Hacker News: Making a Relevant Search.

Juan Agustin Gregorieu[1]

[1]*Northeastern University*

Hacker News (HN) can be defined as a mixture of both a news website and a forum. It is oriented towards presenting news related to technology and entrepreneurship. News are posted by HN users and the platform strives to guarantee good quality news, and dicussions generated from each of them.

HN uses a search engine powered by Algolia to allow users search over the posts. I have detected that the results returned by Algolia are not what is expected from a search engines that works over news. In this paper I am going to explore and provide a way of improving the search results given certain characteristics of the posts.

I. INTRODUCTION

Hacker News (HN) can be defined as a mixture of both a news website and a forum. It is mainly focused on providing a list of posts or news regarding technology and entrepreneurship. The users submit content and if it is considered useful by the community, the content is voted and/or commented. Sometimes from this stories an interesting discussion emerges (in the form of comments), creating what we can call a forum. Thanks to how is moderated, the users votes and discussions generated, we can filter relevant technological news easily while including opinions of usually expert users.

HN provides the users with a search engine powered by Algolia to search through its database of posts. With over 1000 new posts added everyday, we would expect that when we search for something, the results returned would be filtered to present only the relevant ones to the user. However, the search engine seems to fail in this task. I have listed some of its problems:

- Given any query string, the search only retrieves posts sorted in descending order of popularity or freshness. It does not determine relevance based on any of those factors but directly sorts the result by any of them. A search engine of this kind should return relevant news, and one classical way of defining relevance in this context is by taking into account both the freshness of the post and its popularity.

- The search engine is quite restrictive in

how we can express our query. A subtle difference between a singular or plural term can return entirely different sets of results, this can happen also with related terms. An example of this can be seen by searching for "React frustration" and "React frustrations".

- Related to the above item, Algolia seems to search for all of the query terms and if all of them are not in the same document, it will not return any result.
- Regarding the UI, the search results usually lack titles or even snippets. So it may not be easy for a user to identify a relevant document very quickly.

In the following sections I am going to explore and propose a way of achieving a relevant search given a user query, using Solr as the backbone of my solution. First, I am going to define my dataset. Secondly, then I am going to identify some features of the stories that can give us an idea of how relevant it is and I am going to create a formula based on that information. This formula is going to generate a score per each story in order to rank results before presenting them to the user. To conclude, I am designing my own top-10 results user interface to display the results generated with my ranking formula.

II. DATASET

Hacker News stores all of its posts in a cloud tool called Googles Big Query. This is the most fresh source of information anyone can get from HN without having to crawl the entire website. Currently the most recent posts are 2 months old, which aligns perfectly with my goals towards this project.

A table called "bigquery-public-data.hacker_news.full" of self-referential nature is provided to the users so they can create their own datasets or generate some statistics. The table stores stories and its comments at the same level, each one has a unique identification number and a "parent" field that references the identification number of a parent record. In the context of my project I will only be using stories that are 3 years old or less (from 2016-11-15 to 2019-09-16), and only getting the first level of comments. The query issued to generate this 1GB dataset of 1,064,628 documents of type "story" was the following:

```
SELECT
  p.id,
  p.title,
  p.url,
  p.text,
  p.by,
  p.score,
```

```
  p.time,
  p.timestamp,
  p.descendants,
  ARRAY(
    SELECT
      AS STRUCT c0.text
    FROM
      'hacker_news.full' c0
    WHERE c0.parent = p.id
  )
FROM
  `hacker_news.full` p
WHERE
  p.type = 'story'
  AND p.timestamp > TIMESTAMP_SUB(
    CURRENT_TIMESTAMP(),
    INTERVAL 26280 HOUR
  )
```

III. SCHEMA

The first step needed to start using Solr with the data collected is to define a good schema. This consists in defining which are going to be the fields of my document, and which of them are going to be indexed and therefore searchable. In addition, the data types need to be stated in our definition, very similar to the way we create tables in a relational database.

HN Schema	
Field Name	Field Type
title	text_en
url	string
text	text_en
author	string
timestamp	pdate
popularity_points	pint
story_comments	text_html
comments_number	pint

Title, text, comments and author are indexed (searchable). Moreover, the field **title** has an extra attribute called *omitTermFreqAndPositions* set to *true* since I do not want the term frequency to be used in this field to boost the score. For instance, if I search for "lucene", and a title mentions that word 2 times, by default (only for *text_** field types) Solr is going to boost the score for that field since the query term is mentioned twice. By setting this attribute I can avoid that situation.

I have chosen the field type *text_en* since it provides us with several useful tools at indexing and query times. By default, at indexing time it tokenizes the text using the following filters:

- A stop words filter.
- A case folding filter.
- An English possessive nouns filter.

- A Porter stemming filter.

And at query time, its configuration allows us to use all of the above mentioned items but with one addition which consists of a filter for synonyms.

Moreover, I created a new field type that is not included by default in Solr. This field is called *text_html* and has the same characteristics as *text_en* but it includes one additional tool called *HTMLStripCharFilterFactory* which basically helps to tokenize and strip HTML chars from a text at indexing time. I use it for the comments field since the dataset returns some HTML tags within each of them and I do not want to index them.

IV. RELEVANCE

Each time a user issues a query in Solr, the search engine tries to match it against certain fields in the schema. The decision of which fields we should match and how to weight them is not trivial. In the case of HN, I have identified three fields that should be considered:

- Title: The title of the story. Is usually very short and provides a considerably short description of the topic of the story.
- Description: A brief description or opinion stated by the user. This field is usually empty or null.
- Comments: All comments issued by the users. As I am going to detail later, all comments are strictly moderated in order to assure good quality in every story.

A match in any of those fields should be enough to increase the score of the document. But, a match in a field like *title* should be more important that a match on the field *text*. In addition, although comments are important, a match in any of them is not that important as in the previous mentioned fields, and it must not boost the score that much: imagine we issue the query "react" and a user uses that word in a comment of certain story to provide an argument, but the topic of the story is not about it, then it should not be scored too high. So, I decided to apply the following boosts to each field:

```
title^1.4 text^0.3 comments^0.4
```

This can be very easy to interpret. For instance, if a query matches the title, then the score for that field will be boosted by by a 40% of the original score. The match in the text is multiplied by a 0.3, which means we are taking the 30% of the score calculated by Solr to generate the final score (or reducing it by a 70%). The reason I am boosting

this field in such a way is that after uploading my dataset to R and analyzing its data, I had identified that 94.7% of the records do not contain any value in it, so a higher boost would give an enormous advantage to the stories that have a value for text. And regarding the comments, the same idea applies, only 22.50% of the stories have comments. So, in this case, I am only taking the 40% of its score for the final value of the field.

As said before, Hacker News Search Engine does not seem to be working correctly. If we submit any query to it we will see that the results are not as expected. It turns out that either sorts them by popularity of the posts or by its date, and we want relevant results, not sorted ones. Since HN is a news website, users may want some fresh documents when they perform a search, along with documents that are popular among the community and that have a fair enough number of documents. Therefore, to get relevant documents on every search I need to develop a formula that combines the factors I mentioned earlier and boosts the score accordingly. I am considering the following characteristics to develop my formula:

1. Popularity: In the HN context, a story can be up-voted, meaning that it is found interesting by a person. Therefore, the more popularity points a story has the more relevant it may be.

2. Freshness: A user of a news website may find more relevant a document that is relatively new than a document that is very old. Especially in a tech context where things change at a very fast pace. For instance, if I want to search for news related to Python, a result from 3 years ago seems to be highly irrelevant knowing that each year many new versions are released.

3. Number of Comments: HN moderates the comments of every story that has been posted in order to guarantee high quality discussions. Therefore I make the assumption that stories with more comments are stories that can be more helpful to the user that is searching for some particular topic.

A. Popularity

We want that a story that contains more popularity points gets a better score. So if I have a story with 20 points and another story with 5, the first one will be scored higher than the second one. This seems similar to the typical *TF* score: the more times we see a particular term in a document, the higher it should be scored.

So, how do we calculate the score? An-

alyzing the example of the two stories of 20 and 5 popularity points, we could start using the raw points to influence the score of the document. But this would mean that the first story is about 4 times more relevant than the second one, which is incorrect. Therefore, we need a smooth function that solves that problem, like $\log(1 + x)$. So, I am going to use it as a starting point (Figure 1) .

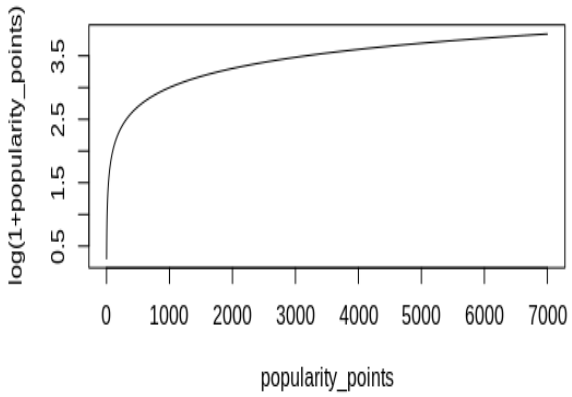

FIG. 1. f(x) = log(x + 1)

By using Solr and supplying it a dataset, I can easily obtain the maximum value of a field. In the case of the popularity, the maximum value equals to 6015. Evaluating the function on real values could throw some helpful insights:

$$\log(1 + 0) = 0$$

$$\log(1 + 6015) = 3.78$$

By analyzing the results, we see that when the story has no popularity at all, the function will return a value equal to 0. And if we evaluate the function using the maximum value of the field, the function adopts a value of 3.78. A boost of 3.78 seems to aggressive, and shows how this function is going to favor a lot very popular posts. We can use the results of Solr's stats component to get information that can help us find a more suitable ranking function:


```
"stats": {
    "stats_fields": {
      "popularity_points": {
        "min": 0.0,
        "max": 6015.0,
        "count": 1064628,
        "missing": 0,
        "sum": 14468834,
        "sumOfSquares": 4099545000,
        "mean": 13.590506730989604,
        "stddev": 60.54737505232242
      }
    }
  }
```


The maximum value encountered is 6015 but the average of popularity points is about 13.60. So most of the stories in my dataset of over 1 million records have popularity points around the value of 13.60. I am going to round this value to 14 for future use. I uploaded my dataset to R to run some simple statistics and I identified that around 3.58% of the stories in our dataset have no popularity points at all, and 90.20% have a score lower than 14. By analyzing Figure 2 we can

see how the values are spread using the logarithmic function.

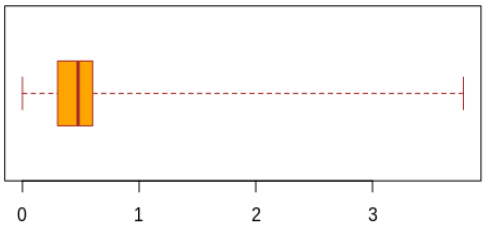

FIG. 2. Boxplot for $f(x) = \log(1+x)$ applied to popularity points.

Since the logarithmic function we are using grows without bounds as the input increases, a very high value like 6000 can give a huge boost to some very popular documents, therefore posts with more than 14 popularity points will be favoured more. To make it fair for the rest of the documents, a better approach is to choose a function that grows unbounded and smoother. A function like this exists and it requires to define two values:

1. A maximum: The maximum value that our function can take. This is basically the value our function will try to approach but will not reach. This value can go from 1 to 2, and usually 1.5 is chosen.

2. A horizon: The horizon of the function. The function valued in this value equals to 1. In this case horizon is equal to the average of popularity points across my dataset.

With the previous values in mind, I can construct the formula:

$$f(x) = \frac{maximum - maximum^2}{\frac{1}{horizon} * x + maximum - 1} + maximum$$

Now we replace "horizon" in the formula with 14, and "maximum" with 15:

$$\begin{aligned} f(x) &= \frac{1.5 - 1.5^2}{\frac{1}{14} * x + 1.5 - 1} + 1.5 \\ &= \frac{-0.75}{0.071428571 * x + 0.5} + 1.5 \end{aligned} \tag{4.1}$$

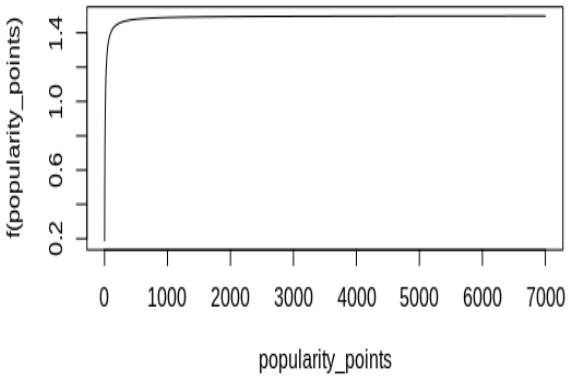

FIG. 3. $f(x) = \frac{-0.75}{0.071428571*x+0.5} + 1.5$

The function now looks smoother and seems representative of how an user should perceive relevance when it comes to popularity points (Figure 3).

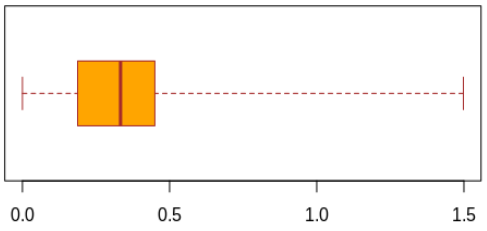

FIG. 4. Boxplot for $f(x) = \frac{-0.75}{0.071428571*x+0.5} + 1.5$ applied to popularity points.

We can test how our equation works by providing it with the same numbers we tested before:

$$f(0) = \frac{-0.75}{0.071428571 * 0 + 0.5} + 1.5 \\ = 0 \tag{4.2}$$

$$f(14) = \frac{-0.75}{0.071428571 * 14 + 0.5} + 1.5 \\ = 1 \tag{4.3}$$

$$f(6015) = \frac{-0.75}{0.071428571 * 6015 + 0.5} + 1.5 \\ = 1.49 \tag{4.4}$$

By analyzing the previous values we can see that for the maximum value of popularity points encountered in our dataset, the function is returning a fairer number (1.49) than the one of 3.78 that the logarithmic function was generating. I have managed to find a function more representative of the users' perception of relevance, you can look at the boxplot in Figure 4 to see how the values are distributed using the new formula.

Solr provides many tools that allow to create functions as the one described above, in this case, the corresponding function is the following:

```
sum(
    recip(
        'popularity_points',
        0.071428571,
        -0.75,
        0.5
    ),
    1.5
)
```

We can decompose the above function in other simpler functions:

```
sum(recip(x, m, a, c), max)
= sum(div(a, linear(x, m, c)))
= sum(div(a, sum(prod(m, x), c)))
```

Where the arguments are:

- $x = popularity\_points$
- $m = \frac{1}{horizon}$
- $a = max - max * max$
- $c = max - 1$

B. Freshness

As I said earlier, another important aspect of a news search engine is that relevance is influenced by the freshness of a document. In

the HN case, each document has a timestamp associated with it that indicates the date and time when the story was posted. Intuitively, we want to increase the score of those documents that have been posted recently. My approach is to subtract from the current date the date of the post, in this way I can get its age. The higher the age of document, the less likely it is going to be relevant. Analyzing the last example, the query"react" was issued and a post of about 2 years ago was retrieved and ranked very highly, which does not make sense in our current context.

So, as said in the last paragraph, the higher the age, the less it should affect our scores. A good option for this scenario is the reciprocal of a function.

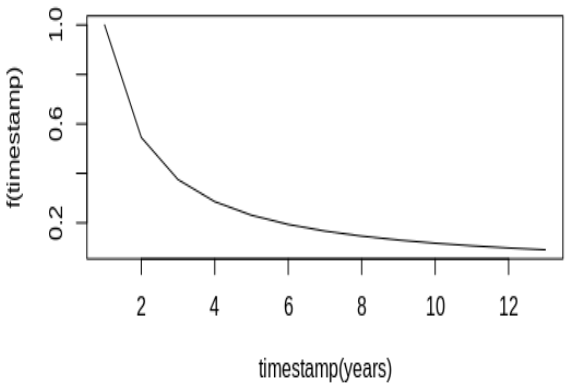

FIG. 5. $f(x) = \frac{3.78E10}{x+3.78E10}$

As we can see in Figure 5, as the timestamp (measured in milliseconds) assumes higher values, the function value decreases. The generic formula is $f(x) = \frac{c}{x+c}$, and to make it suitable to our problem I defined c as one tenth of the horizon. Since HN started on February 19 of 2007, the horizon is 12 years in the past. And since Solr handles time in milliseconds, we can get this value easily using its own formulas:

We can run $ms(NOW, NOW - 12YEARS)$ in Solr. Solr returned 378,691,191,000 ms which is equal to 3.78E11. Since we want one tenth of this value, the one we using in the formula is 3.78E10.

Now, replacing it on the function we get:

$$f(x) = \frac{3.78E10}{x + 3.78E10}$$

Let us value the function in 0 and 3.78E11:

$$f(0) = \frac{3.78E10}{3.78E10} = 1$$

$$\begin{aligned} f(3.78E11) &= \frac{3.78E10}{3.78E11 + 3.78E10} \\ &= 0.09 \approx 0 \end{aligned} \tag{4.5}$$

The above results seem to proxy very well the users' perception of relevance for the freshness of a story.

The corresponding function in Solr es equivalent to:

```
recip(
    ms(NOW/DAY,timestamp),
    1,
```

```
    3.78E10,
    3.78E10
)
```

C. Number Of Comments

The HN community is very concerned about displaying only high quality stories and high quality comments associated to them: only relevant comments that contribute to the discussion are allowed. Therefore, I am going to take advantage of this particular situation and score documents based on the number of comments: the higher the number of documents, the highest I am boosting a document score because it contains more information about the topic I am searching for.

The formula I am applying is the same I applied to the popularity points, but with a different horizon. To determine which is the new horizon, we are analyzing the distribution of data in the *comments_number* field.


```
"stats": {
    "stats_fields": {
      "min": 0.0,
        "max": 2372.0,
        "count": 1046461,
        "missing": 0,
        "sum": 6514544.0,
        "sumOfSquares": 1218887910,
        "mean": 6.225309877768976,
        "stddev": 33.55619251353792
      }
    }
  }
```


We are facing a situation very similar to the one faced when searching for a good ranking function for popularity points. The maximum number of comments for a story is 2372, but the average is of 6.25. After rounding the mean, the new horizon I am considering it equal to 6, leaving us with the following formula:

$$f(x) = \frac{-0.75}{0.1667 * x + 0.5} + 1.5$$

We can test how it works by evaluating with the values we are interested in:

$$f(0) = \frac{-0.75}{0.1667 * 0 + 0.5} + 1.5 = 0$$

$$f(6) = \frac{-0.75}{0.1667 * 1.40 + 0.5} + 1.5 \approx 1$$

$$f(2372) = \frac{-0.75}{0.1667 * 1.40 + 0.5} + 1.5 \approx 1.49$$

Again, I have reached satisfying values for my ranking function.

V. IMPLEMENTING THE QUERY FUNCTIONS

Since we are using multiplicative boosting and each of the ranking functions are mostly returning value in the range of $[0, 1]$ (timestamp) or $[0, 1.5]$ (popularity points and number of comments) we are adding a 1 to this range in order to get a value between $[1, 2]$ and therefore avoid the possible scenario of multiplying the Solr score by a zero.

I now have defined all of functions that will allow to proxy the users' perception of relevance when searching through my proposed search engine. However, we still need to combine and weight them appropriately. To do so, I am going to define a generic formula and tune it until I had reached what I think are relevant results. The function has the following form.

$$score = (w_f * s_f + w_p * s_p + w_c * s_c) * s_s$$

Where p, c, and f are popularity points, number of comments and freshness respectively. Each s is the score I generate after I apply the corresponding functions to each field, and w are weights I am assigning to proxy the grade of importance of each attribute. s_s is the score Solr provides as a result of matching the query string against the story.

After testing with different values, I was able to find what I consider to be very good weights for the formula I described above. My initial approach was considering that $w_c < w_p < w_f$. So, based on that and after experimenting with many different values I reached to the following weights:

$$score = (10 * s_f + 5 * s_p + 2 * s_c) * s_s \quad (5.1)$$

VI. REQUEST HANDLER

Every configuration that I mentioned in the previous sections is going to be included into Solr in the form of a *Request Handler*. A *Request Handler* defines how Solr is going to process a certain request without having to add them to the query string when requesting results. Also, it is very useful to offer the client many different ways of querying the data. In this case I offer only one way (apart from the default one Solr offers) called *hnselect*.

```
<requestHandler name="/hnselect"
  ↪class="solr.SearchHandler">
   <lst name="defaults">
      <str name="defType">edismax</str>
      <str name="qf">title^1.4 text^0.3
        ↪comments^0.4</str>
      <str name="q.alt">*:*</str>
      <str name="mm">2</str>
      <str
        ↪name="echoParams">explicit</str>
      <str name="wt">json</str>
      <str name="indent">true</str>
      <str name="tie">1.0</str>
      <str name="hl">true</str>
      <str name="hl.method">unified</str>
      <str name="hl.tag.ellipsis">...
        ↪</str>
      <str name="hl.fragsize">200</str>
      <str name="hl.fl">text
        ↪comments</str>
      <int name="hl.snippets">3</int>
      <str name="hl.tag.ellipsis">...
        ↪</str>
   </lst>
   <lst name="appends">
      <str name="boost">
        sum(
```

```
            product(
              2,
              sum(
                recip(
                  comments_number,
                  0.7143,
                  -0.75,
                  0.5
                ),
                1.5,
                1
              )
            ),
            product(
              15,
              sum(
                recip(
                  ms(NOW/DAY,timestamp),
                  1,
                  3.78E10,
                  3.78E10
                ),
                1.5,
                1
              )
            ),
            product(
              5,
              sum(
                recip(
                  popularity_points,
                  0.07142,
                  -0.75,
                  0.5
                ),
                1.5,
                1
              )
            )
          )
        </str>
    </lst>
</requestHandler>
```

One important observation is that if a given query matches the three fields, *edismax* by default will return the maximum of those three different scores (disjunction max query), which does not apply to what I want. So I used a tie of 1.0 in order to calculate the final score based on the sum of the subquery scores (disjunction sum query). Consequently lower scores will influence the final Solr score.

VII. USER INTERFACE

To display the results I have developed a user interface using *React* library along with *Semantic UI* to style each component. The results are presented in a vertical list, each of them consists of the following items (see Figure 6):

- A blue and underlined title.
- A link under the title. This link is the one the story points to, but not the link of the story itself.
- A snippet containing a portion of the document. The snippet has a size of 200 characters and is generated by Solr using the *unified highlighting component*. It displays a contiguous series of sentence fragments separated by ellipses ("...") with query terms in bold. The snippet is generated over two fields: *text* and *comments*.

- Metadata that includes the author of the story, the number of comments, number of likes, and date it was published.

When the user submits a query, a list of the top-10 best results is displayed. At the end of the list we can see a paginator available to wade through the rest of the results retrieved (if system retrieves enough results).

FIG. 6. UI structure of a result.

VIII. RESULTS

The results provided were satisfactory. As an example, I am searching for the word "react" in both search engines and compare its top-3 results. In Figure 7 we can see the results returned by the default Algolia search and in Figure 8 the ones returned by my proposed search engine. As you can see, both return totally different results. Algolia returns three posts, of which the first and third ones have a very old age and therefore are irrelevant (I am not considering the second one since its one month old and therefore not included in my dataset). As an user, I would not consider relevant a React news article that is 2 years old, even if it was very popular (reason why it was retrieved as the first result). On the other hand, my proposed search engine returns three popular and fresh posts, all from different months of the same year, and with quite an amount of comments.

One big problem I faced by reading my results is that the snippets are not descriptive enough, and this is a problem inherent to the nature of the posts itself. They are not enough self descriptive and some lines from the comments do not reflect the main topic in most cases. So, one option would be to remove the snippets from the results, and other option would be to generate a snippet based on the URL the story points to, this should contain more information about the topic of the story.

IX. ADDITIONAL NOTES

In addition to the measures of popularity, freshness and number of comments, I had also considered to increase the score of a post based on how authoritative was the author. At the beginning I tried using the average number of votes the author used, but it was not a good measure: a relatively new author could have a few very popular publica-

FIG. 7. Algolia Results.

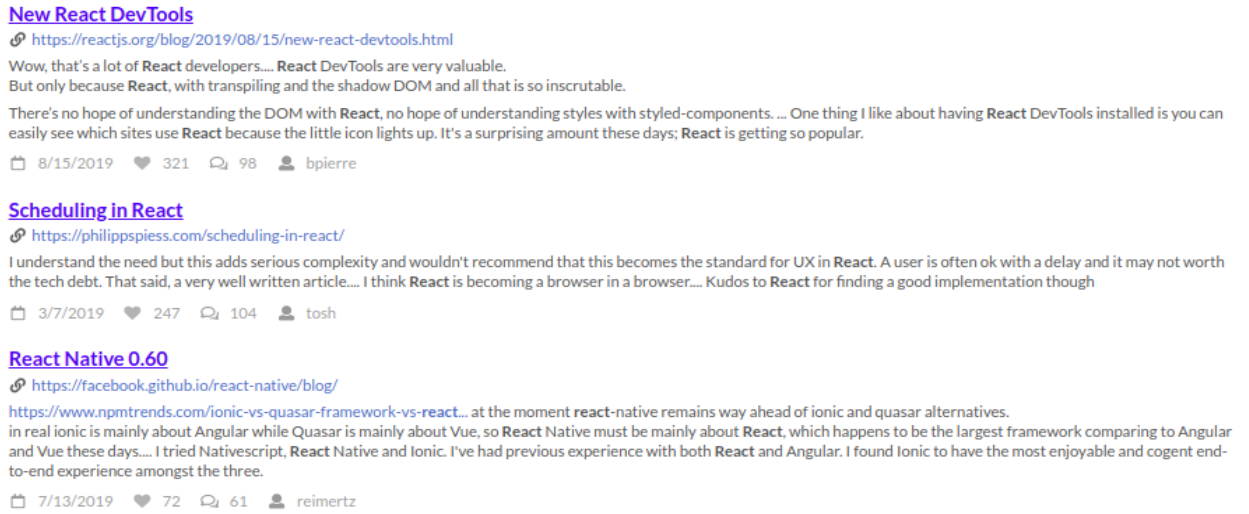

FIG. 8. My search engine results.

tions and therefore it would would a higher average than a user that has been publishing for a long time but its average popularity its somehow lower. After doing some research I discovered an interesting metric used in the scientific community called *h-index*, this metric covers both the number of publications a user made, and the impact of those publications (in number of votes). Basically it means that a particular author has published n posts that have each voted at least n times. To implement this I created a new core called *hn_authors_core* and used a Solr function called *join* (similar to the SQL one) to filter the results based on a minimum *h-index* of the corresponding author. After analyzing the results I noticed that it was not why I expected since the users are already filtering the good posts by voting and commenting it, and it does not matter how much the author has contributed to the community but how interesting the posts are. Therefore, I aban-

doned the idea of using this metric.

Another observation and improvement that can be done in the future is to find a way of creating a snippet from the URL a story points to. This requires finding a strategy to crawl on the website and generating the summary based on the query string. But 9.67% of the posts do not possess a URL, so in those cases we should consider to avoid displaying a snippet at all.

X. BIBLIOGRAPHY

- Berryman, J., & Turnbull, D. (2016). *Relevant Search: With applications for Solr and Elasticsearch.* Manning Publications.

- Mitchell, M., Parisa, K., Pugh, E., & Smiley, D. (2015). Apache Solr *Enterprise Search Server* (3rd ed.). Packt Publishing.

- Hearst, M., & Hearst, M. (2009). CH. 5: PRESENTATION OF SEARCH RESULTS. In *Search user interfaces.* Cambridge: Cambridge University Press.