Graph Database

Final Project

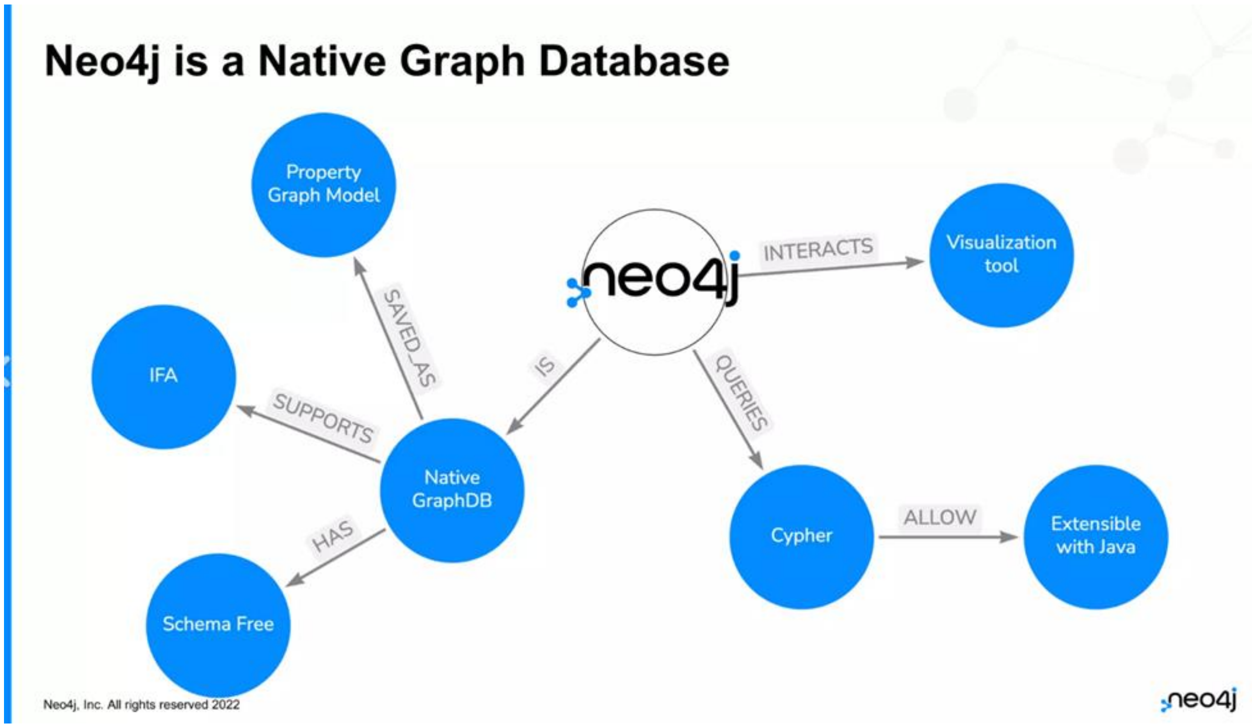
Prof: Dr Maroun Ayli

Made by: Jreige Finianos
Karim Osmani
Christa Maria Matta
Ahmad Hussein

The objective of this project is uncover the strengths of a graph database over other types of dbs.

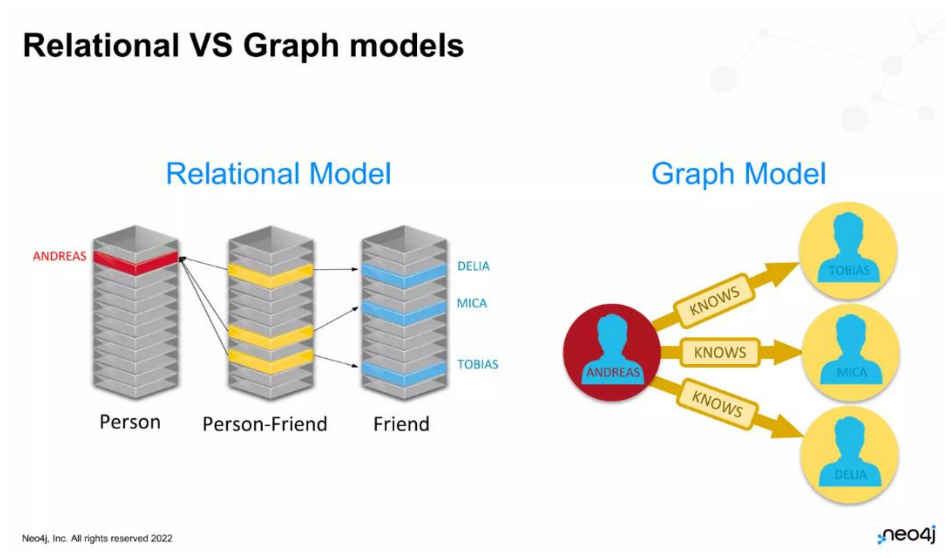
1. Neo4j

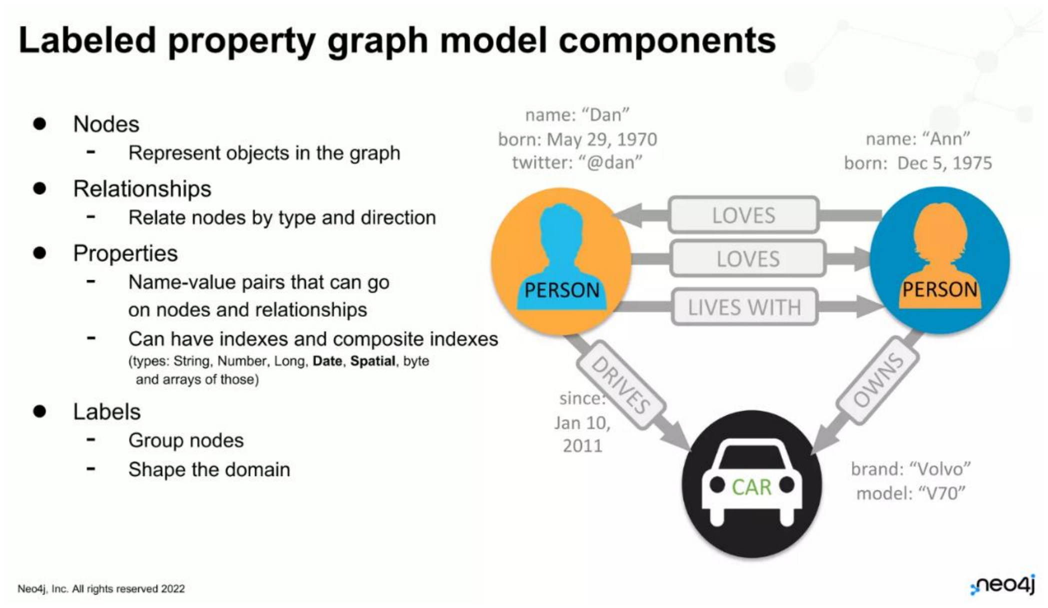
a) What is Neo4j?



Neo4j is a highly scalable, native graph database that is designed to store and process graph-structured data. It is an open-source NoSQL database that uses a graph-based data model to represent and store data. In a graph database like Neo4j, data is modeled as nodes, relationships between nodes, and properties associated with both nodes and relationships.

b) What is a Graph Database?





Labeled property graph model components

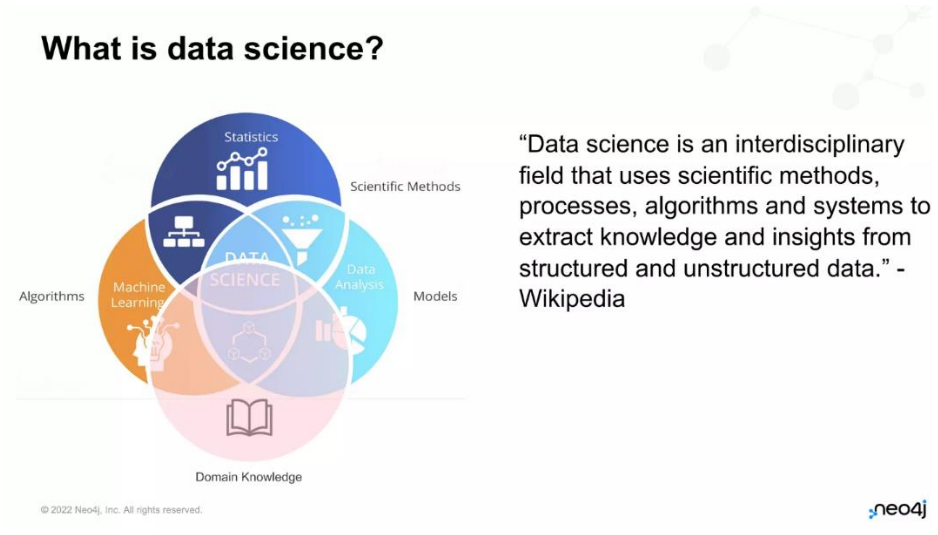
- Nodes
  - Represent objects in the graph
- Relationships
  - Relate nodes by type and direction
- Properties
  - Name-value pairs that can go on nodes and relationships
  - Can have indexes and composite indexes (types: String, Number, Long, **Date**, **Spatial**, byte and arrays of those)
- Labels
  - Group nodes
  - Shape the domain

Neo4j, Inc. All rights reserved 2022

neo4j

A graph database management system (henceforth, a graph database) is an online database management system with Create, Read, Update, and Delete (CRUD) methods that expose a graph data model. Graph databases are generally built for use with transactional (OLTP) systems. Accordingly, they are normally optimized for transactional performance, and engineered with transactional integrity and operational availability in mind. Graph database uses graph structures with nodes, edges, and properties to represent and store data. Unlike traditional relational databases, which use tables and rows to organize and relate data, graph databases focus on the relationships between entities.

c) What is Graph Data Science?

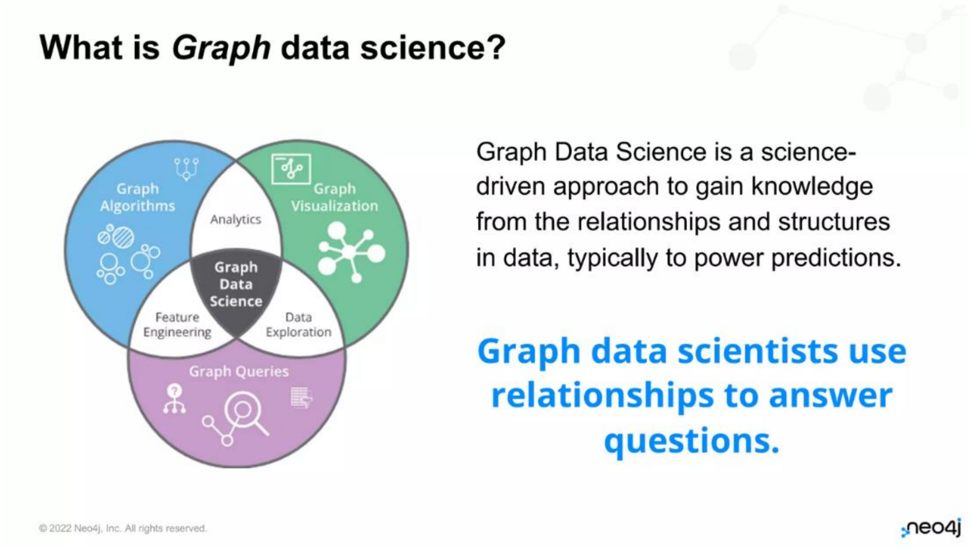


What is data science?

"Data science is an interdisciplinary field that uses scientific methods, processes, algorithms and systems to extract knowledge and insights from structured and unstructured data." - Wikipedia

© 2022 Neo4j, Inc. All rights reserved.

neo4j



What is *Graph* data science?

Graph Data Science is a science-driven approach to gain knowledge from the relationships and structures in data, typically to power predictions.

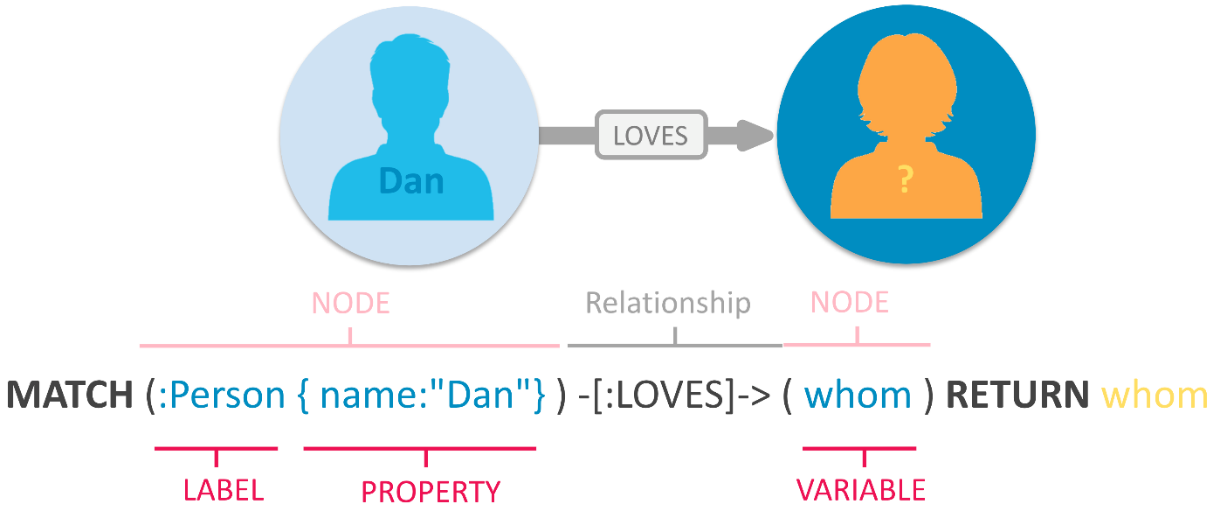
Graph data scientists use relationships to answer questions.

© 2022 Neo4j, Inc. All rights reserved.

neo4j

Graph Data Science (GDS) refers to the application of data science techniques and methodologies to analyze and extract meaningful insights from graph-structured data. Graph data, organized as nodes and edges in a graph database, is used to represent and model relationships between entities. Graph Data Science focuses on uncovering patterns, trends, and knowledge hidden within these interconnected structures

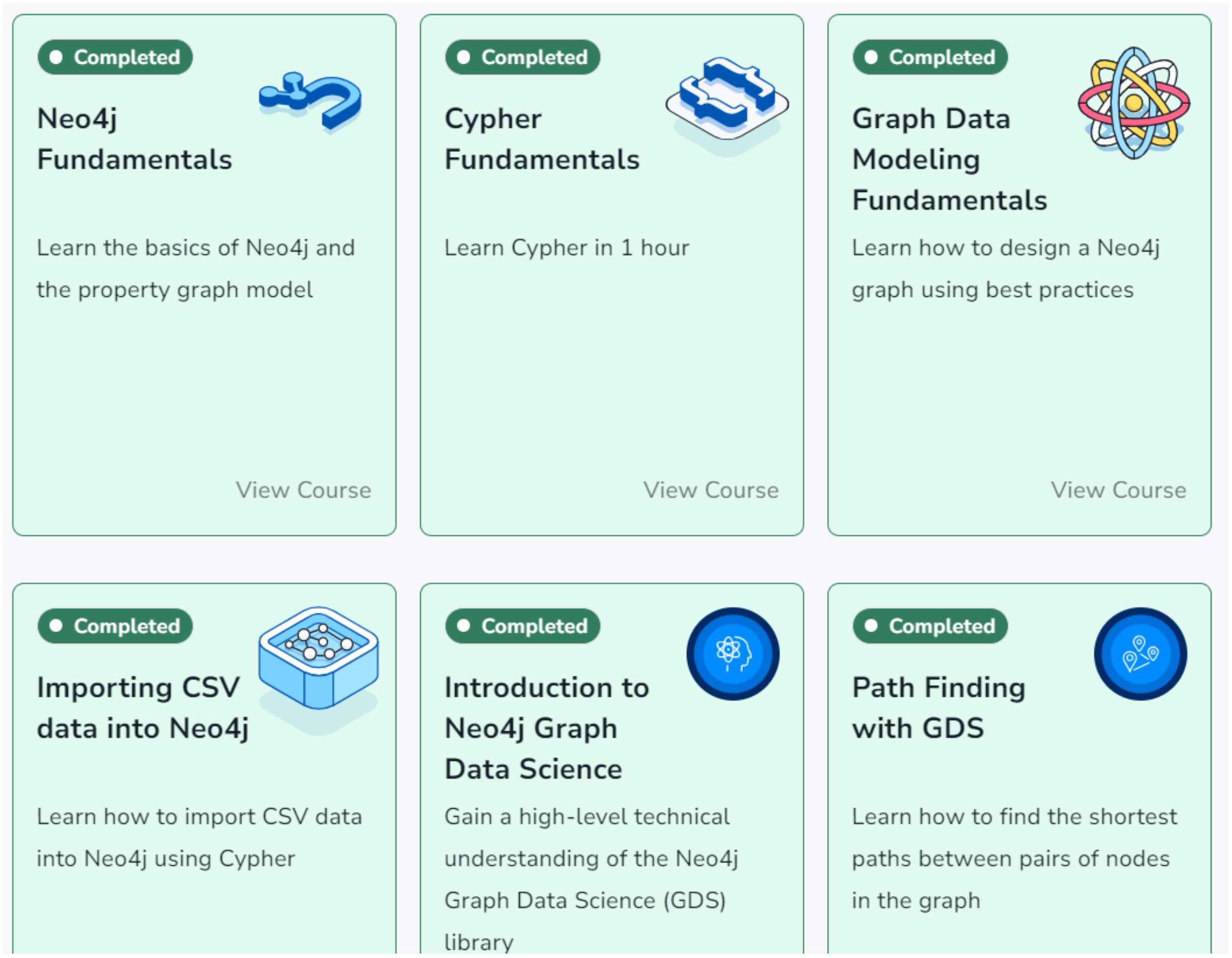
d) Neo4j's Cypher Language



```
MATCH (:Person { name:"Dan"} ) -[:LOVES]-> ( whom ) RETURN whom
```

Cypher is an expressive (yet compact) graph database query language. Although currently specific to Neo4j, its close affinity with the habit of representing graphs as diagrams makes it ideal for programmatically describing graphs. Cypher allows users to interact with the graph data model by performing operations like querying, updating, and deleting data. It is designed to be expressive, readable, and efficient for working with graph structures.

Free courses are available on neo4j website to learn the language and practice.



Completed

Neo4j Fundamentals

Learn the basics of Neo4j and the property graph model

View Course

Completed

Cypher Fundamentals

Learn Cypher in 1 hour

View Course

Completed

Graph Data Modeling Fundamentals

Learn how to design a Neo4j graph using best practices

View Course

Completed

Importing CSV data into Neo4j

Learn how to import CSV data into Neo4j using Cypher

Completed

Introduction to Neo4j Graph Data Science

Gain a high-level technical understanding of the Neo4j Graph Data Science (GDS) library

Completed

Path Finding with GDS

Learn how to find the shortest paths between pairs of nodes in the graph

We have completed these courses to be able to use cypher language.

2. The Dataset

For our project we have chosen the pre-built Twitch dataset on Neo4j.

Twitch is an online platform that allows users to share their content via live stream. Twitch streamers broadcast their gameplay or activity by sharing their screen with fans who can hear and watch them live.

The Twitch social network composes of users. A small percent of those users broadcast their gameplay or activities through live streams. In the graph model, users who do live streams are tagged with a secondary label Stream. Additional information about which teams they belong to, which games they play on stream, and in which language they present their content is present. You also know how many followers they had at the moment of scraping, the all-time historical view count, and when they created their user account. The most relevant information for network analysis is knowing which users engaged in the streamer's chat. You can distinguish if the user who chatted in the stream was a regular user (CHATTER relationship), a moderator of the stream (MODERATOR relationship), or a VIP of the stream.

The network information was scraped between the 7th and the 10th of May 2021.

3. Queries

a) Main Insights

We have performed simple queries and queries using GDS library for graph algorithms (code available in a separate output).

Using neo4j's graph database we managed to compute shortest paths, find statistics, detect communities, etc that are hard or even impossible to discover in a normal database for multiple reasons:

- Some problem would cause a headache for any relational database developer. We would need to hypothesize the order in which we would join the tables and build a complex SQL query or stored procedures to calculate the fastest route.

- Another problem with traditional databases is that we don't know beforehand how many relationships we must traverse to get from node A to node B, which leads to potentially complex and computationally expensive queries.

On the other hand, graph databases are designed to solve these problems with a few lines of code. Not only will the queries be simpler and easier to maintain, but they will almost certainly be more performant:

- When querying a relational database, any joins are calculated at runtime by scanning indexes. Not only is this a computationally expensive, but the speed of the query is also proportional to the total size of the data. The more information added to the database, the slower your queries will be.

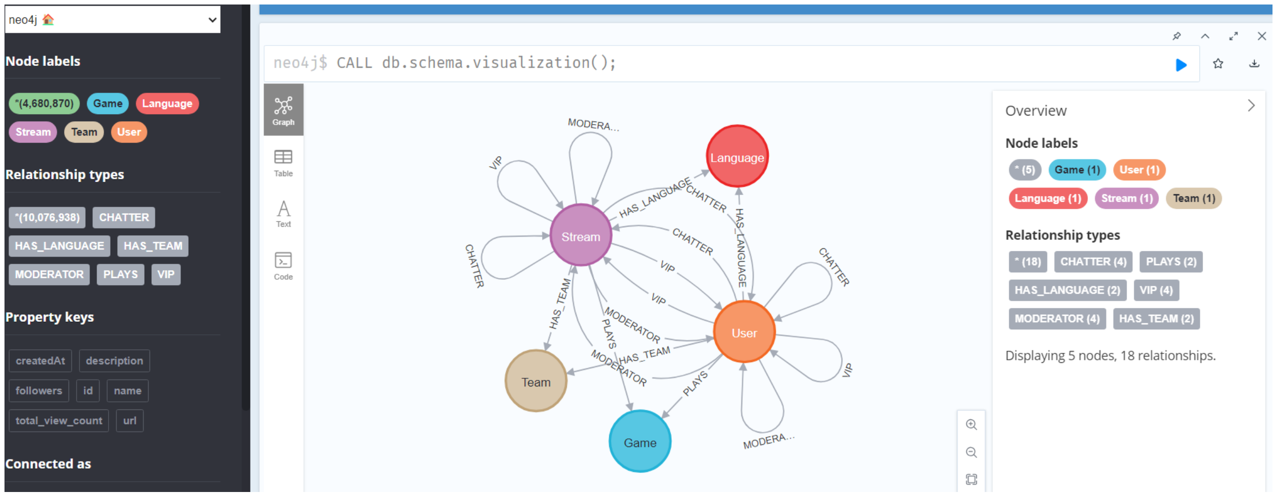
- In Neo4j, the data is stored in such a way that every node is aware of every incoming and outgoing relationship, and traversals through the network are instead performed by pointer chasing in memory.

b) Normal Queries

We have performed normal queries to help us visualize the data, find some useful information.

1. Query: Database schema (nodes and relationships)
   CALL db.schema.visualization();

   Result:



```
neo4j$ CALL db.schema.visualization();
```

neo4j
Node labels
*(4,680,870) Game Language Stream Team User
Relationship types
*(10,076,938) CHATTER HAS_LANGUAGE HAS_TEAM MODERATOR PLAYS VIP
Property keys
createdAt description followers id name total_view_count url
Connected as
Graph
Table
Text
Code
Stream
Language
User
Team
Game
MODERA...
VIP
CHATTER
HAS_LANGUAGE
HAS_TEAM
MODERATOR
PLAYS
Overview
Node labels
* (5) Game (1) User (1) Language (1) Stream (1) Team (1)
Relationship types
* (18) CHATTER (4) PLAYS (2) HAS_LANGUAGE (2) VIP (4) MODERATOR (4) HAS_TEAM (2)
Displaying 5 nodes, 18 relationships.

Interpretation:

Node Labels:

- **Game**: Represents the various games that are being streamed or played on Twitch.
- **Language**: Represents the different languages that streams or users are categorized by.
- **Stream**: Represents individual streaming sessions or channels.
- **Team**: Indicates groups or teams of streamers on Twitch.
- **User**: Represents the users on Twitch, which could be streamers, viewers, or both.

Relationship Types:

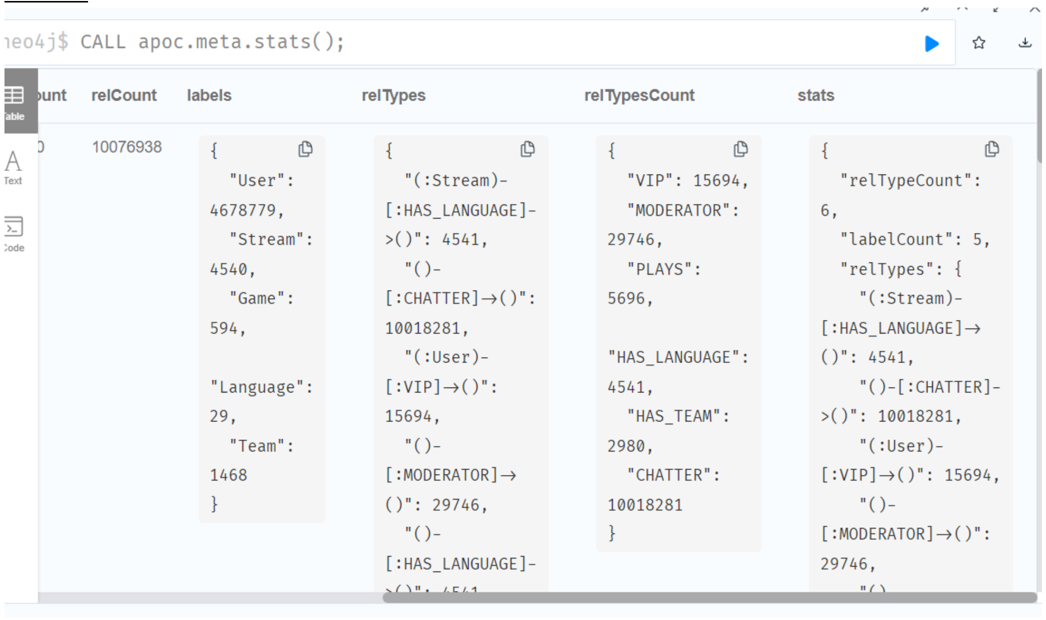
- **CHATTER**: This relationship indicates users who chat or interact in a stream.
- **HAS_LANGUAGE**: This relationship connects users or streams with a specific language, indicating the language used by the user or in the stream.
- **HAS_TEAM**: Indicates that a user is part of a team.
- **MODERATOR**: Suggests that a user has moderator privileges in a stream.
- **PLAYS**: Connects a user or stream to a game, indicating that the user plays this game or the game is being played in the stream.
- **VIP**: Represents users who have VIP status in a stream.

Property Keys:

- Various properties like **createdAt**, **description**, **followers**, **id**, **name**, **total_view_count**, **url**, etc., are used to store specific information about each node. For instance, **createdAt** is the date a stream was started, or a user joined Twitch.

2. Query:
   //Counts of nodes, node labels, relationships, relationship types,
   //property keys and statistics using the APOC library.
   CALL apoc.meta.stats();

   Result:



```
ieo4j$ CALL apoc.meta.stats();
```

ount	relCount	labels	relTypes	relTypesCount	stats
0	10076938	{ "User": 4678779, "Stream": 4540, "Game": 594, "Language": 29, "Team": 1468 }	{ "(:Stream)-[:HAS_LANGUAGE]->()": 4541, "()-[:CHATTER]->()": 10018281, "(:User)-[:VIP]->()": 15694, "()-[:MODERATOR]->()": 29746, "()-[:HAS_LANGUAGE]->()": 4541	{ "VIP": 15694, "MODERATOR": 29746, "PLAYS": 5696, "HAS_LANGUAGE": 4541, "HAS_TEAM": 2980, "CHATTER": 10018281 }	{ "relTypeCount": 6, "labelCount": 5, "relTypes": { "(:Stream)-[:HAS_LANGUAGE]->()": 4541, "()-[:CHATTER]->()": 10018281, "(:User)-[:VIP]->()": 15694, "()-[:MODERATOR]->()": 29746, "()

   Interpretation:
   - Using the **CALL apoc.meta.stats();**, we gather statistical information about the structure of Neo4j graph. The APOC library is an add-on that provides many useful procedures and functions, including meta information about your database.

   - The **CHATTER** relationship has the highest count with over 10 million relationships, indicating a very high level of interaction among users in streams. This is a typical example where graph databases excel because they can handle this high volume of relationships efficiently.

   - The number of **VIP** and **MODERATOR** relationships suggests that there are quite a few privileged users, which is an indicator of user engagement and community management within the platform.

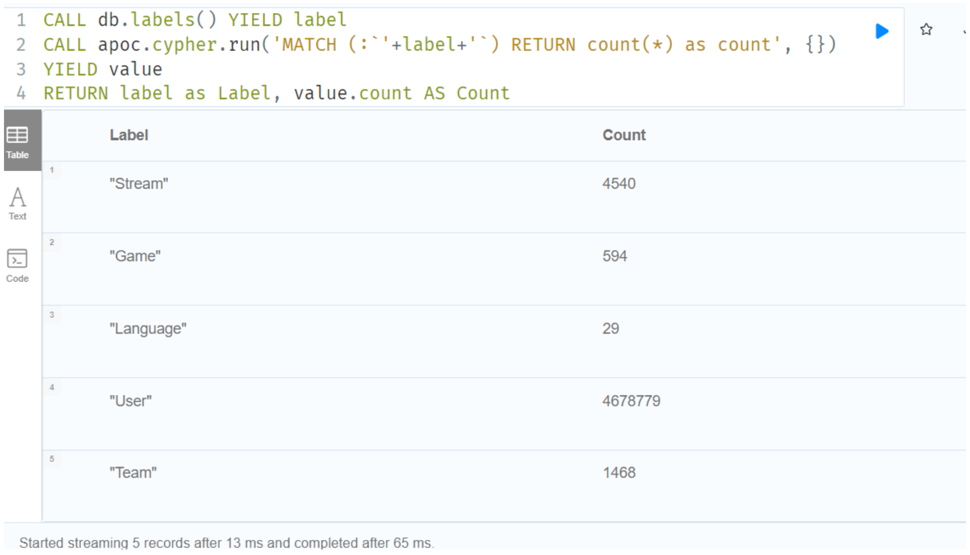
   - There are 29 **Language** nodes, which might seem small but has a significant number of **HAS_LANGUAGE** relationships. This could imply that streams are categorized into several languages, allowing for language-specific querying and analysis.

- The **Game** nodes are relatively few compared to **User** nodes, but the **PLAYS** relationship is substantial, indicating that these games are likely very popular and central to the user experience on Twitch.

- The presence of **Team** nodes and **HAS_TEAM** relationships points towards the existence of organized groups or communities within the Twitch ecosystem, which could be valuable for targeted marketing, community analysis, or feature development.

3. Query:
   //Node labels and count
   CALL db.labels() YIELD label
   CALL apoc.cypher.run('MATCH (:`'+label+'`) RETURN count(*) as count', {})
   YIELD value
   RETURN label as Label, value.count AS Count

   Result:



```
CALL db.labels() YIELD label
CALL apoc.cypher.run('MATCH (:`'+label+'`) RETURN count(*) as count', {})
YIELD value
RETURN label as Label, value.count AS Count
```

	Label	Count
1	"Stream"	4540
2	"Game"	594
3	"Language"	29
4	"User"	4678779
5	"Team"	1468

Started streaming 5 records after 13 ms and completed after 65 ms.

   Interpretation:
   - The query provided combines two Cypher statements using the APOC library to retrieve the count of nodes for each label.

   - The **User** label has by far the most nodes, which suggests that the majority of the data in the database is centered around Twitch users.

   - **Stream** nodes, while significantly fewer than **User** nodes, are still quite numerous, which indicate that there are multiple streams per user or that a substantial number of users are involved in streaming.

- **Game** nodes are relatively small in number, suggesting that the number of unique games is limited, but each game could be associated with many streams and users, implying a many-to-many relationship.

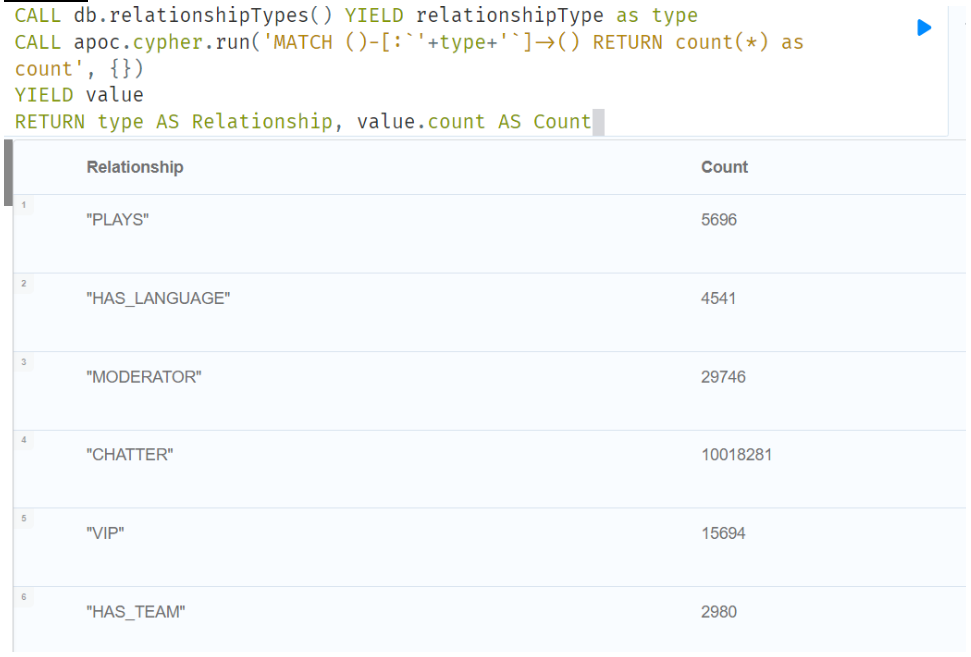
- **Language** nodes have the smallest count, which indicate a small set of languages used within the Twitch dataset or that language information is not the primary focus of the dataset.

- **Team** nodes are moderate in number, showing that team-based streaming or activities are a recognized but not dominant aspect of the Twitch community within the dataset.

4. Query:

//Relationship types and count
CALL db.relationshipTypes()
YIELD relationshipType as type
CALL apoc.cypher.run('MATCH ()-[:`'+type+'`]->() RETURN count(*) as count', {})
YIELD value
RETURN type AS Relationship, value.count AS Count

Result:



```
CALL db.relationshipTypes() YIELD relationshipType as type
CALL apoc.cypher.run('MATCH ()-[:`'+type+'`]→() RETURN count(*) as
count', {})
YIELD value
RETURN type AS Relationship, value.count AS Count
```

	Relationship	Count
1	"PLAYS"	5696
2	"HAS_LANGUAGE"	4541
3	"MODERATOR"	29746
4	"CHATTER"	10018281
5	"VIP"	15694
6	"HAS_TEAM"	2980

Interpretation:

- The **CHATTER** relationship type has the highest count, indicating that the interaction between users in the context of chatting is the most common activity captured in the database.

- The **MODERATOR** relationship type is also quite high, suggesting that moderation activity is significant on the platform.

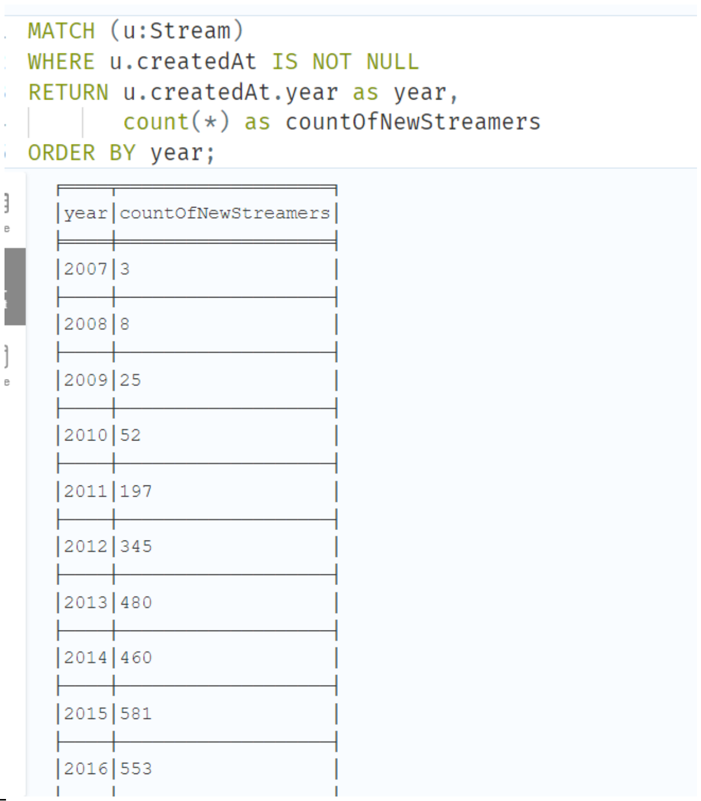
- The **VIP** relationship type has a substantial number of relationships, which indicate a significant number of users with special status or privileges in certain streams.

- **PLAYS** and **HAS_LANGUAGE** relationship types have lower counts in comparison to others, but they are still notable as they reflect important aspects of the Twitch platform: the games being played, and the languages being used.

- **HAS_TEAM** has the lowest count among the relationships shown, which suggest that not all users are associated with a team, or team-related activities are less common compared to other types of interactions.
-

5. Query:

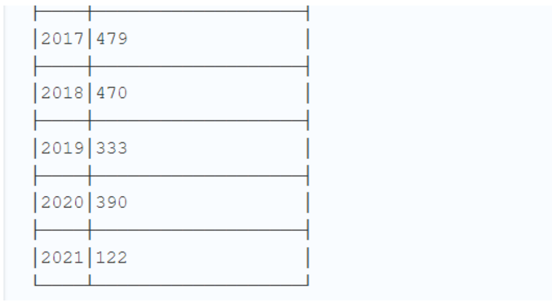
```
//Nb of streamers per their account creation year
MATCH (u:Stream)
WHERE u.createdAt IS NOT NULL
RETURN u.createdAt.year as year, count(*) as countOfNewStreamers
ORDER BY year;
```



```
MATCH (u:Stream)
WHERE u.createdAt IS NOT NULL
RETURN u.createdAt.year as year,
       count(*) as countOfNewStreamers
ORDER BY year;
```

Result:

```
╒════╤═══════════════════╕
│year│countOfNewStreamers│
╞════╪═══════════════════╡
│2007│3                  │
├────┼───────────────────┤
│2008│8                  │
├────┼───────────────────┤
│2009│25                 │
├────┼───────────────────┤
│2010│52                 │
├────┼───────────────────┤
│2011│197                │
├────┼───────────────────┤
│2012│345                │
├────┼───────────────────┤
│2013│480                │
├────┼───────────────────┤
│2014│460                │
├────┼───────────────────┤
│2015│581                │
├────┼───────────────────┤
│2016│553                │
```



```
├────┼───────────────────┤
│2017│479                │
├────┼───────────────────┤
│2018│470                │
├────┼───────────────────┤
│2019│333                │
├────┼───────────────────┤
│2020│390                │
├────┼───────────────────┤
│2021│122                │
└────┴───────────────────┘
```

Interpretation:

- This query returns the number of Twitch streamers who created their accounts each year. It matches nodes with the label **Stream** that have a non-null **createdAt** property, extracts the year from the **createdAt** property, counts the number of nodes for each year, and orders the results by year.

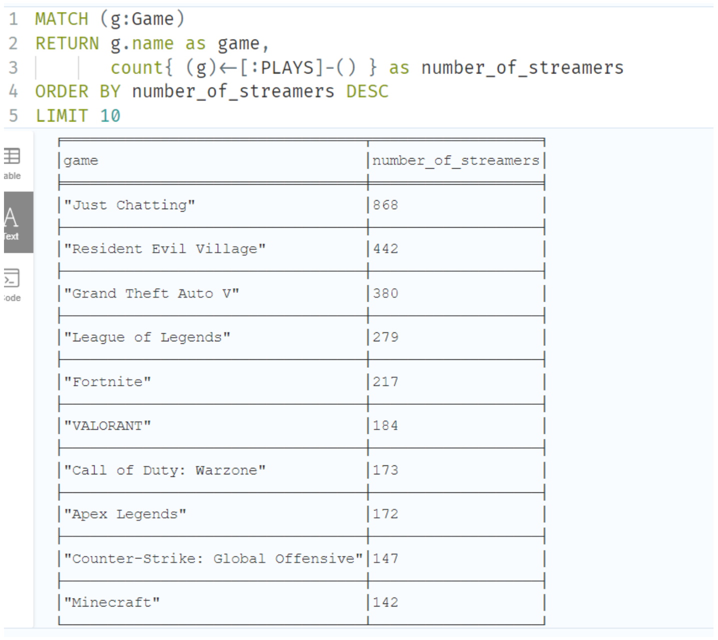
- The growth in the number of streamers creating accounts on Twitch shows a noticeable increase from 2007 to 2015, with a peak in 2015.

- After 2015, there is a slight decline in the number of new streamers each year, with some fluctuations. This could be due to various factors, such as market saturation, the rise of competing platforms, changes in Twitch policies, or other external factors.

- The significant drop in 2021 might indicate incomplete data for that year, or it could suggest a decline in new streamer account creations.

6. Query:
//Games that have the highest count of streamers playing them
MATCH (g:Game)
RETURN g.name as game, count{ (g)<-[:PLAYS]-() } as number_of_streamers
ORDER BY number_of_streamers DESC
LIMIT 10

Result:



```
MATCH (g:Game)
RETURN g.name as game,
       count{ (g)←[:PLAYS]-() } as number_of_streamers
ORDER BY number_of_streamers DESC
LIMIT 10
```

```
╒═════════════════════════════════╤═══════════════════╕
│game                             │number_of_streamers│
╞═════════════════════════════════╪═══════════════════╡
│"Just Chatting"                  │868                │
├─────────────────────────────────┼───────────────────┤
│"Resident Evil Village"          │442                │
├─────────────────────────────────┼───────────────────┤
│"Grand Theft Auto V"             │380                │
├─────────────────────────────────┼───────────────────┤
│"League of Legends"              │279                │
├─────────────────────────────────┼───────────────────┤
│"Fortnite"                       │217                │
├─────────────────────────────────┼───────────────────┤
│"VALORANT"                       │184                │
├─────────────────────────────────┼───────────────────┤
│"Call of Duty: Warzone"          │173                │
├─────────────────────────────────┼───────────────────┤
│"Apex Legends"                   │172                │
├─────────────────────────────────┼───────────────────┤
│"Counter-Strike: Global Offensive"│147               │
├─────────────────────────────────┼───────────────────┤
│"Minecraft"                      │142                │
└─────────────────────────────────┴───────────────────┘
```

Interpretation:

- This query identifies the top 10 games with the highest count of streamers, it does this by matching nodes with the label **Game** and then counting the incoming **PLAYS** relationships to each game node.

- "Just Chatting" is the most popular category, which is not a game per se but a category where streamers interact with their viewers. Its high ranking reflects the importance of community and interaction on Twitch beyond just gameplay.

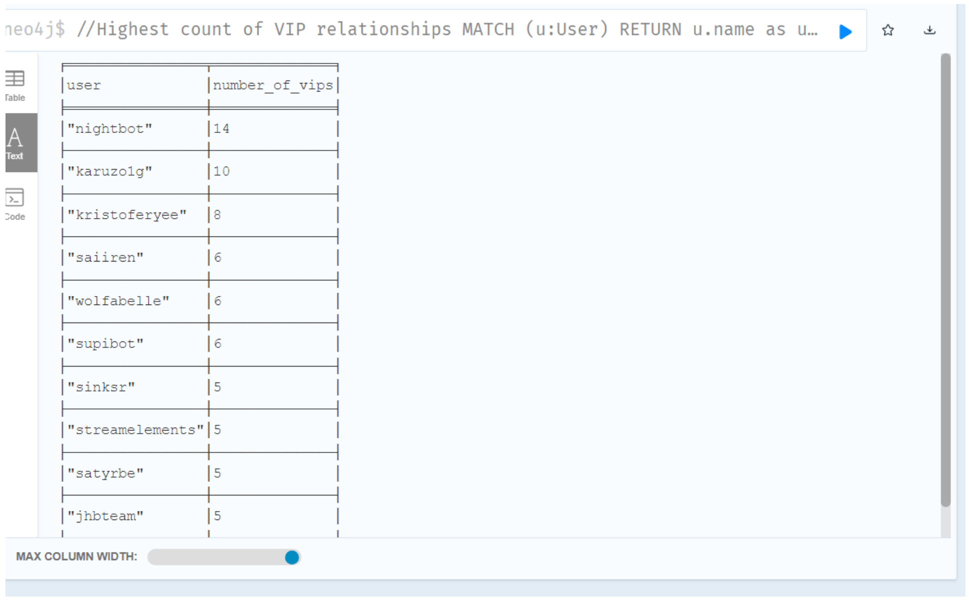
- "Resident Evil Village" and "Grand Theft Auto V" are among the top games, indicating their popularity within the Twitch community at the time this data was collected.

- There is a diverse range of game genres represented in the top 10, from action-adventure games and battle royale games to first-person shooters and sandbox games.

- The presence of long-standing titles like "League of Legends", "Grand Theft Auto V", and "Minecraft" demonstrates their enduring appeal and the loyalty of their streaming communities.

7. Query:
//Highest count of VIP relationships
MATCH (u:User)
RETURN u.name as user, count{ (u)-[:VIP]->() } as number_of_vips
ORDER BY number_of_vips DESC
LIMIT 10;

Result:


```
neo4j$ //Highest count of VIP relationships MATCH (u:User) RETURN u.name as u…
```
```
╒════════════════╤══════════════╕
│user            │number_of_vips│
╞════════════════╪══════════════╡
│"nightbot"      │14            │
├────────────────┼──────────────┤
│"karuzolg"      │10            │
├────────────────┼──────────────┤
│"kristoferyee"  │8             │
├────────────────┼──────────────┤
│"saiiren"       │6             │
├────────────────┼──────────────┤
│"wolfabelle"    │6             │
├────────────────┼──────────────┤
│"supibot"       │6             │
├────────────────┼──────────────┤
│"sinksr"        │5             │
├────────────────┼──────────────┤
│"streamelements"│5             │
├────────────────┼──────────────┤
│"satyrbe"       │5             │
├────────────────┼──────────────┤
│"jhbteam"       │5             │
```

Interpretation:

- This query finds the top 10 users with the highest number of VIP relationships. VIP status in Twitch streams indicates a user who is given special privileges or recognition within a channel.

- "nightbot" has the highest number of VIP relationships, which is unusual as "Nightbot" is typically a bot used for moderation. This might suggest that the bot is being recognized as a VIP for administrative purposes or that the name is being used for other reasons.

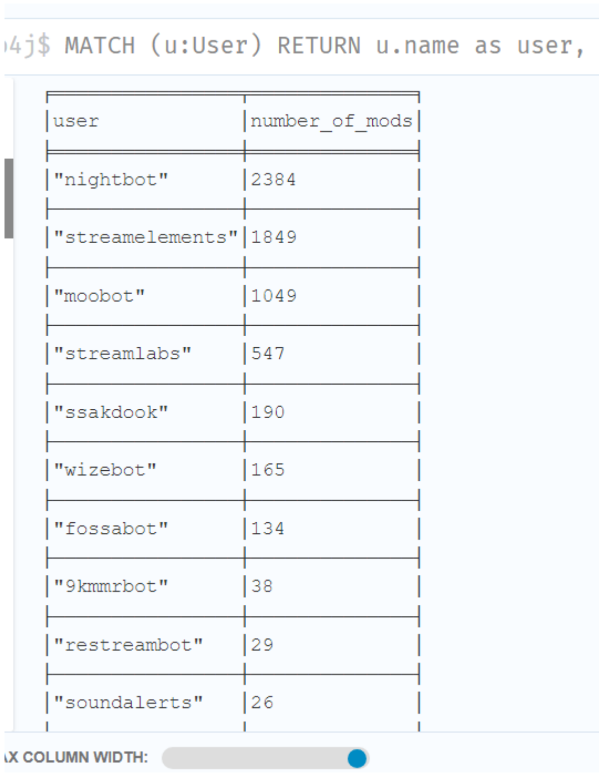
- The users listed vary in the number of VIP relationships they have, indicating different levels of influence or recognition within the Twitch community.

- The presence of other bot-like names such as "supibot" and "streamelements" indicate automated systems or services that are being given VIP status in channels.

8. Query:
//Highest count of moderators relationships
//Moderators (also known as mods) ensure that the chat meets the behavior and content standards set by the broadcaster by removing offensive posts and spam that detracts from conversations.
MATCH (u:User)
RETURN u.name as user, count{ (u)-[:MODERATOR]->() } as number_of_mods
ORDER BY number_of_mods DESC

LIMIT 10;

Result:



```
4j$ MATCH (u:User) RETURN u.name as user,
```

user	number_of_mods
"nightbot"	2384
"streamelements"	1849
"moobot"	1049
"streamlabs"	547
"ssakdook"	190
"wizebot"	165
"fossabot"	134
"9kmmrbot"	38
"restreambot"	29
"soundalerts"	26

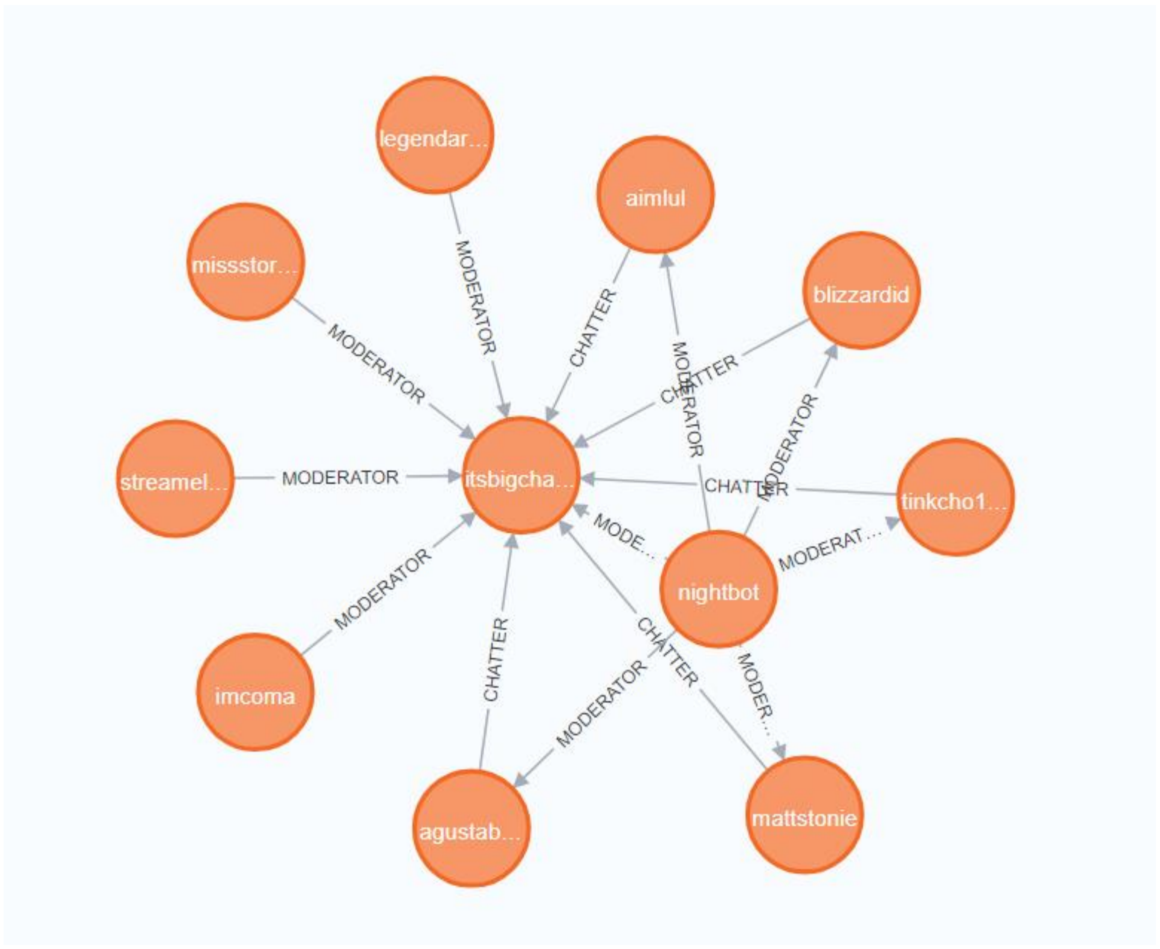
Interpretation:

- This query is useful to see which users have been granted moderator status across different Twitch channels.
- The users listed are primarily bots ("nightbot", "streamelements", "moobot", etc.), which are commonly used as moderators in Twitch channels to automate chat moderation tasks.
- The high count of moderator relationships for these bots indicates their widespread use across multiple channels on Twitch.
- The use of bots for moderation points towards a need for automated chat moderation due to the large scale of interactions and potential for spam or inappropriate content in busy channels.
- Human moderators (if any among the listed names) may have significant influence and responsibility within the Twitch community due to their roles in a large number of channels.

9. Query:

```
//Matching five streamers and separately five users that chatted in the original streamer's broadcast
MATCH (s:Stream)
WITH s LIMIT 1
CALL { WITH s
MATCH p=(s)<--(:Stream)
RETURN p
LIMIT 5
UNION WITH s
MATCH p=(s)<--(:User)
RETURN p LIMIT 5 }
RETURN p
//We can see that streamers behave like regular users. They can chat in other streamer's broadcasts, be their moderator or VIP.
```

Result:



Interpretation:

- This query aims to match one stream and then find five streamers and five users that interacted with that stream, either by chatting or having some other defined relationship like moderator or VIP.

- Moderator Relationships: Some users, like aimlul, blizzardid, and nightbot, have MODERATOR relationships with itsbigchase, meaning they have permissions to moderate the chat for itsbigchase's stream.

- Chatter Relationships: Several users are connected with CHATTER relationships, indicating they have participated in the chat during itsbigchase's broadcast.

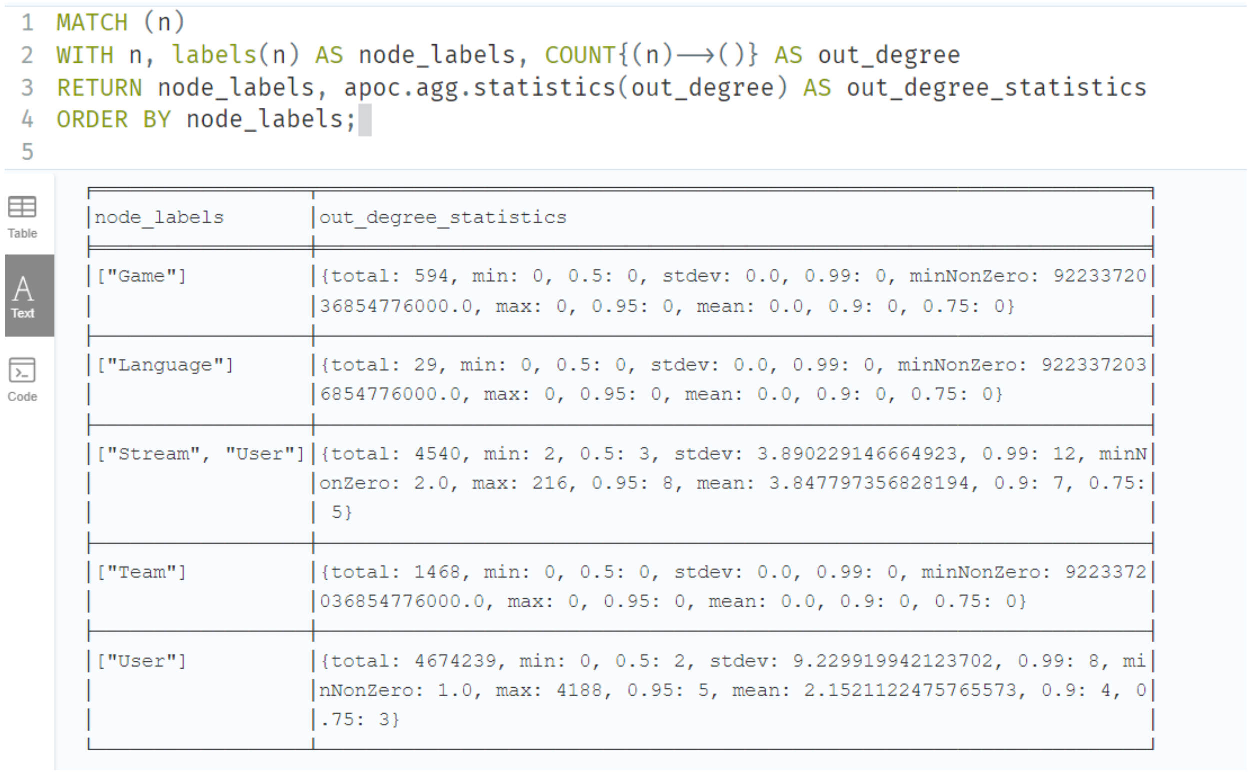
- Other Streamers: The query also seems to have returned other streamers (nodes that are of type Stream and not User) that have some sort of relationship with itsbigchase, which might not be the intended result of the query. The expected behavior would be to find streamers who are acting as users in this context (chatting, moderating, etc.).

- This visualization provides insights into the social dynamics of Twitch, highlighting active users and potential influencers within the community. It can also be used to identify key collaborative or competitive relationships between streamers.

10. Query:

```
//Out degree of all nodes label
MATCH (n)
WITH n, labels(n) AS node_labels,
COUNT{(n)-->()} AS out_degree
RETURN node_labels, apoc.agg.statistics(out_degree) AS out_degree_statistics
ORDER BY node_labels;
```

Result:



```
MATCH (n)
WITH n, labels(n) AS node_labels, COUNT{(n)-->()} AS out_degree
RETURN node_labels, apoc.agg.statistics(out_degree) AS out_degree_statistics
ORDER BY node_labels;
```

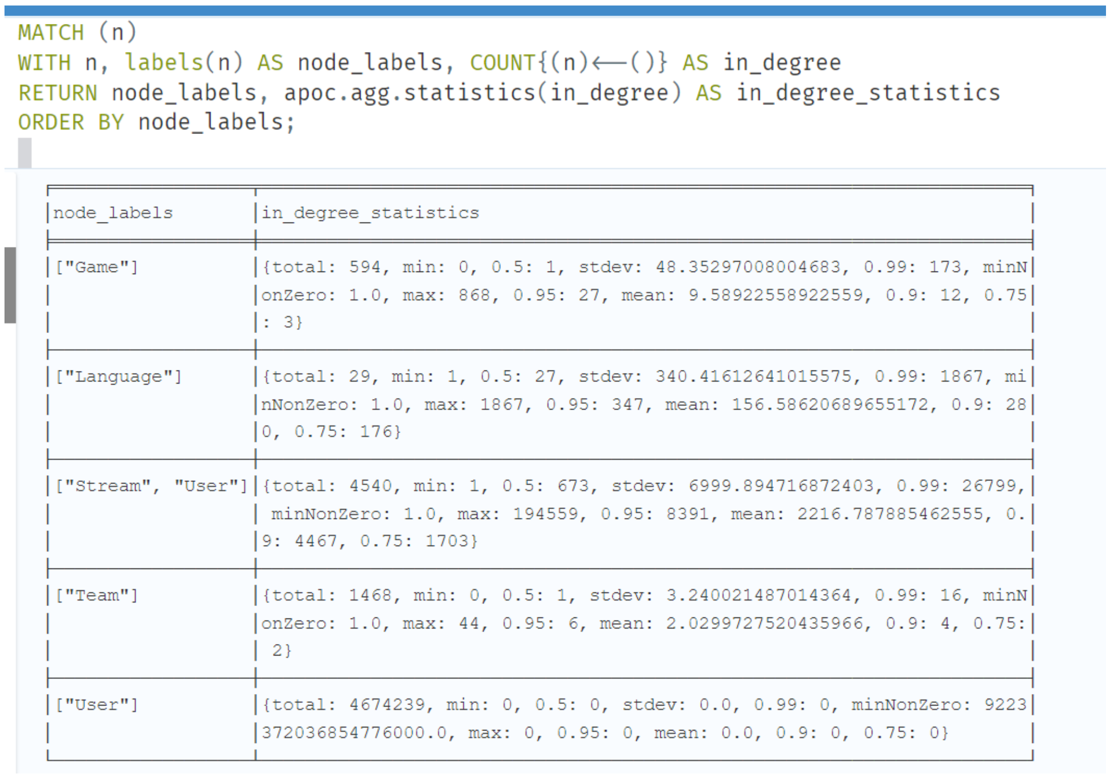
```
╒══════════════════╤═════════════════════════════════════════════════════════════════════╕
│node_labels       │out_degree_statistics                                                │
╞══════════════════╪═════════════════════════════════════════════════════════════════════╡
│["Game"]          │{total: 594, min: 0, 0.5: 0, stdev: 0.0, 0.99: 0, minNonZero: 92233720│
│                  │36854776000.0, max: 0, 0.95: 0, mean: 0.0, 0.9: 0, 0.75: 0}          │
├──────────────────┼─────────────────────────────────────────────────────────────────────┤
│["Language"]      │{total: 29, min: 0, 0.5: 0, stdev: 0.0, 0.99: 0, minNonZero: 922337203│
│                  │6854776000.0, max: 0, 0.95: 0, mean: 0.0, 0.9: 0, 0.75: 0}           │
├──────────────────┼─────────────────────────────────────────────────────────────────────┤
│["Stream", "User"]│{total: 4540, min: 2, 0.5: 3, stdev: 3.890229146664923, 0.99: 12, minN│
│                  │onZero: 2.0, max: 216, 0.95: 8, mean: 3.847797356828194, 0.9: 7, 0.75:│
│                  │ 5}                                                                  │
├──────────────────┼─────────────────────────────────────────────────────────────────────┤
│["Team"]          │{total: 1468, min: 0, 0.5: 0, stdev: 0.0, 0.99: 0, minNonZero: 9223372│
│                  │036854776000.0, max: 0, 0.95: 0, mean: 0.0, 0.9: 0, 0.75: 0}         │
├──────────────────┼─────────────────────────────────────────────────────────────────────┤
│["User"]          │{total: 4674239, min: 0, 0.5: 2, stdev: 9.229919942123702, 0.99: 8, mi│
│                  │nNonZero: 1.0, max: 4188, 0.95: 5, mean: 2.1521122475765573, 0.9: 4, 0│
│                  │.75: 3}                                                              │
└──────────────────┴─────────────────────────────────────────────────────────────────────┘
```

Interpretation:

- The out-degree of a node is the number of outgoing relationships it has. These statistics can provide insights into how nodes of different types are connected within your graph.
- User nodes are the most connected, which aligns with the expectation that users are the primary actors in the Twitch ecosystem, engaging in various activities that result in outgoing relationships.
- The high maximum out-degree for User nodes indicates that there are some highly active users on the platform with a large number of interactions.

11. Query:
//In degree of all nodes label MATCH (n) WITH n, labels(n) AS node_labels, COUNT{(n)<--()} AS in_degree RETURN node_labels, apoc.agg.statistics(in_degree) AS in_degree_statistics ORDER BY node_labels;

Result:



```
MATCH (n)
WITH n, labels(n) AS node_labels, COUNT{(n)<--()} AS in_degree
RETURN node_labels, apoc.agg.statistics(in_degree) AS in_degree_statistics
ORDER BY node_labels;
```

```
╒═══════════════════╤══════════════════════════════════════════════════════════════════════════╕
│node_labels        │in_degree_statistics                                                      │
╞═══════════════════╪══════════════════════════════════════════════════════════════════════════╡
│["Game"]           │{total: 594, min: 0, 0.5: 1, stdev: 48.35297008004683, 0.99: 173, minN│
│                   │onZero: 1.0, max: 868, 0.95: 27, mean: 9.58922558922559, 0.9: 12, 0.75│
│                   │: 3}                                                                      │
├───────────────────┼──────────────────────────────────────────────────────────────────────────┤
│["Language"]       │{total: 29, min: 1, 0.5: 27, stdev: 340.41612641015575, 0.99: 1867, mi│
│                   │nNonZero: 1.0, max: 1867, 0.95: 347, mean: 156.58620689655172, 0.9: 28│
│                   │0, 0.75: 176}                                                             │
├───────────────────┼──────────────────────────────────────────────────────────────────────────┤
│["Stream", "User"]│{total: 4540, min: 1, 0.5: 673, stdev: 6999.894716872403, 0.99: 26799,│
│                   │ minNonZero: 1.0, max: 194559, 0.95: 8391, mean: 2216.787885462555, 0.│
│                   │9: 4467, 0.75: 1703}                                                      │
├───────────────────┼──────────────────────────────────────────────────────────────────────────┤
│["Team"]           │{total: 1468, min: 0, 0.5: 1, stdev: 3.240021487014364, 0.99: 16, minN│
│                   │onZero: 1.0, max: 44, 0.95: 6, mean: 2.0299727520435966, 0.9: 4, 0.75:│
│                   │ 2}                                                                       │
├───────────────────┼──────────────────────────────────────────────────────────────────────────┤
│["User"]           │{total: 4674239, min: 0, 0.5: 0, stdev: 0.0, 0.99: 0, minNonZero: 9223│
│                   │372036854776000.0, max: 0, 0.95: 0, mean: 0.0, 0.9: 0, 0.75: 0}         │
└───────────────────┴──────────────────────────────────────────────────────────────────────────┘
```

<u>Interpretation:</u>

- The statistics reveal the existence of super-connected nodes, particularly within the User category, which might represent highly influential users or bots on Twitch.

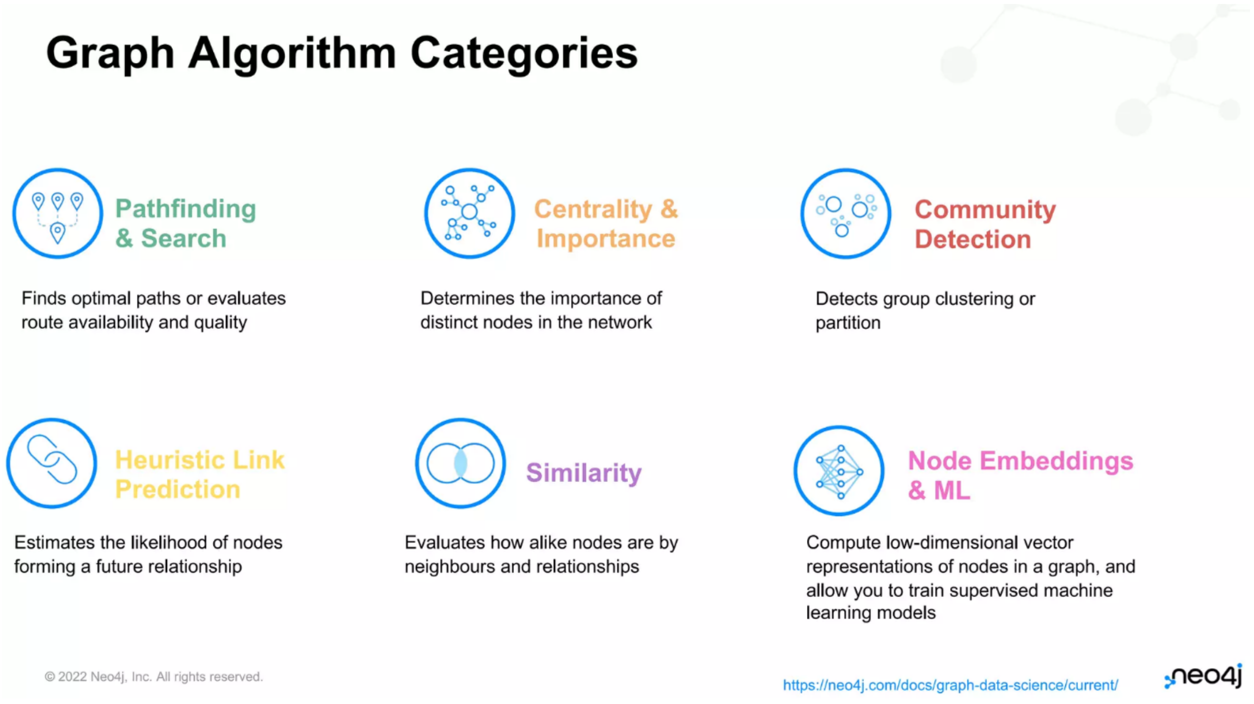
- The large maximum in-degrees for Language and Stream categories suggest that there are particular nodes within these categories that serve as hubs, perhaps channels that are conducted in a common language or users who are central to the Twitch community.

The above insights extracted show the significant advantages offered by Graph databases when it comes to understanding and interpreting complex networks, particularly in environments characterized by rich interconnectivity and numerous relationships, such as social platforms, recommendation systems, and more. They excel in their ability to compute relationship-based metrics efficiently, which traditional relational databases may struggle with due to the need for complex joins and queries. The schema-less nature of graph databases allows for greater flexibility in querying and data modeling, enabling the quick adaptation of the database to new types of data or query requirements without restructuring the entire schema. Furthermore, tools like the APOC library extend the capabilities of graph databases like Neo4j by providing powerful aggregation functions that simplify the calculation of detailed statistics on node relationships. These capabilities are essential for identifying patterns and insights within the data, such as pinpointing key influencers, popular topics, or dominant community dynamics. Such insights are not just academically interesting but hold practical value for platform owners and marketers in making informed decisions that drive user engagement and business growth.

Moving to the graph algorithms using the GDS library.

c) Graph Algorithms Queries

The GDS library executes graph algorithms on a specialized in-memory graph format to improve the performance and scale of graph algorithms. Using native or cypher projections, we can project the stored graph in our database to the in-memory graph format.



Graph Algorithm Categories

Pathfinding & Search
Finds optimal paths or evaluates route availability and quality

Centrality & Importance
Determines the importance of distinct nodes in the network

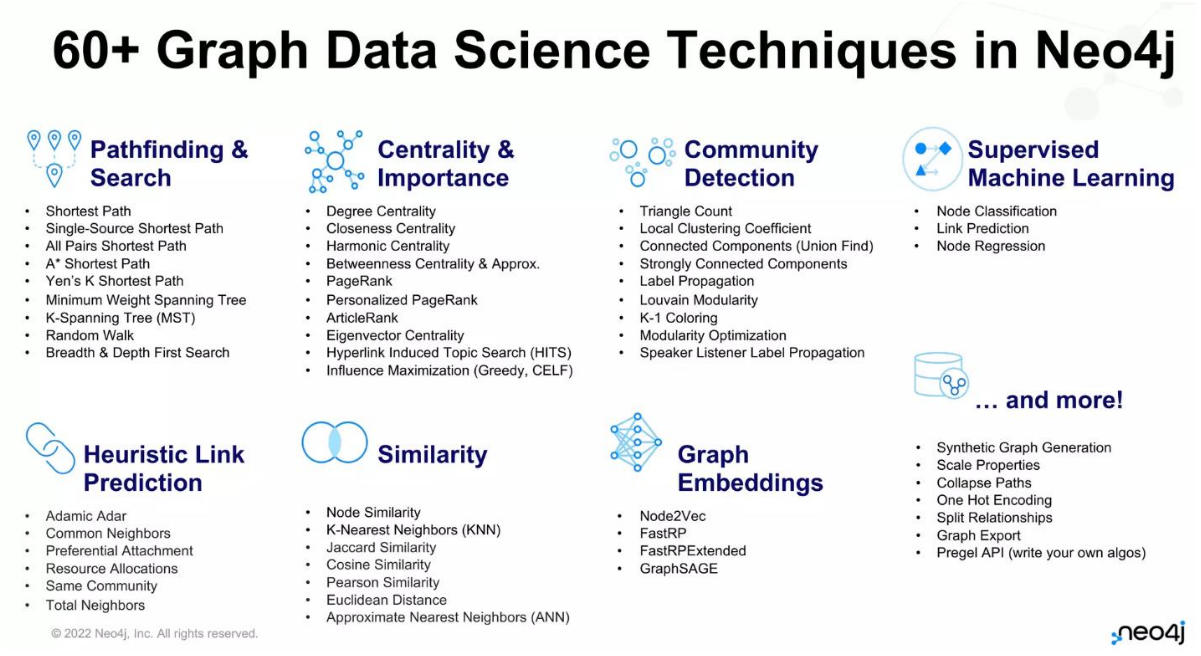
Community Detection
Detects group clustering or partition

Heuristic Link Prediction
Estimates the likelihood of nodes forming a future relationship

Similarity
Evaluates how alike nodes are by neighbours and relationships

Node Embeddings & ML
Compute low-dimensional vector representations of nodes in a graph, and allow you to train supervised machine learning models

© 2022 Neo4j, Inc. All rights reserved.
https://neo4j.com/docs/graph-data-science/current/
neo4j



60+ Graph Data Science Techniques in Neo4j

Pathfinding & Search
- Shortest Path
- Single-Source Shortest Path
- All Pairs Shortest Path
- A* Shortest Path
- Yen's K Shortest Path
- Minimum Weight Spanning Tree
- K-Spanning Tree (MST)
- Random Walk
- Breadth & Depth First Search

Centrality & Importance
- Degree Centrality
- Closeness Centrality
- Harmonic Centrality
- Betweenness Centrality & Approx.
- PageRank
- Personalized PageRank
- ArticleRank
- Eigenvector Centrality
- Hyperlink Induced Topic Search (HITS)
- Influence Maximization (Greedy, CELF)

Community Detection
- Triangle Count
- Local Clustering Coefficient
- Connected Components (Union Find)
- Strongly Connected Components
- Label Propagation
- Louvain Modularity
- K-1 Coloring
- Modularity Optimization
- Speaker Listener Label Propagation

Supervised Machine Learning
- Node Classification
- Link Prediction
- Node Regression

Heuristic Link Prediction
- Adamic Adar
- Common Neighbors
- Preferential Attachment
- Resource Allocations
- Same Community
- Total Neighbors

Similarity
- Node Similarity
- K-Nearest Neighbors (KNN)
- Jaccard Similarity
- Cosine Similarity
- Pearson Similarity
- Euclidean Distance
- Approximate Nearest Neighbors (ANN)

Graph Embeddings
- Node2Vec
- FastRP
- FastRPExtended
- GraphSAGE

... and more!
- Synthetic Graph Generation
- Scale Properties
- Collapse Paths
- One Hot Encoding
- Split Relationships
- Graph Export
- Pregel API (write your own algos)

© 2022 Neo4j, Inc. All rights reserved.
neo4j

We have done several graph algorithms to uncover insights that cannot be done using normal databases or can be but are expensive, require long queries and time consuming. To do some we needed to create graph projections, some used the same but it's better to drop one before creating another so here are the graph projections used:

- Projecting all User and Stream nodes and possible relationships between them (CHATTER, MODERATOR, and VIP)

```
CALL gds.graph.project('twitch',

 ['User', 'Stream'],

 ['CHATTER', 'VIP', 'MODERATOR'])
```

- Projecting all User, Stream and Game nodes and HAS_TEAM, PLAYS relationships between them

```
CALL gds.graph.project('twitch_games',
 ['User', 'Stream','Game'],
 ['HAS_TEAM','PLAYS'])
```

- Projecting all stream nodes with their follower's property to use for the KNN

```
CALL gds.graph.project(
  'myGraph',
  {
    Stream: {
      properties: ['followers']
    }
  },
  '*'
);
```

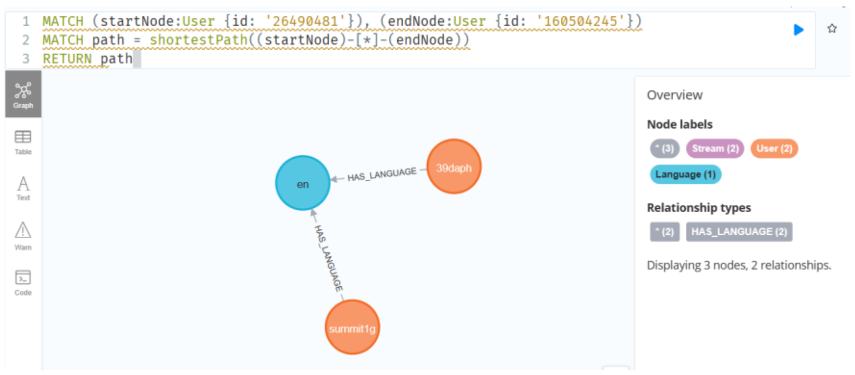
Syntax to drop a graph projection CALL gds.graph.drop("twitch")

Here are the graph algorithms that we used:

1) Pathfinding & Search
   a) Shortest Path
      The shortestPath function is used to find the shortest path between the start and end nodes. The [*] syntax in the pattern specifies that any relationship type and any number of hops can be traversed between the nodes.
      To find the shortest path between two nodes in Neo4j, we can use the built-in shortest path algorithm provided by Neo4j's Cypher query language.
      the general code:

```
MATCH (startNode:Label {id: 'start_node_id'}), (endNode:Label {id: 'end_node_id'})
MATCH path = shortestPath((startNode)-[*]-(endNode))
RETURN path
```

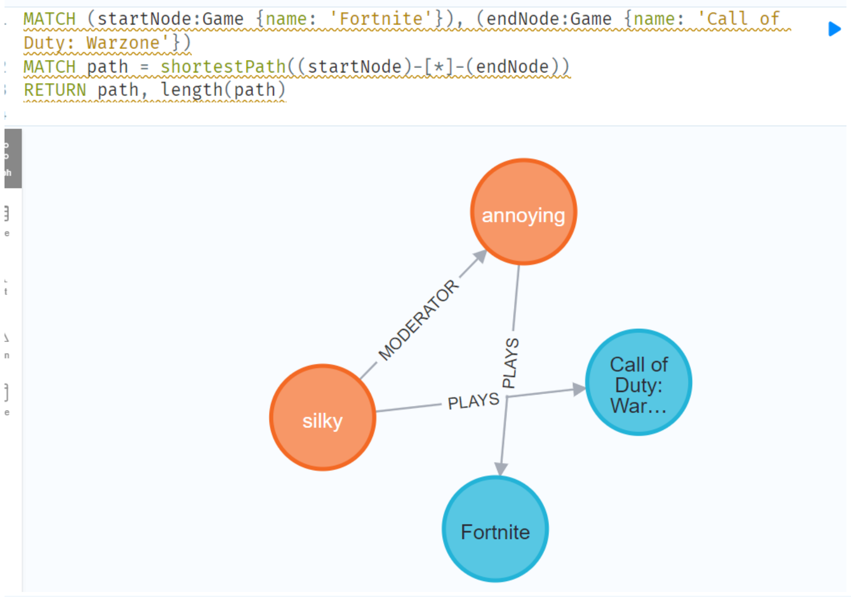
- Shortest between 2 users using their ids



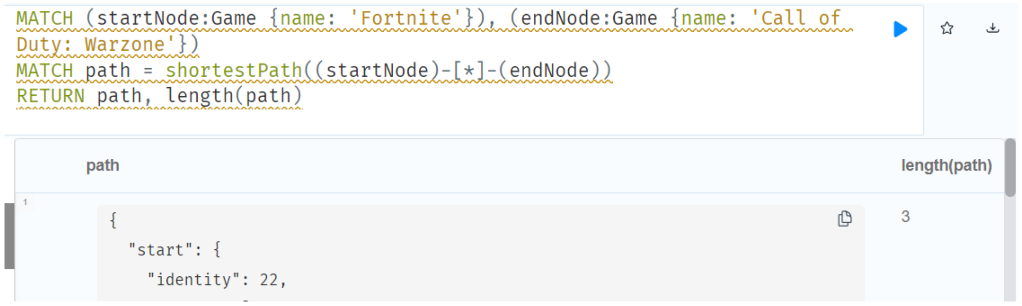
```
MATCH (startNode:User {id: '26490481'}), (endNode:User {id: '160504245'})
MATCH path = shortestPath((startNode)-[*]-(endNode))
RETURN path
```

We can also get the length of the path using the length function,

- Shortest path between 2 games using their names



```
MATCH (startNode:Game {name: 'Fortnite'}), (endNode:Game {name: 'Call of
Duty: Warzone'})
MATCH path = shortestPath((startNode)-[*]-(endNode))
RETURN path, length(path)
```



```
MATCH (startNode:Game {name: 'Fortnite'}), (endNode:Game {name: 'Call of
Duty: Warzone'})
MATCH path = shortestPath((startNode)-[*]-(endNode))
RETURN path, length(path)
```

path	length(path)
{ "start": { "identity": 22,	3

Can't be displayed on the graph

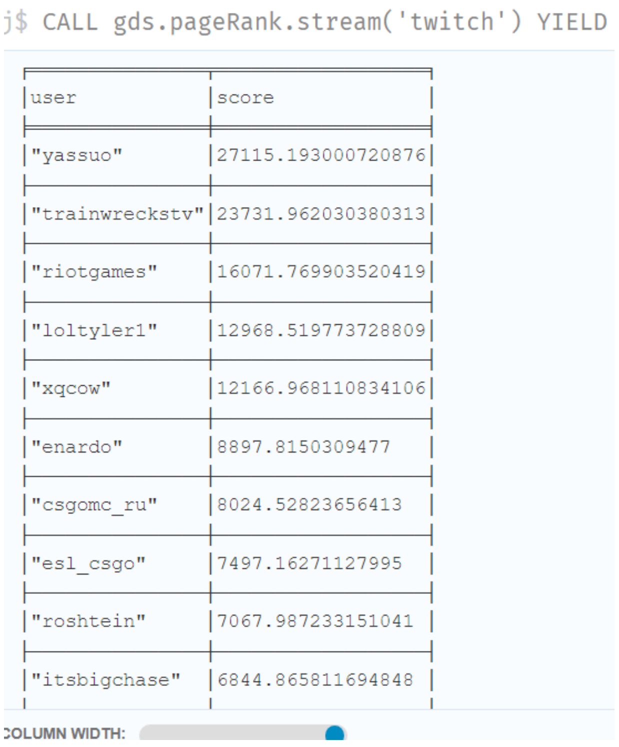
2) Centrality & Importance

a) Page rank:

PageRank is one of the most famous graph algorithms it finds nodes based on their relationships. It is used to calculate node importance by considering the inbound relationships of a node as well as the importance of the nodes linking to it.

i. Page rank on entire graph (projected) twitch
Here's a breakdown of the query:
CALL gds.pageRank.stream('twitch'): This initiates the PageRank algorithm on the graph named 'twitch'.
YIELD nodeId, score: This yields the nodeId and PageRank score for each node in the graph.
WITH nodeId, score: Passes the nodeId and score to the next part of the query.
ORDER BY score DESC LIMIT 10: Orders the results by score in descending order and limits the output to the top 10 nodes.
RETURN gds.util.asNode(nodeId).name as user, score: Returns the name of each user (assuming there is a 'name' property on the nodes) and their PageRank score.

```
CALL gds.pageRank.stream('twitch')
YIELD nodeId, score
WITH nodeId, score
ORDER BY score
DESC LIMIT 10
RETURN gds.util.asNode(nodeId).name as user, score
```

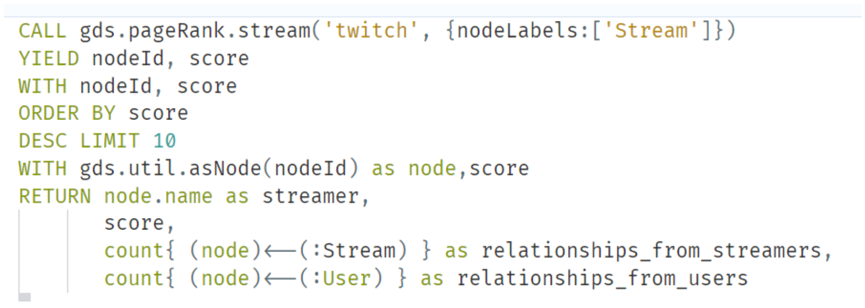


```
j$ CALL gds.pageRank.stream('twitch') YIELD
```

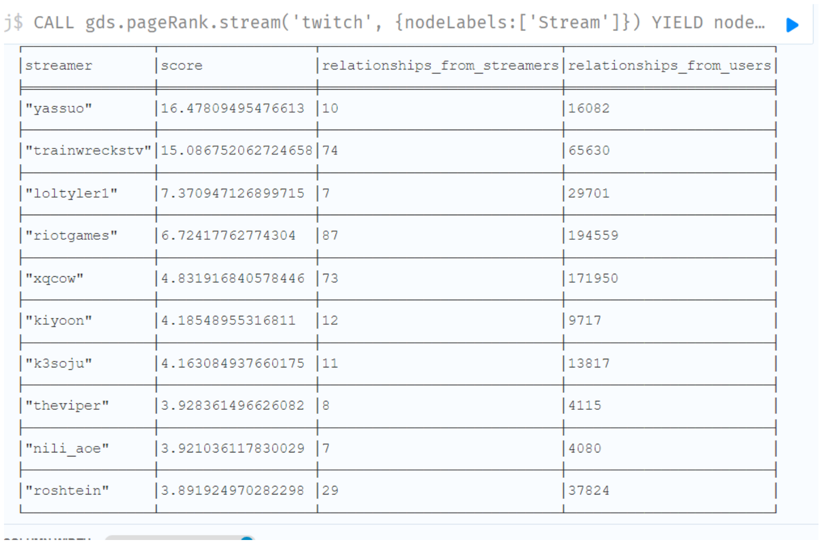
```
╒══════════════╤══════════════════╕
│user          │score             │
╞══════════════╪══════════════════╡
│"yassuo"      │27115.193000720876│
├──────────────┼──────────────────┤
│"trainwreckstv"│23731.962030380313│
├──────────────┼──────────────────┤
│"riotgames"   │16071.769903520419│
├──────────────┼──────────────────┤
│"loltyler1"   │12968.519773728809│
├──────────────┼──────────────────┤
│"xqcow"       │12166.968110834106│
├──────────────┼──────────────────┤
│"enardo"      │8897.8150309477   │
├──────────────┼──────────────────┤
│"csgomc_ru"   │8024.52823656413  │
├──────────────┼──────────────────┤
│"esl_csgo"    │7497.16271127995  │
├──────────────┼──────────────────┤
│"roshtein"    │7067.987233151041 │
├──────────────┼──────────────────┤
│"itsbigchase" │6844.865811694848 │
```

This table shows the top 10 users based on their PageRank scores, with higher scores indicate greater importance in the graph.

ii. Page rank on a certain node
We can also select to do the page rank only for a certain node, example on Stream nodes only and we can also display information related to it :



```
CALL gds.pageRank.stream('twitch', {nodeLabels:['Stream']})
YIELD nodeId, score
WITH nodeId, score
ORDER BY score
DESC LIMIT 10
WITH gds.util.asNode(nodeId) as node,score
RETURN node.name as streamer,
       score,
       count{ (node)<—(:Stream) } as relationships_from_streamers,
       count{ (node)<—(:User) } as relationships_from_users
```

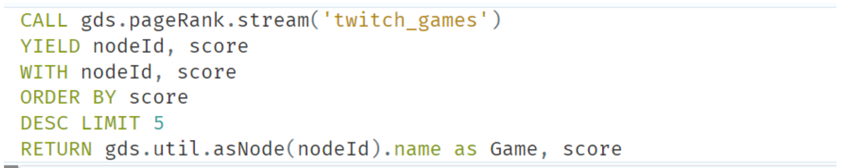


```
j$ CALL gds.pageRank.stream('twitch', {nodeLabels:['Stream']}) YIELD node…
```

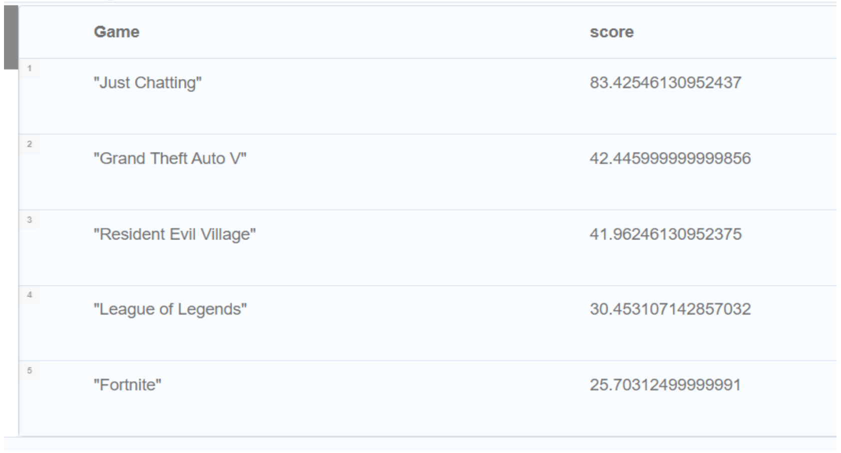
```
│streamer      │score             │relationships_from_streamers│relationships_from_users│
╞══════════════╪══════════════════╪════════════════════════════╪════════════════════════╡
│"yassuo"      │16.47809495476613 │10                          │16082                   │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"trainwreckstv"│15.086752062724658│74                          │65630                   │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"loltyler1"   │7.370947126899715 │7                           │29701                   │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"riotgames"   │6.72417762774304  │87                          │194559                  │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"xqcow"       │4.831916840578446 │73                          │171950                  │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"kiyoon"      │4.18548955316811  │12                          │9717                    │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"k3soju"      │4.163084937660175 │11                          │13817                   │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"theviper"    │3.928361496626082 │8                           │4115                    │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"nili_aoe"    │3.921036117830029 │7                           │4080                    │
├──────────────┼──────────────────┼────────────────────────────┼────────────────────────┤
│"roshtein"    │3.891924970282298 │29                          │37824                   │
```

This query calculated the top 10 PageRank of Twitch streams, ordered the result by score in descending order, and we also displayed their counts of relationships from both Stream and User nodes.

iii. Page rank on entire twitch_games graph



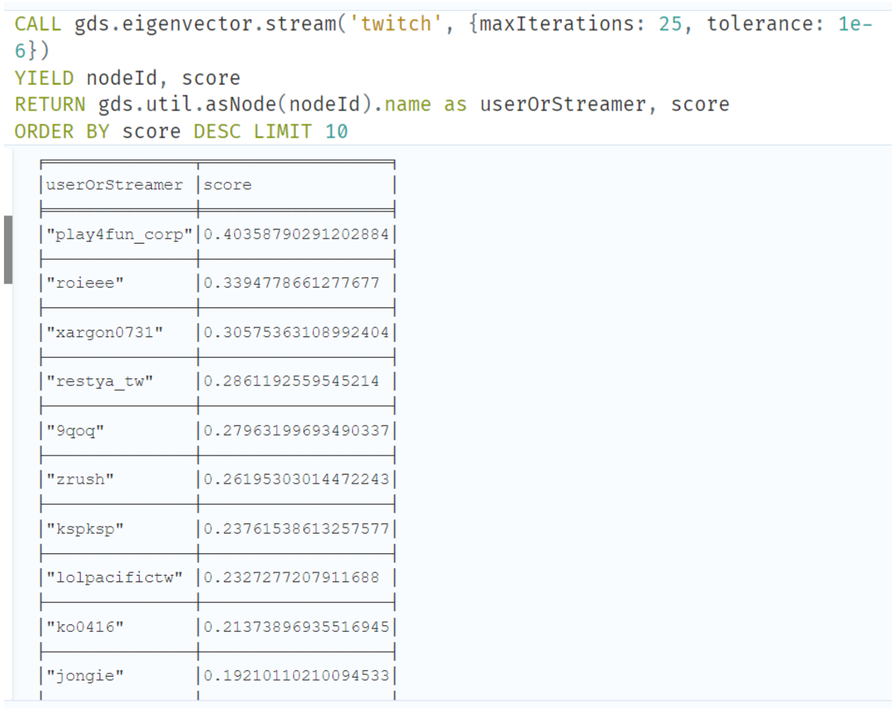
```
CALL gds.pageRank.stream('twitch_games')
YIELD nodeId, score
WITH nodeId, score
ORDER BY score
DESC LIMIT 5
RETURN gds.util.asNode(nodeId).name as Game, score
```



	Game	score
1	"Just Chatting"	83.42546130952437
2	"Grand Theft Auto V"	42.445999999999856
3	"Resident Evil Village"	41.96246130952375
4	"League of Legends"	30.453107142857032
5	"Fortnite"	25.70312499999991

b) Eigenvector Centrality

Eigenvector Centrality is an algorithm that measures the transitive influence of nodes. Relationships originating from high-scoring nodes contribute more to the score of a node than connections from low-scoring nodes. A high eigenvector score means that a node is connected to many nodes who themselves have high scores.



```
CALL gds.eigenvector.stream('twitch', {maxIterations: 25, tolerance: 1e-
6})
YIELD nodeId, score
RETURN gds.util.asNode(nodeId).name as userOrStreamer, score
ORDER BY score DESC LIMIT 10
```

```
╒═══════════════╤═══════════════════╕
│userOrStreamer │score              │
╞═══════════════╪═══════════════════╡
│"play4fun_corp"│0.40358790291202884│
├───────────────┼───────────────────┤
│"roieee"       │0.3394778661277677 │
├───────────────┼───────────────────┤
│"xargon0731"   │0.30575363108992404│
├───────────────┼───────────────────┤
│"restya_tw"    │0.2861192559545214 │
├───────────────┼───────────────────┤
│"9qoq"         │0.27963199693490337│
├───────────────┼───────────────────┤
│"zrush"        │0.26195303014472243│
├───────────────┼───────────────────┤
│"kspksp"       │0.23761538613257577│
├───────────────┼───────────────────┤
│"lolpacifictw" │0.2327277207911688 │
├───────────────┼───────────────────┤
│"ko0416"       │0.21373896935516945│
├───────────────┼───────────────────┤
│"jongie"       │0.19210110210094533│
```

Interpretation:

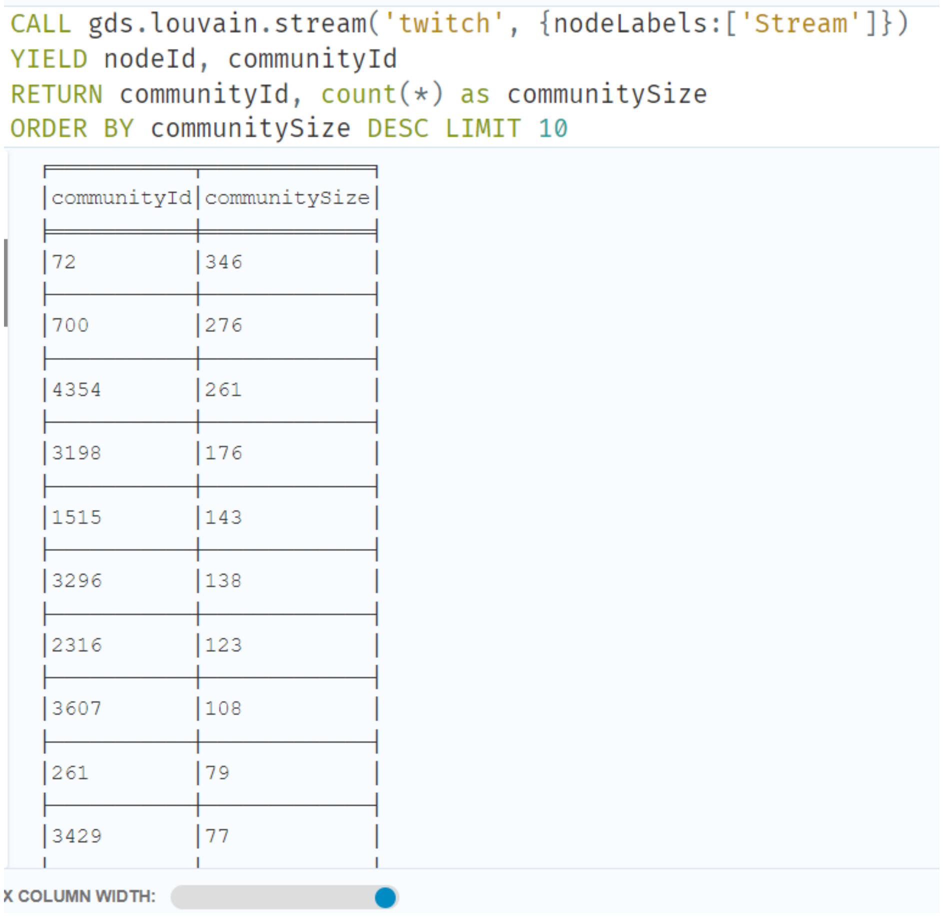
- **Top Nodes:** The nodes listed with the highest Eigenvector Centrality scores (e.g., "play4fun_corp," "roieee," etc.) are the ones that are deemed more central in the network.

3) Community Detection

a) Louvain Modularity

The Louvain method is an algorithm to detect communities in large networks. It maximizes a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities. This means evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network.

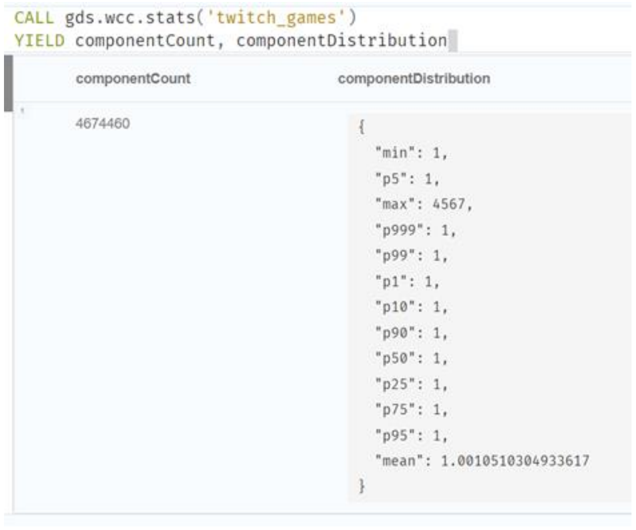
Top 10 Stream communities in the twitch graph projection:



```
CALL gds.louvain.stream('twitch', {nodeLabels:['Stream']})
YIELD nodeId, communityId
RETURN communityId, count(*) as communitySize
ORDER BY communitySize DESC LIMIT 10
```

```
╒═══════════╤═════════════╕
│communityId│communitySize│
╞═══════════╪═════════════╡
│72         │346          │
├───────────┼─────────────┤
│700        │276          │
├───────────┼─────────────┤
│4354       │261          │
├───────────┼─────────────┤
│3198       │176          │
├───────────┼─────────────┤
│1515       │143          │
├───────────┼─────────────┤
│3296       │138          │
├───────────┼─────────────┤
│2316       │123          │
├───────────┼─────────────┤
│3607       │108          │
├───────────┼─────────────┤
│261        │79           │
├───────────┼─────────────┤
│3429       │77           │
```

b) Weakly Connected Components (WCC)
The Weakly Connected Components (WCC) algorithm finds sets of connected nodes in directed and undirected graphs. Two nodes are connected, if there exists a path between them.

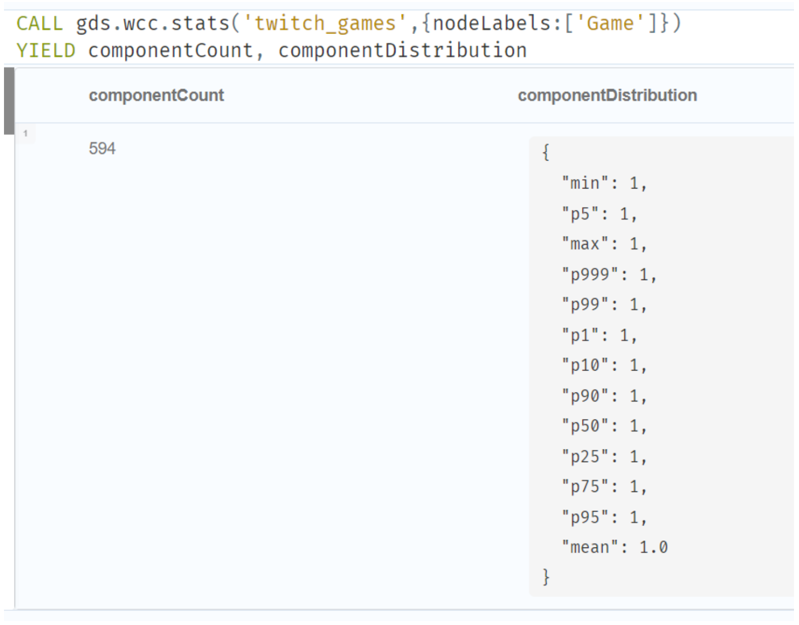


```
CALL gds.wcc.stats('twitch_games')
YIELD componentCount, componentDistribution
```

componentCount	componentDistribution
4674460	{ "min": 1, "p5": 1, "max": 4567, "p999": 1, "p99": 1, "p1": 1, "p10": 1, "p90": 1, "p50": 1, "p25": 1, "p75": 1, "p95": 1, "mean": 1.0010510304933617 }

Total number of components: 4,674,460. The output suggests that the majority of the components have a count of 1, indicated by the percentiles and the mean. However, there is at least one component with a count of 4567 (the maximum). The distribution seems to be heavily skewed towards lower counts, given

We can also perform the WCC on a specific node, for example Game node:

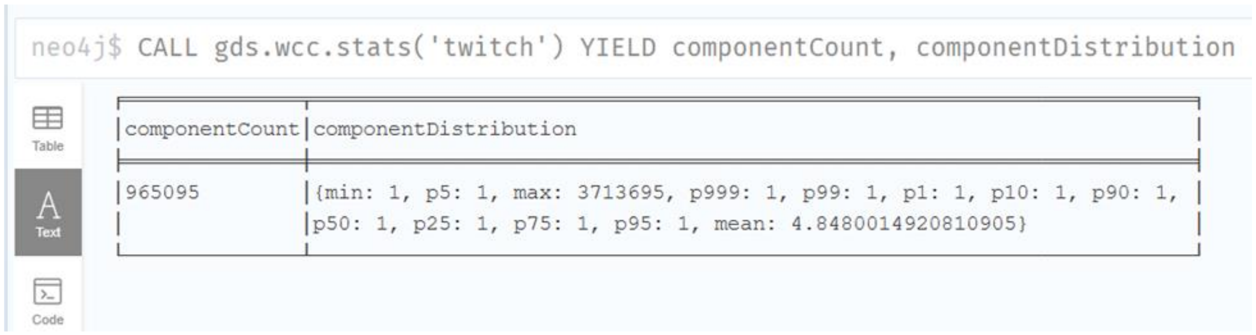


```
CALL gds.wcc.stats('twitch_games',{nodeLabels:['Game']})
YIELD componentCount, componentDistribution
```

```
componentCount    componentDistribution
594               {
                    "min": 1,
                    "p5": 1,
                    "max": 1,
                    "p999": 1,
                    "p99": 1,
                    "p1": 1,
                    "p10": 1,
                    "p90": 1,
                    "p50": 1,
                    "p25": 1,
                    "p75": 1,
                    "p95": 1,
                    "mean": 1.0
                  }
```

This indicates that all 594 components have a count of 1. The distribution is uniform, and there are no variations in the count of components; each component occurs exactly once. The minimum, maximum, percentiles, and mean all confirm that every component has the same count of 1. The result suggests that each node in the graph forms its own connected component. In the context of weakly connected components, this means that there are no directed paths between nodes (isolated); each node is disconnected from every other node in terms of incoming and outgoing relationships.

Next we also performed the WCC on the twitch projection:

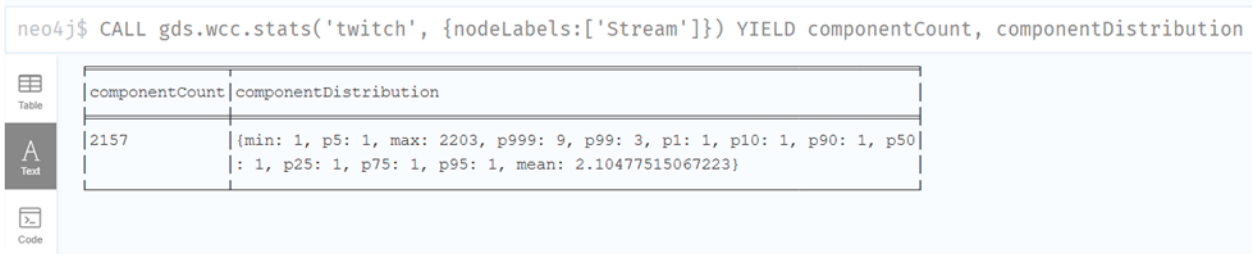


```
neo4j$ CALL gds.wcc.stats('twitch') YIELD componentCount, componentDistribution

|componentCount|componentDistribution                                               |
|965095        |{min: 1, p5: 1, max: 3713695, p999: 1, p99: 1, p1: 1, p10: 1, p90: 1, |
|              |p50: 1, p25: 1, p75: 1, p95: 1, mean: 4.8480014920810905}           |
```

The output suggests that the weakly connected components vary in size, with the majority being small (below the mean of 4.85) and some larger components, as indicated by the maximum of 3,713,695.

WCC on Stream node only:



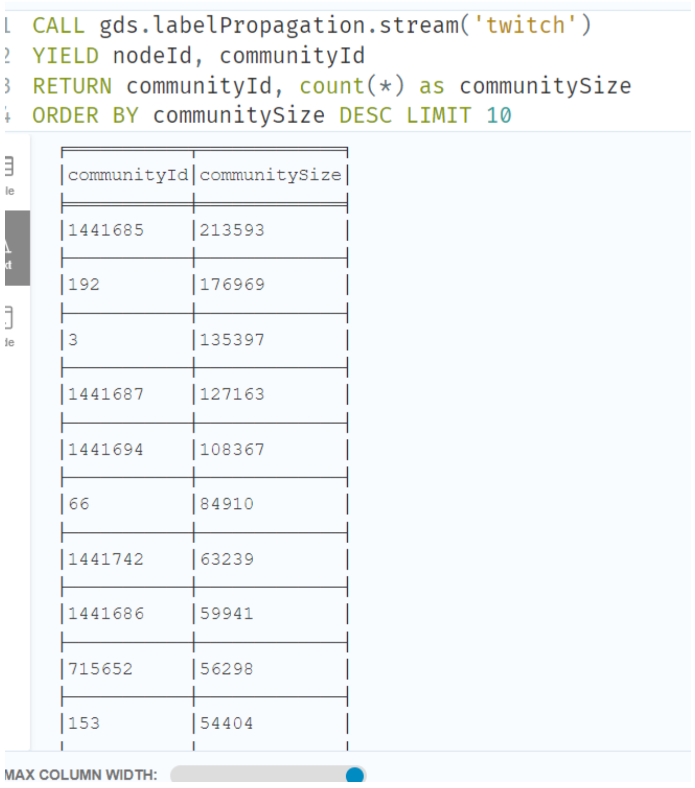
```
neo4j$ CALL gds.wcc.stats('twitch', {nodeLabels:['Stream']}) YIELD componentCount, componentDistribution

|componentCount|componentDistribution                                                  |
|2157          |{min: 1, p5: 1, max: 2203, p999: 9, p99: 3, p1: 1, p10: 1, p90: 1, p50|
|              |: 1, p25: 1, p75: 1, p95: 1, mean: 2.10477515067223}                   |
```

In this case, the weakly connected components are relatively small, with the majority having a size of 1 (indicated by the mean of 2.10). The largest component has a size of 2,203, and the 99.9th percentile (p999) indicates that only 0.1% of the components have a size greater than or equal to 9.

c) Label Propagation Algorithm (LPA):

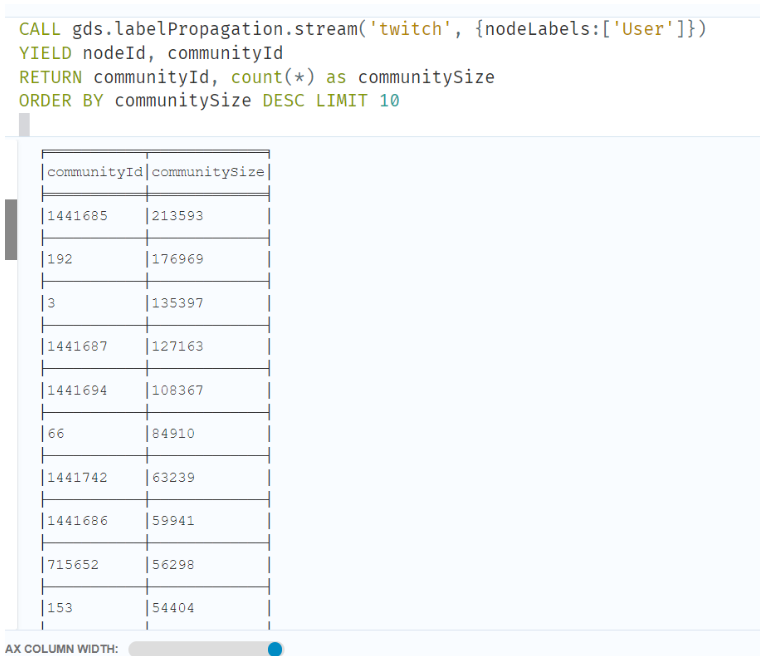
The Label Propagation algorithm (LPA) is a fast algorithm for finding communities in a graph. It detects these communities using network structure alone as its guide, and doesn't require a pre-defined objective function or prior information about the communities. LPA works by propagating labels throughout the network and forming communities based on this process of label propagation.



```
CALL gds.labelPropagation.stream('twitch')
YIELD nodeId, communityId
RETURN communityId, count(*) as communitySize
ORDER BY communitySize DESC LIMIT 10
```

communityId	communitySize
1441685	213593
192	176969
3	135397
1441687	127163
1441694	108367
66	84910
1441742	63239
1441686	59941
715652	56298
153	54404

LPA on twitch projection



```
CALL gds.labelPropagation.stream('twitch', {nodeLabels:['User']})
YIELD nodeId, communityId
RETURN communityId, count(*) as communitySize
ORDER BY communitySize DESC LIMIT 10
```

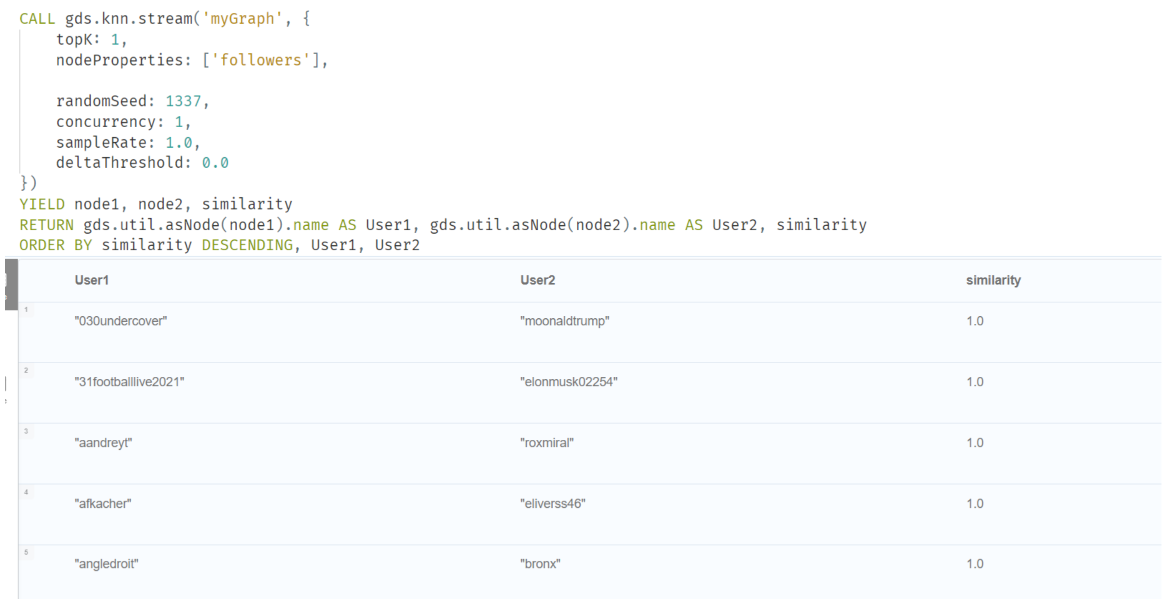
```
╒═══════════╤═════════════╕
│communityId│communitySize│
╞═══════════╪═════════════╡
│1441685    │213593       │
├───────────┼─────────────┤
│192        │176969       │
├───────────┼─────────────┤
│3          │135397       │
├───────────┼─────────────┤
│1441687    │127163       │
├───────────┼─────────────┤
│1441694    │108367       │
├───────────┼─────────────┤
│66         │84910        │
├───────────┼─────────────┤
│1441742    │63239        │
├───────────┼─────────────┤
│1441686    │59941        │
├───────────┼─────────────┤
│715652     │56298        │
├───────────┼─────────────┤
│153        │54404        │
```

LPA to identify user community

4) Similarity

a) K-Nearest Neighbors

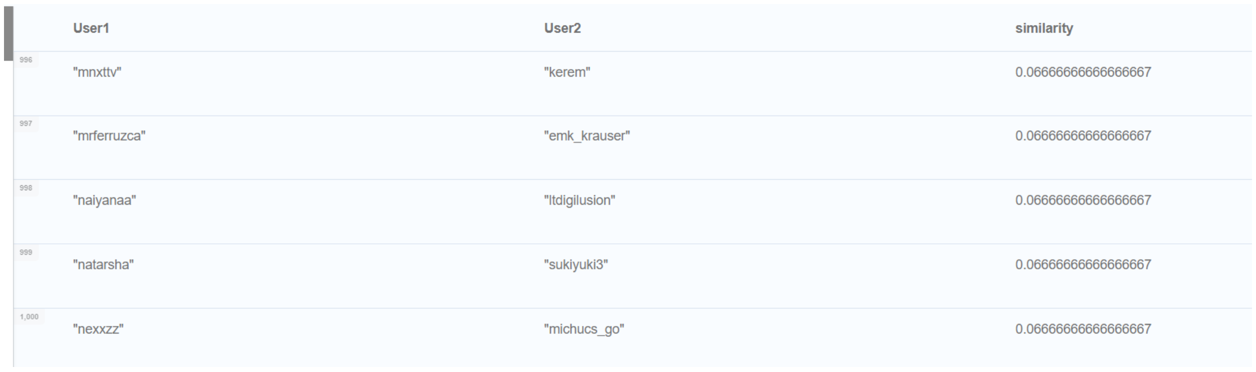
The K-Nearest Neighbors algorithm computes a distance value for all node pairs in the graph and creates new relationships between each node and its k nearest neighbors. The distance is calculated based on node properties.



```
CALL gds.knn.stream('myGraph', {
    topK: 1,
    nodeProperties: ['followers'],

    randomSeed: 1337,
    concurrency: 1,
    sampleRate: 1.0,
    deltaThreshold: 0.0
})
YIELD node1, node2, similarity
RETURN gds.util.asNode(node1).name AS User1, gds.util.asNode(node2).name AS User2, similarity
ORDER BY similarity DESCENDING, User1, User2
```

User1	User2	similarity
"030undercover"	"moonaldtrump"	1.0
"31footballlive2021"	"elonmusk02254"	1.0
"aandreyt"	"roxmiral"	1.0
"afkacher"	"eliverss46"	1.0
"angledroit"	"bronx"	1.0



	User1	User2	similarity
996	"mnxttv"	"kerem"	0.06666666666666667
997	"mrferruzca"	"emk_krauser"	0.06666666666666667
998	"naiyanaa"	"ltdigilusion"	0.06666666666666667
999	"natarsha"	"sukiyuki3"	0.06666666666666667
1,000	"nexxzz"	"michucs_go"	0.06666666666666667

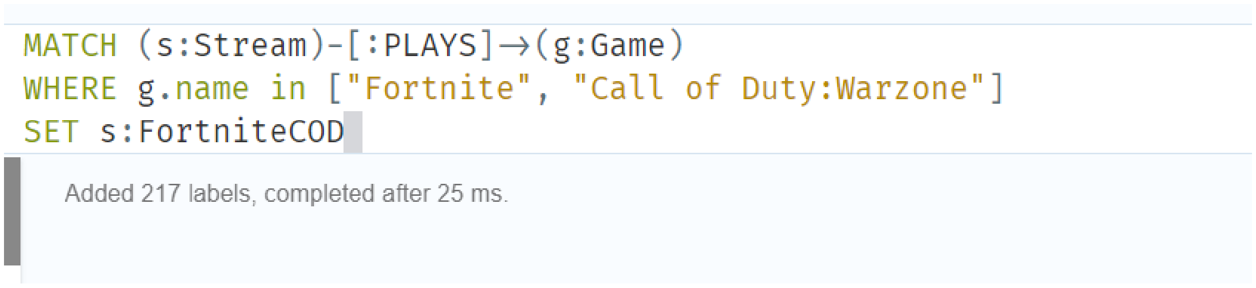
The output is a table of 1000 rows ordered by descending order starting with users with high similarity 1 to the lowest 0.066.

b) Node Similarity Algorithm

The Node Similarity algorithm uses the Jaccard similarity coefficient to compare how similar a pair of nodes are. We will assume that if two streamers share at least 5% of the audience.

Before getting to the algorithm to examine the shared audience between streamers who play Fortnite or Call of Duty:Warzone on stream, we need to perform 3 steps:

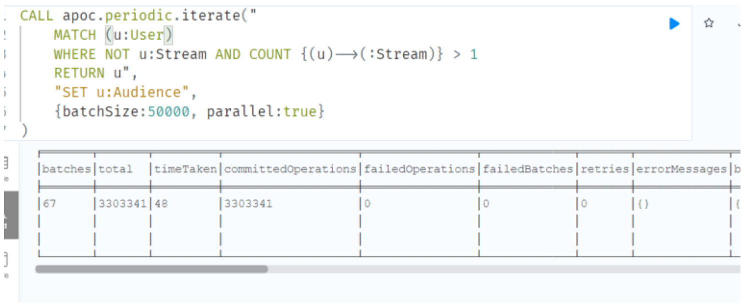
1) To simplify queries, we will tag the mentioned streamers with a secondary label (FortniteCOD).



```
MATCH (s:Stream)-[:PLAYS]→(g:Game)
WHERE g.name in ["Fortnite", "Call of Duty:Warzone"]
SET s:FortniteCOD
```

```
Added 217 labels, completed after 25 ms.
```

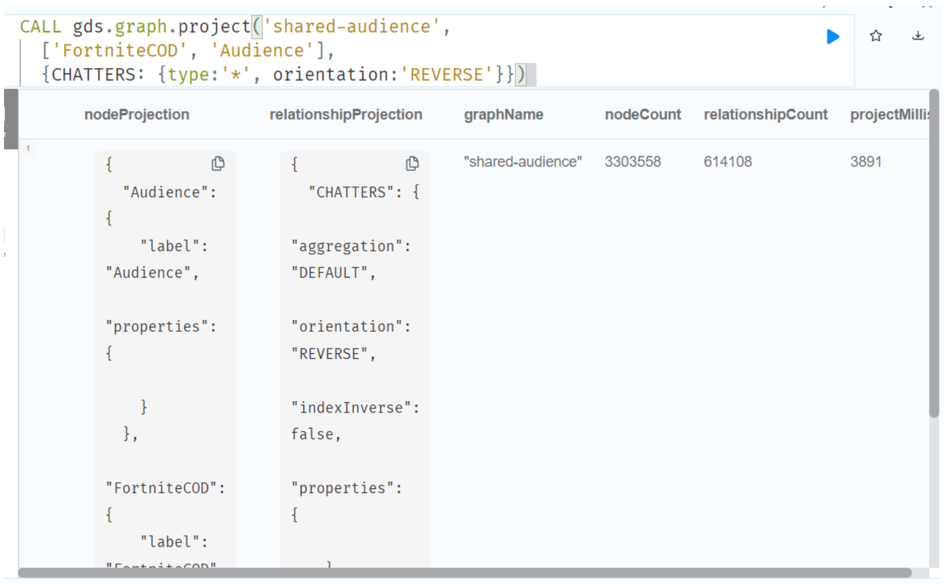
2) Next, using the apoc.periodic.iterate procedure to batch update users who have an out-degree higher than 1. With this step, we filtered regular users who have chatted in more than a single stream.



```
CALL apoc.periodic.iterate("
    MATCH (u:User)
    WHERE NOT u:Stream AND COUNT {(u)⟶(:Stream)} > 1
    RETURN u",
    "SET u:Audience",
    {batchSize:50000, parallel:true}
)
```

```
|batches|total  |timeTaken|committedOperations|failedOperations|failedBatches|retries|errorMessages|b
|67     |3303341|48       |3303341            |0               |0            |0      |{}           |{
```

3) Create a graph projection



```
CALL gds.graph.project('shared-audience',
  ['FortniteCOD', 'Audience'],
  {CHATTERS: {type:'*', orientation:'REVERSE'}})
```

nodeProjection	relationshipProjection	graphName	nodeCount	relationshipCount	projectMilli
{ "Audience": { "label": "Audience", "properties": { } }, "FortniteCOD": { "label":	{ "CHATTERS": { "aggregation": "DEFAULT", "orientation": "REVERSE", "indexInverse": false, "properties": {	"shared-audience"	3303558	614108	3891

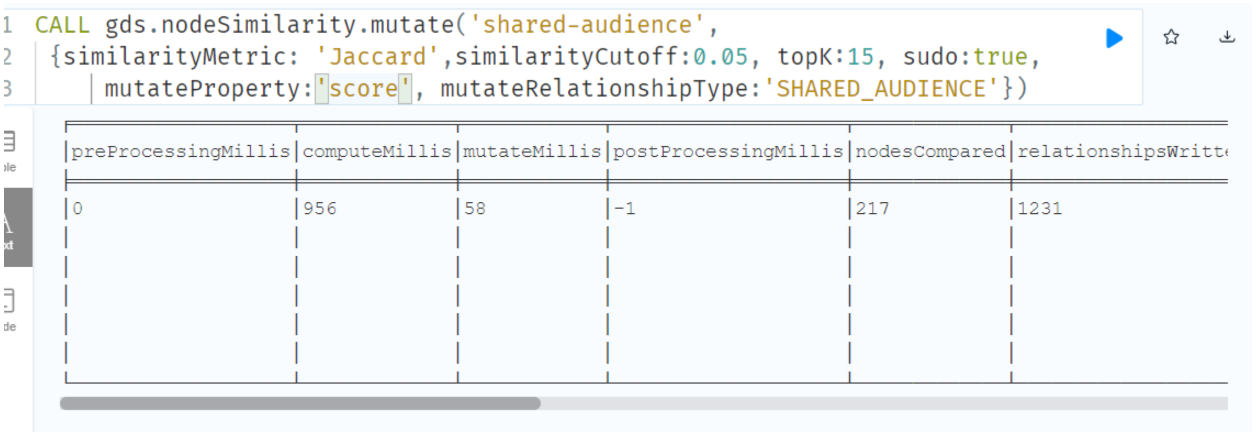
The orientation: 'REVERSE' indicates that the direction of the 'CHATTERS' relationship should be reversed.

Now the node similarity algorithm:

It seems like you're using a query for the Neo4j Graph Data Science library to compute node similarity using Jaccard similarity and then mutating the graph by creating relationships between similar nodes. Here's a breakdown of the parameters you've provided:

- **gds.nodeSimilarity.mutate**: This is the procedure for computing node similarity and mutating the graph.
- **'shared-audience'**: The graph name or graph ID on which the node similarity computation and mutation will be performed.
- **{similarityMetric: 'Jaccard', similarityCutoff: 0.05, topK: 15, sudo: true, mutateProperty: 'score', mutateRelationshipType: 'SHARED_AUDIENCE'}**: These are the configuration options for the node similarity computation and graph mutation.
    - **similarityMetric: 'Jaccard'**: Specifies that Jaccard similarity will be used for computing node similarity.
    - **similarityCutoff: 0.05**: Sets a threshold for similarity. Nodes with a similarity score below this threshold won't be considered similar.
    - **topK: 15**: Specifies that only the top 15 most similar nodes will be considered for each node.
    - **sudo: true**: Indicates that the procedure should be executed with administrative privileges.

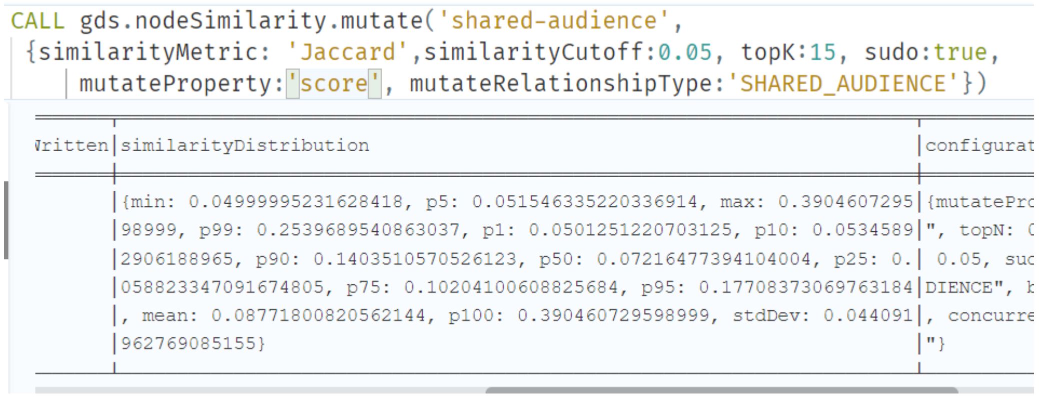
- **mutateProperty: 'score'**: Specifies the property name that will be created on the relationships to store the similarity score.
- **mutateRelationshipType: 'SHARED_AUDIENCE'**: Specifies the type of relationship that will be created between similar nodes. In this case, it's a relationship type named 'SHARED_AUDIENCE'.



```
CALL gds.nodeSimilarity.mutate('shared-audience',
 {similarityMetric: 'Jaccard',similarityCutoff:0.05, topK:15, sudo:true,
    mutateProperty:'score', mutateRelationshipType:'SHARED_AUDIENCE'})
```

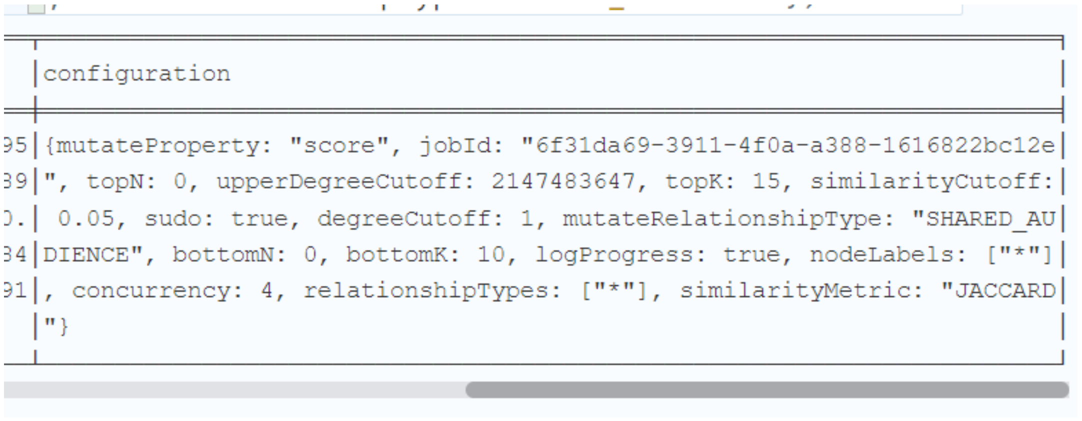
```
|preProcessingMillis|computeMillis|mutateMillis|postProcessingMillis|nodesCompared|relationshipsWritt
|0                  |956          |58          |-1                  |217          |1231
```

- **Graph Statistics:**
  - **nodesCompared**: 217 nodes were compared for similarity.
  - **relationshipsWritten**: 1231 relationships were written to the graph.



```
CALL gds.nodeSimilarity.mutate('shared-audience',
 {similarityMetric: 'Jaccard',similarityCutoff:0.05, topK:15, sudo:true,
    mutateProperty:'score', mutateRelationshipType:'SHARED_AUDIENCE'})
```

```
Written|similarityDistribution                                                    |configurat
       |{min: 0.04999995231628418, p5: 0.051546335220336914, max: 0.3904607295|{mutatePro
       |98999, p99: 0.2539689540863037, p1: 0.0501251220703125, p10: 0.0534589|", topN: 0
       |2906188965, p90: 0.1403510570526123, p50: 0.07216477394104004, p25: 0.| 0.05, suc
       |058823347091674805, p75: 0.10204100608825684, p95: 0.17708373069763184|DIENCE", b
       |, mean: 0.08771800820562144, p100: 0.390460729598999, stdDev: 0.044091|, concurre
       |962769085155}                                                         |"}
```

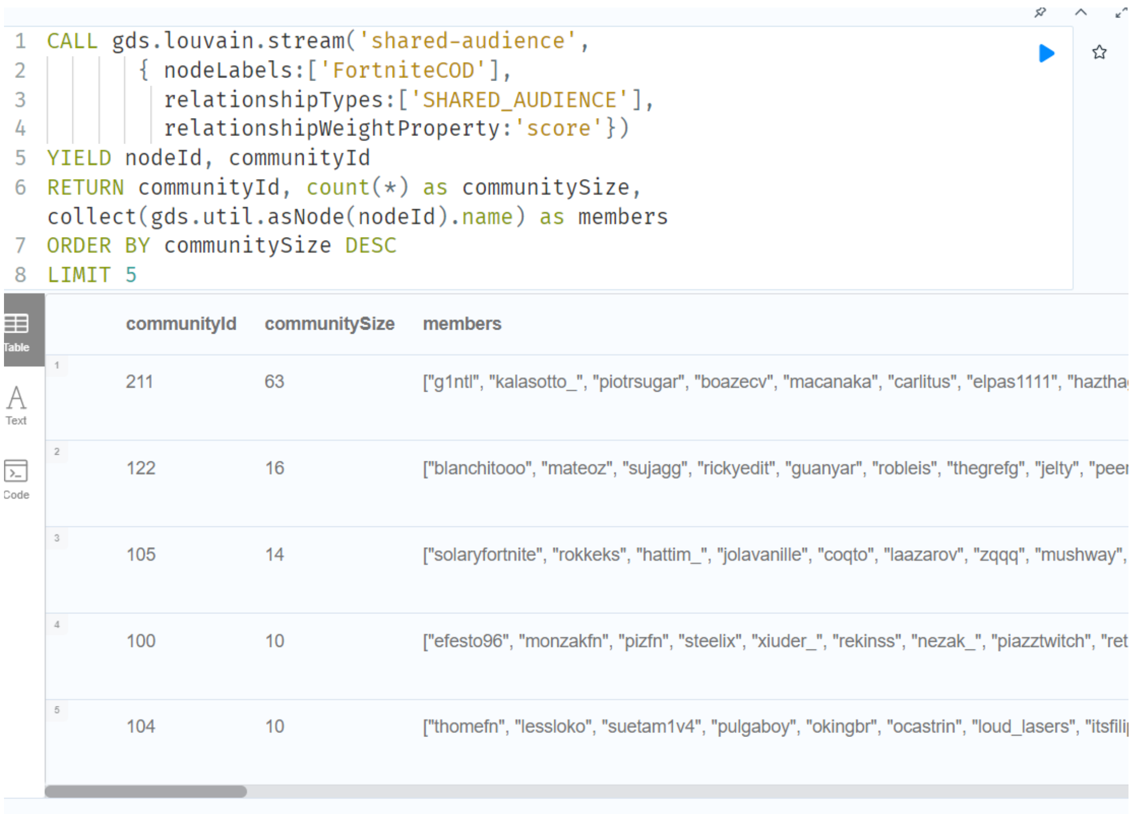


```
  |configuration                                                         |
95|{mutateProperty: "score", jobId: "6f31da69-3911-4f0a-a388-1616822bc12e|
39|", topN: 0, upperDegreeCutoff: 2147483647, topK: 15, similarityCutoff:|
0.| 0.05, sudo: true, degreeCutoff: 1, mutateRelationshipType: "SHARED_AU|
34|DIENCE", bottomN: 0, bottomK: 10, logProgress: true, nodeLabels: ["*"]|
91|, concurrency: 4, relationshipTypes: ["*"], similarityMetric: "JACCARD|
  |"}                                                                    |
```

Here are some potential insights the provided similarity distribution:

1. **Range of Similarity Scores:**
    - The minimum similarity score is 0.04999995231628418, indicating that some pairs of nodes have very low similarity.
    - The maximum similarity score is 0.390460729598999, suggesting that there are pairs of nodes with relatively high similarity.
2. **Central Tendency:**
    - The mean similarity score is 0.08771800820562144, providing an average measure of similarity across all pairs of nodes.
    - The median (p50) similarity score is 0.07216477394104004, giving the middle point of the distribution.
3. **Percentiles:**
    - Percentile values (p25, p75, p95, etc.) offer a breakdown of where specific percentages of the data fall. For example, p25 is 0.058823347091674805, indicating that 25% of node pairs have a similarity score at or below this value.
4. **Variability:**
    - The standard deviation (stdDev) is 0.044091962769085155, representing the degree of variability or dispersion in the similarity scores.
5. **Threshold for Similarity:**
    - The similarityCutoff is set at 0.05, meaning that nodes with a similarity score below this threshold won't be considered similar.
6. **Top Similar Nodes:**
    - The topK parameter is set to 15, indicating that only the top 15 most similar nodes will be considered for each node.

Finally, we performed the Louvain algorithm to detect communities and to see some members.


```
CALL gds.louvain.stream('shared-audience',
        { nodeLabels:['FortniteCOD'],
          relationshipTypes:['SHARED_AUDIENCE'],
          relationshipWeightProperty:'score'})
YIELD nodeId, communityId
RETURN communityId, count(*) as communitySize,
collect(gds.util.asNode(nodeId).name) as members
ORDER BY communitySize DESC
LIMIT 5
```

	communityId	communitySize	members
1	211	63	["g1ntl", "kalasotto_", "piotrsugar", "boazecv", "macanaka", "carlitus", "elpas1111", "haztha
2	122	16	["blanchitooo", "mateoz", "sujagg", "rickyedit", "guanyar", "robleis", "thegrefg", "jelty", "peer
3	105	14	["solaryfortnite", "rokkeks", "hattim_", "jolavanille", "coqto", "laazarov", "zqqq", "mushway",
4	100	10	["efesto96", "monzakfn", "pizfn", "steelix", "xiuder_", "rekinss", "nezak_", "piazztwitch", "ret
5	104	10	["thomefn", "lessloko", "suetam1v4", "pulgaboy", "okingbr", "ocastrin", "loud_lasers", "itsfili

The GDS library provide useful insights that are impossible to discover using other types of databases. It is a powerful toolset for leveraging graph algorithms and techniques for extracting insights from graph-structured data. Its ability to navigate complex relationships and reveal hidden patterns makes it particularly valuable in scenarios where traditional databases may fall short in capturing the richness of interconnected information.

4. Sources:

https://graphacademy.neo4j.com/categories/

https://neo4j.com/docs/graph-data-science/current/algorithms/

https://www.slideshare.net/neo4j/workshop-neo4j-graph-data-science

https://www.slideshare.net/neo4j/neo4j-graph-data-science-training-june-9-10-slides-6-graph-algorithms

https://www.youtube.com/watch?v=8jNPelugC2s

OReilly Graph Databases book.