A Tool for Obtaining Sets of Video- and Metadata via Dynamic Adaptive Streaming Over HTTP and the YouTube Data API v3

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

***Abstract*—YouTube provides a massive amount of publicly available audiovisual content. Analyzing this data can provide invaluable information, and YouTube is especially interesting as there is a similarly massive amount of related metadata available. Analyzing video together with metadata such as recording time and location, view count, like and dislike counts, description, tags and last but not least comments enables a whole new depth of video analysis. With this in mind we have developed a tool that makes it easy to download a set of videos with related metadata and comments. The tool groups the data in an SQL database. When downloading data our tool tries to be as unbiased as possible, by downloading as much data as it can and leaving the query parameters in the users control. Being unbiased is important because having a representative video set to, for instance, train a machine learning video analysis algorithm is essential for that algorithm's ability to understand a video outside of the training set. During development we discovered interesting aspects about the YouTube API. The API seems to be inconsistent in the replies it gives, the API endpoint gives slightly different results when queried with the same query multiple times. This behavior is not unexpected - and negligible. It does not compare to the issue that is the massive decline in returned results when stepping backwards in time. Compared to requesting videos from yesterday, requesting videos from a day a months back return only about 2% of the unique video ID count.**

YouTube uses Dynamic Adaptive Streaming over HTTP (DASH) to provide its media content. Our tool uses this to provide the user with information about available qualities for a given video or video query set, and subsequently issues HTTP GET requests as described in the DASH Media Presentation Description (MPD). YouTube's implementation of DASH does not differ much from ISO 23009-1 [2] where DASH is defined.

***Index Terms*—MPEG-DASH, YouTube, Metadata, unbiased dataset**

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 [15] [16].

This makes YouTube a fantastic resource of both videos and metadata intended for uses such as e.g. statistical analysis and machine learning: Fields like these require considerable amounts of data to be expected to yield reasonable results. Of particular note is the relation between the videos and the corresponding metadata, as it may provide the means to interpret the media data in its true context.

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 has a more intimate knowledge about the dataset he wants to obtain. This makes the tool versatile, as you could 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 IDs [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 database diagram to show how all the data is related.

As we tested our tool and our working data set grew up to hundreds of thousands, if not millions of entries, we realized we could not simply store the resulting datasets in a straightforward manner. We needed a system that could handle the potentially massive I/O spikes our tool would create. We went for a relational database solution that can both handle the expected stress and facilitate connecting related data. SQL is a widely adapted database format, supported by plugins (both official and unofficial) in almost every programming language there is, and it can also handle the incredible stress we will subject it to. Additionally, it makes it easy for even a novice user to extract meaningful data from the database.

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 wheigh 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[3]. This is inherently different from the video set a normal user sees while browsing.

With this foundation our work has been centered around

[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] Assuming the query responses themselves are unbiased

trying to gather as much information as possible from YouTube whilst being both unbiased and efficient resource-wise: We attempt to minimize CPU load, network usage, storage required, while also minimizing the API quota[4] costs for the user.

II. Related Work

A lot of work has by now been done related to DASH and Adaptive streaming in general. DASH [2] was the first adaptive streaming technology using HTTP that became an international standard. This is in large due to pioneering work from Adobe with their Adobe Systems HTTP Dynamic Streaming [7], Apple with HTTP Live Streaming (HLS) [8], and Microsoft with Microsoft Smooth Streaming [9], DASH has quickly spread, and YouTube has now implemented DASH as their preferred streaming technology [6]. D. Krishnappa et al. (2013) [10] discussed possible resource savings of implementing DASH in YouTube, somewhat related to our work.

youtube-dl (sic) [4] is an open source tool developed by P. Hagemeister et al. (2008-today) that provides the ability to download videos from YouTube and other similar media hosting sites. Their source code has been the only significant inspiration for our own source code. We had the option of simply including their source code and interact with it directly, as a way to outsource the video file downloading, but the code required is not complex enough to justify the loss of control. One of the most usable features youtube-dl provides, that our tool currently does not provide, is the merging of sound and video into a single file. This would make storage on our part a bit less complicated, but on the other hand having split video and sound files can be an advantage for analysis tools.

III. DASH

A. Overview

Dynamic Adaptive Streaming over HTTP (DASH) is a protocol, defined in ISO/IEC 23009-1 [2], that facilitates streaming of multimedia over the internet in varying network conditions. The goal of DASH and other segment-based streaming service structures is to improve the user experience [6] 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 while simultaneously saving the service provider a significant amount of network traffic [10, p. 412] [6].

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 measures available network and buffer resources, before the streaming is commenced at the best feasible 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.

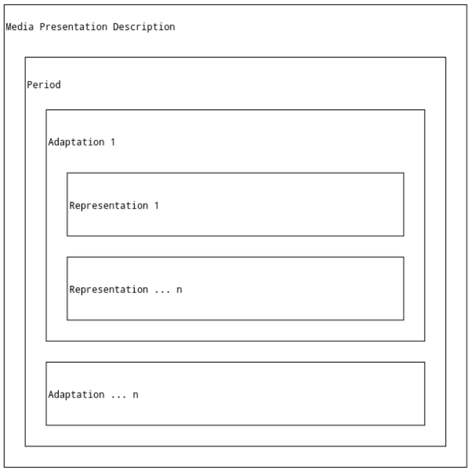

Fig. 1: Diagram of the YouTube Media Presentation Description.

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 [6], 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, 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. Media Presentation Description

Fig. 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)

[4] A form of credit assosiated with an API key, quota is spent making API requests.

```
{
    "@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 an mp4 1080p (*full HD*) representation

might have multiple representations each designating a different media quality.

Note that this differs from what one might consider the *standard ISO DASH*[5] 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[6] containing the entire video in the quality specified in the representation. The media 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.

C. Representation details

1) `@id`: Specifies a unique (within the Period) identifier for this Representation, used for easy lookup and the potential for (reckless!) hard-coding of identifiers instead of checking for video parameters as strings. Each id is linked with a specific type of video. In the case of YouTube's version of MPDs, 137 is defined to always be a 1920x1080 mp4 video, for example [3, line 303][7]

2) `@codecs`: Specifies the codecs present with this Representation. The field should also include the profile and level information where applicable. For this video, Fig. 2, the codec specifies a H.264/AVC video, High Profile, Level 40 [11, p. 12].

3) `@width` and `@height`: Specifies the resolution of the video in pixels.

4) `@startWithSAP`: The SAP, or Stream Access Point, describes a position in a Representation enabling playback of a media stream to be started using only the information contained in Representation data starting form that position onwards [2, p. 11]. The usage of this type of parameter is not specified exactly in the standard, but its general use is described. In other words, using this attribute and only the attributes in the current Representation, the player must be able to play the media segment described by the current Period.

5) `@maxPlayoutRate`: Specifies the maximum playout rate as a multiple of the regular playout rate. In Fig. 2 it is set to 1: It is not supported.

6) `@bandwidth`: A little more complicated than the other fields. If a Representation is continuously delivered at this bitrate in a constant bitrate channel of `@bandwidth` bits per second starting at SAP 1, a client can be assured of having enough data for continuous playback providing playout begins after `@minBufferTime * @bandwidth` bits have been received. In short: If the client is able to maintain a download rate consistently equal to or greater than `@bandwidth` during the entire playback, the user will never experience stalling.

7) Other: Not all identifiers are specified in the ISO DASH standard. YouTube provides some of its own, and these are prefixed with *yt:*. The `@yt:contentLength` field, for example, specifies the size of the Representation in bytes.

When switching from Presentation X to Presentation Y at segment n, the `@baseURL/#text` (URI describing the media resource of Presentation Y) is used in conjunction with `@bandwidth` of Presentation Y to calculate the chunk offset of segment $n+1$ into the media file.

IV. Architecture of the Tool

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

A. Client

The user interface of the tool is written in AngularJS[8], 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, each serving a specific function: Management of API keys, creation of search queries, creation of tasks, and results presentation.

[5] The DASH standard is incredibly 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.

[6] Or more specifically, a single URI

[7] 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

[8] https://angularjs.org/

1) User management: Due to the potential usage by multiple users, a rough user management is the foundation to handle concurrent sessions and a login is required to access the pages described as in the following. All the data in the database can be grouped by a single user.

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 one key is required per API request (in the absence of OAuth 2.0), having access to multiple extends the tool's features to generate keys for a specific purpose or dataset.

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 complete query for syntax 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 used to execute tasks on the Task page. The user selects the task he or she wants performed and a query which defines the set of videos on which to operate. The first task should always be to fetch video IDs for the given query, as the rest of the available tasks depends on the video ID. Multiple tasks may be launched in parallel, and their progress is tracked 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 table showing the intersection between datasets. This allows the user to quickly identify closely related queries, and can be helpful in e.g. parameter studies. The result page can show different statistics, and the different parts are loaded asynchronously on demand. It is relatively easy to expand this page by adding more queries or statistics.

B. Server

The server is written in python, using flask[9] as the server framework. Flask integrates with Jinja[10] for templating, and has support for a range of third party plugins to handle everything from encryption to database integration. The communication between the server and the client is established by a REST API.

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

Celery[11] 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[12] is a networked in-memory key-value database. 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 to update the current task's state. It is because of this feature an in-memory database to keep track of running tasks preferential. Given that our fetcher modules, the actual tasks that are being run, update their own status after every single request to the YouTube API, a disk I/O-bound database would potentially severely limit performance.

The frameworks are not used to their full extent in our current single-server environment, but having these frameworks already present will be of great aid in possible future expansion.

C. Selecting a representative sample of videos from YouTube

Before we present our implementation we want to give a brief summary of different kinds of strategies to receive data from YouTube since there might be some limitations and choosing a strategy depends on various individual decision criteria.

1) Crawling the YouTube website: An unbiased set of videos and metadata may be a desire of, or indeed a requirement for, a statistical analyst. YouTube's recommendation system is tailored to provide the most relevant content to one user during a single session, or over the course of multiple sessions. Up to 20 recommended videos are shown per video page. Among these are paid promotions by other users or networks, top trending videos, and videos directly related to the last submitted search query, (probably) amongst others. If one was to naively crawl through content served to them by the YouTube recommendation system one would certainly not venture far from the starting point due to an inherent confirmation bias in algorithms typically chosen for this purpose [12]: YouTube will continue to serve videos related to the ones one is already watching, thus it is difficult to break out of even a certain category of content. One might think of this as actively hindering us in our hunt for an unbiased representative dataset.

2) Random generation of video IDs: We considered for a moment attempting to create a brute force solution to create truly unique and random video IDs and trying as many of these as possible, but there is a tiny problem with that approach: Assume that YouTube uses a modified Base64 encoding for URLs with 11 digits for their video IDs. A ballpark (insane over-)estimation for the total amount of videos on YouTube is 300 hours per minute since Google's acquisition in 2006 (9 years). If we estimate the average video to be five minutes long, that leaves us with

$$\frac{85\,140\,000\,000}{5} = 17\,028\,000\,000$$

[9] http://flask.pocoo.org/

[10] http://jinja.pocoo.org/docs/dev/templates/

[11] http://www.celeryproject.org/

[12] http://www.redis.io

videos. With 64 to the power of 11 possible video IDs we would have to generate more than four billion random IDs before we could expect to get a single hit. Not feasable.

3) The YouTube API: YouTube provides a REST[13] Application Programming Interface (API) [**?**]youtube-api) which can be queried for information with a set of parameters, in absence of a standard website interface to utilize (like the one provided by `www.youtube.com`). 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. It accepts a rich list of customizable parameters as input allowing users to create highly specific and optimized queries to achieve the best possible efficiency for the specific use case.

For a given query, the 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 IDs 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.

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 variations in the returned sample, and we will discuss this and the related issues 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 measure 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 chain, 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.

4) 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 videos 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 disastrous. It is reasonable to assume that the set of videos with location data contains a well distributed and unbiased subset, but this can not easily be verified.

5) 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 of 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 channel’s full video list is provided by YouTube, outside our control. Thus the returned sample can not safely be assumed to be representative.

6) 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 unfeasible due to the remarkably big list of words one would need[14], but for some cases the API quota cost would be extremely high measured 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 points on queries that return stale results. In addition to this, if one is to vary the search term, the user is left without the ability to manually narrow the result set. We want to allow the user the ability to create sets of data tailored to specific tasks, should it be required.

7) publishedAfter and publishedBefore: By having a date range as the varying parameter for a search 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 this parameter alone is the best way to achieve a unbiased set of videos. A nice bonus is that it also allows customization of the query without leaving videos out of the resulting set.

[13] Representational State Transfer:
http://www.jopera.org/docs/publications/2008/restws

[14] 171 000 words in english alone, then add all other languages, not to mention names and other word-constructs

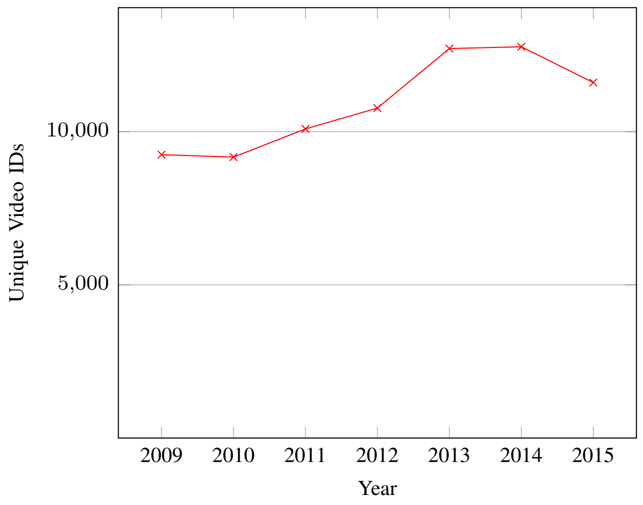

Fig. 3: Unique video IDs for January, by year [Accessed: 2015-10-31]

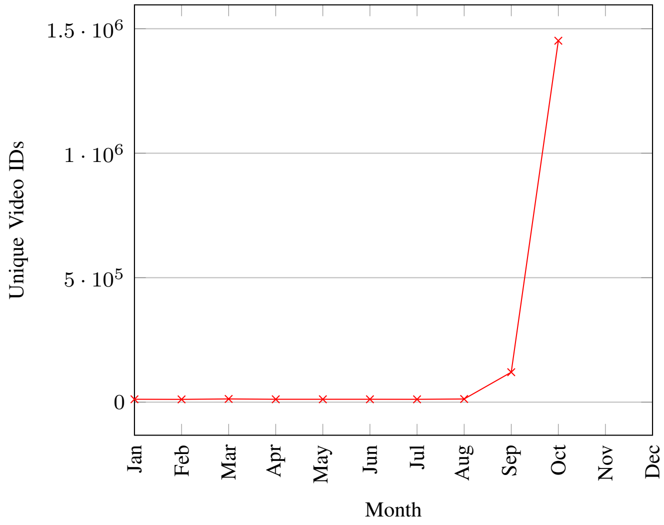

Fig. 4: Unique video IDs for 2015, by month [Accessed: 2015-10-31]

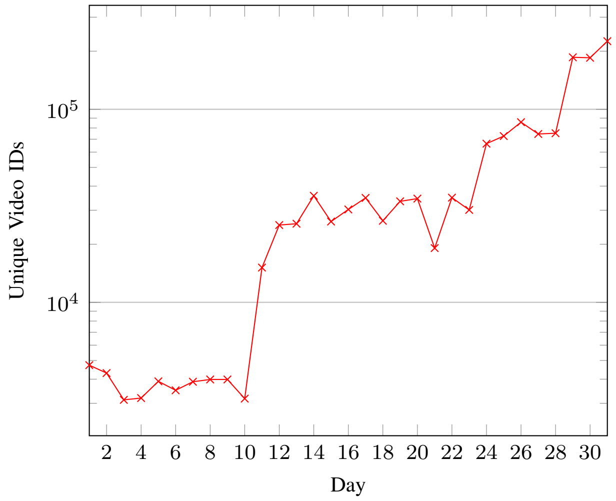

Fig. 5: Unique video IDs for October 2015 [Accessed: 2015-10-31]

V. API Issues

We have experienced that the API in some situations returns and indicates that there are multiple pages, even if there are less than 50 videos in the result list. The returned result also contains a field representing the number of items returned, but this field often does not match the actual amount of items in the result [5, Issue 5173]. Returning results as lists with open spots while at the same time indicating that more results is on the next page 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. This leads to flaws or imperfections in the logic in the clients making queries to the API, as one can not trust either the total count field, the presense of a next page token or the actual number of returned results.

Our experience when querying for the next page when the list is not full, shows that the page and all subsequent pages contain the exact same set of videos. This is a known issue in the API [5, Issue 6406]. In combination with the aforementioned issues this leads us to the conclusion that iterating over the pages a for a single static parameter set 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 and also a know issue with the API [5, Issue 4275].

The unfortunate consequences of these issues include excess API quota spending, excess time consumption whilst getting results from the API, increased network and local resource consumption and uncertainty with regards to the integrity of the results.

A. *Decline in unique results*

As a part of our experiment we issued a lot of requests to the YouTube API to gather data. A strange pattern quickly emerged, requests for timeslots longer back than a few days contained drastically less returned unique videos, and stepping a month back, a request for all videos uploaded a given day will be about 98% less than a request for videos uploaded yesterday. This decline in returned video IDs seem to go in steps, as visualised in Fig. 5. For the past three days the returned unique video IDs are around 200 000. The next five days back return around 65 000 to 75 000 video IDs each, and the subsequent 12-14 days return 20 000 to 30 000 videos each. From that point and backwards the number of videos per day remain stable at around 2 000 to 4 000, slowly declining to only a few hundred returned videos per day. Fig. 3 shows the return when asking for videos published in January from 2009 to 2015.

At September 26th 2015 the API returned a total 246 106 video IDs for that day. For the same day, if queried a little more than a month later, the API returns a mere 4 394 videos. Fig. 4 shows how after a month has passed the amount of videos dramatically decline. The missing videos are seemingly random, everything from strange videos with very few views to relatively popular videos (20 000+ views) is missing in the data set when requeried a month later. The remaining data set also seems random, containing videos with view counts from ranging in number from single digits to tens of millions. Perhaps with more research the nature of this video ID purging can be understood.

1) Accounting for API inconsistances: Contrary to natural intuition, the API may return a wholly different set of videos for two queries one might have thought to return the same as a single *superquery*. By example: If you search for ”All videos between date X and date Y you might get 1 000 results. But if you split the query into two queries: One specifically for 2D videos and one specifically for 3D videos; you might get a subset of the results for the single query, but in addition, the API might return a much larger set than before.

By splitting the set timeframe in two, creating two subqueries in the process, one can ”trick” the API into returning more videos for a given timeframe than the original query would have yielded. We have expanded our possible result space simply by using unused parameters to create distinct subqueries with increasing granularity. Our tool will do this through narrowing the timeframe of the query (and in the process providing a section on this on the result page), but it would be fairly trivial to implement an algorithm that uses all possible unused parameters and some clever pruning to increase the possible result space.

VI. Future work

Perhaps the most obvious issue to address in future work is the inconsistancy 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 knowledge about the YouTube API, as well as perhaps give a final answer to whether the returned video set is indeed biased.

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 we would recommend making some feature like this if SQL querying becomes a major time and resource consumer.

There is a lot of room for performance improvements, especially when it comes to concurrency. The tool was tested on a variety of platforms, and performed reasonably well on most of them, but on certain configurations the API requests were inexplicably horrendously slow. The cause of this might be a weird combination of hardware, drivers, and TCP implementation in the OS kernel - we simply had no time to narrow down the possible causes. Adding support to run multiple Celery jobs per task might have alleviated some of the performance hangups.

VII. Conclusion

This paper presented how YouTube distributes it’s media content through dynamic streaming over HTTP, 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 [5] and some of them were related to the behaviors we have experienced.

Appendix

Contributions to various parts of the project; names listed in order of approximate contribution weight

Paper contributions			
Abstract	Kragset		
Introduction	Kragset	Wallenburg	
Related work	Kragset		
Dash	Wallenburg		
Architecture	Krieger	Wallenburg	
ID-fetch-algorithm	Krieger	Kragset	Wallenburg
Figures	Kragset	Wallenburg	
API-issues	Kragset	Wallenburg	
Future work	Kragset	Wallenburg	
Conclusion	Krieger	Kragset	
References	Wallenburg		
LaTeX setup	Wallenburg		
Tool contributions			
Fetchers			
Comment Fetcher	Wallenburg		
ID Fetcher	Krieger		
Metadata Fetcher	Krieger	Wallenburg	
MPD Fetcher	Kragset	Krieger	
Video Fetcher	Kragset	Krieger	
Fetcher Base	Krieger		
Frameworks			
Database setup	Krieger	Kragset	Wallenburg
Celery setup	Krieger		
Redis setup	Krieger		
Frontend	Krieger		
Client/Server arch.	Krieger		
Miscellaneous			
Code cleanup	Wallenburg	Kragset	Krieger
Database diagram	Kragset		

Acknowledgements

We would like to thank Christian Fredrik Fossum Ressell for his contributions and moral support throughout the project.

References

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

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

[3] 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].

[4] P. Hagemeister et al. *youtube-dl* [Online]. Available: https://github.com/rg3/youtube-dl [Accessed: 2015-11-01].

[5] *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].

[6] 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].

[7] *HTTP Dynamic Streaming* [Online]. Available: http://www.adobe.com/products/hds-dynamic-streaming.html [Accessed: 2015-11-02].

[8] *Apple HTTP Live Streaming* [Online]. Available: https://developer.apple.com/streaming/ [Accessed: 2015-11-02].

[9] *Microsoft Smooth Streaming* [Online]. Available: http://www.iis.net/downloads/microsoft/smooth-streaming [Accessed: 2015-11-02].

[10] D.K. Krishnappa et al. *DASHing YouTube: An analysis of using DASH in YouTube video service* [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6761273 [Accessed: 2015-11-02].

[11] R. Gellens et al. *The 'Codecs' and 'Profiles' Parameters for "Bucket" Media Types* [Online]. Available: https://tools.ietf.org/html/rfc6381 [Accessed: 2015-11-02].

[12] J. Davidson et al. *The YouTube Video Recommendation System* [Online]. Available: http://www.inf.unibz.it/ ricci/ISR/papers/p293-davidson.pdf [Accessed: 2015-11-02].

[13] T. Stockhammer, "Dynamic Adaptive Streaming over HTTP Standards and Design Principles", in International Multimedia Conference, San Jose, CA, 2011.

[14] *API Reference* [Online]. Available: https://developers.google.com/youtube/v3/docs [Accessed: 2015-11-02].

[15] 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].

[16] 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].