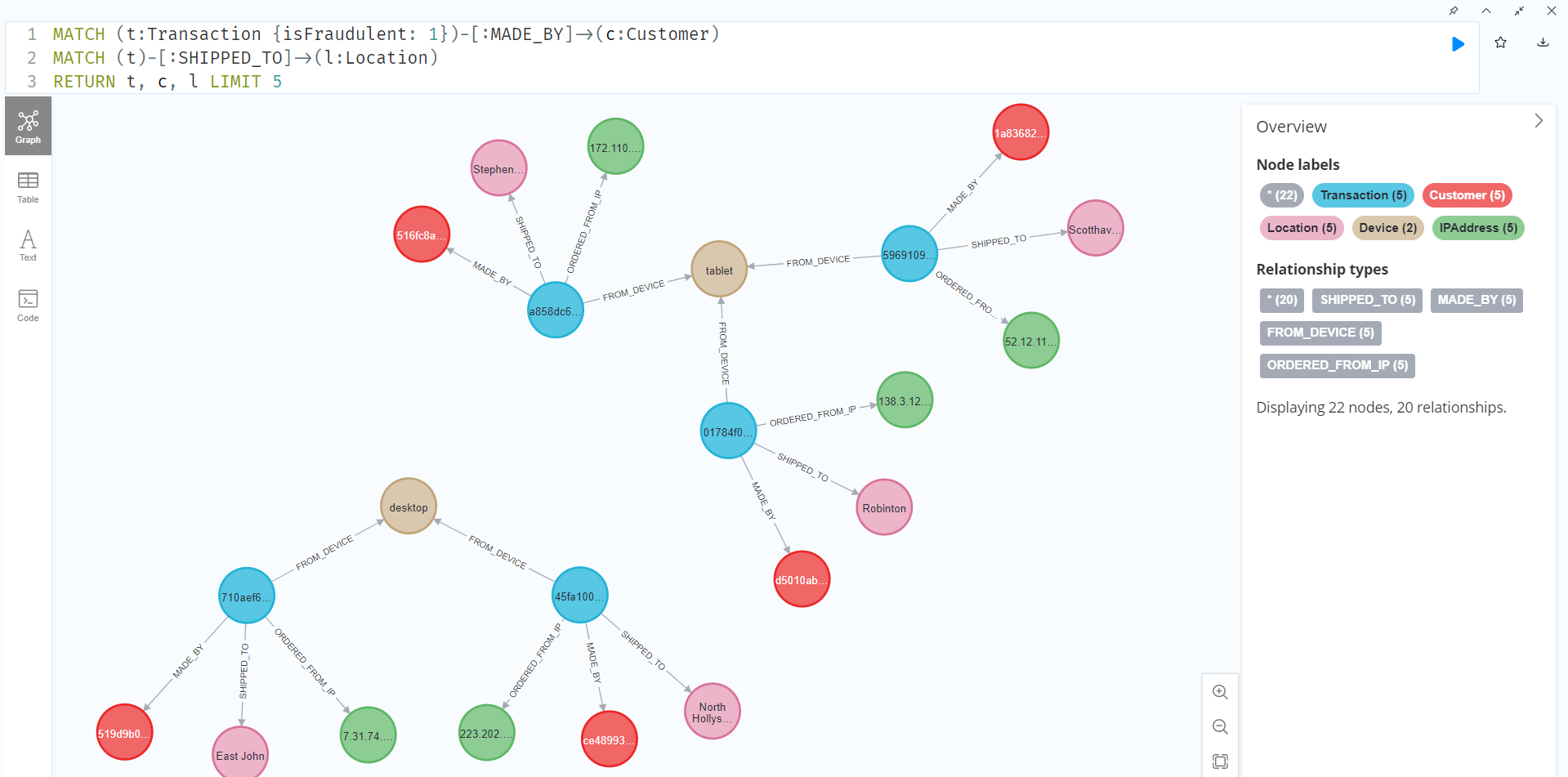
**Show all fraudulent transactions and associated data in neo4j browser**

MATCH (t:Transaction {isFraudulent: 1})-[:MADE\_BY]->(c:Customer)

MATCH (t)-[:SHIPPED\_TO]->(l:Location)

RETURN t, c, l LIMIT 5

**(I have modified the visualization by expanding the specific transaction nodes to know its connections)**

****

**Top 5 Product Categories with Highest Fraud Rate**

print("Display top 5 product categories with the highest fraud rate")

gds.run\_cypher('''

        MATCH (t:Transaction)

        WITH t.product\_category AS category,

            sum(CASE WHEN t.isFraudulent = 1 THEN 1 ELSE 0 END) AS fraudCount,

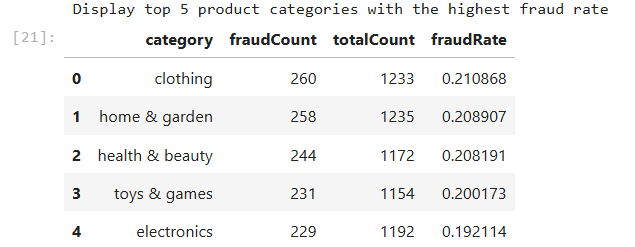
            count(\*) AS totalCount

        RETURN category, fraudCount, totalCount, (toFloat(fraudCount) / totalCount) AS fraudRate

        ORDER BY fraudRate DESC

        LIMIT 5

''')

****

**Fraud Patterns by Payment Method**

print("Analyzing fraud patterns by payment method...")

gds.run\_cypher('''

MATCH (t:Transaction)-[:MADE\_BY]->(c:Customer)

WITH t.payment\_method AS paymentMethod,

     count(\*) AS totalCount,

     sum(CASE WHEN t.isFraudulent = 1 THEN 1 ELSE 0 END) AS fraudCount

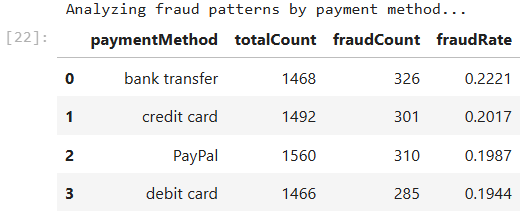
WITH paymentMethod, totalCount, fraudCount, toFloat(fraudCount) / totalCount AS fraudRate

RETURN paymentMethod, totalCount, fraudCount, round(fraudRate, 4) AS fraudRate

ORDER BY fraudRate DESC

LIMIT 5

''')

****

print("Transaction Volume by Payment Method")

gds.run\_cypher('''

MATCH (t:Transaction)

WITH t.payment\_method AS method,

     sum(t.amount) AS totalAmount,

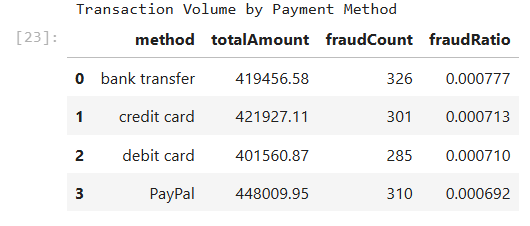
     sum(CASE WHEN t.isFraudulent = 1 THEN 1 ELSE 0 END) AS fraudCount

RETURN method, totalAmount, fraudCount,

       toFloat(fraudCount) / toFloat(totalAmount) AS fraudRatio

ORDER BY fraudRatio DESC

''')

****

**Devices Used in Fraudulent Transactions**

print("Analyzing devices used in fraudulent transactions:")

gds.run\_cypher('''

    MATCH (t:Transaction)-[:FROM\_DEVICE]->(d:Device)

    WITH d.device AS deviceType,

         sum(CASE WHEN t.isFraudulent = 1 THEN 1 ELSE 0 END) AS fraudCount,

         count(\*) AS totalCount

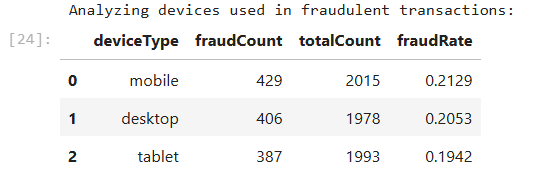
    WITH deviceType, fraudCount, totalCount,

         toFloat(fraudCount) / totalCount AS fraudRate

    RETURN deviceType, fraudCount, totalCount, round(fraudRate, 4) AS fraudRate

    ORDER BY fraudRate DESC

''')

****

**High-Value Transactions**

print("Display customers with high-value transactions")

gds.run\_cypher('''

        MATCH (t:Transaction)-[:MADE\_BY]->(c:Customer)

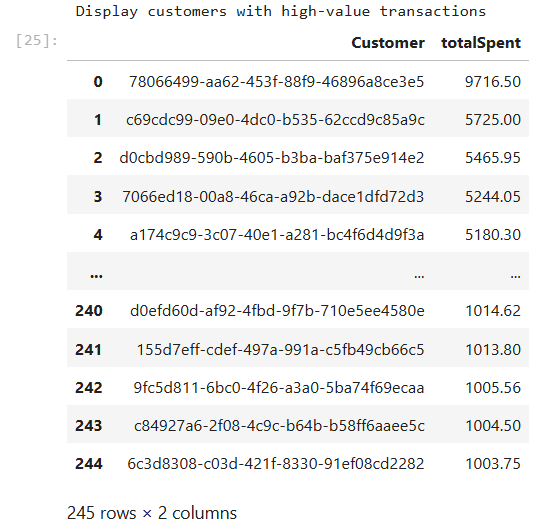
        WITH c, sum(t.amount) AS totalSpent

        WHERE totalSpent > 1000

        RETURN c.cid AS Customer, totalSpent

        ORDER BY totalSpent DESC

''')

****

**Transaction Patterns by Time of Day**

print("Transaction patterns by time of day in hrs")

gds.run\_cypher('''

    MATCH (t:Transaction)

    WITH CASE

        WHEN t.transactionHour >= 0 AND t.transactionHour < 6 THEN '00-05'

        WHEN t.transactionHour >= 6 AND t.transactionHour < 12 THEN '06-11'

        WHEN t.transactionHour >= 12 AND t.transactionHour < 18 THEN '12-17'

        ELSE '18-23' END AS timeSlot,

        count(\*) AS transactionCount,

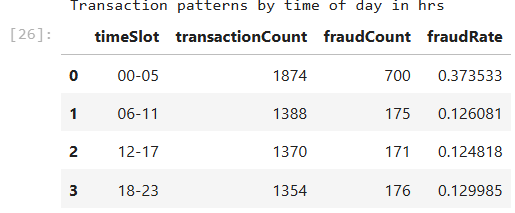
        sum(CASE WHEN t.isFraudulent = 1 THEN 1 ELSE 0 END) AS fraudCount

    RETURN timeSlot, transactionCount, fraudCount,

           (toFloat(fraudCount) / transactionCount) AS fraudRate

    ORDER BY timeSlot

''')

****

**Customers with Recently Created Accounts**

print("Display customers with recently created accounts (potential risk factor)")

gds.run\_cypher('''

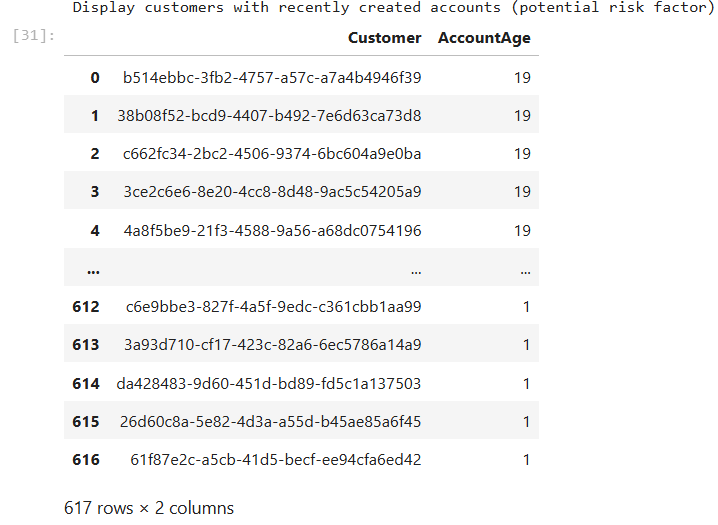
        MATCH (c:Customer)

        WHERE c.accountAgeDays < 20

        RETURN c.cid as Customer, c.accountAgeDays as AccountAge

        ORDER BY c.accountAgeDays DESC

''')

****

**Transaction Hotspots by Location**

print("Show transaction hotspots by location")

gds.run\_cypher('''

        MATCH (t:Transaction)-[:SHIPPED\_TO]->(l:Location)

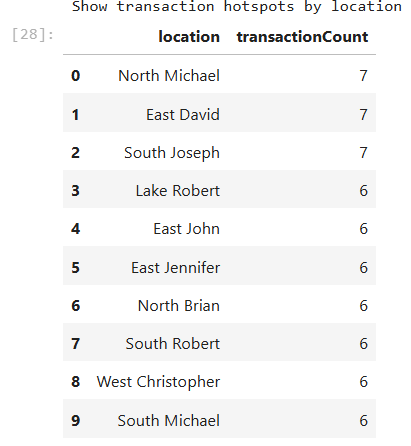
        WITH l.location AS location, count(\*) AS transactionCount

        RETURN location, transactionCount

        ORDER BY transactionCount DESC

        LIMIT 10

''')

****

**Fraudulent Transaction Distribution by Location**

print("Show fraudulent transaction distribution by location")

gds.run\_cypher('''

        MATCH (t:Transaction {isFraudulent: 1})-[:SHIPPED\_TO]->(l:Location)

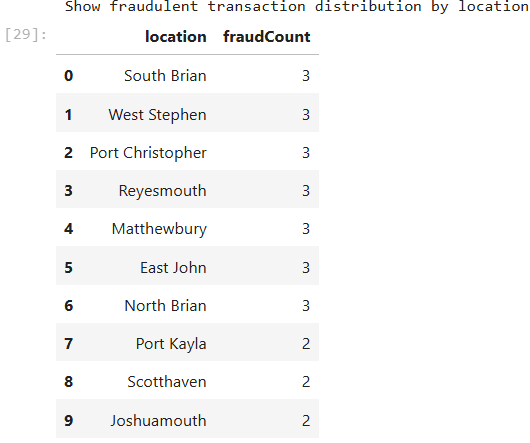
        WITH l.location AS location, count(\*) AS fraudCount

        RETURN location, fraudCount

        ORDER BY fraudCount DESC

        LIMIT 10

''')

****

print("Heat Map of Transaction Amounts by Location")

gds.run\_cypher('''

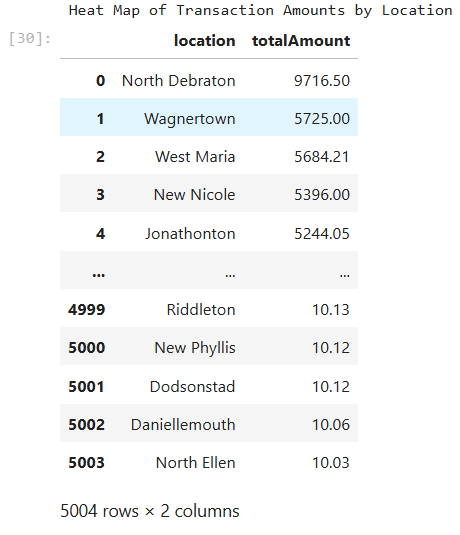
      MATCH (t:Transaction)-[:SHIPPED\_TO]->(l:Location)

      WITH l.location AS location, sum(t.amount) AS totalAmount

      RETURN location, totalAmount

      ORDER BY totalAmount DESC

''')

****

**The Louvain algorithm is a popular method for community detection in large networks, and it's particularly useful for identifying clusters or communities within the data.**

**Community Detection Using Louvain Algorithm**

# Clear any existing graph projection

clear\_graph\_by\_name('comm-projection')

# Create a graph projection for community detection

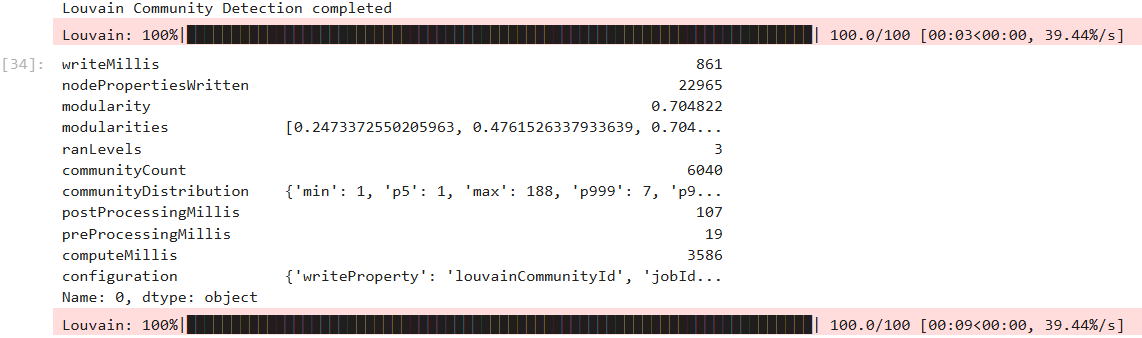
g\_lcc, \_ = gds.graph.project('comm-projection', ['Transaction', 'Customer', 'Device', 'IPAddress', 'Location'],

                         {'MADE\_BY': {'orientation': 'NATURAL'}, 'FROM\_DEVICE': {'orientation': 'NATURAL'},

                          'SHIPPED\_TO': {'orientation': 'NATURAL'}, 'ORDERED\_FROM\_IP': {'orientation': 'NATURAL'}})

print("Louvain Community Detection completed")

gds.louvain.write(g\_lcc, writeProperty='louvainCommunityId')

****

The graph analysis revealed a modularity score of 0.704822, indicating a strong community structure, with values closer to 1 suggesting better-defined communities. The algorithm identified 6,040 distinct communities, indicating many smaller clusters within the dataset.

Community size distribution showed most communities are small, with a minimum size of 1 and a 99.9th percentile size of 7, but a few large ones exist, with the largest community having 188 members.

Processing times included 631 milliseconds for writing community IDs, 104 milliseconds for post-processing, 22 milliseconds for pre-processing, and 1,615 milliseconds for the community detection itself. Higher modularity values across different Louvain algorithm levels indicated more meaningful community structures.

**Analyze Louvain Communities**

print("Louvain Communities Ordered by count of Fraudulent Transactions")

gds.run\_cypher('''

    MATCH (t:Transaction)

    WITH t.louvainCommunityId AS community,

        count(t) AS cnt,

        sum(t.isFraudulent) as fraudCount

    RETURN community,

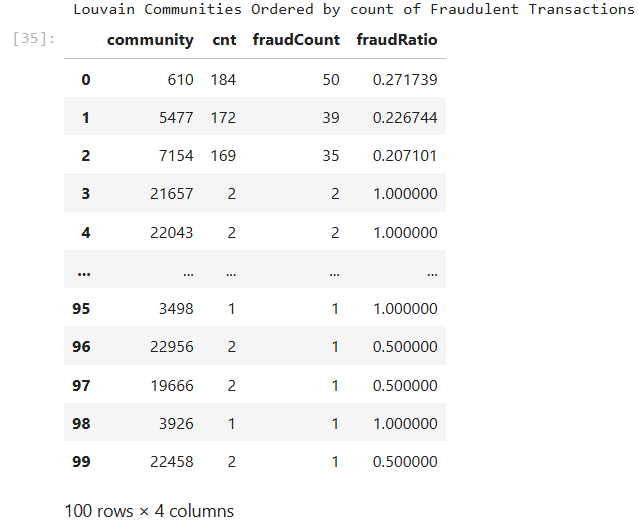
        cnt,

        fraudCount,

        toFloat(fraudCount)/toFloat(cnt) AS fraudRatio

    ORDER BY fraudCount DESC LIMIT 100

''')

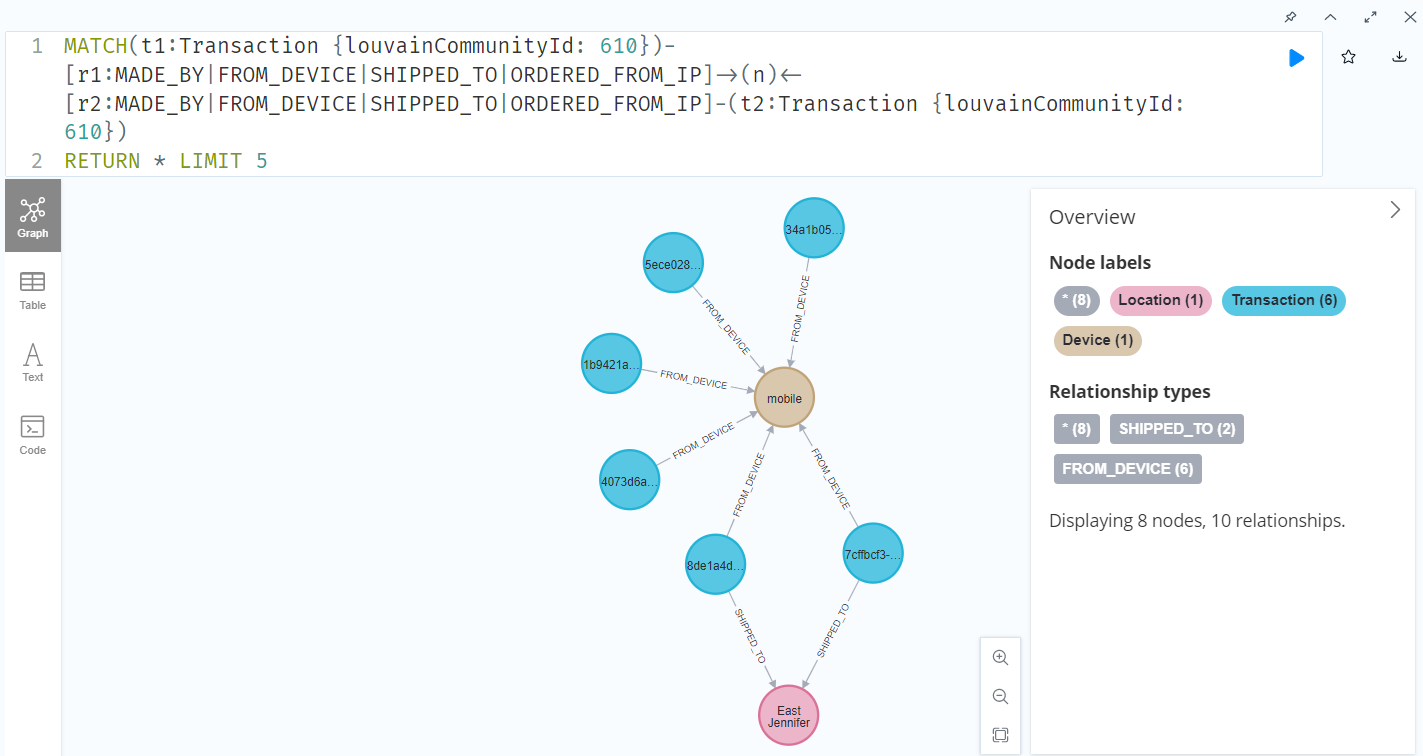
****

Community 610 had a total of 184 transactions, with 50 being fraudulent, resulting in a fraud ratio of 27.17%. In contrast, Communities 21657, 22043, and 3498 each had 2 or fewer transactions, all of which were fraudulent, giving them a fraud ratio of 100%.

**Query Analysis for community id 610**

MATCH(t1:Transaction {louvainCommunityId: 610})-[r1:MADE\_BY|FROM\_DEVICE|SHIPPED\_TO|ORDERED\_FROM\_IP]->(n)<-[r2:MADE\_BY|FROM\_DEVICE|SHIPPED\_TO|ORDERED\_FROM\_IP]-(t2:Transaction {louvainCommunityId: 610})

RETURN \* LIMIT 5

****

**Identify Potential Fraud Rings**

print("Identify potential fraud rings using Louvain Community Detection")

gds.run\_cypher('''

        MATCH (t:Transaction {isFraudulent: 1})-[:MADE\_BY]->(c:Customer)

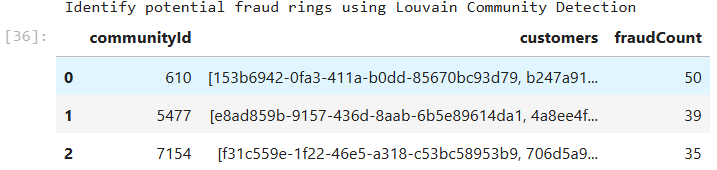
        WITH t.louvainCommunityId AS communityId, collect(c.cid) AS customers, count(\*) AS fraudCount

        WHERE fraudCount > 5

        RETURN communityId, customers, fraudCount

        ORDER BY fraudCount DESC LIMIT 5

''')

****

The communities with the highest number of fraudulent transactions are potential fraud rings. These are areas where fraudulent activity is more concentrated.

**Degree Centrality**

print("Degree Centrality")

degree\_graph = gds.graph.get('comm-projection')

gds.degree.stream(degree\_graph)

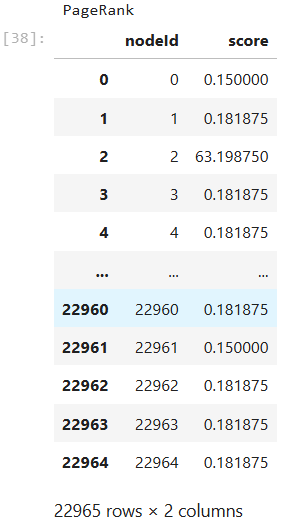
****

Nodes with higher degree centrality scores are more connected within the network, often serving as key players or hubs in fraudulent transactions. On the other hand, nodes with a degree centrality score of 0 have no direct connections and are isolated within the network. These isolated nodes may indicate either a lack of involvement in the fraud network or that they are not yet fully integrated into the fraudulent activities.

**PageRank**

print("PageRank")

gds.pageRank.stream(degree\_graph)

****

Nodes with very high PageRank scores, such as 63.198750, are considered more important within the network. These nodes are well-connected to other significant nodes and play a central role in the network's structure. Conversely, nodes with lower PageRank scores are less influential. Their lower importance may stem from having less significant connections or being more isolated from key nodes in the network.

**Clean up**

print("Fraud Detection Analysis completed")

clear\_graph\_by\_name('comm-projection')

driver.close()