UNIVERSITAT POLITÈCNICA DE CATALUNYA

FACULTAT D'INFORMÀTICA DE BARCELONA

BWave

Joint Project - Semantic Data Management

Spring 2022

Authors:
ABUSALEH, Ali G. A, *email*: ali.g.a.abusaleh@estudiantat.upc.edu
Lorencio Abril, Jose Antonio, *email*: jose.antonio.lorencio@estudiantat.upc.edu
Mayorga Llano, Mariana, *email*: mariana.mayorga@estudiantat.upc.edu

Professors: **Oscar Romero**

Contents

1 Introduction

In the rapidly evolving digital landscape, social networks have become indispensable platforms for connecting individuals and facilitating communication. BWave endeavors to elevate user interaction by developing a unique social network that revolves around a conversational chatbot. This chatbot is meticulously designed to comprehend user interests, enabling the system to provide tailored recommendations of other users with shared interests for exploration and engagement.

2 M1 Graph-based solutions in BWave

In our project, we leverage graph-based solutions to enhance the functionality and effectiveness of BWave, our social networking platform. By utilizing a property graph model in Neo4j within the Exploitation Zone, we harness the power of graph databases to capture and model the complex relationships present in social networks.

Firstly, the choice to adopt a graph model in Neo4j for the Exploitation Zone is based on the inherent nature of social networks and the relationships present within the data. Graphs provide a natural and efficient way to capture and model connections between users, locations, and interests, among other elements. By leveraging this model, we can uncover valuable insights about social interactions, communities, and user preferences.

Secondly, graph's ability to facilitate fast and intuitive traversal of relationships offers a powerful foundation for developing recommendation systems in social networks. The graph structure allows us to incorporate multiple factors, including user preferences, social connections, shared interests, and past behaviors, resulting in more accurate and personalized recommendations.

By leveraging Neo4j's graph algorithms for property graphs and their querying capabilities, we can implement sophisticated recommendation algorithms that enhance user engagement, enable content discovery, and foster meaningful interactions within our social network. Additionally, Neo4j's graph algorithms could allow us to perform other relationship-centric analyses for further implementations, such as identifying influencers, detecting social communities, or finding common friends, expanding the analytical capabilities of our platform.

Furthermore, the scalability of graph databases like Neo4j ensures that our platform can accommodate a growing user base and handle large-scale graphs efficiently. This scalability ensures that our social networking platform delivers the performance and user experience expected in a dynamic and evolving digital landscape.

Additionally, for the BWave recommender system in the Analytical Zone, we leverage a combination of techniques based on cosine similarity between embeddings and Jaccard similarity coefficient to provide enhanced friend recommendations, bringing together the strengths of both approaches.

3 M2 Graph family evaluation

In our current application, the focus is on leveraging the power of graph-based solutions to enhance user interaction and provide tailored recommendations within a social network environment. At this stage, knowledge graphs and property graphs cover this main requirements, making any of them a viable solution.

Although we need a basic semantics, this is covered by the use of openAI when it analyses the users conversations and posts. The additional advanced reasoning, inference, or semantic modeling capabilities that knowledge graphs typically offer are not required by our application at this point, and considering that complex ontologies, formal semantics, and reasoning capabilities can introduce additional complexity and overhead that may not be necessary for our application, the decision was made on using a property graph.

Compared to Knowledge graphs, property graphs additionally offer greater flexibility in terms of schema evolution and agile development. They do not require rigid ontologies or predefined schemas, allowing for dynamic updates and modifications to the graph structure as the project evolves. We consider this flexibility is particularly beneficial in BWave to be able to adapt as our project evolves and new types of relationships, interests, or attributes emerge. Property graphs design provides the flexibility necessary to adapt to changing data structures without requiring extensive schema modifications, offering a seamlessly way to incorporate this unstructured variations into our data model and ensuring that our platform remains relevant and up-to-date with the objective of enabling us to keep pace with emerging trends, user interactions, and evolving user preferences.

However, this flexibility also introduces the challenge of managing data modeling in property graphs and complexity to ensure data consistency and integrity within a flexible graph structure. Nevertheless, we have addressed this concern in the earlier stages of data processing in the Formatted Zone, where we are ensuring the quality of the data we are handling in the Exploitation Zone which mitigates this risk.

Therefore, although both families can provide the basic requirements for the project, considering that knowledge graphs may introduce unnecessary complexity for our specific use case and that propery graphs offer a greater flexibility, we concluded that the advantages of using a property Graph in Neo4J overweighs its potential drawbacks, making it an optimal choice for BWave.

4 M3 Graph Design

For the graph design, we are considering the nodes and relationships shown in Figure 1. As we can see, it is a user-centric approach considering that our focus is on recommendations of users to users.

Although the 'User' node has attributes of its own, considering the high frequency of filters needed over the users' gender, language and location, these concepts where designed as different nodes to improve performance on queries that require filters based on these values, especially considering that the recommender system is one of our main functionalities and it takes these factors into consideration.

The granularity of the location of the user was designed such as 'City' and 'Country' are represented as different nodes connected by a relationship of belongingness. This was a decision based on improving the performance of the filters by city. We could have added information about the country as an attribute, however, this would have implied redundancy and challenges for maintenance in case of updates, therefore, we decided to keep the country as a node itself.

Regarding users' interests, in this first approach, we decided to focus on Music, Movies, and Food. Users are connected to each item that they liked with different relationships depending on the type of item (Music, Movie, or Food) to easily filter between topics. We also considered adding this information as an attribute to the item nodes, however, this would have impacted

performance and increased redundancy. Connecting these items to a new node with the topic was also considered, but this implied an extra hop if we wanted to focus on items from a specific user for a particular topic. Therefore, we maintained this distinction between the relationships.

Relationships between users and items are also differentiated by the preference of the user (if they like it or not). The decision of making different relations based on preferences relies on the importance to differentiate not only what elements has the user talked about but also how they feel about these elements.

Figure 1: BWave Graph Design

5 M4 Graph Population

Automated data pipelines were implemented with Apache Airflow to extract the data from the Landing Zone and load the validated data incrementally into the Formatted Zone. It was additionally implemented between the Formatted Zone and the Exploitation Zone in Neo4J. This automation eliminates manual intervention and reduces the risk of human errors, ensuring consistent and accurate data preparation. This data transfer is scheduled to be executed daily, ensuring that the graph database stays up to date with the latest validated data. This seamless data transfer ensures the availability of relevant insights and recommendations within our social networking platform.

6 M5 Graph Exploitation

In the BWave recommender system, we leverage a combination of a first approach with cosine similarity between embeddings and a second approach based on Jaccard similarity coefficient to provide enhanced friend recommendations, bringing together the strengths of both implementations.

6.1 Data Preparation

As shown in Figure 2 and Figure 3, after Facebook's posts and conversations from the chatbot are collected in text format, our data goes through a process of NLP where OpenAI is used to identify the users' interests structured in JSON files. Considering the variations presented in the format of the mentioned files, the data goes through a validation process in the Formatted Zone to ensure data quality. If the JSONS follows the structure (or if we are able to match the obtained structure to the desired one) and the type of data it must have, then it moves forward to be imported into our Graph in the Exploitation Zone, from where the Analytical Zone runs the recommender systems.

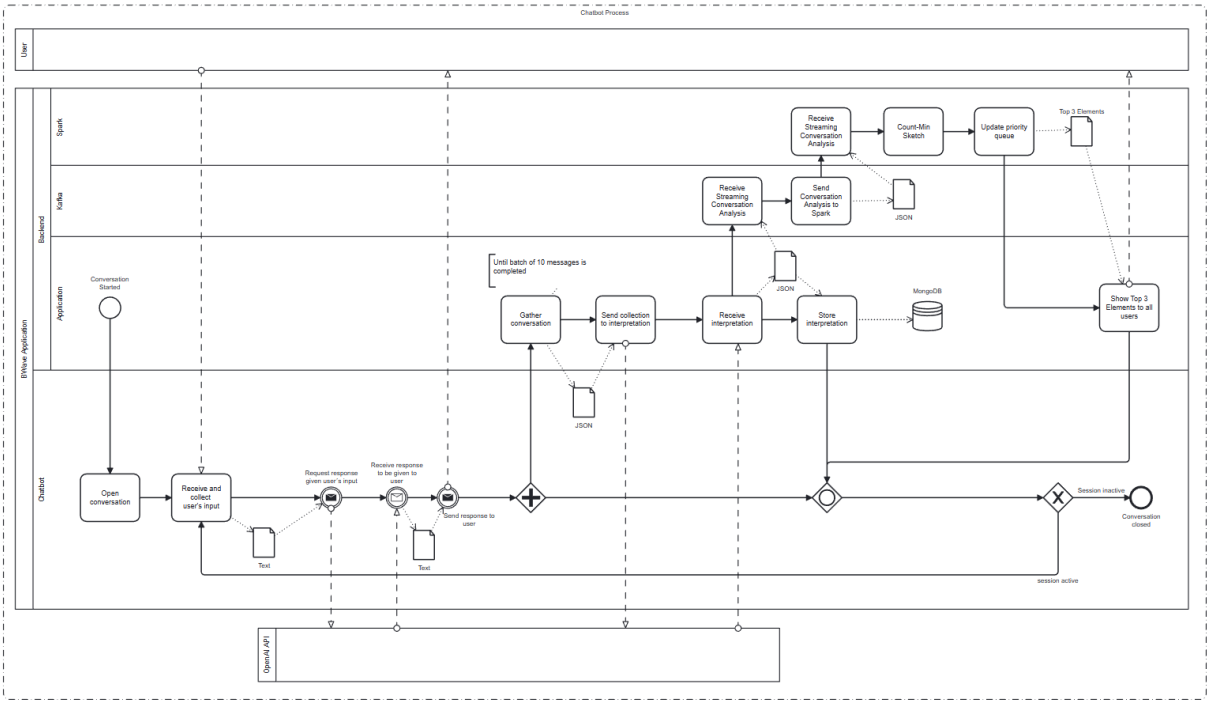


Figure 2: BWave Chatbot Main Process

6.2 Creation of Models

For this process, two recommender systems where implemented. Although both aim for the same objective, their characteristics differ from one another.

6.2.1 Reccommender based on Jaccard Similarity Coefficient

Our first approach uses the Jaccard similarity coefficient (Appendix A.3), which is a measure that quantifies the similarity between two sets. It is used when dealing with categorical or binary data, such as presence or absence of features or items, which in our case would be the likes and dislikes of elements.

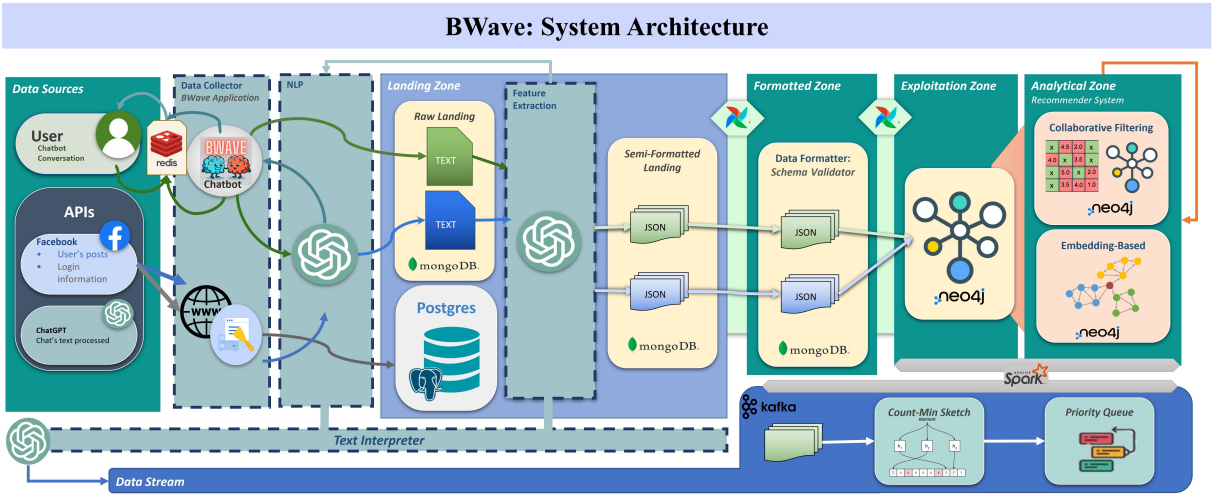


Figure 3: BWave Architecture

Firstly, we filter our recommendation for users who live in the same country. Then, the query matches per user the likes and dislikes in food, movies, and music categories. It collects the liked items and stores them in the "userItemsLikes" collection. The same is done with the disliked items, which are stored in the "userItemsDisLikes" collection.

Next, the query calculates the intersection and union of liked items between users, as well as the intersection and union of disliked items. This allows for the identification of commonly liked and disliked items and the consolidation of all liked and disliked items.

With these collections and counts, the query calculates the Jaccard similarity coefficient, which is determined by the proportion of common liked and disliked items to the total count of liked and disliked items. It ranges between 0 and 1, where a value of 1 indicates that the sets are identical, and 0 indicates no similarity. In our application, the Jaccard similarity coefficient is valuable to identifying the similarity between users based on shared interests.

Furthermore, we propose a revised approach in which the different topics taken into account for the Jaccard similarity measure are computed independently, enabling us to provide the user not only with friends recommendation, but also showing what things they share with this other person. This approach is presented in the Appendix A.4.

6.2.2 Reccomender based on Cosine Similarity between Embeddings

To implement the recommender based on embeddings, two essential steps need to be performed: projecting the catalog of the graph and calculating the embeddings. By incorporating the steps, we ensure the availability of relevant and current information for the recommender system. This approach enables us to provide accurate and timely recommendations to users based on the most up-to-date graph data.

Projecting Catalog The projection of the catalog is a one-time process unless there are schema changes in the graph . This step involves mapping the relevant entities and relationships in the graph to create a catalog that serves as a foundation for subsequent operations (Appendix A.1).

Embedding Creation The calculation of embeddings is an iterative process that needs to be performed regularly to update the embeddings . In our workflow, we have integrated this step

into the Airflow pipeline, ensuring that the embeddings are recalculated on a daily basis. By leveraging Airflow's scheduling capabilities, we maintain up-to-date embeddings that reflect any changes in the underlying graph data.

In our research, we utilized the node2vec algorithm to generate embeddings for the nodes in our graph (Appendix A.2). We made use of the gds.beta.node2vec.stream procedure, which takes the name of our graph as a parameter and additional options such as the desired embedding dimension which in this case was set to 7 considering one dimension per attribute.

In our implementation, a dimensionality of 7 was chosen for the node2vec algorithm based on careful considerations and assumptions. Each dimension in the embedding space represents a specific user attribute, allowing us to capture the diverse characteristics of the users in our graph.

Analyzing the graph schema shown in Figure 1, we identified 6 direct connections between the User's Node and other nodes representing various attributes such as movies, music, food preferences, language, gender, and city. These attributes provide valuable insights into users' interests and preferences, allowing for a more comprehensive understanding of their profiles. Additionally, we included the User's Node itself as an attribute dimension to capture the user's overall behavior and interactions within the graph.

However, we made a decision to exclude the "country" attribute as a separate dimension. This choice was motivated by the low probability of having users from the same city in different countries. Although there might be instances where this occurs, considering the country as a separate dimension would introduce unnecessary complexity and offer limited additional information.

By selecting a lower-dimensional embedding space, specifically with a dimensionality of 7, we aimed to strike a balance between capturing the essential features of the graph and maintaining computational efficiency. Higher-dimensional embeddings can provide a more fine-grained representation of the graph, but they come at the cost of increased computational resources and response time. With our chosen dimensionality, we can effectively capture the relevant information and attributes of the graph while minimizing the computational overhead, enabling faster response times during graph querying and analysis.

Therefore, the dimensionality of 7 in our node2vec implementation allows us to create meaningful embeddings that encapsulate user attributes, enabling us to leverage the power of the knowledge graph while ensuring efficient processing and analysis of the graph data. Once calculations are finished, we assign the resulting embeddings to the corresponding user nodes in the graph so that the recommender can utilize them to generate personalized recommendations for users based on their similarity to other users or items in the graph.

Cosine Similarity Recommender

For our second approach, we have used embeddings to calculate the cosine similarity between users (Appendix A.5). The Cosine Similarity is a measure that quantifies the resemblance between two vectors by calculating the cosine of the angle between them. In the context of embeddings, cosine similarity allows us to measure the similarity of two vectors based on their orientation in a high-dimensional vector space. Cosine similarity ranges between -1 and 1, where a value of 1 indicates perfect similarity, 0 indicates no similarity, and -1 indicates perfect dissimilarity.

First, the code retrieves the embedding for the user, which represents the numerical representation of the user's characteristics and preferences. Then, using the gds.similarity.cosine()

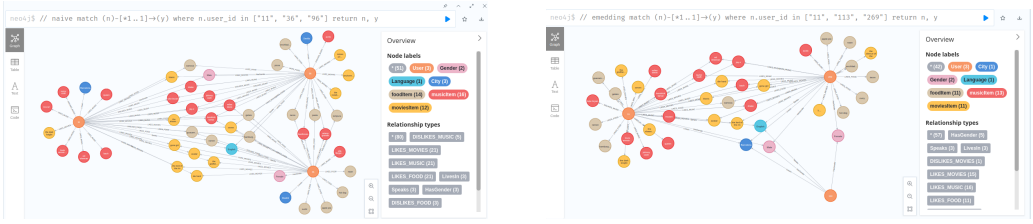
function, we are able to calculate the cosine similarity between the embedding of the users by calculating the cosine of the angle between them. As a result, we now have our recommendation candidates.

6.3 Added Value

Recommender Systems Both recommender approaches serve different requirements, in our use case, since the schema is fixed by leveraging the OpenAI language model, in terms of accuracy, the Jaccard similarity coefficient captures user preferences based on behavior and collective patterns, while cosine similarity between embeddings captures semantic resemblance and incorporate additional information, enabling the system to identify connections beyond explicit preferences. This combination can result in more accurate and personalized friend recommendations by considering explicit and implicit user preferences. It can additionally promote recommendation diversity since Jaccard similarity coefficient focuses on collective behavior (common intersections) ensuring that recommendations cover a wide range of user preferences, while cosine similarity between embeddings can identify a niche or specific interest by treating the nodes and relations as one block.

In the figure [4], we can see different recommended matches for a specific user[1], in this case, figure [4(a)], shows users using Jaccard similarity approach, in this approach user's preferences of the suggested match are not affecting the results and the top-2 matches are calculated based on the common interests rather than each user node and interests. On the other side, figure [4(b)] is taking into account the whole user's context for calculating the similarity, in order to better understand the differences, we plot the user's embeddings in a figure in 2D just for the demonstration purposes (embeddings calculated as 7D in the code) [5]. Therefore, we can see how Embedding is giving different suggestions than the Jaccard since the vectors' representation of Jaccard suggestions is not fully aligned with the targeted user's vector.

Therefore, Jaccard similarity approach will give easier-to-understand and faster results in our case where the graph schema is restricted and under our control. In addition, as we saw, using Jaccard similarity to obtain shared interest is very straightforward. On the other hand, Embedding-based will serve more relevant results when we start expanding the graph with more unrestricted data to capture more complex and/or unknown relationships for further analysis. For instance, if we want not only to recommend those elements that both users are connected to, but also other things, such as a new possible food that they both may like, the cosine similarity approach would be an easier solution.



(a) Jaccard similarity Top-2 users (b) Embedding-based Top-2 users

Figure 4: Jaccard vs Cosince similarity Top-2 users

[1]Targeted User has ID=11

[2]E1, E2, N1, N2, User represents Embedding Top-1, Top-2, Jaccard Top-1, Top,2 targeted user respectively.

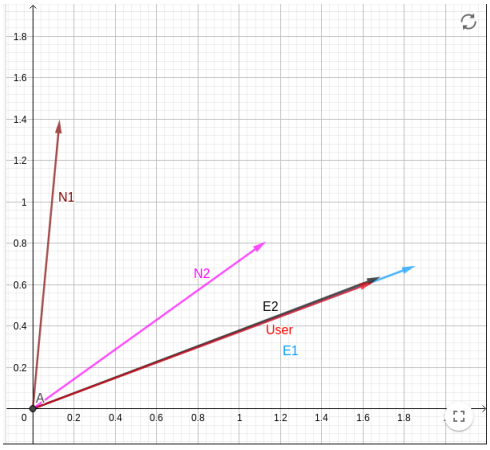


Figure 5: 2D representation of Top-2 User's suggestions [2]

Solutions to Recommender Systems' Common Issues Data sparsity is a common challenge in recommender systems, especially when the number of interactions is limited, and can be a challenge at early stages. However, the combination of the approach based on Jaccard similarity coefficient and cosine similarity between embeddings can further reduce this impact. The former can leverage the behavior of similar users to fill data gaps and generate relevant recommendations, while the latter can also handle sparse data by incorporating additional information and capturing implicit preferences. Together, these approaches can improve recommendation quality even when data is sparse.

Additionally, although we are mitigating this issue by retrieving users posts to get more information about their preferences, the integration of cosine similarity between embeddings allows BWave to handle the cold-start problem more effectively. When there is limited interaction data for new users or items, cosine similarity between embeddings can leverage available attributes, metadata, or contextual information to generate initial representations. This enables the system to provide meaningful recommendations from the start, even for users or items with sparse data.

7 M7 Proof of Concept

To validate the concepts discussed in the previous sections, we have implemented a proof of concept (PoC) in collaboration with the team. The code and resources for the PoC can be found in our joint project repository on GitHub: Git Repository[3]. A simple example is shown in Appendix B.

To run the project and execute the PoC, it is necessary to set up the environment with the following requirements and configurations:

- Requirements:
  - Neo4j 4.2.19.
  - Neo4j Graph Data Science Library 2.3.8.

[3]Please use the following Github account to access the project in Github
Username: "bwave23"
Password: "upcjointproject"

  - Generated JSON data loaded into MongoDB
    * Download the JSON files to upload.
    * Run the script to import them into MongoDB.
  - Cypher scripts for the PoC.

- Configurations:
  - Enable Neo4j GDS and APOC libraries. Configuration steps may vary depending on your operating system. Please refer to the official documentation here for more information.
  - Disable security in Neo4j configurations. Configuration steps may vary depending on your operating system. Please refer to the official documentation here for more information.

Once the requirements are installed and the environment is properly configured, follow the steps below to execute the PoC:

- Loading the data: the easiest way to load the data is to manually run the scripts

  bwave_airflow/scripts/create_formatted_zone.py,

  bwave_airflow/scripts/conversation_analysis_formatter.py,

  bwave_airflow/scripts/fb_posts_analysis_formatter.py,

  bwave_airflow/scripts/create_update_graph.py,

  bwave_airflow/scripts/project_catalog.py,

  and finally *bwave_airflow/scripts/calculate_embeddings.py*.

  If you want to automate the process, the directory *bwave_airflow/* is designed to work in Apache Airflow. If you have Apache configured, you only need to copy this folder inside the *airflow/dags* folder[4].

- Jaccard Similarity Recommender:
  - Execute the script mentioned in Listing **??** by replacing "user_id" with the targeted user's ID. This script calculates the Jaccard similarity coefficient between the targeted user and other users in the system, providing recommendations based on common likes and dislikes.

- Embedding-based Similarity:
  - Project the graph catalog by executing the project_catalog.cypher script. This step prepares the graph structure for embedding calculations.

[4]Note that you might need to update some path variables. Refer to the README.md in our repository for further details on this.

- Calculate the embeddings by executing the calculate_embeddings.cypher script. This step generates embeddings for users based on their interactions and preferences.
- Execute the recommender script mentioned in Listing A.5 by replacing "User_ID" with the targeted user's ID. This script utilizes cosine similarity between user embeddings to provide personalized recommendations.

By following these steps, you will be able to execute an end-to-end example of our PoC, showcasing the functionality and effectiveness of the proposed recommendations system.

8 M8 Limitations and Further Work

In this proof-of-concept project, we've developed a system that analyzes conversation topics and provides user recommendations based on their interests. While successful, it is evident that our implementation has room for improvement and refinement.

One of the most prominent limitations involves the scope of our application, which currently considers only three topics for analysis. This was a strategic decision to enhance the precision of our conversation analysis, focusing our attention on key aspects of the business concept. However, in a real-world setup, the complexity of the graph schema would undoubtedly increase, accommodating a broader range of topics and user characteristics. The challenges and potential benefits of this expansion merit further investigation.

Our recommendation algorithms currently provide two-fold recommendations, prioritizing common interests (Jaccard similarity) and utilizing a more flexible approach to exploit other aspects of the graph structure (cosine similarity with embeddings). Notwithstanding, Graph Convolutional Networks (GCN) represent a promising area for future exploration. While initially not selected due to their complexity, they could further leverage the graph topology to uncover less apparent relations for recommendation.

Regarding how we keep the preferences of the users updated, our current approach is to update this information to the latest information provided by the user. For example, if they liked 'pizza' yesterday, and today they don't like it anymore, then the relationship would change to reflect this. We have thought about different approaches to address this problem, such as maintaining a counter of how many times the user has stated to like something, and same with dislikes, but we were not able to design a consistent approach. Our intuition tells us that such approach exists, and that we should look carefully into this issue in future development.

Finally, we acknowledge the challenge of synonym detection. In the open context of our application, recognizing that 'tuna pizza' and 'pizza with tuna' refer to the same concept is crucial for accurate analysis. Although currently disregarded due to its complexity, the use of semantic similarity or word embeddings could be considered for future development.

In conclusion, while our proof-of-concept project shows promising results, we recognize the importance of addressing these limitations and expanding on the areas outlined for further work. Doing so will increase the robustness and precision of our system, ensuring its adaptability and effectiveness in the dynamic world of user preferences and interests.

9 References

1. Alberto Abello, Sergi Nadal. (2023) *Big Data Management*. Chapters 1 - 10.
2. Polikoff, I. (2020, August 19). Knowledge Graphs vs. Property Graphs – Part I. Retrieved from https://tdan.com/knowledge-graphs-vs-property-graphs-part-1/27140
3. Polikoff, I. (2020, September 16). Knowledge Graphs vs. Property Graphs – Part II. Retrieved from https://tdan.com/knowledge-graphs-vs-property-graphs-part-ii/27271
4. Foote, K. D. (2022, January 5). Property Graphs vs. Knowledge Graphs. Retrieved from https://tdan.com/knowledge-graphs-vs-property-graphs-part-ii/27271
5. Singh Walia, M. (2022, January 28). A Comprehensive Guide on Neo4j. Retrieved from https://www.analyticsvidhya.com/blog/2022/01/a-comprehensive-guide-on-neo4j-graph-database/
6. Neo4j. (n.d.). Top Ten Reasons for Choosing Neo4j. Retrieved from https://neo4j.com/top-ten-reasons/
7. Neo4j. (n.d.). Why Graph Databases? Retrieved from https://neo4j.com/why-graph-databases/
8. Javed, M. (2020, November 4). Using Cosine Similarity to Build a Movie Recommendation System. Retrieved from https://towardsdatascience.com/using-cosine-similarity-to-build-a-movie-recommendation-system-ae7f20842599
9. Elkhattam, A. (n.d.). Building a Content-based Recommender using a Cosine-Similarity Algorithm. Retrieved from https://a-elkhattam.medium.com/imdb-movie-recommendation-chatbot-942f84dfa0dc
10. Meor Amer. (2022, June 21). Article Recommender with Text Embedding, Classification, and Extraction. Retrieved from https://txt.cohere.com/article-recommender/
11. Maretha, A. (2022, July 31). Item Recommendation using Jaccard Coefficient. Retrieved from https://amaretha.medium.com/item-recommendation-using-jaccard-coefficient-f74df5c255b3
12. Putty, M. (n.d.). Measuring similarity in recommendation systems. Retrieved from https://levelup.gitconnected.com/measuring-similarity-in-recommendation-systems-8f2aa8ad1f44
13. Nandi, M. (2017, July 14). Recommender Systems through Collaborative Filtering. Retrieved from https://www.dominodatalab.com/blog/recommender-systems-collaborative-filtering
14. Ayub, M., Ghazanfar, M. A., Maqsood, M., & Saleem, A. (n.d.). A Jaccard base similarity measure to improve performance of CF based recommender systems. IEEE. Retrieved from https://ieeexplore.ieee.org/document/8343073
15. Bag, S., Kumar, S. K., & Tiwari, M. K. (2019). An efficient recommendation generation using relevant Jaccard similarity. Information Sciences, 475, 228-247. Retrieved from

https://doi.org/10.1016/j.ins.2019.01.023

16. Fkih, F. (2022, October). Similarity measures for Collaborative Filtering-based Recommender Systems: Review and experimental comparison. Retrieved from https://doi.org/10.1016/j.jksuci.2021.09.014

17. Wang, Y., Deng, J., Gao, J., & Zhang, P. (2017, August). A Hybrid User Similarity Model for Collaborative Filtering. Retrieved from https://doi.org/10.1016/j.ins.2017.08.008

18. Apache Kafka. (n.d.). Kafka Streams. Retrieved from https://kafka.apache.org/documentation/streams/

19. Babatunde, A. (2022, November 30). Why Spark Structured Streaming Could Be The Best Choice. Retrieved from https://www.netguru.com/blog/spark-streaming#:~:text=Apache%20Spark%20Structured%20Streaming%20is,with%20virtually%20no%20code%20changes.

Appendix A Appendix A: Code

A.1 Project Catalog Creation

```
CALL gds.graph.project(
  'myGraph', // Name of the graph
  {
    User: {
      label: 'User'
    },
    Language: {
      label: 'Language'
    },
    musicItem: {
      label: 'musicItem'
    },
    foodItem: {
      label: 'foodItem'
    },
    Country: {
      label: 'Country'
    },
    Gender: {
      label: 'Gender'
    },
    City: {
      label: 'City'
    },
    moviesItem: {
      label: 'moviesItem'
    }
  },
  {
    LIKES_MOVIES: {
      type: 'LIKES_MOVIES',
      orientation: 'UNDIRECTED'
    },
    IsIn: {
      type: 'IsIn',
      orientation: 'UNDIRECTED'
    },
    DISLIKES_MUSIC: {
      type: 'DISLIKES_MUSIC',
      orientation: 'UNDIRECTED'
    },
    DISLIKES_MOVIES: {
      type: 'DISLIKES_MOVIES',
      orientation: 'UNDIRECTED'
    },
    HasGender: {
      type: 'HasGender',
      orientation: 'UNDIRECTED'
    },
    LIKES_FOOD: {
      type: 'LIKES_FOOD',
      orientation: 'UNDIRECTED'
    },
```

```
    FOLLOWS: {
      type: 'FOLLOWS',
      orientation: 'UNDIRECTED'
    },
    LivesIn: {
      type: 'LivesIn',
      orientation: 'UNDIRECTED'
    },
    LIKES_MUSIC: {
      type: 'LIKES_MUSIC',
      orientation: 'UNDIRECTED'
    },
    Speaks: {
      type: 'Speaks',
      orientation: 'UNDIRECTED'
    },
    DISLIKES_FOOD: {
      type: 'DISLIKES_FOOD',
      orientation: 'UNDIRECTED'
    }
  }
)
```

A.2 Embedding Creation Code

```
    CALL gds.beta.node2vec.stream(
    'myGraph', -- Name of the graph
    {embeddingDimension: 7} -- Configuration options
    )
    YIELD nodeId, embedding
    WITH gds.util.asNode(nodeId) AS node, embedding
    SET node.embedding = embedding
```

A.3 Jaccard Similarity Code

```
MATCH (xuser:User {{user_id: "{user_id}"}})-[:LivesIn]->(:City)-[:IsIn]->(
    country:Country)<-[:IsIn]-(:City)<-[:LivesIn]-(otherUser:User)
WHERE xuser <> otherUser

MATCH (xuser)-[:LIKES_FOOD|LIKES_MOVIES|LIKES_MUSIC]->(x)
WITH xuser, otherUser, COLLECT(DISTINCT x) AS userItemsLikes


MATCH (xuser)-[:DISLIKES_FOOD|DISLIKES_MOVIES|DISLIKES_MUSIC]->(x)
WITH otherUser, userItemsLikes, COLLECT(DISTINCT x) AS userItemsDisLikes



MATCH (otherUser)-[:LIKES_FOOD|LIKES_MOVIES|LIKES_MUSIC]->(x)
WITH otherUser, userItemsLikes, userItemsDisLikes, COLLECT(DISTINCT x) AS
    otherUserItemsLikes

MATCH (otherUser)-[:DISLIKES_FOOD|DISLIKES_MOVIES|DISLIKES_MUSIC]->(x)
WITH otherUser, userItemsLikes, userItemsDisLikes,otherUserItemsLikes,
    COLLECT(DISTINCT x) AS otherUserItemsDisLikes
```

```
18
19
20 WITH otherUser, userItemsLikes, userItemsDisLikes,otherUserItemsLikes,
       otherUserItemsDisLikes,
21      apoc.coll.intersection(userItemsLikes, otherUserItemsLikes) AS
       LikecommonItems,
22      apoc.coll.union(userItemsLikes, otherUserItemsLikes) AS allLikedItems,
23      apoc.coll.intersection(userItemsDisLikes, otherUserItemsDisLikes) AS
       DisLikecommonItems,
24      apoc.coll.union(userItemsDisLikes, otherUserItemsDisLikes) AS
       allDisLikedItems
25
26 WITH otherUser, LikecommonItems, allLikedItems,DisLikecommonItems,
       allDisLikedItems,
27      size(LikecommonItems) AS commonLikedCount,
28      size(allLikedItems) AS allLikedCount,
29      size(DisLikecommonItems) AS commonDisLikedCount,
30      size(allDisLikedItems) AS allDisLikedCount
31
32 WITH otherUser, commonLikedCount, allLikedCount, commonDisLikedCount,
       allDisLikedCount,
33      (commonLikedCount + commonDisLikedCount)  * 1.0 / (allLikedCount +
       allDisLikedCount) AS similarity
34
35 RETURN otherUser.username AS RecommendedFriend, similarity
36 ORDER BY similarity DESC
```

A.4 Jaccard similarity distinguishing the different topics

```
1  MATCH (xuser:User {{user_id: "{user_id}"}})-[:LivesIn]->(:City)-[:IsIn]->(
       country:Country)<-[:IsIn]-(:City)<-[:LivesIn]-(otherUser:User)
2  WHERE xuser <> otherUser
3
4  OPTIONAL MATCH (xuser)-[:LIKES_FOOD]->(food)
5  WITH xuser, otherUser, COLLECT(DISTINCT food) AS userLikesFood
6
7  OPTIONAL MATCH (xuser)-[:LIKES_MOVIES]->(movie)
8  WITH xuser, otherUser, userLikesFood, COLLECT(DISTINCT movie) AS
       userLikesMovie
9
10 OPTIONAL MATCH (xuser)-[:LIKES_MUSIC]->(music)
11 WITH xuser, otherUser, userLikesFood, userLikesMovie, COLLECT(DISTINCT
       music) AS userLikesMusic
12
13 OPTIONAL MATCH (otherUser)-[:LIKES_FOOD]->(food)
14 WITH xuser, otherUser, userLikesFood, userLikesMovie, userLikesMusic,
       COLLECT(DISTINCT food) AS otherUserLikesFood
15
16 OPTIONAL MATCH (otherUser)-[:LIKES_MOVIES]->(movie)
17 WITH xuser, otherUser, userLikesFood, userLikesMovie, userLikesMusic,
       otherUserLikesFood, COLLECT(DISTINCT movie) AS otherUserLikesMovie
18
19 OPTIONAL MATCH (otherUser)-[:LIKES_MUSIC]->(music)
20 WITH xuser, otherUser, userLikesFood, userLikesMovie, userLikesMusic,
       otherUserLikesFood, otherUserLikesMovie, COLLECT(DISTINCT music) AS
       otherUserLikesMusic
```

```
21
22 WITH xuser, otherUser, userLikesFood, userLikesMovie, userLikesMusic,
       otherUserLikesFood, otherUserLikesMovie, otherUserLikesMusic,
23      apoc.coll.intersection(userLikesFood, otherUserLikesFood) AS
       commonLikesFood,
24      apoc.coll.intersection(userLikesMovie, otherUserLikesMovie) AS
       commonLikesMovie,
25      apoc.coll.intersection(userLikesMusic, otherUserLikesMusic) AS
       commonLikesMusic,
26      size(apoc.coll.intersection(userLikesFood, otherUserLikesFood)) AS
       commonLikesFoodCount,
27      size(apoc.coll.intersection(userLikesMovie, otherUserLikesMovie)) AS
       commonLikesMovieCount,
28      size(apoc.coll.intersection(userLikesMusic, otherUserLikesMusic)) AS
       commonLikesMusicCount,
29      size(userLikesFood) AS userLikesFoodCount,
30      size(userLikesMovie) AS userLikesMovieCount,
31      size(userLikesMusic) AS userLikesMusicCount,
32      size(otherUserLikesFood) AS otherUserLikesFoodCount,
33      size(otherUserLikesMovie) AS otherUserLikesMovieCount,
34      size(otherUserLikesMusic) AS otherUserLikesMusicCount,
35      (size(apoc.coll.intersection(userLikesFood, otherUserLikesFood)) + size
       (apoc.coll.intersection(userLikesMovie, otherUserLikesMovie)) + size(
       apoc.coll.intersection(userLikesMusic, otherUserLikesMusic))) * 1.0 /
36      (size(userLikesFood) + size(userLikesMovie) + size(userLikesMusic) +
       size(otherUserLikesFood) + size(otherUserLikesMovie) + size(
       otherUserLikesMusic)) AS similarity
37
38 WITH xuser, otherUser, commonLikesFood, commonLikesMovie, commonLikesMusic,
       similarity
39 ORDER BY similarity DESC
40 LIMIT 3
41
42 RETURN otherUser.username AS RecommendedFriend, similarity,
43      CASE WHEN size(commonLikesFood) > 0 THEN commonLikesFood[0].name ELSE
       null END AS TopLikedFood,
44      CASE WHEN size(commonLikesMovie) > 0 THEN commonLikesMovie[0].name ELSE
        null END AS TopLikedMovie,
45      CASE WHEN size(commonLikesMusic) > 0 THEN commonLikesMusic[0].name ELSE
        null END AS TopLikedMusic
```

A.5 Cosine Similarity Code

```
1 MATCH (xuser:User {{user_id: "{user_id}"}})
2 WITH xuser
3
4 // Retrieve the embedding for the xuser
5 CALL {{
6   WITH xuser
7   MATCH (u:User) WHERE u.user_id = xuser.user_id
8   RETURN u.embedding AS embedding
9   LIMIT 1
10 }}
11 WITH xuser, embedding
12
13 // Calculate similarity with other users based on embeddings
```

```
MATCH (otherUser:User)
WHERE otherUser <> xuser
WITH xuser, embedding, otherUser, otherUser.embedding AS otherUserEmbedding

// Calculate cosine similarity between embeddings
WITH xuser, embedding, otherUser, gds.similarity.cosine(embedding,
    otherUserEmbedding) AS similarity

// Order the other users by similarity in descending order
ORDER BY similarity DESC

RETURN otherUser.username AS RecommendedFriend, similarity
```

Appendix B Running example

In Figure 6, one can observe the final outcome of our proof of concept. In this image, we can see how an user named Jose interacts with the bot, and is able to see the topics that people are liking the most, regarding food, music and movies. Whenever he wants, he can press the buttons to search for friends, getting recommendations based on what the system has learnt about him.

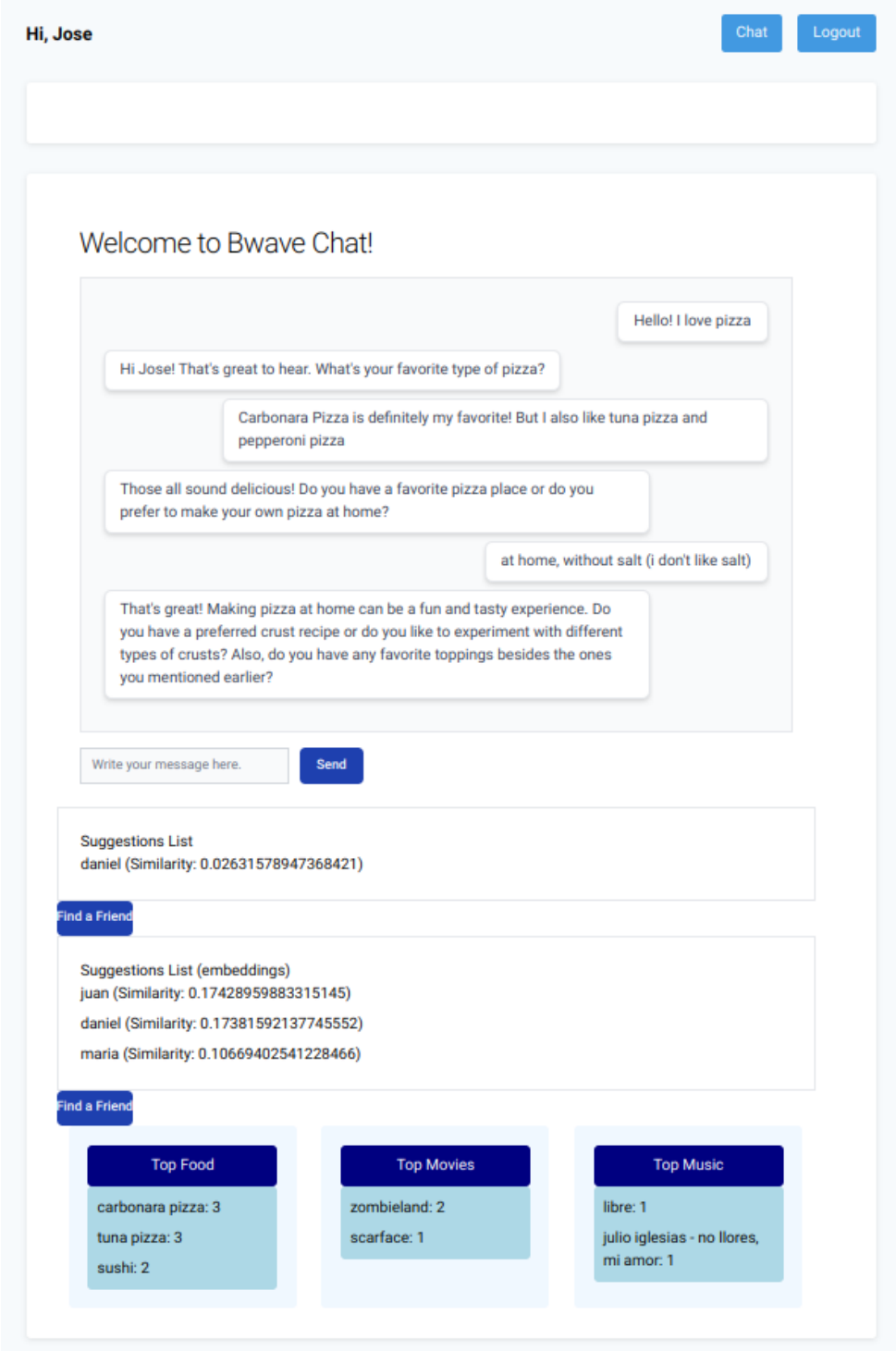


Figure 6: Full running example