Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Автоматизована система вияву аномальної активності у соціальній мережі Telegram

Текстова частина до кваліфікаційної роботи

за спеціальністю «Інженерія програмного забезпечення» - 121

Керівник кваліфікаційної роботи

Асистент

Бабич Т.А.

_____ (Підпис)

“___” _____ 2023 року

Виконав студент

ІПЗ-4 Охріменко М. С.

“___” _____ 2023 року

Київ 2023

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра математики факультету інформатики

ЗАТВЕРДЖУЮ

Асистент

_____ Бабич Т.А.

„_____” _____ 2022 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на кваліфікаційну роботу

студенту Охріменко Михайлу Сергійовичу

факультету інформатики 4 курсу бакалаврської програми

ТЕМА: **Автоматизована система вияву аномальної активності у соціальній мережі Telegram**

Зміст ТЧ до кваліфікаційної роботи:

Індивідуальне завдання

Календарний план

Анотація

Вступ

Дослідження та літератури

Розробка методології

Програмна реалізація MVP

Результати

Висновки

Використана література

Додатки

Дата видачі „___” _____ 202 р.

Керівник_____

(підпис)

Завдання отримав_____

(підпис)

Календарний план виконання роботи

№	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Визначення теми кваліфікаційної роботи	Жовтень 2022	
2.	Отримання завдання на кваліфікаційну роботу	Жовтень 2022	
3.	Огляд літератури	Листопад - Грудень 2022	
4.	Написання теоретичної частини кваліфікаційної роботи	Січень 2022	
5.	Написання програмної реалізації	Лютий 2023	
6.	Написання практичної частини кваліфікаційної роботи	Березень - Квітень 2023	
7.	Захист кваліфікаційної роботи	Травень 2023	

Студент _____

Керівник _____ “_____”_____ 2022

Анотація

Дане дослідження присвячено розробці ефективної системи для виявлення активності ботів та тролів у месенджері Telegram. У роботі було створено систему, що складається з різних підсистем, кожна з яких виконує специфічні задачі: від отримання даних до аналізу та визначення шаблонів. В основу системи було покладено принципи модульності, масштабованості та надійності, що дозволило забезпечити гнучкість і адаптивність у процесі розробки.

За допомогою Python, протоколу MTProto та графової бази даних Neo4j, було створено систему, що ефективно виявляє аномальну активність. Проте, попри успішність поточної версії, система має потенціал для подальшого розвитку і вдосконалення, зокрема за допомогою інтеграції машинного навчання.

Загалом, результати цього дослідження представляють собою значний крок у боротьбі з дезінформацією та шкідливою активністю на соціальних медіа-платформах, втілюючи нові методи виявлення тролів та ботів на Telegram.

Table of Contents

I. Introduction

A. Background on Telegram and its Use

Telegram is a cloud-based instant messaging service that was introduced on August 14, 2013. It provides a variety of communication options, including end-to-end encrypted chats, voice calls, video calls, and the ability to transmit text, images, and videos, among others. Telegram's most popular features are its channels, groups, and public conversations, which allow users to broadcast messages to large audiences or participate in community discussions files. [1]

By 2023, monthly Telegram usage had surpassed 700 million. The company has expanded rapidly because it offers a product that is highly sought after by consumers all across the world, especially in Ukraine. Due to its widespread adoption and utility, Telegram has also become a breeding ground for malicious activity like the dissemination of false information, PSYOPS, and propaganda by bots and trolls.

On Telegram, bots and abusers pose a significant challenge. They are capable of manipulating public opinion and spreading half-truths and deceptive information. This could impact political, economic, and other aspects of people's lives. This has led to growing concerns regarding the effect these activities have on the integrity of public discourse and democratic processes.

In response to these obstacles, there has been a growing demand for automated systems capable of detecting suspicious activity in social networks, such as Telegram. These systems may employ various techniques, including Machine Learning and data analysis. This research intends to contribute to this field by developing and structuring methodologies and creating the minimum viable product for a system that can retrieve, detect, and block potential bots.

B. Definition of the Problem: Suspicious Activity by Bots and Trolls

In today's digital era, social media platforms are being used as a frontline in information wars. Threatening the credibility of online discussions are malevolent actors such as bots and trolls. Disinformation can be spread at an unparalleled volume and speed via bots, or automated accounts designed to do specific tasks. Trolls, on the other hand, are human-operated accounts that engage in disruptive conduct such posting angry words with the intention of provocation, misinformation, or manipulation of certain demographics.

These actors can distort public opinion, spread disinformation and influence political, economic and other social processes in a way needed to their owners. The tactics can be very different and depend on a group which is attacking. It may be subtle and sophisticated which for example uses existing tension in society for manipulation or very simple and artless messages which usually are spread automatically or wrote by inexperienced attackers. These actors adapt to changing environments, features and algorithms of social networks to maximize their rogue impact [2].

The problem is further compounded with a fact that Telegram is a messenger, despite of traditional social networks as Twitter or Facebook. It means that there are very few metrics which we could use for identifying rogue activity. With its anonymity and privacy Telegram offers a fertile ground for operation of bots and trolls. Its strong commitment to privacy and security is beneficial for users but also is exploited by malicious actors to evade detection and analysis. [3]

The challenge, therefore, is to develop effective methodologies for detecting such rogue activity, understand their tactics, targets and impacts. This requires a multidisciplinary approach that combines technical solutions and data analysis with involvement of social sciences and cybersecurity.

C. Research Objectives: Pattern Recognition and Database Creation

This research has two main goals. Firstly, it aims to develop new methodologies that are suitable for Telegram and encounter its specifics as Messenger platform. These methodologies will focus on effectively detecting and identifying suspicious troll activity with Telegram communication methods.

The second aim of this research is to develop MVP of an automated system for detecting Telegram bots. The MVP will serve as an example of practical implementation of the developed methodologies, providing a functional tool that can be used to identify and mark suspicious bot accounts which then may be blocked in specific chats.

This research will contribute to fighting against spreading disinformation. It will provide valuable insights and tools for researchers, administrators of Telegram channels and policymakers, helping to make online discourse more constructive.

E. Definitions

MVP - Minimum Viable Product

ML - Machine Learning

PSYOPS - Psychological operations

II. Literature Review

A. Overview of Bots and Trolls on social media

Social media platforms have become a popular place for bots and trolls to operate. The ability to remain anonymous, quickly reach large audiences, and the lack of strict rules from government bodies have made these platforms a hotbed for mass disinformation campaigns. However, first bots and trolls didn't use in such mass organized activities.

Since the early days of the Internet, bots have been used for distributing advertisement. Their popularity has risen with the proliferation of social media platforms and public social networks. These bots can generate or reuse messages automatically. In this regard, they bear a striking resemblance to spambots in terms of their capacity to spread messages among large user groups. [9] The term "troll" according to Phillips [10][6] refers to individuals who purposefully engages in disruptive behavior within online environments. Trolls seek to disrupt constructive discussions, increase tension in some social groups by posting inflammatory and offensive content. Their motivation often arises from a desire to take control over other people and derive satisfaction from manipulating emotional state of unsuspecting users.

Although today bots and trolls continue to perform tasks they made 20 years ago, they have also evolved into tools employed by various malicious governmental and non-governmental groups. [11][12] These groups use them for the purpose of large-scale manipulation of public opinions to increase tension inside a society, cause disturbance in political, economic and other processes within a country or organization. [12] A troll account may be operated by an individual, a group of individuals or even software [13]. The motivations driving these individuals extend beyond a mere desire for control over others' opinions. Trolling has transformed into a profession, where workers receive a stable salary for bullying and manipulation as part of their regular 9-to-5 work routine. [14]

B. Case Studies: Prigozhin's Troll Factories and the Internet Research Agency

In the early 2010s, Prigozhin began setting up troll factories. During this time, the popularity of social media sites and their sway over public opinion skyrocketed [14][15]. The IRA has large and complex operations. They use various strategies, such as spreading disinformation through automated accounts and new channels, creating and managing networks of social media accounts, gaming social media algorithms to increase the visibility of specific content, and influencing public opinion [16].

Prigozhin's troll factories can be dated back to the first years of this decade. There was a dramatic increase in the impact of social media on public opinion during this time period [14]. The IRA's methods are complex and widespread. One example is the use of automated accounts and news channels to disseminate misleading or false information and manipulate public opinion. Other methods include creating and managing networks of social media accounts, manipulating social media algorithms to boost the visibility of certain content, and so on [16].

Their activities are not limited to any platform or country. They have been found to operate on major social media platforms such as Facebook and Twitter, and their influence campaigns have targeted audiences in the United States, the United Kingdom, Ukraine and other countries [6][12][14].

The IRA has been implicated in numerous cases of manipulating public opinion, including the U.S. presidential election and various political processes in Ukraine. In the USA, it is considered that the IRA, with its botnets, played a significant role in the 2016 presidential election. According to a report by the U.S. Special Counsel's investigation, the IRA created numerous fake social media account that were connected to tens of millions of users. These accounts were used to spread doubts and increase division and tension among supporters of different political parties [17]. In Ukraine, the IRA has employed similar tactics to influence public opinion regarding integration with the European Union. Agency reportedly created a network of bots and fake social media accounts to spread pro-Russian propaganda and discredit pro-European point of view. This operation was part of a broader Russian strategy aimed at undermining Ukraine's integration efforts [18].

While the IRA's actions were directly focused on political processes, they have also been involved in disinformation campaigns related to public health. In the United States, the IRA has spread disinformation about measles and COVID-19 pandemic on Twitter. They were reported to spread panic and misinformation, further destabilizing political processes and public trust in institutions. This tactic of exploiting crises shows the IRA's opportunistic approach to information warfare [8][19][20].

C. Case Studies: Where is Stepan Mazura, research of Facebook troll network

The case of Stepan Mazura, as discovered by Texty.org.ua [13], provides a crucial example of the sophisticated nature of troll operations and their tactics used on social media platforms, particularly Facebook, today.

Stepan Mazura was a digital persona that portrayed itself as a Ukrainian patriot. However, behind this user was a former combatant of the Russian “hybrid army” at Donbass, Sergei Zhuk. This person and its network actively called for the forceful overthrow of the Ukrainian government. This network was discovered to be composed of around 2000 Facebook accounts connected with certain groups linked to Mazura, engaging in the dissemination of calls for riots and protests under the guise of patriotic slogans.

The concept of creating a network like this is not new. Authors mentioned that historically it can be traced back to tactics used by the Soviet secret police know as GPU (the State Political Directorate) during the period 1921-1926. The GPU crafted a fake underground organization known as the “Monarchic Organization of Central Russia” (MOCR) to attract anti-communists individuals. This operation known as Operation Trust, was a successful endeavor that used double agent and misinformation to manipulate anti-Soviet immigrants not to organize terrorist attacks in the Soviet Russia.

The construction of troll networks is a modern adaptation of GPU’s methods. These networks primarily consist of trolls which can either belong to real humans or be automated bots. Generally, a troll is 80% of time is a program and 20% -- a human. To increase influence of troll accounts automatic software, send friend requests, make comments which often post some nonsense, exchange likes with similar programs, and tries to connect with popular bloggers. These actions increase account weight in recommendation algorithms and make accounts visible for a wider audience. Then, at critical junctures when a specific topic needs to be amplified, real people who can communicate more effectively with the audience take control over these accounts.

The targets of these networks are often patriotic individuals who are deeply concerned about the state of their country. In today’s post-factual world, emotional component has more influence than pure facts. Trolls exploit this by appealing to emotions and conspiracy theories. It has been observed that people who mistrust traditional media are more susceptible to disinformation.

The tactics employed by Russian trolls in Ukraine is similar to those used in the USA. They often post content that appeals to nationalistic sentiments in the same time insulting politicians. These tactics aim to influence the policy of other governments, undermine confidence in the country’s leaders and institutions, destroy its relationships

with other nations or discredit and weaken governmental and non-governmental opponents.

D. Existing Approaches to Detect Bots and Trolls

As the online communities and social media platforms continue to grow, it stands a question how mitigate malicious behavior of bots and trolls. Bots and trolls pose a significant threat to the authenticity, integrity and safety of online environments, as they disrupt discussions, propagate misinformation, and manipulate public opinion. Consequently, researchers and practitioners have developed various methods for detection and prevention these actions. This section aims to explore the existing methodologies and techniques employed in the detection of bots and trolls in various media platforms, which encompass a range of technological, algorithmic and linguistic strategies.

- User Feedback Reports
  User feedback reports are a straightforward and intuitive method for detecting bots and trolls. Users who encounter suspicious activities or behaviors can report them to the platform administrators, providing a first line of defense against these malicious actors. This method is particularly effective in identifying explicit instances of bots' activity. However, this approach has many limitations. It heavily relies on the vigilance and understanding of users, who may not always be able to accurately identify more sophisticated malicious behavior. Furthermore, this method can be time-consuming and labor-intensive for administrators of platforms, who must review and verify user reports.
- Machine Learning
  Machine learning allows for an automated and more complex method of identifying bots and trolls. These instruments can sift through mountains of data in search of telltale signs of intrusion. In order to tell robots from people, for instance, supervised learning algorithms can be educated with the help of labeled data. These algorithms can gather information from previous botnet instances to better detect and prevent future ones. Nonetheless, these methods are only as good as the quantity and completeness of their training data. The model may produce false-positive findings [21] if the training data does not properly reflect the harmful behavior.
- Linguistic Analysis

This approach involves analyzing the language, tone, and style of posts to identify potential bot or troll activity. Bots, for example, often use repetitive phrases or exhibit unusual syntax that can be detected through linguistic analysis. Trolls, on the other hand, may use inflammatory or provocative language to provoke reactions from other users. This approach is limited by language barriers and the ability of and trolls to adapt their language to evade detection [22].

- Graph-Theoretic Model
  This technique offers a more mathematically abstract means of identifying harmful behavior. Users are represented as nodes in a network and their interactions between them as edges in these models. In these graphs, malicious actors frequently display behavior that is distinct from that of typical users. For instance, if their interaction levels are low but their connection counts are high, it's a red flag that they don't really engage with the community. Graph-theoretic methods are able to effectively detect bots and trolls since they can recognize these patterns. These models, however, necessitate substantial processing resources, making them impractical for small media outlets [13].
- Holistic Approach
  The holistic approach integrates multiple methods to provide a more comprehensive and accurate detections. This method may combine all of the previously mentioned methods. By integrating different techniques, the holistic approach can overcome the limitations of each particular method and provide a more accurate understanding of suspicious activity. On the other hand, this approach is complex, resource-intensive and requires a lot of expertise to implement effectively.
- Multi-Feature Analysis
  This type involves analyzing multiple features of user behavior, such as posting frequency, content and network connections. By considering a wide range of features, this method provides more specific understanding of bots and trolls activity. However, the method is limited to the quality of data. If data does not completely represent bot and troll behavior or amount of data is limited, the approach won’t be able to effectively detect suspicious activity [23].
- Detection of Fickle Trolls
  Fickle trolls are the accounts which frequently change their behavior and identity to evade detection and maximize disruptive impact. They may change their

posting style, language, or the topics they discuss to seem like a regular user. Detecting these trolls requires advanced techniques that can adapt to changing patterns of behavior. One such technique is continuous tracking changes in a user's identity and the topics they post about. By identifying users who frequently change their behavior in these ways, it is possible to detect fickle trolls. This approach may not be practical for all platforms and requires a lot of expertise and understanding of tactics of malicious actors [24].

III. Methodology

A. Data Collection

The data collection process for the study was extensive. The primary focus was on Telegram public channels and groups, a tool that has witnessed tremendous growth in usage in recent years. Since the start of the Russo-Ukrainian War, public channels and groups have been a quick way to transmit information. However, they quickly became a breeding ground for bot and troll disinformation campaigns, notably in comments.

We gathered this information using Telegram's API, a sophisticated tool that provided us with access to a variety of information. Messages, user IDs, timestamps, and other pertinent metadata were included. The API enabled us to collect data for the study in a methodical and efficient manner.

During the data collecting period, we were able to collect information from a total of 895,513 accounts. These identities were responsible for 206,295 posts and 14,926,289 comments. This information was gathered from a total of 70 public discussions with open comments.

The collected data represents period of half of year. This timeframe was chosen to provide a balance between obtaining a large enough dataset for meaningful analysis and ensuring that the data was recent and relevant.

The channels and groups from which we collected data were carefully selected based on several criteria. These included their popularity, activity level, and diversity of topics, government channels and government-connected ones. By selecting channels and groups that met these criteria, we were able to capture a wide range of bot and troll behaviors. This was crucial in ensuring the validity and reliability of our research.

While there are several tools available for detecting bots, such as Combot Anti-Spam, which implements various ways to detect automated software, the research focused on detecting troll activity. Trolls, unlike bots, are usually human-operated so their behavior has more complex patterns. Consequently, the detection became even more complex than automated software.

B. Database Creation and Usage

The collected data was organized and stored in a graph database. The choice of a graph database was deliberate and based on several factors. Graph databases are particularly

well-suited for storing and analyzing relationship-oriented data. They provide a flexible, efficient, and more intuitive way to work with data that has complex interconnections.

In the context of the research, the graph database was used to store data about Telegram users, their messages, and the channels or groups they were part of. Each user, message, and channel or group was represented as a node in the graph. Relationships between these nodes, such as a user posting a comment or a post being placed in a channel, were represented as edges connecting the nodes. This structure allowed us to easily query and analyze the data in a way that took into account the relationships between different entities.

The creation of the database was not a one-time process but rather an ongoing one that spanned the entire duration of the research. As new data was collected, it was added to the database. This ensured that the analysis was always based on the most recent and relevant data.

The database was utilized not just to store data, but also to analyze it. Because of its powerful querying capabilities, the graph database was utilized to find patterns and trends in the data. For example, all messages written by a specific user or all users active in a specific channel might be easily identified. Relationships between distinct entities, such as individuals who submitted similar messages or channels with similar user activity patterns, could also be discovered.

It is crucial to emphasize, however, that while the graph database was a formidable research tool, it also had limits. Because the data was mostly text messages from a messenger service, there were few linkages between entities. For example, it lacked information on who was friends with whom and what groups a user belonged to. As a result, the sorts of relationships that could be represented in the graph database were limited.

Despite these drawbacks, the graph database proved to be a useful research tool. It offers a versatile and efficient method of storing and analyzing data. As new forms of data become available in future research, it is projected that the utility of graph databases in this sort of research would only grow.

C. Data Analysis

The subsequent phase of the methodology was data analysis. This process was complex and intricate, necessitating the application of both quantitative and qualitative

methods. The analysis of data from a platform like Telegram, a messenger service, presents unique challenges due to the nature of the available data. Unlike social networking platforms, messenger services like Telegram do not provide a wealth of user information such as friends, groups, or fields of interest. This necessitates a more nuanced approach to data analysis, focusing primarily on user activity and content.

Quantitative analysis was employed to distinguish patterns and trends within the data. This form of analysis is particularly useful in identifying trolls, who, unlike bots, are human-operated accounts. Trolls often exhibit distinct behavioral patterns that can be identified through statistical methods. For instance, one of the factors we considered was the frequency of messages. While bots often post messages at a much higher frequency than regular users, trolls may exhibit a more nonsystematic pattern of activity, frequently aligned with particular events or discussions.

Another unique aspect of troll behavior is that a single troll account may be operated by multiple individuals. This can result in a single account exhibiting a wide range of behaviors and activity patterns, making it more challenging to identify and track. However, this can also be a potential indicator of a troll account. If an account exhibits significant shifts in activity patterns or language use, it may suggest that the account is being operated by multiple individuals.

One of the key techniques that was used in qualitative analysis was semantic similarity analysis. This involves comparing the semantic content of different messages to identify similarities. Trolls often reuse the same phrases or talking points across different conversations or channels. By identifying these recurring patterns, we were able to flag potential troll accounts for further investigation.

However, it's important to note that semantic similarity analysis is not without its challenges. Trolls often adapt their language and tactics in response to detection efforts, which means that the patterns we identify may change over time. Therefore, our analysis methods needed to be flexible and adaptable, capable of evolving in response to changing troll behavior.

D. Pattern Recognition

After analyzing the data, there were identified patterns that were indicative of bot and troll activity.

For both, these patterns included high frequency of messages, posting at odd hours, and lack of interaction with other users. Bots often post messages in a repetitive manner, using similar phrases or links. They also tend to ignore responses from other users, focusing instead on disseminating their own content.

For trolls, the patterns were slightly different. Trolls often engage in inflammatory or provocative behavior, aiming to disrupt conversations and provoke reactions from other users. They may use offensive language, make controversial statements, or spread misinformation. They also tend to be more active in discussions, often responding to other users in an attempt to provoke further conflict.

IV. Development of the MVP

A. General System Design: Architecture and Key Components

The system was built with modularity, scalability, and robustness in mind. Python, a high-level, interpreted programming language famed for its simplicity and readability, was used to create it. Python was an excellent choice for this project due to its huge standard library and support for many programming paradigms [5].

The system was organized into four major subsystems, each of which was in charge of a certain duty in the pipeline. Three components were created as distinct Python modules and one as a node.js application, allowing for the encapsulation of data and procedures related to each task. The four main components were as follows:

Retrieval: This component was in charge of gathering data from Telegram using its API. It used the MTProto protocol to retrieve and preserve information from public channels and groups. Messages, user IDs, timestamps, and other pertinent metadata were collected [7].

Analysis: This component was responsible for analyzing the collected data. It used both quantitative and qualitative methods to identify patterns and trends indicative of bot and troll activity. The analysis was performed using various statistical methods and text analysis techniques.

Rest_Api: This component served as the interface between the system and the end-users, allowing users to interact with the system and retrieve the results of the analysis. The Rest API module was built using Python's Flask framework [25], a lightweight and flexible framework that is well-suited for developing web applications and services. The API provided endpoints for various functionalities, such as retrieving the results of the analysis or updating information about entities.

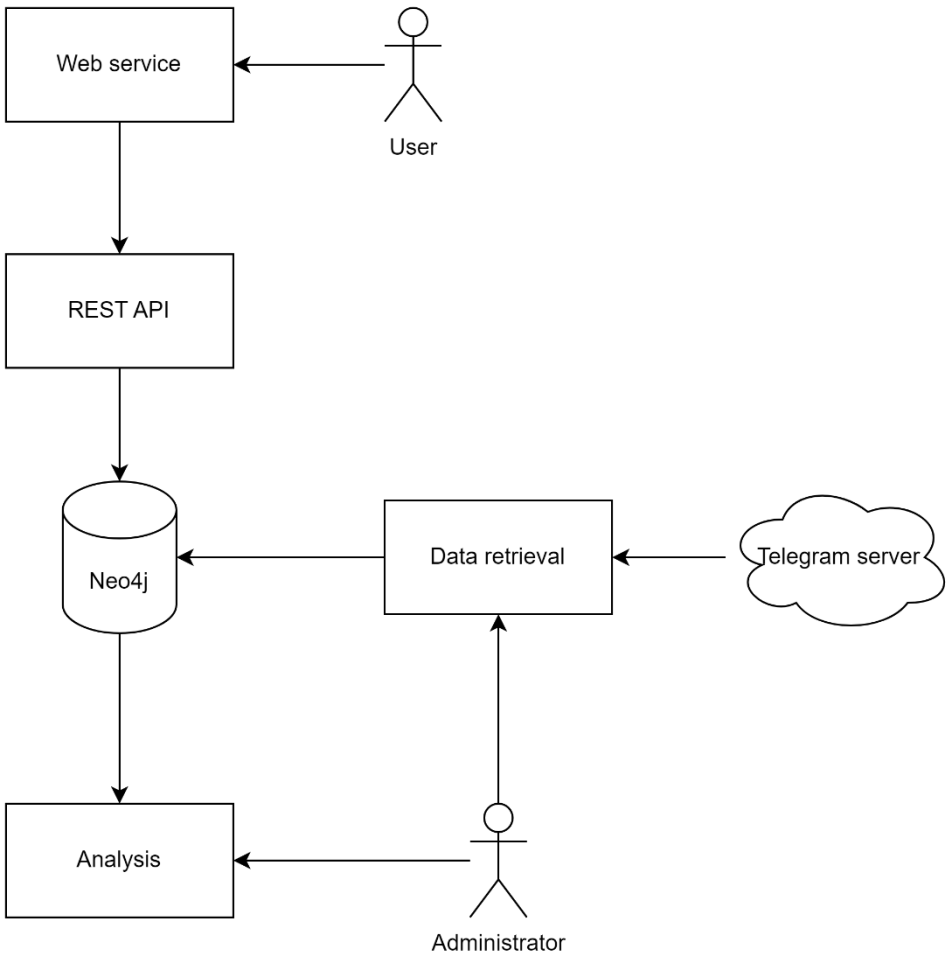

Fig. 1. Diagram of the whole system

The system was designed to be modular, with each component functioning independently of the others. This allowed for flexibility in the development process, as changes or improvements to one component did not require modifications to the others.

In addition to these main components, the system also utilized several libraries and tools to enhance its functionality. One of these was the sentence-transformers library, a Python framework for state-of-the-art sentence, text and image embeddings. The library was used in the Analyzer component to perform semantic similarity analysis, a technique used to identify similar phrases or talking points across different conversations or channels. This was particularly useful in identifying trolls, who often reuse the same phrases or talking points [26].

The sentence-transformers library provides pre-trained models for creating sentence embeddings. For this project, we used the multilingual model 'distiluse-base-multilingual-cased'. This model was chosen because it is trained on a variety of languages, including Russian, which was relevant to our research given the focus on the analysis of the Russian trolls' accounts.

B. Retrieval subsystem

The retrieval subsystem is responsible for the acquisition of data from Telegram using its API. This subsystem is implemented in Python and using the Telegram API and the MTProto protocol to fetch data from public channels and groups.

The retrieval subsystem is designed with a focus on multiprocessing, utilizing Python's multiprocessing module to achieve parallelism and enhance performance. This is particularly important given the large volumes of data that need to be retrieved and processed.

Consider the following diagram:

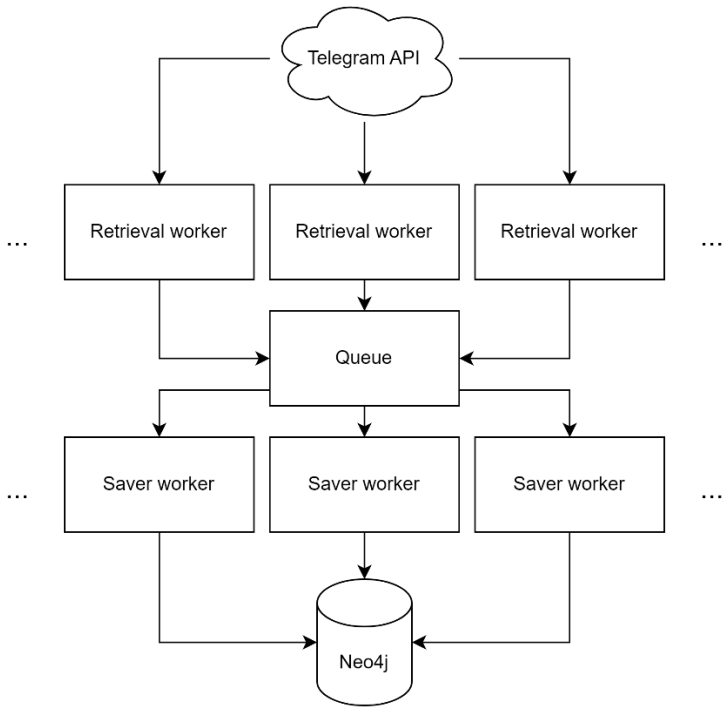

Fig. 2. Diagram of the multiprocessing setup, showing the separate processes for data fetching and saving.

The retrieval process is divided into two main stages: fetching posts, chats, accounts and comments from Telegram, and saving this data into a Neo4j graph database. The fetching process is performed in parallel using multiple processes, each retrieving data for a specific date range. This approach allows for efficient utilization of system resources and faster data retrieval.

Once the data is fetched, it is placed into a multiprocessing Queue. This Queue serves as a bridge between the fetching and saving stages, allowing for smooth data flow and separation of concerns.

The saving stage involves batch processing of the data from the Queue and saving it into a Neo4j database. Batch processing is used to speed up the saving process, reducing the number of database operations and thus improving performance. The saving process is performed using a Cypher query, a declarative graph query language used by Neo4j. The query merges the post and its comments into the database and creates relationships between the chat, post, and comments.

Here is an example of a Cypher query used in the saving process:

```
"""MATCH (ch:Chat {id: $chat_id}) MERGE (p: Post {id: $post_id, text: $post_text, timestamp: datetime(
$post_timestamp), views: $post_views, reactions: $post_reactions, shares: $post_shares,
suspicious_percentage: 0.0})

MERGE (ch)-[:HAS_POST]->(p)

FOREACH (comment IN $comments | MERGE (c: Comment {id: comment.comment_id, timestamp: datetime(comment.date),
text: comment.text}) MERGE (a: Account {id: comment.account.account_id}) ON CREATE SET a.username =
comment.account.username, a.first_name = comment.account.first_name, a.second_name =
comment.account.second_name, a.rogue_percentage = 0.0 ON MATCH SET a.username = comment.account.username,
a.first_name = comment.account.first_name, a.second_name = comment.account.second_name

    MERGE (a)-[:MADE]->(c)
    MERGE (p)-[:CONTAINS_COMMENT]->(c)

    FOREACH (ignoreMe IN CASE WHEN comment.reply_to_id IS NOT NULL THEN [1] ELSE [] END |
        MERGE (c2: Comment {id: comment.reply_to_id})
        MERGE (c)-[:REPLY_TO]->(c2)
        )
)
""",
```

Fig. 3. Cypher batch query for efficient data storage in the Neo4j database

This query first matches the chat in the database, then merges the post and its comments. It also creates relationships between the chat, post, and comments in the database.

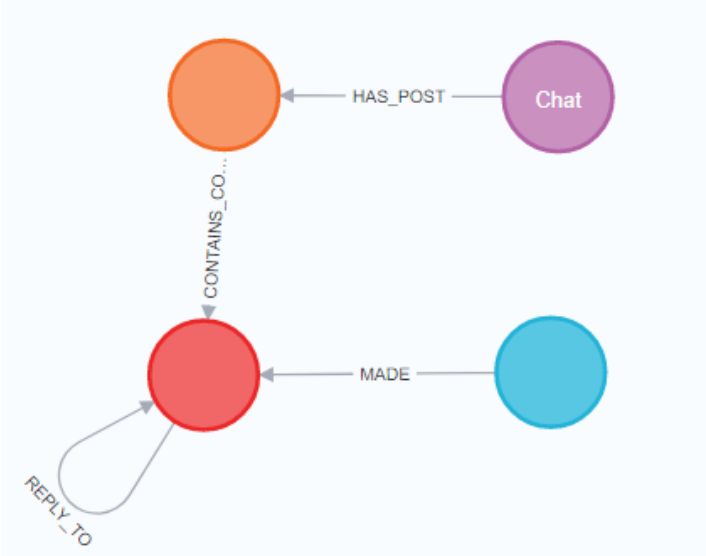

Fig. 4. Cypher batch query for efficient data storage in the Neo4j database. In the graph, the violet node represents a Chat, the orange node —a Post, the red node — Comment, and the blue node — an Account.

C. Analyzer subsystem

The Analyzer subsystem is designed to identify anomalies and suspicious activities within the data retrieved from social media platforms. The subsystem is structured around several key components and concepts:

Anomaly Analysis: This is a concept that involves examining data or entities (such as posts or comments) to identify unusual patterns or behaviors. Different types of anomaly analyses are implemented to focus on various aspects, such as the number of comments or reactions a post receives, the behavior of accounts interacting with a post, and the semantic similarity between comments.

Semantic Similarity Analysis: This analysis uses advanced natural language processing techniques to calculate the semantic similarity between comments on a post. If two comments are found to be very similar, it could indicate potential spam or bot activity, leading to an increase in the suspicion level associated with the post or the accounts that made the comments.

Rogue Account Analysis: This analysis focuses on identifying accounts that exhibit suspicious behavior. The percentage of rogue accounts interacting with a post is calculated, and if it exceeds a certain threshold, the suspicion level of the post is increased.

Analysis Service: This is a service that applies different types of anomaly analyses to an entity. It uses a flexible design pattern that allows for the easy addition of new types of analyses as needed.

Database Interaction: The subsystem includes functions to interact with the database, such as retrieving a post or a chat, increasing the suspicion level of a post or an account, and updating the suspicion level in the database.

The Analyzer subsystem uses multiprocessing and batched saving with a query to speed up the process of data retrieval and analysis. This efficient design allows for the rapid processing of large amounts of data, which is crucial for quick troll detection on social media platforms.

D. REST API subsystem

The REST API provides an interface for the client to interact with the system, allowing for the retrieval, creation, and modification of data. The subsystem is structured around the typical REST API structure, consisting of Controllers, Repositories, and Models.

Controllers: Controllers are responsible for handling HTTP requests and responses. They define the endpoints that the client can interact with and specify the HTTP methods (GET, POST, PUT, DELETE) that can be used.Each controller is associated with a specific type of data (accounts, chats, or plots) and uses dependency injection to utilize the services needed to process the data.

Repositories: Repositories handle the interaction with the database. They define the queries needed to retrieve, create, or modify data in the database. Each repository is associated with a specific type of data and uses the Py2neo library to interact with the Neo4j graph database.

Models: Models define the structure of the data in the system. They specify the attributes that each type of data should have and can include methods for processing the data. The Account model is an example of this.

Services: Services contain the business logic of the application. They use the repositories to interact with the database and perform any necessary processing on the data.

The REST API subsystem is implemented using the Flask framework, which provides a simple and flexible way to create web applications in Python. The Flask-Injector library is used to handle dependency injection, allowing for a clean and modular code structure.

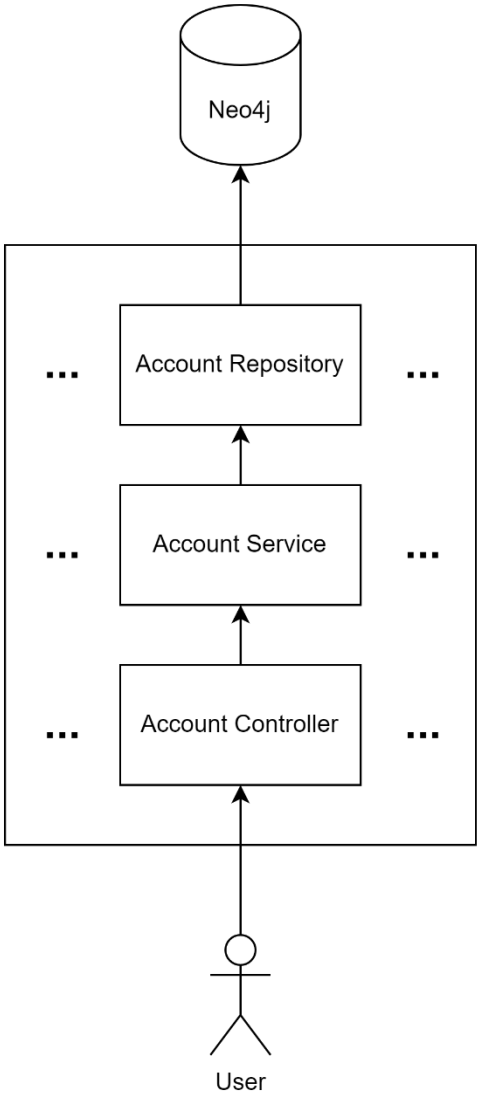

Fig. 5. An outline of the whole REST API on example of Account entity

E. Web frontend subsystem

The web frontend subsystem serves as an example of the user interface for the troll detection system. It is responsible for presenting data to the user in a clear and intuitive manner, allowing users to interact with the system and retrieve the information they need. This subsystem is implemented using React, a popular JavaScript library for building user interfaces.

React allows for the creation of reusable components, which can be combined to build complex user interfaces. Each component has its own state and props, allowing for the creation of dynamic and interactive interfaces [27]. The web frontend subsystem uses a variety of components to display different types of data, including account information and various plots.

The FusionCharts library is used to create the plots. FusionCharts is a JavaScript charting library that provides a wide variety of charts and graphs [28], making it an excellent choice for displaying the complex data involved in troll detection. The code retrieves data from the REST API and uses library to generate interactive plots that provide insights into the behavior of accounts.

The web frontend subsystem also includes a search feature, which allows users to search for accounts and check their "roughness". This feature uses the REST API to retrieve account data based on the user's search query, and presents the results in a user-friendly format.

The following diagrams provide a visual representation of the web frontend subsystem:

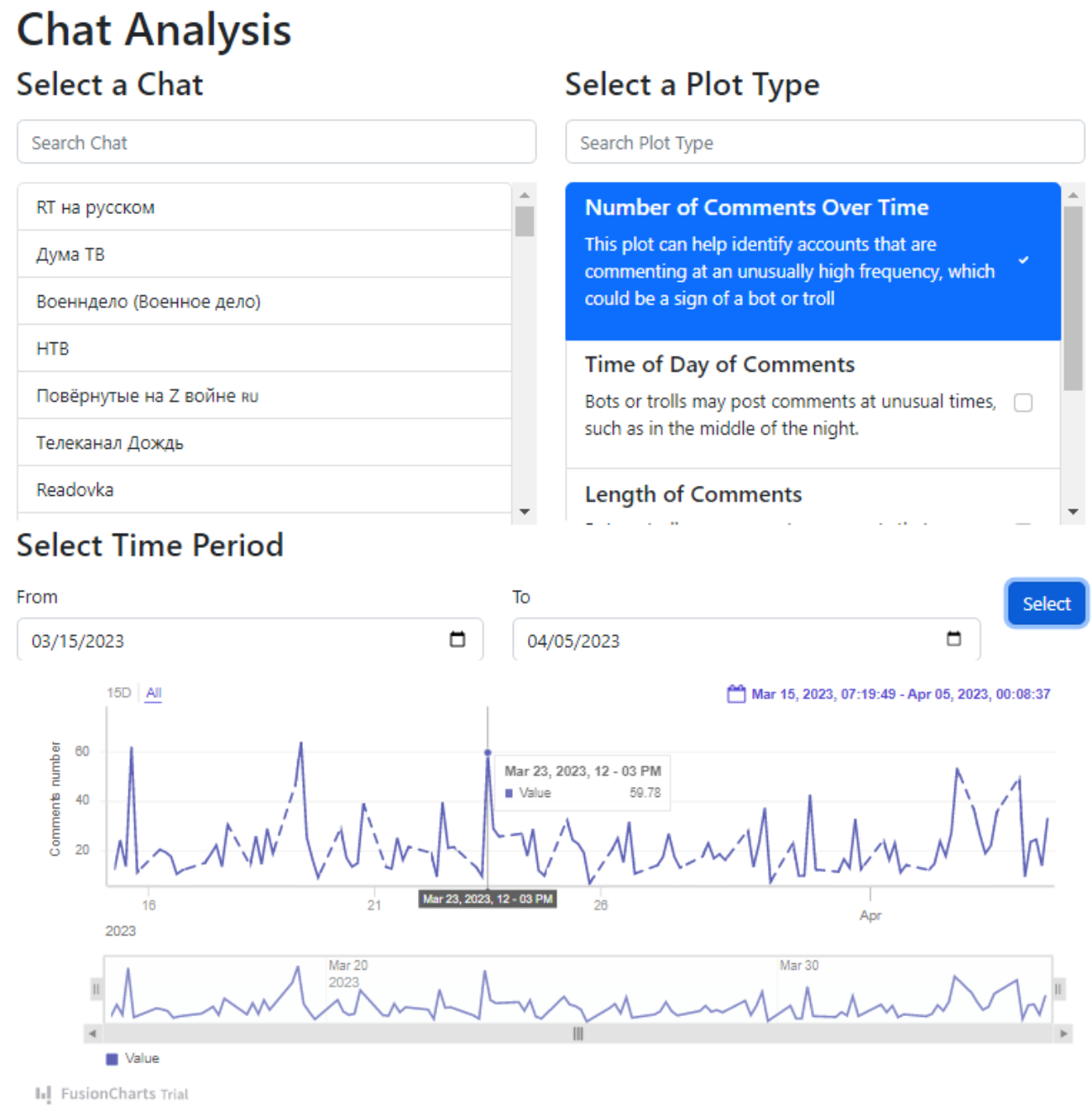

Fig. 6. The page for building a plot.

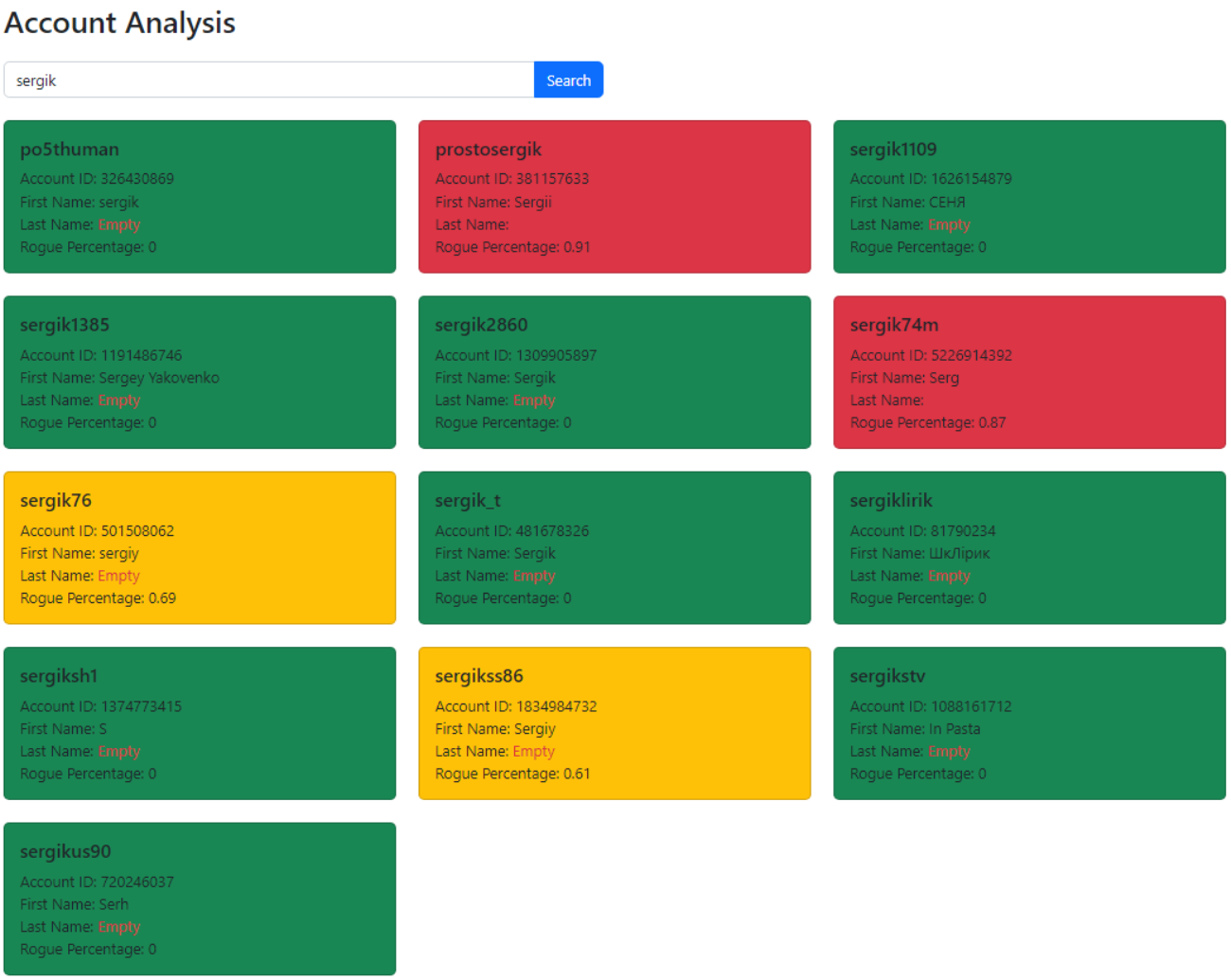

Fig. 7. The page for searching for an account. If data is missing because of privacy setting of an account "Empty" label is shown

V. Results

The implementation of the system has yielded good results in terms of data collection and analysis. The system has been successful in scanning and retrieving data from public channels with open comments, providing a comprehensive overview of the activity within these channels.

The data collected with a total of 895513 accounts, 206295 posts, and a staggering 14926289 comments from 70 chats. This vast amount of data provides a rich source for analysis and the identification of potential rogue accounts and posts.

The system's effectiveness is further demonstrated by its ability to detect accounts with a high likelihood of being trolls. A total of 3,589 accounts have been identified with a high probability of being trolls, while 6,973 accounts have been flagged with a medium probability. This identification is based on the system's analysis of account behavior, including the frequency and nature of posts and comments.

In addition to identifying potential rogue accounts, the system has detected posts that have been attacked by trolls. A total of 1,837 posts have been identified as being attacked by trolls with varying degrees of intensity. This information is valuable in understanding the nature of trolling behavior and can be used to develop strategies to mitigate such attacks.

However, the process was not without its challenges. One of the main challenges was the fact that Telegram allows users to disable certain features such as comments or reactions. This meant that it took some time to find a substantial number of channels with enabled comments. Additionally, the presence of various groups of trolls, such as "Цифровая Армия России" (Digital Army of Russia), meant that there were many different patterns and vectors of attack to consider.

Despite these challenges, the system has proven to be a valuable tool in the analysis of online behavior and the identification of potential rogue accounts and posts.

The following screenshots provide a visual representation of the data collected and analyzed by the system:

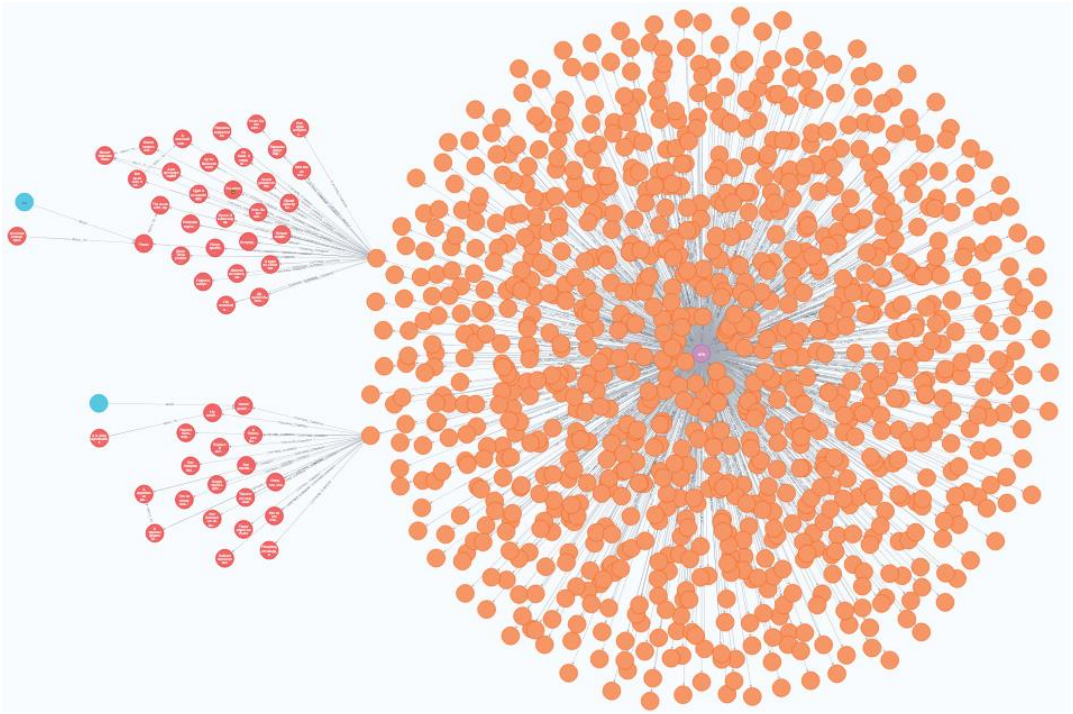

Fig. 8. Part of posts comments and accounts for the Russian government channel “НТВ”

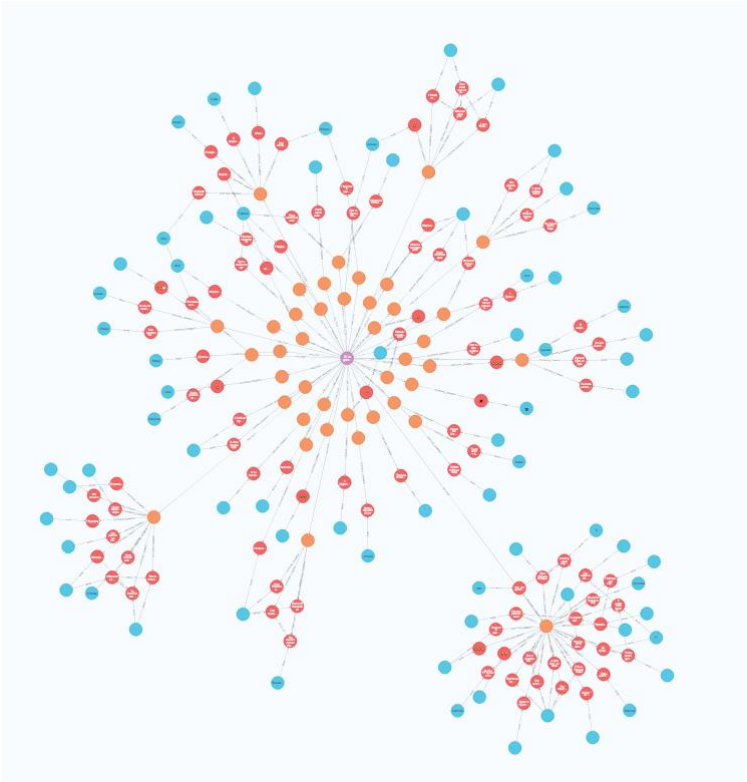

Fig. 9. Detailed view of accounts identified as potential trolls and posts, comments in Russian government channel(RT на русском)

VI. Conclusion

The research conducted in this study has led to the development of a robust and effective system for detecting bot and troll activity on Telegram. The system, developed as a Minimum Viable Product (MVP), comprises of several subsystems each designed to perform a specific task in the pipeline - from data retrieval to analysis and pattern recognition.

The system's design and implementation were guided by a focus on modularity, scalability, and robustness. Each subsystem was designed to function independently, allowing for flexibility and adaptability in the development process. The use of Python and its extensive libraries, the MTProto protocol for data retrieval, and the Neo4j graph database for data storage and analysis, all contributed to the system's effectiveness.

However, while the system has proven effective in its current form, there is potential for further development and improvement. Currently, the subsystems operate independently, but developing communication between the subsystems could enhance the system's efficiency and effectiveness. This would allow for a more integrated and coordinated approach to detecting bot and troll activity.

Application of machine learning tools could further enhance the system's capabilities. Machine learning algorithms could be trained on a dataset of known troll activity to identify patterns and behaviors indicative of such activity. However, the creation of such a dataset would require a significant amount of manual labeling and verification, which could be a potential direction for future research.

In conclusion, this research resulted in new methods of detection trolls and bots on Telegram. While there is potential for further development and improvement, the system as it stands represents a significant step forward in the fight against disinformation and malicious activity on social media platforms.

VII. References

[1] Telegram. Telegram FAQ. https://telegram.org/faq.

[2] I. Arpinar, “Truth or Troll: An Automated Framework for Identifying Authoritarian Regime Trolls in Twitter,” 2018. https://www.semanticscholar.org/paper/Truth-or-Troll%3A-An-Automated-Framework-for-Regime-Arpinar-Rasheed/a3308a8f83275925f62233b734cac1a675ebe037.

[3] Weisburd, Andrew, Clint Watts, and JM Berger. Trolling for Trump: How Russia Is Trying to Destroy Our Democracy. War on the Rocks. (2016). https://warontherocks.com/2016/11/trolling-for-trump-how-russia-is-trying-to-destroy-our-democracy/.

[4] E. Ferrara, O. Varol, C. A. Davis, F. Menczer, and A. Flammini, “The rise of social bots,” Communications of the ACM, vol. 59, no. 7, pp. 96–104, Jun. 2016, doi: 10.1145/2818717.

[5] “What is Python? Executive Summary,” Python.org. https://www.python.org/doc/essays/blurb/

[6] Forelle, M. C., Howard, P. N., Monroy-Hernandez, A., & Savage, S. (2015). Political Bots and the Manipulation of Public Opinion in Venezuela. https://doi.org/10.2139/ssrn.2635800

[7] “MTProto Mobile Protocol.” https://core.telegram.org/mtproto

[8] Weng, Z., & Lin, A. (2022). Public Opinion Manipulation on Social Media: Social Network Analysis of Twitter Bots during the COVID-19 Pandemic. https://doi.org/10.3390/ijerph192416376

[9] E. F. O. V. Flammini Clayton Davis, Filippo Menczer, Alessandro, “The Rise of Social Bots,” July 2016 | Communications of the ACM, Jul. 01, 2016. https://cacm.acm.org/magazines/2016/7/204021-the-rise-of-social-bots/fulltext

[10] W. Phillips and R. M. Milner, The Ambivalent Internet: Mischief, Oddity, and Antagonism Online. 2017. [Online]. Available: https://openlibrary.org/books/OL29385520M/Ambivalent_Internet

[11] Y. Benkler, R. Farris, and H. Roberts, “Network Propaganda,” in Oxford University Press eBooks, 2018. doi: 10.1093/oso/9780190923624.001.0001.

[12] P. Singer and E. T. Brooking, LikeWar: The Weaponization of Social Media. 2018. [Online]. Available: https://ci.nii.ac.jp/ncid/BB27080012

[13] N. R. I. M. Zog Pavlo Solodko, Orest, “The Troll Network,” ТЕКСТИ.ORG.UA. https://texty.org.ua/d/fb-trolls/index_eng.html

[14] A. Chen, “The Agency,” The New York Times, Sep. 20, 2020. [Online]. Available: https://www.nytimes.com/2015/06/07/magazine/the-agency.html

[15] У. Криминальная, “В США начали охоту на проплаченных интернет-троллей из России,” УКРАЇНА КРИМІНАЛЬНА, Dec. 2018, [Online]. Available: https://cripo.com.ua/news/?p=175890/

[16] R. DiResta, “The Tactics & Tropes of the Internet Research Agency,” DigitalCommons@University of Nebraska - Lincoln. https://digitalcommons.unl.edu/senatedocs/2/

[17] R. P. Mueller, Report On The Investigation Into Russian Interference In The 2016 Presidential Election. 2019. [Online]. Available: https://apo.org.au/node/231061

[18] “The menace of unreality: how the Kremlin weaponizes information, culture and money,” The Library of Congress. https://www.loc.gov/item/2015433465/

[19] K. Kirk, “How Russia Sows Confusion in the U.S. Vaccine Debate,” Foreign Policy, Apr. 09, 2019. [Online]. Available: https://foreignpolicy.com/2019/04/09/in-the-united-states-russian-trolls-are-peddling-measles-disinformation-on-twitter/

[20] J. Robbins, “The Diversity of Russia’s Military Power: Countering Russian Disinformation,” Center for Strategic and International Studies (CSIS), 2020. [Online]. Available: https://www.jstor.org/stable/resrep26533.8

[21] “Detection of Fake and Provokative Comments in Social Network Using Machine Learning,” IEEE Conference Publication | IEEE Xplore, Jan. 01, 2020. https://ieeexplore.ieee.org/document/9039423

[22] K. Baraniuk, “Corpus-Based Analysis as a Method to Identify Russian Trolling Activity,” Polish Political Science Yearbook, Jun. 2017, doi: 10.15804/ppsy2017115.

[23] V. Solopova, O.-I. Popescu, C. Benzmüller, and T. Landgraf, “Automated Multilingual Detection of Pro-Kremlin Propaganda in Newspapers and Telegram Posts,” Datenbank-spektrum, Mar. 2023, doi: 10.1007/s13222-023-00437-2.

[24] H. Shafiei and A. Dadlani, “Detection of fickle trolls in large-scale online social networks,” Journal of Big Data, vol. 9, no. 1, Feb. 2022, doi: 10.1186/s40537-022-00572-9.

[25] “Welcome to Flask — Flask Documentation (2.3.x).” https://flask.palletsprojects.com/

[26] “SentenceTransformers Documentation — Sentence-Transformers documentation.” https://www.sbert.net/

[27] “React.” https://react.dev/

[28] “FusionCharts.” https://www.fusioncharts.com/

VII. Appendices

Analysis system

```
# File Path: src\analyzer\analysis.py

from abc import ABC, abstractmethod

from py2neo import Graph

from config import Config

from constants import CONFIG_FILE_PATH

from model import db_convertors

from sentence_transformers import SentenceTransformer, util

from model import Account, Chat, Post, Comment



class AnomalyAnalysis(ABC):
    @abstractmethod
    def analyze(self, entity):
        pass



class PostAnomalyAnalysis(AnomalyAnalysis):
    def __init__(self, graph: Graph):
        self.graph = graph

    def analyze(self, post):
        avg_comments_query = """
```

```
        MATCH (ch:Chat {id: $chat_id})-[:HAS_POST]->(p:Post)-[:CONTAINS_COMMENT]-
>(c:Comment)
        WITH p, count(c) AS num_comments
        RETURN avg(num_comments) AS avg_comments
        """
        avg_reactions_query = """
        MATCH (ch:Chat {id: $chat_id})-[:HAS_POST]->(p:Post)
        WITH p, apoc.convert.fromJsonMap(p.reactions) as reactions
        UNWIND reactions as reaction
        UNWIND keys(reaction) as key
        WITH sum(reactions[key]) as reaction_sum, p
        RETURN avg(reaction_sum) as avg_reactions
        """
        total_comments_number_for_post_query = """
        MATCH (ch:Chat {id: $chat_id})-[:HAS_POST]->(p:Post {id: $post_id})-
[:CONTAINS_COMMENT]->(c:Comment)
        RETURN count(c) as comments_number
        """
        avg_comments = self.graph.run(avg_comments_query,
chat_id=post.chat.chat_id).data()[0]['avg_comments']
        avg_reactions = self.graph.run(avg_reactions_query,
chat_id=post.chat.chat_id).data()[0]['avg_reactions']
        comments_number = \
            self.graph.run(total_comments_number_for_post_query, post_id=post.post_id,
                           chat_id=post.chat.chat_id).data()[0][
                'comments_number']
```

```
        suspect = (comments_number > 2 * avg_comments) or
(sum(post.reactions.values()) > 2 * avg_reactions)

        if suspect:
            post.suspicious_percentage += 0.1  # Increase the suspicious_percentage by 0.1

        return [post]



class RogueAccountsAnomalyAnalysis(AnomalyAnalysis):
    def __init__(self, graph: Graph, rogue_percentage_account_min: float,
                 percentage_of_rogue_accounts_threshold: float):
        self.graph = graph
        self.rogue_percentage_account_min = rogue_percentage_account_min
        self.percentage_of_rogue_accounts_threshold =
percentage_of_rogue_accounts_threshold

    def analyze(self, post):
        rogue_accounts_query = """
        MATCH (ch:Chat {id: $chat_id})-[:HAS_POST]->(:Post)-[:CONTAINS_COMMENT]-
>(:Comment)<-[:MADE]-(a:Account)
        WHERE a.rogue_percentage > $account_rogue_percentage
        RETURN count(DISTINCT a) AS rogue_count
        """
```

```
        total_accounts_query = """
        MATCH (ch:Chat {id: $chat_id})-[:HAS_POST]->(:Post)-[:CONTAINS_COMMENT]-
>(:Comment)<-[:MADE]-(a:Account)
        RETURN count(DISTINCT a) AS total_count
        """
        rogue_count = self.graph.run(rogue_accounts_query, chat_id=post.chat.chat_id,

account_rogue_percentage=self.rogue_percentage_account_min).evaluate()
        total_count = self.graph.run(total_accounts_query,
chat_id=post.chat.chat_id).evaluate()

        if total_count > 0 and rogue_count / total_count >
self.percentage_of_rogue_accounts_threshold:
            post.suspicious_percentage += 0.1

        return [post]



class SemanticSimilarityAnalysis(AnomalyAnalysis):
    def __init__(self, graph, model, min_similarity_ratio: float):
        self.graph = graph
        self.model = model
        self.min_similarity_ratio = min_similarity_ratio

    def analyze(self, post):
        comment_texts_query = """
```

```
        MATCH (ch:Chat {id: $chat_id})-[:HAS_POST]->(p:Post {id: $post_id})-[:CONTAINS_COMMENT]->(c:Comment)<-[:MADE]-(a:Account)
        WITH a, collect(c) as comments
        RETURN a, comments
        """
        result = self.graph.run(comment_texts_query, chat_id=post.chat.chat_id, post_id=post.post_id).data()
        comments = []
        for record in result:
            account = db_convertors.map_db_node_to_account(record['a'])
            print(account)
            for comment in record['comments']:
                comment = db_convertors.map_db_node_to_comment(comment, account, post)
                comments.append(comment)

        if len(result) < 2:
            return False

        # Calculate semantic similarity for comments
        embeddings = self.model.encode([comment.text for comment in comments], convert_to_tensor=True)
        similarity_matrix = util.pytorch_cos_sim(embeddings, embeddings)

        for i in range(len(similarity_matrix) - 1):
            for j in range(i + 1, len(similarity_matrix)):
```

```
            if similarity_matrix[i][j] > self.min_similarity_ratio:
                increase_suspicion_percentage(post, comments[i].account)
                increase_suspicion_percentage(post, comments[j].account)

        return [
            list(map(lambda comment: comment.account, comments)) + list(map(lambda comment: comment.post, comments))]



def increase_suspicion_percentage(post, account):
    post.suspicious_percentage += 0.05
    account.rogue_percentage += 0.1



class AnalysisService:
    def __init__(self, analyses):
        self.analyses = analyses

    def analyze(self, entity):
        results = []
        for analysis in self.analyses:
            results.append(analysis.analyze(entity))
        return results
```

```
def get_post_from_db(graph, chat, post_id):
    post_query = """
    MATCH (ch: Chat {id: $chat_id})-[:HAS_POST]->(p:Post {id: $post_id})
    RETURN p.shares as shares, p.reactions as reactions, p.text as text, p.id as id, p.views as views, p.timestamp as timestamp, p.suspicious_percentage as suspicious_percentage
    """
    result = graph.run(post_query, post_id=post_id, chat_id=chat.chat_id).data()
    return db_convertors.map_db_node_to_post(result[0], chat)


def get_chat_from_db(graph, chat_id):
    chat_query = """
    MATCH (c:Chat {id: $chat_id})
    RETURN c.id AS id, c.name AS title, c.username AS username
    """
    result = graph.run(chat_query, chat_id=chat_id).data()
    if result:
        return db_convertors.map_db_node_to_chat(result[0])
    else:
        return None


def update_suspicious_percentage(graph, entities):
    for entity in entities:
```

```
        entity_type = type(entity)
        if entity_type == Post:
            graph.run("MATCH (p:Post {id : $id}) SET p.suspicious_percentage =
$suspicious_percentage",
                      id=entity.post_id, suspicious_percentage=entity.suspicious_percentage)
        elif entity_type == Account:
            graph.run("MATCH (p:Post {id : $id}) SET p.rogue_percentage =
$rogue_percentage",
                      id=entity.post_id, rogue_percentage=entity.suspicious_percentage)



if __name__ == "__main__":
    import sys


    if len(sys.argv) != 3:
        print("Usage: python main.py <chat_id> <post_id>")
        sys.exit(1)


    chat_id = int(sys.argv[1])
    post_id = int(sys.argv[2])


    # Initialize the Neo4j graph connection
    config = Config(CONFIG_FILE_PATH)
    graph = Graph(f"bolt://{config.NEO4J_HOSTNAME}:{config.NEO4J_PORT}",
                  auth=(config.NEO4J_LOGIN, config.NEO4J_PASSWORD))
```

```
    # Retrieve a chat from the database
    chat = get_chat_from_db(graph, chat_id)
    print(chat)

    # Retrieve a post from the database
    post = get_post_from_db(graph, chat, post_id)
    print(post)

    # Initialize the PostAnomalyAnalysis with the graph
    post_analysis = PostAnomalyAnalysis(graph)

    # Initialize the RogueAccountsAnomalyAnalysis with the graph
    rogue_analysis = RogueAccountsAnomalyAnalysis(graph,
rogue_percentage_account_min=0.9,
                                                  percentage_of_rogue_accounts_threshold=0.3)

    # Initialize the SentenceTransformer model
    model = SentenceTransformer('distiluse-base-multilingual-cased-v1')

    # Initialize the SemanticSimilarityAnalysis with the model
    semantic_analysis = SemanticSimilarityAnalysis(model, min_similarity_ratio=0.7)

    # Initialize the AnalysisService with the analyzer objects
    service = AnalysisService([post_analysis, rogue_analysis, semantic_analysis])
```

```
    if post:
        # Run the analyzer on the retrieved post
        results = service.analyze(post)
        for result in results:
            print(result)
            update_suspicious_percentage(graph, result)
            print("Values updated")
    else:
        print("Post not found in the database.")
```

Account model

```
# File Path: src\model\account.py
from dataclasses import dataclass, field

from model.base import Base


@dataclass
class Account(Base):
    account_id: int
    first_name: str
    second_name: str
```

```
    username: str
    rogue_percentage: float = field(default=0.0)

    def __hash__(self):
        return hash((self.account_id, self.first_name, self.second_name, self.username, self.rogue_percentage))
```

Base

```
# File Path: src\model\base.py
from dataclasses import dataclass, asdict, is_dataclass
from typing import Dict, Any

import marshmallow_dataclass


@dataclass
class Base:
    def to_dict(self) -> Dict[str, Any]:
        def _to_dict(obj):
            if is_dataclass(obj):
                return {k: _to_dict(v) for k, v in asdict(obj).items() if v is not None}
            elif isinstance(obj, list):
                return [_to_dict(item) for item in obj]
            else:
```

```
            return obj


    return _to_dict(self)

@classmethod
def to_schema(cls):
    return marshmallow_dataclass.class_schema(cls)()
```

Chat model

```
# File Path: src\model\chat.py
from dataclasses import dataclass

from model.base import Base


@dataclass
class Chat(Base):
    title: str
    username: str
    chat_id: int

    def __hash__(self):
        return hash((self.title, self.username, self.chat_id))
```

Comment model

```
# File Path: src\model\comment.py

import datetime

from dataclasses import dataclass


from model.base import Base

from model.account import Account

from model.post import Post



@dataclass

class Comment(Base):

    comment_id: int

    text: str

    date: datetime.datetime

    post: Post

    reply_to_id: int

    account: Account


    def __hash__(self):

        return hash((self.comment_id, self.text, self.date, self.post, self.account))
```

Post model

```
# File Path: src\model\post.py

import datetime

from dataclasses import dataclass, field


from model.base import Base

from model.chat import Chat



@dataclass

class Post(Base):

    text: str

    post_id: int

    date: datetime.datetime

    chat: Chat

    reactions: dict

    views: int

    shares: int

    suspicious_percentage: float = field(default=0.0)


    def __hash__(self):

        return hash((self.text, self.post_id, self.date))
```

Data collection system

```
# File Path: src\retrieval\app.py

import queue

import argparse


from telethon.sync import TelegramClient

from multiprocessing import Pool, Queue, Process, Value, Manager

from py2neo import Graph


from config import Config

from constants import CONFIG_FILE_PATH, LOG_DIR

from model.convertors import chat_to_my_chat, message_to_post,
message_to_comment

from interchange.time import DateTime


import datetime

import logging

import os

import json


logger = logging.getLogger("main")

logger.setLevel(logging.DEBUG)


if not os.path.exists(LOG_DIR):
```

```
    os.makedirs(LOG_DIR)


fh = logging.FileHandler(f"{LOG_DIR}/all.log", 'a', 'utf-8')

fh.setLevel(logging.DEBUG)

formatter = logging.Formatter("%(asctime)s - %(name)s - %(levelname)s -
%(message)s")

fh.setFormatter(formatter)

logger.addHandler(fh)


config = Config(CONFIG_FILE_PATH)


graph = Graph(f"bolt://{config.NEO4J_HOSTNAME}:{config.NEO4J_PORT}",

            auth=(config.NEO4J_LOGIN, config.NEO4J_PASSWORD))



def fetch_posts_and_comments(chat, data_queue, start_date, end_date, api_session,
targets_queue):

    logger.info("Started fetcher with session: " + api_session)

    with TelegramClient(api_session, config.API_ID, config.API_HASH) as client:

        posts = filter(

            lambda post: post.replies is not None and post.replies.replies != 0,

            client.iter_messages(chat.username, offset_date=end_date))

        for post_raw in posts:

            # Break if we come to the start date

            if post_raw.date.date() < start_date:
```

```
            break

        # Parse post
        post = message_to_post(post_raw, chat)

        comments_raw = list(filter(lambda comment: comment.from_id is not None,
                                   client.iter_messages(chat.username, reply_to=post.post_id)))
        comments = list(map(lambda comment: message_to_comment(post, comment), comments_raw))
        comments.reverse()

        data_queue.put(("Comments", comments))
        targets_queue.put(post.sender_id)
        # print("Putting data to queue in processor:", multiprocessing.cpu_count())
        print("[" + str(api_session) + "]:" + str(post_raw))
    logger.info("Finished fetcher with session: " + api_session)


def save_chat(chat, graph):
    res = graph.run(
        """
        MERGE (c:Chat {id: $chat_id, username: $chat_username, name: $chat_name })
        """,
        {
            "chat_id": chat.chat_id,
```

```
        "chat_username": chat.username,
        "chat_name": chat.title
    }
  )


def save_post_with_comments(post, comments, graph):
  res = graph.run(
    """MATCH (ch:Chat {id: $chat_id}) MERGE (p: Post {id: $post_id, text: $post_text,
timestamp: datetime(
    $post_timestamp), views: $post_views, reactions: $post_reactions, shares:
$post_shares,
    suspicious_percentage: 0.0})

    MERGE (ch)-[:HAS_POST]->(p)

    FOREACH (comment IN $comments | MERGE (c: Comment {id:
comment.comment_id, timestamp: datetime(comment.date),
    text: comment.text}) MERGE (a: Account {id: comment.account.account_id}) ON
CREATE SET a.username =
    comment.account.username, a.first_name = comment.account.first_name,
a.second_name =
    comment.account.second_name, a.rogue_percentage = 0.0 ON MATCH SET
a.username = comment.account.username,
    a.first_name = comment.account.first_name, a.second_name =
comment.account.second_name
```

```
        MERGE (a)-[:MADE]->(c)
        MERGE (p)-[:CONTAINS_COMMENT]->(c)

        FOREACH (ignoreMe IN CASE WHEN comment.reply_to_id IS NOT NULL THEN [1] ELSE [] END |
            MERGE (c2: Comment {id: comment.reply_to_id})
            MERGE (c)-[:REPLY_TO]->(c2)
            )
    )
    """,
    {
        "chat_id": post.chat.chat_id,

        "post_id": post.post_id,
        "post_text": post.text,
        "post_timestamp": DateTime.from_native(post.date).replace(tzinfo=None),
        "post_views": post.views,
        "post_reactions": json.dumps(post.reactions),
        "post_shares": post.shares,
        "post_is_propaganda": post.is_propaganda,
        "comments": [
            # There is a bug in interchange.time convertors, datetime not working ok.
            # Set post to none not to care about that
```

```
            {**comment.to_dict(), "post": None, "date":
DateTime.from_native(comment.date).replace(tzinfo=None)}
            for comment in comments
        ]
    }
  )



def save_to_neo4j(data_queue, stop, label):
  graph = Graph(f"bolt://{config.NEO4J_HOSTNAME}:{config.NEO4J_PORT}",
            auth=(config.NEO4J_LOGIN, config.NEO4J_PASSWORD))
  logger.info("Started saver")
  while True:
    try:
      try:
        # No data for a now
        # data value could be empty
        entity, *data = data_queue.get(timeout=1)
      except queue.Empty:
        if stop.value:
          break
        continue
      # print("Retrieving data from queue:", entity, "in processor:",
multiprocessing.cpu_count())
```

```
            if entity == "Chat":
                save_chat(data[0], graph)
            elif entity == "Comments":
                comments = data[0]
                post = comments[0].post
                save_post_with_comments(post, comments, graph)
                print(label, post)
        except Exception as e:
            logger.error(str(e) + str(post))
            print(e)
    logger.info("Stopped saver")


def divide_dates_into_n_pairs(date1, date2, n):
    delta = (date2 - date1) / n

    intervals = [date1 + delta * i for i in range(n + 1)]

    pairs = list(zip(intervals[:-1], intervals[1:]))

    return pairs


def parse_arguments():
```

```
    parser = argparse.ArgumentParser(description='Telegram Data Retrieval')
    parser.add_argument('-c', '--chat_links', nargs='+', required=True, help='Links to the Telegram chats')
    parser.add_argument('-sd', '--start_date', required=True, help='Start date (YYYY-MM-DD)')
    parser.add_argument('-ed', '--end_date', required=True, help='End date (YYYY-MM-DD)')
    parser.add_argument('-t', '--threads', type=int, default=30, help='Number of retrieval threads')
    return parser.parse_args()


if __name__ == '__main__':
    args = parse_arguments()

    chat_links = args.chat_links
    start_date = datetime.datetime.strptime(args.start_date, '%Y-%m-%d').date()
    end_date = datetime.datetime.strptime(args.end_date, '%Y-%m-%d').date()
    threads = args.threads

    dates = divide_dates_into_n_pairs(start_date, end_date, threads)

    for chat_link in chat_links:
        data_queue = Queue()
        stop_saver = Value('b', False)
        targets_queue = Queue()
```

```
    with TelegramClient("sessions/bot-session0", config.API_ID, config.API_HASH) as client:
        chat = chat_to_my_chat(client.get_entity(chat_link))
        data_queue.put(("Chat", chat))

    fetch_args = [
        (chat, data_queue, start, end, "sessions/bot-session" + str(idx), targets_queue)
        for idx, (start, end) in enumerate(dates)
    ]

    print(fetch_args)

    fetch_processes = [Process(target=fetch_posts_and_comments, args=args) for args in fetch_args]
    save_processes = [Process(target=save_to_neo4j, args=(data_queue, stop_saver, "Saver-" + str(i))) for i in
                      range(3)]

    for p in fetch_processes:
        p.start()
    for p in save_processes:
        p.start()

    for p in fetch_processes:
```

```
    p.join()


# Stops when queue is empty
stop_saver.value = True
for p in save_processes:
    p.join()
```