Modeling Chat Data using a Graph Data Model

Using a Graph Data Model to illustrate the chatting interaction among users with Chat Data. A user in a team can create a chat session and then create chat (i.e. chat item) in the chat session. Otherwise, a user could be mentioned by a chat item, and a chat item can response to another chat item, which represent the communication among the users in the same team. Moreover, a user can also join in an existed team chat session or leave it.

Creation of the Graph Database for Chats

CSV Files

File Name	Description	Fields
chat_create_team_chat.csv	userid	the user id assigned to the user
	teamid	the id of the team
	TeamChatSessionID	a unique id for the chat session
	timestamp	a timestamp denoting when the chat session created
chat_item_team_chat.csv	userid	the user id assigned to the user
	teamchatsessionid	a unique id for the chat session
	chatitemid	a unique id for the chat item
	timestamp	a timestamp denoting when the chat item created
chat_join_team_chat.csv	userid	the user id assigned to the user
	TeamChatSessionID	a unique id for the chat session
	timestamp	a timestamp denoting when the user join in a chat session
chat_leave_team_chat.csv	userid	the user id assigned to the user
	teamchatsessionid	a unique id for the chat session
	timestamp	a timestamp denoting when the user leave a chat session
chat_mention_team_chat.csv	ChatItemId	the id of the ChatItem
	userid	the user id assigned to the user
	timeStamp	a timestamp denoting when the user mentioned by a chat item
chat_respond_team_chat.csv	chatid1	the id of the chat post 1
	chatid2	the id of the chat post 2
	timestamp	a timestamp denoting when the chat post 1 responds to the chat post 2

Loading Process

Using Cypher Query Language to load the CSV data into neo4j, each row of script is parsed for refine the nodes, the edges and its timestamp. Let's consult the following script as an example:

```
LOAD CSV FROM "file:///chat-data/chat_item_team_chat.csv" AS row
MERGE (u:User {id: toInt(row[0])})
MERGE (c:TeamChatSession {id: toInt(row[1])})
MERGE (i:ChatItem {id: toInt(row[2])})
MERGE (u)-[:CreateChat{timeStamp: row[3]}]->(i)
MERGE (i)-[:PartOf{timeStamp: row[3]}]->(c)
```

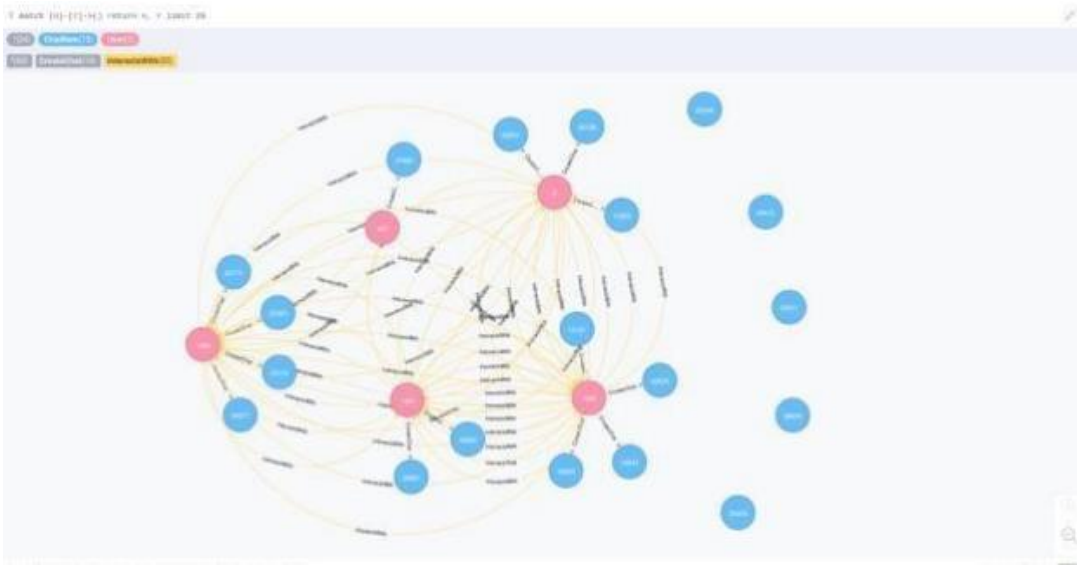
The first line gives the path of the file, this command reads the chat_item_team_chat.csv at a time and create user nodes. The 0th column value is converted to an integer and is used to populate the id attribute. Similarly the other nodes are created.

Line 5, `MERGE (u)-[:CreateChat{timeStamp: row[3]}]->(i)` creates an edge labeled "`CreateChat`" between the User node `u` and the ChatItem node `i`. This edge has a property called `timeStamp`. This property is filled by the content of column 3 of the same row.

Line 6, `MERGE (i)-[:PartOf{timeStamp: row[3]}]->(c)` creates an edge labeled "`PartOf`" between the ChatItem node `i` and the TeamChatSession node `c`. This edge has a property called `timeStamp`. This property is filled by the content of column 3 of the same row.

4.2.3 A Screenshot of Some Part of the Graph

The graphs must include clearly visible examples of most node and edge types. Below are two acceptable examples. The first example is a rendered in the default Neo4j distribution, the second has had some nodes moved to expose the edges more clearly. Both include examples of most node and edge types.



The longest conversation chain and its participants

• finding the longest conversation chain

The longest conversation chain is traced via ChatItem nodes which are connected by ResponseTo edges, order the length and find the longest one. Running the following query, we get the longest conversation chain has path length of 9, i.e. the longest conversation has 10 chats.

- Query

```
match p = (i1)-[:ResponseTo*]->(i2) return length(p)
order by length(p) desc limit 1
```

- Screenshot

• how many unique users were part of this chain? Since we've already known the longest conversation chain has path length of 9, next we will find all the distinct users which are part of this path. After running the following query, 5 unique users are extracted.

- Query

```
match p = (i1)-[:ResponseTo*]-
>(i2) where length(p) = 9 with p
match (u)-[:CreateChat]->(i) where
i in nodes(p) return
count(distinct u)
```

- Screenshot

The relationship between top 10 chattiest users and top 10 chattiest teams

Describe your steps from Question 2. In the process, create the following two tables. You only need to include the top 3 for each table. Identify and report whether any of the chattiest users were part of any of the chattiest teams.

Chattiest Users

We firstly match the CreateChat edge from User node to ChatItem node, then return the ChatItem amount per user, and order by the amount in descending order.

- Query

```
match (u)-[:CreateChat*]->(i) return u.id,
count(i)
order by count(i) desc limit 10
```

- Screenshot

Users	Number of Chats
394	115
2067	111
209	109
1087[1]	109

Chattiest Teams

We firstly `match` the `PartOf` edge from ChatItem node to TeamChatSession node, `match` the OwnedBy edge from TeamChatSession node to Team node, then return the TeamChatSession amount per team, and order by the amount in descending order.

- Query

```
match (i)-[:PartOf*]->(c)-[:OwnedBy*]-
>(t) return t.id, count(c) order by
count(c) desc limit 10
```

- Screenshot

Teams	Number of Chats
82	1324
185	1036
112	957

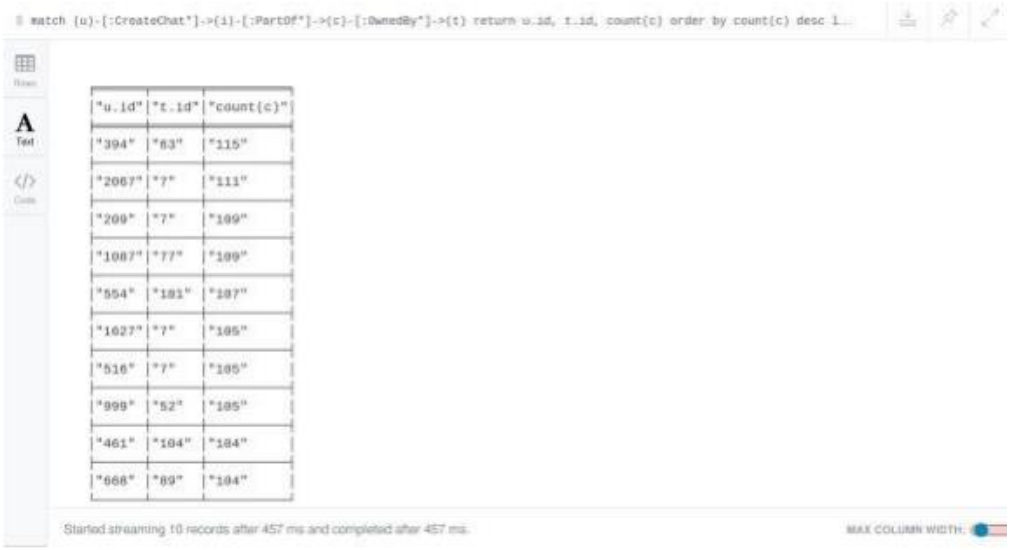
Final Result

Combine the two query above together as follows:

- Query

```
match (u)-[:CreateChat*]->(i)-[:PartOf*]->(c)-[:OwnedBy*]->(t)
return u.id, t.id, count(c)
order by count(c) desc limit 10
```

- Screenshot



As result shows, the user 999, which in the team 52 is part of the top 10 chattiest teams, but other 9 users are not part of the top 10 chattiest teams. This states that most of the chattiest users are not in the chattiest teams.

How Active Are Groups of Users?

In this question, we will compute an estimate of how “dense” the neighborhood of a node is. In the context of chat that translates to how mutually interactive a certain group of users are. If we can identify these highly interactive neighborhoods, we can potentially target some members of the neighborhood for direct advertising. We will do this in a series of steps.

- We will construct the neighborhood of users. In this neighborhood, we will connect two users if
  - one Mentioned another user in a chat, one CreateChat in response to another user's ChatItem

```
match (u1:User)-[:CreateChat]->(i)-[:Mentioned]->(u2:User)
create (u1)-[:InteractsWith]->(u2)
```

  - one created a ChatItem in response to another user's ChatItem

```
match (u1:User)-[:CreateChat]->(i1:ChatItem)-[:ResponseTo]-
(i2:ChatItem) with u1, i1, i2 match (u2)-[:CreateChat]-(i2) create
(u1)-[:InteractsWith]->(u2)
```

- The above scheme will create an undesirable side effect if a user has responded to her own chatItem, because it will create a self-loop between two users. So, after the first two steps we need to eliminate all self-loops involving the edge "InteractsWith".

```
match (u1)-[r:InteractsWith]->(u1) delete r
```

For each of these neighbors, we need to find

  - the number of edges it has with the other members on the same list

```
match (u1:User)-[r1:InteractsWith]->(u2:User) where u1.id <> u2.id with
u1, collect(u2.id) as neighbors, count(distinct(u2)) as neighborAmount
match (u3:User)-[r2:InteractsWith]->(u4:User) where (u3.id in neighbors)
AND (u4.id in neighbors) AND
(u3.id <> u4.id) return
u3.id, u4.id, count(r2)
```

  - If one member has multiple edges with another member we need to count it as 1 because we care only if the edge exists or not.

```
match (u1:User)-[r1:InteractsWith]->(u2:User) where u1.id <> u2.id with
u1, collect(u2.id) as neighbors, count(distinct(u2)) as neighborAmount
match (u3:User)-[r2:InteractsWith]->(u4:User) where (u3.id in neighbors)
AND (u4.id in neighbors) AND
(u3.id <> u4.id) return u3.id,
u4.id, count(r2), case
when count(r2) > 0 then
1 else 0 end as value
```

  - Last step, combine all steps above together.

```
match (u1:User)-[r1:InteractsWith]->(u2:User) where u1.id <> u2.id with u1,
collect(u2.id) as neighbors, count(distinct(u2)) as neighborAmount match
(u3:User)-[r2:InteractsWith]->(u4:User) where (u3.id in neighbors)
AND (u4.id in neighbors) AND (u3.id <> u4.id) with u1, u3, u4,
neighborAmount, case when (u3)-->(u4) then 1 else 0 end as value return
u1, sum(value)*1.0/(neighborAmount*(neighborAmount-1)) as coeff order by
coeff desc limit 10
```

Most Active Users (based on Cluster Coefficients)

User ID	Coefficient
209	0.9523809523809523
554	0.9047619047619048
1087	0.8

Recommended Actions

- Offer more products to iPhone users
  According to the decision tree classification, it reflects that most users which on the platform iPhone are HighRollers, so offering more products to them can increase our revenue.
- Provide more products to "high level spending user"
  It is similar as the first one, but the users are not totally identical. Thanks to Clustering results, we know that the "high level spending user" clicked less but buy more than others, we can provide more products to them for increasing the revenue.
- Provide some fixed pay packages or promotion to users, especially to "low level spending user"
  This action can stimulate consumption of users, and since the paying probability of "low level spending user" is low, the promotion can encourage them to purchase.