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 over 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. Our tool aims to aid a group of users who may not be looking for a particular subset of data, but rather a large, unbiased sample. Our approach is to let the user create a search query for a specific timeframe, optionally limiting results further with additional fields, and querying the YouTube API for every available video within that timeframe. We deem this to be the easiest way to create an unbiased (or at the very least nearly so) set of videos using YouTube's API, seeing as YouTube will not let us create a truly random sample set[3].

We provide the means to: build a large database of video ID's[1] within a set timeframe; fetch most metadata[2] for the given videos; fetch the videos themselves, with a set quality; and connect related data points with a SQLite database.

II. Related work

III. DASH

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.

[3]The API seems to have some severe undocumented limitations regarding what videos it will return for a given query.

[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

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.

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.

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

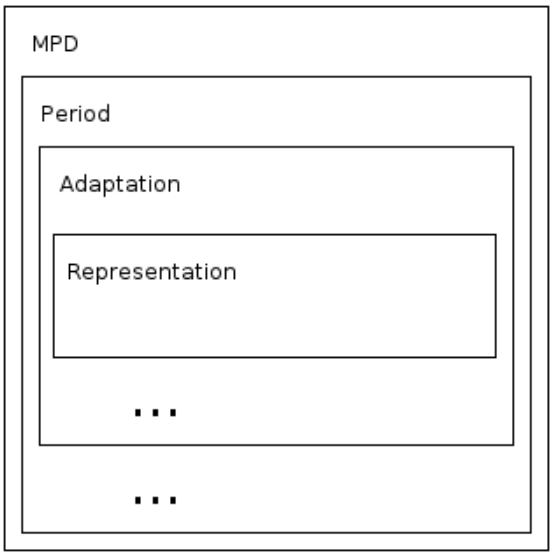


Fig. 1. YouTube DASH MPD

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.

One of the core features of DASH is the ability to request different qualities for each video. For many YouTube videos you can choose from a range of qualities between 122p and 1080p. Recently, YouTube also added 4k videos.

This 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. 137 is always a 1920x1080 mp4 video, for example [https://github.com/rg3/youtube-dl/blob/master/youtube_dl/extractor/youtube.py].

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 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*


```
{
    "@id": "137",
        "@codecs": "avc1.640028",
        "@width": "1920",
        "@height": "1080",
        "@startWithSAP": "1",
        "@maxPlayoutRate": "1",
        "@bandwidth": "4133205",
        "@frameRate": "24",
        "BaseURL": {
            "@yt:contentLength": "148765820",
            "#text": "http://r8---sn-uxaxovg-vnad.
        },
        "SegmentBase": {
            "@indexRange": "711-1942",
            "@indexRangeExact": "true",
            "Initialization": {
                "@range": "0-710"
            }
        }
}
```


1) Links: http caching: https://developers.google.com/web/fundamentals 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.

For the ID-fetching algorithm we utilize the search.list API endpoint, which returns a collection of search results for a given set of parameters. The API provides a rich set of parameters to tune, but there are some inherent limitations that have to be taken into consideration when designing an unbiased algorithm.

The search.list response will contain at most 50 videos. There may be multiple pages, as indicated by a `nextPageToken`, but no more than ten pages are returned for any one set of parameters. We want to access the complete list of video ID's for a given set of parameters, thus the chosen algorithm has to split the user input into multiple subqueries that by themselves do not match more than 500 videos.

The search.list response contains a maximum of 50 videos per page and a total of 10 pages. This restrictions leads to the fact that for a single static parameter set there is a maximum of 500 video provided by the API. To access the total 500 videos, the 10 pages must be iterated, each iteration utilizing a new request which results in a total of 10 requests and 1000 quota unit costs.

The main problem with the 500 video limit for a static parameter set is, that it an not be a good sample. There might be a lot of other videos which match the requested parameter, but YouTube restricts his response to the same 500 videos, even if the same request as been performed multiple times. Indeed there is some small variation of each request's response measurable, but only a few videos change and therefore it can be neglected.

Our approach of receiving an unbiased sample of videos from the API was to have a static parameter set with some variable parameter which allows us in theory to et all videos from YouTube. The user then has the possibility to select a subsample of all the videos available on the YouTube API for further research. After all videos are fetched, the user has a lot of metadata

For the next section we have to define some words. A "static parameter set" is meant to be a global search request, like "All videos which are related to the word 'fun', are 2D and have a high video quality". A "variable parameter" is a parameter which can be changed for every single request, without changing the meaning of the "static parameter set", but gives us the ability to create many different variations of the "static parameter set" in order to exceed the 500 videos maximum.

There are only four real variable parameters in the search.list API endpoint available. All other parameters are in some kind static and would result to a maximum of 500 videos.

channelid — indicates that the API response should only contain resources created by the channel location+locationRadius — defines a circular geographic area and restricts the search to videos that specify a geographic location in their metadata which falls within the

requested area q — specifies a query term to search for publishedAfter/publishedBefore

2) 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 5-10We can not verify that this is the average on all videos uploaded on YouTube, but the existens of non-location related videos indicates that this parameter is not a good choice to receive an unbiased sample of videos.

3) channelId: In order to use this parameter, we would need to have a list of all channels available on YouTube. This is as much difficult as getting all videos of YouTube and therefore this parameter does not suit our needs. Some channels also have some thousands videos uploaded and with a static parameter set and only the variable channelId, the API response would have a maximum of 500 videos for these channels.

4) q - search term: Using this parameter as variable would lead to not having this parameter in a static set and therefore it becomes impossible to receive all videos for a single search term. We wanted to have this parameter in a static way in order to provide results for search queries like "All videos which are related to the search team 'fun'" Although if you try to iterate over this parameter, a huge list of keywords would be needed and it does not scale well since the iteration has to be done for every single API search. A search like "All videos which are 3D" with a keyword list containing 100.000 words, would result to perform 100.000 requests and costs 1 million units whereas there are only a few 3D videos on YouTube.

5) publishedAfter and publishedBefore: Those parameters suit most to receive an unbiased sample of YouTube videos. They are completely independent from the other parameters and therefore they allow us to modify them without affecting the other parameters and the result of the request.

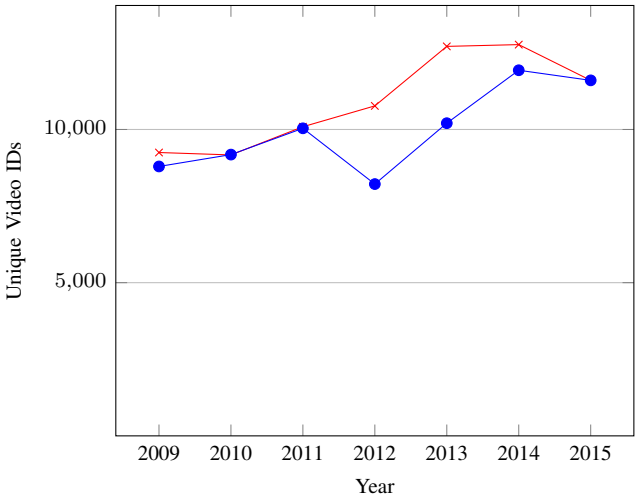
In our tool, we provide the full list of all search.list parameters which can optional be set in a static way.

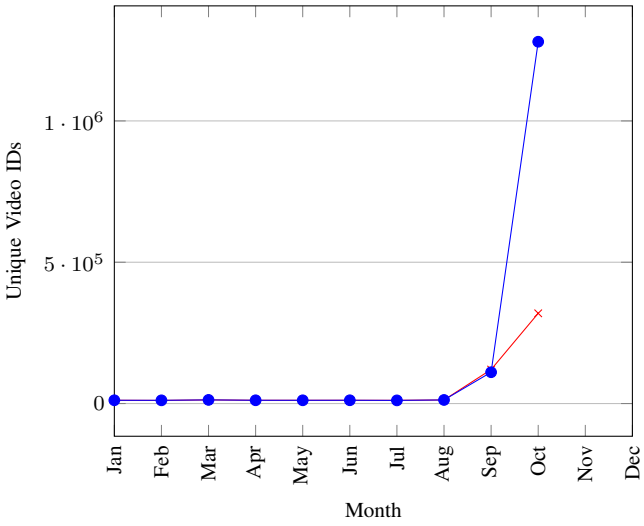
After identifying publishedBefore/After as the most suitable variable parameters for our intend and to exceed the 500 video maximum limit for a "static parameter set", we had to think about how to utilize them in an algorithm.

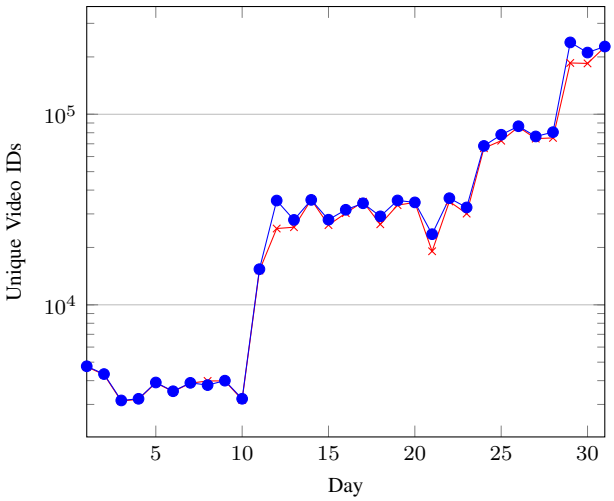
The first thought was to let the user define a "static parameter set" and a period of time in which the search should be performed. The fir

We have experienced that the API in some situations tells there are multiple pages even if there are less then 50 videos and each page contains the exactly same videos. So iterating over the pages a single static parameter set has, became a bad idea because we did not know if there are indeed new videos on the next page or Google just wants us to reduce the total results we get and maximize the quota unit costs spent.

The IDFetch and other YouTube API related algorithm are utilizing HTTP persistent connections, which allows to use a single TCP connection to send and receive multiple HTTP requests and responses.







VI. Future work

VII. Conclusion

This article presented the YouTube implementation of DASH compared to ISO/IEC 23009-1. 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 analyzes. The main decision criteria on implementing the searching for videos were concurrency and maximization

of the amount of responded videos while minimizing the quota costs and requests needed to achieve this. While creating the tool, there were 2305 open issues associated to the YouTube API on the Google Data APIs issue tracking platform [5] and some of them were related to the behaviors we have experienced.

REFERENCES

[1] YouTube official stats for the press https://www.youtube.com/yt/press/statistics.html

[2] 300 timmar film laddas upp på Youtube i minuten http://www.dagensmedia.se/nyheter/article3863831.ece

[3] 300+ HOURS OF VIDEO UPLOADED TO YOUTUBE EVERY MINUTE http://www.reelseo.com/youtube-300-hours/

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

[5] Google Code server-side issues and feature requests for YouTube Data API v3 https://code.google.com/p/gdata-issues/issues/list?q=label:API-YouTube