A Big Data System for Community Evolution Graph Analysis on Twitch

Dimitris Michailidis
University of Amsterdam
jimichailidis@gmail.com

Jeroen Schmidt
University of Amsterdam
jeroen.f.l.schmidt@gmail.com

Michael Schniepp
University of Amsterdam
michael.schniepp@student.uva.nl

Ben Wilmers
University of Amsterdam
benwilmers@me.com

ABSTRACT

Twitch.tv is the biggest livestreaming platform on the internet. More than 15 million unique viewers tune in everyday to watch their favourite broadcasters. This rapid growth has led to the emergence of massive social communities for users to be part of and actively participate in. These communities continue to grow and dynamically change over time as Twitch becomes more popular, creating space for Big Data applications to emerge. In this paper we propose an architecture for a Big Data system that captures these communities and how they evolve over time. We validate the system by performing analysis on the Greek community and finally propose an architecture to address scalability issues.

KEYWORDS

big data, live streaming, Twitch, community evolution, graph analysis

1 INTRODUCTION

Since its inception in 2011, video game streaming site Twitch.tv has ballooned in size, currently boasting over 15 million unique daily viewers and 2.2 million unique monthly streamers - making it the largest gaming community in history[3] and the 4th largest stream of data on the internet. The site provides a platform for streamers to broadcast themselves playing video games live to up to hundreds of thousands of concurrent viewers. Spectators are able to interact with streamers through chat and donation messages and can follow or subscribe to their favourite broadcasters. By following a streamer, a user will be notified when they start a new broadcast. In this way streamers are able to build up an immense audience, with the largest channels commanding follower counts in the millions. The growth of Twitch.tv as a social website has engendered the emergence of massive social communities where users form implicit groups defined by factors such as language, video games, and streaming style. Twitch.tv promotes the categorisation of streamers by language and game, allowing users to browse current streamers playing a specific game or communicating via a specific language.

These communities have given rise to some interesting social dynamics which have yet to be explored fully. These are not limited to interactions between individuals but also encompass the dynamics between entire fan-bases and communities on a larger scale. In this paper we explore how a large scale system can be designed and how graph clustering models can be utilised to analyse Twitch community evolution over time.

With that goal in mind, we define the following research questions:

- Can a big data system be designed to mine and store large scale graph data and perform on demand community analysis?
- How can graph clustering models be utilised to analyse twitch community evolution in time?
- Is the model able to capture the dynamic evolution of the Greek Twitch community in time?
- How can the system be scaled to analyse the global Twitch communities?

The dynamics to be observed in this study will reflect the interaction of participants of such communities. These communities we can define as collections of gamers, that is, individuals who take considerable interest in the playing and viewing of video games. Gamers who regularly or professionally broadcast their gaming are considered streamers. Groups of gamers or viewers who regularly watch specific streamers related through game or content type can be considered a community. We find that many individuals who have shared viewing preferences often interact with streamers and fellow viewers in the stream chat or other related forums. Through our analysis we set out to confirm these relationships between users and streamers and further attempt to see how they evolve over time.

Answering these questions will be of value in several ways. Insight into community development will enable advertisers to better optimise their advertising strategy in regards to targeting specific groups of users. Our system will also be of use to Twitch streamers aiming to identify emerging communities and orient their networking strategy and content approach in order to grow their channel effectively.

The rest of the paper is structured as follows: first we give an overview of the work that has already been done on Twitch communities and what these studies lack. We then provide a description of the data we worked with and the proposed design of our system. We go on to answer the research questions by describing the prototype implementation and proposing a scalable design before we finalise the report with conclusions, reflections and future work.

2 RELATED WORK

Twitch has been raising the interest of researchers for a while, with many studies conducted on all different aspects of the platform. In this section we focus on previous studies that involve community analysis, either empirically or using modelling techniques.

Hamilton et al. [2] conceptualised a Twitch community through four components: membership, influence, fulfilment of needs, and emotional connection. They then conducted ethnographic investigation via semi-structured interviews with eleven streamers and four viewers, trying to identify these components via the different experiences each person exposed. They conclude that the format of Twitch streams create a framework that enables all these components.

Nascimento et al. [5] introduced a modelling approach for streamers, spectators and their actions. They defined different actions as joining a channel, leaving a channel and sending a message to the chat. They then used system dynamics approaches to model and understand the dynamic behaviour of the streamers and their spectators in a specific game community (Starcraft). They concluded that it is possible to identify clear behaviour patterns and their semantic meaning within a community, as well as use these patterns to predict different events within a streaming session. However, they did not investigate how elements of different communities interact with each other.

Regarding graph analysis research, Twitch Science Team [7] published a study on visually analysing communities based on amount of follower overlap between channels. They effectively identify different communities based on games, genres and the companies who developed them. They conclude that streamers who broadcast similar games usually form dense communities that are distinguishable from each other. Moreover, communities with similar interests in terms of genre and developer appear closer to each other and further away from non similar communities. Finally, mainstream channels appear in the centre of the map with large overlap between other famous channels. However, this study only covers a small period of time (December 2015) and does not explore how these communities evolve over time.

Later on, Churchill et al. [1] classify streamers into three subcultures (Casual, Speedrun and Competitive) and introduce new community visualisation methods that effectively capture the social relationships and dynamics among them. They confirm [7] that franchises from similar companies attract similar audiences and conclude that the different subcultures are clearly segregated and separable from each other.

In this paper, we propose a big data system to analyse how Twitch communities evolve in time. Our architecture design can be utilised to download, store and perform on demand graph analyses on different time periods. We prove the design by studying the case of the Greek Twitch community and performing clustering on a graph of 84 streamers and more than 300,000 followers.

3 THE DATA

To enable the construction and analysis of directed and aggregated graphs, one must transform data into a format that resembles entities and relationships. Luckily, Twitch provides a public access API that allows for follow relationship data to be mined in this exact format, as shown in Table 1. In detail, consuming the *followers* endpoint returns responses about a channel in JSON format. Each response contains the following data:

- *total*: total followers of the channel.
- *pagination*: a key to use as a parameter in next requests to achieve pagination.
- *data*: an array containing up to 100 follow relationships at a time in the following format: [from_id] [from_name] [to_id] [to_name] [followed_at].

To access the API an application must be registered in the Twitch developers portal[1]. Following that, simple authentication allows the creation of a bearer token that expires in sixty days and can be used for up to eight hundred requests per minute. Twitch allows for multiple applications and therefore multiple tokens per developer account, thus not limiting the data mining capabilities. However, the API does not support real time data streaming of relationship information so an iterative data acquisition process needs to be created. We describe this process in detail in Section 4. The consumption of the API is free of charge at the time this paper is written.

4 PROPOSED DESIGN

In this section we propose a big data architecture design that enables large scale community analysis to be conducted on Twitch. We describe the methodology, the architecture details and each component in the process.

4.1 Methodology

To design a system to answer our research questions, we need to set the foundations of how to approach the problem. At first, we define the end goals as follows:

- *Capture new data as it is created*: so that the relationship graph is always up to date and new entities are integrated without big delays.
- *Filter based on time-windows*: efficient graph filtering based on a given time window.
- *Graph based community analysis*: perform community identification on the extracted graph.
- *Community visualisation*: display the communities in an intuitive and interpretable way.

With these goals in mind, we define the three main components of the proposed system as follows: (1) Recurrent Data Acquisition, which continuously streams data from Twitch and enhances the total graph, (2) On Demand Graph Computation, which is used to produce and store aggregated subgraphs per time window and (3) On Demand Graph Analysis, which accesses the subgraphs to perform community analyses, visualisations, applications, etc. In the following section we describe the architecture of the aforementioned components.

4.2 Architecture

The proposed architecture can be seen in Figure 1. Each component is described in the following subsections.

4.2.1 Recurrent Data Acquisition. In order to analyse the evolution of communities within Twitch, we chose to focus on follow relationships as a way to quantify connections between individuals. Users on twitch can follow streamers whose content they enjoy watching in order to be notified when they are streaming. Thus,

[1] https://dev.twitch.tv

total	62696				
pagination cursor	eyJiIjpudWxsLCJhIjp7IkN1cnNvciI6IjE1NTI0OTI4MTM1Mjg4NTI5OTkifX0				
data	*from_id*	*from_name*	*to_id*	*to_name*	*followed_at*
	423680558	tastzop	43255578	GRamers	2019-03-16T12:55:11Z
	420501885	stam1228	43255578	GRamers	2019-03-16T09:31:59Z
	...	...	...	...	..

Table 1: An example of Twitch API followers endpoint response for the channel "GRamers". The results contain the total number of followers, the pagination cursor to use for the next request and a list of follow relationships containing the (id, name) pairs of the follower and the channel, as well as the date that the follow was activated.

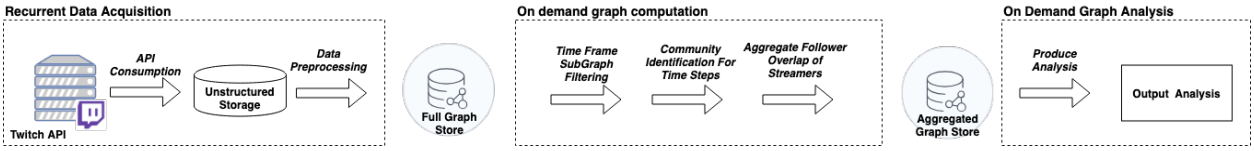

Figure 1: Architecture design of the proposed system. There are three main parts. (1) Recurrent Data Acquisition, which continuously streams data from Twitch and enhances the total graph, (2) On Demand Graph Computation which is used to produce and store aggregated subgraphs per time window and (3) On Demand Graph Analysis, which accesses the subgraphs to perform community analyses, visualisations, applications, etc.

the number of shared followers between two streamers can be used as an indicator of the overlap between their audience. Alternative metrics such as subscriptions and viewers could also be used to provide insight but our research determined that this data would be more difficult to obtain and focusing on followers would facilitate a less demanding analysis framework.

Therefore, we determined that the data ingestion required for our project is the acquisition of all follower relationships on Twitch as well as the time at which each follow was initiated by the user. After the initial ingestion process, the system should periodically obtain the most recent follows.

The Twitch API conveniently provides a function which returns a list of followers for a given streamer ordered by follow date which can be used for this purpose. As the API is consumed, the data is staged in unstructured storage. A relational database could also be used in this case but it is not preferable, since this is just a staging phase and neither persistence nor structure is required. Moreover, a relational database is more expensive than simple unstructured storage.

Having obtained every following relationship between twitch users, this data can then be processed to obtain a representation as a binary directed graph where every node is a user and every edge is a follow. To achieve this, the data undergoes cleaning and preprocessing and is transferred to a Graph store.

These steps are repeated periodically at a fixed time interval in order to have an up-to-date graph representation of the Twitch follower network.

4.2.2 On Demand Graph Computation. The second section of our architecture consists of On Demand Graph Computation. In order to answer our stated research question, we determined that our system should be able to provide a snapshot of the Twitch follower network for a given time slice on demand such that we might extract insights into the development of communities. We determined that this was achievable from the graph store using the following 3 steps:

- Time Frame Subgraph Filtering
- Community Identification on the Subgraph
- Aggregate Follower Overlap of Streamers

The result of this is an aggregated undirected non-binary graph representing the follower network for a given time slice - The nodes in this case are still users but the weight of the edge between two nodes is defined as the number of mutual followers they have. Community identification algorithms are applied to the filtered subgraph prior to aggregation in order to enrich the aggregated subgraph with additional information about each node. The aggregated subgraph is then stored in the same format as the total graph and can later be used to produce analyses.

4.2.3 On Demand Graph Analysis. The final step within the architecture is the access of the subgraphs from the aggregated graph store and the generation of analyses from it. In this phase end-users can query and load different subgraphs from different time slices in order to analyse and visualise the evolution of each community in time. Furthermore, they can aggregate graphs from different communities, filter them by streaming category and export the results in different presentable formats.

5 IMPLEMENTATION

In this section we outline the proposed technical implementation of the architecture described in Section 4, provide insight into the scalability challenges and justify our decisions on methods, tools and software. Figure 6 shows an overview of the proposed implementation. We start by describing a small prototype implementation and then proceed in explaining how to make it scalable.

5.1 Prototype Implementation

Due to technical limitations it was not feasible to mine the whole Twitch relationship graph, so we validated the proposed architecture on a smaller scale prototype. We decided to use the Greek-speaking community as a proxy for the overall site behaviour. The reasons for this decision are as follows:

(1) *Small but Active*: we are able to mine the whole following history of the top 84 Greek channels in a feasible time. It continuously evolves, following the latest streaming trends.
(2) *Covers whole spectrum*: the mined channels stream all kinds of content, from competitive gaming to social chatting.
(3) *Dense*: there is a significant follower overlap between most of the streamers thus enabling us to conduct community analysis without access to the whole graph.
(4) *Homogeneous*: Streamers mostly broadcast in Greek language, therefore they are more likely to be followed by Greeks.

Figure 2 shows the process flow of the prototype implementation. The workflow is divided into four different stages.

5.1.1 Scraping Top Streamers. One of the implementation challenges we faced was creating a comprehensive list of influential streamers in order to mine their followers. The Twitch API has an endpoint for streamers who are currently live in a certain language, but does not provide the top streamers in terms of followers, viewers, etc. To obtain this information, we used scrapy[2] to crawl data from TwitchTracker[3], an aggregator for streamers and their statistics. We created a pool of influential Greek streamers by scraping data from the following channels: "Most Followers", "Most Viewers", "Rating", "Hours Watched" and "Total Views" and saved them to a .json file. Since these channels can contain the same streamers multiple times, we process the data to remove duplicates and then convert it to a more universal file format(.csv). Finally, we update this pool daily by adding new streamers found in the aggregator.

5.1.2 Mining Followers. To create the community graph and perform analysis, we mine the followers of each of the influencing channels in the streaming pool using the Twitch API. To obtain the data we run a python script that consumes the *followers* endpoint and sends requests with the 'from_id' parameter set to that of the streamer. Each request returns 100 followers and a pagination cursor to use for the next request (see Table 1). For each streamer a csv file is created with all their followers and the day they started following them. These files are also updated on a daily basis. Using rotating applications and bearer tokens we are able to mine around 250,000 relationships in an hour, on a single process. For the purpose of this prototype we mined more than 700,000 follow relationships between 300,000 accounts, 84 of which are streamers. This captures the whole evolution of these streamers' communities in the past 8 years. Table 2 shows a summary of the mined data.

Property	Count
Nodes:	330515
Edges:	785619
Streamer Nodes:	84
Account Nodes:	330515
Follow Edges:	779104
Streamer Overlap Edges:	6515

Table 2: Dataset used to create the prototype. All relationships were mined from the Twitch API. Streamer Overlap Edges were calculated using cypher in Neo4j.

5.1.3 Creating Full Graph Database. Once all followers of the channels are collected, we move on to preprocessing, aggregation and creation the final graph. As Table 1 shows, follow relationships are already downloaded in a binary graph format. However, we noticed that sometimes there is useless noise, probably caused by legacy software. For example, for some streamers a follower can appear multiple times (either because they are a returning follower or they were suspended for some time). Occasionally a row will contain *from* and *to* ids but will not have information about the name of the follower (probably caused by a permanent suspension). We clean the files of all noise and aggregate them into a single csv file, which is then imported to Neo4j as a graph database. We chose Neo4j because it comes in a free, open-source community version accessible through a desktop application user interface. Neo4j provides the ability to receive raw data into its graph database management tool that will make performing graph computations much simpler.

5.1.4 Time Slicing. To create different snapshots of the graph in time, we use the Cypher query language[4] which is built into Neo4j. These snapshots contain all the streamers and their followers at that time. Figure 3 displays the first snapshot of the Greek community, at 31/01/2013. Blue nodes represent streamers. In the middle we can see the shared follow relationships between the streamer nodes.

[2]https://scrapy.org
[3]https://twitchtracker.com
[4]https://neo4j.com/developer/cypher-query-language/

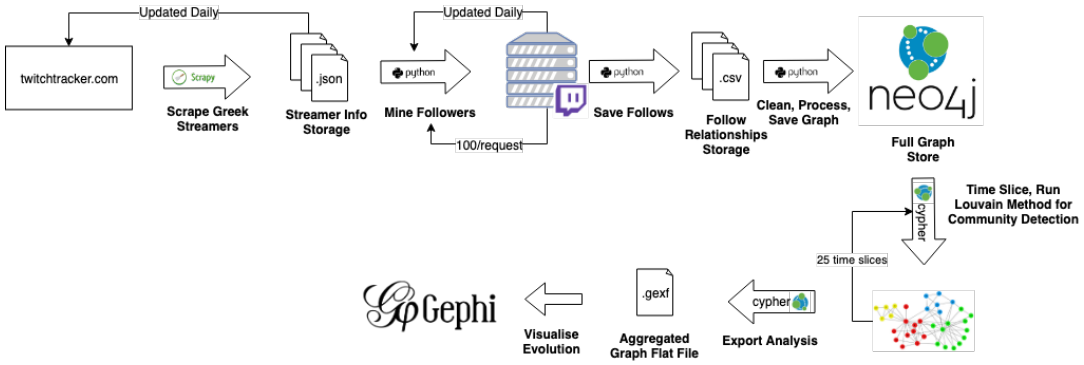

Figure 2: Process flow of the prototype implementation. Involves data collection, preprocessing, graph creation, time slices filtering, community detection and evolution visualisation.

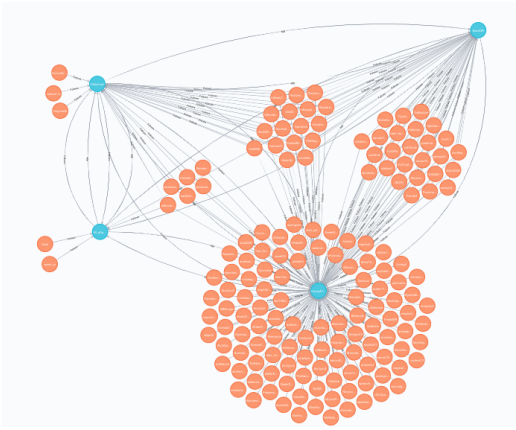

Figure 3: A snapshot of the Greek community in 31/01/2013. The blue nodes represent the streamers and the orange nodes their followers. Follower nodes in between represent shared relationships. In the aggregation phase these nodes are merged and displayed by a weighted edge between the streamers.

5.1.5 Community Detection. To detect communities in the created snapshots, we use the Louvain Algorithm[6] for community detection. The method maximises a modularity score for each community, where the modularity quantifies the quality of an assignment of nodes to communities by evaluating how much more densely connected the nodes within a community are, compared to how connected they would be in a random network. It is one of the fastest modularity-based algorithms and works well with large graphs. The modularity is a value between -1 and 1 and is defined as follows:

$$Q = \frac{1}{2m} \sum_{ij} \left[A_{ij} - \frac{k_i k_j}{2m} \right] \delta(c_i, c_j) \tag{1}$$

Where A_{ij} is the weight of the edge between nodes i and j, k_i and k_j the sum of the weights of the edges attached to nodes i and j, 2m is the sum of all the edge weights in the graph and c_i and c_j are the communities of the nodes. In our case the graph is binary so the weights of the nodes are always 1.

The Louvain algorithm is a time independent community detection algorithm. This means that at every time step the Louvain algorithm allocates a random community ID to the communities it has identified and as such it is not possible to keep track of how the communities move through time by just running the Louvain algorithm on snapshots. We developed a naive method to identify and track community movement over snapshots of time. This algorithm can be found in appendix A algorithm 1; but a summary is provided bellow;

(1) Find time independent communities for each time slice
(2) Determine the most important node per community at time $t-1$
(3) Use the most important nodes per community from $t-1$ to remap the community codes at time t

5.1.6 Analysis Visualisation. Following the community detection and the graph aggregation, we import the graph file into Gephi to perform visualisations. Figure 4 shows a snapshot of the Greek community on March 2019. All 84 streamers are represented in the graph. The more shared followers between streamers the bolder the edge is.

In the following section we present the graph analysis results on the Greek community and how it evolved from 2013 to March 2019.

5.2 Results and Analysis

We analysed the evolution of a select group of 83 Greek streamers beginning in 2013, the year of Twitch.tv's inception. From this date we could see the growth of a only a few active channels to the full 83 currently active. Examining various time slices we can see limited connectivity across these early streamers as Twitch popularity was in its infancy. Beginning as early as 2014 and 2015 we can see the start of a community building as a handful of popular Greek Youtube content creators took to Twitch as a new/additional platform for content distribution, also an indicator that creators bring their community with them when moving platforms. As the broadcasters and viewers settled into the platform, the communities began approaching a state of maturation. We can see in 2019 there

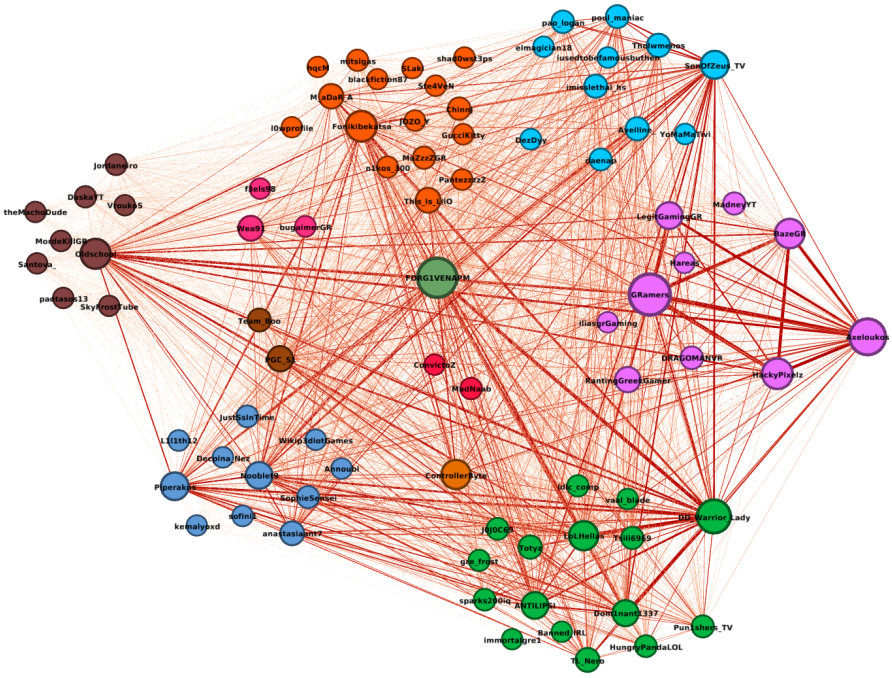

Figure 4: Snapshot of the Greek community in March 2019. Nodes represent prominent streamers and the edges are weighted based on followers overlap. 8 different communities are identified. Apart from game and genre related communities, the system also identifies other organic ones, like youtubers.

are 11 clearly identified community groupings. After examining the content of these channels we are able to confirm that there is notable separation based on games played. For example, the dark blue circles in Figure 4 is dominated by streamers who play the game League of Legends, whereas the orange coloured grouping represents streams that mostly play a category of game known as First Person Shooter (FPS). Furthermore, the Youtube creators community appears to have resisted in time, remaining very active and still dense (pink nodes). This acts as a proof that the system not only confirms game-specific communities but can also identify other, differently related groups of streamers.

Examination of the groupings and relationships further highlights an important aspect of Twitch culture, this being that the most popular channels tend to share followers irrespective of the games they play. This could be viewed as an example of the Pareto Principle, in which a small selection of channels retain a majority of viewers, typically sharing many followers. Figure 4 illustrates how each community has one or more dominant channels which share follower overlap with the other community leaders. Additionally, we have also found that there are communities formed around specific individuals, independent of games played. This is a common phenomenon on Twitch where popular streamers may frequently collaborate with smaller streamers, resulting in the growth of the small collaborating channels. From here additional communities are formed on the basis of collaborating streamers in a clique-like fashion.

Because we scraped only currently active channels we believe we may not have a fully accurate view of the state of Greek streams starting in 2013. Channels that were once active and are not anymore may not be included in this subset, although this does give some insight into how these channels have grown in time as well as how the Greek streaming subset has grown into the matured state it is in currently. Figure 5 demonstrates this and the inflow of new viewers in the overall subsect. It is also particularly interesting to note the appearance of a single community identified in one slice but disappears later as groups become absorbed into other communities. In order to get an accurate measure of how the overall subset evolves in time, we would need to collect channel data from now and henceforth. This way we could see if active channels go into decline and disappear as well as taking in new growing channels.

In the following section we describe how the system can further be scaled by proposing an implementation design based solely on automatically scalable cloud providers.

6 ADDRESSING SCALABILITY CHALLENGES

In the previous sections we proposed an architecture of a system that performs on demand graph analysis on Twitch and then validated it with a prototype. In this section we address the scalability challenges of the system and suggest an implementation design that can effectively overcome these issues.

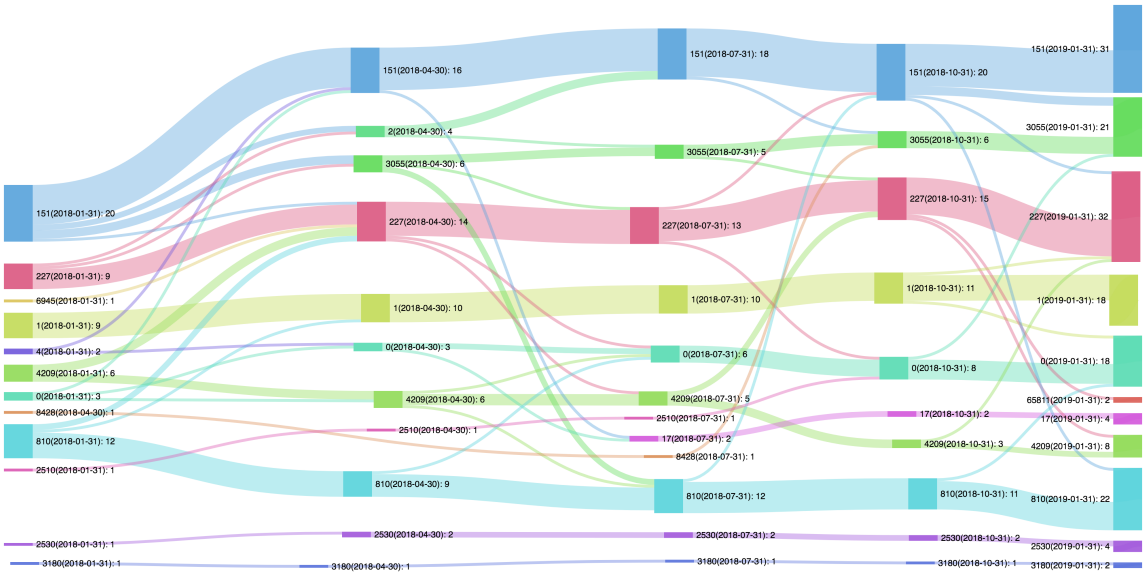

Figure 5: Community Evolution for 5 time slices between 2018 and 2019 Q1. Left hand number represents community ID. Right hand number represents number of streamers in group. Middle value represents date of observation

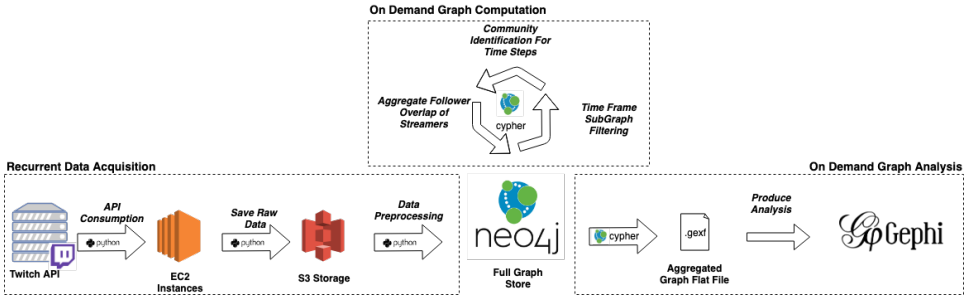

Figure 6: Scaled implementation of the system architecture. All three different processes are unfolded and represented by cloud services that are used to perform the computations.

6.1 Implementation of Data Acquisition

The rapid growth of Twitch calls for a process that continuously and effectively captures new information as it is created. Since Twitch does not set a usage limit for consuming their API, parallel Amazon Elastic Compute (EC2) instances can be used with different bearer tokens in order to discover new streamers and mine follower data. EC2 provides cheap and scalable computation, while it charges only for the time each instance is running. Moreover, since the API consumption processes are computationally cheap, the simplest EC2 tier is needed, therefore the cost is very small.

For each streamer a .csv file is created containing all of their followers up until the time of the data mining. Since these files can be very large, we propose the use of Amazon Simple Storage Service (S3) to store them. S3 guarantees automatic scalability, availability and security, as well as high performance. In this stage we only store raw text data thus S3 ensures that the cost of storage will not explode.

6.2 Graph Store

At this stage the graph is a fully connected binary graph with nodes being accounts(either streamers or spectators) and edges being follow relationships. From within Neo4j, we can leverage its native implementation of the Cypher Graph Query language to manipulate and query the data. Neo4j has a High Availability (HA) extension for use in cluster computing environments [4]. This extension ensures that Neo4j is scalable in the following ways:

- Full data redundancy
- Service fault tolerance
- Linear read scalability

The Neo4j HA cluster comes in the form of a master-slave architecture. The system has two primary components, the database itself and the cluster management component. In the Neo4j HA cluster, the full graph is replicated on each node, ensuring that the data will always be safe as long as a single node still stands. Naturally, this means that one will need enough storage per node

to store the entire graph; note the database requirements in Table 3. Fault tolerance is handled by the cluster's automatic master election. This ensures there is always a master node to manage transactions. Further, write operations are coordinated by the master node but writing can still be done by slaves while syncing with the master. Similarly, read operations can be performed by slave nodes but are not limited by a master node sync and thus by adding nodes one can linearly scale read capability.

Nodes	Edges	Size On Disk	Num Cores
300,000.00	700,000.00	1.5	3
1,000,000.00	3,000,000.00	5.7	3
2,000,000.00	6,000,000.00	11.4	3
5,000,000.00	15,000,000.00	28.5	4
10,000,000.00	30,000,000.00	56.9	4
15,000,000.00	45,000,000.00	82.2	4
20,000,000.00	60,000,000.00	115.1	5
25,000,000.00	75,000,000.00	139.5	5

Table 3: Estimates of Neo4j Database Requirements

6.3 Implementation of Graph On Demand Computation

The goal of our system is to enable exploration of the evolution of communities in time using snapshots of different time slices. This whole process is done within Neo4j, leveraging Cypher and built in graph computation algorithms. We use an iterative process comprising the following steps: (1) Filtering by time slice, (2) Community identification and (3) Aggregation of follower overlap between streamers.

After the snapshot is built and the communities are identified, the edges are aggregated into weighted relationships between streamers. At this stage the subgraphs are weighted and undirected, with nodes being streamers and edges being follow overlap between them. These subgraphs are then saved in flat graph files (.gexf) on S3.

We propose the continual usage of Neo4j because it would mean an easy extension of our prototype and because it is able to scale out very well for large graph networks(as per the previous section). We are also confident that the compute required will linearly scale as the network grows due to the Louvain algorithm being highly efficient with a complexity of $O(m)$ [8]; where m is the number of nodes.

Work done by s. Shanbhaq [8] investigated the performance of the Louvain algorithm, see Figures 7 & 8 in appendix B. He also proposed a more efficient implementation of the algorithm that yielded even faster run times, meaning that a more efficient implementation is also available if resource restrictions are present. If alternatives are needed for the scaled out compute then the more efficient algorithm could be implemented in the following alternative technologies which allow for graph compute; Spark Graphx [5] and SNAP [6].

[5]https://spark.apache.org/graphx/ - deprecated
[6]http://snap.stanford.edu/

6.4 Implementation of Graph Visualisation

We are not aware of any free network visualisation tools that are able to easily plot large networks of more than 100000 nodes. We believe that it does not make sense to plot the network with that many nodes in order to visually understand how communities evolve. Instead, we recommend that visualisation be performed on subgraphs of the aggregated graph. For example, subgraphs and visualisations could be built around specific countries, or top n streamers, etc. We recommend the use of Sankey diagrams (as in Figure 5) if the behaviour of more than 100000 nodes is required. This would scale easily because the nodes would be combined into aggregated community statistics.

At the moment, research and limited proprietary solutions exist that use GPU technology to visualise very large network graphs. Graphistry [7] report that they are able to plot very large networks by offsetting the graphical workload onto GPU hardware on their servers and then serving it to a client; but we have not been able to assess how well this solution works for our use case.

7 CONCLUSION AND FUTURE WORK

Here we present a summary of our work and results in addressing the research questions mentioned in Section 1. First, we successfully designed a system to mine and store large scale graph data and perform on-demand community analysis. We then validated this system through the use of a prototype for analysis of the Greek Twitch community which was able to capture the dynamic evolution of the community in time. To do this we utilised Louvain Modularity for community detection over time slices of our data, which did in fact give some insight into the growth and development of relationships between the channels we selected. Finally, we delineated and justified the theoretical implementation of a scaled prototype which could be used to analyse the global Twitch community.

Through analysis of graphs generated using our prototype system we were able to extract various insights about the Greek Twitch community. We found that streamers tended to be clustered together based on the types of game they broadcasted. Additionally, we noted that content creators from other platforms brought their communities with them when they transitioned to Twitch and remained closely connected. Further examination also revealed that larger channels tended to have many shared followers regardless of the type of content they produced.

Future work could consist of implementing the scaled prototype described in Section 6. This would comprise the use of Amazon EC2 instances and S3 storage in order to deal with increased computational and storage requirements. Furthermore, benchmarking of different technologies could be used to determine the ideal combination of entities for the scaled up system. Application of this system to the entire Twitch user base could then be utilised to extract novel insights into the structure and evolution of the global Twitch userbase as a whole.

8 TEAM MEMBERS CONTRIBUTIONS

All 4 members contributed equally to all parts of the project, including design development, report writing and presentation.

[7]https://www.graphistry.com/how-it-works

REFERENCES

[1] Benjamin CB Churchill and Wen Xu. The modem nation: A first study on twitch. tv social structure and player/game relationships. In *2016 IEEE International Conferences on Big Data and Cloud Computing (BDCloud), Social Computing and Networking (SocialCom), Sustainable Computing and Communications (SustainCom)(BDCloud-SocialCom-SustainCom)*, pages 223–228. IEEE, 2016.

[2] William A. Hamilton, Oliver Garretson, and Andruid Kerne. Streaming on twitch: Fostering participatory communities of play within live mixed media. In *Proceedings of the 32Nd Annual ACM Conference on Human Factors in Computing Systems*, CHI '14, pages 1315–1324, New York, NY, USA, 2014. ACM.

[3] Mark R. Johnson and Jamie Woodcock. âĂŸitâĂŹs like the gold rushâĂŹ: the lives and careers of professional video game streamers on twitch.tv. *Information, Communication & Society*, 22(3):336–351, 2019.

[4] David Montag. Understanding neo4j scalability, Jan 2013.

[5] G. Nascimento, M. Ribeiro, L. Cerf, N. CesÃąrio, M. Kaytoue, C. RaÃŕssi, T. Vasconcelos, and W. Meira. Modeling and analyzing the video game live-streaming community. In *2014 9th Latin American Web Congress*, pages 1–9, Oct 2014.

[6] X. Que, F. Checconi, F. Petrini, and J. A. Gunnels. Scalable community detection with the louvain algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 28–37, May 2015.

[7] Elliot Star. Visual mapping of twitch and our communities, 'cause science!, Feb 2015.

[8] Christian L. Staudt, Michael Hamann, Alexander Gutfraind, Ilya Safro, and Henning Meyerhenke. Generating realistic scaled complex networks. *Applied Network Science*, 2(1), oct 2017.

A APPENDIX - CODE

B APPENDIX - FIGURES

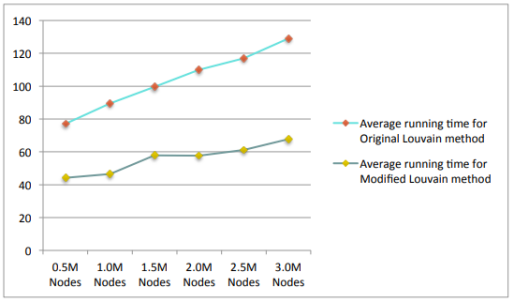

Figure 7: Execution Time (seconds) Vs Number of Nodes[8]

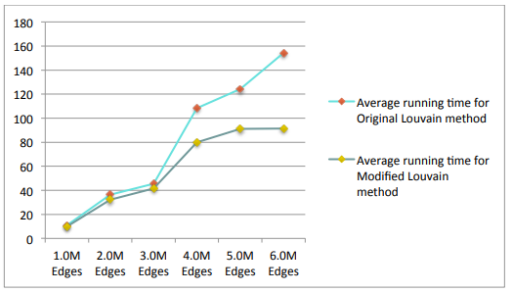

Figure 8: Execution Time (seconds) Vs Number of Edges[8]

Algorithm 1 Tracking Community Evolution

n := streamers or follower node
e := edge relation between two nodes
$d(e)$:= date when edge x (follow) was created
$c(t, n)$:= community of n at date t
V := set of Nodes n
E := set of edges e
$C(t)$:= set of node communities $c(t, n)$ at date t
Define a Graph $G = (V, E)$

Louvain(g) := Louvain Community Detection for graph g

dates := list of date intervals
for (each date t in dates) **do**
    E_s := subset of edges E where $d < t$
    V_s := subset of nodes V that have edges in E_s
    $G_s = (E_s, V_s)$
    C(t) := Louvain(G_s)

Comment: Identify how community groups change from $t - 1$ to t and reassign the community ID numbers at time t to align them with that of $t - 1$
for (each date t) **do**
    $C_{old} := C(t - 1)$

    Comment: Account for similar community numbers between t and $t - 1$ that are coincidentally equal due to temporal in dependant Louvain community number allocation
    $C_{now} := C(t)$
    false_community_numbers := $C_{now} \cap C_{old}$
    for (community c in false_community_numbers) **do**
        c := random number $\neq \forall x \in C_{old}$
        Update $C(t)$ with c

    Comment: Find the nodes that represent the communities for each date $t - 1$ through degree of centrality
    for (each community c in C_{old}) **do**
        E_s := subset of edges E where $d < t$
        V_s := Nodes that exist in C_{old}
        $G_s = (E_s, V_s)$
        n_{best} := node n with best degree of centrality in G_s
        $V_{central} \leftarrow$ append n_{best}

    Comment: Account for cases where more then one node in n_{best} (at time $t - 1$) move to the same community at time t
    $C_{new} := C(t)$
    for community c in C_{new} **do**
        V_c := Nodes that exist in c
        $V_{\text{conflict pairs}} \leftarrow$ apprend $V_{central} \cap V_c$

    for $V_{\text{conflict pair}}$ in $V_{\text{conflict pairs}}$ **do**
        $V_{remove} \leftarrow$ append nodes from $V_{\text{conflict pair}}$ with the least number of nodes in their respective corresponding communities in C_{old}
        $V_{central} \leftarrow$ remove nodes in $V_{\text{conflict pairs}}$

    Comment: Reassign community IDs at time t by using the community ID of central nodes from $V_{central}$ (from time $t - 1$)
    for node n_c in $V_{central}$ **do**
        $c := c(t, n_c)$
        V_s := Nodes with community equal to c
        for node n_r in V_s **do**
            $c(t, n_r)$:= c