SDM Joint Project

Nima Kamali Lassem, Kristóf Balázs, Stefanos Kypritidis

1 Introduction

In our project, we utilize a graph-based solution using Neo4j to enhance the generation of matching interest questions within the "Let's Talk" application. The graph database stores and connects users, keywords, topics, and questions through nodes and relationships. This structure allows for efficient querying of contextually relevant questions by traversing the graph, rather than relying on rigid relational schemas. Moreover, the Neo4j database serves as our exploitation zone, containing data that has been cleaned and transformed in the previous big data zones and thus is a vital part of our Let's Talk application.

2 Purpose statement

This project benefits from integrating a graph-based solution by enabling highly personalized and context-aware question recommendations for every pair of users. Neo4j's graph structure supports fast and intuitive relationship mapping between users, keywords, conversation topics, and questions, significantly enhancing user experience and the app's ability to achieve deeper, quicker, and more natural social connections.

Specifically, the graph-based system enables the following:

- Identification of communities of users with shared interests.
- Assessment of question similarity to recommend questions related to those a user has previously liked.
- Generation of personalized conversation starters tailored to specific user pairs, enhancing engagement and interaction quality

3 Graph family

For our Let's Talk application, we have chosen to use property graphs as the underlying graph model due to their flexibility, performance, and suitability for our use case. Unlike knowledge graphs, which are built around formal ontologies and semantic reasoning, our system relies on direct, observable relationships

between the entities of users, keywords, questions, and topics. These relationships do not require inference or hierarchical reasoning, making the lightweight structure of a property graph more appropriate. Our data model consists of simple node types with straightforward relationships, which aligns well with the schema-optional design of property graphs.

A key requirement for our application is quickness and responsiveness, especially when users are about to start a new conversation and expect immediate, personalized conversation starters. Property graphs—particularly when implemented using Neo4j—offer highly efficient graph traversal and fast query execution using the Cypher query language. This performance advantage is critical, as the system must rapidly identify shared interests and select relevant questions for pairs of users with minimal delay. Knowledge graphs, which often rely on SPARQL and semantic layers, tend to be slower and less appropriate for high-frequency, user-facing queries like those required in our application.

Furthermore, property graphs provide a scalable and developer-friendly foundation for ongoing growth and feature development. As the company may grow and new features may be introduced, we can easily extend the graph schema by adding new nodes or relationship types without restructuring the database. The mature ecosystem around Neo4j, including its Graph Data Science tools, allows us to perform advanced analytics like community detection and similarity scoring—capabilities that directly support our goal of providing mutually engaging questions and, in the future, user matching. Overall, property graphs provide the right balance of performance, flexibility, and simplicity for our big data-driven application.

Our application is highly scalable thanks to the use of Neo4j Aura, a fully managed graph database-as-a-service that is integrated with Google Cloud Services. Neo4j Aura allows us to scale seamlessly by simply upgrading our subscription tier to access more storage, memory, and compute resources as our user base and data volume grow. This eliminates the need for manual infrastructure management or complex database scaling strategies. Additionally, Neo4j Aura offers automatic backups, high availability, and optimized performance [1], ensuring that our graph database remains responsive and reliable even under increased load. This is very important for Let's Talk application, since we aim for increased usage and data complexity over time.

4 Graph Design

Our graph schema contains nodes which represent entities such as *User*, *Keyword*, *Topic*, and *Question*, and edges which capture the semantic connections between them. The model follows a simple yet expressive property graph structure that reflects the core relationships of our data.

Node Types

- **User**: Represents an individual using the application. Properties include:

  - `user_id`
  - `name`

- **Keyword**: A short tag or concept associated with questions and topics. Properties include:
  - `keyword`
- **Topic**: A broader subject area than keywords. Properties include:
  - `topic`
- **Question**: A stored conversation prompt. Properties may include:
  - `id`
  - `title`
  - `source`
  - `correct_answer` (optional)
  - `incorrect_answers` (optional)
  - `author` (optional)
  - `created_utc_ts` (optional)
  - `num_comments` (optional)
  - `subreddit` (optional)
  - `upvotes` (optional)
  - `body` (optional)
  - `original_topic` (optional)

Relationship Types

- `(:User)-[:INTERESTED_IN_TOPIC]->(:Topic)`: Connects a user to a topic they are interested in.
- `(:User)-[:LIKES]->(:Question)`: Indicates that a user has liked a question.
- `(:User)-[:INTERESTED_IN_KEYWORD]->(:Keyword)`: Indicates a user is interested in a particular keyword.
- `(:Question)-[:HAS_TOPIC]->(:Topic)`: Links a question to a relevant topic.
- `(:Question)-[:HAS_KEYWORD]->(:Keyword)`: Associates a question with a keyword.
- `(:Keyword)-[:BELONGS]->(:Topic)`: Groups a keyword under a broader topic.

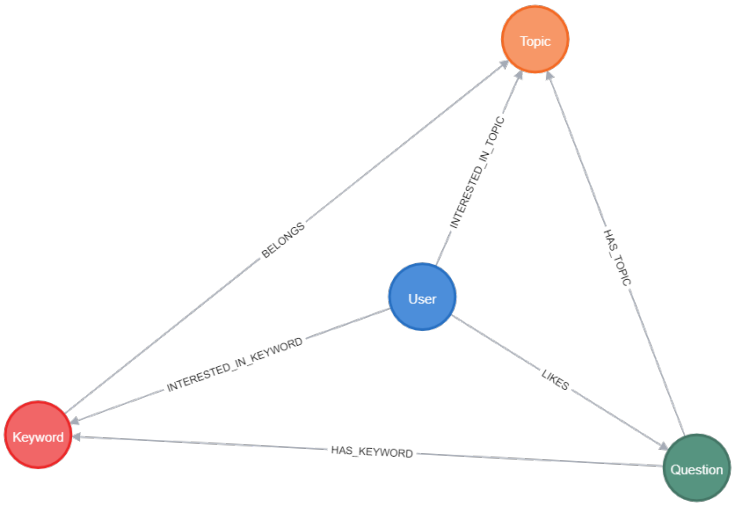

Figure 1: Neo4j Schema

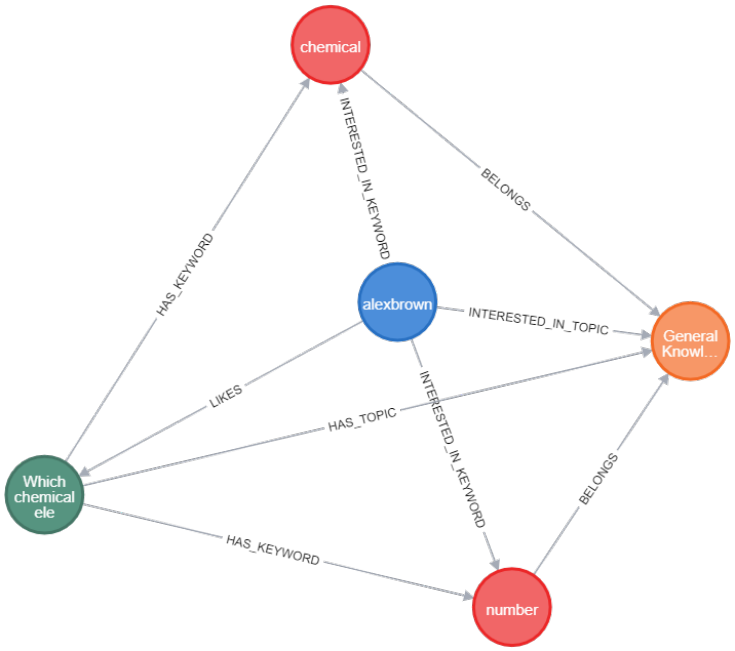

Figure 2: Instances Example

In figure 2 the relationships between the user *alexbrown*, the topic *General Knowledge*, the keywords *chemical* and *number*, and the question *"Which chemical element, number 11 in the Periodic table, has the symbol Na?"* are shown.

Alexbrown is shown to be interested in both the topic and the two keywords, and has liked the question, which is tagged with the same keywords and associated with the *General Knowledge* topic.

5 Loading data

This section details the design and implementation of the data flows used to populate the Neo4j graph database for the Let's Talk application. The graph is constructed from data collected from multiple external sources, processed using big data technologies, and loaded into Neo4j Aura. The architecture follows a multi-zone approach (Landing, Trusted, and Exploitation zones)

5.1 Data Acquisition: Landing Zone

The first stage in our pipeline involves gathering raw data from various external sources. These are chosen to provide rich, diverse, and engaging question content that can be matched to user interests. The data is ingested through the following sources:

- Reddit API: Extracts trending topics and user-generated questions from communities such as r/AskReddit (streaming data).
- Stack Exchange API: Collects popular and insightful questions.
- Trivia APIs: Supplies short, factual questions ideal for casual interactions.
- Wikipedia: Retrieves structured text from Wikipedia

All raw data is stored in its original format in Google Cloud Storage (GCS), forming the Landing Zone of our architecture.

5.2 Data Preparation: Trusted Zone

In this stage, the raw data is processed and cleaned using Apache Spark, enabling scalable and efficient handling of large volumes of semi-structured content. The following transformations are applied:

- Deduplication of questions
- Text normalization (e.g., removal of HTML tags, punctuation cleanup)
- Keyword extraction based on question
- Fix topic naming to match various sources
- Feature engineering using attributes and metadata
- Assinging keywords and topics from our real data to our synthetically generated users

- Schema transformation of questions into a structured format suitable for graph ingestion.

Also, the Wikipedia data was processed with local language model (Ollama) to generate relevant questions from that content. Finally, the processed data is stored as Delta Lake tables in GCS, forming the Trusted Zone of the pipeline.

5.3 Graph Loading: Exploitation Zone

The final stage of the flow involves loading the trusted data into the graph database. We use Neo4j AuraDB Professional hosted in the cloud. The graph loading process is implemented using:

- PySpark (Python + Apache Spark) for distributed data processing
- Neo4j Connector for Apache Spark to translate Spark DataFrames into Cypher queries.

The detailed script can be found in our Github A.1 under *scripts_exploitation_zone/script_load_neo4j.py*

6 Graph Analytics

We implemented two graph analytics tasks: one for community detection and one for node similarity. However, we weren't able to use the Aura Graph Analytics Serverless feature in our Neo4j Aura instance because we signed up with the minimum configuration (1 GB memory and 2 GB storage) due to limited free credits from Google Cloud Services. This feature requires at least 4 GB of memory. Instead, we ran the analytics locally using a Neo4j database instance and a sample of the full dataset. It's worth noting that the syntax for these graph analytics tasks is the same across environments, so if we had access to a higher-tier Neo4j Aura instance, the same code would have worked there too. Finally, the code can be found in the *neo4j_graph_analytics* folder.

6.1 Question similarity

Before, applying any algorithm, we had to create a graph data science (gds) projection containing only the questions, the keywords and the edges between them. Next, we applied the Node Similarity algorithm on the projection solely to the Question nodes using the overlap similarity coefficient. This coefficient is also known as the Szymkiewicz–Simpson coefficient and is calculated by dividing the number of shared elements (the intersection) by the size of the smaller of the two sets [2]. This coefficient is preferred over Jaccard Similarity—which compares the intersection to the union of both sets—because it handles cases where questions have very different numbers of keywords more effectively. By focusing on the smaller set, it avoids underestimating similarity just because one

question has many more keywords than the other. In other terms, questions with many keywords shouldn't be penalized for having more unique terms.

When two questions share some common keywords, the algorithm creates a new edge between them, with a weight showing how strong the similarity is. In detail, we also set a threshold of 0.2, meaning edges are only created for pairs of questions with a minimum of 0.2 overlap similarity coefficient. These edges can then be used to suggest follow-up questions to pairs of users, especially when they have just liked a question. Finally, an example can be seen in this Figure 5.

This kind of graph analytics adds great value to the app while taking full advantage of graph capabilities. Instead of just generating questions based on general interests, we can keep the conversation flowing with questions that are naturally connected to previously liked questions or that users may enjoy in their current mood. It also makes the system smarter over time, opening up possibilities for the future, like recommending questions based on interaction history.

6.2 Community detection

For community detection, our goal is to identify groups of users—also known as clusters—who share similar interests. To achieve this, we can use the Louvain algorithm available in Neo4j Aura. The Louvain method is effective for detecting communities in large networks by optimizing modularity, a measure that compares the density of connections within communities to those between them [3]. However, for the algorithm to work properly, the nodes must be interconnected, as it relies on these connections to identify communities.

The preprocessing step involves creating edges between users who share similar interests. This is the same approach we previously used for connecting similar question nodes. First, we create a GDS projection that includes only user and topic nodes, along with the edges connecting them. We then run the node similarity algorithm using the overlap coefficient, which is suitable since users can have varying numbers of interests. Weighted edges are added between users who share an overlap coefficient of at least 0.1. Once this is done, the resulting subgraph is ready for community detection.

Next, we create another GDS projection, this time including only user nodes and the weighted similarity edges between them. Once the projection is ready, we run the Louvain algorithm on the weighted graph to detect communities—groups of users that are more densely connected than would be expected in a random network. The algorithm assigns each user to a community, and a new attribute is created to store the *communityId* for each user node. This attribute is only set for users who are assigned to a community. Also, an example can be seen in this Figure 6

Identifying communities of users based on shared interests can significantly enhance the future potential of the "Let's Talk" app. By clustering users with similar passions, the app could recommend like-minded individuals to connect with, encouraging meaningful, face-to-face conversations in line with the platform's core mission. This could also lead to organizing interest-based events like

speed dating, sports hangouts, or casual meetups, making it easier for people to connect in person. On top of that, it could assist with more tailored conversation starters based on what others in the same community enjoy. And from a business perspective, knowing these communities would allow us to offer more targeted ads, which are typically more valuable than the general ads, helping us grow our revenue in a smart way.

7 Proof of Concept

7 Proof of Concept

To demonstrate the practical implementation and effectiveness of our graph-based question recommendation system, we created a proof of concept application. The application is accessible through our Website and is hosted on Hugging Face Spaces, to demonstrate the real-world applicability of "Let's Talk".

The proof of concept implements the core functionality described in our graph design, providing an intuitive web interface where users can select two individuals from our database and receive personalized question recommendations based on their common interests. The application successfully demonstrates how graph traversal algorithms can efficiently identify shared keywords and topics between users, subsequently recommending contextually relevant questions from multiple sources, including Stack Exchange, Reddit, Wikipedia, and trivia databases.

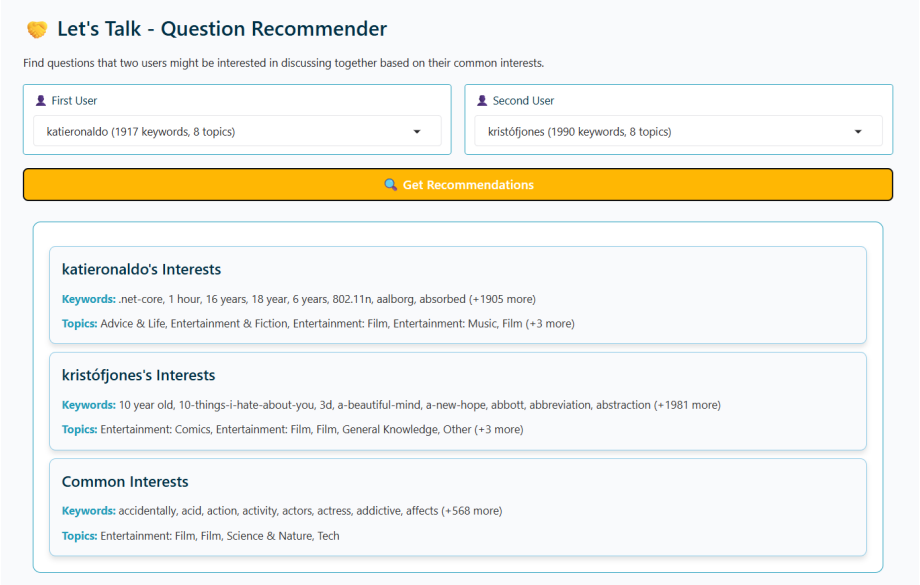

Figure 3: Interface - Common Interests

We use Gradio as the web framework, which provides rapid prototyping capabilities and seamless integration with our Python-based Neo4j backend. The choice of Gradio is particularly well-suited for data science applications, allowing us to focus on the core graph analytics rather than complex web development. The application connects directly to our Neo4j Aura cloud database, demonstrating the scalability and performance benefits of our graph-based approach in a production-like environment. Furthermore, hosting on Hugging Face gives us zero-downtime deployment, meaning users can continue using the older version of the app while the new version is being deployed. Hugging Face Spaces also allows us to easily embed this app into our landing page, as previously noted.

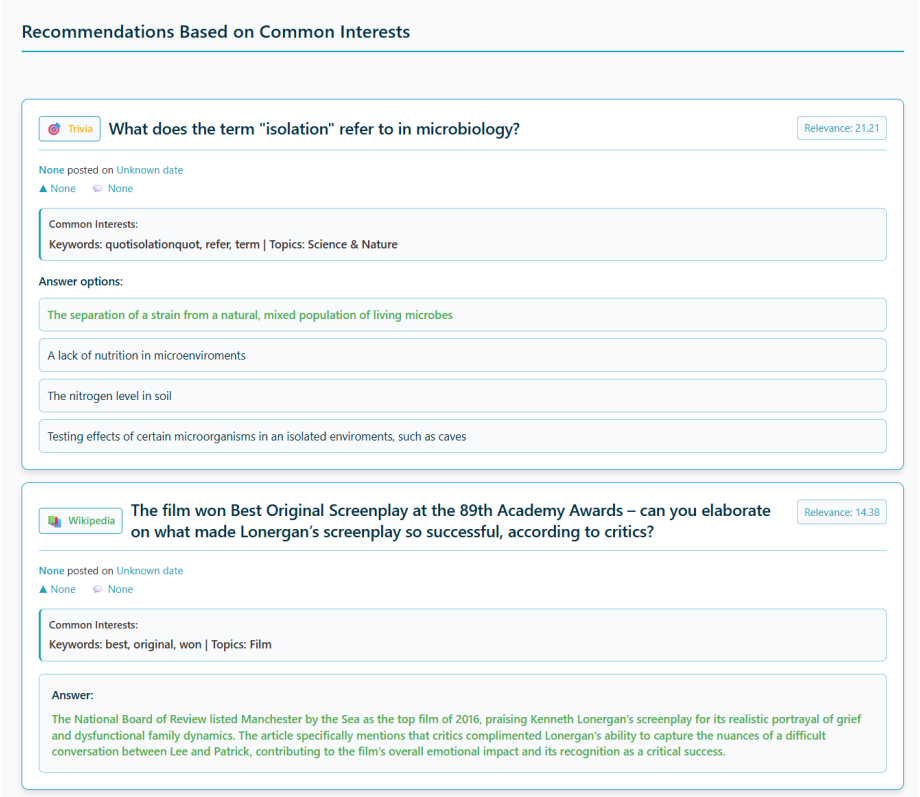

Figure 4: Interface - Recommendations

The user interface, as shown in Figure 3, presents a clean and intuitive design where users can select two individuals from dropdown menus populated with real users from our database. Upon selection, the system displays comprehensive interest analysis, showing individual user interests in keywords and topics, as well as their common interests. The recommendation engine then presents a

curated list of questions with relevance scores, source attribution, and rich metadata including author information, creation dates, and engagement metrics. The relevance score is computed using a multi-factor algorithm that combines interest matching weights with temporal and randomization components. The base score assigns higher weights (2.0) to interests shared between both users and lower weights (1.0) to interests matching only one user. This base score is then enhanced with temporal relevance that favors more recent questions, and a randomization factor `(0.6 + 0.8 * rand())` plus source-specific random boosts to ensure diverse recommendations across different question sources (Stack Exchange, Reddit, Wikipedia, Trivia) as can be seen in 4.

The proof of concept successfully validates our hypothesis that graph-based recommendation systems can provide superior personalization compared to traditional approaches. The application demonstrates real-time query performance, processing complex graph traversals and returning recommendations within seconds, even when analyzing users with hundreds of interests. This performance validation is crucial for our target use case of facilitating spontaneous conversations, where response time directly impacts user experience.

Furthermore, the implementation showcases the flexibility of our property graph model through its ability to handle diverse question types and sources seamlessly. Questions from Stack Exchange display with formatted code snippets and technical metadata, trivia questions present multiple-choice formats with highlighted correct answers, and Wikipedia-derived questions include comprehensive answers. This diversity demonstrates how our graph schema successfully accommodates the heterogeneous nature of conversational content while maintaining consistent relationship patterns for recommendation algorithms.

References

[1] Neo4j Inc., "Neo4j auradb – fully managed graph database in the cloud," 2025, accessed: 2025-05-27. [Online]. Available: https://neo4j.com/cloud/platform/aura-graph-database/faq/

[2] T. Ma, L. Guo, H. Yan, and L. Wang, "Cobind: quantitative analysis of the genomic overlaps," *Bioinformatics Advances*, vol. 3, no. 1, p. vbad104, 2023.

[3] V. D. Blondel, J.-L. Guillaume, R. Lambiotte, and E. Lefebvre, "Fast unfolding of communities in large networks," *Journal of statistical mechanics: theory and experiment*, vol. 2008, no. 10, p. P10008, 2008.

A Access to Source Code

A.1 GitGub

The source code for this project is available at the following GitHub repository: **GitHub Repository:** `https://github.com/UPCLetsTalk/BigGroup`. To test the application, use the following credentials:

- **Username:** testupc
- **Password:** Y7d!qR#3mVzP@9xTasdfsdfd

A.2 Hugging Face Spaces - Gradio

The source code for the prototype of this project is publicly accessible at:

- **Repository:** https://huggingface.co/spaces/NimaKL/LetsTalk/tree/main
- **Gradio App Code:** https://huggingface.co/spaces/NimaKL/LetsTalk/blob/main/app.py
- **Test the Prototype:** https://huggingface.co/spaces/NimaKL/LetsTalk/

A.3 Visualing graph analytics examples

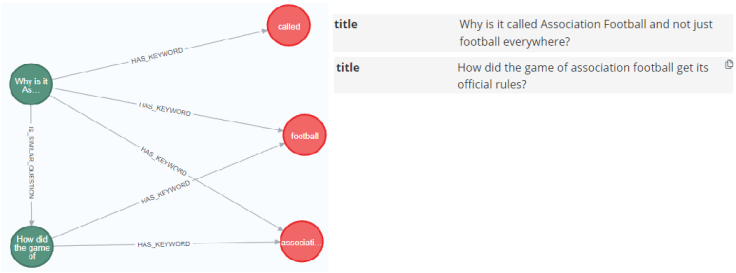

Figure 5: Question similarity example

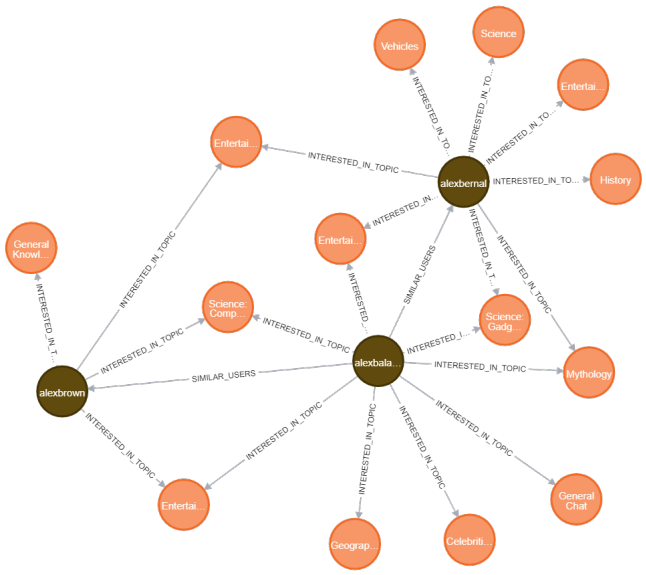

Figure 6: Community detection example (users of the same community)