2017

A Robust Framework for Mining YouTube Data

Zifeng Tian
tian2@marshall.edu

A ROBUST FRAMEWORK FOR MINING YOUTUBE DATA

A thesis submitted to
the Graduate College of
Marshall University
In partial fulfillment of
the requirement for the degree of
Master of Science
in
Computer Science
by
Zifeng Tian
Approved by
Dr. Wook-Sung Yoo, Committee Chairperson
Dr. Haroon Malik
Dr. Jamil M Chaudri

Marshall University
December 2017

APPROVAL OF THESIS/DISSERTATION

We, the faculty supervising the work of Zifeng Tian, affirm that the thesis, *A Robust Framework for Mining YouTube Data*, meets the high academic standards for original scholarship and creative work established by the Master of Science in Computer Science and the Weisberg Division of Computer Science. This work also conforms to the editorial standards of our discipline and the Graduate College of Marshall University. With our signatures, we approve the manuscript for publication.

Dr. Wook-Sung Yoo, Dec. 07, '17
Professor and Chair,
Weisberg Division of Computer Science Committee Chairperson Date

Dr. Jamil M Chaudri
Professor,
Weisberg Division of Computer Science Committee Member Date 2017 Dec 07

Dr. Haroon Malik Dec 07, 2017
Assistant Professor,
Weisberg Division of Computer Science Committee Member Date

ACKNOWLEDGEMENTS

I would like to appreciate my advisor Dr. Haroon Malik of the Weisberg Division of Computer Science at Marshall University for his wonderful help in both my thesis research and writing. I could not even start my thesis research without his enlightenment.

I would also like to faithfully thank Dr. Yoo and Dr. Chaudri for being my thesis committee and for being source of inspiration in the classes they taught.

Furthermore, I would like to express my sincere gratitude to my parents and my girlfriend, Mengsha Liu, for providing me with continuous support and unfailing patience. My father and mother sponsored me in studying without any doubt or hesitation. My girlfriend even quit her job in case that I need her support. Their backing is the guarantee that I could devote all myself into the thesis work.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

YouTube is currently the most popular and successful video sharing website. As YouTube has broad and profound social impact, YouTube analytics has become a hot research area. The videos on YouTube have become a treasure of data. However, getting access to the immense and massive YouTube data is a challenge. Previous research, studies, and analysis so far, are only conducted on very small volumes of YouTube video data. To date, there exists no mechanism to systematically and continuously collect, process and store the rich set of YouTube data. This thesis presents a methodology to systematically and continuously mine and store the YouTube data. The methodology has two modules: *a video discovery* and *a video metadata collection*. YouTube provides an API to conduct search requests analogous to the search performed by a user on the YouTube website. However, the YouTube API's '*search*' operation was not designed to return large volumes of data and only provides limited search results (metadata) that can easily be handled by a human. The proposed *discovery* process makes the search process scalable, robust and swift by (a) initially taking a few carefully selected video IDs (seeds) as input from each of the video categories in YouTube (so to get a wider coverage), and (b) later, using each of them to find related videos over multiple generations. The thesis employed in-memory data management for the discovery process to suppress redundancy explosion and rapidly find new videos. Further, the batch-caching mechanism is introduced to ensure that the high velocity data generated by the discovery process do not result in memory explosion; thereby increasing the reliability of the methodology. The performance of the proposed methodology is gauged over the period of two months. Within two months, 16,000,000 videos were discovered and complete metadata of more than 42,000 videos was mined. The thesis also explores serveral new possible dimensions that can be possible extensions to the proposed framework. The two most promiment dimensions are (a) *channel discovery*: Every YouTube user that has ever made a comment contributes to a channel. A channel can hold hundreds of YouTube videos and related metadata. Discovering channels can speed up the video discovery up to 100-fold; and (b) *channel metadata collection*: Since the volume of videos in a channel is massive, therefore, a mechanism needs to be developed to use multiple machines running software agents that can collaborate and communicate with each other to collect metadata of billions of videos in a distributed fashion.

CHAPTER 1

INTRODUCTION

1.1 Prior to YouTube

The Internet has witnessed an explosion of networked video sharing in these years. Among them, YouTube is one of the most successful ones [1]. Since being established by Google Inc. in February 2005, YouTube has become the largest video sharing site on the Internet. In only three years after the launch of YouTube, i.e., in 2008, it is estimated that there are over 45,000,000 videos in the repository and that the collection is growing at an astounding rate of seven hours of video being uploaded every minute [2]. To date, YouTube still maintains the privilege of being the largest video sharing site on the Internet [3].

YouTube is not the earliest online video platform. Actually, online videos existed much longer before YouTube entered the scene [4]. Video content in standard Video on Demand (VoD) systems has been historically created and supplied by a limited number of media producers, such as licensed broadcasters and production companies. Under this circumstance, the video content popularity was somewhat controllable through professional marketing campaigns [5]. Many of the activities such as Uploading, managing, sharing and watching videos were challenging due to the absence of an easy-to-use cohesive and unified platform. Also, there was very little in the way of content reviews or ratings [1].

As the technique of Internet improved, Web 2.0 became a trend in WWW technology, and it marked the new generation of web-based communities such as social networking sites, wikis, and blogs, which aimed to facilitate creativity, collaboration, and sharing among users. Web 2.0 changed how users contribute to the Web. Unlike the so-called Web 1.0 sites, which host content from established providers, users are now able to post their own content and view

content posted by their peers [3]. YouTube, as one of the new generation of video sharing sites, has overcome the problems that the traditional video platforms have, which have been discussed above. The new generation sites are also known as user generated content (UGC), and YouTube is currently the world's largest UGC VoD system [1]. The arrival of UGC has remodeled the online video market. Nowadays, hundreds of millions of Internet users are self-publishing consumers. In addition, the scale, dynamics, and decentralization of the UGC videos make a traditional content popularity prediction unsuitable. UGC popularity is more ephemeral and has a much more unpredictable behavior as well [5].

1.2 YouTube as Social Networking Website

Meanwhile, as a social network website, YouTube is designed to be, and very much is, two-way communication [6]. The UGC VoD systems allow content suppliers to upload videos effortlessly, and to tag uploaded videos with keywords. Content-creators can upload their videos using the simple UGC devised VoD system; an identifying code is attached to the video. The videos are made available to the viewers in the following two ways: the links are either mailed to the potential viewers or embed in the blogs for the general public. The UGC VoD system also enables communities and groups by strengthening the social network existing in YouTube [4].

As a social networking website, YouTube also contains the characteristics that other social networking websites have. One of the most interesting characteristics is the small-world phenomenon, which indicates that people are linked to all others by short chains of acquaintances. On YouTube, the "related" videos could be treated as the relationship between people, and there are definite small-world characteristics among them [1].

1.3 **Technologies applied in YouTube**

YouTube uses the Sorenson Spark H.263 video codec with pixel dimensions of 320 by 240. Its video playback technology is based on Adobe Flash Player. This technology allows YouTube to display videos with quality comparable to more established video playback technologies (i.e., Windows Media Player, QuickTime and RealPlayer). YouTube officially accepts uploaded videos in .WMV, .AVI, .MOV and .MPG formats, which are converted into .FLV (Adobe Flash Video) format after uploading. Each YouTube video contains an HTML markup. The Markup aids to embed the video in a page. This feature can be disabled by the uploader. [1]. YouTube's .FLV videos are not streamed to the user, but are instead downloaded over a normal HTTP connection. They are also not rate controlled to the playback rate of the video, but are sent at the maximum rate that the server and user can accomplish, and there is no user interactivity from the server's point of view. In order to fast forward, the user must wait for that part of the video to download, and pausing the playback does not pause the download [4].

YouTube allocated each video a unique 11-digit ID. The id is composed of a-z, 0-9, A-Z, and _. Each video comprises a series of metadata. The metadata includes a video ID, publish date, uploader, duration, category, user rating, number of views, ratings, comments, and a list of "related videos" which is recommended through a specific adsorption algorithm [1]. The adsorption algorithm is the core of the video suggestion system of YouTube. This algorithm is able to create a personalized page of video recommendations that not only shows the latest and most popular videos, but also provides users with a recommendation video list that depends on their own viewing habits. By using the adsorption algorithm, the expected efficacy of suggestions in YouTube has been improved. Since YouTube has a large number of users who view multiple videos, the statistics of video co-view number become very meaningful. These

data give, for any pair of videos, the number of people that viewed both videos. Based on these data, the developers of YouTube created a video-video co-view graph, and furthermore developed the adsorption algorithm for the recommendation system [2].

The basic interface for the users to interact with YouTube videos is “channel.” Whenever a user uploads a video, rates a video, or publishes a comment, these kinds of activities are assigned to a specific channel, which is linked either to the user’s Google account, or a Google+ page. If a channel is connected to a user’s Google account, both this channel and the Google account are uniquely managed by the Google account owner, i.e., this channel represents the Google account user. However, if the channel is connected to a Google+ page, the relationship between channels and users becomes somewhat more complicated: although one channel can still be connected with one single Google+ page, the relationship between Google+ page and Google account is “many-to-many.” In this way, multiple users can get access to one channel through the shared Google+ page; in contrast, one user can access multiple channels through the Google+ pages that he or she manages [YouTube Help → YouTube and Google+ → Manage multiple channels]. Similarly, channels also have distinct IDs, but with a different length from those of the videos, which is 24 digits. Each channel contains a series of metadata as well as a description of the channel, publish date, the number of videos that the channel published, etc.

CHAPTER 2

RELATED WORK

2.1 Review of previous work

In past, few researchers have used YouTube data (both the videos and associated metadata) to conduct various analysis/study. Many of them collected the data themselves, and a few used the data dumps available online. However, all the studies/analysis done so far are either conducted on very little volume of YouTube data or on cherry picking of the YouTube metadata attributes, usually the ones that are easy to harvest from YouTube. To date, there is no mechanism to systematically and on continuous basis collect, process and store the rich set of YouTube data. This chapter summarizes how researchers in the past have exploited YouTube data.

2.2 Summarized researches exploiting YouTube data

2.2.1 Abhari and Soraya

Abhari and Soraya [7] analyzed the YouTube video statistics to make YouTube traffic understandable. They crawled the YouTube site for only five months to get the required video metadata. Based on the collected data they implemented a workload simulation. For the purpose, they used multiple APIs. Abhari and Soraya periodically retrieved the data for the top 100 most viewed videos in a day and later in a week. Later, they expanded their video search by taking into account the relatively popular video for each of the 100 most viewed videos, i.e., they collected more video information which is "related" to the videos that have been crawled. The videos that are crawled by these two phases are separately treated as "popular videos" and "regular videos." The collected statistics include *video duration, file size, average rating, rating*

count, and view count. After crawling for around five months in total, the information of about 4,300 popular videos and 43,544 regular videos was successfully gathered.

2.2.2 Yoganarasimhan

Yoganarasimhan [8] studied how social network structure impacts the content propagation. He focused on the influence of the size and structure of the local network around a "node" on the total diffusion of products that are linked to it. He wrote a custom script using 'Perl' (a scripting language) for collecting the data from YouTube. The Perl script is an HTML parser that extracts the video related data. Meanwhile, Yoganarasimhan used another set of Perl scripts to gather the information of the uploader, i.e., the entity/person who uploaded the video to YouTube. The data belonging to the following attributes were collected: *views, number of ratings, average rating, number of comments, number of favorites, and number of honors*. Yoganarasimhan randomly picked 1,939 videos to monitor and processed the data collection every 24 hours for 38 days. In total, 85% of the videos had data for 31 days or more.

2.2.3 Gill *et al*.

Gill *et al*. [3] presented a YouTube traffic characterization study. The data used for analysis were collected by the YouTube Data API, which is provided by YouTube for the developers. Given a video ID, the information of the respective video was collected, which contains: *file size, video duration, bitrate, video age, a rating of video, and video category*. Gill also focused on the top 100 videos on YouTube every day. The monitoring of the videos started from Jan 14, 2007, and ended at April 8, 2007, i.e., a very short period.

2.2.4 Santos *et al*.

Santos *et al*. [9] collected a representative sample data of YouTube. They analyzed the sample data's structural properties and social relationships among users, videos and between

users and videos. To retrieve the data from YouTube, they custom built a crawler and extractor. By using the crawler and extractor, they indexed more than 11,000,000 videos and 620,000 users. Nevertheless, they were only able to collect the metadata of 300,000 videos and 12,000 users. Their extracted data set includes the following attributes: *number of videos watched by a user, number of subscribers, number of channel views, number of video views, number of comments, and number of times favorited*.

2.2.5 Cheng *et al*.

Cheng *et al*. [1] presented a systematic and in-depth measurement study on the statistics of YouTube videos. The method used by them for crawling YouTube was a combination of a) YouTube API and b) a scraper to scrape the information from YouTube video webpages. They also expanded the video list by accessing the related videos of the source videos. The video statistics contains *video category, video age, video duration, file size and bitrate, and a number of views*. After four months of crawling, about 3.2 million distinct videos' information was collected. Cheng et al. also used the same method to collect the metadata of YouTube videos in their measurement study on the characteristics of YouTube videos [4].

2.2.6 Cha et al.

Cha *et al.* [5] provided a study of YouTube in terms of a UGC system based on large volumes of data collected. The meta-information of YouTube videos was crawled by visiting the indexed pages that link all videos belonging to a category. Cha et al. limited the data collection to two of the categories, i.e., 'Entertainment' and 'Science & Technology.' The metadata that is recorded includes: (1) fixed information such as *uploader, upload time, and length*; (2) time-varying information such as *views, ratings, stars, and links*. About 1.9 million YouTube videos were traced and crawled.

2.2.7 Chatzopoulou *et al*.

Chatzopoulou *et al*. [10] conducted an in-depth study of the fundamental properties of video popularity in YouTube. They collected the metadata of 37 million videos by using YouTube Data APIs. Their crawling system contains a data server and several crawlers that communicate with YouTube and send the collected data back to the data server. They crawled YouTube for more than four months and generated a huge dataset that contained twenty tables regarding information about various perspectives. The collected metadata attributes comprise of *view count, number of comments, number of favorites, ratings, average ratings, etc*.

2.2.8 Figueiredo *et al*.

Figueiredo *et al.* [11] presented a study of characterizing the growth patterns of video popularity on YouTube. They used Google charts API to collect the YouTube data. In total, three types of dataset were created: Top, YouTomb, and Random topics. Top set contains the videos that are listed in various "top lists" (i.e., most viewed and most commented videos); YouTomb is a project commenced by MIT (Massachusetts Institute of Technology) that monitors the videos that had been removed from YouTube because of copyright violation and the researchers use it as the data source; Random topics set is generated by using YouTube search API and setting the randomly selected topics as the API input. The number of collected videos are: 27,212 in the Top set, 120,862 in the YouTomb set, and 24,484 in the Random topics set.

2.2.9 Siersdorfer *et al.*

Siersdorfer *et al.* [12] presented an in-depth study of commenting and comment rating behavior. The study was performed on a sample of more than 6 million comments on 67,000 YouTube videos. As mentioned in their previous work [13], they used YouTube API for retrieving the YouTube video information. The collected dataset had a final size of 67,290 videos

and about 6.1 million comments. They also collected metadata of the YouTube videos such as *title, tags, category, description, upload date*, as well as statistics such as *number of comments, views*, and *star rating for video*.

Author	Date	Method	Data Volume	Type	Duration	Purpose
Abhari and Soraya	2009	API	More than 47,000 videos	Video Metadata	5 months	Analyze YouTube traffic using video statistics
Yoganarasimhan	2011	Custom script	1939 videos	Video Metadata	38 days	Study the influence of the size and structure of the local network around a node on the total diffusion of products that are seeded by it.
Gill *et al.*	2007	API	Top 100 videos everyday	Video Metadata	3 months	Get the video file's detail information for their YouTube traffic characterization study
Santos *et al.*	2007	Custom crawler	300,000 videos	Video Metadata	N/A	Analyze YouTube video structural properties and social relationships among users
Cheng *et al.*	2007, 2008	API and scraper for webpage info	3.2 million videos	Video Metadata	5 months	Study on the statistics and characteristics of YouTube videos
Cha *et al.*	2007	Custom crawler	1.9 million videos	Video Metadata	N/A	Study of YouTube in terms of a UGC system
Chatzopoulou *et al.*	2010	API	37 million videos	Video Metadata	4 months	Study of the fundamental properties of video popularity in YouTube
Figueiredo *et al.*	2011	API	More than 170,000 videos	Video Metadata	1 day	Characterize the growth patterns of video popularity on YouTube
Siersdorfer *et al.*	2009, 2010	API	67,290 videos and 6 million comments	Video Metadata and comments	N/A	Analyze commenting and comment rating behavior

Table 1: Summarized researches exploiting YouTube data

CHAPTER 3

METHODOLOGY

3.1 **Framework**

To overcome the shortcomings of previous researchers and to fill in the gaps, the thesis proposes a methodology as a complete framework, to mine YouTube videos and related meta-data. The proposed framework consists of two parts.

- The video discovery, and
- The video metadata collection.

Figure 1 and Figure 2 illustrate, at a high-level, what constitutes our framework and the interaction between two parts of the framework.

At an abstract level, Figure 1 shows the underlying methodology for mining the YouTube data. First, video IDs are discovered and collected from YouTube. Later, IDs are stored in the database through MySQL JDBC, a tool for connecting Java environment and MySQL database. For every video ID that is collected, we retrieve the metadata information from it, parse it, and extract the information that can be directly stored in the database.

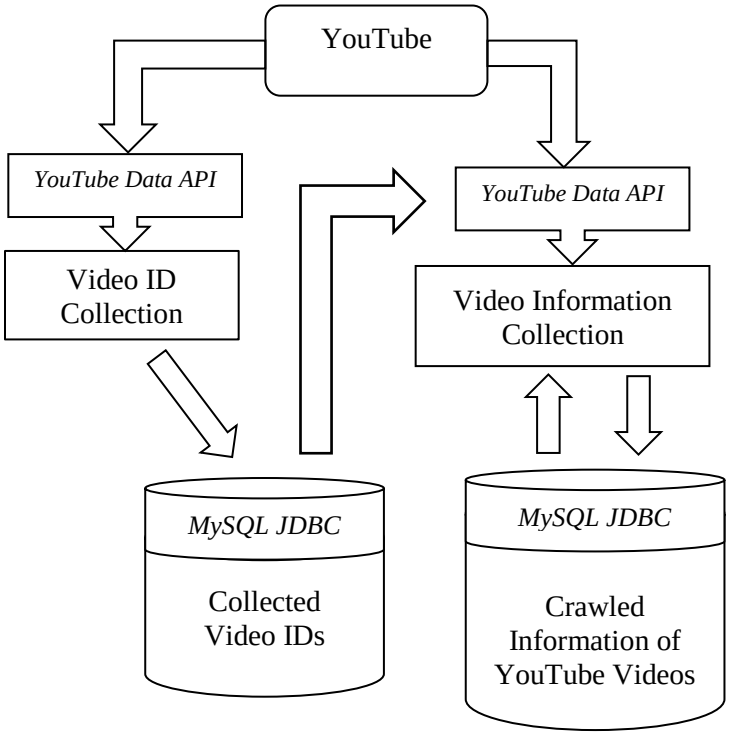

Figure 1: Illustration of the dataflow and interaction

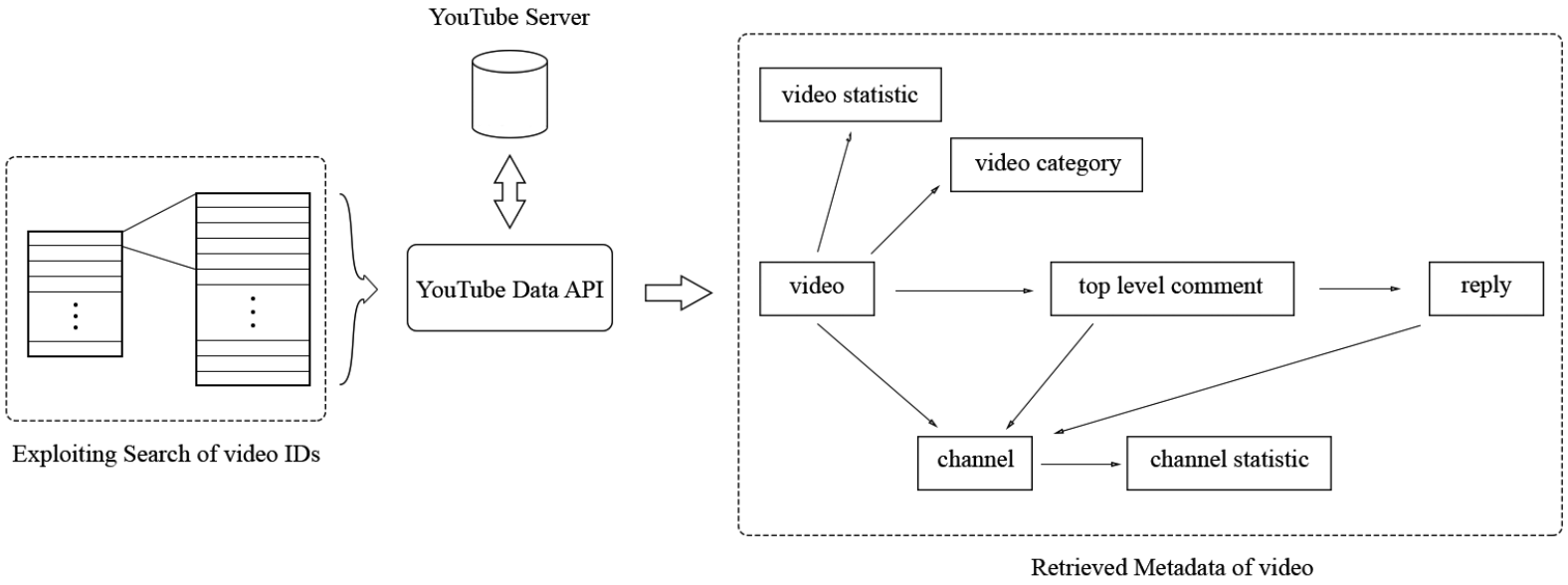

Figure 2: Illustration of the framework

Figure 2 illustrates how the framework is structured. We first use a set of known video Ids as seeds. We grow each seed by discovering related videos with it. Later, we retrieve the metadata of the videos and store the metadata in different relational tables in our database. In this chapter, we first introduce the techniques and tools used in our framework. Then we describe in detail each part of our mining framework.

3.2 **Tools**

This section provides details on the tools and techniques used to construct the YouTube mining framework. Besides the Java development kit and environment, the two most important tool/techniques are:

3.2.1 YouTube Data API

Google Inc. provides an official API, YouTube Data API, for the developers who are interested in analyzing the data of YouTube [14]. By using this API, developers are able to make HTTP requests directly from within their applications as if they are implemented on the web. In order to use this API, a Google Account is required to generate an API key and obtain an authorization credential, so that the API requests can be submitted successfully. According to the YouTube Data API documentation, a resource is an individual data entity with a unique identifier. In this thesis, the resources that have been used include channel, search result, video, video category, comment thread. Some operations, including a list (GET), insert (POST), update (PUT), delete (DELETE), are supported to help perform functions.

Google allocates each user some daily quota usage to guarantee that the developers do not overuse the service, which would lower the service quality for other users. Calculation of quota usage is based on assigning a cost to each request, though the costs differ from each other.

No.	Name of Video	Video ID
1	Sucker for Pain - Lil Wayne, Wiz Khalifa & Imagine Dragons w/ Logic & Ty Dolla $ign ft X Ambassadors	-59jGD4WrmE
2	A Great Big World, Christina Aguilera - Say Something	-2U0Ivkn2Ds
3	Meg & Dia - Monster (DotEXE Dubstep Remix)	-0oZNWif_jk
4	BANDA MS - NO ME PIDAS PERDÓN (VIDEO OFICIAL)	-7w9tdzndjc
5	I GOT KICKED OUT?!	-6dNin-p1Kg

Table 2: Sample of YouTube video IDs and titles

For example, a *read* operation that only retrieves the ID of a resource costs about 1 unit of quota; a video upload may cost about 1,600 units of quota. Presently, a newly registered Google account has a daily quota of 1,000,000 units. Old accounts that were registered before April 20th, 2016 have 50 million units per day.

3.2.2 MySQL Connector/J

MySQL Connector/J is a JDBC Type-4 driver (Database-Protocol driver) that implements the JDBC API, which is provided by MySQL. A Type-4 driver is a pure Java implementation of the MySQL protocol; thus, it doesn't rely on the MySQL client libraries [15].

MySQL is a widely used open-source relational database management system (RDBMS) developed by Oracle. Despite the data collected for conducting experiments in the thesis is of large volume, still, a relational database model was sought over other available models. The reason for using a relational database is that the other models, though strong in processing big data, lack in observing the robust relation between features. In addition, the main development environment of the crawler project is in Java, thus facilitates the use of MySQL Connector/J as the bridge between Java environment and MySQL database.

All types of metadata related to a YouTube video is tied to a unique identifier called video ID. A sample of several video IDs is listed in Table 2. YouTube does not publish the list of

the video IDs nor is there any collection available publicly. Every day thousands of videos are either added, modified or removed. The key challenge for the collection of a large volume of YouTube metadata on a continuous basis is to discover a large volume of video IDs. The IDs of the videos can be used by our framework as an input.

3.2.3 Exploiting Search Feature

The YouTube Data API provides a resource called '*Search'* in order to retrieve a search request analogue to the search performed by a user on the YouTube website. As shown in Figure 3, users can not only specify the keywords, but also some other attributes such as *channel ID* (collect videos uploaded by a specific channel), *video category* (collect the videos only from a specific category), *publish after/before* and order the returned result by *data/view count/rating*. By using this resource, generating a few hundred unique video IDs is convenient and fast. However, the YouTube API's '*search'* operation was not designed to return large volumes of data. It was designed to simulate the search activity in YouTube website on behalf of the users and provide limited search results (metadata) that can easily be handled by a human. Thus, the 'Search' resource provided by YouTube Data API is not scalable and cannot fill our requirement of providing access to large volumes of video IDs.

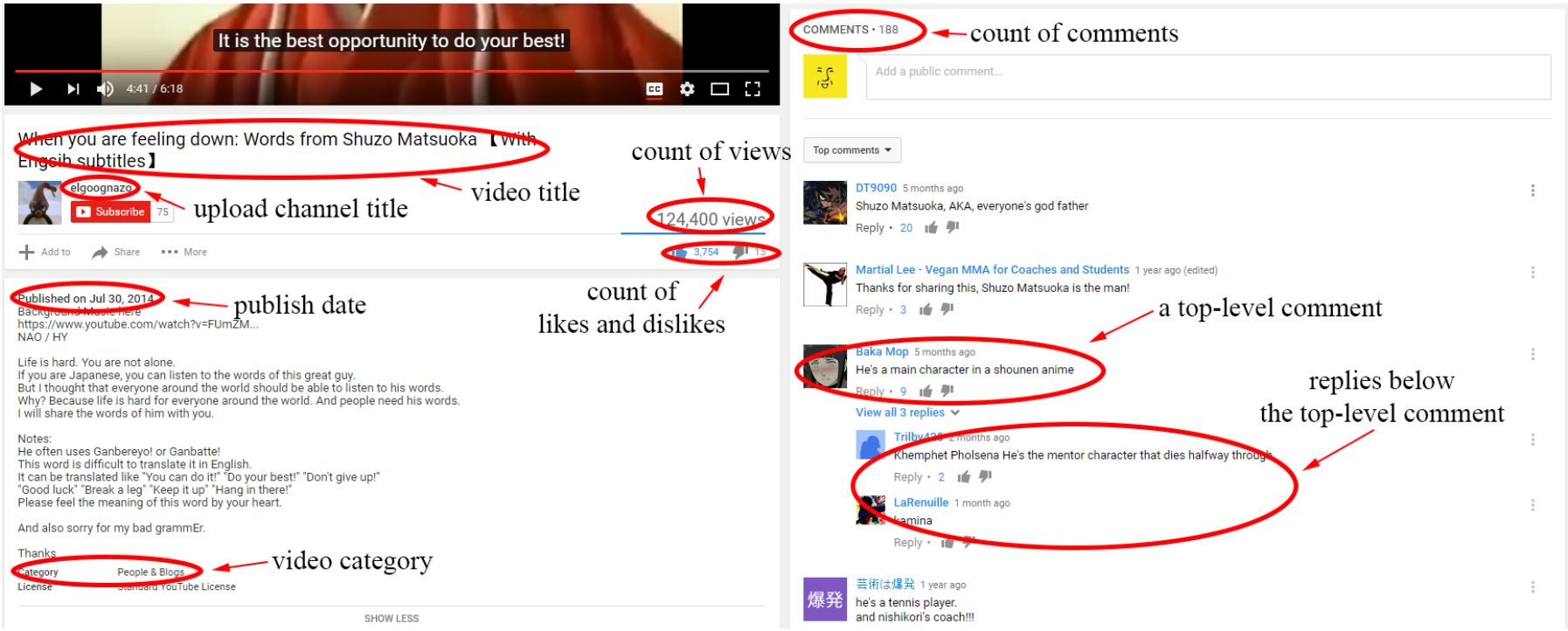

Figure 3: A screenshot of one YouTube video

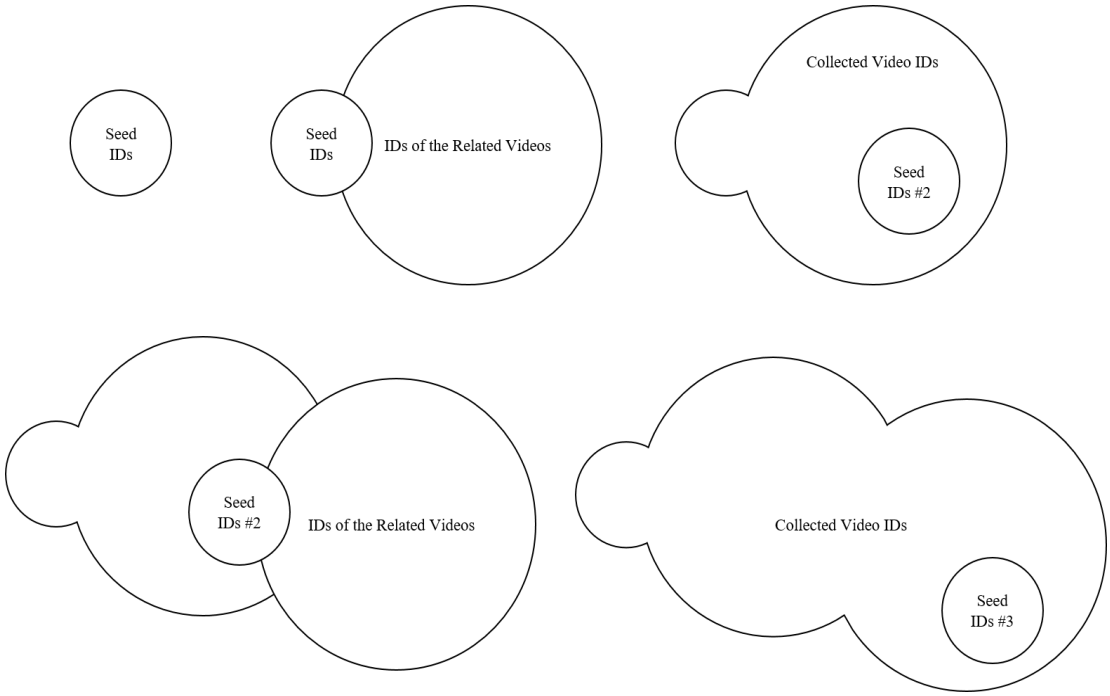

Figure 4: Seed cultivation for the video discovery process

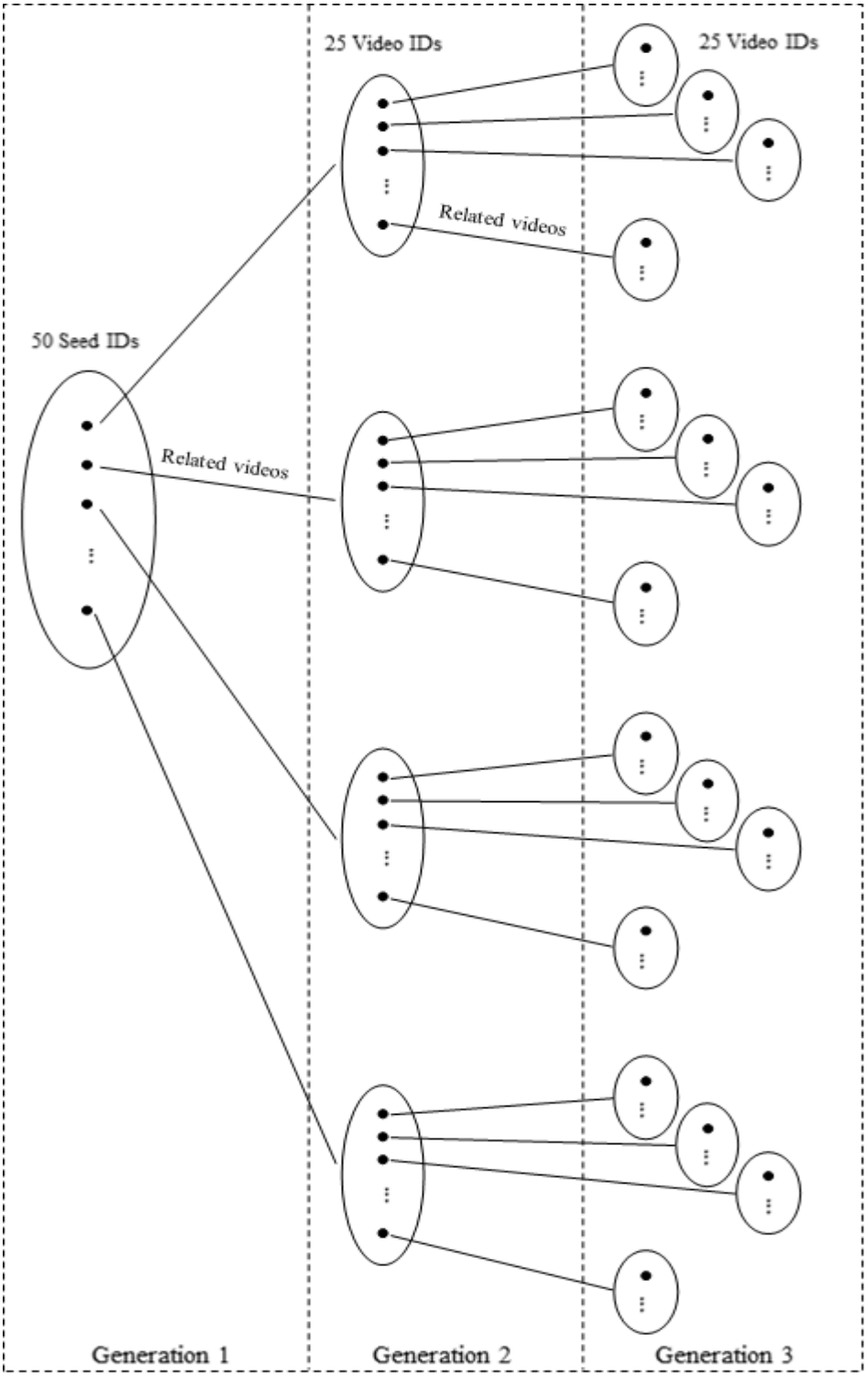

Figure 5: Seed growth over the two Generations

Nevertheless, *Search* resource provides another method. In this method, one can specify a video id. The method returns up to a few hundred videos that are related to the supplied video. By recursively using this method, a small size of dataset of video IDs can be used as 'seeds.' These seeds can potentially be grown exponentially. Figure 4 depicts the high-level concept of seed cultivation for the video discovery process. Based on the intuition, we designed the '*Discovery Module.*' To test the hypothesis, we did a pilot test.

- We randomly collected 50 videos as seeds and stored them as comma separated (CSV) file.
- Each seed was grown to cultivate a set of 25 related videos, as shown in Figure 5. Thus, in the first generation, 50 new sets containing 25 new seeds each are cultivated. Thereby, increasing the video id count to one thousand and fifty videos:

$$50\ (\textit{seed videos}) \times 25\ (\textit{cultivated set for each seed}) = 1{,}250$$

- Iteratively, we cultivated each set of seeds in the second generation to produce more seeds. Thus, in the second generation, we were able to grow the set of videos IDs to thirty-one thousand and two hundred and fifty videos (31,250), i.e., 99% in volume with respect to the starting set containing 50 seeds. In the third generation, the set contained 781,250 video IDs.

We verified the quality of the collected data set using a custom written script and found several problems. The first and the most important issue identified in our pilot study was that the final set of video IDs (i.e., in the third generation), contained lots of duplicate video IDs. We investigated the issue further to identify the reason for such redundant video ID despite the fact that '*search'* method provided by the YouTube API returns the set of distinct video id for a given seed. We found that one video can be related to more than one video thus can be returned as a

redundant video while cultivating seeds over multiple generations. We have calculated the basic statistics of the duplicate video IDs in the first five generations in a single running test, and the result is shown in Figure 6 and Figure 7. We can clearly see that as the generation increases, the percentage of duplicate video IDs grows larger and larger. The amount of duplicate video IDs increases faster than that of the distinct ones.

The redundant videos act as a noise in our data set and must be removed. The heuristics used by the 'Search' method of the YouTube API, to find 'relevant' videos are not known, i.e., related videos are recommended based on the number of customers who viewed the two videos together, or relation is based on the recent videos in the similar category, or based on the publisher.

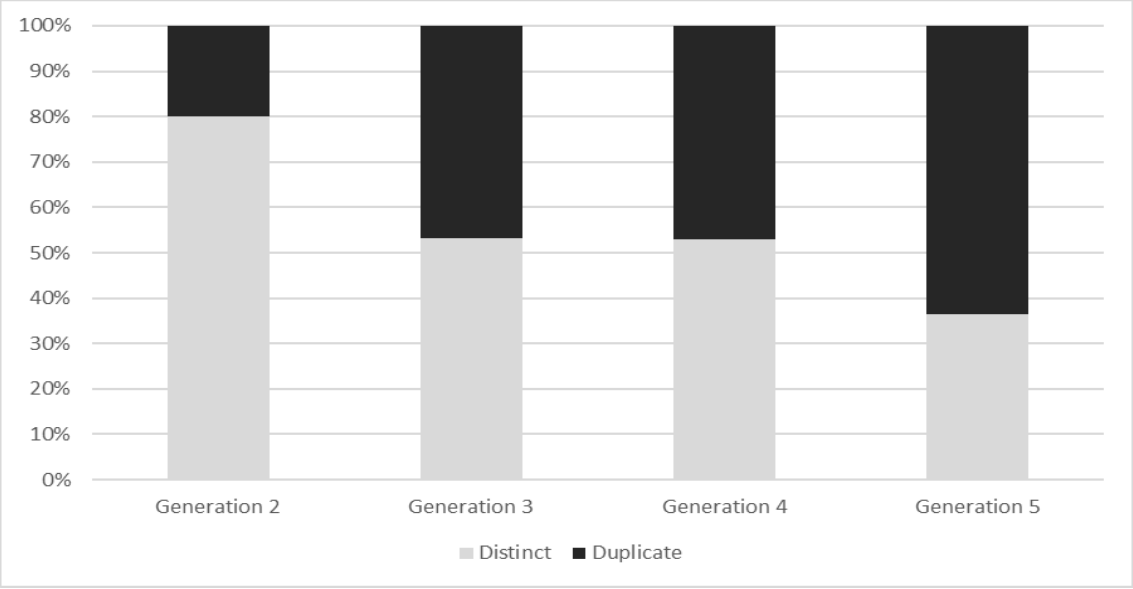

Figure 6: Duplicate IDs percentage in each generation

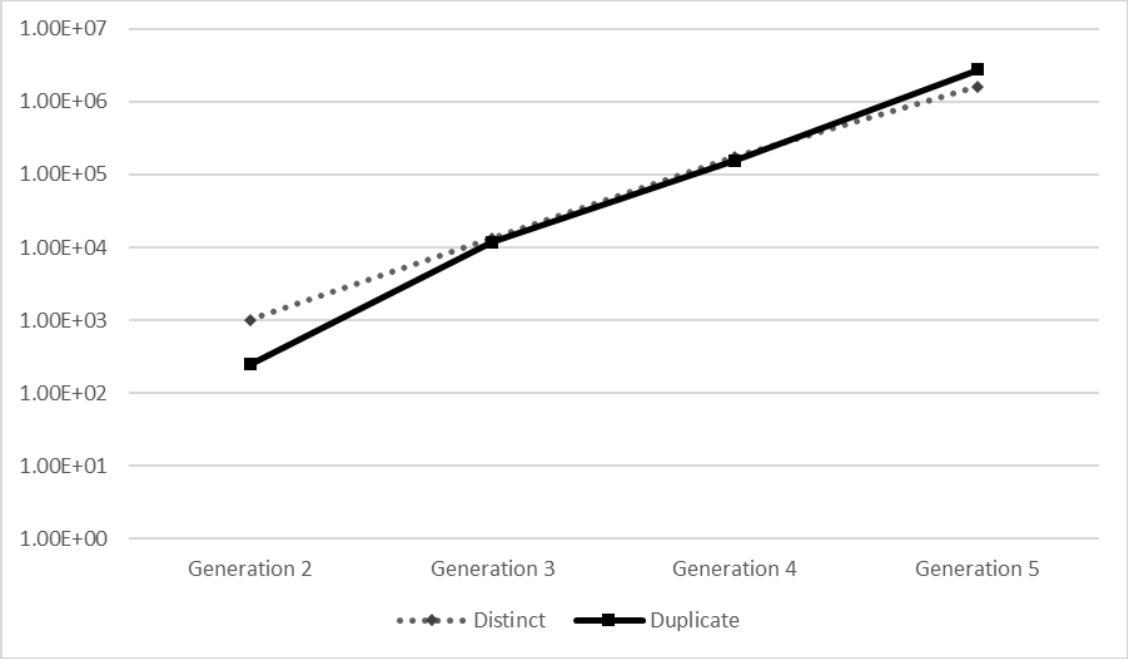

Figure 7: Number of Duplicate and Distinct IDs in Each Generation

3.2.4 Removing Duplicate Video IDs

In order to accomplish the video discovery process, a mechanism is required to avoid duplicates without sacrificing the performance of the video discovery process. One possible solution is using a relational database for data storage. As restricted by the feature of the relational database, if an entity is to be stored in the database, it must have a "key" value (primary key) so that it can be distinguished from the other entities. When storing an entity, the relational database system will verify whether its key already exists or not. If it does not, the new entity will be kept in the database; otherwise, the entity will be identified as duplicated and can be discarded if required.

For the thesis, the video IDs can be used as the key so that they can be distinctly stored. However, the discovery process is generating a larger and larger dataset of IDs in an extremely high rate (nearly exponentially), and more and more video IDs become "duplicate," i.e., have already been identified as related videos of other videos before. Thus, the traffic burden of

sending queries and data to the database turns out to be very high, but many of them are useless duplicate values. In addition, sending queries to a database requires accessing hard drive, meaning that it costs much longer time than methods that process in the main memory, let alone the internet speed limit when the database is set remotely.

Another possible solution is to remove the duplicates in the main memory. As discussed above, removing duplicates in main memory is much faster than the methods that access the hard drive. Since the development environment is in Java, the data collections that are implemented in Java standard libraries become the first choices, such as HashSet, ArrayList, TreeSet, etc. In the thesis, HashSet is the method chosen for solving the duplicate problem.

HashSet is a collection that uses a hash table for storage. In Java, it extends the AbstractSet class and implements the Set interface [16]. An element in a HashSet is associated with a value called *hash code* which is mapped to the element value. The calculation is done automatically when the elements are inserted into the HashSet. Since AbstractSet only allows unique value stored in it, as an inheritance of AbstractSet, HashSet also has the same feature. Before storing an element, HashSet will first check whether the element is already stored or not: if not, simply puts the element into the set; otherwise makes no change. Compared with other kinds of collection (for example, ArrayList or TreeSet), HashSet is the fastest one for checking whether an element is already in the collection. Rather than calling the compare method of the datatype of the stored information, HashSet uses hash code as the unique identifier for each element in it. In case that two different elements have the same hash code, a normal comparison method is also integrated. If the hash code generation algorithm properly distributes the elements among the "buckets," i.e., the hash codes of all the elements in HashSet are all distinct, then the basic operations (*add, remove, contains,* and *size*) have constant time performance $O(1)$ [16]. For

comparison, ArrayList performs *O*(n) in the operation of "*contains*" which is required for avoiding duplicates, and TreeSet performs *O*(log n) in "*contains.*" Thus, HashSet performs persistently well no matter how large an increase in dataset size. Although ArrayList and TreeSet have some additional functions that HashSet does not have, these functions are useless in removing duplicate IDs, which is our main purpose. In practice, HashSet is so fast in removing duplicates that the time used in the phase can be nearly ignored.

In fact, database indexing has the similar mechanism of HashSet; however, as discussed earlier, passing the original data that contains a large number of duplicate elements is not efficient. Nevertheless, the size of the dataset is so large that if all the workload is assigned to the main memory of the local machine, it will break easily because of hitting the memory limit. Assume that we are using 50 video IDs as initial seeds and find 25 related videos for each of them, then in the 5^{th} generation (expanded four times), we will get more than 200MB data in the memory. Then the 6^{th} generation will break halfway since the memory limit is reached. Thus, the HashSet in memory cannot keep growing unlimited but has to get released regularly. The size of seed videos used for searching related videos is also capped with 10,000 for each "generation." When gathered a set of related videos, we insert them into the database, choose 10,000 from them as new seed videos, and then release the set. This results in that part of the newly collected videos already exist in the database, but not in the current HashSet, which makes the database share part of the duplicate removing work. In this way, the speed of processing decreases, but meanwhile the free space in memory increases, which can be treated as a space-speed tradeoff method.

Another problem occurs when using the API. The video ID discovery requires YouTube Data API sending HTTP requests frequently to the Google server, which may occasionally get

“503 internal server error.” To avoid this error, an exponential backoff algorithm is applied in the error catching block. This algorithm ensures that the pending requests get delayed and sent; if the same error still occurs, the period of delay doubles and try another time. This process is recursively executed until the requests are successfully sent or the process exceeds a re-try time limit. This problem not only happens in the discovery of video IDs but also in the collection of video metadata. The same solution is also used in it to get rid of the internal server error.

3.3 **Video Metadata Collection**

3.3.1 Metadata Structure

After gathering a large number of video IDs and storing them in the database, it was possible to use them for retrieving detailed information of each single video ID with the help of YouTube Data API. The resources of API used in this procedure include video, video category, channel, and comment thread.

Since each resource provides a big size JSON-like structured data shown in Figure 8 and not all of them are important or often cared, only part of each result is specified in API request. In addition, some kinds of metadata stay constant, such as ID, video length, video upload channel, comment published time, etc.; other metadata varies over time, such as video view count, number of channel's comment count, and channel's subscriber count.

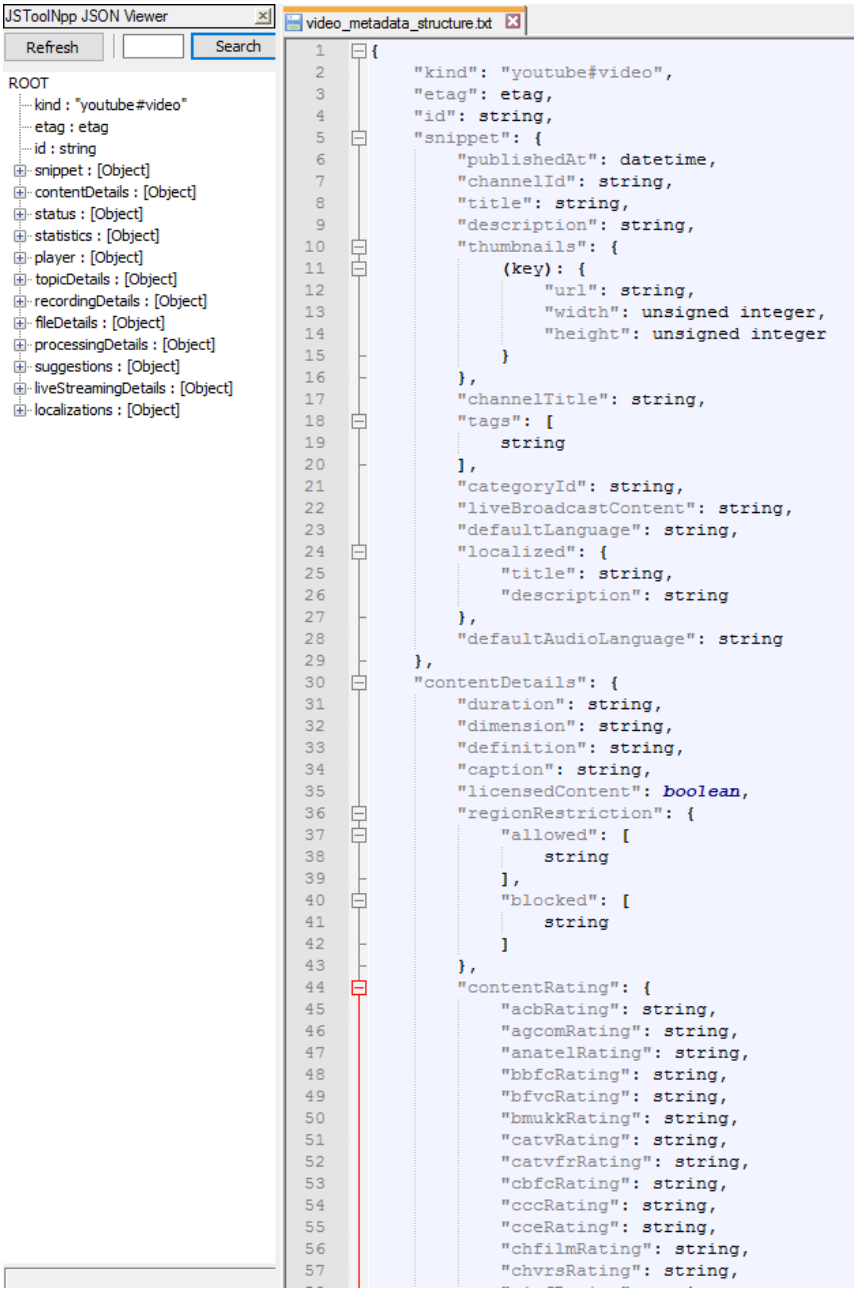

Figure 8: Structure of video metadata which is collected by YouTube Data API

In this way, the collected metadata can be separated into two types:

1. Invariant Data and
2. Dynamic Data

Below we provide what constitutes the two types of data.

3.3.1.1 The Invariant Data

The **Invariant** data include:

a) Video:

- *Video ID*: an 11-digit unique identifier of the video. The digits are composed of 0-9, a-z, A-Z, - and _.
- *Category ID*: a 2-digit unique identifier of the video category. The digits only contain 0-9.
- *Video publish time*: the video publish time. If a video is uploaded as public, this value is the exact upload time of the video; however, if a video is uploaded as private and then made published, this value will specify the time that the video gets shifted to publish. The format of this value is in ISO 8601 (YYYY-MM-DDThh:mm:ss.sZ) standard.
- *Duration*: the length of the video.
- *Video title*: the shown name of the video.
- *Video description*: the video's description with a maximum length of 5000 bytes. All UTF-8 characters are valid, except < and >.

b) Video Category:

- *Category ID*: same as the attribute of Video. A 2-digit unique identifier of the video category. The digits only contain 0-9.
- *Category title*: the shown name of the video category.

c) Channel:

- *Channel ID*: a 24-digit unique identifier of the channel. The digits are composed of 0-9, a-z, A-Z, - and _.
- *Channel publish date*: the time that the channel was created. The format of this value is in ISO 8601 (YYYY-MM-DDThh:mm:ss.sZ) standard.
- *Channel title*: the shown name of the channel.
- *Channel description*: the channel's description with a maximum length of 1000 bytes.

3.3.1.2 The Dynamic Data

The dynamic data include:

a) Comment Thread:

Comments are classified according to their position. One is called top level comment, which is directly shown in the video comment block; the other is called reply, which is commenting and replying to the top level comments or other replies under top level comments.

i. Top level comment:

- *Comment ID*: a unique identifier of the comment.
- *Video ID*: the ID of the video which the comment belongs to.
- *Channel ID*: the ID of the channel who creates the comment. In YouTube, a channel normally represents a user (except Google+ users), so when a user makes a comment, the user's channel ID means the ID of the user.
- *Comment like count*: how many people like this comment.
- *Comment published time*: the time that the comment was originally created. The format of this value is in ISO 8601 (YYYY-MM-DDThh:mm:ss.sZ) standard.
- *Comment update at*: the time that the comment was last updated. The format of this value is in ISO 8601 (YYYY-MM-DDThh:mm:ss.sZ) standard.
- *Comment text*: the text content of the comment.
- *Total reply count*: the number of replies following the comment.

ii. Reply:

- *Comment ID*: a unique identifier of the comment.
- *Top level comment ID*: the ID of the "parent" comment of a reply. No matter if a reply is commenting on a top level comment or other replies, this reply should be associated to a top level comment, which is the "parent" comment of it.
- *Channel ID*: same as the one of top level comments, the ID of the channel who creates the comment.
- *Comment like count*: how many people like this comment.
- *Comment published time*: the time that the comment was originally created. The format of this value is in ISO 8601 (YYYY-MM-DDThh:mm:ss.sZ) standard.
- *Comment update at*: the time that the comment was last updated. The format of this value is in ISO 8601 (YYYY-MM-DDThh:mm:ss.sZ) standard.
- *Comment text*: the text content of the comment.

b) Video Statistics:

- *Video ID*: the unique identifier of the video.
- *Video timestamp*: this parameter is not collected through the API. Actually, this is the local time (U.S. Eastern) when collecting the statistics of the video. Since the statistics of videos are always changing, it is necessary to keep tracking the timestamp of collected statistics information. In addition, tracking the timestamp allows a video owning multiple statistics information in different time, which can help with the analysis of statistics trend.
- *Video comments count*: the number of comments of the video.
- *Video dislike count*: the number of "dislikes" of the video.

- *Video like count*: the number of "likes" of the video.
- *Video view count*: the number of how many times the video has been viewed.

c) Channel Statistics:

- *Channel ID*: the unique identifier of the channel.
- *Channel timestamp*: the purpose of creating this attribute is similar to that of creating the video timestamp. This is the local time (U.S. Eastern) when collecting the statistics of the channel.
- *Channel comment count*: the number of comments made to the channel.
- *Channel subscriber count*: the number of subscribers to the channel.
- *Channel video count*: the number of videos that belong to the channel.
- *Channel view count*: the number of how many times the channel has been viewed.

3.3.2 Metadata collection process

The initialization of the whole process was sending queries to the database that has already stored the "discovered" video IDs and use them to gather more of the related video IDs. By sending a video ID to the API, it was possible to collect the various metadata of the video, the video's category, the video's publisher (channel), and the comments of the video. Then the publisher's channel ID and the commenters' channel IDs were sent to the API as the source of collecting the metadata of channels.

According to the API documentation [14], the YouTube Data API can accept fifty comma-separated IDs at one time when setting video or channel IDs as the parameter. Compared with inputting IDs one at a time, this method significantly increases the API response speed, meanwhile reduces the times of sending requests to the Google YouTube server. In our metadata process, we retrieve twenty video IDs that have been already stored in our database and combine

and format them into a comma-separated value so that the API can identify and have the Google YouTube server accept it. The reason for sending twenty IDs each time instead of sending fifty is that when a video is popular, it may contain many thousands of comments and replies which will also generate a big size of channel IDs; for retrieving these metadata we need to send requests to the Google YouTube server through the API; however, if sending requests to the Google YouTube server too frequently, it will return a server error to temporarily interrupt the requests. Nevertheless, if the number of video IDs we send to the YouTube server each time is small, the time efficiency will decrease, especially when we send only one video ID each time. For the balance of the time efficiency and stability, twenty video IDs each time is a proper size to send to the server (this variable may differ in different running environments though).

The returned results from YouTube server are in JSON-like structures, which is easy to read and modify, but hard to insert into the database. Thus, results were converted to single sub-layer JSON objects so that they can be stored in the database. Figure 9 shows an example of how video metadata format is converted.

Figure 9: Structure demonstration of received video metadata before and after conversion

3.4 **Database Storage**

Both the Video ID discovery procedure and the metadata collection procedure are using database storage for the purpose of keeping the information sustainable and relatively safe. Using database also makes it possible to distribute the workload into multiple machines so that the speed of information collection improves. In case of maintaining the close relationship between different resources of video metadata, the MySQL is used as the relational database management system for the project.

3.4.1 Database in Video ID discovery phase

In this phase, the database table structure is simple. Since YouTube video IDs are distinct, they are used as the primary keys in the table of video ID. The additional information that stores in this table are: (1) a number that automatically increments by 1 to keep tracking the sequence of the video ID collection in case the discovery procedure breaks and need recovery from the latest point; (2) a number that counts how many times the video ID has been used for the next procedure *metadata collection*, which can help guarantee that the IDs are averagely referred to. The (2) part will be discussed more in the next sub-session.

API calling in this phase is not very fast, but it is not possible to improve this on the client side. For example, for each 10,000 seed videos, it will cost 60-80 seconds for calling API to gain the related videos of the seed videos, and less than two seconds for the duplication removing in HashSet (local memory). Thus, the speed bottleneck is inserting the collected IDs into the database. Using batch insertion can significantly increase the speed of inserting, compared with making queries separately.

3.4.2 Database in metadata collection phase

In this phase, the collected JSON-like metadata are inserted into the database, of which tables' structures are the same as those mentioned in the section 3.4.1.

Unlike the first phase discussed above, during the "Database in metadata collection" phase, there exists a performance limit for API calls. Since one video may have thousands of comments or even more, the corresponding number of channels becomes large, which results in a slow speed in the process information retrieval of the channels. In addition, the large information size of channels challenges the insertion batch as well. In YouTube, the text content (comment text, video description, etc.) contains abundant types of characters, such as Arabic, Chinese, Japanese, and a lot of emoji. To support the characters, UTF-8 is used as the encoding method. However, in MySQL, what's called "UTF-8" is actually a subset of UTF-8, which does not support many characters that used to be supported by UTF-8. Luckily, a character set named utf8mb4 is created to cover the whole set of characters that are supported by UTF-8. This character set is very important in handling exceptions while using batch insertion because using batch insertion makes it impossible to track individually inserted queries and handle their exceptions. If an inserted entity contains characters that are not supported by the database, the information of that entity will be completely lost. Since many people like using special characters like emoji in their text, the information is unignorably lost if the character set is not set up properly.

CHAPTER 4

CONCLUSION AND FUTUREWORK

This chapter concludes the thesis. The concepts presented throughout this thesis are summarized. The limitations and possible directions for future work are also presented.

4.1 Summary

YouTube is the largest video sharing repository in the world. Millions of videos have been uploaded into the repository, and several hundreds of hours' videos are being uploaded every minute. YouTube not only promotes self-publishing contents, but also provides two-way communications so that users are connected to the whole world, and can interact with each other by commenting, rating videos, subscribing channels, and so on. As a matter of fact, YouTube has broad and profound social impact on many areas, and became the best choice if someone is aiming on reaching a wider audience. Therefore, YouTube analytics has become a hot research area.

Nevertheless, quality of analytics depends on merit, volume and granularity of the data in hand. However, getting access to the immense and massive YouTube data is still challenging. For the purpose of overcoming the challenge, the thesis proposes the framework in Chapter 3. This framework consists of two parts: video ID discovery, which aims to discover large volumes of YouTube video IDs; and video metadata collection, which uses the video IDs that are discovered in the previous part and the video ID discovery part, to retrieve the metadata of them.

In the video ID discovery part, the related videos of a given video (seed) are searched to generate a rapid-growing dataset. In terms of storage, thesis uses HashSet technique for the data collection and managing duplicate video IDs efficiently. The thesis also limits the size of seed IDs to control the memory usage so that memory is not bottlenecked. In addition, an exponential

backoff algorithm is applied to avoid the “503 internal server error” which occasionally occurs while sending requests to YouTube at high velocity.

In the video metadata collection part, the metadata is split into two groups: invariant data, and dynamic data. Invariant data is normally collected once, and dynamic data, which is changing all the time, is to be collected multiple times. All the metadata received through YouTube Data API is in JSON structure; the metadata is preprocessed and reformed to a group of simple “key-value” pairs. This preprocess ensures that the metadata can be easily inserted into our database.

Over the period of two months, using our methodology, the methodology discovered 16,000,000 videos and mined the complete metadata of more than 42,000 videos. Since this is an on-going work, in future, the expansion of the proposed framework and data collected is expected on continuous basis.

4.2 **Limitations**

This section lists the limitations of the proposed framework. The limitations are planned to be addressed as future work.

1. Since the propoed framework relies on the YouTube/Google proprietary API for mining the data, any change in their API will lead to modifying the code used by our framework, to parse the JSON Object.
2. If YouTube decided to remove any invariant/variant data contents, then it is a must to alter the database schema used by our proposed frame work accordingly.
3. Currently, the framework is running on a dedicated machine. If the machine fails, for some reasons, the framework will also collapse. In future, it is aimed to scale the

framework by running it on multiple machines to (a) increase the reliability of the framework and (b) speed up the harvesting of data from YouTube.

4.3 **Future work**

While executing the data collection process, several new possible extensions were discovered that can be made to extend the framework. This section lists the two promising future avenues.

4.3.1 Separate channel metadata collection

Channel metadata collection is an important but time-consuming process. Currently, metadata collection is implemented as a thread, of which the loop is: retrieve some video IDs from database → collect video metadata (also including the metadata of video category, statistics, and comments), meanwhile gather the channel IDs that related to these videos → collect channel metadata → retrieve more video IDs → …(loop). However, one video can be related to a large number of channels, especially when the video is very popular. Every YouTube user that has ever made a comment to a video will be counted as a channel in the channel metadata collection, and it can result in a big bang. For example, if one video has 1,000 comments from different YouTube users, then there will be 1,000 channels when collecting the metadata of channels. Thus, channel metadata collection would be the most time-consuming section in the thread loop.

On the other hand, there is duplicate work during channel metadata collection. One user can provide comments for multiple videos, which implies that one channel ID is related to multiple videos. If we collect channel metadata right after we have gathered the related channel IDs of a video, we may process the same channel multiple times. To avoid the meaningless duplicate work, the thesis used HashSet to store channel IDs

before proceding to the channel metadata collection process. Currently, in each loop, 20 videos are processed. After the related channel IDs of the 20 videos are gathered, these channel IDs are stored in the HashSet to retrieve channel metadata. Thus, inside each loop, the channel ID duplicates are removed. However, this method can only guarantee a small coverage, i.e., the duplicates in between different thread loops cannot be eliminated.

The combination of these two problems can significantly influence the performance of the proposed framework in terms of time efficiency, i.e., increased latency. The latency is not noticed until the underlying code gets executed for a long period. A possible solution could be separating the processes of channel metadata collection as a new "channel thread," rather than being merged with the video metadata collection. By separating the thread loops, we can focus on just collecting video metadata in a "video thread." In addition, instead of collecting channel metadata, we only store the channel ID in database during the video thread. Meanwhile, we use the separated channel thread to crawl the channel IDs and retrieve the corresponding channel metadata, which is similar to what we are doing in the video thread. The channel thread can be executed in parallel with the video thread. The workload can be also distributed to different machines.

4.3.2 Recursive ID collection of videos and channels

For the purpose of exploiting search videos as exhaustively as possible, a recursive ID collection process can be introduced. Using video IDs, it is possible to discover channel IDs by tracking the video uploader's and the commenters' channel IDs; and in contrast, using the channel IDs, we are able to discover video IDs by tracking the

videos in the channels' playlists, subscriptions, etc. This method can help dig deeper and more completely in discovering video IDs as well as channel IDs.

REFERENCES

[1] X. Cheng, C. Dale and J. Liu, "Statistics and Social Network of YouTube Videos," in *2008 16th Interntional Workshop on Quality of Service*, Enschede, 2008.

[2] S. Baluja, R. Seth, D. Sivakumar, Y. Jing, J. Yagnik, S. Kumar and M. Aly, "Video suggestion and discovery for youtube: taking random walks through the view graph," in *Proceedings of the 17th international conference on World Wide Web*, Beijing, 2008.

[3] P. Gill, M. Arlitt, Z. Li and A. Mahanti, "YouTube Traffic characterization: a view from the edge," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, San Diego, 2007.

[4] X. Cheng, C. Dale and J. Liu, "Understanding the Characteristics of Internet Short Video Sharing: YouTube as a Case Study," 2007.

[5] M. Cha, H. Kwak, P. Rodriguez, Y. Y. Ahn and S. Moon, "I Tube, You Tube, Everybody Tubes: Analyzing the World's Largest User Generated Content Video System," in *Proceedings of the 7th ACM-SIGCOMM conference on Internet measurement*, San Diego, 2007.

[6] M. Prensky, "Why You Tube matters. Why it is so important, why we should all be using it, and why blocking it blocks our kids' education," *Horizon,* vol. 18, no. 2, pp. 124-131, 2010.

[7] A. Abhari and M. Soraya, "Workload generation for YouTube," *Multimedia Tools and Applications,* vol. 46, no. 1, pp. 91-118, 2009.

[8] H. Yoganarasimhan, “Impact of social network structure on content propagation: A study using YouTube data,” *Quantitative Marketing and Economics,* vol. 10, no. 1, pp. 111-150, 2011.

[9] R. L. T. Santos, B. P. S. Rocha, C. G. Rezende and A. Loureiro, “Characterizing the YouTube video-sharing community,” 2007.

[10] G. Chatzopoulou, C. Sheng and M. Faloutsos, “A First Step Towards Understanding Popularity in YouTube,” in *INFOCOM IEEE Conference on Computer Communications Workshops,* San Diego, 2010.

[11] F. Figueiredo, F. Benevenuto and J. M. Almeida, “The tube over time: characterizing popularity growth of youtube videos,” in *WSDM ’11 Proceedings of the fourth ACM international conference on Web search and data mining*, Hong Kong, China, 2011.

[12] S. Siersdorfer, S. Chelaru, W. Nejdl and J. S. Pedro, “How useful are your comments?: analyzing and predicting youtube comments and commet ratings,” in *WWW ’10 Proceedings of the 19th international conference on World wide web*, Raleigh, North Carolina, USA, 2010.

[13] S. Siersdorfer, J. S. Pedro and M. Sanderson, “Automatic video tagging using content redundancy,” in *SIGIR ’09 Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval*, Boston, MA, USA, 2009.

[14] “YouTube Data API,” [Online]. Available: https://developers.google.com/youtube/v3.

[15] “MySQL Connector/J,” [Online]. Available: https://dev.mysql.com/doc/connector-j/5.1/en/connector-j-overview.html.

[16] “HashSet docs,” [Online]. Available: https://docs.oracle.com/javase/7/docs/api/java/util/HashSet.html.

APPENDIX A

IRB LETTER

MARSHALL UNIVERSITY
www.marshall.edu

Office of Research Integrity

April 20, 2017

Zifeng Tian
Weisburg Division of Computer Science
Marshall University

Dear Zifeng:

This letter is in response to the submitted thesis abstract entitled *"A Robust Framework for Mining YouTube Data."* After assessing the abstract, it has been deemed not to be human subject research and therefore exempt from oversight of the Marshall University Institutional Review Board (IRB). The Code of Federal Regulations (45CFR46) has set forth the criteria utilized in making this determination. Since the information in this study does not involve human subjects as defined in the above referenced instruction, it is not considered human subject research. If there are any changes to the abstract you provided then you would need to resubmit that information to the Office of Research Integrity for review and a determination.

I appreciate your willingness to submit the abstract for determination. Please feel free to contact the Office of Research Integrity if you have any questions regarding future protocols that may require IRB review.

Sincerely,

Bruce F. Day, ThD, CIP
Director

APPENDIX B

SOURCE CODE

>>>\youtube\auth\Auth.java

```
package youtube.auth;

import com.google.api.client.auth.oauth2.Credential;
import com.google.api.client.auth.oauth2.StoredCredential;
import com.google.api.client.extensions.java6.auth.oauth2.AuthorizationCodeInstalledApp;
import com.google.api.client.extensions.jetty.auth.oauth2.LocalServerReceiver;
import com.google.api.client.googleapis.auth.oauth2.GoogleAuthorizationCodeFlow;
import com.google.api.client.googleapis.auth.oauth2.GoogleClientSecrets;
import com.google.api.client.http.HttpTransport;
import com.google.api.client.http.javanet.NetHttpTransport;
import com.google.api.client.json.JsonFactory;
import com.google.api.client.json.jackson2.JacksonFactory;
import com.google.api.client.util.store.DataStore;
import com.google.api.client.util.store.FileDataStoreFactory;

import java.io.File;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.Reader;
import java.util.List;

/**
 * Shared class used by every sample. Contains methods for authorizing a user and caching credentials.
 */
public class Auth {

    /**
     * Define a global instance of the HTTP transport.
     */
    public static final HttpTransport HTTP_TRANSPORT = new NetHttpTransport();

    /**
     * Define a global instance of the JSON factory.
     */
    public static final JsonFactory JSON_FACTORY = new JacksonFactory();

    /**
     * This is the directory that will be used under the user's home directory where OAuth tokens will be stored.
     */
    private static final String CREDENTIALS_DIRECTORY = ".oauth-credentials";

    /**
     * Authorizes the installed application to access user's protected data.
     *
     * @param scopes              list of scopes needed to run youtube upload.
     * @param credentialDatastore name of the credential datastore to cache OAuth tokens
     */
    public static Credential authorize(List<String> scopes, String credentialDatastore) throws IOException {

        // Load client secrets.
        Reader clientSecretReader = new InputStreamReader(Auth.class.getResourceAsStream("/client_secrets.json"));
        GoogleClientSecrets clientSecrets = GoogleClientSecrets.load(JSON_FACTORY, clientSecretReader);
```

```
        // Checks that the defaults have been replaced (Default = "Enter X here").
        if (clientSecrets.getDetails().getClientId().startsWith("Enter")
                || clientSecrets.getDetails().getClientSecret().startsWith("Enter ")) {
            System.out.println(
                    "Enter Client ID and Secret from https://console.developers.google.com/project/_/apiui/credential "
                            + "into src/main/resources/client_secrets.json");
            System.exit(1);
        }

        // This creates the credentials datastore at ~/.oauth-credentials/${credentialDatastore}
        FileDataStoreFactory fileDataStoreFactory = new FileDataStoreFactory(new File(System.getProperty("user.home") + "/" +
CREDENTIALS_DIRECTORY));
        DataStore<StoredCredential> datastore = fileDataStoreFactory.getDataStore(credentialDatastore);

        GoogleAuthorizationCodeFlow flow = new GoogleAuthorizationCodeFlow.Builder(
                HTTP_TRANSPORT, JSON_FACTORY, clientSecrets, scopes).setCredentialDataStore(datastore)
                .build();

        // Build the local server and bind it to port 8080
        LocalServerReceiver localReceiver = new LocalServerReceiver.Builder().setPort(8080).build();

        // Authorize.
        return new AuthorizationCodeInstalledApp(flow, localReceiver).authorize("user");
    }
}
```

>>> video_category_discoveror\VideoCategoryEnumerate.java

```
package video_category_discoveror;

import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;

import com.google.api.services.youtube.YouTube;
import com.google.api.services.youtube.model.VideoCategory;
import com.google.api.services.youtube.model.VideoCategoryListResponse;

public class VideoCategoryEnumerate {

        private YouTube youtube;
        private String apiKey;

        public VideoCategoryEnumerate(YouTube youtube, String apiKey) {
                this.youtube = youtube;
                this.apiKey = apiKey;
        }

        public ArrayList<String> getCategoryID() throws IOException {
                ArrayList<String> result = new ArrayList<String>();

                YouTube.VideoCategories.List cList = youtube.videoCategories().list("snippet").setKey(apiKey);
                int intID = 0;
                int emptyCount = 0;
                for (intID = 0; emptyCount < 15; intID++) {
                        String sID = String.valueOf(intID);
                        cList.setId(sID);
                        VideoCategoryListResponse cResponse = cList.execute();
```

```
                        if (cResponse.getItems().isEmpty()) {
                                emptyCount++;
                        } else {
                                emptyCount = 0; // Reset empty counter.
                                Iterator<VideoCategory> itor = cResponse.getItems().iterator();
                                while (itor.hasNext()) {
                                        VideoCategory vCategory = itor.next();
                                        System.out.println("Category ID: " + sID + "\tCategory Title: " +
vCategory.getSnippet().getTitle()
                                                        + "\tAssignable: " +
vCategory.getSnippet().getAssignable());
                                        result.add(vCategory.getId());
                                }
                        }

                }

                // **After verifying, the categories with ID "18", "21", "38", and "42"
                // are empty, i.e., no videos are assigned with these categories. Thus
                // they should be removed from the set.
                result.remove("18");
                result.remove("21");
                result.remove("38");
                result.remove("42");

                return result;
        }

}


>>> video_id_generator\VideoIdCreator.java

package video_id_generator;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.List;

import com.google.api.services.youtube.YouTube;
import com.google.api.services.youtube.model.SearchListResponse;
import com.google.api.services.youtube.model.SearchResult;

import data_collector_ver4.MySQLAccess;
import video_category_discoveror.VideoCategoryEnumerate;

public class VideoIdCreator {

        private YouTube youtube;
        private String apiKey;

        // This attribute passes the initial seed videos' size.
        private int iniSeedSize;
        // Set the maximum seed size for each time expanding.
        private final int expSeedSize = 10000;
        private String order;
```

```
// Page * maximumResult = Related video size.
// i.e. 3 pages and 50 maximum results gives 150 result for each seed video
// when searching for its related videos.
private int page;
private long maximumResult;

// Set the path of where to write the result.
private String filepath;

/**
 * Order can be: date, viewCount, rating, relevance, title (alphabetically)
 * maximumResult cannot exceed 100.
 *
 * @param youtube
 * @param apiKey
 * @param order
 * @param page
 * @param maximumResult
 */
public VideoIdCreator(YouTube youtube, String apiKey, String order, int seedSize, int page, long maximumResult,
                      String filepath) {
            this.youtube = youtube;
            this.apiKey = apiKey;
            this.order = order;
            this.iniSeedSize = seedSize;
            this.page = page;
            this.maximumResult = maximumResult;
            this.filepath = filepath;
}

/**
 *
 * @param expandTime
 * @return
 * @throws Exception
 */
public LinkedHashSet<String> videoIdSetCreate(int expandTime) throws Exception {

            // Create two "cursors".
            LinkedHashSet<String> currentResultSet = new LinkedHashSet<String>();
            LinkedHashSet<String> tempSeedSet = new LinkedHashSet<String>();

            // Initiate the start point.
            currentResultSet = videoIdSeed();
            tempSeedSet = currentResultSet;

            // Store the initial set of videoIDs.
            fileWrite(currentResultSet);
            dbWrite(currentResultSet);

            // If expandTime is -1, then go infinite loops.
            if (expandTime != -1) {
                        while (expandTime > 0) {
                                    System.out.println("Current seed size: " + tempSeedSet.size());
                                    System.out.println("Maximum seed size is set to: " + expSeedSize);
                                    System.out.println("--Start expanding...\n(Remaining expand time: " + expandTime + ")");
                                    expandTime--;
                                    tempSeedSet = videoIdExpand(tempSeedSet, currentResultSet);
                                    currentResultSet.addAll(tempSeedSet);
                                    System.out.println("--current result set size: " + currentResultSet.size());
                                    System.out.println("----------------------------------------------");
                                    // Store the expanded set of videoIDs.
```

```
                                        fileWrite(tempSeedSet);
                                        dbWrite(tempSeedSet);
                                }
                        }else {
                                while (true) {
                                        System.out.println("Current seed size: " + tempSeedSet.size());
                                        System.out.println("Maximum seed size is set to: " + expSeedSize);
                                        System.out.println("--Start expanding...\n(Remaining expand time: " + "\u221E" + ")");
                                        tempSeedSet = videoIdExpand(tempSeedSet, currentResultSet);
                                        currentResultSet.addAll(tempSeedSet);
                                        System.out.println("--current result set size: " + currentResultSet.size());
                                        System.out.println("----------------------------------------------");
                                        // Store the expanded set of videoIDs.
                                        fileWrite(tempSeedSet);
                                        dbWrite(tempSeedSet);
                                }
                        }
                        return currentResultSet;

                }

                // Use video category enumerate to ensure that the seed videos are from
                // distinct categories.
                private LinkedHashSet<String> videoIdSeed() throws IOException {

                        // Get all the categories of YouTube videos.
                        VideoCategoryEnumerate vCE = new VideoCategoryEnumerate(youtube, apiKey);
                        ArrayList<String> vCList = vCE.getCategoryID();

                        // Calculate how many videos should be selected from each category.
                        int categorySize = vCList.size(); // Actually this is constant: 32

                        int blockVideoNum = iniSeedSize / categorySize;
                        int extraVideoNum = iniSeedSize % categorySize;

                        System.out.println();
                        System.out.println("blockVideoNum: " + blockVideoNum);
                        System.out.println("extraVideoNum: " + extraVideoNum);
                        LinkedHashSet<String> videoListSet = new LinkedHashSet<String>();

                        Iterator<String> vCItor = vCList.iterator();
                        while (vCItor.hasNext()) {
                                String vCategory = vCItor.next();
                                int nMaximumResult = blockVideoNum;
                                if (extraVideoNum > 0) {
                                        nMaximumResult += 1;
                                        extraVideoNum -= 1;
                                }
                                System.out.println("--------------------------");
                                System.out.println("Current Category: " + vCategory + "\tnew maximum result: " +
nMaximumResult);
                                YouTube.Search.List videoList = youtube.search().list("id").setKey(apiKey)

                .setFields("items/id/videoId,nextPageToken,pageInfo").setOrder(order).setType("video")
                                                .setMaxResults((long) nMaximumResult).setVideoCategoryId(vCategory);

                                SearchListResponse videoListResponse = videoList.execute();
                                List<SearchResult> videoResultList = videoListResponse.getItems();

                                // while (videoListResponse.getNextPageToken() != null) {
                                // videoListResponse =
                                // videoList.setPageToken(videoListResponse.getNextPageToken()).execute();
```

```
                        // videoResultList.addAll(videoListResponse.getItems());
                        // System.out.println("Infinite loop...");
                        // }

                        if (videoResultList.isEmpty()) {
                                System.out.println("Empty category in ID: " + vCategory);
                        } else {
                                Iterator<SearchResult> videoResultIter = videoResultList.iterator();
                                while (videoResultIter.hasNext()) {
                                        SearchResult video = videoResultIter.next();
                                        System.out.println(video.toString());
                                        videoListSet.add(video.getId().getVideoId());
                                }
                        }
                }
                return videoListSet;
        }

        private LinkedHashSet<String> videoIdExpand(LinkedHashSet<String> seedSet, LinkedHashSet<String>
currentResultSet)
                        throws Exception {

                // Create a container that stores the result for returning.
                LinkedHashSet<String> expandedSet = new LinkedHashSet<String>();

                int page = this.page;

                YouTube.Search.List videoList = youtube.search().list("id").setKey(apiKey)

        .setFields("items/id/videoId,nextPageToken,pageInfo").setType("video").setMaxResults(maximumResult);

                // Iterate the seed set, and for each of them, generate its related
                // video IDs.
                Iterator<String> seedSetIter = seedSet.iterator();

                int cnt = 0;
                long sleepTime = 10000;
                // Set a size limit to the the expand seeds
                long seedLimit = expSeedSize;

                // Calculate total time usage.
                long startTime = System.currentTimeMillis();
                long endTime = startTime;
                double timeDiff = 0;
                // Calculate API execution time usage.
                long exeStartTime = System.currentTimeMillis();
                long exeEndTime = exeStartTime;
                double exeTimeDiff = 0;

                while (seedSetIter.hasNext() && (seedLimit > 0)) {
                        cnt++;
                        seedLimit--;
                        if ((cnt % 500) == 0) {
                                endTime = System.currentTimeMillis();
                                timeDiff = ((double) endTime - startTime) / 1000;
                                System.out.println(cnt + " seeds searched,\tTime used: " + timeDiff + " sec"
                                                + "\tCalling API time used: " + exeTimeDiff / 1000 + " sec.");
                                startTime = endTime;
                                exeTimeDiff = 0;
                        }

                        videoList.setRelatedToVideoId(seedSetIter.next());
```

```
SearchListResponse videoListResponse = new SearchListResponse();

try {
	exeStartTime = System.currentTimeMillis();
	videoListResponse = videoList.execute();
	exeEndTime = System.currentTimeMillis();
	exeTimeDiff += (exeEndTime - exeStartTime);
} catch (Exception e) {
	// Retry at most 3 times to make the request.
	int tryCount = 0;
	int maxRetryTime = 3;
	while (true) {
		try {
			System.out.print("Error occurs. Retrying...");
			// Wait for a few seconds and re-try.
			Thread.sleep(sleepTime);
			exeStartTime = System.currentTimeMillis();
			videoListResponse = videoList.execute();
			exeEndTime = System.currentTimeMillis();
			exeTimeDiff += (exeEndTime - exeStartTime);
			break;
		} catch (Exception e1) {
			// Double the waiting time if request failed.
			sleepTime = sleepTime * 2;
			// Set maximum waiting time to 1 minute.
			if (sleepTime >= 60000) {
				sleepTime = 60000;
			}
			// If failed 3 times, throw the exception.
			if (++tryCount >= maxRetryTime) {
				throw e1;
			}
		}
	}
	System.out.println("Done.");
}

List<SearchResult> videoResultList = videoListResponse.getItems();
page--;

// Keep fetching the next page until it's null or reaching the page
// limit that has been set in the attribute field.
while (videoListResponse.getNextPageToken() != null && page > 0) {
	// Reset the sleep time in case that it has already grown to a
	// big number (even 1 minute is a little bit long for the first
	// few tries.)
	sleepTime = 10000;
	page--;
	videoList.setPageToken(videoListResponse.getNextPageToken());
	try {
		exeStartTime = System.currentTimeMillis();
		videoListResponse = videoList.execute();
		exeEndTime = System.currentTimeMillis();
		exeTimeDiff += (exeEndTime - exeStartTime);
	} catch (Exception e) {
		// Retry at most 3 times to make the request.
		int tryCount = 0;
		int maxRetryTime = 3;
		while (true) {
			try {
				System.out.print("Error occurs. Retrying...");
				// Wait for a few seconds and re-try.
```

```
                                                Thread.sleep(sleepTime);
                                                exeStartTime = System.currentTimeMillis();
                                                videoListResponse = videoList.execute();
                                                exeEndTime = System.currentTimeMillis();
                                                exeTimeDiff += (exeEndTime - exeStartTime);
                                                break;
                                        } catch (Exception e1) {
                                                // Double the waiting time if request failed.
                                                sleepTime = sleepTime * 2;
                                                // Set maximum waiting time to 1 minute.
                                                if (sleepTime >= 60000) {
                                                        sleepTime = 60000;
                                                }
                                                // If failed 3 times, throw the exception.
                                                if (++tryCount >= maxRetryTime) {
                                                        throw e1;
                                                }
                                        }
                                }
                                System.out.println("Done.");
                        }
                        videoResultList.addAll(videoListResponse.getItems());
                }

                // Add the result to the container set.
                Iterator<SearchResult> videoResultIter = videoResultList.iterator();
                while (videoResultIter.hasNext()) {
                        expandedSet.add(videoResultIter.next().getId().getVideoId());
                }
        }

        // Remove the IDs that already exist in the result set.
        System.out.println("--Before removing duplicate size: " + expandedSet.size());
        expandedSet.removeAll(currentResultSet);
        System.out.println("--After removing duplicate size:  " + expandedSet.size());
        return expandedSet;
}

// A short method that writes the result into a file.
private void fileWrite(LinkedHashSet<String> result) throws IOException {

        try {
                File file = new File(filepath);

                BufferedWriter bWriter = new BufferedWriter(new FileWriter(file, true));

                Iterator<String> resultItor = result.iterator();
                while (resultItor.hasNext()) {
                        bWriter.write(resultItor.next() + ",");
                }

                bWriter.close();
        } catch (IOException e) {
                throw e;
        }
}

private void dbWrite(LinkedHashSet<String> result) throws SQLException {

        MySQLAccess dbAccess = new MySQLAccess();
        dbAccess.videoIDListCreator(result);
```

```
	}
}
```

>>>data_collector_ver4\YouTubeAPIProcess.java

```
package data_collector_ver4;

import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.List;

import org.json.JSONObject;

import com.google.api.services.youtube.YouTube;
import com.google.api.services.youtube.model.Channel;
import com.google.api.services.youtube.model.ChannelListResponse;
import com.google.api.services.youtube.model.Comment;
import com.google.api.services.youtube.model.CommentThread;
import com.google.api.services.youtube.model.CommentThreadListResponse;
import com.google.api.services.youtube.model.Video;
import com.google.api.services.youtube.model.VideoCategory;
import com.google.api.services.youtube.model.VideoCategoryListResponse;
import com.google.api.services.youtube.model.VideoListResponse;

/**
 * Designed to combine the multiple processes into one single block. Input
 * should be a set of video, and then generate a plenty of information of
 * YouTube videos. <b>
 *
 * @author Tian
 *
 */
public class YouTubeAPIProcess {

	// Initiate the attributes that are required in the information retrieving
	// process.
	private YouTube youtube;
	private String apiKey;

	private LinkedHashSet<String> videoIdSet = new LinkedHashSet<String>();
	private LinkedHashSet<String> channelIdSet = new LinkedHashSet<String>();
	private LinkedHashSet<String> categoryIdSet = new LinkedHashSet<String>();

	// Initiate the storages of entities that will be inserted to database.
	private ArrayList<JSONObject> videoTableList = new ArrayList<JSONObject>();
	private ArrayList<JSONObject> videoStatisticTableList = new ArrayList<JSONObject>();
	private ArrayList<JSONObject> videoCategoryTableList = new ArrayList<JSONObject>();
	@SuppressWarnings("unchecked")
	// Comments have two type: top level comment, and reply. However, they are
	// retrieved at the same time: when gathering a video's comments, both top
	// level comments and replies are collected at one request. Thus, I used an
	// array to store them. (A better solution can be: create a new class that
	// stores the 2 JSONObject, and return the new class type other than return
	// an array.)
	private ArrayList<JSONObject>[] videoCommentTableList = (ArrayList<JSONObject>[]) new ArrayList[2];
	private ArrayList<JSONObject> channelTableList = new ArrayList<JSONObject>();
```

```
private ArrayList<JSONObject> channelStatisticTableList = new ArrayList<JSONObject>();

// A test attribute for counting users that don't have channel IDs.
private static long noChannelUserCount = 0;

// Basic properties for setting and retrieving attributes.
// ----------------------------------------------------------------
protected YouTubeAPIProcess(YouTube youtube, String apiKey, LinkedHashSet<String> videoIdSet) {
		this.youtube = youtube;
		this.apiKey = apiKey;
		this.videoIdSet = videoIdSet;
}

public void setVideoIdSet(LinkedHashSet<String> videoIdSet) {
		this.videoIdSet = videoIdSet;
}

// public void setChannelIdSet(LinkedHashSet<String> channelIdSet) {
// this.channelIdSet = channelIdSet;
// }
//
// public void setCategoryIdSet(LinkedHashSet<String> categoryIdSet) {
// this.categoryIdSet = categoryIdSet;
// }

public LinkedHashSet<String> getCategoryIdSet() {
		return categoryIdSet;
}

public LinkedHashSet<String> getVideoIdSet() {
		return videoIdSet;
}

public LinkedHashSet<String> getChannelIdSet() {
		return channelIdSet;
}

public ArrayList<JSONObject> getVideoTableList() {
		return videoTableList;
}

public ArrayList<JSONObject> getVideoStatisticTableList() {
		return videoStatisticTableList;
}

public ArrayList<JSONObject> getVideoCategoryTableList() {
		return videoCategoryTableList;
}

public ArrayList<JSONObject>[] getVideoCommentTableList() {
		return videoCommentTableList;
}

public ArrayList<JSONObject> getChannelTableList() {
		return channelTableList;
}

public ArrayList<JSONObject> getChannelStatisticTableList() {
		return channelStatisticTableList;
}
// ----------------------------------------------------------------
```

```
	// Execution method handles the basic process unit, and each one unit
	// contains multiple API calls.
	public YouTubeAPIProcessResult execute() throws Exception {
		generateVideoTableList();
		System.out.println("--Video info retrieved.");
		generateVideoStatisticTableList();
		System.out.println("--Video Statistic info retrieved.");
		generateVideoCategoryTableList();
		System.out.println("--Category info retrieved.");
		generateVideoCommentTableList();
		System.out.println("--Comment info retrieved.");
		generateChannelTableList();
		System.out.println("--Channel info retrieved.");
		generateChannelStatisticTableList();
		System.out.println("--Channel Statistic info retrieved.");
		YouTubeAPIProcessResult processResult = new YouTubeAPIProcessResult(videoTableList,
videoStatisticTableList,
				videoCategoryTableList, videoCommentTableList, channelTableList,
channelStatisticTableList);
		return processResult;
	}

	private void generateVideoTableList() throws IOException {
		videoTableList = new ArrayList<JSONObject>();

		YouTube.Videos.List videoList = youtube.videos().list("id,snippet,contentDetails").setKey(apiKey).setFields(

	"items(id,contentDetails/duration,snippet(categoryId,channelId,description,publishedAt,title))");

		// pre-process the video IDs to make it acceptable by the API.
		String videoIdListCSV = hashSetToCSV(videoIdSet);
		ArrayList<String> splittedVideoIdListCSV = csvSplitter(videoIdListCSV);
		// Each instance contains 50 video IDs separated by commas.
		Iterator<String> videoIdIterator = splittedVideoIdListCSV.iterator();

		while (videoIdIterator.hasNext()) {

			videoList.setId(videoIdIterator.next());
			VideoListResponse videoListResponse = videoList.execute();

			java.util.List<Video> videos = videoListResponse.getItems();
			Iterator<Video> videoIterator = videos.iterator();

			// Create a JSONObject that has the same structure as the MySQL
			// database table. Use the JSONObject to save each entities' values.
			while (videoIterator.hasNext()) {
				Video video = videoIterator.next();
				JSONObject videoInfoTable = new JSONObject().put("VideoId", video.getId())
						.put("CategoryId", video.getSnippet().getCategoryId())
						.put("ChannelId", video.getSnippet().getChannelId())
						.put("VideoPublishedAt",
video.getSnippet().getPublishedAt().toString())
						.put("Duration", video.getContentDetails().getDuration())
						.put("VideoTitle", video.getSnippet().getTitle())
						.put("VideoDescription", video.getSnippet().getDescription());
				videoTableList.add(videoInfoTable);

				// channel and category IDs that related to a particular video
				// should be recorded.
				channelIdSet.add(video.getSnippet().getChannelId());
				categoryIdSet.add(video.getSnippet().getCategoryId());
			}
```

```
		}
	}

	private void generateVideoStatisticTableList() throws IOException {
		// The process is similar to the method above :)
		videoStatisticTableList = new ArrayList<JSONObject>();

		YouTube.Videos.List videoList = youtube.videos().list("id,statistics").setKey(apiKey)
				.setFields("items(id,statistics)");

		String videoIdListCSV = hashSetToCSV(videoIdSet);
		ArrayList<String> splittedVideoIdListCSV = csvSplitter(videoIdListCSV);
		Iterator<String> videoIdIterator = splittedVideoIdListCSV.iterator();

		while (videoIdIterator.hasNext()) {

			videoList.setId(videoIdIterator.next());
			VideoListResponse videoListResponse = videoList.execute();

			java.util.List<Video> videos = videoListResponse.getItems();
			Iterator<Video> videoIterator = videos.iterator();

			while (videoIterator.hasNext()) {
				Video video = videoIterator.next();
				JSONObject videoStatisticTable = new JSONObject().put("VideoId", video.getId())
						.put("VideoTimeStamp", new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date()))
						.put("VideoCommentsCount",
video.getStatistics().getCommentCount())
						.put("VideoDislikeCount", video.getStatistics().getDislikeCount())
						.put("VideoLikeCount", video.getStatistics().getLikeCount())
						.put("VideoFavoriteCount", video.getStatistics().getFavoriteCount())
						.put("VideoViewCount", video.getStatistics().getViewCount());

				videoStatisticTableList.add(videoStatisticTable);
			}
		}
	}

	private void generateVideoCategoryTableList() throws IOException {

		StringBuilder categoryIdBuilder = new StringBuilder();
		videoCategoryTableList = new ArrayList<JSONObject>();

		YouTube.VideoCategories.List videoCategories = youtube.videoCategories().list("snippet").setKey(apiKey)
				.setFields("items(id,snippet/title)");

		Iterator<String> categoryIdSetIterator = categoryIdSet.iterator();
		while (categoryIdSetIterator.hasNext()) {
			categoryIdBuilder.append(categoryIdSetIterator.next() + ",");
		}
		String categoryIdCSV = categoryIdBuilder.toString().replaceAll(",$", "");

		ArrayList<String> splittedCategoryIdCSV = csvSplitter(categoryIdCSV);
		Iterator<String> categoryIdIterator = splittedCategoryIdCSV.iterator();

		while (categoryIdIterator.hasNext()) {
			videoCategories.setId(categoryIdIterator.next());
			VideoCategoryListResponse videoCategoryListResponse = videoCategories.execute();

			List<VideoCategory> videoCategoryList = videoCategoryListResponse.getItems();
			Iterator<VideoCategory> videoCategoryIterator = videoCategoryList.iterator();
```

```
                while (videoCategoryIterator.hasNext()) {
                        VideoCategory videoCategory = videoCategoryIterator.next();
                        JSONObject videoCategoryTable = new JSONObject().put("CategoryId",
videoCategory.getId())
                                .put("CategoryTitle", videoCategory.getSnippet().getTitle());
                        videoCategoryTableList.add(videoCategoryTable);
                }
        }
    }

    @SuppressWarnings("unchecked")
    private void generateVideoCommentTableList() throws Exception {
        String videoIdListCSV = hashSetToCSV(videoIdSet);
        String[] videoIdList = videoIdListCSV.split(",");
        // Create a
        // 0: top level comment; 1: reply.
        videoCommentTableList = (ArrayList<JSONObject>[]) new ArrayList[2];
        for (int i = 0; i < videoCommentTableList.length; i++) {
            videoCommentTableList[i] = new ArrayList<JSONObject>();
        }
        for (String videoId : videoIdList) {
            String curVideoId = videoId;
            Comment curComment = new Comment();

            try {
                YouTube.CommentThreads.List videoCommentsList =
youtube.commentThreads().list("snippet,replies")

    .setKey(apiKey).setVideoId(videoId).setTextFormat("plainText").setMaxResults((long) 100)
                        .setFields(

    "items(replies(comments(id,snippet(authorChannelId,likeCount,parentId,publishedAt,textDisplay,updatedAt))),"
                                +
"snippet(topLevelComment(id,snippet(authorChannelId,likeCount,publishedAt,textDisplay,updatedAt)),"
                                +
"totalReplyCount,videoId)),nextPageToken");
                CommentThreadListResponse videoCommentsListResponse =
videoCommentsList.execute();
                List<CommentThread> commentThreadList = videoCommentsListResponse.getItems();
                // Collect every pages.
                // Set the upper bound of comments to 1000 for now.
                while (videoCommentsListResponse.getNextPageToken() != null) {
                    videoCommentsList =
videoCommentsList.setPageToken(videoCommentsListResponse.getNextPageToken());

                    // Set the initial waiting time for waiting for next try.
                    long sleepTime = 5000;
                    try {
                        videoCommentsListResponse = videoCommentsList.execute();
                    } catch (Exception e) {
                        // retry max 3 times.
                        int tryCount = 0;
                        int maxRetryTime = 3;
                        while (true) {
                            try {
                                tryCount++;
                                System.out.println("**Error occurs, retry time: " +
tryCount + "...");
                                Thread.sleep(sleepTime);
                                videoCommentsListResponse =
videoCommentsList.execute();
                            } catch (Exception e1) {
```

```
                                                        // Double the waiting time if request failed.
                                                        sleepTime = sleepTime * 2;
                                                        // Set maximum waiting time to 1 minute.
                                                        if (sleepTime >= 60000) {
                                                                sleepTime = 60000;
                                                        }
                                                        // If failed 3 times, throw the exception.
                                                        if (tryCount >= maxRetryTime) {
                                                                throw e1;
                                                        }
                                                }
                                        }
                                }
                                commentThreadList.addAll(videoCommentsListResponse.getItems());
                                // Upper bound implementation.
                                if (commentThreadList.size() >= 1000) {
                                        break;
                                }
                        }

                        // Start iterator.
                        Iterator<CommentThread> iterComment = commentThreadList.iterator();
                        while (iterComment.hasNext()) {
                                CommentThread videoComment = iterComment.next();
                                Comment topLevelComment =
videoComment.getSnippet().getTopLevelComment();
                                curComment = topLevelComment;

                                // avoid null author IDs.
                               if  (!topLevelComment.getSnippet().getAuthorChannelId().toString().isEmpty())
{
                                        JSONObject topLevelCommentTable = new
JSONObject().put("TLCommentId", topLevelComment.getId())
                                                        .put("VideoId",
videoComment.getSnippet().getVideoId())
                                                        .put("ChannelId",
                                                                authorChannelIdFormat(
        topLevelComment.getSnippet().getAuthorChannelId().toString()))
                                                        .put("TLCommentLikeCount",
topLevelComment.getSnippet().getLikeCount())
                                                        .put("TLCommentPublishedAt",
topLevelComment.getSnippet().getPublishedAt().toString())
                                                        .put("TLCommentUpdatedAt",
topLevelComment.getSnippet().getUpdatedAt().toString())
                                                        .put("TLCommentTextDisplay",
topLevelComment.getSnippet().getTextDisplay())
                                                        .put("TotalReplyCount",
videoComment.getSnippet().getTotalReplyCount());

                                        videoCommentTableList[0].add(topLevelCommentTable);

                                        channelIdSet.add(
        authorChannelIdFormat(topLevelComment.getSnippet().getAuthorChannelId().toString()));

                                        // If reply exists, add them as well.
                                        if (videoComment.getSnippet().getTotalReplyCount() != 0) {
                                                List<Comment> replies =
videoComment.getReplies().getComments();
                                                Iterator<Comment> iterReply = replies.iterator();
                                                while (iterReply.hasNext()) {
```

```
                                                        Comment reply = iterReply.next();
                                                        if
(!reply.getSnippet().getAuthorChannelId().toString().isEmpty()) {
                                                                JSONObject replyTable = new
JSONObject().put("ReplyId", reply.getId())

        .put("TLCommentId", reply.getSnippet().getParentId())
                                                                        .put("ChannelId",

        authorChannelIdFormat(

                reply.getSnippet().getAuthorChannelId().toString()))

        .put("ReplyLikeCount", reply.getSnippet().getLikeCount())

        .put("ReplyPublishedAt", reply.getSnippet().getPublishedAt().toString())

        .put("ReplyUpdatedAt", reply.getSnippet().getUpdatedAt().toString())

        .put("ReplyTextDisplay", reply.getSnippet().getTextDisplay());


        videoCommentTableList[1].add(replyTable);

                                                                // Save the author's channel id to
                                                                // channelIdList.
                                                                channelIdSet.add(

        authorChannelIdFormat(reply.getSnippet().getAuthorChannelId().toString()));
                                                        }
                                                }
                                        }

                                } else {
                                        // Sometimes a user may not have a channel ID. Instead,
                                        // they are using google+ account to making comments. In
                                        // this case, although I'm not storing the google+
                                        // information yet, it can be separately stored in a new
                                        // table. However, since the number of google+ user is
                                        // far too small, it's not sure yet whether it deserves
                                        // a new table to store the information.
                                       String str = topLevelComment.getSnippet().getAuthorDisplayName();
                                        String                          googleplus                      =
topLevelComment.getSnippet().getAuthorGoogleplusProfileUrl();
                                        System.out.println("--The author \"" + str + "\"'s channel ID not
found."
                                                        + "\n\tThe google+ url is: " + googleplus);
                                }
                        }
                } catch (com.google.api.client.googleapis.json.GoogleJsonResponseException e) {
                        if (e.getStatusCode() != 403) {
                                if (e.getStatusCode() == 404) {
                                        System.out.println("**No video specified.**");
                                } else if (e.getStatusCode() == 400) {
                                        System.out.println("Problem exists in video: " + curVideoId);
                                        Thread.sleep(5000);
                                } else if (e.getStatusCode() == 500 || e.getStatusCode() == 503) {
                                        Thread.sleep(5000);
                                } else {
                                        throw e;
                                }
                        }
```

```
                        } catch (NullPointerException e) {
                                // TODO: handle exception
                                if (!curComment.containsKey("authorChannelId")) {
                                        // This happens when a user is using google+ account other
                                        // than the youtube account.
                                        System.out.println("--Author doesn't have channel ID." + "==> Total: " +
++noChannelUserCount);
                                } else {
                                        throw e;
                                }
                        }
                }
        }

        private void generateChannelTableList() throws Exception {
                channelTableList = new ArrayList<JSONObject>();
                StringBuilder channelIdBuilder = new StringBuilder();

                // Channel ID set grows when other method are executing and getting new
                // channel
                Iterator<String> channelSetIterator = channelIdSet.iterator();
                while (channelSetIterator.hasNext()) {
                        channelIdBuilder.append(channelSetIterator.next() + ",");
                }

                String channelIdCSV = channelIdBuilder.toString().replaceAll(",$", "");

                YouTube.Channels.List channels = youtube.channels().list("id,snippet").setKey(apiKey)
                                .setFields("items(id,snippet(country,description,publishedAt,title))");

                ArrayList<String> splittedChannelIdCSV = csvSplitter(channelIdCSV);
                Iterator<String> channelIdIterator = splittedChannelIdCSV.iterator();

                int chanCount = 0;
                while (channelIdIterator.hasNext()) {
                        channels.setId(channelIdIterator.next());
                        ChannelListResponse channelListResponse = null;
                        long sleepTime = 5000;
                        try {
                                channelListResponse = channels.execute();
                        } catch (Exception e) {
                                // retry max 3 times.
                                int tryCount = 0;
                                int maxRetryTime = 3;
                                while (true) {
                                        try {
                                                tryCount++;
                                                System.out.println("**Error occurs, retry time: " + tryCount + "...");
                                                Thread.sleep(sleepTime);
                                                channelListResponse = channels.execute();
                                        } catch (Exception e1) {
                                                // Double the waiting time if request failed.
                                                sleepTime = sleepTime * 2;
                                                // Set maximum waiting time to 1 minute.
                                                if (sleepTime >= 60000) {
                                                        sleepTime = 60000;
                                                }
                                                // If failed 3 times, throw the exception.
                                                if (tryCount >= maxRetryTime) {
                                                        throw e1;
                                                }
                                        }
```

```
                        }
                    }

                    List<Channel> channelList = channelListResponse.getItems();
                    Iterator<Channel> channelIterator = channelList.iterator();
                    while (channelIterator.hasNext()) {
                        chanCount++;
                        Channel channel = channelIterator.next();
                        JSONObject channelTable = new JSONObject().put("ChannelId", channel.getId())
                                .put("ChannelPublishedAt",
channel.getSnippet().getPublishedAt().toString())
                                .put("ChannelTitle", channel.getSnippet().getTitle())
                                .put("ChannelDescription", channel.getSnippet().getDescription());
                        channelTableList.add(channelTable);
                    }
                }
                System.out.println("# of channels: " + chanCount);
        }

        private void generateChannelStatisticTableList() throws Exception {
                channelStatisticTableList = new ArrayList<JSONObject>();
                StringBuilder channelIdBuilder = new StringBuilder();

                Iterator<String> channelSetIterator = channelIdSet.iterator();
                while (channelSetIterator.hasNext()) {
                    channelIdBuilder.append(channelSetIterator.next() + ",");
                }

                String channelIdCSV = channelIdBuilder.toString().replaceAll(",$", "");

                YouTube.Channels.List channels = youtube.channels().list("id,statistics").setKey(apiKey)
                        .setFields("items(id,statistics)");

                ArrayList<String> splittedChannelIdCSV = csvSplitter(channelIdCSV);
                Iterator<String> channelIdIterator = splittedChannelIdCSV.iterator();

                while (channelIdIterator.hasNext()) {
                    channels.setId(channelIdIterator.next());
                    ChannelListResponse channelListResponse = null;
                    long sleepTime = 5000;
                    try{
                    channelListResponse = channels.execute();
                    }catch (Exception e) {
                        // retry max 3 times.
                        int tryCount = 0;
                        int maxRetryTime = 3;
                        while (true) {
                            try {
                                tryCount++;
                                System.out.println("**Error occurs, retry time: " + tryCount + "...");
                                Thread.sleep(sleepTime);
                                channelListResponse = channels.execute();
                            } catch (Exception e1) {
                                // Double the waiting time if request failed.
                                sleepTime = sleepTime * 2;
                                // Set maximum waiting time to 1 minute.
                                if (sleepTime >= 60000) {
                                    sleepTime = 60000;
                                }
                                // If failed 3 times, throw the exception.
                                if (tryCount >= maxRetryTime) {
                                    throw e1;
```

```
                                                }
                                        }
                                }
                        }
                        List<Channel> channelList = channelListResponse.getItems();
                        Iterator<Channel> channelIterator = channelList.iterator();
                        while (channelIterator.hasNext()) {
                                Channel channel = channelIterator.next();
                                JSONObject channelStatisticTable = new JSONObject().put("ChannelId", channel.getId())
                                                .put("ChannelTimeStamp", new SimpleDateFormat("yyyy-MM-dd
HH:mm:ss").format(new Date()))
                                                .put("ChannelCommentCount",
channel.getStatistics().getCommentCount())
                                                .put("ChannelSubscriberCount",
channel.getStatistics().getSubscriberCount())
                                                .put("ChannelVideoCount", channel.getStatistics().getVideoCount())
                                                .put("ChannelViewCount", channel.getStatistics().getViewCount());
                                channelStatisticTableList.add(channelStatisticTable);
                        }
                }
        }

        private String authorChannelIdFormat(String originalString) {
                return originalString.split("=")[1].replace("}", "");
        }

        // YouTube data API only accept less than 50 length's CSV string, so a
        // splitter is needed.
        private ArrayList<String> csvSplitter(String csvString) {
                ArrayList<String> splittedCSVString = new ArrayList<String>();
                int numberPerChunk = 50; // Split by 50.
                String str = new String();
                // get the index of #50 comma.
                int position = ordinalIndexOf(csvString, ",", numberPerChunk - 1);
                while (position != -1) {
                        str = csvString.substring(0, position);
                        csvString = csvString.substring(position + 1);
                        splittedCSVString.add(str);
                        position = ordinalIndexOf(csvString, ",", numberPerChunk - 1);
                }
                splittedCSVString.add(csvString);
                // The returned value includes a series of string, and each of them
                // contains no more than 50 IDs (or more often, contains exactly 50
                // IDs).
                return splittedCSVString;
        }

        // Assistance subroutine for helping the splitter above.
        private int ordinalIndexOf(String string, String subString, int index) {
                int position = string.indexOf(subString, 0);
                while (index-- > 0 && position != -1) {
                        position = string.indexOf(subString, position + 1);
                }
                return position;
        }

        // This is a transfer function that making a hashSet to a CSV formatted
        // string, which can be used as inputs for the YouTube Data API.
        private String hashSetToCSV(LinkedHashSet<String> idSet) {
                StringBuilder idStringBuilder = new StringBuilder();
                Iterator<String> setIterator = idSet.iterator();
                while (setIterator.hasNext()) {
```

```
                        idStringBuilder.append(setIterator.next() + ",");
                }
                // Dollar is the symbol of the end of a string. The last time the string
                // builder append a value, there is an extra comma at the end of the
                // string, which should be removed.
                return idStringBuilder.toString().replaceAll(",$", "");
        }

}
```

>>>data_collector_ver4\YouTubeAPIProcessResult.java

```
package data_collector_ver4;

import java.util.ArrayList;
import org.json.JSONObject;

/**
 * A structure that stores the result of API process.
 *
 * @author tian
 *
 */
public class YouTubeAPIProcessResult {

        // Initiate the attributes of tables that will be inserted to database.
        private ArrayList<JSONObject> videoTableList = new ArrayList<JSONObject>();
        private ArrayList<JSONObject> videoStatisticTableList = new ArrayList<JSONObject>();
        private ArrayList<JSONObject> videoCategoryTableList = new ArrayList<JSONObject>();
        @SuppressWarnings("unchecked")
        private ArrayList<JSONObject>[] videoCommentTableList = (ArrayList<JSONObject>[]) new ArrayList[2];
        private ArrayList<JSONObject> channelTableList = new ArrayList<JSONObject>();
        private ArrayList<JSONObject> channelStatisticTableList = new ArrayList<JSONObject>();

        @SuppressWarnings("unchecked")
        public YouTubeAPIProcessResult() {
                videoTableList = new ArrayList<JSONObject>();
                videoStatisticTableList = new ArrayList<JSONObject>();
                videoCategoryTableList = new ArrayList<JSONObject>();
                videoCommentTableList = (ArrayList<JSONObject>[]) new ArrayList[2];
                channelTableList = new ArrayList<JSONObject>();
                channelStatisticTableList = new ArrayList<JSONObject>();
        }

        public YouTubeAPIProcessResult(ArrayList<JSONObject> videoTableList, ArrayList<JSONObject>
videoStatisticTableList,
                        ArrayList<JSONObject> videoCategoryTableList, ArrayList<JSONObject>[]
videoCommentTableList,
                        ArrayList<JSONObject> channelTableList, ArrayList<JSONObject> channelStatisticTableList) {
                this.videoTableList = videoTableList;
                this.videoStatisticTableList = videoStatisticTableList;
                this.videoCategoryTableList = videoCategoryTableList;
                this.videoCommentTableList = videoCommentTableList;
                this.channelTableList = channelTableList;
                this.channelStatisticTableList = channelStatisticTableList;
        }

        public ArrayList<JSONObject> getVideoTableList() {
                return videoTableList;
        }
```

```
	public void setVideoTableList(ArrayList<JSONObject> videoTableList) {
		this.videoTableList = videoTableList;
	}

	public ArrayList<JSONObject> getVideoStatisticTableList() {
		return videoStatisticTableList;
	}

	public void setVideoStatisticTableList(ArrayList<JSONObject> videoStatisticTableList) {
		this.videoStatisticTableList = videoStatisticTableList;
	}

	public ArrayList<JSONObject> getVideoCategoryTableList() {
		return videoCategoryTableList;
	}

	public void setVideoCategoryTableList(ArrayList<JSONObject> videoCategoryTableList) {
		this.videoCategoryTableList = videoCategoryTableList;
	}

	public ArrayList<JSONObject>[] getVideoCommentTableList() {
		return videoCommentTableList;
	}

	public void setVideoCommentTableList(ArrayList<JSONObject>[] videoCommentTableList) {
		this.videoCommentTableList = videoCommentTableList;
	}

	public ArrayList<JSONObject> getChannelTableList() {
		return channelTableList;
	}

	public void setChannelTableList(ArrayList<JSONObject> channelTableList) {
		this.channelTableList = channelTableList;
	}

	public ArrayList<JSONObject> getChannelStatisticTableList() {
		return channelStatisticTableList;
	}

	public void setChannelStatisticTableList(ArrayList<JSONObject> channelStatisticTableList) {
		this.channelStatisticTableList = channelStatisticTableList;
	}

}
```

>>>data_collector_ver4\YouTubeAPIProcessThread.java

```
package data_collector_ver4;

import java.util.LinkedHashSet;

import com.google.api.services.youtube.YouTube;

public class YouTubeAPIProcessThread extends Thread {

	private YouTube youtube;
	private String apiKey;
	private LinkedHashSet<String> videoIdSet;
```

```
	public YouTubeAPIProcessThread(YouTube youtube, String apiKey, LinkedHashSet<String> videoIdSet) {
		this.youtube = youtube;
		this.apiKey = apiKey;
		this.videoIdSet = videoIdSet;
	}

	@Override
	public void run() {

		try {
			// Execute the API process to collect the result.
			YouTubeAPIProcess aProcess = new YouTubeAPIProcess(youtube, apiKey, videoIdSet);
			YouTubeAPIProcessResult processResult = aProcess.execute();

			MySQLAccess insertionToDatabase = new MySQLAccess();
			insertionToDatabase.writeToDatabase(processResult.getChannelTableList(),
					processResult.getChannelStatisticTableList(),
processResult.getVideoCategoryTableList(),
					processResult.getVideoTableList(), processResult.getVideoStatisticTableList(),
					processResult.getVideoCommentTableList()[0],
processResult.getVideoCommentTableList()[1]);

		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

	}

}



>>> data_collector_ver4\ArrayListSplit.java

package data_collector_ver4;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.LinkedHashSet;

/**
 * This class is a tool to split a LinkedHashSet into smaller blocks by a given
 * block size. The result sets are stored in an arrayList.
 *
 * @author tian
 *
 * @param <T>
 */
public class ArrayListSplit<T> {

	private LinkedHashSet<String> originSet;
	private int blockSize;

	// Constructor receives the set that requires to be split, and the block
	// size for each split block.
	public ArrayListSplit(LinkedHashSet<String> originSet, int blockSize) {
		this.originSet = originSet;
		this.blockSize = blockSize;
	}
```

```
	// This method executes the split function.
	public ArrayList<LinkedHashSet<String>> split() {

		// Calculate the total number of blocks. If the set size is divisible by
		// the block size then add 0, otherwise add an additional 1.
		int count = originSet.size() / blockSize + (originSet.size() % blockSize == 0 ? 0 : 1);
		// Create the storage for the split sets.
		ArrayList<LinkedHashSet<String>> result = new ArrayList<LinkedHashSet<String>>(count);

		Iterator<String> iter = originSet.iterator();
		for (int i = 0; i < count; i++) {
			// Create a temporary set that stores the elements from the origin
			// set. This set has a fixed size which is set up through the
			// blockSize parameter.
			LinkedHashSet<String> set = new LinkedHashSet<String>(blockSize);
			// Put the elements into the temporary set using a iterator of the
			// original set.
			for (int j = 0; j < blockSize && iter.hasNext(); j++) {
				set.add(iter.next());
			}
			// Add the temporary set into the result arrayList.
			result.add(set);
		}

		return result;
	}
}
```

>>> data_collector_ver4\MySQLAcess.java

```
package data_collector_ver4;

import java.io.IOException;
import java.io.InputStream;
import java.sql.BatchUpdateException;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.sql.Timestamp;
import java.text.DateFormat;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Date;
import java.util.Iterator;
import java.util.LinkedHashSet;
import java.util.Properties;

import org.json.JSONException;
import org.json.JSONObject;

import com.google.api.services.youtube.YouTube.Search;

public class MySQLAccess {
	protected Connection connect = null;
	private Statement statement = null;
	protected ResultSet resultSet = null;
```

```
	private String host;
	private String port;
	private String dbname;
	private String user;
	private String passwd;

	private final String PROPERTIES_FILENAME = "MySQL.properties";
	private Properties properties = new Properties();

	public MySQLAccess() {
		// In the constructor, load the properties of the server setting.
		try {
			InputStream in = Search.class.getResourceAsStream("/" + PROPERTIES_FILENAME);
			properties.load(in);
		} catch (IOException e) {
			System.err.println(
					"There was an error reading " + PROPERTIES_FILENAME + ": " + e.getCause()
+ " : " + e.getMessage());
			System.exit(1);
		}
		host = properties.getProperty("host");
		port = properties.getProperty("port");
		dbname = properties.getProperty("dbname");
		user = properties.getProperty("user");
		passwd = properties.getProperty("passwd");
	}

	protected void establishConnection() throws SQLException {
		// This will load the MySQL driver, each DB has its own driver
		try {
			Class.forName("com.mysql.jdbc.Driver");
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}

		// Setup the connection with the DB
		connect = DriverManager.getConnection("jdbc:mysql://" + host + ":" + port + "/" + dbname + "?" + "user=" +
user
				+ "&password=" + passwd + "&character_set_server=utf8mb4" +
"&rewriteBatchedStatements=true");

	}

	// channel table insertion.
	private void writeChannelToDataBase(ArrayList<JSONObject> channelTableList) throws Exception {

		// The table list contains many entities of channels. They need to be
		// inserted one by one.
		String query = "INSERT INTO Channel (ChannelId, ChannelPublishedAt, ChannelTitle, ChannelDescription)
"
				+ "VALUE (?, ?, ?, ?) ON DUPLICATE KEY UPDATE ChannelId=ChannelId";
		PreparedStatement preparedStatement = connect.prepareStatement(query);

		for (JSONObject channelTable : channelTableList) {
			// Scratch the information from stored JSON object.
			String channelId = channelTable.getString("ChannelId");
			Timestamp channelPublishedAt =
stringToTimestamp(channelTable.getString("ChannelPublishedAt"));
			String channelTitle = channelTable.getString("ChannelTitle");
			String channelDescription = channelTable.getString("ChannelDescription");
```

```
		// Setup the query string.

		try {
			// Pass the values into the statement.
			preparedStatement.setString(1, channelId);
			preparedStatement.setTimestamp(2, channelPublishedAt);
			preparedStatement.setString(3, channelTitle);
			preparedStatement.setString(4, channelDescription);

			preparedStatement.addBatch();
		} catch (java.sql.SQLException e) {
			// // The most common error that occurs is caused by the
			// encoding
			// // format. There're Emojis and some languages that cannot be
			// // accepted by the database, which are mostly contained in
			// the
			// // channel title or the channel description. When this error
			// // happens, locate where it comes from (title or
			// description),
			// // and try to avoid updating that column (or both)
			// information.
			// if (e.getMessage().contains("ChannelTitle") &&
			// !e.getMessage().contains("ChannelDescription")) {
			// preparedStatement.setString(1, channelId);
			// preparedStatement.setTimestamp(2, channelPublishedAt);
			// preparedStatement.setString(3, "");
			// preparedStatement.setString(4, channelDescription);
			// } else if (!e.getMessage().contains("ChannelTitle") &&
			// e.getMessage().contains("ChannelDescription")) {
			// preparedStatement.setString(1, channelId);
			// preparedStatement.setTimestamp(2, channelPublishedAt);
			// preparedStatement.setString(3, channelTitle);
			// preparedStatement.setString(4, "");
			// } else if (e.getMessage().contains("ChannelTitle") &&
			// e.getMessage().contains("ChannelDescription")) {
			// preparedStatement.setString(1, channelId);
			// preparedStatement.setTimestamp(2, channelPublishedAt);
			// preparedStatement.setString(3, "");
			// preparedStatement.setString(4, "");
			// } else {
			// System.out.println(e.getMessage());
			// }
			// try {
			// preparedStatement.executeUpdate();
			// } catch (SQLException e1) {
			// System.out.println(e1.getMessage());
			// }
		}
}

try {
		preparedStatement.executeBatch();
} catch (BatchUpdateException e) {
		// TODO: handle exception
		System.out.println(e.getMessage());
		int[] counts = e.getUpdateCounts();
		int successCount = 0;
		int notAvaliable = 0;
		int failCount = 0;
		for (int i = 0; i < counts.length; i++) {
			if (counts[i] >= 0) {
				successCount++;
```

```
                        } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                                notAvaliable++;
                        } else if (counts[i] == Statement.EXECUTE_FAILED) {
                                failCount++;
                        }
                }
                System.out.println("---Channel---");
                System.out.println("Number of affected rows: " + successCount);
                System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                System.out.println("Failed count in batch: " + failCount);
        }
}

// video category insertion.
private void writeVideoCategoryToDatabase(ArrayList<JSONObject> videoCategoryTableList)
                throws SQLException, IOException {

        // Setup the query string.
        String query = "INSERT INTO VideoCategory (CategoryId, CategoryTitle) "
                        + "VALUE (?, ?) ON DUPLICATE KEY UPDATE CategoryId=CategoryId";
        PreparedStatement preparedStatement = connect.prepareStatement(query);

        for (JSONObject videoCategoryTable : videoCategoryTableList) {
                // Scratch the information from stored JSON object.
                String categoryId = videoCategoryTable.getString("CategoryId");
                String CategoryTitle = videoCategoryTable.getString("CategoryTitle");

                try {
                        // Pass the values into the statement.
                        preparedStatement.setString(1, categoryId);
                        preparedStatement.setString(2, CategoryTitle);

                        preparedStatement.addBatch();
                } catch (SQLException e) {
                        // Do nothing.
                }
        }

        try {
                preparedStatement.executeBatch();
        } catch (BatchUpdateException e) {
                // TODO: handle exception
                System.out.println(e.getMessage());
                int[] counts = e.getUpdateCounts();
                int successCount = 0;
                int notAvaliable = 0;
                int failCount = 0;
                for (int i = 0; i < counts.length; i++) {
                        if (counts[i] >= 0) {
                                successCount++;
                        } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                                notAvaliable++;
                        } else if (counts[i] == Statement.EXECUTE_FAILED) {
                                failCount++;
                        }
                }
                System.out.println("---Video category---");
                System.out.println("Number of affected rows: " + successCount);
                System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                System.out.println("Failed count in batch: " + failCount);
        }
```

```
	}

	// video table insertion.
	private void writeVideoToDatabase(ArrayList<JSONObject> videoTableList)
			throws SQLException, JSONException, ParseException, IOException {

		// Setup the query string.
		String query = "INSERT INTO Video (VideoId, CategoryId, ChannelId, VideoPublishedAt, Duration, VideoTitle, VideoDescription) "
				+ "VALUE (?, ?, ?, ?, ?, ?, ?) ON DUPLICATE KEY UPDATE VideoId=VideoId";
		PreparedStatement preparedStatement = connect.prepareStatement(query);

		for (JSONObject videoTable : videoTableList) {
			// Scratch the information from stored JSON object.
			String videoId = videoTable.getString("VideoId");
			String CategoryId = videoTable.getString("CategoryId");
			String ChannelId = videoTable.getString("ChannelId");
			Timestamp VideoPublishedAt = stringToTimestamp(videoTable.getString("VideoPublishedAt"));
			String Duration = videoTable.getString("Duration");
			String VideoTitle = videoTable.getString("VideoTitle");
			String VideoDescription = videoTable.getString("VideoDescription");

			try {
				// Pass the values into the statement.
				preparedStatement.setString(1, videoId);
				preparedStatement.setString(2, CategoryId);
				preparedStatement.setString(3, ChannelId);
				preparedStatement.setTimestamp(4, VideoPublishedAt);
				preparedStatement.setString(5, Duration);
				preparedStatement.setString(6, VideoTitle);
				preparedStatement.setString(7, VideoDescription);

				preparedStatement.addBatch();

			} catch (SQLException e) {
				// // In the video info insertion step, the most common error is
				// // caused by the video description. If it happens, identify
				// it
				// // and avoid inserting that column.
				// if (e.getMessage().contains("VideoDescription")) {
				// preparedStatement.setString(1, videoId);
				// preparedStatement.setString(2, CategoryId);
				// preparedStatement.setString(3, ChannelId);
				// preparedStatement.setTimestamp(4, VideoPublishedAt);
				// preparedStatement.setString(5, Duration);
				// preparedStatement.setString(6, VideoTitle);
				// preparedStatement.setString(7, "");
				//
				// preparedStatement.addBatch();
				// } else {
				// // Do nothing.
				// }
			}
		}

		try {
			preparedStatement.executeBatch();
		} catch (BatchUpdateException e) {
			// TODO: handle exception
			System.out.println(e.getMessage());
			int[] counts = e.getUpdateCounts();
			int successCount = 0;
```

```
                        int notAvaliable = 0;
                        int failCount = 0;
                        for (int i = 0; i < counts.length; i++) {
                                if (counts[i] >= 0) {
                                        successCount++;
                                } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                                        notAvaliable++;
                                } else if (counts[i] == Statement.EXECUTE_FAILED) {
                                        failCount++;
                                }
                        }
                        System.out.println("---Video---");
                        System.out.println("Number of affected rows: " + successCount);
                        System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                        System.out.println("Failed count in batch: " + failCount);
                }
        }

        // video statistic insertion.
        private void writeVideoStatisticToDatabase(ArrayList<JSONObject> videoStatisticTableList)
                        throws SQLException, JSONException, ParseException, IOException {

                // Setup the query string.
                String query = "INSERT INTO VideoStatistic (VideoId, VideoTimeStamp, VideoCommentsCount,
VideoDislikeCount, VideoFavoriteCount, VideoLikeCount, VideoViewCount) "
                                + "VALUE (?, ?, ?, ?, ?, ?, ?) ON DUPLICATE KEY UPDATE VideoId=VideoId";

                PreparedStatement preparedStatement = connect.prepareStatement(query);

                for (JSONObject videoStatisticTable : videoStatisticTableList) {
                        // Scratch the information from stored JSON object.
                        String videoId = videoStatisticTable.getString("VideoId");
                        Date date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
                                        .parse(videoStatisticTable.getString("VideoTimeStamp"));
                        Timestamp videoTimeStamp = new java.sql.Timestamp(date.getTime());
                        long videoFavoriteCount = videoStatisticTable.getLong("VideoFavoriteCount");
                        long videoViewCount = videoStatisticTable.getLong("VideoViewCount");
                        // Some videos may not allow these information. By default, set them
                        // to 0.
                        long videoLikeCount = 0;
                        long videoDislikeCount = 0;
                        long videoCommentsCount = 0;
                        // If there's no information in such videos, there will be an JSON
                        // exception when I call the non-exist key.
                        try {
                                videoLikeCount = videoStatisticTable.getLong("VideoLikeCount");
                                videoDislikeCount = videoStatisticTable.getLong("VideoDislikeCount");
                                videoCommentsCount = videoStatisticTable.getLong("VideoCommentsCount");
                        } catch (Exception e) {
                                // The variables remain the default values which are all 0.
                        }

                        try {
                                // Pass the values into the statement.
                                preparedStatement.setString(1, videoId);
                                preparedStatement.setTimestamp(2, videoTimeStamp);
                                preparedStatement.setLong(3, videoCommentsCount);
                                preparedStatement.setLong(4, videoDislikeCount);
                                preparedStatement.setLong(5, videoFavoriteCount);
                                preparedStatement.setLong(6, videoLikeCount);
                                preparedStatement.setLong(7, videoViewCount);
```

```
                        preparedStatement.addBatch();
                    } catch (SQLException e) {
                        // Do nothing.
                    }
                }

                try {
                    preparedStatement.executeBatch();
                } catch (BatchUpdateException e) {
                    // TODO: handle exception
                    System.out.println(e.getMessage());
                    int[] counts = e.getUpdateCounts();
                    int successCount = 0;
                    int notAvaliable = 0;
                    int failCount = 0;
                    for (int i = 0; i < counts.length; i++) {
                        if (counts[i] >= 0) {
                            successCount++;
                        } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                            notAvaliable++;
                        } else if (counts[i] == Statement.EXECUTE_FAILED) {
                            failCount++;
                        }
                    }
                    System.out.println("---Video statistics---");
                    System.out.println("Number of affected rows: " + successCount);
                    System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                    System.out.println("Failed count in batch: " + failCount);
                }
            }

            // channel statistic insertion.
            private void writeChannelStatisticToDatebase(ArrayList<JSONObject> channelStatisticTableList)
                    throws SQLException, JSONException, ParseException, IOException {

                // Setup the query string.
                String query = "INSERT INTO ChannelStatistic (ChannelId, ChannelTimeStamp, ChannelCommentCount,
ChannelSubscriberCount, ChannelVideoCount, ChannelViewCount) "
                        + "VALUE (?, ?, ?, ?, ?, ?) ON DUPLICATE KEY UPDATE ChannelId=ChannelId";

                PreparedStatement preparedStatement = connect.prepareStatement(query);

                for (JSONObject channelStatisticTable : channelStatisticTableList) {
                    // Scratch the information from stored JSON object.
                    String channelId = channelStatisticTable.getString("ChannelId");
                    Date date = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss")
                            .parse(channelStatisticTable.getString("ChannelTimeStamp"));
                    Timestamp channelTimeStamp = new java.sql.Timestamp(date.getTime());
                    long channelCommentCount = channelStatisticTable.getLong("ChannelCommentCount");
                    long channelSubscriberCount = channelStatisticTable.getLong("ChannelSubscriberCount");
                    long channelVideoCount = channelStatisticTable.getLong("ChannelVideoCount");
                    long channelViewCount = channelStatisticTable.getLong("ChannelViewCount");

                    try {
                        // Pass the values into the statement.
                        preparedStatement.setString(1, channelId);
                        preparedStatement.setTimestamp(2, channelTimeStamp);
                        preparedStatement.setLong(3, channelCommentCount);
                        preparedStatement.setLong(4, channelSubscriberCount);
                        preparedStatement.setLong(5, channelVideoCount);
                        preparedStatement.setLong(6, channelViewCount);
```

```
                        preparedStatement.addBatch();
                    } catch (SQLException e) {
                        // Do nothing.
                    }
                }

                try {
                    preparedStatement.executeBatch();
                } catch (BatchUpdateException e) {
                    // TODO: handle exception
                    System.out.println(e.getMessage());
                    int[] counts = e.getUpdateCounts();
                    int successCount = 0;
                    int notAvaliable = 0;
                    int failCount = 0;
                    for (int i = 0; i < counts.length; i++) {
                        if (counts[i] >= 0) {
                            successCount++;
                        } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                            notAvaliable++;
                        } else if (counts[i] == Statement.EXECUTE_FAILED) {
                            failCount++;
                        }
                    }
                    System.out.println("---Channel statistics---");
                    System.out.println("Number of affected rows: " + successCount);
                    System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                    System.out.println("Failed count in batch: " + failCount);
                }
            }

            // top level comment insertion.
            private void writeTopLevelCommentToDatebase(ArrayList<JSONObject> topLevelCommentTableList)
                    throws SQLException, JSONException, ParseException, IOException {

                // Setup the query string.
                String query = "INSERT INTO TopLevelComment (TLCommentId, VideoId, ChannelId,
TLCommentLikeCount, TLCommentPublishedAt, TLCommentUpdatedAt, TLCommentTextDisplay, TotalReplyCount) "
                        + "VALUE (?, ?, ?, ?, ?, ?, ?, ?) ON DUPLICATE KEY UPDATE
TLCOmmentId=TLCOmmentId";

                PreparedStatement preparedStatement = connect.prepareStatement(query);

                for (JSONObject topLevelCommentTable : topLevelCommentTableList) {
                    // Scratch the information from stored JSON object.
                    String tlcommentId = topLevelCommentTable.getString("TLCommentId");
                    String videoId = topLevelCommentTable.getString("VideoId");
                    String channelId = topLevelCommentTable.getString("ChannelId");
                    long tlcommentLikeCount = topLevelCommentTable.getLong("TLCommentLikeCount");
                    Timestamp TLCommentPublishedAt =
stringToTimestamp(topLevelCommentTable.getString("TLCommentPublishedAt"));
                    Timestamp TLCommentUpdatedAt =
stringToTimestamp(topLevelCommentTable.getString("TLCommentUpdatedAt"));
                    String TLCommentTextDisplay = topLevelCommentTable.getString("TLCommentTextDisplay");
                    long TotalReplyCount = topLevelCommentTable.getLong("TotalReplyCount");

                    try {
                        // Pass the values into the statement.
                        preparedStatement.setString(1, tlcommentId);
                        preparedStatement.setString(2, videoId);
                        preparedStatement.setString(3, channelId);
                        preparedStatement.setLong(4, tlcommentLikeCount);
```

```
                                        preparedStatement.setTimestamp(5, TLCommentPublishedAt);
                                        preparedStatement.setTimestamp(6, TLCommentUpdatedAt);
                                        preparedStatement.setString(7, TLCommentTextDisplay);
                                        preparedStatement.setLong(8, TotalReplyCount);

                                        preparedStatement.addBatch();
                                } catch (SQLException e) {
                                        // // As before, the encoding incompatible may cause errors. If
                                        // it
                                        // // occurs, avoid updating the column that possibly causes the
                                        // // error (here the column is TLCommentTextDisplay).
                                        // if (e.getMessage().contains("TLCommentTextDisplay")) {
                                        // preparedStatement.setString(1, tlcommentId);
                                        // preparedStatement.setString(2, videoId);
                                        // preparedStatement.setString(3, channelId);
                                        // preparedStatement.setInt(4, tlcommentLikeCount);
                                        // preparedStatement.setTimestamp(5, TLCommentPublishedAt);
                                        // preparedStatement.setTimestamp(6, TLCommentUpdatedAt);
                                        // preparedStatement.setString(7, "");
                                        // preparedStatement.setInt(8, TotalReplyCount);
                                        //
                                        // preparedStatement.addBatch();
                                        // } else {
                                        // // Do not add this comment.
                                        // }
                                }
                        }

                        try {
                                preparedStatement.executeBatch();
                        } catch (BatchUpdateException e) {
                                // TODO: handle exception
                                System.out.println(e.getMessage());
                                int[] counts = e.getUpdateCounts();
                                int successCount = 0;
                                int notAvaliable = 0;
                                int failCount = 0;
                                for (int i = 0; i < counts.length; i++) {
                                        if (counts[i] >= 0) {
                                                successCount++;
                                        } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                                                notAvaliable++;
                                        } else if (counts[i] == Statement.EXECUTE_FAILED) {
                                                failCount++;
                                        }
                                }
                                System.out.println("---Top level comment---");
                                System.out.println("Number of affected rows: " + successCount);
                                System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                                System.out.println("Failed count in batch: " + failCount);
                        }
                }

                private void writeReplyToDatabase(ArrayList<JSONObject> replyTableList)
                                throws SQLException, JSONException, ParseException, IOException {

                        // Setup the query string.
                        String query = "INSERT INTO Reply (ReplyId, TLCOmmentId, ChannelId, ReplyLikeCount,
ReplyPublishedAt, ReplyUpdatedAt, ReplyTextDisplay) "
                                        + "VALUE (?, ?, ?, ?, ?, ?, ?) ON DUPLICATE KEY UPDATE
TLCOmmentId=TLCOmmentId";
```

```
PreparedStatement preparedStatement = connect.prepareStatement(query);

for (JSONObject replyTable : replyTableList) {
	// Scratch the information from stored JSON object.
	String replyId = replyTable.getString("ReplyId");
	String tlcommentId = replyTable.getString("TLCommentId");
	String channelId = replyTable.getString("ChannelId");
	long replyLikeCount = replyTable.getLong("ReplyLikeCount");
	Timestamp replyPublishedAt = stringToTimestamp(replyTable.getString("ReplyPublishedAt"));
	Timestamp replyUpdatedAt = stringToTimestamp(replyTable.getString("ReplyUpdatedAt"));
	String replyTextDisplay = replyTable.getString("ReplyTextDisplay");

	try {
		// Pass the values into the statement.
		preparedStatement.setString(1, replyId);
		preparedStatement.setString(2, tlcommentId);
		preparedStatement.setString(3, channelId);
		preparedStatement.setLong(4, replyLikeCount);
		preparedStatement.setTimestamp(5, replyPublishedAt);
		preparedStatement.setTimestamp(6, replyUpdatedAt);
		preparedStatement.setString(7, replyTextDisplay);

		preparedStatement.addBatch();
	} catch (SQLException e) {
		// // Similarly, avoid updating column "ReplyTextDisplay" if it
		// // results in an encoding incompatible error.
		// if (e.getMessage().contains("ReplyTextDisplay")) {
		// preparedStatement.setString(1, replyId);
		// preparedStatement.setString(2, tlcommentId);
		// preparedStatement.setString(3, channelId);
		// preparedStatement.setInt(4, replyLikeCount);
		// preparedStatement.setTimestamp(5, replyPublishedAt);
		// preparedStatement.setTimestamp(6, replyUpdatedAt);
		// preparedStatement.setString(7, "");
		//
		// preparedStatement.addBatch();
		// } else {
		// System.out.println(e.toString());
		// }
	}
}

try {
	preparedStatement.executeBatch();
} catch (BatchUpdateException e) {
	// TODO: handle exception
	System.out.println(e.getMessage());
	int[] counts = e.getUpdateCounts();
	int successCount = 0;
	int notAvaliable = 0;
	int failCount = 0;
	for (int i = 0; i < counts.length; i++) {
		if (counts[i] >= 0) {
			successCount++;
		} else if (counts[i] == Statement.SUCCESS_NO_INFO) {
			notAvaliable++;
		} else if (counts[i] == Statement.EXECUTE_FAILED) {
			failCount++;
		}
	}
	System.out.println("---Reply---");
	System.out.println("Number of affected rows: " + successCount);
```

```
                        System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                        System.out.println("Failed count in batch: " + failCount);
                }
        }

        // A method that combine and organize the insertions.
        public void writeToDatabase(ArrayList<JSONObject> channelTableList, ArrayList<JSONObject>
channelStatisticTableList,
                        ArrayList<JSONObject> videoCategoryTableList, ArrayList<JSONObject> videoTableList,
                        ArrayList<JSONObject> videoStatisticTableList, ArrayList<JSONObject>
topLevelCommentTableList,
                        ArrayList<JSONObject> replyTableList) throws Exception {

                establishConnection();
                System.out.println("------------------------------------------------");
                System.out.println("----------------DATABASE INSERTING---------------");
                writeChannelToDataBase(channelTableList);
                System.out.println("--Channel updated.");
                writeChannelStatisticToDatebase(channelStatisticTableList);
                System.out.println("--Channel Statistic updated.");
                writeVideoCategoryToDatabase(videoCategoryTableList);
                System.out.println("--Category updated.");
                writeVideoToDatabase(videoTableList);
                System.out.println("--Video updated.");
                writeVideoStatisticToDatabase(videoStatisticTableList);
                System.out.println("--Video Statistic updated.");
                writeTopLevelCommentToDatebase(topLevelCommentTableList);
                System.out.println("--TLComment updated.");
                writeReplyToDatabase(replyTableList);
                System.out.println("--Reply updated.");
                System.out.println("----------------INSERTION COMPLETE---------------");
                System.out.println("------------------------------------------------");
                close();

                Thread.sleep(2000);

        }

        // You need to close the resultSet
        protected void close() {
                try {
                        if (resultSet != null) {
                                resultSet.close();
                        }

                        if (statement != null) {
                                statement.close();
                        }

                        if (connect != null) {
                                connect.close();
                        }
                } catch (SQLException e) {
                        e.printStackTrace();
                }
        }

        private Timestamp stringToTimestamp(String timeString) throws ParseException {
                DateFormat df1 = new SimpleDateFormat("yyyy-MM-dd'T'HH:mm:ss.SSS");
                Date result = df1.parse(timeString.replace("Z", ""));
                java.sql.Timestamp ts = new java.sql.Timestamp(result.getTime());
                return ts;
```

```
	}

	// This method gets a set of video IDs by accessing the database.
	public LinkedHashSet<String> readVideoIdList(int limit) throws Exception {

		establishConnection();

		LinkedHashSet<String> retrievedVideoId = new LinkedHashSet<String>();

		// Query strings:
		String selectQuery = "select VideoId " + "from VideoIdRecord " + "order by CrawledTime asc " + "limit ?";
		String updateQuery = "update VideoIdRecord " + "set CrawledTime = CrawledTime + 1 " + "where VideoId
= ?";

		PreparedStatement selectStatement = connect.prepareStatement(selectQuery);
		PreparedStatement updateStatement = connect.prepareStatement(updateQuery);
		try {
			selectStatement.setInt(1, limit);

			resultSet = selectStatement.executeQuery();

			while (resultSet.next()) {
				String videoId = resultSet.getString("VideoId");
				retrievedVideoId.add(videoId);

				updateStatement.setString(1, videoId);
				updateStatement.executeUpdate();
			}
		} catch (Exception e) {
			// TODO: handle exception
			throw e;
		} finally {
			close();
		}
		return retrievedVideoId;
	}

	// This method is used for creating a big video ID base list, which will be
	// used for further crawling.
	public void videoIDListCreator(LinkedHashSet<String> videoIdSet) throws SQLException {

		establishConnection();
		// ** probably can be done with LOAD DATA INFILE, which is much faster
		// than insertion.

		// ** currently using plain INSERT
		Iterator<String> videoIdItor = videoIdSet.iterator();
		//
		String query = "INSERT INTO VideoIdRecord (VideoId, CrawledTime)" + "VALUE (?, ?)"
				+ "ON DUPLICATE KEY UPDATE VideoId=VideoId";

		PreparedStatement preparedStatement = connect.prepareStatement(query);

		int count = 0;
		// Use a timer to count the batch initialization time.
		long startTime = System.currentTimeMillis();
		while (videoIdItor.hasNext()) {
			String videoId = videoIdItor.next();
			count++;
			if (count % 1000 == 0) {
				System.out.println(String.format("%d", (count / 1000)) + " thousand collected.");
			}
```

```
                        try {
                                preparedStatement.setString(1, videoId);
                                preparedStatement.setInt(2, 0);
                                preparedStatement.addBatch();

                        } catch (SQLException e) {
                                // TODO: handle exception
                                e.printStackTrace();
                        }
                }
                long endTime = System.currentTimeMillis();
                System.out.println("Batch initialization time: " + ((double) endTime - startTime) / 1000 + " sec");

                try {
                        System.out.print(count + " INSERT queries are added to the batch. \nNow inserting to the
database...");
                        long batchStartTime = System.currentTimeMillis();
                        int[] updateCount = preparedStatement.executeBatch();
                        long batchEndTime = System.currentTimeMillis();
                        System.out.println("Done.\nBatch run time: " + ((double) batchEndTime - batchStartTime) / 1000 +
" sec");
                        System.out.println("Total inserted: " + updateCount.length);
                } catch (BatchUpdateException e) {
                        // TODO: handle exception
                        System.out.println(e.getMessage());
                        int[] counts = e.getUpdateCounts();
                        int successCount = 0;
                        int notAvaliable = 0;
                        int failCount = 0;
                        for (int i = 0; i < counts.length; i++) {
                                if (counts[i] >= 0) {
                                        successCount++;
                                } else if (counts[i] == Statement.SUCCESS_NO_INFO) {
                                        notAvaliable++;
                                } else if (counts[i] == Statement.EXECUTE_FAILED) {
                                        failCount++;
                                }
                        }
                        System.out.println("Number of affected rows: " + successCount);
                        System.out.println("Number of affected rows (not avaliable): " + notAvaliable);
                        System.out.println("Failed count in batch: " + failCount);
                } catch (SQLException e1) {
                        // TODO: handle exception
                        e1.printStackTrace();
                } finally {
                        preparedStatement.close();
                        close();
                }

                System.out.print("\nID record updated.");

        }

}
```

>>>data_collector_ver4\DBAccessUpdating.java

```
package data_collector_ver4;
```

```
import java.sql.PreparedStatement;
import java.sql.SQLException;

public class DBAccessUpdating extends MySQLAccess {

	public void updateRecordID() throws SQLException {

		super.establishConnection();
		String query = "SELECT COUNT(*) FROM videoidrecord";
		PreparedStatement pStatement = super.connect.prepareStatement(query);
		super.resultSet = pStatement.executeQuery();

		long ID = 0;
		while (super.resultSet.next()) {
			ID = resultSet.getLong("COUNT(*)");
			System.out.println(ID);
		}

		String query2 = "UPDATE videoidrecord SET ID=? WHERE ID=NULL";
		PreparedStatement pStatement2 = super.connect.prepareStatement(query2);
		for (long l = 1; l < ID; l++) {
			pStatement2.setString(1, String.valueOf(l));
			pStatement2.executeUpdate();
		}

		super.close();
	}

}



>>> data_collector_ver4\Main.java

package data_collector_ver4;

import java.io.File;
import java.io.IOException;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.LinkedHashSet;
import java.util.Scanner;

import com.google.api.services.youtube.YouTube;

import video_id_generator.VideoIdCreator;

public class Main {

	/**
	 * main class that handles the whole process:
	 * <q>1. Select whether or not update new video IDs to the database;
	 * <q>2. Retrieve a subset of video IDs from database.videoIdRecord;
	 * <q>3. Process the API to get the information of the retrieved videos;
	 * <q>4. Insert the informations, and mark the successfully crawled videos
	 * as "crawled".
	 * <q>5. Re-do from step2.
	 *
	 * @param args
	 * @throws Exception
	 */
	public static void main(String args[]) throws Exception {
```

```
		String mode = args[0];
		YouTubeAuth yAuth = new YouTubeAuth();
		YouTube youtube = yAuth.getYouTube();
		String apiKey = yAuth.getApiKey();
		MySQLAccess dbAccess = new MySQLAccess();

		// Whether or not importing new IDs to the DB.
		if (mode.equals("0")) {
			System.out.println("Mode: 0");
			// skip generation of new video IDs
			// Loop of main process block:
			int count = 1000;
			while (count-- > 0) {
				System.out.println("Process remain: " + count + " times.");
				// Each block read 20 videos:
				LinkedHashSet<String> videoIdSet = dbAccess.readVideoIdList(20);
				System.out.println("--VideoId read.");
				YouTubeAPIProcessThread apiProcessThread = new
YouTubeAPIProcessThread(youtube, apiKey, videoIdSet);
				apiProcessThread.run();
				apiProcessThread.join();
			}

		} else if (mode.equals("1")) {
			System.out.println("Mode: 1");
			// upload a list of video IDs to the database.
			// input the size of the video ID.
			int seedSize = 0;
			int pageNum = 0;
			long resultPerPage = 0;
			Scanner input = new Scanner(System.in);
			System.out.println(
					"-------------------------------------------------------------------------------------------
");
			System.out.println(
					"Ready to generate a video ID list. At first, a given number of seed videos are
collected from \ndifferent categories of YouTube."
							+ "\nThen from each seed video, retrieve its \"related
videos\" and treat them as new \"seed videos\"."
							+ "\nWhen retrieving related videos, the API generates a few
pages of videos of which maximum \nresult is limited by 100.");
			System.out.println(
					"-------------------------------------------------------------------------------------------
");
			String[] inputString;
			// Read values from input.
			do {
				System.out.print(
						"Please input the seed size of videos, total page number, and maximum
result per page (separated by comma): ");
				inputString = input.nextLine().split(",");
			} while (inputString.length != 3);
			seedSize = Integer.valueOf(inputString[0].trim());
			pageNum = Integer.valueOf(inputString[1].trim());
			resultPerPage = Integer.valueOf(inputString[2].trim());
			VideoIdCreator vIdCreator = new VideoIdCreator(youtube, apiKey, "date", seedSize, pageNum,
					resultPerPage, makeFilePath(input));
			// Also input the expanding (i.e., get access to the related videos)
			// times.
			System.out.println(System.in.available());
			System.out.println("Please input the expand time of the seed videos: ");
```

```
                        int expandTime = input.nextInt();
                        System.out.println("Input completed. \n**Notice: Current maximum seed videos are set to 10000");
                        LinkedHashSet<String> videoIdSet = vIdCreator.videoIdSetCreate(expandTime);
                        input.close();
                        System.out.println("In total " + videoIdSet.size() + " IDs are created.");
                        System.out.println("Insertion finished.");
                } else {
                        System.out.println(
                                        "Invalid argument. " + "\nValid argument options: \n 0 - Skip the generation of
new video IDs"
                                                        + "\n 1 - Start generating new video IDs and upload them to
the DB");
                        System.exit(0);
                }

        }

        private static String makeFilePath(Scanner input) throws IOException {
                String dateStr = new SimpleDateFormat("yyyyMMdd").format(new Date());
                String pathname = "videoId" + File.separator + dateStr + "_videoId.txt";
                try {
                        File file = new File(pathname);
                        if (!file.exists()) {
                                System.out.println("filepath: " + pathname);
                                file.getParentFile().mkdirs();
                                file.createNewFile();
                                if (file.exists()) {
                                        System.out.println("->file created.");
                                }
                        } else {
                                System.out.println("File already exist. Rename(1) or replace(2)?");
                                String mode = new String();
                                do {
                                        mode = input.nextLine();
                                } while (!(mode.equals("1") || mode.equals("2")));
                                if (mode.equals("1")) {
                                        String time = new SimpleDateFormat("HHmmss").format(new Date());
                                        pathname = pathname.split("\\.")[0] + "_" + time + ".txt";
                                        System.out.println("filepath: " + pathname);
                                        file.getParentFile().mkdirs();
                                        file.createNewFile();
                                        if (file.exists()) {
                                                System.out.println("->file created.");
                                        }
                                } else if (mode.equals("2")) {
                                        System.out.println("File is deleted: " + file.delete());
                                        System.out.println("filepath: " + pathname);
                                        file.getParentFile().mkdirs();
                                        file.createNewFile();
                                        if (file.exists()) {
                                                System.out.println("->file created.");
                                        }
                                } else {
                                        System.out.println("Should never reach this point.");
                                }
                        }

                } catch (IOException e) {
                        throw e;
                }

                return pathname;
```

```
        }
    }
```