HackerSearch: an Information Retrieval system

Ana Barros
FEUP
up201806593@edu.fe.up.pt

João Costa
FEUP
up201806560@edu.fe.up.pt

João Martins
FEUP
up201806436@edu.fe.up.pt

ABSTRACT

This document describes the development of an Information Retrieval system which takes textual data, corresponding to the year 2018, collected from the “HackerNews” link aggregator and corresponding linked URLs. The following sections explain in detail the processes of collecting, processing and analyzing the collected data, alongside with describing the developed information search system. More specifically, this includes its indexing, query parsing and results evaluation.

KEYWORDS

datasets, information retrieval, search engine, HackerNews, Y Combinator, HackerSearch, data analysis, indexing, Solr, JSON

1 INTRODUCTION

“HackerNews” is a link aggregator, mainly used to share news and products. It focuses on tech-related topics, such as computer science, and entrepreneurship. “HackerNews” is backed by “Y Combinator” foundation, which funds early startups biannually[5].
The community in this website usually interacts by sharing links to articles (blogs, news websites, and product landing pages) and then, voting and commenting on them. What sets this community apart is the vast technical knowledge shown in most comments, giving them a high readability value which sometimes surpasses the value of the post discussed.

This article aims to describe the development of “HackerSearch”. This platform will serve as an intuitive and powerful search engine for “HackerNews” posts/content.
The report is divided in different sections, each describes different tasks completed in order to achieve the final goal for this project. The Data Collection section 2 describes the process of retrieving the data. The Data Processing section 3 describes the filtering and preparation of the gathered data. The Data Analysis section 5 lists the process of exploring the data with statistics, characterizing the final datasets and their properties, the conceptual model as well as the follow-up information needs in the data domain. The indexing process contains the definition of the schema used, documents structure and tokenizers, analyzers and filters, and it is described in the Indexing section 6. The performance of our system is discussed in the Evaluation section 8. The Search System section 10 describes and discusses features added to improve the usability of the tool, such as, spellchecking and suggestions based on input text.

2 DATA COLLECTION

“HackerNews” provides an *official public API*[9] to fetch website’s data. This data is output in **JSON format**, and the **curl** command line utility is used to fetch it.

2022.

2.1 Posts/Stories

There are three types of posts in “HackerNews”: **story**, **job**, and **poll**. There was a decision that our dataset would only contain stories, as those comprise around 99.99% of posts on the website. From now on, all references to post/story are interchangeable.

The API[9] does not provide a way to perform bulk downloads. In order to streamline the process of fetching data, a starting dataset[10] was used. This dataset contained the raw data of stories collected using the API[9]. It is fetched by cloning the GitHub repository using the git command-line utility. The data comes divided into several JSON files containing a single list with multiple objects representing stories. This was done to bypass GitHub’s limit on file size. Upon downloading, these lists merge into a single one using the **jq**[6] command-line utility, so the stories can be filtered more easily and stored in the **stories.json** file.
Given the large volume of data available, it was decided to only deal with information related to stories posted in 2018. This is equivalent to around 50000 stories.

2.2 Comments

The dataset mentioned above only contains data about stories. To complement this, the API[9] was used to download the data of the top two comments of each story. This translates to around 100000 comments.
It was possible to download the data by using curl with persistent connections, thus reducing the time expended establishing connections with the API[9], and multiple sub-processes, to parallelize work. In order to easily parallelize the work and make use of persistent connections in curl, it was necessary to use a file containing URLs as input. Using **jq**[6] (to select the IDs of the top comments) and **awk** (to build the URLs using the IDs that were being piped) helped to build the file. Afterwards, it was split info multiple files (one for each sub-process) using GNU’s split, so each sub-process could take care of their part of the URLs.

Jq[6] was used to concatenate all the downloaded comment data into a single JSON list stored in the **comments.json** file.

2.3 URL content

As stories without a link don’t really do well upon on “HackerNews”, most stories (90%) do not have textual content (apart from the title). Instead, the website incentivizes users to share URLs to the content they want to discuss/share. This limits the searchability of stories.

To work around this, the group developed a small program using Mozilla’s Readability tool[8] (part of the Firefox browser) to fetch the main content of the URLs linked by the stories in text form. This allows us to perform general web scraping in many websites with different layouts/content, which wouldn’t be feasible with a specialized ’web spider’.

This code uses **node-fetch**[11] to fetch the target website's data and **jsdom**[7] to feed this data to the readability tool[8].

Jq[6] selected the stories containing a URL, and GNU's split divided the information of these stories into multiple files (each used by a respective sub-process). A JSON object stored the website's extracted textual content alongside its post ID. Each sub-process stores the content it is responsible for in a file. The JSON objects in these files join into a single JSON list using **jq**[6] and stored into the **html_content.json** file.

3 DATA PROCESSING

During collection and before analysis, the data went through a pipeline, illustrated by *Figure 1*, where it was transformed and filtered.

3.1 Initial Processing

The processing and filtering processes described in this subsection happened during the data collection process.

In order to reduce the number of stories (while increasing their relevance), they were filtered out based on the following criteria:

- Stories deleted by their owner.
- Dead stories (flagged by the users): If a story has enough reports, it is deleted.
- Stories that 'failed': A story 'fails' when it scores lower than 5 points, thus not being able to reach the front page of the website. Every story starts with 1 point and gains 1 extra point for each vote it receives.

These filtering steps were all performed by a single **jq**[6] script that encompasses all conditions mentioned above.

Comments suffered a similar filtering process (using **jq**[6]): Removing dead and deleted comments. It should be noted that some less popular stories have less than 2 comments in the final dataset, because users might not have engaged enough in their discussion or all their comments don't appear in the final dataset out for the reasons above.

The content of the URLs of the stories isn't always useful as textual data: some URLs link to binary data (images, PDF files, videos, etc…) and others might lead to web pages that our tool isn't able to extract information from (e.g.: YouTube's website). In these cases where the URL doesn't lead to any useful information and the story does not have textual content, it was decided to 'drop' this from the dataset.
When the URL yields useful information (most cases), the yielded data passes through a compression process to remove repeated blank spaces and newlines, and filtered to remove non-printable characters. This is done using a series of piped **sed** and **GNU's tr** commands. These processes are important because the website data is frequently unstructured and large.

3.2 Post-Processing

After gathering all the data, it was necessary to perform some final cleanups and save it in a more fitting format: at this point, the data is in three JSON files which needed to be unified.

The 3 JSON files containing the data turned into python's **pandas[13] DataFrames** to be worked on. The steps described below use the **pandas[13] and NumPy[12]** modules in python3.

All the stories in the dataset had a **type** field with the value **story**. This happened because the dataset contained only posts of that type (as mentioned in the Posts/Stories Data collection subsection 2.1. This field turned into a new one of the same name that divides the stories into the dataset in four categories, recognized by the "HackerNews" website, that can be filtered in their tabs. These categories are:

- **LaunchHN** - these are stories about new "Y Combinator" backed start-ups. This type of stories start their title with *"Launch HN:"*.
- **AskHN** - these are questions to the website's community. This type of story can't have a URL, and their title almost always starts with *"Ask HN:"*. Every story that doesn't have a URL falls in this category.
- **ShowHN** - these are stories that usually want to share a product landing/main page. This category is very similar to the **Normal** category discussed below, being only distinguished by their title starting with *"Show HN:"* and thus being shown in their filtered page.
- **Normal** - this is the category where most stories fall under. These always have a URL and rarely contain a textual description. Most time these link to news and scientific articles, or blog posts.

The percentage distribution of each of these categories in the final dataset will be discussed below in the Data Analysis section 5.

Both the story data and the comments contain a **'kids'** field. This field is a list of the direct descendants of the respective story/comment. In the case of the stories, their data also contains a **descendants** field that represents the number of child comments in total (across all nesting levels).
The group decided that the **'kids'** field has no value for the final dataset and, in the case of the comments, should be used to derive a **descendants** field before being dropped. This solution is not optimal as it doesn't take into account the number of child comments across all nesting levels, but only the top-level. It was still decided to use this solution, as it was infeasible to fetch the data of every comment (over 2.5 million) of the chosen stories in order to count the number of child comments.

Some data acquired from the "HackerNews" API[9], namely the **text** fields of both the stories and the comments, contained HTML special chars escapes. These were 'unescaped' so they could be stored cleanly.

4 CONCEPTUAL MODEL

After the gathering and passing through the data processing pipeline, the final data on the **pandas' DataFrames** was exported to a sqlite3 [15] database (described in this section), using the **sqlalchemy python module**.

For the data domain, the team created a conceptual model (Figure 2) in order to understand the data organization. There are four tables: *Type, Story, Comment*, and *URL*. The main attributes of the *Story* table are:

- story_by: the username of the user who posted the story.
- story_descendants: the number of comments of the story.
- story_score: the score of the post (a sum of its upvotes).
- story_time: the (UNIX) timestamp of the story posting.

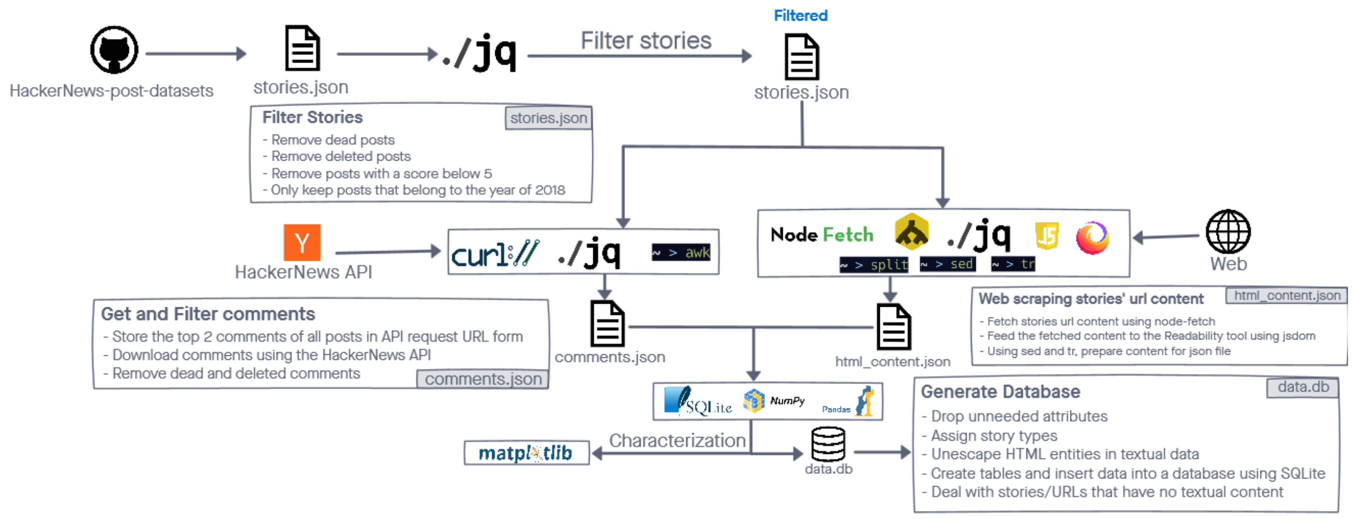

HackerNews-post-datasets
stories.json
./jq
Filter stories
Filtered
stories.json
Filter Stories
stories.json
- Remove dead posts
- Remove deleted posts
- Remove posts with a score below 5
- Only keep posts that belong to the year of 2018
HackerNews API
curl:// ./jq
~ > awk
Node Fetch ./jq
~ > split
~ > sed
~ > tr
Web
Get and Filter comments
- Store the top 2 comments of all posts in API request URL form
- Download comments using the HackerNews API
- Remove dead and deleted comments
comments.json
comments.json
html_content.json
Web scraping stories' url content
html_content.json
- Fetch stories url content using node-fetch
- Feed the fetched content to the Readability tool using jsdom
- Using sed and tr, prepare content for json file
SQLite
NumPy
Pandas
Generate Database
data.db
- Drop unneeded attributes
- Assign story types
- Unescape HTML entities in textual data
- Create tables and insert data into a database using SQLite
- Deal with stories/URLs that have no textual content
matplotlib
Characterization
data.db

Figure 1: Dataflow Diagram

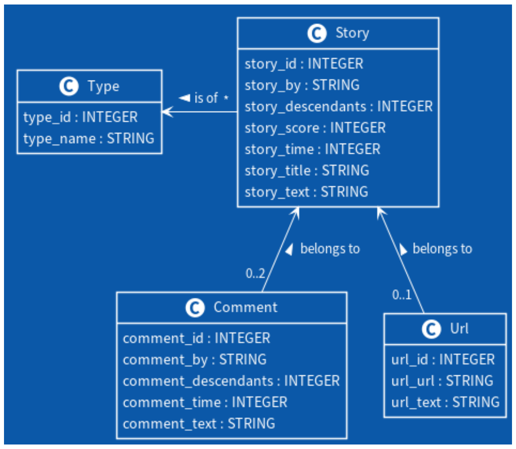


Figure 2: Conceptual Model

- story_title: the title of the story.
- story_text: the textual content of the story.
- story_type: the type of the story (foreign key to the *Type* table).

The *Type* class exists in order to represent the type of story. All stories have a type. This class contains only two attributes:

- type_id: the integer that identifies a story type.
- type_name: the name of the type.

The *Url* class represents the textual data obtained from URLs that are linked inside a story. Not every story needs to have a URL. However, a story cannot have more than one URL. The main attributes of this table are:

- url_url: the URL of the resource.
- url_text: the filtered textual contents of the web resource the URL links to.

Finally, the *Comment* table represents a comment of a story. Each comment only belongs to one story. Each story might have one, two, or no comments. These are its main attributes:

- comment_by: the user who posted the comment.
- comment_descendants: the number of replies of a comment (as described in the Post Processing subsection 3.2.
- comment_time: the (UNIX) timestamp of when comment was posted.
- comment_text: the content of the parent story.

5 DATA CHARACTERIZATION/ANALYSIS

This section describes the data characterization using graphics, plots, and textual descriptions.

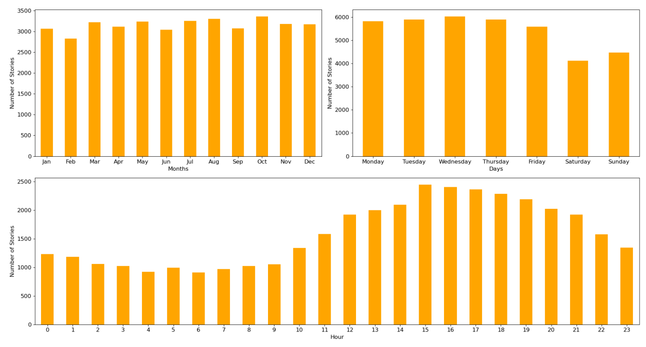


Figure 3: Number of stories per time

By analyzing Figure 3, it can be concluded that there is a higher frequency of story posts during the afternoon. Throughout the week, posts are more frequent during the weekdays as in contrast with the weekend. Finally, there is not a particular time of the year when the number of stories posted differentiates from the rest.

There is a unbalanced distribution of posts based on type, as evidenced by Figure 4. As described previously, there are four categories of stories: “Normal”, “AskHN”, “ShowHN”, and “LaunchHN”. The reality is that 85, 5% of all posts (which translates to almost 33000 posts) are classified as “Normal” posts while there are less than 100 posts classified as “LaunchHN”. This is expected considering that most posts on “HackerNews” intend on sharing a news article or blog post (most common category) and “LaunchHN” posts are a special category for new Y Combinator startups (less common category). The other two categories fall somewhere in between.

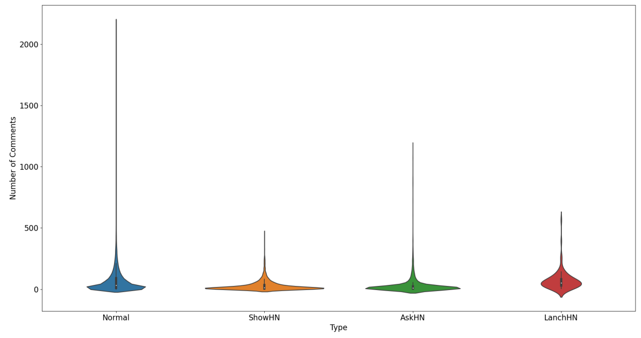


Figure 4: Number of comments per type

‘Normal” stories have a higher ceiling for number of comments (followed by “AskHN”), but the median number of comments is roughly the same for all story types. This is surprising considering that stories that do not contain a URL (the case of “AskHN”) are penalized in terms of exposition, requiring more engagement from users in order to reach the front page (succeeding).

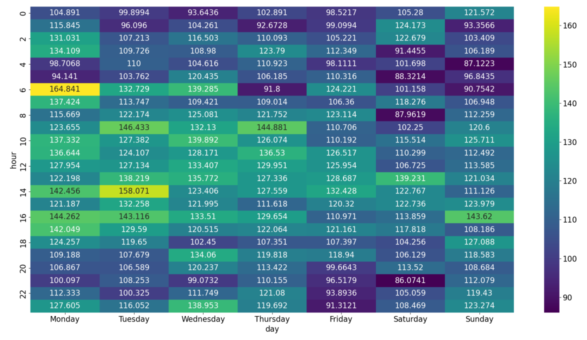


Figure 5: Average score in reference to time and day of the week

By analyzing Figure 5, a heatmap that relates the score of a story with the day and hour of when it was posted, it can be concluded that stories posted at the beginning of the week (Monday, Tuesday, Wednesday) and in the middle of the day (10AM-4PM), achieve higher scores. The cells with a higher average happen in stories posted between 6AM and 2PM on a Monday and from 9AM to 6PM on a Tuesday. The high discrepancy between the cells with the highest score values and its neighboring cells can be explained by the fact that most outliers occur within these timeframes. For example, one of the highest rated stories (2776 up-votes) was posted on Wednesday at 11PM, which has an average score of 138, while its neighboring cells have a score of 111 and 100.

It can be concluded by analyzing Figure 6 that the average score of a story can be related to the number of comments it has (user engagement). The trend is that a higher number of comments leads to a higher score. Stories with the highest scores appear between the interval of 750 and 1000 number of comments.

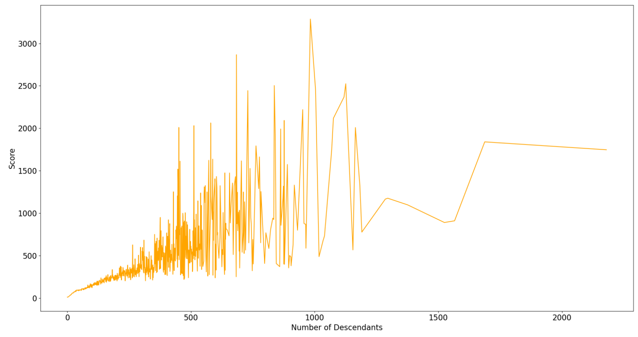


Figure 6: Score of story per number of comments

6 INDEXING PROCESS

The indexing process consists of the preparation of the documents, the creation of a collection and the definition of the schema. For this task, the chosen information retrieval tool was “Solr”[14].

6.1 Documents

As previously mentioned, the group created a SQLite[15] database containing the data after all the processing. However, in order to import the data to the chosen information retrieval tool, the database was transformed into a single JSON file (documents). This meant performing a join of all tables shown in the conceptual model (*Figure2*). The relation between Story and Comment is one-to-many. With this, preforming a join of these tables would result in duplicated entries in the search results (one for each comment on a relevant story). As such, the comments of each story are sub-documents of the story document.
The group converted the UNIX timestamps in the database to time strings, so they could be used in Solr’s range queries.

A document contains all information about the story it refers to (title, text, URL, etc…), the scrapped content of the website it links to (if any), and a sub-document for the story’s comments. The most import field of the comment’s document is its content (the comment’s text). These documents, compose the *hackersearch* collection.

6.2 Schema definition: indexed fields

The structure of the schema can be analyzed in Table 1. Each entry represents a field of the document: one cell indicating the type of the field, and another whether it is indexed or not. The latter was a decision made based on the search tasks defined in the Search Tasks section 7.1: fields that aren’t indexed, can’t be searched on (this includes filtering and faceting tasks).

The fields of the comment’s sub-document are present on the schema table and prefixed by *comments.*. The two last fields are Solr’s copy-fields.

6.3 Schema definition: copy-fields

Table 2 contains the copy-fields’ definition: the *source* and the *destination* of the copy operation.

The *search* field is the default searching field: when no field to search on is defined, all fields copied to the *search* field will

Field	Type	Indexed
story_id	int	false
story_author	singleToken	false
story_descendants	int	true
story_score	int	true
story_time	date	true
story_title	text	true
story_text	text	true
story_type	category	true
url	url	true
url_text	text	true
comments.comment_id	int	false
comments.comment_author	singleToken	false
comments.comment_descendants	int	false
comments.comment_time	date	false
comments.comment_text	text	true
newssite_filter	newsfilter	true
search	text	true

Table 1: Document Fields

Source	Dest
url	newssite_filter
story_type	search
story_title	search
story_text	search
url	search
url_text	search
comments.comment_text	search

Table 2: Copy Fields

be searched on. The *newssite_filter* field filters news stories from non-news stories.

6.4 Schema Definition: analyzers, tokenizers, and filters

In Solr, each field has a field-type that defines how they are processed in the indexing and querying processes. This is achieved with the use of tokenizers and filters:

- **title** – The type of the stories' titles. Very similar to the **text** type.
- **text** – The type of all text fields (story text, URL content, comment text, and search).
- **url** – The type of the stories' URLs. Allows the user to focus their search on specific domains.
- **newsfilter** – The type of the *newssite_filter* field. Allows the user to tailor their results towards news or non-news content.
- **category** – The type of the stories' categories.
- **singleToken** – The type of text fields that aren't indexed (e.g.: a story's author).
- **date** – The type of fields representing points in time. Useful for sorting/filtering for recent stories.
- **int** – The type of integer fields (e.g.: the number of descendants of a story).

6.4.1 Field-type: text. Stream treatment with this field-type is the same during index and query time. Before being passed to the tokenizer, fields of this type pass through a **charFilter**. This filter removes HTML tags from the textual content of the field, for example: `<a href=https://solr.apache.org>Solr</a>` becomes `Solr`. This is important because, although the content obtained using the "HackerNews API" was stripped of its HTML special entities, it still contains HTML tags.

Even after passing through the **charFilter**, these contents can still contain URLs. In order to deal with this, the tokenizer used is the *UAX29URLEmailTokenizerFactory*. This tokenizer identifies special tokens, like URLs and email addresses, and leaves each of them as a single token. Besides the special treatment described above, this tokenizer splits tokens on whitespace, punctuation, and hyphens.

The filtering pipeline starts after the tokenization process of the string:

- The filter converts non ASCII characters to their ASCII equivalents, if one exists. Since the language of the website is exclusively English, this helps deal with typos/rare special terms. This also discards the originals.
- Characters change to lower case.
- There is a synonym filter to increase the detection of related topic/words. As of now, this filter is non-exhaustive and only used as a proof of concept of what is possible. It appends occurrences of the phrase **version control** with names of version control systems, like **git**, **svn**, and names of products/companies, like **GitHub** and **GitLab**.
- English words reduce to their root form and possessive cases disappear.
- Removal of stop words (e.g. the, a, an, etc…).

6.4.2 Field-type: title. Stream treatment with this field-type is the same during index and query time. This field-type is very similar to the **text field-type** section 6.4.1. The main differences are the tokenizer used and some filters. Since titles in "HackerNews" can't contain URLs or email addresses, the used tokenizer splits tokens on whitespace, punctuation, and **@** characters. In contrast with the **text field-type** section 6.4.1, stop words in the title of a story are not filtered. Also, the original tokens are kept when converted to ASCII: stylized product names often appear in the title.

6.4.3 Field-type: url. When searching on the *url* field, the focus is on the domain of the URL. During index time, the tokenizer creates one token for each (sub-)domain of the URL. During query time, the tokenizer splits the queries as standard (on whitespace, punctuation, etc…), and the filters convert all characters to lower-case, so it can match the (sub-)domains, which are all lower-case as well.

6.4.4 Field-type: newsfilter. During index time, the tokenizer of this field-type extracts the domain (not the subdomains) of a URL. Afterwards, the filters keep only the domains present on a news website list. The tokens output from this process can only be **empty** (not a news website) or the word **news** (a news website).

During query time, the tokenizer splits the stream as standard (on whitespace, punctuation, etc…). Afterwards, only the tokens **new** and **news** will match with the URLs of news-related websites.

This field allows for a dynamic list of news content domains (news networks, journalism website, etc…) and for users to know how many results in their query are news-related. Furthermore, this field allows the user to apply a filter to their search, so only news websites or non-news websites show up.

6.4.5 Field-type: category. During index time, the categories stay as a single token and change to lower-case. During query time, it splits the stream as standard and converts it to lower-case as well. This allows the user to filter results by their categories, and the system to count the number of results that fall on each category.

6.4.6 Field-type: other. The last three field types are just *wrappers* around built-in Solr[14] field types, without any defined tokenizers or filters:

- **singleToken** – solr.StrField – text fields that aren't searched on.
- **date** – solr.TrieDateField – fields that represent points in time.
- **int** – solr.IntPointField – fields that represent integer numbers.

7 INFORMATION RETRIEVAL

7.1 Search Tasks

Users of this information retrieval tool are able to search for items fitting a description. For example, when searching for "cli work with JSON", the tool would retrieve results talking about/showcasing jq[6] (the tool used to process the JSON files that power "HackerSearch"). An option to separate news related content from technical blog posts could also improve the system., e.g., searching for "github" leading to version control system usage/guides instead of update news and policies changes of GitHub. This could be done by keeping a list of news/journalism websites. In addition, ranking based on popularity, how active the story is, and time is desirable. If possible, the meaning of words should be associated with the context.

7.2 Query demo

Query intention. As a user, I want to see the most recent news stories about Hong Kong. The ones that interest me are the highest rated ones. I don't want to see anything related to China or its relation with Hong Kong.

Query. `"Hong Kong" +(newssite_filter:news) -China`

Sort field. `story_time desc, story_score desc`

Number of results found. 27

Field	Weight
story_type	10
story_title	3
story_text	3
url	2
url_text	1
comments.comment_text	1

Table 3: Field weight

8 EVALUATION

This section describes four queries that display the capabilities of the designed system, and evaluates them. Three different systems (or more/less in some cases) run each query:

- **schemaless** – system with only the **search** field described in the schema section above, without any filters.
- **schema** – system as described in the schema section.
- **weighted schema** – the same as schema, but the search fields have weights as described on the weight table 3.

8.1 Precision Metrics

The followed evaluation process consists of calculating, for each query, a set of precision metrics. To prevent result redundancy, only the systems deemed relevant were chosen, for example, news filtering isn't relevant/possible on the **schemaless** system as it doesn't have the required fields.

For each query and correspondent relevant systems, each metric uses a universe composed of the **top 20** results obtained by the query/system pair. The set of relevant and irrelevant documents used are built with the top 10 results.
The chosen metrics to describe the query results are as follows:

- **Recall@N** - Ratio between the number of retrieved relevant documents and the total number of relevant documents in the *universe* for the N first retrieved documents.
- **Precision@N** - Ratio between the number of retrieved relevant documents and the total number of retrieved documents for the N first retrieved documents.
- **Average Precision (AP)** - The mean of the precision values calculated for the first N documents.
- **F$_\beta$Score** - Calculated with precision (P), recall (R), and β (controls the amount in which the recall is more important than the precision).

$$(1+\beta^2)\cdot\frac{P\cdot R}{(\beta^2\cdot P)+R}$$

  This is useful to verify the balance between the number of relevant documents missed and the number of retrieved irrelevant documents [16].

For the problem at hand, the group decided that a good baseline is to consider the precision as twice as important as the recall, i.e., a β value of 0.5. This is justified by the fact that, in the context of "HackerNews"[5], a user wants to find the first relevant matches with very few irrelevant documents, instead of all relevant matches with some irrelevant documents. Still, the recall metric is valuable, as the system must be able to retrieve an ample part of all relevant documents to provide the user with diverse results.

8.2 Queries

8.2.1 Eclipse IDE query.

Relevance Definition. The goal of this query is to find posts that talk to the Eclipse IDE or related plugins/tools.

Query. eclipse

Results. This query is intentionally ambiguous. Due to this ambiguity, almost all retrieved documents aren't relevant. The majority discuss astronomy and the solar/lunar eclipse phenomena. Others simply use the verb *eclipse* as a way to refer to something that has fallen into disuse. In the end, very few documents reference the Eclipse IDE across all systems, as can be confirmed by the PR-Curve Figure 7. The relevancy results can be found in Table 4, and the metric results can be found in Table 5.

System	Relevant Documents									
System A	N	N	N	N	N	N	N	N	N	R
System B	N	N	N	R	N	R	R	N	N	N
System C	R	N	R	N	R	N	N	N	R	N

Table 4: Eclipse query relevancy results

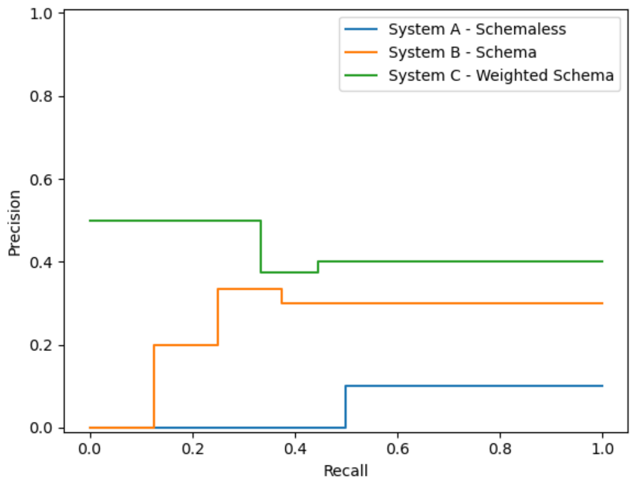


Figure 7: Eclipse query precision recall curve

System	AP	P@10	R@10	F_β
Schemaless	1.00%	10.00%	50.00%	11.90%
Schema	22.20%	30.00%	37.50%	31.25%
Weighted Schema	54.15%	40.00%	44.44%	40.82%

Table 5: Eclipse Query Results

8.2.2 *JSON command line tools.*

Relevance Definition. This query intends to retrieve all posts that showcase tools that manage files/streams in the JSON format, on the command line.

Query. +json tool "command line"

Results. System B achieved better average precision than system A, with a difference of 20%. The usage of the term "command line" seems to reduce the number of irrelevant documents retrieved on all systems, as the usage of the words command and line is relatively common. In fact, the group found that the results of the same query with these terms matching separately proved to produce significantly worse results (reducing precision and recall by 20% in all systems). Relevancy results are represented in Table 6, PR-Curve in Figure 8 and metric results in Table 7.

System	Relevant Documents									
System A	R	R	R	N	N	N	R	R	N	R
System B	R	R	R	R	R	R	N	R	N	N
System C	R	R	R	R	N	R	R	N	N	N

Table 6: JSON command line query relevancy results

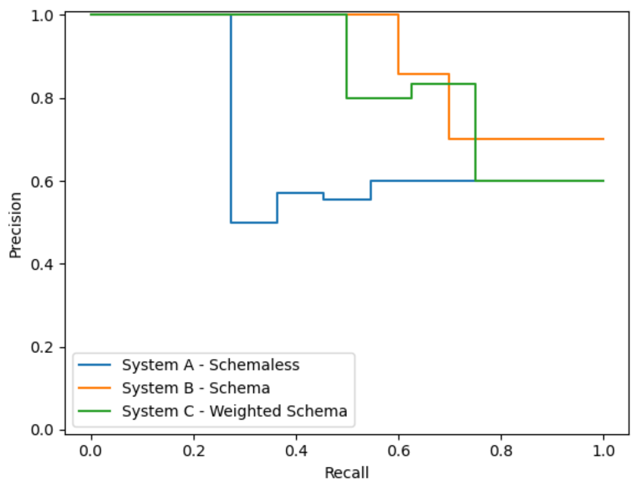


Figure 8: JSON command line query precision recall curve

System	AP	P@10	R@10	F_β
Schemaless	72.02%	60.00%	54.55%	58.82%
Schema	92.10%	70.00%	70.00%	70.00%
Weighted Schema	85.07%	60.00%	75.00%	62.50%

Table 7: JSON Query Results

8.2.3 *Version Control Tools.*

Relevance Definition. This query aims to discover posts that showcase tools for software version control or news related to the topic.

Query. +"version control" tools -svn

Results. All systems seem to produce similar average precision results, while portraying a significant disparity between their recall values, as can be seen below, Figure 9. The majority of results found refer to GitHub or GitLab news. Both relevancy results and metric results can be found in Tables 8 and 9, respectively.

System	Relevant Documents									
System A	R	R	R	R	R	R	N	N	R	N
System B	R	R	R	N	R	R	R	R	R	R
System C	R	R	N	R	R	R	R	R	R	R
System D	R	R	R	R	R	N	R	N	N	N

Table 8: Version control query relevancy results

8.2.4 *News filtering.*

Relevance Definition. This query aims to find stories from news websites discussing topics relating to China.

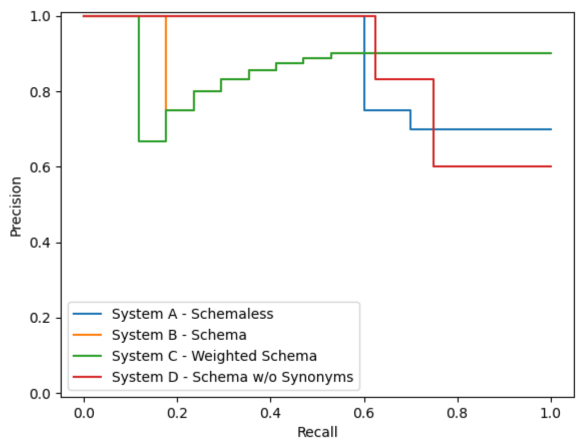


Figure 9: Version control query precision recall curve

System	AP	P@10	R@10	F_β
Schemaless	90.85%	70.00%	70.00%	70.00%
Schema	89.04%	90.00%	52.94%	78.95%
Weighted Schema	85.71%	90.00%	52.94%	78.95%
Schema w/o synonyms	87.07%	60.00%	75.00%	62.50%

Table 9: Version Control Query Results

Query: `China +(newssite\_filter:news)`

Results. The intention of this query is showcasing the news filter effectiveness. As such, only the **weighted schema** performed the query; once with the news filter, and once without the filter. As shown on the PR-Curve Figure 10, when using the filter, the system manages to only find relevant documents, while the other system only finds one relevant document. When not using the filter, most stories are personal blog posts or guide discussing how to optimize content for the Chinese market, and personal takes on Chinese policies/culture. The relevancy results can be found in Table 10 and the metric results are in Table 11.

System	Relevant Documents									
System A	N	N	N	N	N	N	N	N	N	R
System B	R	R	R	R	R	R	R	R	R	R

Table 10: News query relevancy results

System	AP	P@10	R@10	F_β
Schema w/ news filter	2.11%	10.00%	11.11%	10.20%
Schema w/o news filter	100.00%	100.00%	50.00%	83.33%

Table 11: China Query Results

9 REVISIONS INTRODUCED

This section includes all revisions that were made in the context of the third milestone.

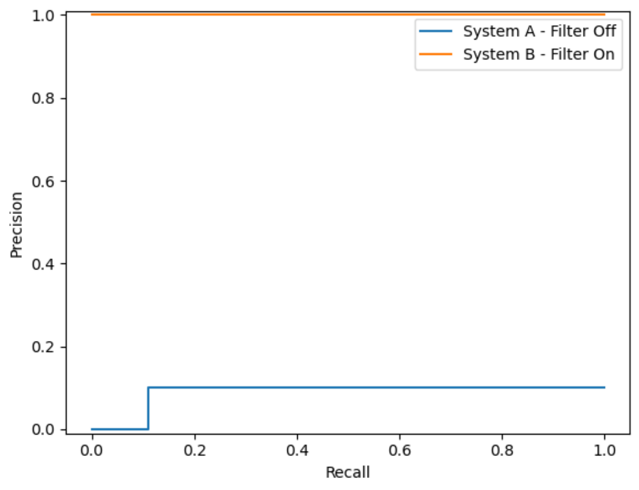


Figure 10: News query precision recall curve

9.1 Table and Figure References

Links to previously unreferenced tables or figures were added to the following sections:

- Query subsection - Relevancy tables, PR-Curves and metric result tables
- Data Characterization section - Plots regarding story and comment characterization

9.2 Mean Average Precision

Additional to the already mentioned precision metrics, the group also calculated the Mean Average Precision metric, found in Table ??. This metric corresponds to the mean of all calculated **AP**s of each query.

System	MAP (%)
Schemaless	54.62%
Schema	67.78%
Weighted Schema	74.98%

9.3 Relevant Documents List

In addition, the evaluation section of each query was updated with a list of relevant and irrelevant found in each analysis.

9.4 Future work update

Updated the Future work and improvements, section 11, to reflect the progress done: removed points that were concluded, added new objectives.

10 SEARCH SYSTEM

This section reflects the addition of extra features, improvements, and the design of a frontend for the search system.

10.1 Frontend

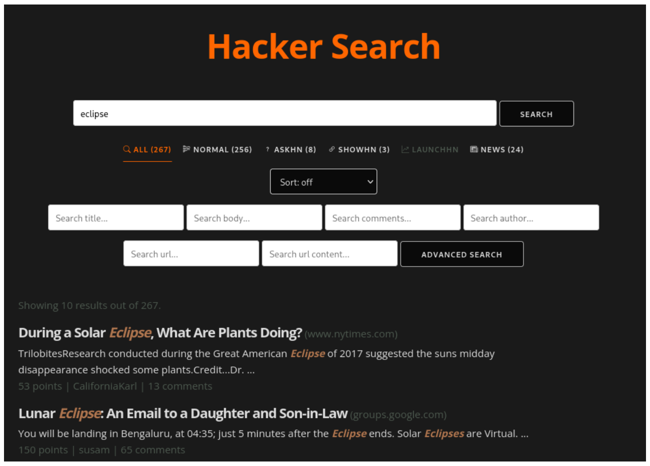
We implemented a simple graphical interface, in order to demonstrate the new abilities of the system being developed. This interface (frontend) interacts with the Solr backend by calls to its REST API. An example page of the frontend can be found in Figure 11

The interface contains a search bar that takes user input to make a query using the *search* field described in previous sections (searches in all fields). If the user wishes to, there is also the ability to do an *advanced search*. This allows the user to query different fields at the same time.

The number of results for the query are shown below along with the results. This also plays in the faceting mechanism that will be described later. In order to avoid loading thousands of results at the same time, the system uses pagination to load 10 stories at a time. The user may issue a request to load extra results, if there are any. The matching terms in the user query present in the documents are highlighted, so the user can more easily choose which hits are relevant for them.

Again related to the faceting mechanism, results can be filtered by type. There is also the ability to sort the results based on date and score.

While writing, users are shown progressive word-completion suggestions and possible matching entries. After the user finishes their query, they can be shown spellchecking suggestions (if there are any). These will be described in detail later.



Showing 10 results out of 267.

During a Solar *Eclipse*, What Are Plants Doing? (www.nytimes.com)
TrilobitesResearch conducted during the Great American *Eclipse* of 2017 suggested the suns midday disappearance shocked some plants.Credit...Dr. ...
53 points | CaliforniaKarl | 13 comments

Lunar *Eclipse*: An Email to a Daughter and Son-in-Law (groups.google.com)
You will be landing in Bengaluru, at 04:35; just 5 minutes after the *Eclipse* ends. Solar *Eclipses* are Virtual. ...
150 points | susam | 65 comments

Figure 11: HackerSearch Frontend

10.2 Spellchecking

Solr provides the *SpellCheckComponent* for spellchecking [3]. To use it, the group added a new field created for spellchecking, *spell*. The *spell* field copies terms from the *story_title* field, and is of a new type called *spellField*. With this field-type, tokens are delimited by whitespace and punctuation. Its filters convert all non-ASCII characters to their ASCII equivalents and turn upper-case characters to lower-case. Finally, it strips any English possessive cases from the tokens.

With this field and field-type in place, the spell check component can be used. A new search component was created using the *IndexBasedSpellChecker* implementation. The only other feasible implementation option available was the *DirectSolrSpellChecker*. The advantage of this alternative is that it doesn't use a Solr index to provide its results, so it doesn't need to rebuild the index when new documents appear [3]. This isn't useful for our project, because the index only needs to be build once (no new documents added), so the speed benefits of the *IndexBasedSpellChecker* are very important. It also allows us to do more complex analysis (something very inefficient on the alternative implementation) on the field used for spellchecking, because the analysis happens only once (at index time).

This new search component is part of the default one, so every query can make use of spell checking suggestion, unless otherwise specified. In addition, in order to make it easier to work with the suggestions, the *collation* mechanism is also used by default. This mechanism instructs Solr to take the best suggestion (if any) for each misspelled word, and return a new query with those as replacements for the misspelled ones. Although rare, these collations aren't verified, so they can return no results when chosen.

Table 12 shows example usages of the developed spellchecking system.

Query	Spellchecking
eclips	eclipse
luanr	lunar
vrsion cntrol	version control

Table 12: Spellchecking result examples

We updated the request handler available at */spell* to make use of these new spellchecking capabilities.

10.3 Suggestions

Solr provides the *SuggestComponent* [4] in order to support suggestions for query items. This provides current word completion and helps highlight possible matching documents. To be able to provide these features, our systems contains two different components that feature two distinct approaches to suggestions.

The first one consists of the suggestions of possible hits for the current query (the one that the user is currently typing). These hits are based on the title of the documents. It uses the *AnalyzingInfixLookupFactory* which analyzes the current word that the user is inputting and then suggests matches based on prefix matches to all tokens in the text [4]. A small suggestion sample for a small amount of queries can be found on Table 13.

Query	Top suggestions
regain	…Autopilot automatically regained…
fin	Ask HN: How do you find roles as a solo developer?

Table 13: Document suggestions

The second one consists of suggestions for current word completion. It uses the *FreeTextLookupFactory* which looks at the previous tokens and the one the user is currently typing, to predict what the user might want to type next. It looks at the last 2 tokens (using **ngrams**). On Table 14, there are examples of word completion suggestions for a number of queries. The last two show that the results also base themselves on the previous tokens in the query: the term **regained** appears next to the term **automatically** in one story title, so it is an exact match.

Query	Suggestions
i miss y	you your years year yc
r	rust real run remote report
automatically r	automatically·regained rust real run remote

Table 14: Word completion suggestions

Both implementations store their index in a *DocumentDictionaryFactory*, which is the most efficient dictionary implementation in Solr that supports our use case [4].

There is a limit of 8 for the maximum of suggestions provided. In addition, the group added a new field-type, *suggestion_type*, to the schema. Stream treatment with this field-type is the same during index and query time. The tokenizer used splits tokens on whitespace and punctuation. The filters then convert tokens to lower case and deletes all the non-alpha-numeric characters. This is useful because when searching, users rarely input characters that aren't letters or numbers. The new field *sugg* uses this field-type, and it is used as a destination for the copy operation of the *story_title* field.

A new request handler, available at */suggest*, exposes both of these suggestion mechanisms.

10.4 Faceting

Faceting can be used to split results into different types of categories. Solr supports this through the use of the *facet.query* and *facet.field* query parameters [1]. This allows the system to arrange search results into buckets corresponding to the different story types: **AskHN**, **AskHN**, **ShowHN**, and **Normal**. Additionally, it is also possible to separate stories that are **news** from those that aren't. We use the following parameters on each query to achieve both effects: *"facet.query": "newssite_filter:news"*, and *"facet.field": "story_type"*.

With this, the frontend provides the user with the count of documents that fall in each category. Furthermore, the user can filter the results to see only the ones that fall on a given bucket by clicking the appropriate bucket, as is shown in Figure 11.

Implementing **pivot (decision tree) faceting** was considered, but the group didn't find any useful use cases for it.

10.5 Highlighting

The group made use of Solr's *HighlighterComponent*, which helps to stand out the matched content of a query. The highlighting method that produced the best results is the *Unified Highlighter*, since it displays the actual Lucene matches [2].

The frontend uses the highlighted contents in the query results, which are surrounded by special HTML tags, to highlight all relevant matches to the user.

An example use of highlighting in the frontend can be found in Figure 11.

10.6 Original keyword priority over its synonyms

When using synonym lists (as described earlier), we were faced with a problem: the original keyword searched was sometimes thrown to the bottom of the results, because the synonyms were more popular.

In order to overcome this, we employed a special weight system for the synonyms. The original keyword is boosted by 2.0. For example, when searching for *git*, the synonyms *version control*, *github*, and *gitlab* wouldn't be boosted, while the term *git* would be boosted by 2.0. This means that we consider the original keyword to be twice as important as its synonyms.

This also meant that the synonym system was expanded further than what was described in earlier sections: more synonyms were created, and the existing one (*version control*) had to be split into multiple synonym associations, so the boosting could be applied.

As seen in figure 12, the new system (**System E**) ranks above all previous systems for the related information need: finding version control tools that aren't SVN (*+"version control" tools -svn*). This indicates that the change is important. Table 15 contains the relevancy of retrieved documents for the new system: all documents found are relevant. With the new information in table 16, the system's Mean Average Precision (MAP) of the system got bumped up to: $(54.15 + 85.07 + 100)/3 \approx 0.7974$.

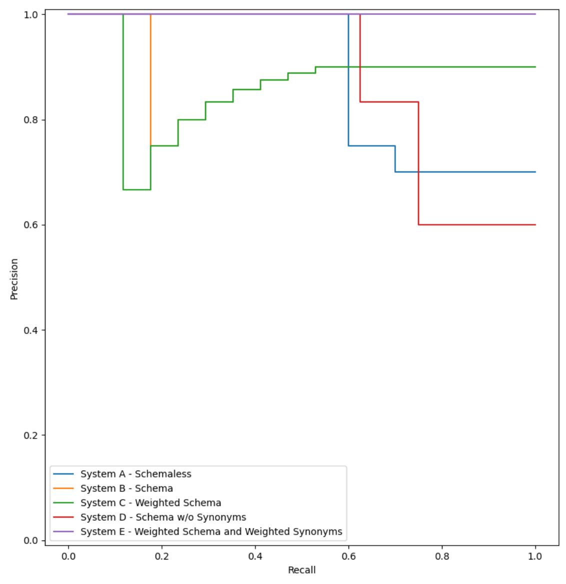


Figure 12: Version control query precision recall curve with weighted synonyms

System	Relevant Documents									
System A	R	R	R	R	R	R	N	N	R	N
System B	R	R	R	N	R	R	R	R	R	R
System C	R	R	N	R	R	R	R	R	R	R
System D	R	R	R	R	R	N	R	N	N	N
System E	R	R	R	R	R	R	R	R	R	R

Table 15: Version control query relevancy results

Even with these improvements, the top 10 results for the query still yield no results mentioning "version control" explicitly, only its

System	AP	P@10	R@10	F_β
Schemaless	90.85%	70.00%	70.00%	70.00%
Schema	89.04%	90.00%	52.94%	78.95%
Weighted Schema	85.71%	90.00%	52.94%	78.95%
Schema w/o synonyms	87.07%	60.00%	75.00%	62.50%
Weighted Synonyms	100.00%	100.00%	55.56%	86.21%

Table 16: Version Control Query Results

synonyms. This can be caused by almost no documents mentioning the term. Still, even though the results didn't improve in the context of the experience, the changes improved the relevancy of the results for the information need, which is a positive result.

10.7 More Like This

By employing Solr's *MoreLikeThis* feature, we created a new request handler at */mlt*. This request handler takes a query that matches a single document, for example ID matching, and returns documents similar to it. This allows users to find more stories relevant to their interests: find stories similar to the ones they like.

In order to compare stories, 4 fields are used: the title, the content, the type, and the URL's domain. The system generates term vectors for these 4 fields, so they can be compared more efficiently and accurately.

Table 17 shows the stop word frequency in story titles. Many stop words do not appear in titles very much. In fact, the most common stop word in story titles is **the**, and it appears in 20% of the documents. We don't want these words to be considered when calculating the similarity of the documents, so we employed two mechanisms to counter this: words shorter than 3 characters, and words that show up in more than 10% of the documents are discarded. This means that the more common stop words, like **and** and **the**, are not considered, and the less common stop words, which tend to be shorter (two characters or fewer), are also not considered for similarity calculation purposes.

Stop word	Frequency (%)
a	17%
and	10%
but	1%
how	5%
or	1%
what	4%
will	1%
the	21%
i	3%
if	1%

Table 17: Frequency of stop words in story titles

In table 18, there are two examples of stories and their most similar story, according to the system. We believe that the fields used for similarity calculation between stories are performing well, with the similar stories found being relevant

Story	Most similar
During a Solar **Eclipse**…	…yesterday's total lunar **eclipse**
Firefox Sideloaded Extensions	Blocking cryptominers Firefox

Table 18: Similar stories examples

11 FUTURE WORK AND IMPROVEMENTS

Regarding data collection/processing improvements, the dataset could be improved by collecting all types of posts from all time, collecting all comments (instead of just the top two of each story), and by finding the real descendants count of comments. Also, the web scrapping part of the pipeline should be able to handle all websites, e.g., YouTube.com by extracting the video description, and converting PDF files to text (processing binary data). It is also difficult to handle cases of 404 errors/missing content when the target website redirects to a custom page on those cases.

Regarding search improvements, the weighted system is performing better than other systems. Still, the weights are probably still not optimal. This can be iterated by further experimentation.

In the future, finding a way to identify stories that could be classified as *niche* would allow users to find results about newer, or unknown/unpopular tools and topics, when they wish to explore those areas.

12 CONCLUSION

The data collection/processing pipeline is autonomous and scales easily by dividing the collection between multiple processes. By using 4 processes, the group was able to retrieve and process a significant amount of information. We believe the pipeline presented could even be scaled to multiple machines, so the information could be retrieved and indexed in real time (*sharding*). Future improvements in **Mozilla's Readability** technology are likely to bring improvements to our data collection pipeline.

Although the data collected to feed the information retrieval system was only a small fraction of the real data available, it is already enough to demonstrate that the system yields some interesting results. Namely, the search results are frequently relevant, and the synonym system concept allows for many more relevant documents to be found and should be expanded further in the future. The various mechanisms around word and query completion/correction (spellchecking) help frontend/interface developers employ user-friendly ergonomic features, which make a great difference (as shown in the examples of previous sections). Furthermore, although the *More Like This* was implemented in a quite simple fashion, it already yields documents that appear very relevant and related to the topic at hand.

As of now, even with the considerable amount of data, both the indexing and retrieval processes are fast. This leaves us hopeful that the system is efficient and is fit to scale to more complex implementations/vast data.

REFERENCES

[1] Apache Software Foundation. 2022. Solr- Faceting (accessed on 15/1/2022). https://solr.apache.org/guide/8_10/faceting.html#field-value-faceting-parameters
[2] Apache Software Foundation. 2022. Solr- Highlighting (accessed on 15/1/2022). https://solr.apache.org/guide/8_10/highlighting.html#choosing-a-highlighter
[3] Apache Software Foundation. 2022. Solr- Spell Checking (accessed on 15/1/2022). https://solr.apache.org/guide/8_10/spell-checking.html

[4] Apache Software Foundation. 2022. Solr- Suggester (accessed on 15/1/2022). https://solr.apache.org/guide/8_10/suggester.html
[5] hackernews. 2021. hackernews (accessed on 12/12/2021). https://news.ycombinator.com/
[6] jq. 2021. jq (accessed on 20/11/2021). https://stedolan.github.io/jq/
[7] jsdom. 2021. jsdom (accessed on 20/11/2021). https://github.com/jsdom/jsdom
[8] Mozilla. 2021. Mozilla's Readability (accessed on 20/11/2021). https://github.com/mozilla/readability
[9] Hacker News. 2021. Hacker News API (accessed on 16/11/2021). https://github.com/HackerNews/API
[10] Masatoshi Nishimura. 2020. HackerNews Post Datasets (accessed on 17/11/2021). https://github.com/massanishi/hackernews-post-datasets
[11] node fetch. 2021. node-fetch (accessed on 20/11/2021). https://www.npmjs.com/package/node-fetch
[12] numpy. 2021. numpy (accessed on 20/11/2021). https://numpy.org/
[13] pandas. 2021. pandas (accessed on 20/11/2021). https://pandas.pydata.org/
[14] solr. 2021. solr (accessed on 11/12/2021). https://solr.apache.org/
[15] sql. 2021. sql (accessed on 21/11/2021). https://sqlite.org/index.html
[16] Wikipedia. 2021. F-score (accessed on 12/12/2021). https://en.wikipedia.org/wiki/F-score](https://en.wikipedia.org/wiki/F-score)