TTDS Group 57 Report

1 Abstract

Our group created a fan fiction search engine called “Story Hunter” for the fan fiction works hosted at “Archive of Our Own” (AO3) website. “Story Hunter” improves upon AO3’s in-built search by allowing to search for the full work’s text and supplying the user with a familiar and intuitive GUI. Our search engine allows to search within 875K documents (with an average length of 2600 terms) and provides logical, proximity, phrase and wild-card search functionalities, as well as the ability to search for work’s tags which can be auto-completed for the user. Documents are ranked based on the BM25L model. As for the GUI, it follows a minimalist, google-like design and offers additional features such as results’ pagination and query syntax suggestion for the user. There is a webscraping script which gathers the previous day’s stories which demonstrates that our search engine can be supplemented with new works and thus stay up to date.

2 Context: Introduction to AO3

2.1 AO3

Archive Of Our Own, also known as AO3, is a popular fan fiction hosting site. As of March 2023, it has over 5M users, who altogether have posted over 10M stories. Each story may consist of one or more chapters, and can range drastically in length, with 2.47 million stories less than 1000 words long, but 95 thousand that count over 100 thousand words. Each story additionally features metadata. The site’s metadata includes an extensive tag system, which categorise stories based on the relationships and characters present, the source material each work is based on, as well as specific tropes, or a contest that the story was submitted as part of. As well as tags, there are numerical features, showing the level of community engagement the story has received, and when the story was last updated.

2.2 Limitations to the Existing AO3 Search Engine

AO3 does provide a search functionality, but it is only designed to filter stories using metadata, not the stories themselves. It is a Boolean search engine, similar to that implemented for coursework 1. It can search the title, author and summary fields for specific text, both individual terms or phrases (but not proximity search), for stories containing (or lacking) specific tags, and apply numerical comparisons to the numerical statistics. It also supports some advanced search features like intra-term wild-card search, but again this feature only works for the data found in the story’s metadata.

Given the main limitation, namely AO3’s search engine not being able to search the text within the stories, the use cases of this search engine is limited. A typical user will probably use this search engine to filter specific tag combinations, and then sort results by how high their engagement levels are. However, it is important to note that, whilst many stories are heavily tagged, just as many have few or no tags. This means, that whilst the search engine is good when you have a type of story you want to read, and are willing to use popularity as a heuristic for quality, it severely lacks if you want to find a specific scene, or track down a favourite extract you’d once read. Another

limitation is that the GUI of the current search engine is unintuitive to use. This can be proven by search fields requiring additional explanation - many fields feature a “?” symbol which provide documentation on how to use the field. We believe that the search engine should be intuitive, and require only minimal explanation.

2.3 Our Dataset

We sourced the AO3 dataset from reddit.com/r/dataset/comments/i254cw/archiveofourown_dataset. It is 502GB in Size, and contains 15 million documents, or chapters, written in English, with an average length of 2600 terms. This is the complete collection of all publicly accessible chapters and their corresponding metadata published on the site before 17/07/2020.

Due to cost constraints, we have limited ourselves to using 875,000 chapters from this set. When combined with the metadata indexes, it approaches the limit of what we can support on our dedicated index server with around 100GB of RAM.

3 Our System

3.1 High Level System Architecture

Queries from the front end are sent to our GCP VM instance ”ttds-server” which will proxy TCP requests to the Search-engine client. The external port exposed to the internet is port 80 which listens through Apache for any TCP connections, then all data is proxied and reverse proxied to port 5000. The retrieval model receives index data from ”index-a” VM using a number of APIs, and generate a ranked list of results. The Instance VM will respond to the request with the relevant data which is collated by the retrieval model. Once collated it is sent to the front end to display to the user. See fig 1 in the appendix for a diagram of the full system architecture.

3.2 Indexing

3.2.1 Multiple Indexes

Our system makes use of five main indexes. The primary index is the chapter index. This contains every preprocessed term in the corpus, the chapters it occurs in, and the positions each term occurs within each chapter. The second index is the tag index. It is similarly structured to the chapter index, but is slightly simpler. For each tag in the corpus, it stores every story ID that uses that tag. Tags are case normalised, but no additional preprocessing is applied, allowing for users to easily search for specific tags. The metadata index contains the statistics for each story, such as engagement levels and the last update, as well as features to be used when previewing results, such as the title, author(s), and the story’s summary. The fourth index contains the term counts for each chapter. This is primarily used in the BM25L ranking model as we require fast access to document lengths to calculate the score for each document. The final index is the permuterm index organized in the trie structure and we use this index for wild-card search (see 3.3.4).

3.2.2 Index Compression

To minimise GCP costs, given the large size of our dataset, it was essential for our indexes to be compressed when saved to disk. As posting lists are monotonically increasing, we compressed each index using delta compression.

Our delta compression encoding and decoding consists of two main components: integer encoding/decoding and string encoding/decoding. Using these components, we built up specific compressors and decompressors for the chapter, tag and metadata indexes. Due to the fact that the metadata index does not store postings list, but a mix of numerical and textual information, it is not quite as effective a compression method as it is for the other indexes. However, the metadata index is dwarfed by the chapter index in size, so this is not a significant concern.

At the beginning of the project, our initial plan that would have allowed us to use the full 502GB dataset was to load chunks of the index in on demand to respond to queries. Whilst we were never able to get this to run at a speed that made the approach feasible, it did mean our index compression was highly optimised to be read into memory quickly. Achieving this involved both low level operations - for example, benchmarking found multiplying by 128 to be faster than bit shifting - and higher level approaches, such as inserting entire posting lists at once into the chapter index rather than a term at a time. When combined, we were able to improve our loading speed by a factor of ten over our initial implementation. Whilst we were able to compress our indexes when written to disk, whilst storing it in memory was a different story. Query processing speed is everything, so delta compression in the posting lists was not an option. Doing so would mean that parts of exact and proximity search would jump from O(lg n) time to O(n), as it would have to keep a running total of all postings to find a target, rather than a more efficient binary search. The one place that we were able to save space was in the metadata index. The summary field within the metadata index is not used for searching - when the index was initially created, the title, author(s), and summary are copied into the first chapter. The summary in the metadata index is held so that it can be sent to the front end as part of the preview shown to users. Therefore, whilst we cannot shoulder the cost of reading in dozens of summaries from file each query, we do store the summary in zipped form.

3.2.3 Index - Retrieval API

The initial idea was to create our own application layer on top of TCP, to get and post requests. However, as we wanted to create a multi request multi index system, it was vital that the system remained reliable, i.e., responses were sent to the correct request and that the entire response was sent through. Hence we decided to use FastAPI, a web application framework similar to Flask but designed to scale for a magnitude requests. FastAPI is designed to use the ASGI (Asynchronous Server Gateway Interface) standard, which allows it to spawn multi workers/processes and within each worker it then spawns several threads, allowing for multiprocessing. As operations are asynchronous and concurrent, fetching data from the server is faster than Flask. Additionally, if the index has to be modified, FastAPI's PUT request handling allows other threads to be locked whilst the changes are made. We created 4 server client systems for the 4 indices, namely, story metadata, term counts, chapter index and tags.

3.2.4 Indexing Limitations

In the interest of full disclosure, our final index is missing a few features we intended to have, and have code to include. To allow users to search for specific stories and authors, we append the title, author, and summary fields to the first chapter of every story. We had an index built that included this. However, in the final week of this project, an attempt to move a repository on our index VM to a location that every team member could access it broke the operating system, trapping it in an endless boot loop. With the size of the raw dataset, the JOIN query required to concatenate these fields to the chapter takes days to generate. Due to the time constraints,

we dropped the concatenation, which allowed for the expensive JOIN to be replaced by a simple SELECT statement. Whilst this does mean you can't search directly for story names or authors, if you know these specifics, Google is well suited for getting those results, whereas our main benefit, search specialised for AO3 stories and metadata, remains available.

3.3 Retrieval Model

3.3.1 Retrieval Model Overview

Our retrieval model has a two stage pipeline. The first is a Boolean search engine which takes terms, tags, and story metadata predicates (e.g. word count > 10,000). This engine supports AND, OR and NOT operations in arbitrarily complex bracketed queries, as well as phrase search, proximity search, and inter and intra wild-cards. The second stage is ranking documents using BM25L.

3.3.2 BM25L

To rank the retrieved documents of a query, we decided to use the BM25 model which approximates relevant documents using a probabilistic approach. Specifically, we use a variant of BM25 known as BM25L under the assumption that a user of AO3 would not be satisfied with shorter documents. As such, we use BM25L as the original BM25 model biases towards shorter documents. The ranking function is as follows:

$$\sum_{t \in q} log(\frac{N+1}{df_t + 0.5}) \cdot (\frac{(k+1) \cdot (c_{td} + \sigma)}{k + c_{td} + \sigma}) \tag{1}$$

$$c_{td} = \frac{tf_{td}}{1 - b + b \cdot (\frac{L_d}{L_{avg}})} \tag{2}$$

Where N is the number of documents, tf_{td} is the term frequency of term t in document d and df_t is the document frequency of term t. L_d is the length of the document d and L_{avg} is the average length of all documents in our database. Parameters k, b and σ are free parameters that can be chosen, however, from established literature, the best practice values for k, b, σ are 1.5, 0.75, 0.5 respectively. Prior testing with these values on the development set of our database showed small changes in the ordering of documents however they were not substantial enough to warrant changing these values.

3.3.3 Inter-term Wild-card Search

Inter-term wild-card search was implemented in our search engine which enables a user to make queries of the form "term1 * term2 * term3 ...", where * can contain 1-20 omitted words. It is important to note that only phrase search supports inter-term wild-card search because inter-term wild-card search either makes no difference or does not make sense for other types of Boolean queries (logical and proximity search):

- Query AND * = Query (AND does an intersection and some subset's intersection with the whole set is that subset itself) → * makes no difference
- Query OR * = * (OR does an union and some subset's and the whole set's union is the whole set) → it does not make sense for a user to search for the entire set of terms
- NOT * = ∅ → it does not make sense for a user to search for none of the terms
- #number(term1, term2, *) = #number(term1, term2) → * makes no difference

The working principle of the phrase search supporting wild-card search is as follows. Firstly, it is ensured that there are no subsequent *, but if there are, the secondary wild-cards are removed ensuring that either 0 or 1 "*" appears between subsequent terms. Then phrase search becomes modified proximity search with varying distance: for those terms which have 0 wild-cards in between, the distance between such terms in a document must be 1; for those terms which have 1 wild-card in between, the distance between such terms in a document can be at most 20. 20 words were chosen because some upper threshold was needed for efficiency: if there was no upper threshold, the runtime for the phrase search would be O(terms * documents * max_position), whereas with the 20-words threshold runtime is O(terms * documents) as max_position is now set to a constant. Restricting the number of omitted words to 20 words was not arbitrary - 20 words is the average length of the English sentence and normally we do not expect a user to search for a phrase where more than one sentence is ommitted.

3.3.4 Intra-term Wild-card Search

All query types (logical, phrase and proximity) support intra-term wild-card search which means that the term can take a form of "som*thing", where that wild-card, marks that any number of omitted letters can appear in the middle of the term. It is important to note that only 1 wild-card character is allowed within each term - if there is more, secondary "*" are ignored. In order for intra wild-card search functionality to work, a DS storing all the permutations of all the terms was needed. Dictionary (key - permutation, value - term) and trie were the main candidates. Dictionary's main advantage is O(1) retrieval which is ideal for a search engine, however, dictionary takes too much space - we would need to store as many entries as there are permutations in the whole dataset and with our dataset's size this was not doable. Therefore, we switched to the trie data structure which stores the permutations in a more efficient manner and has the average retrieval of O(term_length) time which still suits our needs. The permuterm index trie was built in the following manner: for all the partially preprocessed (i.e. before stemming) terms, we found all their permutations and put them in the trie. Each trie's node stored one character and leaf nodes also stored a full partially preprocessed term (because it is impossible to recreate full term by following trie's nodes). The built permuterm index was saved to a file and every time a search engine is opened, it is recreated from the file rather than built from scratch because of the time constraints.

When a user makes a query, a search module inspects if there are any terms containing intra-term wild-cards in it. If there are, it tries to expand them in the following manner: firstly, each such term is rotated until "*" appears at the end, then we trace down the trie data structure by following characters in the rotated term until we reach "*" and stop in the current trie's node. From this node we do DFS traversal which finds all the expansion terms in the leaf nodes. In this way one query is expanded into multiple queries, reflecting all the permutations of query terms, joined with the OR keyword.

Unfortunately, intra-term wild-card searching is far from efficient. When run against our full index, the query "*day" takes about 40 seconds to run. From our testing, we believe this to be due to the fact that our index and retrieval model are on different servers, and the number of iterative TCP requests that our implementation needs to handle the range of possibilities. For example, if our query is "*day is a week day", and the only possible expansions are the seven days of the week, then there will be seven sets of index requests made; one for "monday is a week day", "tuesday is a week day", etc. We know that this is inefficient, and so rebuilt wild-card handling to reduce the number of duplicated requests, and batch the requests instead of making them in sequence. After expanding a wild-card term, we stored every possible candidate in a dedicated object, and then bulk requested posting lists for each candidate. After that, it was a case of carefully handling

these objects as the query was processed, but we were effectively able to treat a wildcard like a single term, needing only one index request. However, when put into practice, this approach failed. When using the full wildcard permutation trie, the set of postings lists being requested was so large that it would consistently cause a kill signal to be sent to the retrieval model. When we then tried a smaller trie, the process hung trying to prepare the request for long enough that it was slower than our naive original implementation.

3.4 Front-End

3.4.1 General Approach for Front-End Decisions

When making decisions for the front-end, we identified that under ideal circumstances we would implement a feature or component and conduct some form of experimental testing (e.g. A/B) to check whether our decision can be verified with statistic significance. However, due to limited time available, we chose to conduct a qualitative comparative analysis of existing search-engine front-ends which are likely to have been through above-mentioned experimental testing.

This influenced our decisions for the result pagination (section 3.4.4) and the general interface design (section 3.4.3). For the remaining aspects of the front-end, we decided based the group's consensus.

3.4.2 Tools

The project required a front-end website. While it is certainly possible to implement a sufficient web front-end using native web technologies (HTML, CSS, JS), we decided to use frameworks we are most familiar with to save time and be able to produce a better end product. We used:

1. **Interface functionality:** Next.js as React-based Javascript framework. Using this React framework, allowed us to choose a component-based implementation approach, avoiding modularity issues and tight coupling problems one can run into with plain JavaScript scripting.

2. **Styling:** Tailwind CSS as CSS-based framework. Allowed us to use component-based styling, suitable for the development approach we chose with Next.js.

3.4.3 Interface Design

Here, "interface" refers to the visual interface of the website, not a programmatic interface.

The general interface design is strongly influenced by the interfaces of the search engines mentioned in section 3.4.1. The first interactive item encountered on the search page is the query input field, as this is the main interaction point with our search engine. Results are presented with their corresponding story title, link address (so that the user can verify to which address they are going, a matter of security), and a preview/excerpt of the story.

3.4.4 Result Pagination

For some queries, the search engine is expected to return a large number of results. In this case, a pagination process needs to happen where the front-end can access and present a subsection of the results. This allows to display results within a short loading time even if the result set is large, rather than loading all results upfront and rendering them on the website. Internally, the front-end integration API (see section 3.4.6) handles subdividing the results.

We identified two approaches for handling pagination in our interface during the qualitative analysis described in section 3.4.1: firstly, pagination links at the end of the page where the user can jump to a different subset of results (as seen for google.com). Secondly, an infinite-scroll pattern which appends the next section of results upon request or scrolling to the end of a webpage (as seen for duckduckgo.com). Even though the second approach does not allow for the user to easily navigate between subsets of results, we chose indfinite-scroll due to it being easier to implement.

3.4.5 Tag Auto-Completion

The tag auto-completion feature was made possible by building a prefix tree from all available tags. Before doing so, we assessed the number of tags and the anticipated cost of that operation. We chose this data structure since the retrieval time is more relevant than the build time. The suggested tags are limited to a maximum of five so that the UI will not become too crowded with tag suggestions. We decided that the user should be able to input tags in the query field rather than the metadata filter form to enable faster filtering by tags (the user won't need to navigate to the filter form).

3.4.6 Integration with Back-End

The website communicates with the back-end via a front-end integration API which accepts requests for queries (and metadata filters) and for tag auto-completion suggestions. Metadata filters are passed as URL parameters which has the advantage of filters being saved in the URL; this way users can send results of queries with applied filters to each other by just sharing the link. The integration API then retrieves the requested data from the back-end.

3.4.7 Additional Features

The website includes a few additional features which are not strictly needed for querying purposes but do improve the UX:

- **Advanced query syntax feedback:** we indicate to the user when they use advanced query syntax correctly or incorrectly when writing the query in the input field (Figure 2 in appendix). This provides instantaneous feedback to the user whether their query is well-formatted.
- **Highlighting of occurring words:** words contained in the query text are bold in the story preview to help the user locate relevant results. Highlighting is case-sensitive.

3.4.8 Chapter Linking Limitation

AO3 does not have a coherent chapter linking standard. This means that if a story is sitting at `<story_base_url>/chapters/1`, the next chapter will not sit at `<story_base_url>/chapters/2` but rather have a random ID assigned. This is why, as a workaround, we link to the first chapter and in the case that the relevant content is in a later chapter, we provide the user with that information so that they can navigate to it manually. Another approach would be to scrape all chapter links for all stories, but we chose to not to this due to the expensiveness of such an operation (both to our and AO3's servers).

3.5 Web-Scraping and Scheduling

As AO3 has an active community, we wanted to implement a web scraper so that our search engine would stay up to date. Although AO3 openly allows scraping, it imposes throttling, which limits how quickly we can scrape. Due to this, we perform one scrape a day, which reads in a limited number of stories published within the last 24 hours. This quantity is nowhere near the amount posted per day, nor does it address the two year gap between the original database and this new input, but it demonstrates the feasibility of live indexing within our resource constraints.

The following are some of the edge cases we dealt with whilst scraping AO3:

- Despite the standardised formatting of story metadata, statistics where sometimes missing, or held within a different element than normal. Default error values were used to account for this, minimising any negative impact when used within the search engine.
- AO3 IP blocking and timeouts - Although AO3 allows scraping, it has policies in place to prevent over-eager bots from DOSing the site. What this meant for our scraper was that as it scraped, we would exceed their limit for requests in a set time frame, and would no longer receive pages. At this point, we then had to wait for five minutes before continuing on with the process. We found this meant that scraping 100 pages (each containing 20 stories) could take up to five hours.
- Unknown chapter counts - Users have the option to specify how many chapters are in a story. But they can also leave this field blank, if they are unsure. When this happens, and there is only one chapter published, the structure of the page makes it appear to the parser like there are additional chapters, with it would scrape as an empty entry.
- Viewing multiple chapters and adult stories - If a story has the "Mature" or "Explicit" warning tag, users first need to click through a confirmation to confirm that they want to view the story. This adds an extra request per story, slowing down our speed. Additionally, if a story has multiple chapters to import, each is on a separate page. To solve this, we found tags to add into the URL that can bypass the warning, and will render every chapter on the same page.

System scheduling the web-scraper was a particularly difficult task as many of the google VM directories were write protected. Placing a system event file within /etc/... and then enabling it as a system boot file allows the VM to look for and enable the python file on boot. To schedule tasks, once the VM has booted we had a python file run a schedule where at 1am each day, the webscraper would collate all of the previous days stories and drop them into a JSON file for the index to pickup at its next update.

4 Individual Contributions

4.1 Eric Janto

Eric was responsible for developing the front-end website, in correspondence with the other group members when it came to major design decisions. This included researching appropriate tools, learning how to use them for this project, and implementing the website. Eric was also assigned to implementing the tag auto-completion feature. This included researching an appropriate data structure, building it using the existing back-end, and working together with Mrinmoy to integrate the auto-completion functionality into the existing APIs. Eric was also driving the front-end integration API which is the crucial link between the front-end and the back-end, working closely with Junhua. Furthermore, he helped Jack with internal port managing and creating corresponding DNS records to publish the website on our domain. Towards the end of the project timeline, Eric was a major contributor towards bug reports and flagging incorrect behaviour for filtering and advanced search queries. Finally, he worked together with Junhua to add an SSL-certificate to the Apache server through which the back-end serves its data. As a team, the two of them also resolved CORS (cross-origin-resource-sharing) issues.

4.2 Jack Watt

The first primary task was to design and organise the high level structure of the system including the pathways for information. This can be seen in the figure in the appendix [fig.1]. An important step as it governed how the project would proceed. Jack was also tasked with running and operating all the primary functions of the GCP platform which hosted the back end systems with multiple machines communicating internally across a VPC network. This took some time to configure, ensuring unfiltered traffic. Internal communications were tested, external network connections were setup. Jack acquired an external static IP's and a DNS PTR record for the server to be exposed to the internet. Every static IP reservation, as well as the DNS PTR Records, had to be published to google name servers for address resolution by global searches. This took about a day to propogate. Jack also was tasked with webscraping AO3 for stories from a particular fandom, this was to show its feasibility without exceeding hardware and time limitations within the scope of the project. Ensuring the webscraper handles all the edge cases was a particularly difficult task since the variability in stories and posts on AO3 is utterly huge. Once systems were in place, schedules had to be set up to allow the running of the scrapes every day, the parsing of JSON into the Index and the refresh of the Index residing in memory. All these schedules are activate on the boot of the VM's.

4.3 Junhua Lin

Junhua was assigned to the retrieval team and was responsible for integrating the Boolean retrieval model and the BM25L ranked retrieval model. The first step was implementing a query parsing system that allowed for a mix of Boolean and plain text queries. Due to early experimentation revealing that Okapi BM25 prioritized shorter documents, Junhua opted to implement the BM25L model instead. To enable query processing, an endpoint was exposed so that the front end can submit user queries along with additional parameters (tags, filters) to the retrieval model. Additionally, the retrieval model must access the backend indexes through an API as they are hosted on a separate machine. Thus to enable fast query processing, all accesses to the indexes are batched to prevent additional TCP overhead. Furthermore, a simple query cache was implemented to prevent unnecessary access to the index for similar queries. As documents in AO3 are labelled with tags,

users may be interested in specific tags. As such, Junhua also integrated tag searching so that only documents with requested tags are retrieved.

4.4 Matthew Wisniewski

Although each section of the team (indexing, retrieval and frontend) largely managed themselves, Matthew was the overall team leader, and took responsibility for ensuring that when these subsystems were combined, they would work together. On the technical side, Matthew's main role was the implementation of the indexing system. Matthew designed and implemented the indexes for the chapter, tag and metadata indexes. Following this, Matthew built the indexes from the raw data held in the initial database, from small test indexes in the early weeks to allow other teams to test their work, to the full index used in the final system. Matthew also implemented the index compression system, thoroughly bench marking and optimising it to minimise runtime. Matthew was involved in the initial design of the API connecting the index and retrieval systems. Matthew worked to take the raw JSON files generated by the web scraper, and convert them into index chunks that could be added to the live index. Matthew also worked closely with Junhua to rework intra-term wild-card searching, in the unsuccessful attempt to optimise its runtime.

4.5 Mrinmoy Sonowal

Mrinmoy primarily worked on indexing and setting up the index APIs. The first task performed was to extract the fields and values from the chapters database and write SQL commands to import the data into python. Mrinmoy worked to create a SQL command that would import data from all 4 databases and perform a merge that would produce the required data in python to create the chaptersIndex using the Positional inverted index class. Mrinmoy worked on creating the term counts class that calculates the number of terms before processing, before stemming, after stemming and after processing. This helped the retrieval team work with these counts. Mrinmoy initially worked to create an application layer protocol by working with socket programming, however, as a multi client, multi request protocol had to be created, in the interest of time, Mrinmoy used the FastAPI library instead to create the 4 server-client systems.

4.6 Skaistė Mielinytė

Skaistė was assigned to the retrieval team and her main responsibilities included enabling traditional retrieval (logical, proximity and phrase search) in our search engine, implementing intra and inter wild-card search and enabling filtering in the backend. As for the traditional retrieval, we already had a codebase from CW1, so only a few updates were needed to remove limits imposed by CW1 and make functions more powerful. Inter wild-card search was implemented like a special case of proximity search which required some optimisations to decrease the runtime. The intra wild-card search functionality was the most challenging task. This task required to do additional research in order to find a solution which would minimize both additional storage needed and waiting time for the query, even though both of these things are actually a trade-off. Implementation using trie data structure worked well on example index, but we acknowledge that it is not compatible with the full index, however because of the time contraints Skaistė couldn't reimplement it. The final task was to set up filtering in the backend based on story's metadata. Filtering stories in the backend rather than frontend was crucial for our system because we work with high loads of data - after filtering stories in the backend, we can do search operations only on the filtered works.

5 Appendix

5.1 System Architecture

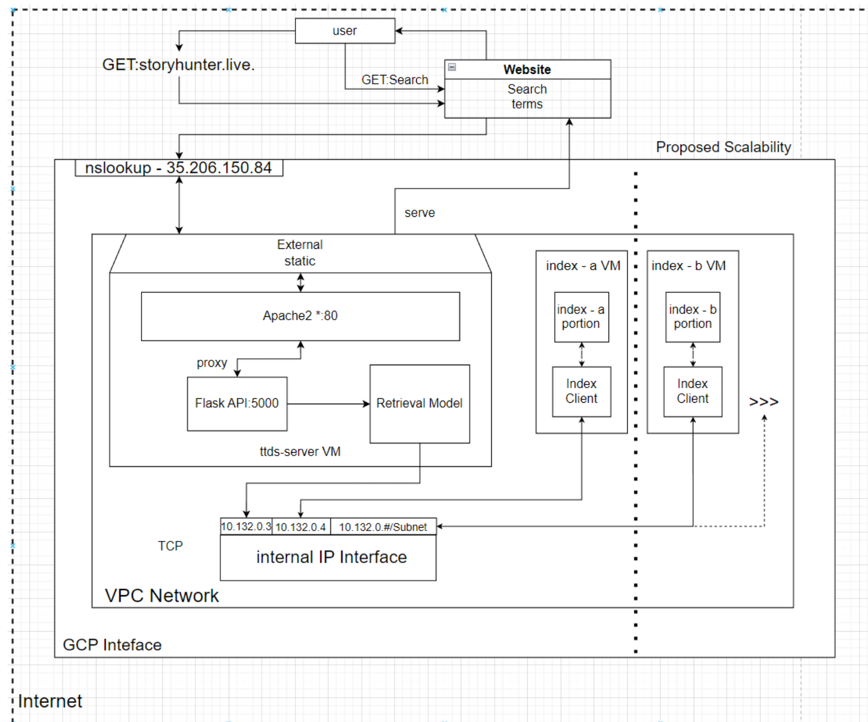

Figure 1: Design of the data and control flow for the entire system

5.2 Website Screenshots

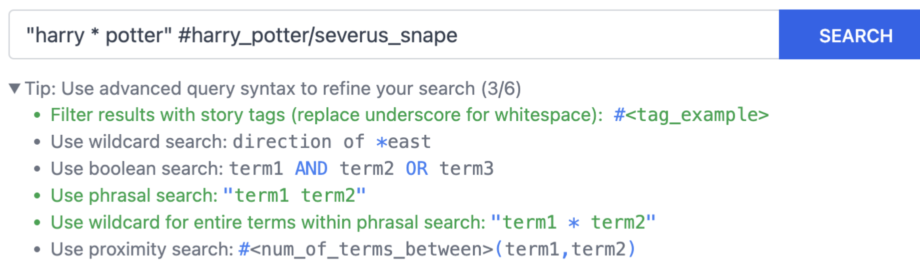

Figure 2: Syntax highlighter for advanced queries, giving feedback to the user whether they formatted their query correctly