Documentation

This project aims to provide the design for regular loading of app metadata from the app stores. Here we will discuss official app stores of iOS and Android, which inarguably capture most of the smartphone market. The document will cover topics like data downloading techniques from the app stores, pipeline to Redshift and the ways to optimize the costs.

Data Download Techniques

Both the app stores(itunes, play store) provide the *robots.txt* at:-

```
https://itunes.apple.com/robots.txt
https://play.google.com/robots.txt
```

These *robot.text* does not disallow a user to crawl the app pages. They also provide site maps which could be parsed to get ids of the apps on the app stores. We will use them to build our solution next.

Apple itunes Store

Apple itunes provides two different sets of API for looking up app metadata.

1. **Search API:** itunes provide free to use search api[1], where

one can search the matching apps with the search terms. For scrapping, such result will be highly difficult to work with. It is better to use lookup api which returns metadata given the app itunes id. This id can be obtained from the site maps retrieved earlier. The api is called with HTTP GET request and returns data in JSON format. The API has very aggressive rate limit of *20 calls per minute*.

2. **Enterprise Feed:** itunes provide a paid solution where a dump of all the resources on itunes(apps, songs, movies etc. metadata) is given in four big files. The files are in text format with *ASCII character-1* as the delimiter.[2]

Google play Store

Google does not provide any API to query App Metadata. So it is up to the user build the own solution. There are multiple ways to solve it.

1. **Do It Yourself(DIY):** Since google does allow users to crawl the app pages, and we can find all such urls in the already provided site map. User can actually build a crawler using scrappy library in python[3]. *Scrappy* also contains Link Extractor class, which could be used to extract links from playstore. This option although free is much hard to maintain, as any change in the playstore might needs us to change the code. Although we expect such instances to be not frequent.

Note that, we could have similar solution for itunes also.

2. **Paid Solutions**: There are multiple vendors in the market which give access to either api or/and a bulk downloader. **42matters** is one such service I used in the past[4]. It provides both file dump and lookup API. The API is rate limited to **50 qps**.

3. **Open Source Solutions:** There are bunch of open source tools too for the same. [5] is one such solution which consumes data in protobuf format to read data from Google.

Limiting the Rate of data loading

All solutions discussed above have limited the number of calls one could make in a time window. This is mostly done to stop DDOS attacks, intentional or unintentional by the users. There are several ways to tackle the throttling done by the servers.

Bypassing the limit

Rate limit can be bypassed for the services which uses ip address to track the number of calls. Web servers as an example mostly use plugins working on same principle. So the web crawler can either use vpn or proxy to rotate the ip addresses. Similarly we can use multiple machines with different ip addresses

However this technique will not work if service required authentication as they will be using account id for the counter.

Moreover Such practices are often looked upon as unethical and it is better to avoid the rate threshold rather than bypassing it.

Avoiding the limit

We could check with the documentation of the APIs, in case of the web crawlers, check the running http server, for the queries per second supported for each ip address or account. Then write code in a way that the client, sleeps after making the available requests in the given time window.

Recovering once the limit has been reached is also equally important. Some services simply returns *HTTP 503: Service Unavailable*, Others may actually return number of calls made and number of calls remaining in the response header, while giving *Retry-After* time. Basically if we get multiple request failures consecutively, we would get the client to sleep for a random time(if retry time is not given in header) before attempting the connection again.

Optimizing the number of calls

We can optimize further by taking data only when required only for the intended apps. Here are some of the optimizations we can use to reduce the number of calls. This would help to manage the rate and also save costs, if using paid services.

1. Generally app stores have a long tail of apps which have very low user base and are seldom useful. We could cut some of

these apps.
2. Apps with very high number of users, will not have any dramatic change in much of metadata like rating, number of downloads, number of reviews etc. Also data like download size, available regions etc. should not change much frequently.
3. We could bucket the apps in different groups each having different rate of refresh based on various properties. Like number of popularity, importance etc.
4. maintain a list of apps we have in a fast memory cache - *Memcache, Redis etc*. Keep taking diff with site maps to get the new apps added, we would need to query them next time.

Designing the System.

Since the system needs to repeat the job but not in realtime, we can setup a cron job for the same. It could be done either by using vanilla Linux's *crontab* function or use other job schedulers. *Jenkins*[6] is one such automation tool with very rich library of plugins. Following are the other design details.

1. Any high level programming language could be used to perform the task. However some languages excels in some key areas. If we need a web crawler, then Python could be a choice with its excellent libraries for string operations and presence of parsers like Scrappy, BeautifulSoup.
2. Instead of making too many http connections, it is recommended to have persistent http connections by setting bigger keep alive time and manage a connection pool

3. We might not need to use asynchronous requests as we could hit the rate limit even in a single thread.
4. For writing into Redshift, we could simply upload a big DB file to S3 and copy it into Redshift, or use a Firehose-Redshift delivery stream. Since we are not dealing with a realtime system, we would save money on the kinesis stream and simply use S3. Put and get requests on S3 have very low costs over the internet and free within the datacenter.
5. We should be taking a machine in AWS only to save the data transfer costs and easy integration. Since we are dealing with low data(about 5 million rows per day) and the job is not cpu, memory or disk intensive. We could take an allrounder small machine - m3.medium. We could use an EC2 spot machine for the task as we require machine only once a day if running as a single cron.
6. Redshift table could be a flat table and be used as an OLAP system. Also keep a created_at columns to mark the time of row creation. This could be used for garbage collection and other queries. We could have app bundle_id as distkey and interleaved sort key of (created_at, other columns). other columns will be the columns used more frequently for filtering.

References

[1] https://affiliate.itunes.apple.com/resources/documentation/itunes-store-web-service-search-api/#searchexamples

[2] https://affiliate.itunes.apple.com/resources/documentation/itunes-

enterprise-partner-feed/#fileformat

[3] https://scrapy.org/

[4] https://42matters.com/

[5] https://code.google.com/archive/p/android-market-api/

[6] https://jenkins.io/