



School of Informatics

Text Technologies for Data Science

Coursework 3

March 2024

Revelio: A News Search Engine

Group 1

S2177703

S2017594

S1908181

S2000527

S2101367

S2543673

Abstract

Revelio is a news search engine that allows users to search for news articles. Currently, the database contains 300,000 articles that have been balanced by their political leanings (left, neutral, right). We expect that this ensures political ‘neutrality’ when users search for results (although dependent on the search query). Users have access to multiple types of searches to fit their needs. They can type free-form text, which returns the most relevant documents following the Okapi BM25 score. Furthermore, they can use boolean searches combining AND, OR, and NOT to refine their search results. Moreover, we provide phrase search (“Insert Phrase) and proximity search (#x)token, token)) as additional ways to influence the search results.

We incorporate multiple optimisations for the base index (Word Trie, caching, batch processing), advanced features (query expansion, content highlighting, query correction), and a novel feature (named entity filtering). With these features, we provide a seamless and efficient user experience, allowing users to filter results by named entities and date.

The server can be accessed at <http://34.105.244.63:5002>

1 Introduction

We started ideating for this project by examining existing search engines and evaluating where their weakness was. Many known engines (Google, Bing, Yahoo) provide a comprehensive user experience that allows them to search for many different types of results (e.g. sports, news, information, etc). News search was something that we found quite interesting - although these search engines managed to return the most relevant and latest articles, they failed to provide any information beyond this. To this end, we found an unexplored niche that we wanted to investigate - displaying the most common entities (people, laws, countries, etc) given a query. After this, we also went a step further to give users more control over their search by integrating named entity filtering in conjunction with date filtering. Therefore, we believe that Revelio provides a comprehensive and well-rounded search experience and to the best of our knowledge, provides access to a never-before-seen feature.

In the rest of this report, we introduce Revelio’s architecture (backend and frontend). Then,

we briefly talk about how we obtained our data and touch on the server that our engine runs on. Section five spans most of the report, where we discuss Revelio’s features in-depth from optimisations to novel features. Moreover, we also write about the GUI design choices. We also perform quantitative and qualitative evaluations to test our system. Finally, we retrospectively discuss our difficulties and limitations.

2 Architecture

2.1 Backend

The architecture starts from the point whether the index is built or not. If the index is built as a cache, then loads the saved index cache and starts to search after query processing. If not, then our program starts to build the index. In the index-building process, the programme checks whether there is a file path cache or not. The file path cache stores all the article paths in a pickle format. With the file path cache, the crawler can retrieve articles linked to the file path. If there is no file path cache, the program builds a file path cache by mapping a file path to its checksum. If it finds a new file path, the file path cache is updated. When the file path cache is built or the cache is found, the program gets the files, and the files collected are saved into the Redis database. When building the index with the collected files, the program builds the index separately with a batch (Batch processing 5.1.4). After building the batches index, we merged them and saved them as a cache in a pickle format. Then, the program begins to search after processing the query.

2.2 Frontend

The website’s home page has an input box that fetches the query entered by the user. The search button triggers the validate form function which checks if the query is null or not. If it is null then the search function doesn’t proceed and if it is not null then the search function is called through Rest API and results are fetched which are then displayed on the results page. During this, as the user enters the query, a drop-down list shows the suggestions, It also handles clicks on suggestion items, setting the input field value to the clicked suggestion and navigating to the search page with the selected suggestion as the query parameter. If the spelling of the query entered is not correct, the spell checker suggests the correct spelling on the results page.

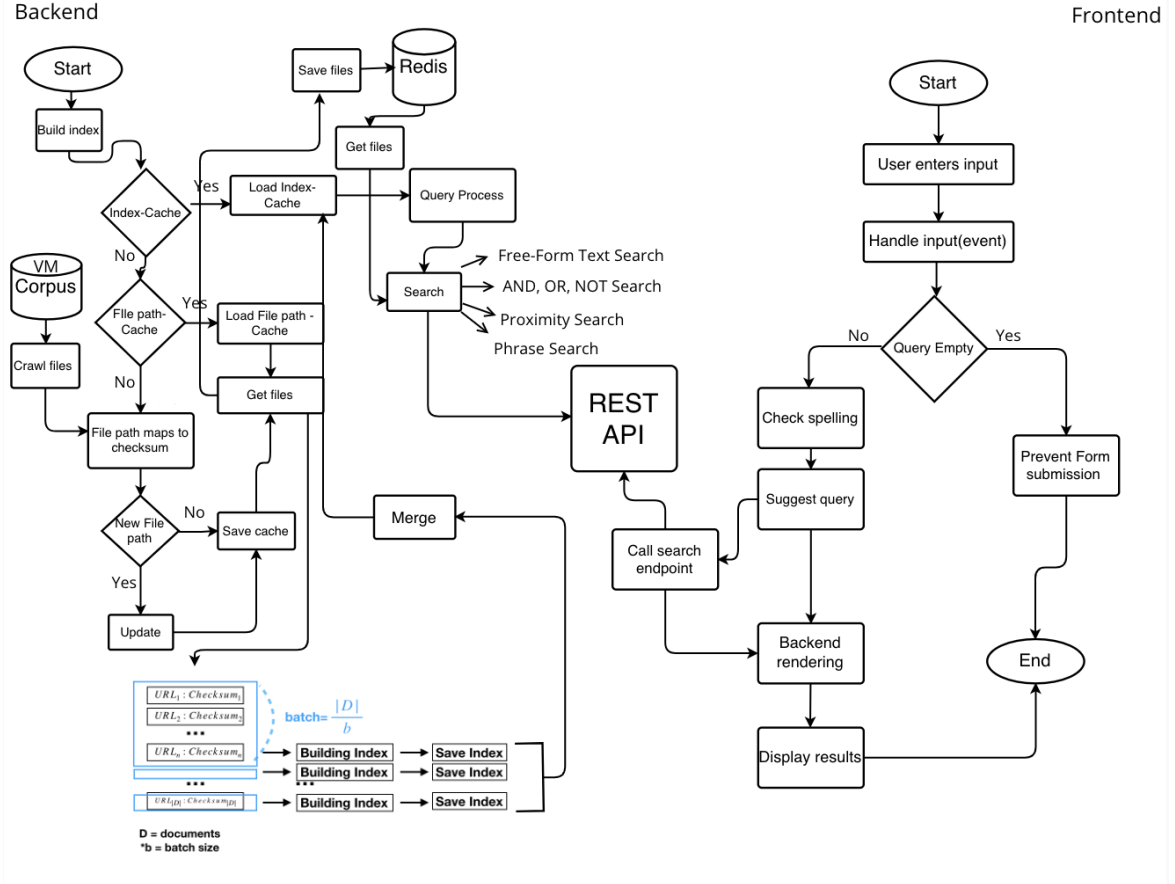


Figure 1: Revelio Architecture

3 Data

The POLUSA Dataset¹ (Polusa) is a news dataset of 905,572 new articles collected from 18 publication outlets across the political spectrum from January 2017 to August 2019 [4].

Each article includes the headline and the body text as well as metadata such as author names, publication dates, URLs, the name of the publication outlet, and political leaning. The distribution of political leaning in the articles is fairly balanced, with 31% of articles coming from left-wing outlets, 27% from politically centred outlets, 16% from right-wing outlets, and the remaining 26% from outlets with undefined political leaning. On average, each article contains around 500 to 1,000 characters.

The raw collection is available in the form of CSV files. We pre-processed the dataset by removing 26 articles without a headline which amounted to less than 0.003% of the collection. For every row in the CSV file, we converted the articles into HTML files and stored them in a back-end directory, where the index logic would fetch them.

¹Permission to use the dataset has been obtained from the dataset creators.

4 Google Cloud Platform Compute Engine

We host the service on a Google Cloud Platform (GCP) Compute Engine instance (see Figure 12 for full configuration). It is possible to run our current code base on non-specialised, general-purpose servers such as the E2 machines, however for faster index building process, the search engine is running on a C2D instance with 64 GB of RAM. With the flexibility of GCP, we were able to switch between various machine configurations during development to ensure that the most suitable one was used to meet the current demands of the search engine.

For vertical scalability, as greater processing power is required in the future, our compute engine instance can easily scale by upgrading the instance - no changes in terms of setup are required. For horizontal scalability, our compute engine instance can be cloned into multiple copies to meet the demands of the volumes of requests. We would need to include an external application load balancer from GCP as the middleman between the frontend and backend.

5 Features in Revelio

Revelio boasts numerous features, both in the backend and frontend, that ensure a seamless and streamlined user experience. From state-of-the-art databases to novel filtering features, we are committed to providing users with the most relevant articles efficiently and handing over agency to them as to what results they see.

5.1 Optimised Positional Index

The search engine constructs an inverted index using a **Prefix Trie** data structure and **Postings Linked Lists**. The index is built by processing a collection of documents and extracting relevant information from each document.

5.1.1 Document Crawling

As this system is intended to work in a live setting, we abstracted any actions related to document crawling. These actions include file discovery and file content update checks(queueing marked files for re-indexing if the content has changed). We implemented the (**Crawler** class) to retrieve documents from specified seed directories(with the provision to add more paths). The crawler loads previously crawled documents from a cache file (**CRAWL_CACHE_PATH**) and identifies new or modified documents to be processed.

The crawling process involves the following steps:

1. **Directory Traversal:** The crawler iterates through the specified base directories and their sub-directories to find files.
2. **File Validation:** Each file is validated against certain criteria, such as having a specific file extension (**SOURCE_FILE_EXTENSION**) and containing the required HTML structure (e.g., presence of **<h1>** and **<p>** tags). Files that do not meet these criteria are skipped.
3. **Checksum Calculation:** For each valid file, the crawler calculates a checksum using the SHA-256 algorithm. The checksum serves as a unique identifier for the file's content.
4. **Cache Comparison:** The crawler compares the calculated checksum with the corresponding entry in the cache file. If the file is new or has been modified (i.e., the checksum differs from the cached value), it is added to the **updated_stack** for further processing.

5. **Cache Update:** The cache is updated with the new or modified files, and the updated cache is saved to the cache file.

5.1.2 Tokenisation and Named Entity Recognition

The **Tokeniser** class is responsible for tokenising the document content and performing Named Entity Recognition (NER). We kept tokenisation logic very minimal, using only whitespaces and punctuation as delimiters, while optionally keeping certain punctuations within the tokens. For obvious reasons, NER is applied before any tokenisation as named entities may be broken by tokenisation if applied after.

Key tokenisation steps:

1. **Named Entity Recognition:** If NER is enabled (**include_entities=True**), the tokeniser uses the spaCy NER model to identify named entities. The entities are stored along with their labels, character offsets, sentiment, and subjectivity.
2. **Token Splitting:** The text string is split into tokens using a regular expression pattern that considers the specified punctuations to keep.
3. **Post-processing:** As a first step, we apply token positioning(**token -> (token, position)**) before preprocessing. This is done to keep relative token positions considering that some tokens may be filtered out. Preprocessing is then done, which includes lowercasing, stopwords removal(if **use_stopping=True**), and stemming (if **use_stemming=True**). Tokens that are too short, too long, or contain repeated characters beyond a threshold are filtered out.

The tokeniser returns a list of tokens and, optionally, a list of named entities for each document.

5.1.3 Document Insertion

Document insertion processes one article at a time on the article's formality. Each article contains metadata. Metadata includes the document's number, headline and its span, posting, and the name entities extracted from the previous tokenising process. Inserting the Document means these metadata are added or updated into each index like 1. In this way, we filled up these metadata into each index.

5.1.4 Index Persistence and Optimisation

We faced an out-of-memory issue at the initial stage of building an index. Increasing the memory of our GCP instance can be a typical solution, but we should think of other approaches due to the server running cost as the higher computing engine is more expensive to run the cloud engine instance. Also, empirically speaking, the crashing point of the our-of-memory issue was arbitrary. Due to these points, we should design batch processing for building an index. Previously, we built an index with all corpus once. With batch processing, we could separately build an index by a batch and merge them afterwards. It improved our memory efficiency, and potentially we can build a larger index with more than our current 300,000 articles with this process. In practice, we set 10,000 as a batch size. We subsequently have 30 batches with 300,000 articles. With batch processing, we could save our memory usage. After building the 30 batches index, we merged and saved them as a cache in a pickle format.

5.2 Trisse Data Structure

The search engine utilises a trie data structure for efficient indexing and retrieval of terms (Figure 2). The trie enables fast lookup, insertion, and retrieval with a time complexity of $\mathcal{O}(k)$, where k is the term length. Trie nodes store term-related information, including unique identifiers, document frequencies, and mappings to postings lists.

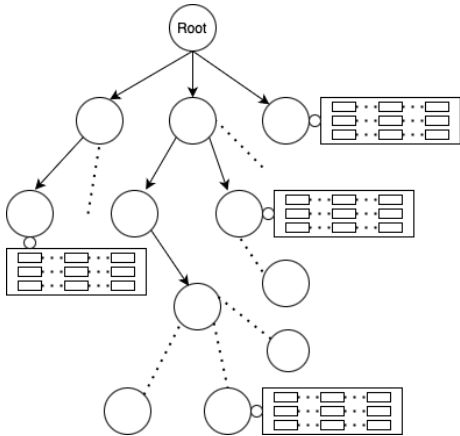


Figure 2: Trie Index Structure

The trie’s prefix sharing reduces memory usage and supports incremental indexing. It also enables quick term lookup during search operations by traversing nodes based on the search term’s characters.

Postings are stored using a linked list structure, with each trie node mapping document IDs to their corresponding postings list. The linked list maintains term positions within documents in

sorted order, enabling efficient merging and intersection of postings lists during query processing.

The choice of using a **Postings Linked List** was made with the foresight of potential performance gains made possible by this structure during query time. An array-based data structure would have offered $\mathcal{O}(1)$ element access but would have incurred performance overheads during posting insertion and querying(querying would have required frequent sort operations). While a linked list has $\mathcal{O}(n)$ complexity for insertion and deletion, it allowed for efficient pointer based algorithms during query processing.

As the postings are kept in sorted order, we also included skip pointers as an additional optimisation step. The merge algorithm traditionally operates at a time complexity of $\mathcal{O}(n + m)$, where n and m are the lengths of the postings lists. But with skip pointers (kept at \sqrt{n} intervals), the traversal through postings lists can be expedited, reducing this to $\mathcal{O}(\sqrt{n})$.

5.3 Live Data

Every day at 22:00, live news is obtained from the Guardian API [8] using the Python job scheduler. Each collection point retrieves 20 articles from the previous 24 hours and then preprocesses them. The metadata for each article, including the headline, body, date published, and URL, is first extracted from the API response. This information is used to generate an HTML file for the article using the *Database2HTML* converter before being stored in the *live* folder of the corpus collection. As the articles are saved, they are assigned a unique incremental ID that enables each live article in the folder to be uniquely identified. Algorithm 4 shows how the live articles are processed by our search engine at each collection point.

5.4 Redis Database

The Redis database acts as a cache that stores the news article html files crawled by the directory crawler during the initial file crawling process. Subsequent retrieval of html files are facilitated by the redis database during index building rather than direct access to the html collection directory which is a bottleneck.

5.5 Advanced Querying

Our query algorithm is extended from Coursework One. Firstly, we provide the basic boolean searches - a combination of AND, OR, and NOT searches. We have extended their functionalities to be compatible with an arbitrary number of tokens. We use a recursive function to break down

the query tokens, and build the final document IDs - postings object bottom-up. This means that we perform the necessary Boolean operations (intersection, union, and difference) by keeping track of Sets of document IDs at each level.

Next, phrase search and proximity searches have largely remained the same. Users can use these functionalities by inserting "Query or #X(Token, Token) (where $X > 0$, $X \in \mathbb{Z}$) into the search bar respectively.

Our most advanced search is the ranked retrieval. We are capable of using either the TF-IDF (Term Frequency - Inverse Document Frequency) or the Okapi BM25 [14] score in our backend. We have decided to incorporate the BM25 score as it can return more relevant results. BM25 not only takes into account the term frequency and document frequency like TF-IDF but it also contains normalising factors in the form of k and b . k controls the *saturation* of each token. That is, this constant performs regularisation to lessen the impact of a dominating term frequency value of a token. On the other hand, b handles the regularisation of document lengths. We have set k to 1.75 and b to 0.5, as these are preferred default values. An area for improvement would be to perform a hyperparameter search on these two constants to optimise them against our current task. However, we did not have enough time, so we used the default values that are known to produce a good baseline.

5.6 Advanced Filtering

News articles are incredibly time-sensitive and users should have the agency to filter articles by their publication date. Therefore, we allow users to filter and refine their searches through various means. Firstly, users can filter by date, such as *Past Week*, *Past Month*, or with a *Custom Range*. Thus, we deemed it necessary to include this feature in Revelio as news articles are incredibly time-sensitive and users should have the agency to filter articles by their publication date.

Named Entities are real-life objects, such as locations, people, and company names. In today's day and age, we encounter countless individuals, organisations, and countries in the news, and it is often not clear what entities are involved in which article. To this end, we offer a new type of filtering not seen in any popular search engine - Named Entity Filtering. Upon searching a query, we also display the top ten most frequent entities seen in the retrieved documents. Users are also able to combine these two filter options to enhance their search experience like never before. Combining these filters with the advanced query options, we equip them with the agency over the types

of results they see while at the same time providing them with interesting information about their query and the contents of the articles. We show the list of allowed entity labels in 2

5.7 Pagination

We perform pagination to efficiently display the first five returned articles. This reduces wait times for the user who would want to see the most relevant results (when using ranked search) as soon as possible. Furthermore, the named entity filters dynamically update as the user loads more articles by pressing the 'Load More' button at the bottom of the results page.

5.8 REST API

Implemented in Flask, the REST API provides a search endpoint to enable separation between the frontend and backend of the search engine. The search endpoint loads the index from cache and opens a TCP socket to listen to requests. The frontend calls the search functionality via the search endpoint with a Javascript client. The endpoint returns the retrieved documents in result cards in JSON format. The Javascript client then renders the results cards in HTML for display.

5.9 Result Cards

We name individual returned articles as '*Result Cards*'.⁶ These result cards contain valuable information that users might need to gather about a particular article. For example, we include a Favicon (a website icon) to indicate which publication each article is from. Furthermore, we show the publication date, article title, parsed URL, and content snippet. We talk more about the latter two features below. We believe that this way of aggregating results provides a more meaningful structure to the overall results page. Moreover, this paves the way for additional functionalities that we could add in the future and does not require a complete restructuring of the results page.

5.10 Content Highlighting

When a user searches for a query, they want articles that are the most related to their search. A main part of 'relevancy' stems from whether the articles contain their query. Therefore, we implemented 'Content Highlighting', which checks whether the article contains a token in the user's query. This feature shows a snippet of each article that contains one of the tokens in the query. We accomplish this by using pattern matching. We lay out the algorithm in 1. Furthermore, we have extended this functionality to better suit phrase

search. Therefore, our content highlighting algorithm highlights entire phrases if they are present in the content. An example is shown in 6, where we search "President Donald Trump. As mentioned before, the double quotation mark (") indicates a phrase search, and we can see this phrase highlighted in the returned article.

5.11 URL Parsing

Another feature that enhances the readability of the returned articles is 'URL parsing'. Once our backend returns the results to the frontend, we redirect users to third-party publishers' websites. However, their URLs more often than not are unreadable, combined with dates, random numbers, and relevant subdomains. Therefore, we show an algorithm 2 that takes in the raw URL and outputs a parsed, clean URL that shows the domain (e.g. *BBC*, *CNN*, etc) and the first subdomain (e.g. *news*, *politics*, etc). This algorithm also filters out any dates which we already display on each result card.

5.12 Query Expansion

A user's search might not contain all the necessary tokens to return articles that they are looking for. To this end, we include a query expansion module in our backend using NLTK (Natural Language ToolKit) ([2]). NLTK is a collection of NLP tools, such as tokenisation, synsets, and more. We used the WordNet ([9]) corpus from NLTK which contains semantic information about words (e.g. synonyms, antonyms, etc).

We also did not blindly apply this to every single type of query, as users might use a Boolean 'AND' to constrain the input space. Expanding these queries could introduce false positives that would clutter the results page and hinder the overall user experience. Therefore, we restricted this feature to only ranked OR searches. When users input a query, three (this can be fine-tuned) words are randomly chosen. Then, we use the NLTK **synsets** to choose a new synonym with the same Part of Speech tag (N, V, ADV, ADJ) and we append these words to the query. For example, the query '*Donald Trump Attacks Capitol*' would be expanded to '*Donald Trump Attacks Capitol Invades*'.

Currently, if a query contains a named entity, that named entity could be mistaken for a regular word and thus, expanded. For example, our model could expand '*trump*' into '*trump cornet*'. We should note that an improvement to this module would be linked to differentiating named entities and regular words during the tokenisation phase as mentioned before. This would avoid the

issue above where we accidentally expand named entities and return irrelevant documents.

5.13 Query Suggestion

We used the Google Suggestion API [7] to provide query suggestions in our search engine. Algorithm 5 showed how we performed query suggestions. The suggestion process runs on the front end and is dynamic, constantly fetching suggestions from the API for the query that the user has typed in so far. If users change their queries, the suggestions also adjust accordingly. The suggestions for the current query are displayed as a list below the search box, similar to how Google does it. Figure 11 shows how these suggestions appear when a user searches for the term "*trump*". When you click on one of the suggestions, the query in the search box changes to the suggested one, and our search engine automatically searches for articles that contain the new query.

5.14 Spelling Correction

Our search engine's spelling correction is implemented using the Javascript BJSpell library [5]. The correction process is carried out on the front end, only after the user submits their search query. If the query is an empty string or contains more than one word, spelling correction is skipped and only query suggestion is performed. Algorithm 6 showed how we performed spelling correction.

This decision for the second case is based on the limitations of the BJSpell library. Specifically, the spelling corrector corrects each misspelt word without accounting for the context in which they appear in the query. For example, if a user searches for "*donald trump*" but accidentally types "*donald trum*," the corrector tries to correct the second word independently and may suggest "*donald drum*" as a correction, lacking consistency. Due to the limited libraries that are compatible with vanilla Javascript, this is the best one our team could find for this functionality.

The spelling corrector generates a maximum of 100 correct words for each misspelt word. Using the jsLevenshtein library [1], we calculate the Levenshtein distance between the misspelt and corrected words to determine which words will be returned as the correction results. We choose to present all corrected words with the shortest distance to the misspelt word. This choice is based on the assumption that a shorter distance indicates a greater similarity between the corrected and misspelt words. Figure 7 shows how the correction results are displayed when a user searches for the term "*healt*".

6 GUI

We have implemented a creative, responsive and feature-rich user interface utilising HTML, CSS and JavaScript as these technologies offer streamlined development processes, scalability, and efficient maintenance. Flask is used for rendering web pages and providing seamless integration between front-end and back-end.

The home page of the interface includes an input bar for entering the query and a search button that renders the search results on the results page. It also includes navigation links - About and Help for users to know more about Revelio. The interface also includes enhanced features like **Query Suggestion**, **Spelling Correction** and **Content Highlighting**. The advanced search allows the user to filter the results based on the Named Entity filter buttons right below the input bar on the results page. User is also provided with Date filters to view the most relevant news. The results page can be seen in Figure 8

Breaking news does not wait for anyone. They can happen at any time, on any day, where users could be outside, or simply not have access to their desktops. Therefore, we deemed it crucial to support both desktops as seen in Figure 9 and mobile devices 10.

7 Quantitative Evaluation

We evaluated our search engine against the Time IR collection [6] using various ranked retrieval techniques. Our test collection contains 423 Time magazine articles and 83 queries. Each query has an average of 5 documents that are relevant to it, with the relevance being binary. During the evaluation, we measured IR metrics such as Precision@5, Recall@5, R-Precision, and AP.

We show the IR evaluation results for the VSM BM25 [14], VSM TF-IDF [15], and the Search Retrieval for the "OR" and "AND" operators [16] 4. The results show that VSM BM25 outperformed the other ranked retrieval methods across all metrics. However, this method still performs poorly, with P@5 and R@5 scores of approximately 0.5. This implies that it could only retrieve 2-3 relevant documents among the top 5 results. Furthermore, these relevant documents represented only about half of all relevant documents for the query.

The characteristics of the test collection may account for some of the poor performance. There were no baselines on the internet for this collection, and the average query length in the collection was around 20 words. This is about 4 times longer than the queries that most users typically submit to a search engine [10]; thus, these queries may not accurately reflect the actual searching

behaviour of users, making our test collection potentially unsuitable for evaluation. Nonetheless, it was the only publicly available IR test collection relevant to our search engine that we could immediately download and use.

The evaluation results are not ideal, and improvements could be made to our search engine in the future. Firstly, if our project were slightly longer, we could submit a form to request the TREC News collection [13]. TREC is a well-renowned IR evaluation campaign used by many researchers, so obtaining this collection could enable us to conduct a more reliable evaluation of the search engine. Initially, we were uncertain about the time it would take for TREC to respond to us, so we decided to use the Time collection instead. Second, we could consider using Learning to Rank (L2R) [3] to rank the documents instead of classical models like TF-IDF and BM-25. This is because L2R outperforms the other models in terms of search performance [11].

Moving on, we show the execution times of the various searches 3. These times are on the whole quite efficient, with most evaluated in less than one second, and we saw that boolean AND and OR executed extremely quickly (at around 60ms). However, we faced a problem when optimising the ranked search. Since this looped through all tokens in the query, and for each query, through all documents that contained the query, the execution time was very slow (with the worst runtime of $O(td)$, where t is the number of tokens and d is the maximum number of articles).

Therefore, we introduced multiple fixes. Firstly, we incorporated a *min-heap* to only keep track of the top N documents at a time. This reduces both the space and time complexity ($O(n)$ to $O(1)$ and $O(n \log n)$ to $O(\log n)$ respectively). Moreover, we also set a time limit on the search. We realised that there was no feasible way for us to iterate through all the returned documents. Therefore, we allow a maximum of four seconds to search for each token in the query, and ten seconds for the overall query. This ensures that the user experience is not hindered by an excruciating wait time to receive the results.

8 Difficulties

A major difficulty we faced was deciding on the scope of our project. We had posed many innovative, seemingly interesting ideas in the beginning, and had started working towards those goals almost simultaneously. However, we realised that not only was this an inefficient use of our time (as we needed to build an MVP first), but we had set the scope to be too large given our resources. Therefore, we went through phases

where we gradually scoped our project down to include the core functionalities, multiple optimisations (such as the Trie, caching, and batch processing), and a handful of novel features (named entity filtering, query correction, etc).

Moreover, we realised that although our boolean, proximity, and phrase searches were quite efficient (completing in under 1 second), the ranked searches took a while to complete (> 90 seconds). Therefore, we decided to introduce a time constraint. A limitation of this would be that for longer queries, our engine would not search later tokens. We could solve this by changing our ranked search to use a machine learning method or PageRank.

9 Limitations

9.1 Breadth of live data

A limitation that we faced with the data was its date range. In real life, news never stops. Publishers constantly push out new articles every hour, and there is always something happening every day. However, we found that our dataset only contained articles over a limited range of dates. To combat this, we introduced the live data feature. This would fill in holes where there is a lack of data, as API such as the Guardian API gives access to data from 1999 [8]. This introduces another problem where the dataset now contains imbalanced political data. A solution would be to use APIs from multiple publications, but due to restricted access to publication APIs for developers (e.g. BBC API is only accessible internally), we were not able to realise this functionality. However, since the data part of the project is modular, our current codebase permits more APIs to be easily integrated in the future when they become available.

9.2 Ranked Search

Due to our implementation of the ranked search (imposing a time cap), this meant that some tokens later in the query might not be considered in the search. This is both a hardware and technical limitation - if we had a more powerful computing engine, the algorithm could sift through more documents. Furthermore, if we had employed a different method of scoring these documents, we would be able to examine all returned documents rather than imposing an arbitrary cut-off point.

9.3 Processing file path

Our articles were saved as an HTML file so that each article has a URL as its file path. We can ac-

cess the article and retrieve it if we know the URL. To retrieve all articles for building an index, we mapped each URL to *docID*, which is the number incremented by one once the URL of the article is mapped at a time. After mapping all URLs to *docID*, we saved them as a cache file. However, this logic caused articles to be retrieved sequentially in our search process. In our corpus, all articles are saved sequentially by publications. 10,000 CNN articles, for instance, are mapped from 0 to 9,999 *docId*. When searching for something, we can only get results from one publisher. We can randomly select an article to avoid this phenomenon when building a file path cache. However, we could not build the index again to sort this issue out due to time constraints after we realised the issue.

9.4 Sentiment

Finally, we cancelled a feature that was almost ready for deployment - sentiment. There were two ideas. The first was to attach a score to each article, and the second was to attach a score to each named entity. Although this seemed like a novel idea, we ultimately decided against this. We reasoned that if we introduce a single value score that sums up an entire article, this could pre-impose some level of unconscious bias for the user. Take, for example, a query containing a politician. Let us imagine that our engine returns five articles - four praising them for bringing a boom to the economy, and another talking about potential crimes that they might be involved in. Our engine would subsequently classify the four articles as 'good' and the other as 'bad' (the user also does not know what that sentiment ultimately conveys - what is 'good'?). Furthermore, the named entity would have a sentiment score that was largely in the 'good' section.

We believed that there would be a lack of valid reasoning to justify these scores, and we were doubtful that they would provide an accurate overview of the article or the named entity. Furthermore, we argued that the job of a search engine was to provide the most relevant results and let the user form their own opinions about the displayed articles. If we had intercepted this flow of information by attaching a score that could introduce bias, then our search engine would no longer be a search engine, but it could potentially contribute to generating 'fake' sentiment regarding a particular article or entity. Therefore, we stuck to only displaying neutral results such as the top ten most frequent named entities.

10 Individual Contributions

At the beginning of the project, we decided to allocate individual tasks across sub-groups. Following this, each member focused on the allocated task in the development of the project.

10.1 S2177703 - Darakshan Ali

The major role I had in the project was to build the front-end. I have built all the components in the GUI. I started by creating an initial architecture by building web-pages using HTML, CSS and JavaScript. I built the Flask application with routes for different endpoints to render web-pages and fetch results in form of results-cards by integrating with back-end. I named our search engine - Revelio which means “to reveal”. I designed the logo for Revelio on Canva.

The most important features of our GUI are - Intuitiveness and Interactivity. The user friendly design is both appealing and interactive. Empty or null queries do not load the results page to avoid confusion and error results. I have accommodated the **Spelling Correction** to load along with the web-page to maintain the consistency. I have implemented **Query Suggestion** using JavaScript and Flask. The suggest function sends a request to a Flask proxy route /suggest with the provided query and fetches the suggestions which are populated on the drop-down list. The query suggestion provides a dynamic search experience as the user types in the search bar and is limited to 10 values at a time with an ease of scrolling for a neat look.

I have built the website to fit both desktop and mobile view for an enhanced user experience. I used CSS to make the website responsive and re-size on all the screens accordingly. I additionally conducted testing and made enhancements to areas requiring improvement within the Frontend. I also worked on the report - designing the front page, helping with formatting and writing about GUI and Architecture - Frontend.

10.2 S2017594 - Daniel Kim

I acted as a full-stack developer in this project. Firstly, I contributed towards the initial **Tokeniser** class and also implemented the ranked search and performed optimisations on it (such as using a min-heap). I worked on the vector space model but this was not used in the final product. Furthermore, I coded the query expansion that uses both synonyms and pseudo-relevance feedback. Although VSM and PRF F were not used in the final product, had we more time, we could have opted to use this model instead as it outperformed the current use of BM25 with

a Boolean OR 4. On the non-customer-facing side, I also developed four loggers (performance, memory, search, index) that helped us tremendously with debugging our project. During the final sprint, I helped identify a major issue we faced when building the index and massively optimised our ranked search.

I also worked on the **SearchRetriever** class that resides in the middle end. This class deals with retrieving search results by returning a **ResultCard** object 6. We also perform relevant content bolding and return the span within the object.

In the frontend, I was responsible for the initial redesign of the search page to be more intuitive and simplistic. Furthermore, I developed both the named entity and date filters that are used in the final product. I also played a part in integrating the frontend and backend as well as fixing bugs that occurred in the frontend.

10.3 S1908181 - Kuan Lon Vu

My primary role was as a backend developer. In data collection, I found the Polusa dataset and reached out to the dataset creator to obtain the dataset since it has access restrictions due to copyrights. The data was analysed for suitability and filtering unsuitable article entries. I also contributed to converting the raw CSV entries into HTML files which are used by the crawler, and in implementing Redis in the SearchRetriever. Moreover, I coded the REST API that provides separation between the frontend and backend, and the javascript client that calls the search endpoint and integrates that into the frontend workflow. I was responsible for selecting the appropriate configuration, setting up the GCP environment, migrating the project from our private server to the GCP instance, and also the eventual index-building process. Additionally, I worked on documentation to keep track of the functionalities and the interactions between the core components such as the workflow of dataset transformation, indexing and search. Due to the fast moving development pace, it was difficult to keep up of the documentation of additional features such as NER. I also helped with the general bug fixes

10.4 S2000527 - Martin Namukombo

I worked on developing the backend of the search engine, particularly the positional index and its optimisations. This was driven in part by familiarity with the our starter code (we had collectively agreed to use a version subset of the indexing code I had submitted as part of the

coursework). I then worked on the **Positional Index** and **Postings Linked List** classes, forming the foundation of the indexing system. Later implemented a trie-based **IndexTrie** and relevant search, update and posting intersection algorithms to improve memory efficiency and overall performance. I also worked on the **Crawler** class for document location indexing and tracking. This included keeping checksum values to queue files for reindexing, if the content had changed.

More general tasks included building an execution pipeline for the positional index. The pipeline encompasses document crawling, preprocessing, tokenisation, postings creation, index updates (saving and building), and metadata management. During the later stages of the project, I optimised the indexing process by implementing concurrency and batch processing to parallelise indexing tasks via process pools encapsulated in a thread pool. Additional changes included implementing checkpointing and index merging mechanisms as a fail-safes to improve efficiency and reliability, resulting in better resource utilisation.

10.5 S2101367 - Sean Choi

I mainly acted as a project manager and a backend developer. Firstly, I suggested a blueprint of the project at the initial development stage, and managed the project in alignment with this blueprint, although we had to alter some components within the agile development process. This guideline helped to define the scope of the project, which was somewhat open in the search engine context. As a result, various functionalities were able to be implemented in parallel within the time constraints. Moreover, by using Trello as a management tool, I gathered all project-relevant information in a single space, which made our project process easier. Check Blueprint, Check Trello

Regarding the backend development, I have implemented the core part of backend architec-

ture and its sub-functionalities, such as crawling, index building, batch processing, and cache process. Also, I should keep testing and doing maintenance of the core part due to the relatively large scale of the project. Especially, index building was time-consuming as it not only kept crashing but also the building index itself took a long time. After the index building, I optimised the ranked search to reduce the completion time. Furthermore, I improved the index efficiency by migrating the Trie data structure from a dictionary.

Additionally, I suggested logos as I have relevant work experience in this part. Finally, I have implemented two sentiment score estimators, including an article-based sentiment estimator and a firm sentiment score estimator for the financial domain. As mentioned in the limitation section 9.4, these two estimators were almost ready for deployment. However, we cancelled these functionalities after a rigorous discussion.

10.6 S2543673 - Tanatip Tingtong

I helped manage the logistics of collecting and formatting news data from the Polusa dataset to create a collection directory on which the crawler could crawl and build an index. In addition, I built sanity checks into the crawler to detect and handle faulty news articles. Secondly, I implemented a program that collects real-time data from the Guardian API. Another aspect of my involvement was the development and integration of query suggestions and spelling corrections on the front end.

Thirdly, I researched for an IR test collection that would be appropriate for evaluating our search engine. I also performed IR evaluations on the various ranked retrieval methods used by our search engine on the chosen test collection and analyzed the results. During the early stages of the project, I helped set up the GCP cluster. Finally, I integrated the NER information of the documents related to the query with the front end.

References

- [1] Gustaf Andersson. *js-levenshtein*. <https://github.com/gustf/js-levenshtein>. 2019.
- [2] Steven Bird, Ewan Klein, and Edward Loper. *Natural language processing with Python: analyzing text with the natural language toolkit*. "O'Reilly Media, Inc.", 2009.
- [3] Björn Ross. *Learning to Rank*. The University of Edinburgh. Nov. 29, 2023. URL: https://opencourse.inf.ed.ac.uk/sites/default/files/2023-12/ttds_2023_19_learningtorank.pdf.
- [4] Lukas Gebhard and Felix Hamborg. "The POLUSA dataset: 0.9 M political news articles balanced by time and outlet popularity". In: *Proceedings of the ACM/IEEE Joint Conference on Digital Libraries in 2020*. 2020, pp. 467–468.
- [5] Andrea Giammarchi. *bjspell*. <https://code.google.com/archive/p/bjspell/>. 2009.

- [6] University of Glasgow. *Time collection*. https://ir.dcs.gla.ac.uk/resources/test_collections/time/.
- [7] Google. *Google Suggestion API*. <https://suggestqueries.google.com/>.
- [8] The Guardian. *The Guardian OpenPlatform*. <https://open-platform.theguardian.com/>. 2009.
- [9] George A Miller. “WordNet: a lexical database for English”. In: *Communications of the ACM* 38.11 (1995), pp. 39–41.
- [10] Mordy Oberstein. *State of Search: Discover How the Web Changed in 2021*. <https://www.semrush.com/blog/state-of-search/>. Apr. 7, 2022.
- [11] Ahmed Saleh et al. “Performance Comparison of Ad-hoc Retrieval Models over Full-text vs. Titles of Documents”. In: *Maturity and Innovation in Digital Libraries: 20th International Conference on Asia-Pacific Digital Libraries, ICADL 2018, Hamilton, New Zealand, November 19-22, 2018, Proceedings 20*. Springer. 2018, pp. 290–303.
- [12] Gerard Salton, Anita Wong, and Chung-Shu Yang. “A vector space model for automatic indexing”. In: *Communications of the ACM* 18.11 (1975), pp. 613–620.
- [13] TREC. *The TREC News Track*. <http://trec-news.org/>. 2018.
- [14] Youssef Al Hariri. *Okapi BM25 Ranking Function*. The University of Edinburgh. Oct. 11, 2023. URL: https://opencourse.inf.ed.ac.uk/sites/default/files/2023-10/ttds_2023_08_ranked_retrieval2_0.pdf.
- [15] Youssef Al Hariri. *TFIDF term weighting*. The University of Edinburgh. Oct. 11, 2023. URL: https://opencourse.inf.ed.ac.uk/sites/default/files/2023-10/ttds_2023_07_ranked_retrieval.pdf.
- [16] Youssef Al Hariri. *TFIDF Variants*. The University of Edinburgh. Oct. 11, 2023. URL: https://opencourse.inf.ed.ac.uk/sites/default/files/2023-10/ttds_2023_07_ranked_retrieval.pdf.

A Equations

$$\text{BM25}(d, q) = \sum_{i=1}^n \text{IDF}(q_i) \cdot \frac{f(q_i, d) \cdot (k_1 + 1)}{f(q_i, d) + k \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)} \quad (1)$$

Figure 3: Formula for BM25. d is the document, q is the query, k and b are hyperparameters, avgdl is the average document length, and $f(q_i, d)$ is the term frequency of token q_i in document d .

$$\vec{q}_m = \alpha \vec{q}_0 + \frac{\beta}{|D_{\text{rel}}|} \sum_{\vec{d}_j \in D_{\text{rel}}} \vec{d}_j - \frac{\gamma}{|D_{\text{irrel}}|} \sum_{\vec{d}_j \in D_{\text{irrel}}} \vec{d}_j \quad (2)$$

Figure 4: Formula for Pseudo Relevance Feedback. α, β, γ show the importance of the original, relevant, and irrelevant document vectors respectively. \vec{q}_0 is the original query vector and \vec{q}_m is the modified query vector.

$$\text{Cosine Similarity}(\vec{Q}, \vec{D}_i) = \frac{\vec{Q} \cdot \vec{D}_i}{\|\vec{Q}\| \|\vec{D}_i\|} \quad (3)$$

Figure 5: Formula for Cosine Similarity. \vec{Q} is the query vector, \vec{D}_i is the i^{th} document vector. The output is a score x where $0 \leq x \leq 1$.

B Algorithms

Algorithm 1 Highlight Content

```
Initialize tokens as an empty list
if Strict Search then
    tokens  $\leftarrow$  query.split
else if Phrase Search then
    tokens  $\leftarrow$  [query]
else if AND or OR but not NOT then
    tokens  $\leftarrow$  query.split
else if Free Form Search then
    tokens  $\leftarrow$  query.split
end if
for all token in tokens do
    for all all occurrences of token in content do
        Calculate the start and end indices for snippet extraction
        Adjust indices to nearest word boundaries
        Create a snippet with the token highlighted
        if end of content not reached then
            return the snippet with trailing ellipsis
        else
            return the snippet
        end if
    end for
end for
return beginning of the article.
```

Algorithm 2 Parse URL

```
url_obj  $\leftarrow$  new URL(url)
hostname  $\leftarrow$  url_obj.hostname
split_path  $\leftarrow$  url_obj.pathname.split('/')
Define datePattern as a regular expression for dates
Define onlyNumbersPattern as a regular expression for numbers
Initialize nonDatePaths as an empty list
for all segment in split_path do
    if segment is not empty and does not match datePattern or onlyNumbersPattern then
        Add segment to nonDatePaths
        if Size of nonDatePaths is 1 then
            break
        end if
    end if
end for
return hostname + '/' + nonDatePaths.join('/') + ' / ...'
```

Algorithm 3 Ranked Search

```
score_heap ← ScoreHeap()
for all token in query do
  if token in index then
    doc_freq, doc_postings_dict ← index[token]
    for all doc_id in doc_postings_dict do
      term_freq ← doc_postings_dict[doc_id].length
      score ← calculateScore(term_freq, doc_freq)
      if score > 0 then
        score_heap.add(score)
      end if
    end for
  end if
end for
return score_heap.get_top(n)
```

Algorithm 4 Live Data Collection

```
global live_id
date = Current Datetime
articles = Fetch 20 News Articles from Guardian API between date - 1 days and date
for all article in articles do
  article_html ← Creates HTML Page for article using Database2HTML converter
  Save article_html into live directory with ID live_id
  live_id = live_id + 1
end for
```

Algorithm 5 Query Suggestion

```
Input: User's Query query
Output: List of Suggestions suggestions
if query is empty then
  return []
end if
google_api = https://suggestqueries.google.com/complete/search
header_params = {client: chrome, hl: en, q: query}
response = HTTP GET Request to google_api with header header_params
suggestions ← response.json()
return suggestions
```

Algorithm 7 Insert Term in Index Trie

```
1: procedure INSERTTERM(trie, term, posting)
2:   node ← trie.root
3:   for char in term do
4:     if char not in node.children then
5:       node.children[char] ← NewNode()
6:     end if
7:     node ← node.children[char]
8:   end for
9:   node.isEndOfWord ← True
10:  node.postings.add(posting)
11: end procedure
```

Algorithm 6 Spelling Correction

Input: User's Query *query*
Output: List of Corrections *corrections*

```
corrections ← []  
min_lev ← 99999  
if query is not empty and has only one word and is misspelt then  
    suggestions ← 100 Corrected Words from BJSPELL for query  
    for all suggestion in suggestions do  
        lev_dist ← Levenshtein Distance between suggestion and query  
        if lev_dist < min_lev then  
            corrections ← [suggestion]  
            min_lev ← lev_dist  
        else if lev_dist == min_lev then  
            Add suggestion to corrections  
        end if  
    end for  
end if  
return corrections
```

Algorithm 8 Search Term in Index Trie

```
1: procedure SEARCHTERM(trie, term)  
2:   node ← trie.root  
3:   for char in term do  
4:     if char not in node.children then  
5:       return None  
6:     end if  
7:     node ← node.children[char]  
8:   end for  
9:   if node.isEndOfWord then  
10:    return node.postings  
11:  else  
12:    return None  
13:  end if  
14: end procedure
```

B.0.1 Searching and Intersection in Postings Linked List

Algorithm 9 Search Posting in Linked List

```
1: procedure SEARCHPOSTING(postingsList, docID)  
2:   node ← postingsList.head  
3:   while node is not None do  
4:     if node.docID = docID then  
5:       return node  
6:     else if node.docID > docID then  
7:       return None  
8:     end if  
9:     node ← node.next  
10:  end while  
11:  return None  
12: end procedure
```

Algorithm 10 Intersect Postings Lists

```
1: procedure INTERSECTPOSTINGS(list1, list2)
2:   result  $\leftarrow$  EmptyList()
3:   p1  $\leftarrow$  list1.head
4:   p2  $\leftarrow$  list2.head
5:   while p1 is not None and p2 is not None do
6:     if p1.docID = p2.docID then
7:       result.add(p1.docID)
8:       p1  $\leftarrow$  p1.next
9:       p2  $\leftarrow$  p2.next
10:    else if p1.docID < p2.docID then
11:      p1  $\leftarrow$  p1.next
12:    else
13:      p2  $\leftarrow$  p2.next
14:    end if
15:  end while
16:  return result
17: end procedure
```

C Revelio Images

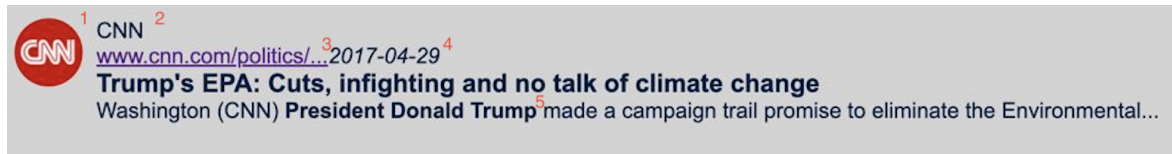


Figure 6: Result from phrase search "President Donald Trump". 1: Favicon. 2: Publisher. 3: Parsed URL. 4: Publication Date. 5: Content span with the highlighted token(s).

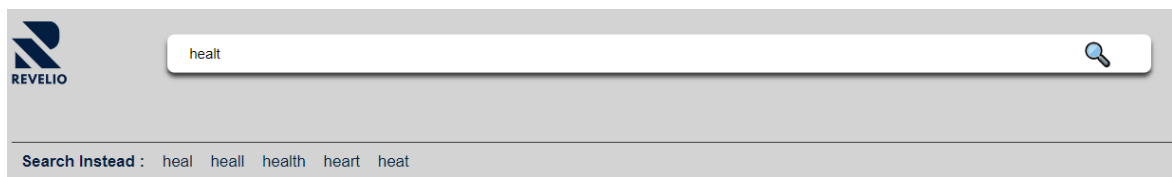


Figure 7: Correction results for the query "healt" being displayed on our search engine

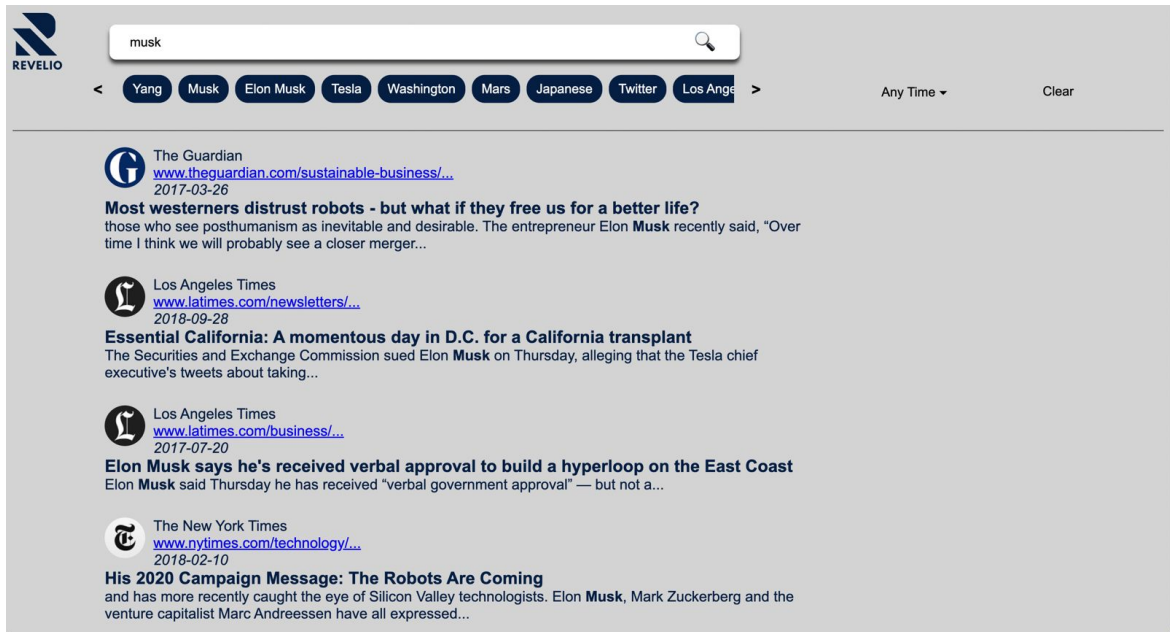


Figure 8: Search Result Page

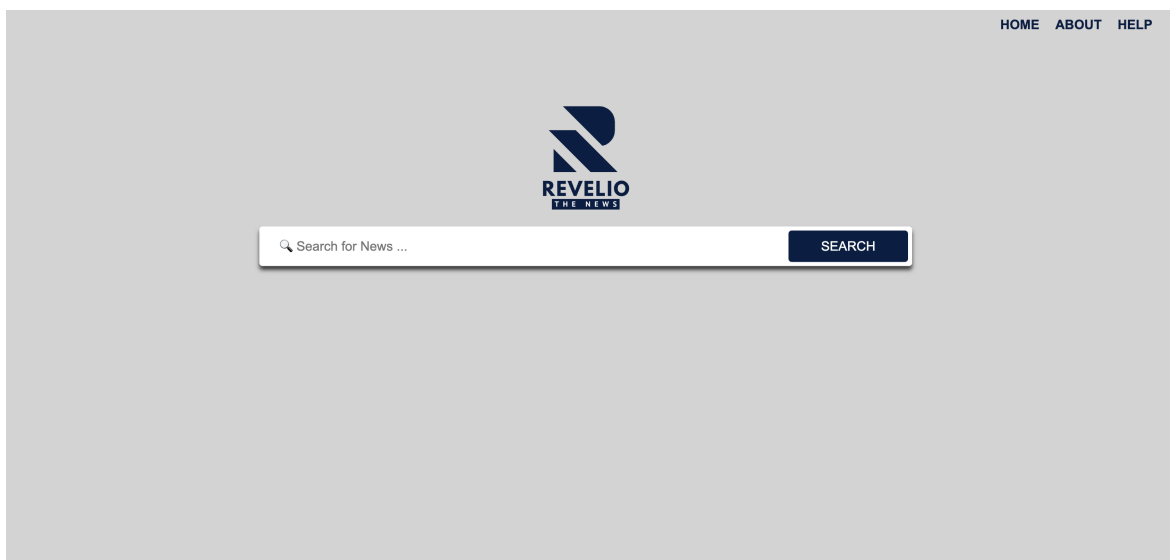


Figure 9: Revelio website Home Page

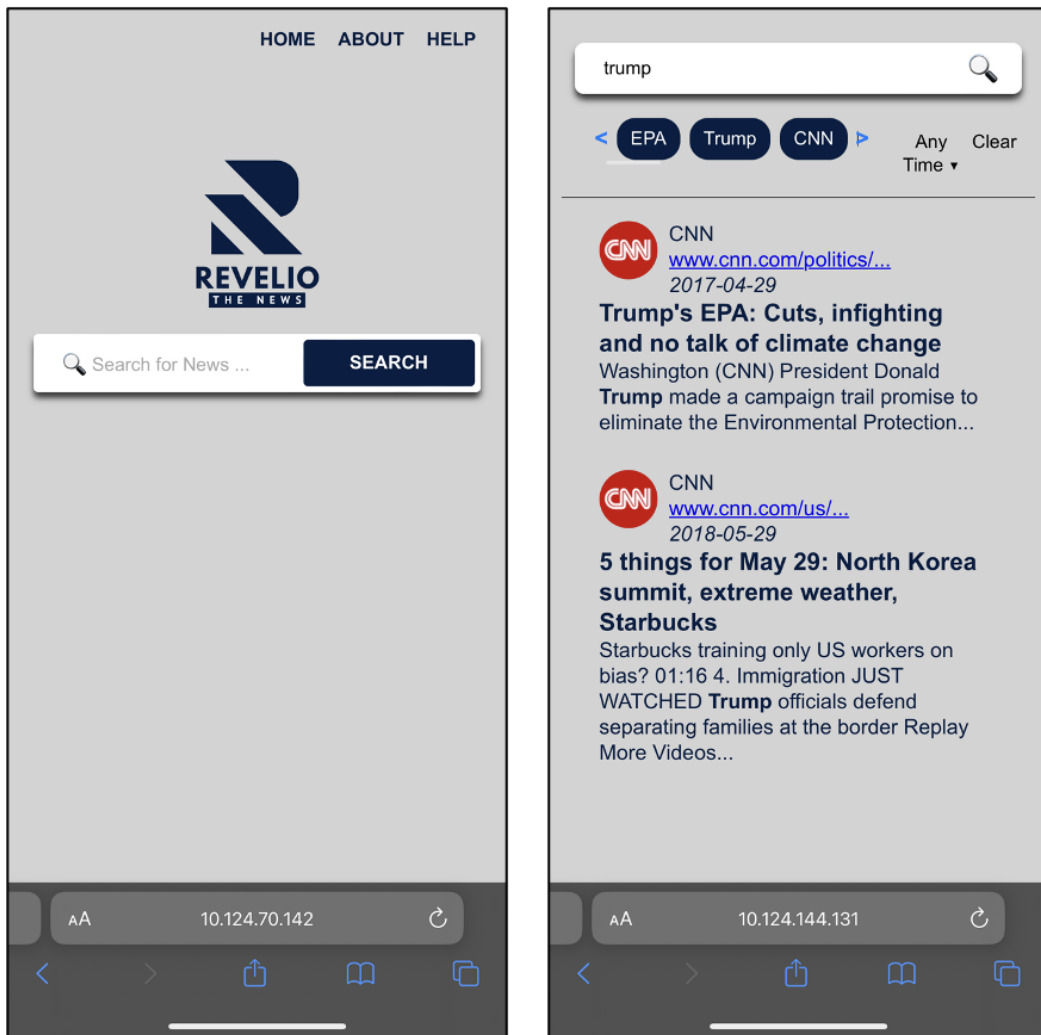


Figure 10: Revelio Mobile View



Figure 11: Suggestion results for the query "trump" being displayed on our search engine

D Compute Engine Instance Configuration

Machine configuration

General purpose	<input checked="" type="radio"/> Compute optimized	Memory optimized
Storage optimized	GPUs	

Machine types for performance-intensive workloads, with highest performance per core

Series	Description	vCPUs
<input type="radio"/> H3	High performance computing workloads	88
<input type="radio"/> C2	Ultra-high performance, compute-intensive workloads	4 - 60
<input checked="" type="radio"/> C2D	Ultra-high performance, compute-intensive workloads	2 - 112

Machine type
Choose a machine type with preset amounts of vCPUs and memory that suit most workloads.

c2d-highmem-8 (8 vCPU, 4 core, 64 GB memory)


	vCPU	Memory
	8 (4 cores)	64 GB

Figure 12: Machine configuration for the GCP compute engine instance: C2D machine type with 64 GB of RAM.

E Tables

Indexes	Structure	Usage
URL to docId	{url : docId}	File path cache
docId to URL	{docId : url}	File path cache
Index	Trie or {}	Index cache
Entity Name to Entity Id	{entity name : entity id}	Named Entities
Entity Id to Entity Name	{entity id : entity name}	Named Entities
Document Metadata	{docId : (span, {entity_id_1 : [label]})}	Named Entities
Length of Document	{doc_id : L_d (number of terms)}	BM25
Vocabulary	Set	.
Vocabulary size	Integer	.

Table 1: Indexes

Label	Description
PERSON	People, including fictional
NORP	Nationalities or Religious or Political Groups
ORG	Companies, Agencies, Institutions, etc
GPE	Countries, Cities, States
LOC	Non-GPE Locations, Mountain Ranges, etc
EVENT	Named Hurricanes, Wars, Sports Events, etc
LAW	Named Documents made into Laws

Table 2: List of accepted named entities

Search Type	Execution Time (ms)
AND / OR	60
NOT	500
Phrase	750
Proximity	500
Ranked	4100

Table 3: Execution times for search types

Ranked Retrieval Methods	IR Metrics				
	P@5	R@5	F1@5	R-Precision	AP
VSM TF-IDF	0.43	0.49	0.41	0.50	0.55
VSM BM-25	0.45	0.52	0.43	0.55	0.59
Search Retrieval OR	0.31	0.39	0.31	0.34	0.35
Search Retrieval AND	0.47	0.22	0.24	0.47	0.13

Table 4: IR evaluation results on the Time collection. Note that ‘Search Retrieval AND’ is not supported on the latest version of Revelio. This is due to us deeming it redundant.

F Vector Space Model

During our development and improvement of the positional index, we also developed a Vector Space Model ([12]) in parallel. This model represents documents and queries as a vector, where each index corresponds to one word in the vocabulary. Then, we fill these vectors with an appropriate scoring system. To retrieve documents, we calculate the cosine similarity 3 between the query vector and every single document vector. We also have the option for using either TF-IDF or BM25 scores, but we found that BM25 outputs a better result than the former. Our quantitative analysis showed that the VSM with BM25 outperformed the ranked OR and AND search by 0.12 and 0.19 F1 scores respectively 4.

Furthermore, there are already advanced modules in place to boost the VSM’s performance. Most notably, we have implemented PRF 2 (Pseudo Relevance Feedback) for query expansion when using the VSM. PRF works by trying to alter the query vector to be closer to the cluster of relevant documents and further away from irrelevant documents. However, we also noted that the initialisation time for the VSM was also unreasonable within the given time constraints. Had we had more time to flesh out our product, we would have used the VSM as our main indexing model, but we used the positional index in our current engine.

As mentioned before, we imposed a time limit on ranked searches. One worry that we had was that due to not searching through all available documents, we would miss the most relevant ones. However, upon examining the scores of the top fifty documents, we saw that empirically, there were no significant differences. I.e., there were no significant impacts on the quality of these documents. Further work should investigate this claim using hypothesis testing.