SDM Project Report

Adrian Lim Patricio
Nishant Sushmakar
Oluwanifemi Favour Olajuyigbe

GitHub Repo Link

Universitat Politècnica de Catalunya

June 2025

Contents

1 Purpose and Role of Graphs in the Project

Tripify leverages graph-based solutions to enhance personalized travel recommendations and create a social community within the app. By modeling users, itineraries, and interactions (such as likes, comments, and favorites) as interconnected nodes and edges in a graph, we enable dynamic filtering and recommendation features. This approach allows users to efficiently discover relevant itineraries (filtered by user persona, city, starred, or commented-on status) and receive suggestions for similar trips based on their activity, recent bookings, and preferences. The project benefits by delivering highly relevant, community-driven recommendations, increasing user engagement, and streamlining the trip planning process for our user base.

1.1 Core Graph-Based Features

Tripify's graph model supports the following:

- **User–Itinerary Graph Modeling**: Each user, itinerary, and interaction (like, comment, star) is represented as a node, with edges denoting relationships such as **LIKES** or **REVIEWS**. This enables complex queries, such as finding all itineraries liked by users with similar travel interests or discovering trending trips in a specific city
- **Filtering and Discovery**: The graph structure supports advanced filtering: Users can search for itineraries by city, by those starred or commented on, or by user persona. Traversing the graph makes it efficient to surface relevant posts based on these parameters.
- **Recommendation Engine**: By analyzing the graph, the system can identify clusters of users with similar behaviors and preferences. Algorithms such as personalized PageRank or community detection are used to recommend itineraries that are popular among similar users or relevant to upcoming trips and favorites.
- **Social Network Integration**: Eventually, the graph-based model will support social features, such as following users, viewing friends' itineraries, or surfacing itineraries with high engagement. This fosters a sense of community and encourages content sharing and interaction.

In addition to the client-side features, **Tripify** also leverages graph data on the backend to analyze user activity patterns at scale. By examining the interconnected relationships between users, itineraries, and interactions, we can uncover valuable insights that support strategic business decisions.

1.2 Graph Analytics for Business Intelligence

We utilize backend graph analytics to visualize and monitor patterns such as user engagement, itinerary popularity, and community clusters. A dedicated dashboard enables real-time visualization of the graph structure, revealing trends including:

- Cities that attract the most active users.
- Locations that act as central connectors between popular travel destinations.
- Clusters of users who share similar interests or engage in frequent interactions.

1.3 Business Strategy Benefits

- **Marketing Optimization:** By identifying cities or regions with higher concentrations of active users, we can focus marketing efforts and promote more activities in those areas to increase user engagement.
- **Content Promotion:** Itineraries with high centrality (those that frequently link different user groups or cities) can be featured prominently in the app, enhancing their visibility and promoting further interaction.
- **Community Insights:** Detecting user clusters based on likes, comments, and shared preferences enables us to personalize recommendations, launch targeted campaigns, and foster a stronger community within the platform.

2 Graph Model Selection and Justification

The property graph model was chosen for our use case because of the following reasons

Schema Flexibility and Evolution

Property graphs excel in social networks because they handle schema evolution naturally. As itinerary platforms grow, they need to add new relationship types (e.g., friend requests, travel buddy connections, group memberships) and node properties (e.g., user preferences, trip ratings, accessibility needs) without rigid schema constraints. In contrast, knowledge graphs require more formal ontology management, making rapid feature development slower and more complex.

Complex Relationship Modeling

Social networks must model intricate relationships that property graphs handle well:

- **Multi-dimensional relationships:** A user can be connected to an itinerary as a creator, collaborator, or follower each with different permissions and metadata.
- **Temporal relationships:** Tracking when users joined trips, friendship timelines, and itinerary version history.
- **Weighted relationships:** Measuring friendship strength, travel compatibility scores, or recommendation relevance.

Knowledge graphs struggle with these nuanced, application specific relationship semantics due to their emphasis on semantic consistency and strict ontologies.

Query Performance for Social Operations

Property graph databases, such as Neo4j, are optimized for traversal heavy social queries:

- Finding mutual friends and their shared travel experiences
- Discovering travel recommendations through friend networks (2–3 hop queries)
- Identifying potential travel companions based on itinerary overlap
- Real-time friend of friend suggestions

These operations require efficient graph traversal algorithms, which property graph databases excel at. In contrast, knowledge graphs prioritize logical inference and semantic reasoning, which can hinder performance in operational query workloads.

Knowledge graphs are ideal for semantic reasoning and formal knowledge representation. However, social networks are fundamentally about dynamic relationships, interactions, and fast operational queries. Property graphs provide both the flexibility and performance required for itinerary-based social networking platforms.

3 Graph Schema and Design

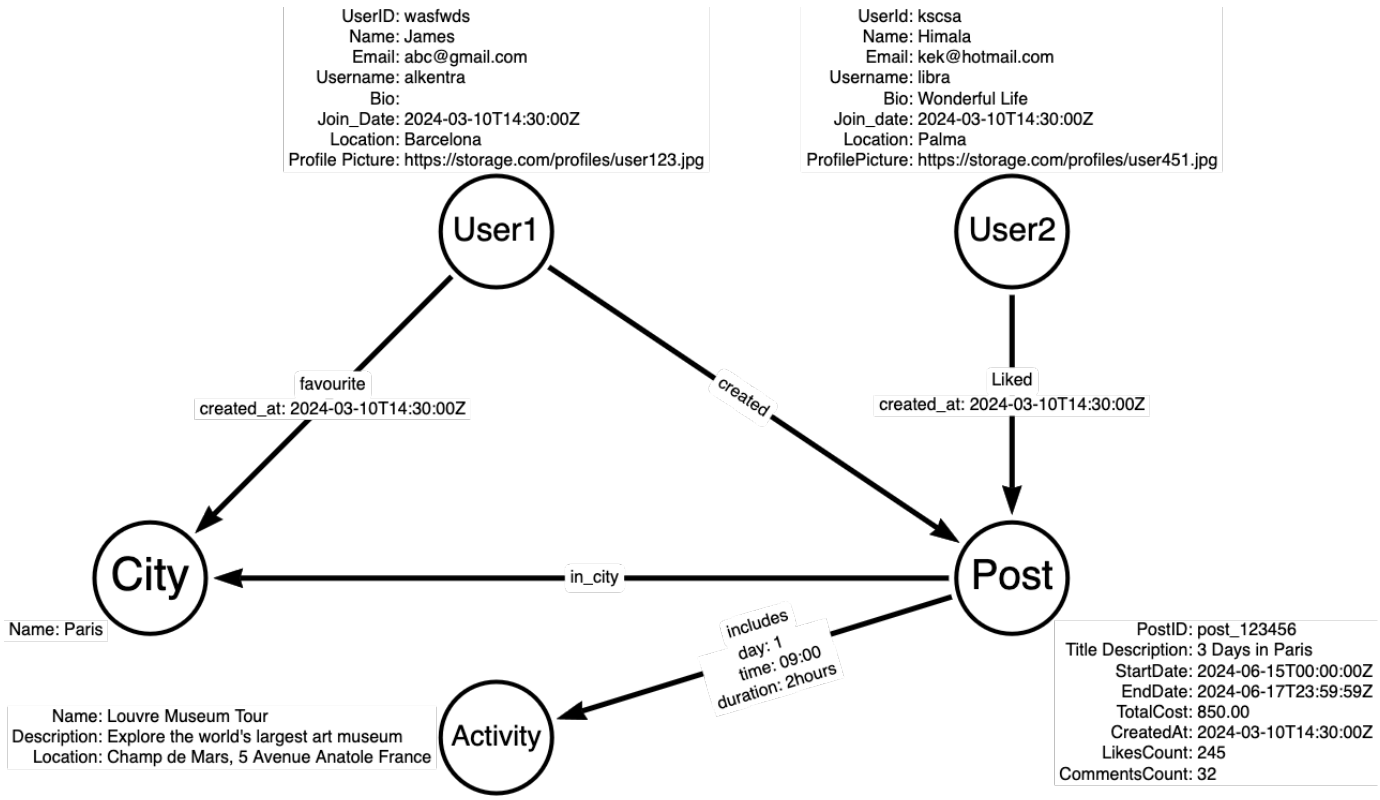

Figure 1: Graph Design

4 Graph Population Workflow

The population of the Neo4j graph in our system is a semi-automatic, multi-stage process that leverages a modern data lake architecture and orchestration tools. The main sources for the graph are synthetic datasets (users, posts, likes), which are generated, processed, and transformed through a series of well-defined zones before being ingested into Neo4j for graph analytics and application features.

4.1 Data Sources and Generation

The process starts with the generation of synthetic data that simulates real-world user interactions within a travel itinerary platform. Using Python and the `Faker` library, we create datasets representing users, their travel posts, and the likes they give to posts. Each user profile includes attributes such as name, email, favorite cities, and a biography. Posts are detailed travel itineraries, each associated with a user, a city, and a sequence of daily activities. The likes dataset captures the relationships between users and the posts they

interact with. This synthetic data is generated in a reproducible way and saved as CSV files, forming the raw input for the rest of the pipeline.

4.2 Data Lake Architecture and Processing Flow

Once the synthetic data is generated, it enters a multi-zone data lake architecture designed for scalability and data governance. The first stop is the **landing zone**, where the raw CSV files are uploaded without modification. This zone acts as a buffer and archive for all incoming data, preserving the original state for traceability.

From the landing zone, data is moved to the **trusted zone**. Here, the CSV files are converted into `Parquet` format, which offers more efficient storage and faster processing capabilities. Since the data is generated according to our schema and does not require cleaning, this step is focused on format optimization and partitioning, typically by date. This ensures that downstream processes can access the data quickly and reliably.

The final stage within the data lake is the **exploitation zone**. In this zone, data is transformed into a structure suitable for graph database ingestion. Using **Databricks** and **Apache Spark**, we process the Parquet files to create files that represent nodes (such as users and posts) and relationships (such as `POSTED` and `LIKED`) in the Neo4j graph. This transformation involves flattening nested structures, extracting relevant attributes, and ensuring the data matches the import requirements of Neo4j.

4.3 Orchestration and Automation

The movement and transformation of data across these zones are orchestrated using **Apache Airflow**. Airflow DAGs are responsible for checking the availability of required files, submitting Databricks jobs for data processing, and handling logging and error management. This orchestration ensures that the pipeline runs on schedule, can recover from failures, and provides traceability through log storage in **Google Cloud Storage**. The use of Airflow makes the entire process semi-automatic, requiring minimal manual intervention while supporting monitoring and control by data engineers. It operates on a daily basis, given that user data is expected to be collected relatively quickly.

4.4 Graph Population and Feedback Loop

After the data is prepared in the exploitation zone, it is uploaded to **Neo4j Aura**, our managed graph database service. The import process creates nodes for users, posts, cities, and activities, and establishes relationships such as `POSTED` (user to post), `LIKED` (user to post), and `IN_CITY` (post to city). This graph structure enables complex queries and analytics that support application features like itinerary recommendations and social networking.

A notable aspect of our design is the **feedback loop**. As users engage with the application and generate new data such as additional likes or posts this information can be collected and reintroduced into the trusted zone. This raw data is preserved and made available for future analysis and relationship building. As new behavioral patterns or insights are surfaced through dashboard analytics, this historical raw data can be re-examined to establish additional relationships or enrich the graph model with new types of connections.This approach ensures that the graph remains adaptable and continuously evolves based on both real-time user activity and retrospective business analysis. It empowers the team to:

- Build new relationships or node types as new insights emerge from dashboard analysis.
- Iterate on the graph schema without losing historical context, since raw data is always available for reprocessing.

- Drive business decisions such as targeted marketing, dynamic content promotion, and community segmentation by leveraging both current and historical user interactions.

By maintaining a robust pipeline for raw data ingestion and feedback, **Tripify** ensures that its graph database is not only current but also future-proof, supporting ongoing innovation and deeper analytics as the platform grows.

In summary, our approach to populating the graph is based on a robust, semi-automated pipeline that begins with synthetic data generation and flows through a multi-zone data lake. The process is orchestrated by **Airflow**, leverages **Databricks** for scalable data transformation, and culminates in the ingestion of well-structured data into **Neo4j**. This architecture not only supports the current needs of our application but is also flexible enough to accommodate real user data and evolving business requirements in the future.

5 Graph Usage and Analysis Strategy

While the previous step described how data flows into the graph database, we focus now on how the graph is used to deliver value to users. In our project, the main goal is to help users discover the most relevant travel itineraries based on their preferences and interactions, specifically their "likes" on posts. The graph structure enables advanced recommendation and filtering features that go beyond traditional relational queries.

5.1 Recommendation Model

The core analytic process is a recommendation engine that leverages the graph's structure. When a user logs in, the system examines their LIKED relationships to identify other users with similar tastes. The model uses collaborative filtering directly on the graph: it finds posts liked by users who have liked the same itineraries as the current user, but which the current user has not yet liked. For instance, we use a query to give recommended itineraries for users based on their favorite cities and liked posts.

```
# Get user's preferred cities from their preferences
preferences_query = """
MATCH (u:User {id: $user_id})-[:FAVORITE]->(c:City)
RETURN collect(c.id) as preferred_cities
"""
# First get posts that the user has liked
liked_posts_query = """
MATCH (u:User {id: $user_id})-[l:LIKED]->(p:Post)-[:IN]->(city:City)
WHERE city.id IN $preferred_cities
MATCH (p)-[:INCLUDES]->(a:Activity)
WITH p, city, collect(a) as activities, true as is_liked, 1 as priority
RETURN p, city, activities, is_liked, priority
"""
# Then get posts from creators of liked posts
creator_posts_query = """
MATCH (u:User {id: $user_id})-[l:LIKED]->(p1:Post)<-[:CREATED]-(creator:User)
MATCH (creator)-[:CREATED]->(p2:Post)-[:IN]->(city:City)
WHERE city.id IN $preferred_cities
AND p2.id <> p1.id
MATCH (p2)-[:INCLUDES]->(a:Activity)
WITH p2 as p, city, collect(a) as activities, false as is_liked, 2 as priority
RETURN p, city, activities, is_liked, priority
"""
# Finally get other posts in preferred cities
other_posts_query = """
MATCH (p:Post)-[:IN]->(city:City)
WHERE city.id IN $preferred_cities
```

```
AND NOT EXISTS((:User {id: $user_id})-[:LIKED]->(p))
AND NOT EXISTS((:User {id: $user_id})-[:LIKED]->(:Post)<-[:CREATED]-(:User)-[:CREATED
    ]->(p))
MATCH (p)-[:INCLUDES]->(a:Activity)
WITH p, city, collect(a) as activities, false as is_liked, 3 as priority
RETURN p, city, activities, is_liked, priority
"""
# Combine all results
combined_query = """
CALL {
    """ + liked_posts_query + """
    UNION
    """ + creator_posts_query + """
    UNION
    """ + other_posts_query + """
}
RETURN p, city, activities, is_liked
ORDER BY priority, p.created_at DESC
LIMIT 20
"""
```

5.2 Filtering and Exploration

In addition to recommendations, users can filter itineraries by various attributes—such as city, activity type, or trip duration—using the graph's structure. This is achieved through Cypher queries that traverse the relevant relationships and apply attribute-based filters, allowing users to quickly discover itineraries that match their specific preferences.

5.3 Validation

To validate the recommendation and filtering processes, we use both synthetic and behavioral metrics. With synthetic data, we can embed known patterns (such as clusters of users with similar preferences) and check if the graph analytics successfully recover these patterns. In a real-world setting, validation would include monitoring user engagement with recommended itineraries, such as click-through rates, likes, and saves, to refine and improve the recommendation algorithms.

Beyond engagement metrics, we leverage a **Streamlit** dashboard to visualize and analyze user activity and the graph structure in real time. Using `Cypher` queries, we extract relevant data about nodes and relationships from **Neo4j**. The graph data is then further analyzed with the `NetworkX` library to compute clustering, shortest paths, and centrality measures. These analytics enable us to:

- **Verify clusters:** Ensure that users with similar preferences are properly grouped together.
- **Assess connectivity:** Use shortest path analysis to understand how easily recommendations can propagate between users, posts, and cities.
- **Identify influence:** Apply centrality measures to highlight influential itineraries, validating whether the most central nodes correspond to the most popular or engaging content.

All these insights are presented in the Streamlit dashboard, providing an interactive and intuitive interface for both technical and business stakeholders. This allows for effective monitoring of system performance, spotting trends, and detecting anomalies.

This comprehensive approach ensures that our recommendation engine not only delivers relevant results but also aligns with real user behavior and community dynamics, supporting continuous improvement and strategic decision-making.

6 Metadata Generation and Management

In our recommender system, metadata is generated during user interactions such as liking posts and marking favorite cities. This metadata includes timestamps, user preferences, and interaction details, which are captured by the system as users engage with the platform.

- **Metadata Generation:** Whenever a user performs an action (e.g., likes a post or sets a favorite city), the system automatically generates metadata describing that action. This process is embedded within the recommendation flow and ensures that relevant user interactions are systematically recorded.
- **Metadata Storage:** The generated metadata is stored in the **Neo4j Aura** graph database as relationships (e.g., `LIKED`, `FAVORITE`). This storage strategy ensures that all relevant user behavior and preferences are persistently recorded and can be efficiently queried during recommendation generation.
- **Metadata Reuse:** Stored metadata is reused by the recommendation engine to enhance personalization. For instance, user likes and favorites are utilized in collaborative filtering algorithms to identify similar users and recommend relevant itineraries. This reuse improves the relevance and accuracy of the recommendations provided to each user.

6.1 BPMN-Based Process Description

Start Event User logs in or starts a session.

User Task User interacts with content (likes, favorites, applies filters).

System Task System generates metadata from the interaction.

Data Store Metadata is stored in **Neo4j Aura**.

System Task System retrieves stored metadata for personalized recommendation.

System Task System generates and displays recommendations to the user.

End Event User session ends.

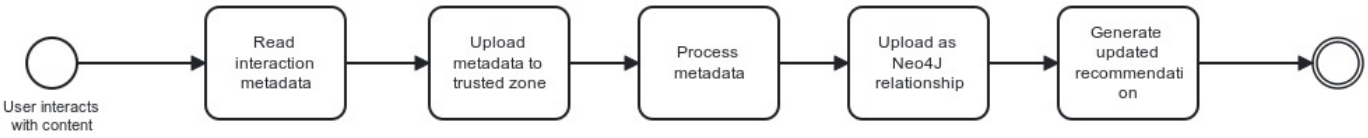

Figure 2: BPMN

7 Proof of Concept Implementation

For our proof of concept, we designed a recommendation engine that leverages the strengths of graph technology to deliver real-time, personalized suggestions to users. Our architecture uses Neo4j Aura, a fully managed cloud-based graph database. We chose Neo4j Aura because it is purpose-built for handling highly interconnected data, which is fundamental for recommendation systems that rely on understanding complex relationships between users, items, and their interactions. Unlike traditional relational databases, Neo4j's native graph model allows us to efficiently traverse and query these relationships without the overhead of complicated joins, making it ideal for scenarios where quick, dynamic recommendations are needed. Another key reason for selecting Neo4j Aura is its scalability and performance. As a cloud-native platform, it offers high availability and elastic scalability, ensuring that our system can handle growing

amounts of data and user activity without compromising on speed. The property graph model also provides the flexibility to evolve our data schema as business needs change, making it easy to introduce new types of nodes or relationships without major restructuring. Additionally, Neo4j Aura's management console simplifies database administration, data visualization, and collaboration, which accelerates development and troubleshooting.

To connect our database with client applications, we built our backend using Flask, a lightweight and flexible Python web framework. Flask was chosen for its simplicity and ease of integration with Neo4j with the help of Python drivers. This setup allows us to rapidly develop RESTful APIs that expose recommendation results to front-end clients or other services. Flask's modular nature also means we can easily extend our backend with features like authentication or logging as the project grows. For querying the graph, we use Cypher, Neo4j's expressive query language. Cypher is specifically designed for pattern matching and graph traversal, which makes it straightforward to implement both collaborative and content-based recommendation strategies. Its human-readable syntax also aids in debugging and maintaining transparency in how recommendations are generated, which is important for both development and user trust.

This architecture not only supports our current needs but also positions us well for future expansion. Neo4j's compatibility with modern data platforms and its robust security features ensure that our system is ready for production use and can adapt as requirements evolve. By combining Neo4j AuraDB, Flask, and Cypher, we demonstrate an end-to-end solution that is robust, scalable, and perfectly suited for real-time, graph-based recommendation scenarios. The itinerary network page is shown below:

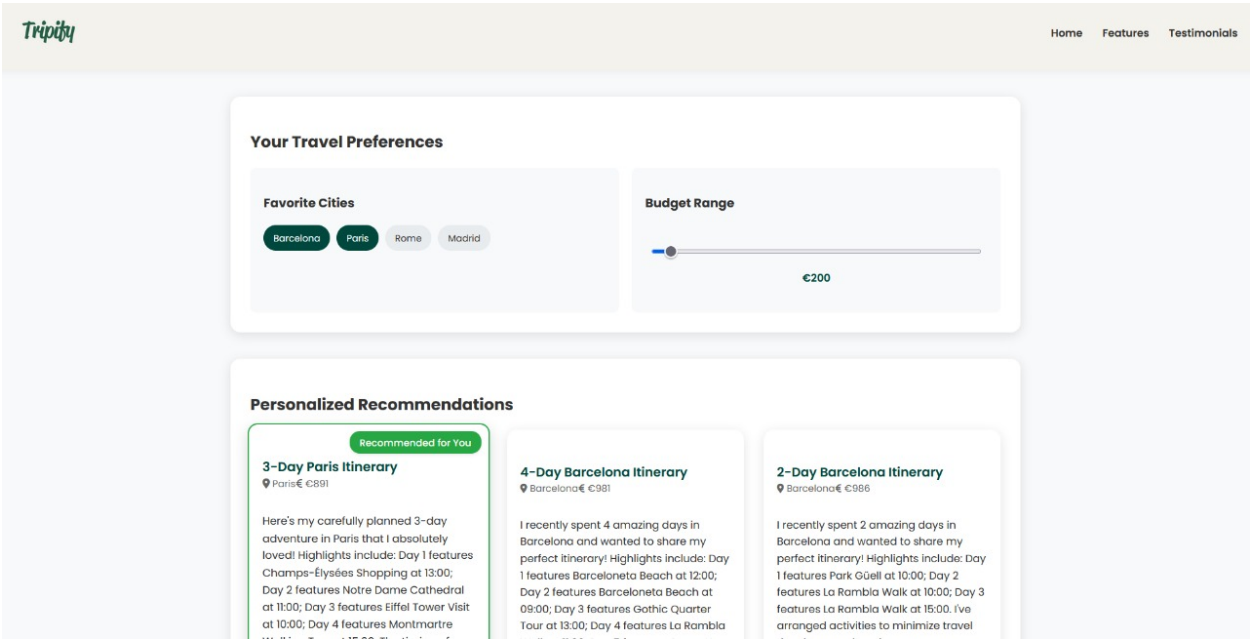

Figure 3: Itinerary Network Page

After implementing the itinerary network and recommendation engine, we also developed a dedicated data analytics page using Streamlit, Cypher, and Networkx. This analytics dashboard provides real-time, interactive visualizations of user activity patterns and graph structure, supporting both technical validation and business decision-making. We chose Streamlit for its rapid prototyping capabilities and ease of use in building interactive dashboards with Python. It allows stakeholders to intuitively explore key metrics and patterns without requiring deep technical knowledge.

For data extraction, we also rely on Cypher to pull relevant subgraphs and user interaction data directly from the graph database, but on top of this, we use the Networkx library to perform advanced graph

analytics, such as clustering, shortest path analysis, and centrality measures. These analyses help us identify influential users or itineraries, understand how information and recommendations flow through the network, and detect user communities or clusters. Such insights are crucial for refining recommendation algorithms, targeting marketing efforts, and optimizing content promotion.

This combination of tools Streamlit for visualization, Cypher for querying, and Networkx for analytics was chosen to provide a flexible, scalable, and transparent analytics workflow. It enables us to both validate the effectiveness of our recommendation engine and continuously improve the platform based on actionable, data driven insights.

Some samples of the dashboard data are shown below:

You can check out the dashboard here: https://tripify-dashboard.streamlit.app/.

Note: It might take a little while to load, so please be patient as the app can be a bit slow initially.

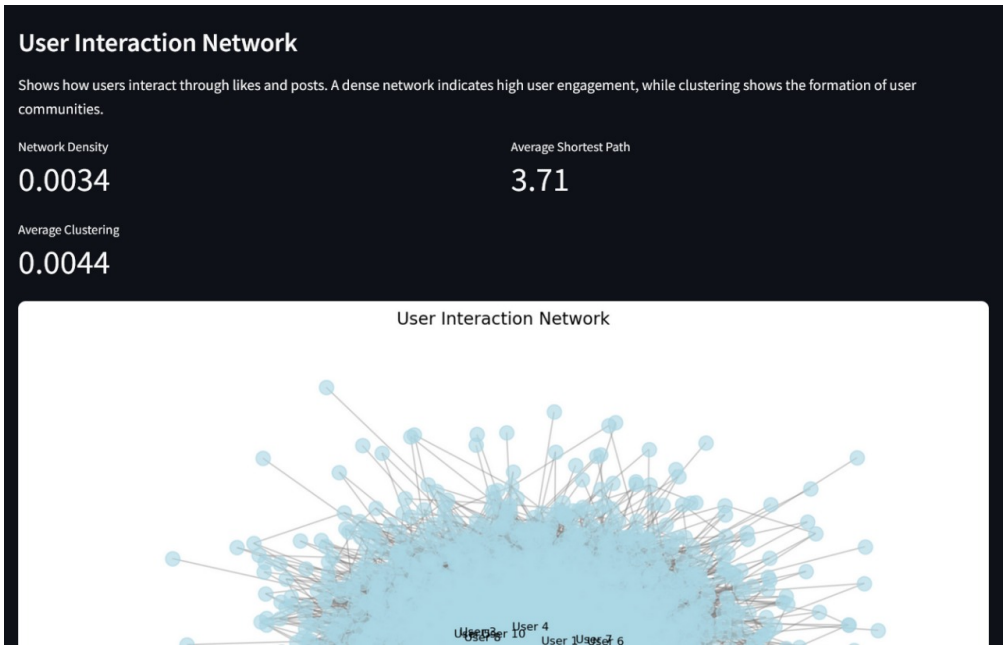

Figure 4: User Interaction Network

Figure 5: Most Popular Cities

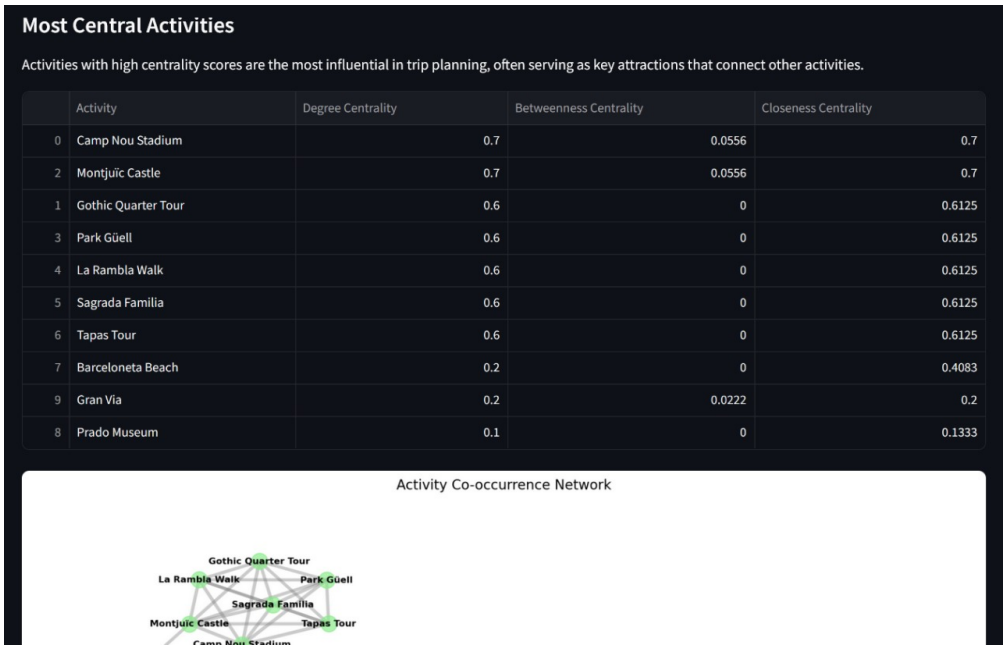

Figure 6: Most Central Activities

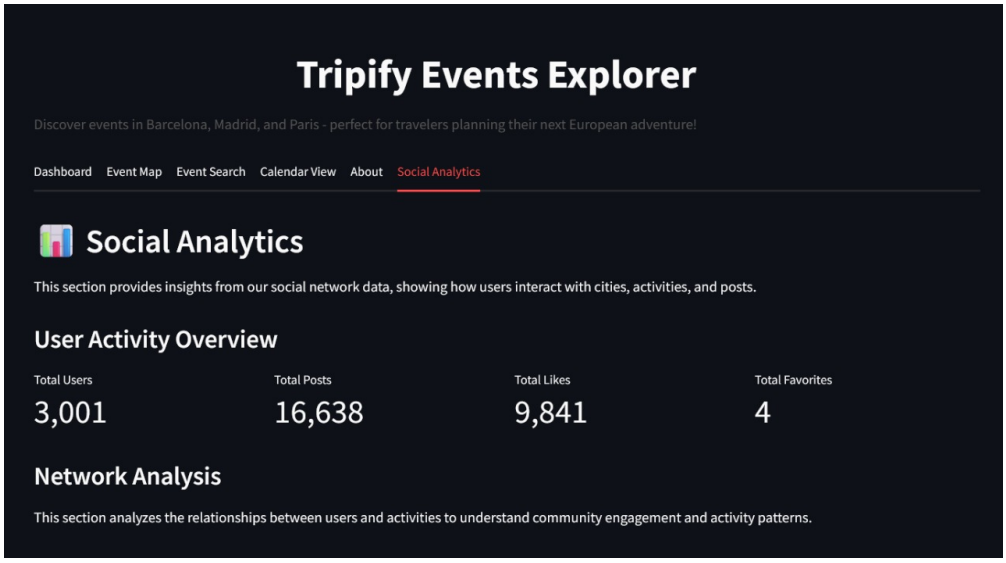

Figure 7: Social Analytics