# Scraping Hackernews Data

*Szymon Lipiński*

## Downloading The Data

The data is downloaded to a set of csv file using the code available at https://github.com/szymonlipinski/hackernews_dowloader.

This made the following files:

```
du -ah /home/data/hn
```

```
## 504M /home/data/hn/1554994838_1565866358__19635134_20704074.data.csv
## 485M /home/data/hn/1401111353_1421029195__994369_8872196.data.csv
## 467M /home/data/hn/1300452676_1326824346__2340190_3475825.data.csv
## 450M /home/data/hn/1209976952_1271054983__181299_1258580.data.csv
## 506M /home/data/hn/1543560409_1554994838__18567251_19635134.data.csv
## 505M /home/data/hn/1556804358_1567598222__19807762_20876231.data.csv
## 493M /home/data/hn/1454584884_1468706219__11033266_12108033.data.csv
## 505M /home/data/hn/1531176188_1543560409__17494018_18567251.data.csv
## 490M /home/data/hn/1421029195_1437923693__8872196_9951246.data.csv
## 499M /home/data/hn/1493867970_1506462121__14262172_15342765.data.csv
## 465M /home/data/hn/1271054983_1300452676__1258580_2340190.data.csv
## 478M /home/data/hn/1367793791_1384391674__5660073_6729887.data.csv
## 82M  /home/data/hn/1554994838_1556804358__19635134_19807762.data.csv
## 505M /home/data/hn/1519132696_1531176188__16420052_17494018.data.csv
## 474M /home/data/hn/1326824346_1348853030__3475825_4586677.data.csv
## 498M /home/data/hn/1468706219_1481804010__12108033_13184043.data.csv
## 502M /home/data/hn/1481804010_1493867970__13184043_14262172.data.csv
## 502M /home/data/hn/1506462121_1519132696__15342765_16420053.data.csv
## 476M /home/data/hn/1348853030_1367793791__4586677_5660073.data.csv
## 498M /home/data/hn/1437923693_1454584884__9951246_11033266.data.csv
## 69M  /home/data/hn/1160418111_1209976952__1_181299.data.csv
## 478M /home/data/hn/1384391674_1401111353__6729887_7799657.data.csv
## 9,7G /home/data/hn
```

## Creating The Database Structure

All the data is too large to keep it in R in memory for processing on my machine. An alternative is to keep it in a database, I chose PostgreSQL.

The table structure for the csv data is:

```
##
## CREATE TABLE raw_data (
##     title TEXT,
##     url TEXT,
##     author TEXT,
##     points INT,
##     story_text TEXT,
##     comment_text TEXT,
##     num_comments INT,
##     story_id INT,
##     story_title TEXT,
##     story_url TEXT,
##     parent_id INT,
##     created_at_i INT,
##     type TEXT,
##     object_id INT
## );
```

All the files have been loaded with:

```
## #!/bin/bash
##
##
## if (( $# != 4 )); then
##     echo "Loads data from csv files to a postgres database"
##     echo "USAGE:"
##     echo "./load_files.sh DBNAME DBUSER TABLE_NAME FILES_DIRECTORY"
##     exit 0
## fi
##
## DBNAME=$1
## DBUSER=$2
## TABLE_NAME=$3
## FILES_DIRECTORY=$4
##
## for f in $FILES_DIRECTORY/*.csv
## do
##     echo "Loading $f"
##     psql $DBNAME -U $DBUSER -c "\\COPY $TABLE_NAME FROM $f WITH CSV DELIMITER ',' HEADER "
## done
```

The loading time was about 6s per file.

# Basic Data Cleaning

## Removing Duplicates

According to the documentation of the downloader program:

```
Some entries in the files are duplicated, which is basically because of the Algolia API limitations. Wha
```

To remove the duplicates, I used a simple query which should create a new table without the duplicated rows. The primary key for the data is the `object_id` column, so to make things faster, I created an index, and used `distinct on`:

```
## BEGIN;
##
## CREATE INDEX i_raw_data_object_id ON raw_data (object_id);
##
## CREATE TABLE data AS
## SELECT DISTINCT ON (object_id) *
## FROM raw_data;
##
```

```
## DROP TABLE raw_data;
##
## COMMIT;
```

## Adding Indices

I also need some indices on the data table for faster searching. I omitted the text columns, except for the ones where I will use the whole text to search, like `type = 'comment'`.

```
## CREATE INDEX i_data_author       ON data (author);
## CREATE INDEX i_data_points       ON data (points);
## CREATE INDEX i_data_num_comments ON data (num_comments);
## CREATE INDEX i_data_story_id     ON data (story_id);
## CREATE INDEX i_data_parent_id    ON data (parent_id);
## CREATE INDEX i_data_created_at_i ON data (created_at_i);
## CREATE INDEX i_data_type         ON data (type);
## CREATE INDEX i_data_object_id    ON data (object_id);
```

## Preprocessing Data

In the further data processing, I will need to repeat some data operations. To speed it up, I will calculate a couple of things and store it in the database. I like to use materialized views for this for two reasons:

1. They can be easily refreshed to recalculate the data again.
2. They don't change the original data.

### Calculating The Dates

The only date field in the data table is the `created_at_i` which is an integer with number of seconds since the Jan 1st, 1970. As I will need to aggregate dates by weeks, days of week, months, years, to decrease the query time later, I will calculate it now:

```
## create materialized view dates as
## select
## object_id,
## timestamp 'epoch' + created_at_i * interval '1 second' as date,
## date_part('year',   timestamp 'epoch' + created_at_i * interval '1 second') as year,
## date_part('month',  timestamp 'epoch' + created_at_i * interval '1 second') as month,
## date_part('week',   timestamp 'epoch' + created_at_i * interval '1 second') as week,
## date_part('day',    timestamp 'epoch' + created_at_i * interval '1 second') as day,
## date_part('dow',    timestamp 'epoch' + created_at_i * interval '1 second') as dow,
## date_part('hour',   timestamp 'epoch' + created_at_i * interval '1 second') as hour,
## date_part('minute', timestamp 'epoch' + created_at_i * interval '1 second') as minute,
## date_part('second', timestamp 'epoch' + created_at_i * interval '1 second') as second,
## to_char(timestamp 'epoch' + created_at_i * interval '1 second', 'yyyy-MM')  as year_month
## from data;
```

For faster searching, I will add some indices on the above view:

```
## create index i_dates_object_id on dates(object_id);
## create index i_dates_year on dates(year);
## create index i_dates_month on dates(month);
## create index i_dates_date on dates(date);
```

## Getting URLs

I will also get all the urls from the specific fields. For now I will mark the source of the url, as it is possible that the urls distribution in stories text is different than in comments.

```
## -- The urls can be everywhere
## -- If the entry type is a story, then it has fields like: title, url
## -- If it's a comment, then it has comment_text, story_title, story_url
## -- Jobs can have url, title, and story_text
## create materialized view
## urls as
## with url_data as
## (
##     select
##         distinct
##         object_id, 'comment_text' as type,
##         unnest(
##             regexp_matches(lower(comment_text), '([\.\w\d]*://[^\s<"]+)',  'g')
##         ) url
##     from data
##     UNION ALL
##     select
##         distinct
##         object_id, 'story_title',
##         unnest(
##             regexp_matches(lower(title), '([\.\w\d]*://[^\s<"]+)',  'g')
##         ) url
##     from data
##     UNION ALL
##     select
##         distinct
##         object_id, 'story_text',
##         unnest(
##             regexp_matches(lower(story_text), '([\.\w\d]*://[^\s<"]+)',  'g')
##         ) url
##     from data
##     UNION ALL
##     select
##         distinct
##         object_id, 'url',
##         unnest(
##             regexp_matches(lower(url), '([\.\w\d]*://[^\s<"]+)',  'g')
##         ) url
##     from data
## ),
## clean_urls as (
##      SELECT DISTINCT object_id, type, rtrim(url, './') as url
##      FROM url_data
##      WHERE url not like '%...'
## ),
## parts as (
##  SELECT
##      object_id, type, rtrim(url, './') as url,
##      (regexp_matches(url, '^(\w*)://([^/]*)/.*$')::TEXT[])[1] as protocol,
##      (regexp_matches(url, '^(\w*)://([^/]*)/.*$')::TEXT[])[2] as domain,
##      (regexp_matches(url, '^(\w*)://([^/]*)/(.*)$')::TEXT[])[3] as path
##  FROM clean_urls
## )
## select
##  *
## from parts;
```

For faster searching, I will add some indices on the above view:

```
## create index i_urls_object_id on urls(object_id);
## create index i_urls_protocol on urls(protocol);
```

## Database Size

The main table size with all indices:

```
require("RPostgreSQL")
drv <- dbDriver("PostgreSQL")
con <- dbConnect(drv, dbname = "hn",
                 host = "localhost", port = 5432,
                 user = "hn", password = "hn")

tables <- dbGetQuery(con, '
  SELECT
    tablename "Table Name",
    pg_size_pretty(pg_relation_size(tablename::text)) "Size"
  FROM
    pg_tables where schemaname=\'public\'
  ORDER BY
    tablename
  ')

views <- dbGetQuery(con, '
  SELECT
    matviewname "View Name",
    pg_size_pretty(pg_relation_size(matviewname::text)) "Size"
  FROM
    pg_matviews where schemaname=\'public\'
  ORDER BY
    matviewname
  ')

indices <- dbGetQuery(con, '
  SELECT
    tablename "Table Name",
    indexname "Index Name",
    pg_size_pretty(pg_relation_size(indexname::text)) "Size"
  FROM
    pg_indexes
  WHERE schemaname = \'public\'
  ORDER BY tablename, indexname;
  ')
```

### Tables

```
##   Table Name    Size
## 1       data 9764 MB
```

### Materialized Views

```
##   View Name    Size
## 1     dates 2156 MB
## 2      urls  858 MB
```

### Indices

```
##    Table Name          Index Name   Size
## 1        data       i_data_author 505 MB
## 2        data i_data_created_at_i 414 MB
## 3        data i_data_num_comments 414 MB
## 4        data    i_data_object_id 414 MB
## 5        data    i_data_parent_id 414 MB
## 6        data       i_data_points 414 MB
## 7        data     i_data_story_id 414 MB
## 8        data         i_data_type 414 MB
## 9       dates        i_dates_date 414 MB
## 10      dates       i_dates_month 414 MB
## 11      dates   i_dates_object_id 414 MB
## 12      dates        i_dates_year 414 MB
## 13       urls    i_urls_object_id 105 MB
## 14       urls     i_urls_protocol 105 MB
```

## URLs Analysis

Get all the URLs regardless the field type, we count one URL per object_id. So, if the same URL appears in a title and description, then it's counted as one. However, it it's in a story and in a comment to this story, then they are counted as two separate URLs.

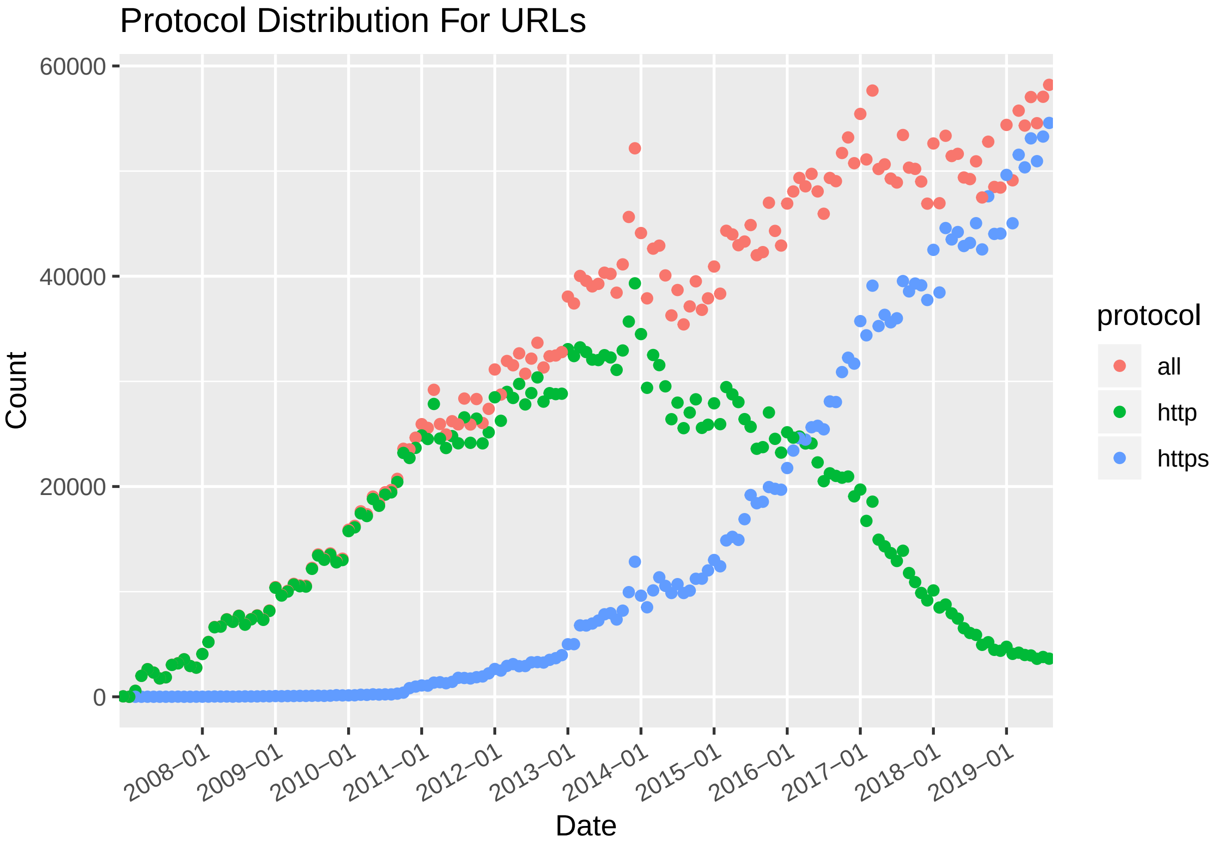
```
urls <- dbGetQuery(con, "
  WITH distinct_urls AS (
    SELECT DISTINCT object_id, protocol, url
    FROM urls
    WHERE protocol IN ('http', 'https')
    UNION
    SELECT DISTINCT object_id, 'all',
           domain || '/' || path as url
    FROM urls
    WHERE protocol IN ('http', 'https')
  )
  SELECT protocol, year_month date, count(*)
  FROM distinct_urls
  JOIN dates USING (object_id)
  WHERE date < date_trunc('month', now())
  GROUP BY protocol, year_month
  ORDER BY year_month desc, protocol
")


dateRange <- dbGetQuery(con, "SELECT min(year) min, max(year) max FROM dates")

library(ggplot2)

breaks <- mapply({function (year) sprintf("%s-01", year)},
                 seq(dateRange$min[1], dateRange$max[1]))

gg <- ggplot(data=urls, aes(x = date, y = count)) +
  geom_point(aes(color=protocol)) +
  labs(title="Protocol Distribution For URLs",
       x="Date",
       y="Count") +
  scale_x_discrete(breaks = breaks) +
  theme(axis.text.x = element_text(angle = 30, hjust = 1))
plot(gg)
```



```
png("protocol_distribution.png")
print(gg)
dev.off()
```

```
## pdf
##   2
```

## Software Versions

- **OS**: *Ubuntu 18.04.3 LTS*
- **Python**: *3.7.1 (default, Oct 22 2018, 11:21:55) , [GCC 8.2.0]*
- **PostgreSQL**: *PostgreSQL 10.10 (Ubuntu 10.10-1.pgdg18.04+1) on x86_64-pc-linux-gnu, compiled by gcc (Ubuntu 7.4.0-1ubuntu1~18.04.1) 7.4.0, 64-bit*
- **R**: *x86_64-pc-linux-gnu, x86_64, linux-gnu, x86_64, linux-gnu, , 3, 4.4, 2018, 03, 15, 74408, R, R version 3.4.4 (2018-03-15), Someone to Lean On*