Youtube analysis and simple video recommender

Christian Burke
Marcus Blaisdell
Linh Nguyen
Thang Nguyen

December 16, 2018

Taking Youtube data to design, test and implement functions to perform on their large datasets. Functions analyzed and included are degree distributions, categorize statistics, top-k queries, user identification in recommendation patterns, range queries, and implementation of a PageRank algorithm. Optimization and time complexities are explored and large data queries are discussed.

INTRODUCTION

Youtube is a video sharing website. Users create an account on the website and can then upload their own videos and watch videos that others have uploaded. The site provides a platform that allows users to provide comments on videos they have watched. Users can search for videos by keyword and the site will provide a list of available videos that match that keyword. Youtube also makes recommendations based on viewer history of videos the user may find to be of interest. In the 2007-2008 time frame that the data used in this study was collected, Youtube used a five-star rating system for users to rate a video. This is the format of the data we analyzed. They have since switched to a like/dislike system.

According to Omnicore[1], Youtube has 1.9 billion active users per month with 30 million of those active daily. Users watch five billion videos daily. There are 300 hours of video uploaded to the site each minute. Big data is classified by four elements, volume, variety, velocity, and veracity. The Youtube data set meets two of these criteria. By volume, there is a large amount of data available in billions of videos. By velocity, a large amount of new data is being created each minute. The challenge is in organizing this data so that it can be easily located when searched for and in analyzing the data to provide a recommendation of a set of videos to watch to a user.

PROBLEM STATEMENT

With a large number of videos in the library and new ones being added at a steady rate, providing up-to-date metrics and providing recommendations is crucial for Youtube. In this project we aim to implement a Youtube data analyzer supported by SQL and graph algorithms which provides basic data analytic functions to Youtube media datasets. It will provide functionality to efficiently report degree distributions, categorize statistics, top-k queries, user identification in recommendation patterns, range queries, and implement PageRank algorithms.

Since the dataset of Youtube is so large, it is difficult to perform simple queries such as top-k queries quickly with basic tables and queries. We need to apply various techniques in order to get speed-ups and feasible access times to these types of queries. These problems are important because Youtube for example will want to quickly recommend videos to users and perform searches with constraints (range queries). The more optimized these queries get, the more users and amount of queries Youtube will be able to handle and potentially get lots of profit because there will be less servers needed to support their customer base.

Youtube could potentially use the algorithms in this paper and apply it to their databases and search engines. Youtube has large amounts of data with accessible information, thus making it a good target for practicing optimization techniques. This will be the main purpose in answering these problems, that we main gain skills and experience to apply in future opportunities and problems.

DATASETS AND TOOLS

videoID	*an 11-character string that uniquely identifies the video*
uploader	*A character string of the name of the user that uploaded the video*
age	*An integer of the number of days since the video was updated to the cutoff date of February 15, 2007 to normalize the numbers across the two years of data collection*
category	*A string containing the category a video was classified as*
length	*An integer of the number of minutes of the length of the video*
views	*An integer of the number of views the video has*
rate	*A float of the rating the video has (Scale: 0-5)*
ratings	*An integer of the number of ratings a video has*
comments	*An integer of the number of comments a video has*
relatedIDs	*An array of 20, 11-character strings that uniquely identifies 20 videos that have been classified by YouTube as being related to the specific video being evaluated*

Figure 1: Data format

The data we used for testing contains basic statistical information about individual videos that was obtained by crawling the Youtube website on 98 separate days over the course of two years[3]. It is described by ten attributes (Figure 1). All the data from this resource was organized into files denoted by depth level. The data gatherers crawled Youtube for different amounts of time and recorded the videos found in files 0.txt, 1.txt, 2.txt and 3.txt, representing the depth from the original videos given to the crawler. For all combined depth 0's, the set contained 17,728 unique videos. For all combined depth 3, the set contained 6,795,767 unique videos.

We used both mySQL and Neo4j to address these problems and run various queries on the existing data set. "Connector/CPP" is a library used for mySQL. This library allows for writing C++ code that can interact with mySQL databases. MySQL was used to make a video recommender, top-k queries, and range queries. We used Neo4j to address the same problems as mySQL with the addition of degree distributions, categorizing statistics and implementing a PageRank algorithm.

Our team used two different methods to load the data into mySQL databases. One method used was a C++ custom parser to upload the data to mySQL. Fortunately very little parsing was performed on the data as it was already available as an ordered, tab delimited text file. It took ~3s to load each 0-depth table. With 93 tables, the total time was ~4m 40s. The 3-depth tables took from 4s to 32s each to load. The other method used to load the data was to use standard mySQL command "LOAD DATA" for reading data from a file. An awk script was used to create files that contain different layouts of the data for different tables for easy loading.

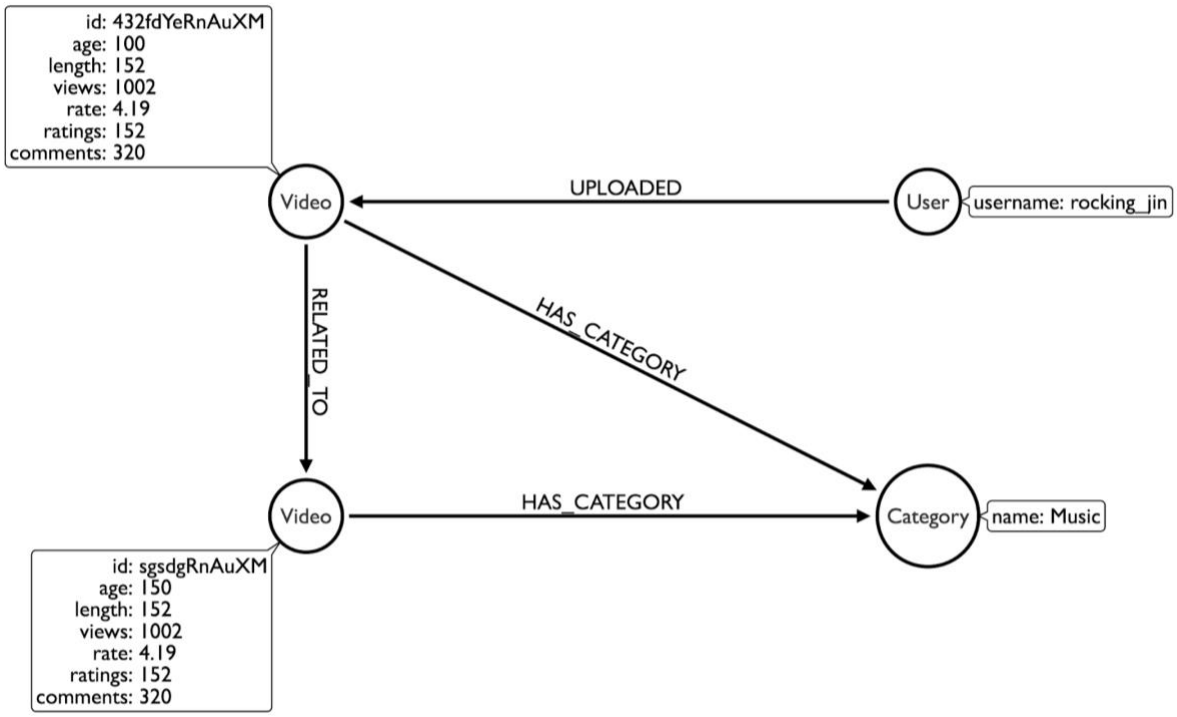

Figure 2: Data workflow and relationships for Neo4j data

We are using Neo4j as our second approach for this project in addition to MySQL. To import data to Neo4j, two Python scripts and one Cypher script were created:

- Conf.py: configuration python script for first setup of Cypher
- Cypher_generator.py: created Cypher queries, execute Cypher, and generate new Cypher script in it.
- DataLoad.cypher: importing a whole dataset to Neo4j. Using Neo4j library apoc.periodic.iterate which will go through each file in the dataset and imports it into Neo4j.

In comparison with MySQL, Neo4j makes things easier to figure out the relationship between videos since relationships take first priority in graph databases. Not only does Neo4j effectively store data relationships; it is also is flexible when expanding a data model or conforming to changing needs. Figure 2 shows the relationships and how the data is laid out.
Complete Neo4j Code is available on GitLab.

DataBase Information:

- Node Labels: (*total: 23023017527) Category, UserUser, Video
- Relationship Types: (*total: 93644590) HAS_CATCATEGORY, RELATED_TO, UPLOADED
- Property Keys: age, comment, id, length, name, pagerank, rate, ratings, username, views

SOLUTIONS

Part A: Network Aggregation:

Users upload many videos on Youtube everyday with unique content. We try to find connection between them to efficiently rank them. Each video has its own id, views, length, category, upload user's name, etc. We made use of that and efficiently report the statistics of Youtube network based on degree distribution and categorized statistics

1. Degree distribution:

Max in-degree and out-degree

For the degree distribution, we try to find the in-degree and out-degree connections. The connection of all categories in and out are the degree of node in the network. After discovered NEO4J, we find it extremely useful with all its function and graph. We used the function MATCH from NEO4J to calculate the degree distribution.

Here is the NEO4J code to find the in-degree, out-degree and rank them.

```
// In-Degree:
MATCH (n:Video)<-[r:RELATED_TO]-(:Video)
WITH n, COUNT(r) AS in_degree
WITH in_degree, COUNT(in_degree) AS frequency
RETURN in_degree,frequency
ORDER BY frequency DESC LIMIT 10

// Out-Degree:
MATCH (n:Video)-[r:RELATED_TO]->(:Video)
WITH n, COUNT(r) AS out_degree
WITH out_degree, COUNT(out_degree) AS frequency
RETURN out_degree,frequency
ORDER BY frequency DESC LIMIT 10
```

Figure 3: Neo4j In-degree/Out-degree queries

Where it counts all the in-degree, out-degree distribution between nodes and consider degree as frequency and then order the frequency from highest connection to lowest then we limit the output to just 10 highest.

Average Degree

This part simply calculate the average in and out degree of each video in the whole dataset.

```
// Average in-degree
MATCH (n:Video)<-[r:RELATED_TO]-(:Video)
WITH n, COUNT(r) AS degree
RETURN avg(degree) AS average_in_degree

// Average out-degree
MATCH (n:Video)-[r:RELATED_TO]->(:Video)
WITH n, COUNT(r) AS degree
RETURN avg(degree) AS average_out_degree
```

```
// Maximum in-degree
MATCH (n:Video)<-[r:RELATED_TO]-(:Video)
WITH n, COUNT(r) AS degree
RETURN MAX(degree) AS maximum_in_degree

// Maximum out-degree
MATCH (n:Video)-[r:RELATED_TO]->(:Video)
WITH n, COUNT(r) AS degree
RETURN MAX(degree) AS maximum_out_degree
```

Figure 4: Neo4j Average and Maximum in-degree and out-degree queries

2. Categorized Statistics

With Neo4j, it is very straightforward to calculate frequency of videos partitioned by a search condition: categorization, size of videos, view count, rate, comments count, video uploaded count.

```
Category wise distribution :
 MATCH (n:Category)-[r:HAS_CATEGORY]-(:Video)
 WITH n, COUNT(r) AS video_count
 RETURN n AS Category, video_count
 ORDER BY video_count DESC LIMIT 10
```

```
Video Rate wise distribution
MATCH (n:Video)
WITH n.rate AS rate, COUNT(n) AS video_count
WHERE rate IS NOT NULL
RETURN rate, video_count
ORDER BY rate DESC
```

Figure 5: Neo4j categorization queries

We also be able to to generate the output to csv file. Here is the sample of output taken from running the query to get the statistic based on the category of the video: The output display in graph style and chart style

```
Video Rating count wise distribution
MATCH (n:Video)
WITH n.ratings AS ratings, COUNT(n) AS video_count
WHERE ratings IS NOT NULL
RETURN ratings, video_count
ORDER BY ratings DESC
```

```
Size(length) wise distribution
 MATCH (n:Video)
 WITH n.length AS length, COUNT(n) AS video_count
 RETURN length, video_count
 ORDER BY video_count DESC LIMIT 10
```

```
Video comments count wise distribution
 MATCH (n:Video)
 WITH n.comments AS comments, COUNT(n) AS video_cour
 WHERE comments IS NOT NULL
 RETURN comments, video_count
 ORDER BY comments DESC
```

```
View count wise distribution :
 MATCH (n:Video)
 WITH n.views AS views, COUNT(n) AS video_count
 WHERE views IS NOT NULL
 RETURN views, video_count
 ORDER BY views DESC
```

```
Users upload count wise distribution
 MATCH (n:User)-[r:UPLOADED]-(v:Video)
 WITH n.username AS User, COUNT(v) AS uploaded_videos_count
 RETURN User, uploaded_videos_count
 ORDER BY uploaded_videos_count DESC
```

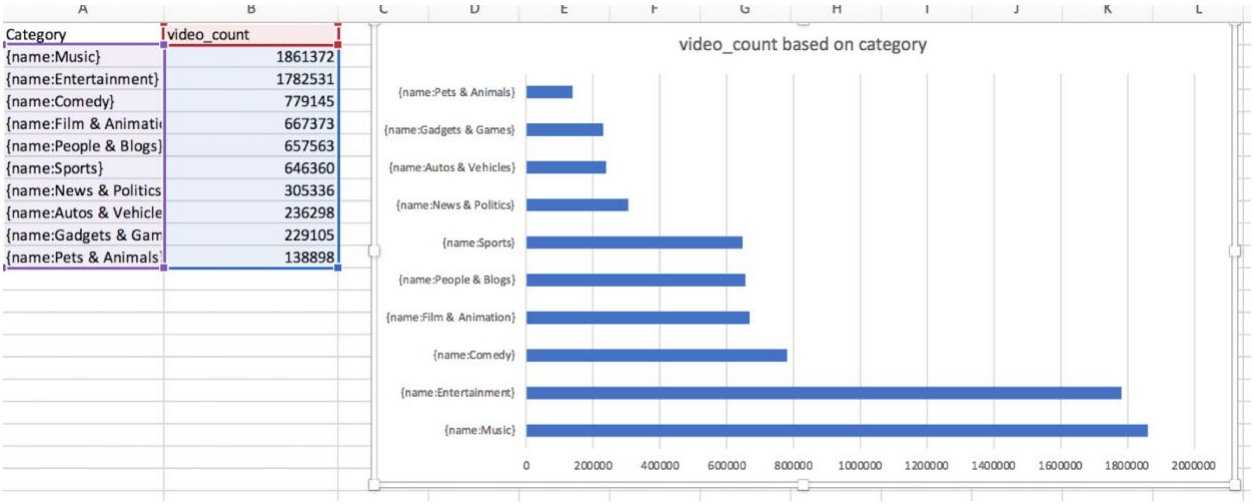

Figure 6: Sample csv output from Neo4j query

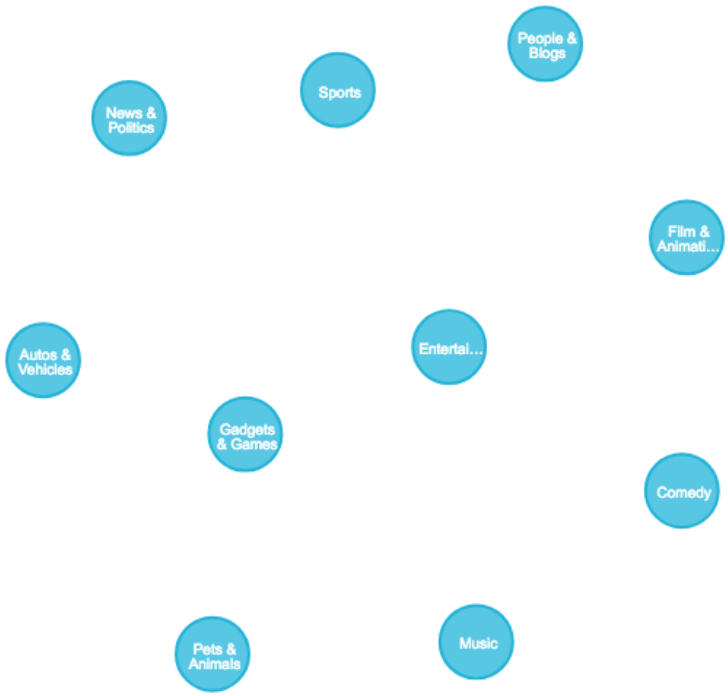

Figure 7: Neo4j output of video_count based on category in graph style

3. Top-k Queries

We used mySQL to generate top K queries and develop a simple recommender algorithm. We used the “connector/CPP” library to interface between C++ and mySQL so that we could write all of the code in C++. This offered more functionality beyond the limited tool set that mySQL has. All of these tests were run using a Macbook Pro with in Intel i5 (early 2015) and 4GB of RAM running Linux Mint, 18.3 on a virtual machine. All of the code written for this project is available on GitLab[2].

To analyze the data in mySQL, we began by creating individual, non-optimized, queries for each of three top K searches: top K category, top K rated videos, and top K most popular videos.

Top K Categories: SELECT category, COUNT(DISTINCT videoID) AS myList FROM <table> WHERE category IN (SELECT DISTINCT category FROM <table>) GROUP BY category ORDER BY myList DESC LIMIT 20;

Top K rated videos: SELECT DISTINCT videoID, rate FROM <table> ORDER BY rate DESC LIMIT 20;

Top K most popular videos (Top K most viewed videos): SELECT DISTINCT videoID, views FROM <table> ORDER BY views DESC LIMIT 20;

K was set at 20 because this matches the number of relatedIDs that are provided in the original data set and maintains a consistent evaluation for comparison purposes. Each query was run ten times to get an average time per query.

A query was also created to return a list of videos within a length range: SELECT DISTINCT videoID, length FROM <table> WHERE length > minLength and length < maxLength ORDER BY length DESC; that runs in an average of 90ms.

To attempt to optimize each of the queries, we considered what the information was that was being queried, and how mySQL was handling it to find a way to improve the time. We are using the “ORDER BY” function of mySQL to sort the query results and using “LIMIT 20” to show only the top 20 results. MySQL uses the quicksort function to sort the results in their “ORDER BY” function. Quicksort has a worst case time complexity of $O(n^2)$. For a big data problem, this is considered intractable as the size of the data can grow to billions of tuples. Since we are only concerned with the top 20 items, it was logical that we should only record the top 20 elements and ignore any record that is beyond that scope. With that in mind, instead of recording all of the search results and then ordering the entire list, we created a custom insertsort function (Figure 7) that only records the top K elements in descending order. This theoretically would have reduced the time complexity from $O(n^2)$ to $O(Kn)$ as it only traverses the store list K times and if an element under evaluation is less than the Kth smallest element already in the list, it is ignored. Practical testing did not substantiate this theory and mySQL outperforms this custom insertsort by a notable amount (Table 1). Each of the standard mySQL queries and the ones with the custom insertsort function were run ten times and averaged for comparison. The results are recorded in Table 1. The attempt at optimization resulted in an increase in processing time for all queries from a 4% increase to as much as a 17X increase in processing time.

insertSortRate (ratesList, struct{videoID, rate}):

- *While i < K:*
  - *If ratesList(i).rate < struct.rate:*
    - *Insert(struct{videoID, rate})*
    - *Return ratesList;*
- *Return ratesList;*

Figure 8: Custom insert function (shown for “rate”, the same function is used for view substituting “view” for “rate”)

Running each query independently seems inefficient as they each must traverse the entire data set once to populate their respective lists. To attempt to optimize the top K query, we combined all three queries to run simultaneously to create the lists and generate metrics in one pass (Figure 8) instead of three separate passes. This resulted in a single pass that can generate all three reports however it results in an increased processing time of an extra 42% of the time it would take to run each one individually for non-optimized queries and an additional 20% for the optimized queries.

- *Create a hash table, categoryHash, (to record the category names)*
- *Create a table ratesList (to record the video ratings)*
- *Create a table viewsList (to record the number of views)*
- *For each table:*
  - *For each row:*
    - *Increment the value in the hash table that matches category by 1*
    - *If rate > kth element of ratesList, insertsort into ratesList (struct{vidoeID, rate})*
    - *If views > kth element of viewsList, insertsort into viewsList (struct{videoID, views})*
- *Iterate through the categoryHash table and insertsort into new list, categoryList(struct{category, count})*
- *Print categoryList*
- *Print first K elements from ratesList*
- *Print first K elements from viewsList*

Figure 9: General Top K algorithm - optimized

	Top K Queries								Top K related to a video category	
	Category (us)	Category optimized (us)	popular (us)	popular optimized (us)	rated (us)	rated optimized (us)	all non-optimized (us)	all optimized (us)	Specific (us)	Range
Pass 1	58863	66993	59292	223555	59966	1050542	241130	1578681	183433	86361
Pass 2	63405	63234	59532	227055	56171	994284	264043	1512599	161093	84644
Pass 3	61506	63713	57227	224848	69907	1033910	248990	1658761	163692	88810
Pass 4	57346	65955	53960	222030	58441	987934	248312	1544520	151887	76885
Pass 5	61827	66111	54181	264225	57900	984595	255827	1558521	163602	89783
Pass 6	60323	62576	54347	247171	56890	1011783	259088	1642207	152412	108761
Pass 7	58001	63093	59941	232828	53834	1006094	278217	1518924	161538	98442
Pass 8	64620	60986	60519	230374	56027	1018026	235124	1569531	172808	99051
Pass 9	64435	63985	64012	231868	56170	992149	234228	1539527	155484	89615
Pass 10	59028	60106	61208	221412	53269	1024808	250707	1481324	170038	84061
Average	60935.4	63675.2	58421.9	232536.6	57857.5	1010413	251566.6	1560460	163598.7	90641.3
% of time		104.4962		398.0298		1746.381	141.9558	119.4268		

Table 1: Averaged times for all queries

4. User Recommendation Algorithm

To answer the question “How can we recommend K videos to a user that has just watched one video”, we developed a recommender algorithm to perform a more specific search based on the category of the video a user has just watched (Figure 9).

- *Create a list specificRatesList*
- *For each table:*
  - *For each row:*
    - *If category == videoID.category:*
      - *normalizedRate = ((rate * ratings) / views)*
      - *insertSort into specificRatesList (struct{videoID, normalizedRate})*

- *insertSort (specificRatesList, videoID, rate, views, K)*
- *While i < K:*
  - *If specificRatesList.rate < rate:*
    - *insert(struct{videoID, normalizedRate})*

Figure 10: Recommender algorithm

This algorithm limits the search to videos within the category of a specific video to create a list of K recommendations based on a normalized rate. The normalized rate seeks to create a uniform comparison between videos with varying numbers of views and ratings such that a video with fewer views can be reasonably compared to one with a larger number of views.

We discussed methods to try to make the recommender algorithm targeted to a user but given the limited information we have available, we did not have a means to accomplish this. The data we have available provides information about videos on the Youtube website but does not contain data specific to a user. Designing a recommender algorithm that will target a user will require information about the user. Some types of useful user information would be: user ID, views history, and ratings of videos the user has given. Without user-specific information the best we can do is to rely on general statistics and the limited information available in a single video. The use of the category as a limiting factor is one option and we can use the same logic to recommend videos based on length, rating, or popularity as well.

Correctness analysis: the algorithm is correct and complete. The input is a finite data set. The evaluations are based on the existence of a condition. “videoID” is a primary key, if a record exists, it has a videoID. If a row has a category, insert category into table categoryList. If a row has a rate, insert videoID and rate into rateList. If a row has a views count, insert videoID and rate into rateList. If a row has a views count, insert videoID and views count into viewsList. Output is the populated lists, output directly follows input. If the evaluation condition is not met, it will skip to the next record and produce no output. The algorithm terminates for all conditions. It searches each row of each table. If a row meets the evaluation criteria, it produces an output and moves to the next row. When there is no more input, the program detects this and halts.

5. Range Queries

For range queries, we want to find all videos in category x with duration within a range [t1, t2]. The following is a basic SQL query to do just this.
SQL code for Range Query:

```
1. SET @t1 = 0;
2. SET @t2 = 100;
3. SET @x = "Music";
4. USE youtube;
5. SELECT video_id, length
6. FROM videos
7. WHERE category = @x && length > @t1 && length < @t2
```

This is a basic SQL query which retrieves all videos in the given range of length in a specific category x. Lines 1-3 set the variables that will be input to the query and are directly corresponding to the question asked. Right now basic values have been plugged in for example. Line 4 is specific to our mySQL database system, youtube is the name of our database being used and thus sets youtube as the active database. Line 5 makes two columns with all video_ids and the length of these videos which satisfy the constraints. Line 6 states that the videos table will be the target table for querying. Line 7 has a series of constraints which limit the displayed videos. It guarantees that the videos being displayed are only of category x, length greater than t1 and length less than t2. This gives us a proper range query that does what we desire.

Optimization techniques:
One way to optimize this search is to keep a seperate table with schema <video_id, length> for each category and order by length. First go to the table with the desired category. In this table use binary search to find a video which satisfies the given time constraints [t1, t2]. Then linearly check all videos above and below that video until the constraints are not satisfied.

Complexity analysis:
Using the standard SQL query designed earlier, it is used on an unordered table of videos. Let N be the total number of videos in the table. This means that the algorithm would search every video and compare category and time with t1 and t2. This algorithm would run in worst-case and best-case O(N).

With our optimization idea we can bring some speedups to the previous algorithm. Let c be number of categories and t be number of videos which satisfy the time constraints. Finding the first video which satisfies the time constraints will take $O(log(c))$ because of binary search. To get all other videos which satisfy the time constraints, it will take O(t) because it will linearly search from the first video found. This gives time complexity worst-case O(t) to the optimized algorithm. Since $t \leq N$ we can say worst-case in terms of N is O(N). The best-case would be if t=1 and the binary search immediately finds the only video which satisfies the time constraints. It will check above and below the video and return the found video. The best-case complexity would be O(1) because it immediately stops at that one video. This optimization does not

reduce the worst-case time complexity, however, the best-case is substantially more efficient and can result in instantaneous lookups.

6. PageRank Algorithm

Before running the Pagerank on the dataset, we need to run the “Data Standardization” which convert the properties in required format and any unnecessary details.
The key point of Data Standardize besides converting is to make sure that removing all the self related connection (some video in related field of itself) so it won't affect the result of the graph algorithm.

```
// CONVERT THE DATA IN REQUIRED FORMAT:
// CONVERT SOME PROPERTIES TO INTEGER AND SOME TO FLOAT
// ALSO THERE ARE SOME VIDEOS WITH NO DATA THAT RESULT IN ERROR AND NULL VALUES REPLACE THESE NULL VALUES BY 0 (ZERO).

call apoc.periodic.iterate('MATCH (v:Video) RETURN v',''
WITH v
SET
    v.age = (CASE WHEN v.age IS NOT NULL THEN toInteger(v.age) ELSE 0 END ),
    v.length = (CASE WHEN v.length IS NOT NULL THEN toInteger(v.length) ELSE 0 END ),
    v.views = (CASE WHEN v.views IS NOT NULL THEN toInteger(v.views) ELSE 0 END ),
    v.rate = (CASE WHEN v.rate IS NOT NULL THEN toFloat(v.rate) ELSE 0 END ),
    v.ratings = (CASE WHEN v.ratings IS NOT NULL THEN toInteger(v.ratings) ELSE 0 END ),
    v.comments = (CASE WHEN v.comments IS NOT NULL THEN toInteger(v.comments) ELSE 0 END ),
    v.comments = toInteger(v.comments)
', {batchSize:10000, parallel:true})

// DELETE SELF RELATED CONNECTIONS: SOME VIDEOS APPEAR IN RELATED
// FIELD OF ITSELF SO NEED TO REMOVE THESE TO INSURE CORRECT RESULTS OF GRAPH ALGORITHMS

MATCH (v:Video)-[r:RELATED_TO]-(v:Video)
DELETE r
```

Figure 11: Neo4j data standardization

We wrote Query to calculate the pagerank for each video and write back the results as a property of video (property named "pagerank")
This query will return only the stats for this operation and result of the pagerank will be written in the property of a video
Then writing the result back to the property because we don't know pagerank calculation is a highly computational algorithm and the result is not going to change as our database has fixed number of videos.
Will need to run this if we add more Videos or RELATED_TO relationships between videos

We are using Neo4j graph algorithm library function algo.pageRank[4] to calculate PageRank:

- Pagerank needs multiple rounds of computation to calculate the score, it's defined here by iterations:20.
- write: true states that result of the PageRank score should be written as a property of Node (In our case Node in Video).
- writeProperty:"pagerank" : states the property name to write this score, so this algorithm will store the result in 'pagerank' property of a video.

- Damping factor is a constant used in the algorithm
- The first parameter for this function is the Node Label, in our case, it's Video because we are calculating PageRank for videos.
- The second parameter is the Relationship type to consider for PageRank calculation, we are calculating this on CONNECTED_TO

```
CALL algo.pageRank('Video', 'RELATED_TO',
  {iterations:20, dampingFactor:0.85, write: true,writeProperty:"pagerank"})
YIELD nodes, iterations, loadMillis, computeMillis, writeMillis, dampingFactor, write, writeProperty
```

Figure 12: Neo4j Algo PageRank call

Then we can explore the result of PageRank by using MATCH function:

```
//  Find top k most influence Videos in Youtube network based on pagerank results
MATCH (n:Video) RETURN n ORDER BY n.pagerank DESC LIMIT 25

// To get the specific fields of the top Videos based on pagerank results
MATCH (n:Video) RETURN n.id AS VIDEO_ID, n.pagerank AS Pagerank_Score, n.views AS Views, n.ratings AS Ratings,
    n.comments AS Comments, n.age AS Age, n.length AS Length  ORDER BY n.pagerank DESC LIMIT 25

// If two or more videos have same pagerank then order them by views count
MATCH (n:Video) RETURN n.id AS VIDEO_ID, n.pagerank AS Pagerank_Score, n.views AS Views, n.ratings AS Ratings,
    n.comments AS Comments, n.age AS Age, n.length AS Length  ORDER BY n.pagerank,n.views DESC LIMIT 25
```

Figure 13: Neo4j PageRank results queries

Conclusion and Future work:

Analyzing Youtube data presents two challenges. The dataset is very large and continually growing at a high rate. To successfully analyze the data requires tools that can accommodate the size and its changing nature. We have analyzed the data using two popular data analysis tools, MySQL and Neo4j. Each tool is able to quickly calculate general statistics on the data as we demonstrated by evaluating top K statistics for four measures, popularity, rating, categorical, and video length range. The graphical database Neo4j offers a benefit over the relational database MySQL however when evaluating the degree relationship of the elements. To evaluate the in-degree and out-degree in MySQL would require at least an $O(n^2)$ search as we look for matches to each video in each available record, a task that is intractable for a dataset of billions of elements. Neo4j however can create relationships between nodes as they are loaded into the database. Once these relationships are created, they are easily searched in O(n) time. This is s significant improvement over the MySQL option.

Both MySQL and Neo4j are useful tools natively although each is enhanced with the addition of support tools. The use of Connector CPP for MySQL offers the ability to incorporate C++ code that helps manage the large data sets and multiple tables and we relied on it to accomplish our statistical analysis. Neo4j is improved with the addition of the Algo library that provides assistance in creating queries for common data analysis.

Based on our evaluations, we find that MySQL performs adequately and is useful for many applications but that Neo4j offers better analysis tools with user friendly GUI browser to interact with database.

Contributions:

- Christian Burke
    - mySQL
    - Range Queries
- Marcus Blaisdell
    - mySQL
    - Top-K Queries
    - User recommendation algorithm
- Linh Nguyen
    - Neo4j
    - PageRank
    - Categorized Statistics
- Thang Nguyen
    - Neo4j
    - Degree Distribution

References

1. Aslam, Salman. “YouTube by the Number: Stats, Demographics & Fun Facts”. *Omnicore, Sept. 18, 2018.* www.omnicoreagency.com/youtube-statistics/

2. Blaisdell, Marcus. “CS415 Course Project”. *GitLab, December, 16, 2018.* gitlab.com/marcus.blaisdell/cs415-course-project

3. Cheng, Xu, Dale, C. Liu, J. “Dataset for ‘’Statistics and Social Network of YouTube Videos’’ ”. *SFU(Simon Fraser University), 2008.*

4. Neo4j Operations Manual. *Neo4*. neo4j.com/docs/graph-algorithms/current/algorithms/page-rank/

**********END OF OFFICIAL DOCUMENT**********