A tool for gathering unbiased sets of videos and metadata from YouTube

Kragset, Hans Petter Taugbøl Krieger, Lukas Wallenburg, Hugo Matthijs Harstad
{hpkragse, hugomw}@ifi.uio.no lukas.krieger@me.com

***Abstract*—**

I. Introduction

YouTube is without a doubt the world's largest host of user-generated content, with over one billion users generating several billion views, spending hundreds of billions of hours every day [1]. Though there does not seem to be an official source, it is believed that by the end of 2014, more than 300 hours worth of content was being uploaded to YouTube every minute [2] [3].

This makes YouTube a fantastic resource of both videos and metadata to be used for analysis for a range of different purposes. Especially with regards to machine learning algorithms, a large, good sample set of videos will be required to train an algorithm. For such an algorithm to stay relevant also for videos that was not in the inital training set, the training set will have to be as unbiased as possible.

It is neither our desire nor task to create a tool that by default limits the returned data set in any way. By letting the users specify as little or as much as they want in their query, we leave as much control as possible with the user, who, of course, knows much better than us what the resulting dataset will be used for. This makes the tool very versatile, as you could, for instance, first download a big set of videos related to the search term "cat", before downloading a completely random video set and using an algorithm trained with the first data set to find cat-related videos in this second set.

More specifically, we provide the means to: build a large database of video ID's [1]; fetch most metadata[2] for the given videos; fetch the videos themselves, with sound; and connect all related data points with a SQL database. Alongside the documentation for the source code there will be a databse diagram show how all the data is related. Making an SQL database made sense because, as we saw our dataset grow with hundreds of thousands of videos with related metadata and comment data, storing it in CSV, XML or some other file format made little sense. SQL is also a widely adapted database format, and all the widely adapted programming languages has support for extracting data from an SQL database in one way or another.

One of the main goals of the tool, as described earlier and in even more detail later, is to be unbiased. In the context of this paper, to be unbiased means to not whiegh videos differently based on their properties. When gathering a set of videos, the only limitations, if any, are the ones provided by the user. The resulting videoset will consist of all videos matching the query, without being wheighted towards popular videos, new videos, high quality videos, advertised videos, recommended videos or any other imaginable parameter. This is inherently different from the video set a normal user sees while browsing, and this is discussed later.

With this foundation our work has been centered around trying to gather as much information as possible from YouTube whilst being unbiased and efficient with resources. Resources can here mean several things, as well as the users CPU, network and storage resources, we also have tried minimising the API quota [3] costs for the user.

II. Related work

III. DASH

A. Overview

Dynamic Adaptive Streaming over HTTP (DASH) is a protocol, defined in ISO/IEC 23009-1 [4], that facilitates streaming of multimedia over the internet in varying network conditions. The goal of DASH is to improve the user experience while streaming media content by splitting the media file in question into smaller parts and letting the streaming client choose between several different versions of the same file depending on preferred quality and network conditions. In effect, it allows streaming clients the ability to aggressively limit stalling in video playback at the cost of perceived video/audio quality.

The core of DASH is the Media Presentation Description (MPD), sometimes referred to as the DASH manifest. The MPD contains all the information needed to display the media. When a user wants to access a streamed media, for instance a video, the streaming client (or website frontend) makes a request to get the MPD for the requested media. The client then parses the MPD and meassures available network and buffer resources, before the streaming is commenced at the best feasable quality. Many clients seem to prefer starting the stream at the lowest available quality, and slowly ramp the quality up if no network problems occur. The client continues to monitor the available resources during playback, adapting quality dynamically if conditions should change.

[1] We fetch as much data as possible within the restraints imposed on us by the API.

[2] We fetch all data associated with a video, as well as its comment threads and replies. Fetching of related videos has been deliberately left out - for now at least

[3] A form of cradit assosiated with an API key, quota is lost when making API requests.

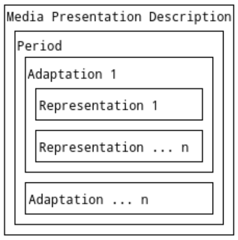


Fig. 1. Diagram of the YouTube Media Presentation Description.

With this approach all decisions about download speed are left to the client alone. The server simply presents the entirety of the media information in the MPD, allowing the client to make its own informed choices. The client may choose to focus on continuous playback, for instance by downloading lower quality segments during heavy network congestion or local processor load, or it may force playback of a set quality no matter the conditions under which it operates. The behavioral pattern for most media streaming clients, among others YouTube's own media player, seems to be to value continuous playback over intermittent delays in high quality playback, unless the user should want to force a specific quality setting thus overriding default behavior.

The developed tool only downloads the media files as specified by the user, and does not care about available resources or the user experience. The DASH manifest is still parsed, and all relevant data is stored in the database to provide an overview of the available medias and qualities, and their respective properties. Although we are not using DASH to accomplish better user experience, the MPD is still very useful as a single point to get information about a streamed media.

B. Details

1 depicts the basic outline of a Media Presentation Description. The uppermost layer of the YouTube MPD contains duration of the media as well as information about the format. Each MPD contains a set of adaptations, each describing a media type. An arbitrary YouTube video might have the following adaptations: audio/mp4, audio/webm, video/mp4, video/webm. There are also different adaptations for 3D videos, live media streams, etc. Each adaptation (media type) might have multiple representations each designating a different media quality.

Note that this differs from what one might consider the *standard ISO DASH*[4] implementation. The standard facilitates using periods to describe discrete segments within the media, with each period containing links to separate files containing separate quality versions of that same period. YouTube works differently: There is only one period, with several adaptations describing different media types, each with several representations describing different qualities of media. YouTube MPD representations contain a single file[5] containing the entire video in the quality specified in the representation. The media

```
{
    "@id": "137",
        "@codecs": "avc1.640028",
        "@width": "1920",
        "@height": "1080",
        "@startWithSAP": "1",
        "@maxPlayoutRate": "1",
        "@bandwidth": "4133205",
        "@frameRate": "24",
        "BaseURL": {
            "@yt:contentLength": "148765820",
            "#text": [OMITTED - URI goes here]
        },
        "SegmentBase": {
            "@indexRange": "711-1942",
            "@indexRangeExact": "true",
            "Initialization": {
                "@range": "0-710"
            }
        }
}
```

Fig. 2. Example of a mp4 1080p representation

player client does not have to do anything more advanced than simply request this file in order to play the designated media. If it is to utilize DASH, however, it needs to calculate which chunks of which file it needs to fetch in order to assure smooth playback.

2 is an example of a Representation of a 1080p mp4 video. The `@id` field specifies an identifier for this Representation, it's used to do what exactly? Each id is linked with a specific type of video. In the case of YouTube, `137` is defined to always be a 1920x1080 mp4 video, for example [5, line 303][6]

The `@codecs` field shall specify the codecs present with this Representation. The field should also include the profile and level information where applicable. For this video, the codec specifies a H.264/AVC video, High Profile, Level 40 (fix).

The `@width` and `@height` fields specifies the resolution of the video in pixels (not the ISO DASH definition, but always true for youtube videos?).

TODO: describe SAP

`@maxPlayoutRate` specifies the maximum playout rate as a multiple of the regular playout rate, in this example it is set to 1, which means that it's not supported on any level.

The `@bandwidth` field is a little more complicated than the other fields. If a Representation is continuously delivered at this bitrate (in a constant bitrate channel of `@bandwidth` bps), starting at SAP 1, a client can be assured of having enough data for continuous playout providing playout begins after `@minBufferTime * @bandwidth` bits have been received. If you consider the value to be bits per second in a channel with constant bitrate,

Not all identifiers are specified in the ISO DASH standard. YouTube provides some of its own, and these are prefixed with yt:. One example is the `@yt:contentLength` field. This

[4]The DASH standard is incedibly loose in order to allow different implementations and the freedom to make changes that benefit the specific media host the most, thus there is not really a *standard* DASH implementation.

[5]Or more specifically, a single URI

[6]The retrieval of MPD's does not seem to be officially supported in the YouTube API v3, thus there are no truly reliable sources. The reference is to a popular youtube downloader tool that seems to have figured out how the MPD is laid out as of 2015-11-01

specifies the size of the Representation in bytes. So the total download size of the Representation will match this value.

BaseURL contains the contentLength and a HTTP URL to be used as a base URL for the Representation.

When switching between different qualities, the base URL is used together with content length and stuff to start downloading at the correct loaction for the next Representation. *super pr0 description here*

IV. DASH

YouTube makes use of the DASH protocol to serve its videos.

Pros and cons of HTTP crap Can be used with normal HTTP servers? DASH makes it convenient to provide media content to users since it enables content delivery from standard HTTP servers to HTTP clients. HTTP provides reliable transfer of data, and enables caching of content by standard HTTP caches. Since DASH is using HTTP as a transport protocol it inherits many advanced features such as redirection, authentication, traversing of NATs/firewalls, and TLS. Media resources are referred to by using HTTP URLs, this provides a unique location for the resources, and a simple and well-tested (?) method of accessing the resources using HTTP GET and HTTP partial GET requests.

1) Links: http caching: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching?hl=en
RFC6381: https://tools.ietf.org/html/rfc6381
https://en.wikipedia.org/wiki/ISO_8601#Durations
https://tech.ebu.ch/docs/events/webinar043-mpeg-dash/presentations/ebu_mpeg-dash_webinar043.pdf
http://www.w3.org/2010/11/web-and-tv/papers/webtv2_submission_64.pdf

V. Architecture

A client-server model with a REST API was chosen as the tool architecture. The proposed solution allows the tool to be used by multiple users simultaneously, as well as enabling deployment to distributed systems with distinct roles, responsibilities, and hardware resources. The frontend is a simple thin client whose sole job is relaying commands to a server and showing the returned results. The server is tasked with obtaining and processing the available YouTube data, and otherwise interacting with the API. Local deployment is a viable option, given sufficient storage and networking capabilities.

A. Client

The user interface of the tool is written in AngularJS, an open-source framework that facilitates easy setup and management of single-page web applications using the model-view-controller (MVC) pattern. [?]

The frontend is separated into four main pages that each have a specific function: Management of API keys, creation of API queries, creation of celery tasks, and results presentation.

1) User management:

2) API key management: The API key management page lets the user register API keys to his account. API keys are instantly validated and, if valid, added to the dropdown list of available keys on the query builder page. While only a single key is required per API request (in the absence of OAuth 2.0), having access to multiple keys makes for quick and easy testing and the ability to generate keys for a specific purpose or dataset. The latter is shown in action on the Result page, where the user can view statistics filtered by a given API key.

3) Query Builder: On the Query builder page the user may build individual search queries using the provided interface. YouTube's API defines a set of options which we present to the user in the form of input boxes. Our algorithm for assuring unique results requires a timeframe, thus the two relevant query fields are required - all other fields are deemed optional. Query fields that specify operations specifically on the user's own videos are deliberately omitted as this is not within the scope of the tool.

Before queries are dispatched to the task workers and any real work is performed, they are validated by issuing a small version of the query for verification only. The user is immediately notified if any of the given query parameters create an invalid combination. This will prove invaluable to new users who can safely learn to use the YouTube API, as well as preventing the storage of invalid queries.

4) Task Page: Stored queries may be executed on the Task page. The user selects the task he wants performed and a query which defines the set of videos on which to operate. The first operation will always be the fetching of video IDs, as the rest of the available tasks depend on this. Multiple tasks may be launched in parallell. Progress bars are updated in real-time.

5) Result page: The Result page contains selected statistics for the dataset returned by a given query. Of particular note is the graph showing the intersection between datasets. This allows the user to quickly identify closely related queries, and can be helpful in e.g. parameter studies.

B. Server

The server is written in python

1) Background tasks: All requests created by the frontend are handled asynchronously by Celery and Redis.

Celery is used as an asynchronous task queue based on distributed messages. It is capable of distributing tasks over a potentially vast network of nodes. [?] Redis is a networked in-memory key-value database. [?] The frameworks aren't used to their full extent given our current single-server environment, but having these frameworks already present will be of great aid in future expansion.

After a task has been scheduled by the user, it is immediately pushed to Redis and put in a pending state until a Celery worker is available to process it. While the task is executing, Celery provides an interface with which to control it. It is because of this feature an in-memory database to keep track of running tasks is preferential. Given that our fetcher modules, the actual tasks that are being run, update their own status in the database after every single request to the YouTube API, a disk I/O-bound database would potentially severely limit performance.

VI. Fetching an unbiased set of video IDs

For video analysis, an unbiased set of sample videos might be a requirement or a desire for the analysist. YouTube, of course, is inherently relying on its ability to provide the users the videos they want, and in turn this leads to (very) unbiased samples if one was to simply add the videos one found while browsing YouTube to a list. To circumvent YouTubes attempts at tailoring a set of videos, the algorithm uses the API to get videos.

A. the YouTube API

YouTube provides a REST API, from now on referred to as "the API", which can be queried for information with a set of parameters. The API consists of different endpoints, and the one used for searching videos is called "search.list". This endpoint returns a result as a json object. This endpoint provides a rich list of customiseable paramaeters, and optimising the use of these is important to achieve high efficiency while using the API.

For a given query, he search.list response will contain at most 50 videos. There may be multiple pages, as indicated by a `nextPageToken` in the json object, but no more than ten pages are returned for any one set of parameters. To get a complete list of video ID's for a given set of parameters the chosen query has to be split up into subqueries that by themselves does not match more than 500 videos. This splitting, and how we recommend doing it, is discussed in detail in this section.

Why is it important to split up the query into subqueries? The main problem with the 500 video limit for a static parameter set is that it can not be a good sample. There might be a lot of other videos which match the requested parameter, but subsequent requests to the API only return more or less the exact same sample. There are minor variatons in the returned sample, and we will discuss this and the issues realted to it in more detail later.

As a side note it should be mentioned that using an actual web crawler will have properties an API reliant fetcher can not recreate. A crawler will have its behaviour logged by YouTube, which in turn provides more content in line with what it thinks the crawler likes. The resulting set of videos will be more like a set of videos a given user is likely to see, it is inherently biased. For some cases such a set of videos might be desirable, for instance if one wants to focus on popular videos, or if one wants to meassure how biased a set becomes over time.

Before continuing a few terms will have to be clarified. A "static parameter set" is the set of parameters that are static, globally, for a set of queries, like "All videos which are related to the word 'fun', that are 2D and have a high video quality". A "variable parameter" is a parameter that can be changed for every single request in a request chaing, while the static parameter set remains untouched. This results in the ability to create many different variations of the "static parameter set" in order to exceed the 500 videos maximum.

From the search.list API endpoint only four different variables can be varied between requests in a chain. All other parameters are in some kind static and would result in a maximum of 500 videos. Following is a description of the variable parameters.

1) location and locationRadius: The problem with this parameter is that not every video on YouTube has specified the location in the metadata. Evaluating some hundred thousands of video's metadata has shown that only 5-10% of the videos has specified a location in their metadata. We can not verify that this is the average on all videos uploaded on YouTube, but just the fact that some (a lot) of videos lack data in this field indicates that trying to vary it to get an unbiased sample will result in a set of videos that be default leaves out a lot of videos. For unbiasedness this is sub-optimal, but not disasterous. It is reasonable to assume that the set of videos with location data contains a well distributed and unbiased (sub-) set, but this can not easily be verified.

2) channelId: In order to use this parameter as a means of getting an unbiased set of videos, we would need to have a list of all channels available on YouTube. This is as difficult as getting all videos of YouTube and therefore varying this parameter is probably not a good approach to get an unbiased set of videos. Some channels also have some thousands videos uploaded, so with a static parameter set and only varying the channelId, the API response would be limited to 500 videos. One could argue that 500 videos is a representative set of videos for a channel, but the problem is that this subset of the channels full video list is provided by YouTube, outside our control. Thus the returned sample can not safely be assumed to be representative.

3) q - search term: In theory this parameter could be varied and cycled through all possible combinations of symbols (words, in a wide sense), and through that get an unbiased set of videos. This is not only unfeasable due to the remarkably big list of words one would need[7], but for some cases the API quota cost would be extremely high meassured as cost per result. For instance, if one was to get an unbiased set of 3D videos, one would have to cycle through the whole word list and end up spending quota on queries that returns without new videos. In addition to this, if one is to vary the search term, the user is left without the ability to narrow down the search with a query. For some purposes, like machine learning algorithms, having a set of videos with cats will be useful, if you want to train the algorithm to look for cats.

4) publishedAfter and publishedBefore: By having a date range as the varying parameter for a serach query, and assuming that the API actually returns all videos matching the query, the resulting set would be unbiased with regards to all aspects except from time. Of course, for a big timeframe there exists more than 500 videos, so to circumvent this cap, the timeframe can be recursively sliced up until it is so small that less than 500 videos match the parameters. Varying this way leaves the rest of the parameters unchanged, and the resulting set becomes as much as possible unbiased.

By slicing the timeframe up like this, one will also be able to get more than 500 videos from a given channel, or from within a geographic zone. It becomes apparent that varying

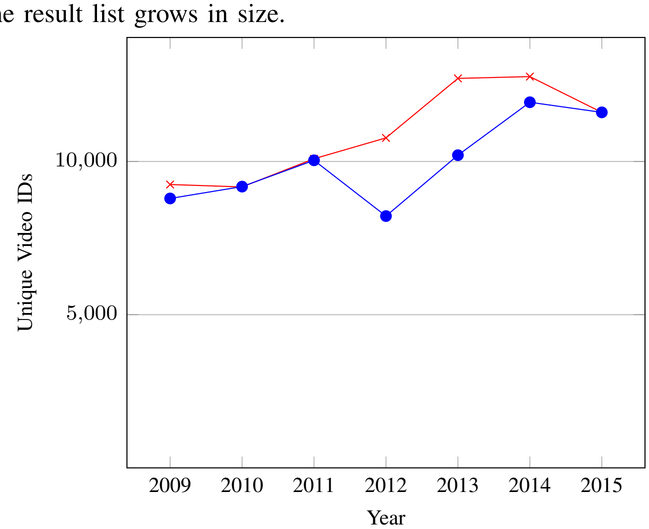
[7] 171 000 words in english alone, then add all other languages, not to mention names and other word-constructs

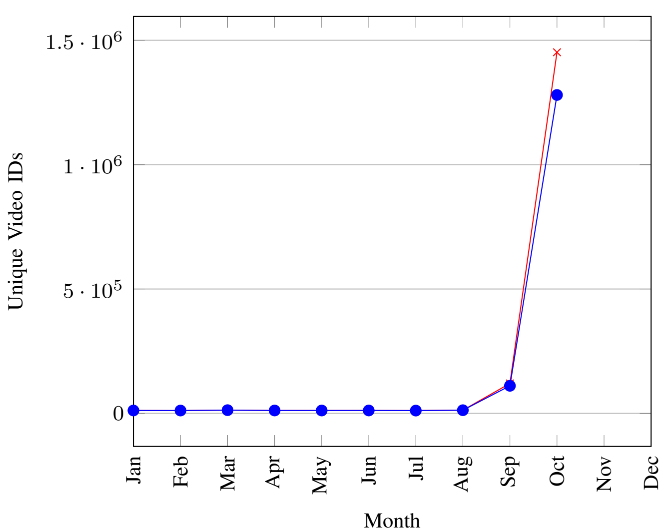
this parameter alone is the best way to achieve a unbiased set of videos. A nice bonus is that it also allows customisation of the query without leaving videos out of the resulting set.

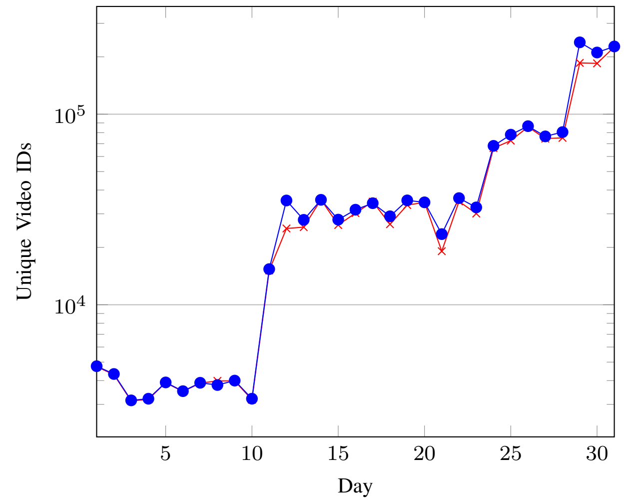
B. API Issues

We have experienced that the API in some situations returns and indicates that there are multiple pages, even if there are less then 50 videos in the result list. This is in itself illogical, as a more natural way to return results would be to fill each page, and only when there are less than 50 videos remaining that match the query return a non-full list. Our experience why querying for the next page when the list is not full, is that the page and subsequent pages contains the exactly same videos. Thus iterating over the pages a single static parameter set has becomes a bad idea because it is impossible to know if there are new videos on the next page or not.

We have also found that when issuing the same request to the server several times, the API might respond slightly differently each time. This result is easily reproducible by selecting a standard query, and seems to be more frequent as the result list grows in size.







VII. Future work

Perhaps the most obvious issue to address in future work is the unrealiability of the YouTube API. Doing research on what causes inconsistent replies could initially improve the effectivity of the developed tool, but perhaps more importantly it could be used to improve the YouTube API.

Investigating why the API return rate drops dramatically at certain intervals when looking back in time, and looking at what kind of videos that disappear from the returned set, will give interesting knowledg about the YouTube API, as well as perhaps give a final answer to wether the returned video set is biased or not.

There are many conceivable additions to the tool as it is now. Improving the statistics view could improve the user experience and make it easier for the user to make decisions about what to do next. A feature that allows the user to export a database containing only a selected set of videos could drastically reduce SQL query times during analysis, and ww would recommend making some feature like this if SQL querying becomes a major time and resource consumer.

VIII. Conclusion

This paper presented the YouTube implementation of DASH. The paper presented how YouTube distributes its media content through the DASH protocol, and how it differs from the ISO specification of DASH. It gave an overview of the YouTube Data API v3, outlined the major limitations of it and also provided some guidelines on how to bypass these limitations in order to obtain a good sample of YouTube videos for further analysis. The main decision criteria on implementing the searching for videos were maximization of the amount of responded videos while minimizing the quota costs and requests needed to achieve this, while at the same time trying to maximize resource usage through concurrency. While creating the tool, there were 2305 open issues associated to the YouTube API on the Google Data APIs issue tracking platform [6] and some of them were related to the behaviors we have experienced.

REFERENCES

[1] YouTube, *Statistics* [Online]. Available: https://www.youtube.com/yt/press/statistics.html [Accessed: 2015-11-01]

[2] K. Djerf. (2014-11-12). *300 timmar film laddas upp på Youtube i minuten* [Online]. Available: http://www.dagensmedia.se/nyheter/article3863831.ece [Accessed: 2015-11-01]

[3] M.R. Robertson. (2014-11-21). *300+ HOURS OF VIDEO UPLOADED TO YOUTUBE EVERY MINUTE* [Online]. Available: http://www.reelseo.com/youtube-300-hours/ [Accessed: 2015-11-01]

[4] *Information Technology - Dynamic adaptive streaming over HTTP (DASH)*, ISO/IEC 23009-1, 2014

[5] P. Hagemeister et al. *youtube.py* [Online]. Available: https://github.com/rg3/youtube-dl/blob/master/youtube_dl/extractor/youtube.py [Accessed: 2015-11-01, git commit 5c43afd]

[6] Google Code server-side issues and feature requests for YouTube Data API v3 [Online]. Available: https://code.google.com/p/gdata-issues/issues/list?q=label:API-YouTube [Accessed 2015-11-01]

[7] M. Ward and S. Robinson. (2013-05-18). *Google I/O 2013 - Adaptive Streaming for You and YouTube* [Online]. Available: https://www.youtube.com/watch?v=UklDSMG9ffU [Accessed: 2015-11-02]