CS7330 Introduction to Web Search and Mining
Project: News Being

Zonghan Yang You Lyu Lingyue Fu Yushi Liu

June 17, 2023

1 Introduction

We created NewsBeing[1], a news search engine. The original data source is `real-news-like` from C4, which is about 40G of news-like documents. The purpose of NewsBeing is to support various kinds of queries on the huge dataset, with some advanced features enabled by current AI techniques to help users understand the news. Specifically, NewsBeing can do:

- **Boolean Search**. The search engine will present results that follow Boolean expressions provided by users. For example, when the query is `(car AND buy AND NOT rent)`, the search engine will retrieve every document that contains both `car` and `buy`, but exclude any document that contains `rent`.
- **Ranked Search**. The search engine will present documents following the order of relevance from specific queries. NewsBeing incorporates a fine-tuned scoring method based on TF-IDF to generate high-quality results. Details are in Section 2.2.
- **News Summary**. For documents generated by NewsBeing, the search engine will generate a brief summary describing the event. In this way, users can better understand the overall context of their query.
- **Interactive Q&A**. The NewsBeing search engine is able to answer simple questions about the document in natural language. For example, you can ask "When did Donald Trump win the election?", and if the document contains such information, NewsBeing will answer it correctly.
- **Q&A-Based Advanced Search**. Like New Bing, we offer an implementation of AI-aided search, which allows asking "real" questions like chatting and replies with succinct answers. The workflow is merely the same with New Bing, but due to the AI model limitations, it could answer bad answers in some cases. We add some tricks to avoid too low quality answers.

The NewsBeing search engine is separated into front-end and back-end, following common practices in real-world web services. The front-end is responsible for displaying generated documents and offers

[1] which could also be interpreted as New Bing SE

smooth interaction logic to users. The back-end is responsible for generating high-quality results with low latency and high throughput, where multiprocessing is used to enhance performance. Also, AI model inferences are introduced to provide advanced NLP support.

2 Back-end Implementation

2.1 Boolean Search

Supporting Boolean Search is the basic requirement of NewsBeing. Users are asked to input Boolean expressions to accurately state their query. Then, the search engine first parses the expression as a syntax tree and retrieves document lists for terms that appear in the query. Then, the engine performs list join operations based on the expression:

- **AND**: The engine finds documents that appeared in both lists, then produces a new list.
- **OR**: The engine finds documents that appeared in either list, then produces a new list.
- **NOT**: We re-designed the logic of NOT operation to speed up the query. Naively, the **OR** operation will return a complement document list containing documents that do not appear in the original list, which is huge in practice. Thus, instead of actually performing the NOT operation, we set a special bit to indicate that a NOT operation is performed on this list and proceed to do other operations. The NOT bit is propagated according to De Morgan's laws. Here are two examples: `(NOT a) OR (NOT b) = NOT (a AND b)`, `NOT(a OR b) = (NOT a) and (NOT b)`.

To optimize performance of Boolean Search, the operations are performed following **ascending order of list size**. At the same time, the original semantics of Boolean expression are preserved.

2.2 Ranked Search

One of the key components of a search engine is the ability to rank search results in order of relevance. In this project, we implemented a ranked search functionality using the BM25 (Best Matching 25) scoring algorithm. BM25 is a widely used ranking function in information retrieval that calculates the relevance score between a query and a document based on term frequencies and document lengths.

The BM25 score is computed for each document in the index and represents the document's relevance to the given query. The higher the BM25 score, the more relevant the document is considered to be. The ranking process involves the following steps:

1. **Query Preprocessing**: Before calculating the BM25 scores, the user query is preprocessed. This typically includes tokenization, removing stop words, and stemming, which helps to normalize the query and improve search accuracy.
2. **Term Frequency and Inverse Document Frequency**: The BM25 algorithm considers the term frequency (tf), the inverse document frequency (idf) and the query term frequency (qtf) of the query terms. The term frequency measures how often a term occurs in a document, the inverse

document frequency measures the rarity of a term across the entire document collection, while the query term frequency measures how often a term occurs in a query. These three factors are combined to compute the relevance score for each document.

3. **BM25 Scoring**: The BM25 scoring formula takes into account the term frequency, inverse document frequency, and document length. It assigns a score to each document based on the query terms and their occurrences in the document. The formula is as follows:

$$BM25(Q, D) = \sum_{q \in Q} \frac{qtf}{k_3 + qtf} \times \frac{k_1 \times tf}{tf + k_1(1 - b + bI_d/avg_l)} \times \log_2 \frac{N - idf + 0.5}{idf + 0.5},$$

where k_1, k_3 are tuning parameters, I_d means the length of document D and avg_l represents average document length in the collection.

4. **Ranking and Presentation**: After computing the BM25 scores for each document, the search results are ranked in descending order based on their scores. The top-ranked documents are presented to the user as the search results. However, in addition to relevance, we also consider the freshness of the documents in our ranking algorithm.

   To incorporate freshness, we select the top 100 relevant documents based on their BM25 scores. For each of these documents, we compute a new score that takes into account their recency. The formula for the new score is as follows:

$$score(D) = \log(BM25(D)) + \frac{0.5}{t_{now} - t_{news}},$$

   where nowtime represents current time and newstime is the publication time of the document D.

   After computing the new scores for each of the top 100 documents, we re-rank the search results based on these scores in descending order. The updated ranking ensures that the most relevant and fresh documents are presented to the user as the top search results.

5. **Relevance and Performance Improvement**: A drawback of the BM25 Scoring method is that it excessively amplifies the effect of *tf*. This scoring method is fine when dealing with regular news items, but when facing some very popular search terms, the BM25 will give high scores to documents that repeat specific words (a.k.a. SEO pollution). For example, when a user is searching for "USA President", instead of giving documents that contain both "USA" and "President" which may be what the user is asking for, the BM25 scoring method will prioritize documents that contain "USA USA USA USA USA USA" because of high *tf-idf* score. To deal with this flaw, we add some tricks to improve the ranked result heuristically:

   - Limit the too high *tf*: $tf' = \min(tf, \varepsilon)$, where ε is set to a constant 0.1. It is a reasonable limitation since it will not be a good indicator if the token occupies the document more than 1/10; intuitively, only garbage documents contain so many repeated keywords.
   - Weighted Boolean Heuristic: Intuitively, user want some “soft” Boolean results about the query, where the more keywords appeared, the better result. We give a weight, $1/i$, to the i-th appeared keywords, to award the appearance heuristically.

– Pruning: Simply extracting all lists and combining them using an `OR`-like operation could lead to bad performance when the query is long. Cooperating with the above tricks, we could do some pruning to determine the results that must not be in the top 100 and skip the results with little score contribution. After pruning, the list maintained in memory is relatively small, and latency is low even when query is long.

2.3 Index Storage and Compression

Due to the large amount of data in realnewslike dataset (about 36G), a careful design of index structure and format can greatly reduce disk space required for News Being. Several design choices are list as follows:

1. **Index Structure and Storage**: We apply the classic Inverted Index method which, when given any term, produces a list of *doc_id* that include docs containing the specific term. Besides, the *tf* values are also stored to support Ranked Search. To improve performance, instead of using raw file system, we utilize Sqlite for the storage of inverted index. The layout is illustrated in Figure 1, with each row storing the term, *doc_id* list and *tf* list. Standard SQL is used to retrieve *doc_ids* and *tfs*. Besides, we enlarge the cache size from Sqlite to reduce latency.

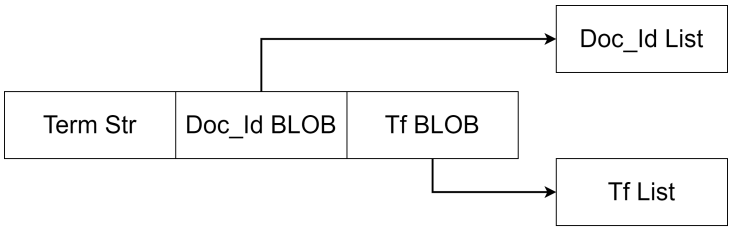

Figure 1: Storage Layout of Inverted Index for Each Term in Sqlite

2. **Index Compression**: We adopt the Google Encoded Polyline Algorithm Format (Polyline Encoding) to compress the *doc_ids* and *tfs*. [2] Polyline Encoding is a lossy compression algorithm to store a series (an array of integers or floats) as a single BASE64 string. The original usage of Polyline Encoding in Google is to compress coordinates, since a consecutive sequence of queries from geographically close areas often provides similar coordinates, which gives Google an opportunity to compress these coordinates. Here we adopt the Polyline Encoding for two reasons. Firstly, the Polyline Encoding is widely adopted in various applications and already has many reliable implementations. Secondly, the distribution of *doc_ids* is very similar to the coordinates, for consecutive *doc_ids* are close to each other. In this case, instead of compress the original values, the Polyline Encoding **compress the differences** between *doc_ids* to maximize compression ratio.

[2] https://developers.google.com/maps/documentation/utilities/polylinealgorithm

2.4 Other Design Practices

2.4.1 Multiprocessing

Due to the large amount of data in realnewslike dataset, we adopt multiprocessing to further improve performance. Figure 2 presents detail architecture of the NewsBeing search engine.

The entire dataset is divided into 16 separate shards, each of which is assigned a worker thread responsible for retrieving and processing index structures. In the case of Boolean Search, the worker thread handles index retrieval and joins document lists based on the requested Boolean query. For Ranked Search, the worker thread calculates BM25 scores for each document and generates a sorted document list for its specific shard. Additionally, a centralized Gateway acts as a proxy. Upon receiving an HTTP request from the front-end, it dispatches index processing tasks to all workers asynchronously and awaits their responses. Then, the Gateway combines the responses received from the worker threads to generate the final results. In Boolean Search, the Gateway randomly samples a few (typically 200 in NewsBeing) documents as the final response since all documents produced by the worker threads are considered valid and equally important. On the other hand, in Ranked Search, the Gateway combines the document lists using Multi-way List Merging based on their scores to produce the top 200 documents.

We have identified two key benefits of implementing multiprocessing in NewsBeing. Firstly, the implementation of sharded data mechanism greatly enhances the scalability of the search engine. By leveraging multiple threads to handle user requests, the response latency is significantly reduced. Furthermore, the separated data shards ensures that the engine can easily accommodate future increases in the number of incoming web pages by adding more servers. Secondly, the asynchronous design of NewsBeing effectively decouples responsibilities of each component, leading to improved overall throughput. This design allows the engine to process tasks independently, thereby increasing efficiency and maximizing the utilization of system resources.

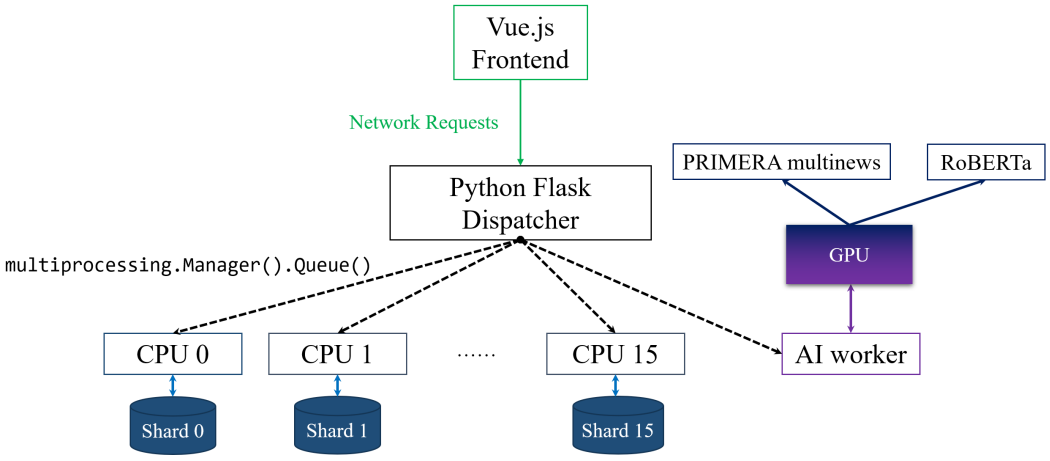

Figure 2: Search Engine Detail Architecture

2.4.2 Sqlite Index

Since we adopted Sqlite as our database for index and tf retrieval, we carefully configured the Sqlite index to speed up the retrieval process. Specifically, we built the index on the "term" attribute because most of the queries are based on specific terms. Thus, instead of searching through the entire table, Sqlite can now quickly retrieve corresponding data items.

2.4.3 LRU Cache

To further improve performance, we also included an LRU cache for fast document retrieval. When a user gives the same query that appeared a period of time ago, the search engine will directly retrieve results and return them to the front-end. In this case, many hot queries will benefit from the cache, which further improves the performance for NewsBeing; it also reduces the burden to design RESTful API, where the same query may be invoked for many times to facilitate better front-end experience.

2.4.4 C++ Incorporation

We also tried rewriting some of the Boolean Search logic in C++ to improve performance. Results showed that the improvement was unnecessary compared to the cost of maintaining cross-language compilation, so we actually deprecated the C++ sources.

3 Front-end Implementation

The front-end web pages are separated from back-end search engine to ensure separate deployment, which is a common practice in today's web projects. It should be noticed that front-end and back-end are deployed on the same server for brevity, but they can also be deployed on separate servers to maximize throughput. Several HTTP interfaces are prepared on back-end side. The front-end web pages acquire search responses by sending HTTP requests.

In NewsBeing, we adopted Vue as our front-end framework for its ease of use. We selected many key UI components from Ant-Design. Axios is used as our asynchronous HTTP communication framework. More details of web pages are presented in Section 5.

4 AI Augmentation

Based on the existing functions of NewsBeing, we added an AI model at the back end to realize the functions of news summary and question and answer.

4.1 Multi-news Summarization

In order to summarize several news articles, we organized a pre-trained model named PRIMERA-multinews[3]. The input of documents is concatenated by the token `<doc-sep>` with some metadata, and

[3]https://arxiv.org/abs/2110.08499

fed to PRIMERA. The model takes in multiple news documents as input and generates a summary that captures the most important information from each document.

PRIMERA-multinews incorporates several key design components to achieve its high performance. One of these is the use of a latent variable model, which allows the model to capture the underlying structure of the input documents and generate summaries that are coherent and informative. Another important component is the use of a hierarchical attention network, which enables the model to focus on the most relevant parts of each document when generating the summary. The model also includes a reinforcement learning-based training strategy, which helps it optimize for the task of summarization by directly maximizing the quality of the generated summaries.

4.2 Interactive Q&A

We use the pre-trained language model RoBERTa [4] to implement QA task. In the News Detail page, user can ask questions, and the model will find the keywords towards the question within the context.

4.3 Q&A-Based Advanced Search

Remember the New Bing has the following structure (Figure 3) mentioned in the last lecture[5]:

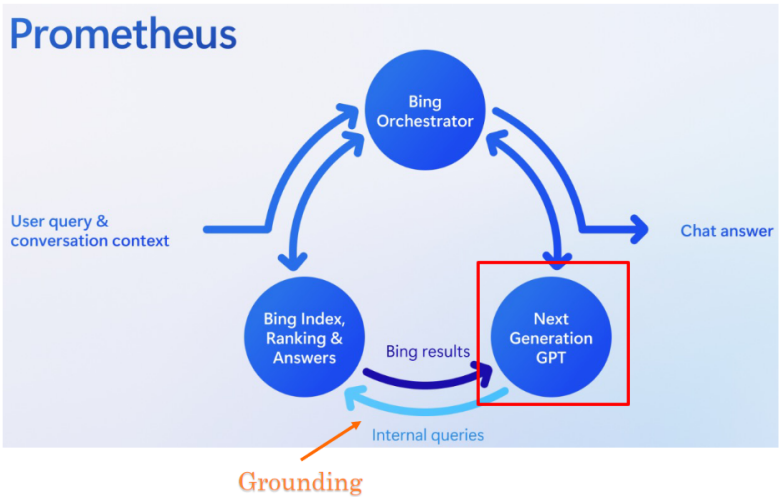

Figure 3: The New Bing Structure from slides.

Deploying a SOTA GPT model seems quite off-topic and computational expensive; however, with the above AI models, we could finish similar tasks, with the following procedure as shown in Figure 4.

- Extract the keywords using the QA model;
- Using the Ranked Search to retrieve the related documents;

[4] https://arxiv.org/abs/1907.11692
[5] https://www.cs.sjtu.edu.cn/ kzhu/wsm/L10.pdf

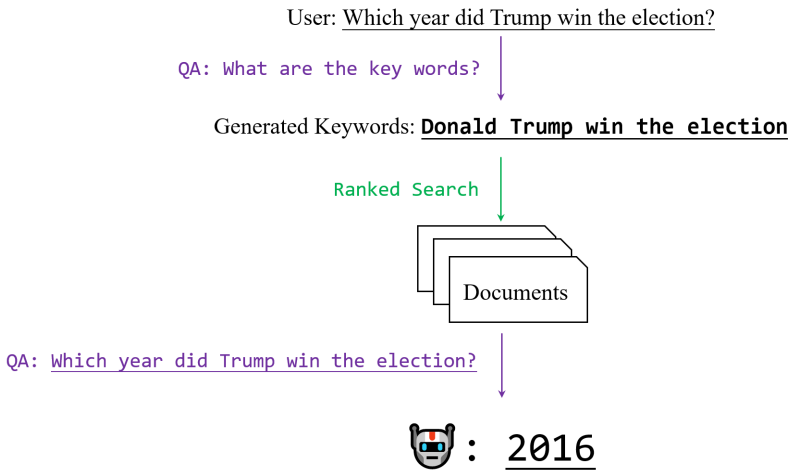

Figure 4: The Chat QA infrastructure.

- Answer the user's question using the retrieved documents with QA model.

Though the answer is not fluent since the QA model can only repeat the related sentences, experiments showed that it could successfully answer Q&A questions accurately.

5 Results

5.1 Boolean Search

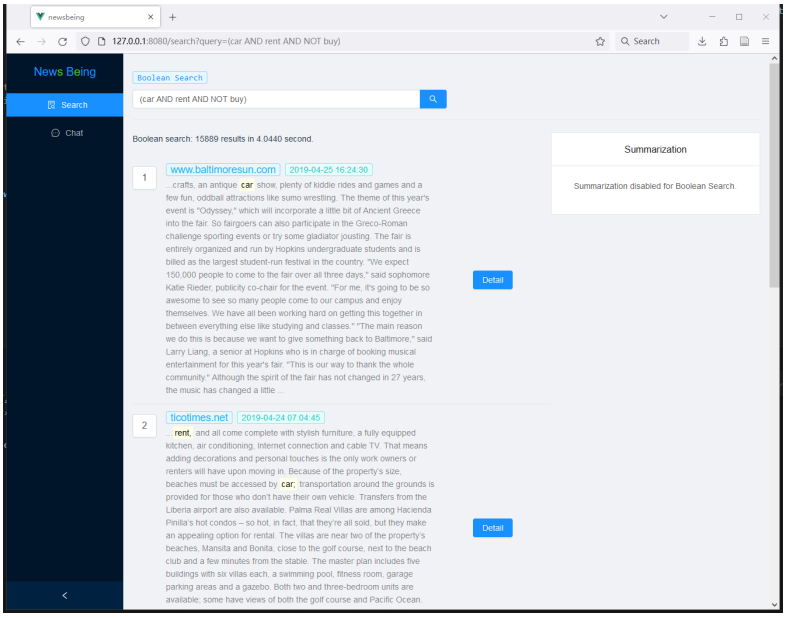

Figure 5: Boolean Search: `(car AND rent AND NOT buy)`

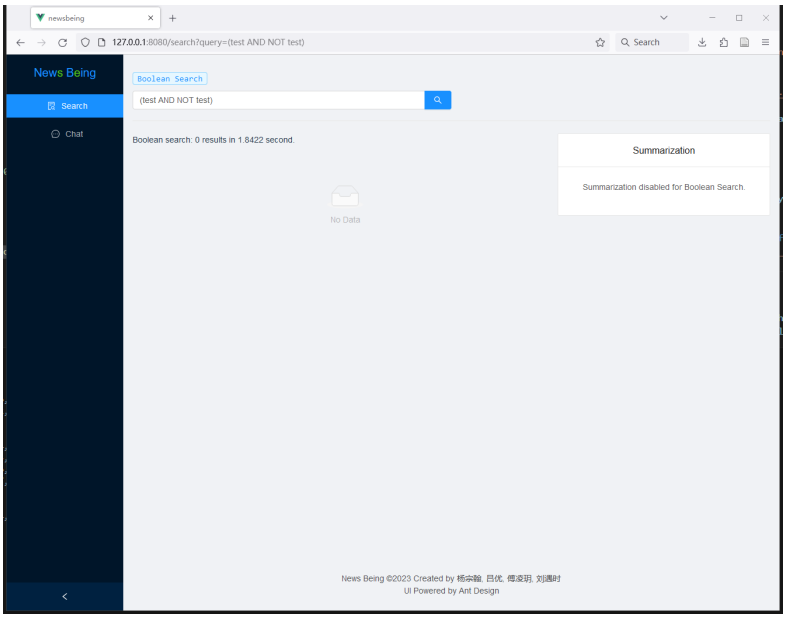

Figure 6: Boolean Search: `(test AND NOT test)`

5.2 Ranked Search and Summary

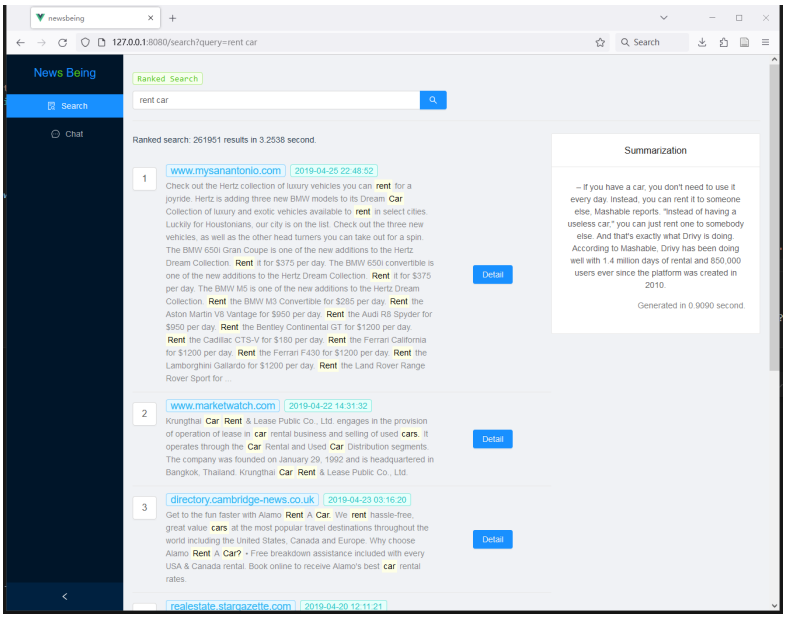

Figure 7: Ranked Search: `rent car`

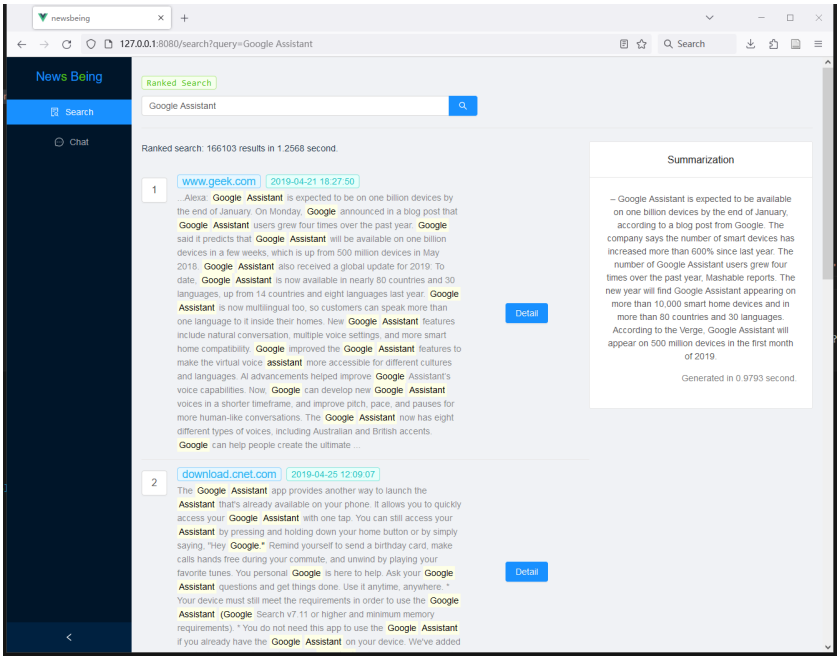

Figure 8: Ranked Search: `Google Assistant`

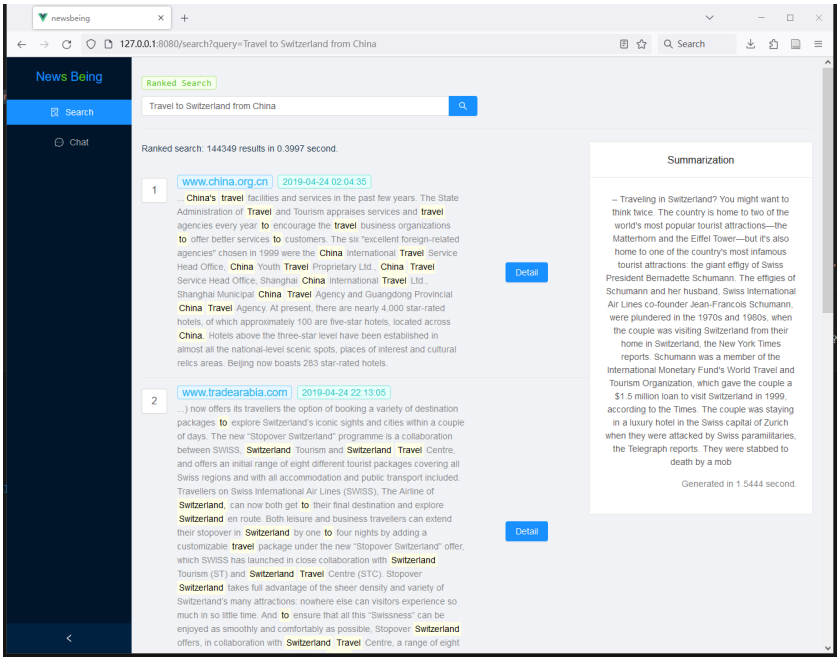

Figure 9: Ranked Search: `Travel to Switzerland from China`

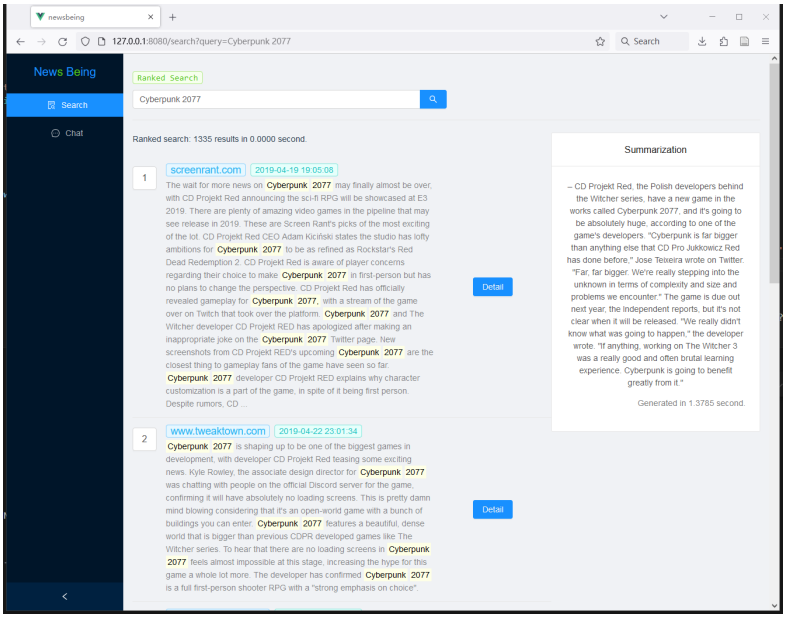

Figure 10: Ranked Search: `Cyberpunk 2077`

5.3 Interactive Q&A

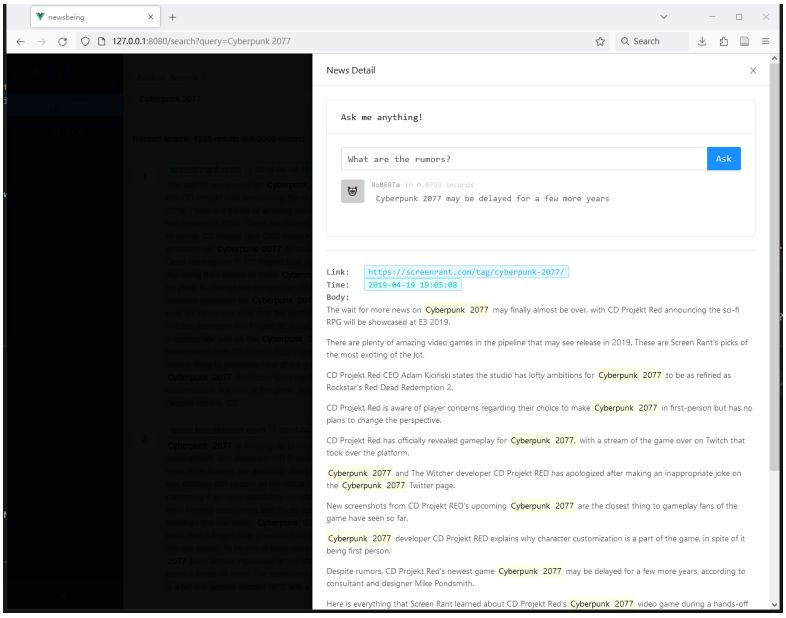

Figure 11: Q&A: `Cyberpunk 2077: What are the rumors?`

5.4 Q&A-Based Advanced Search

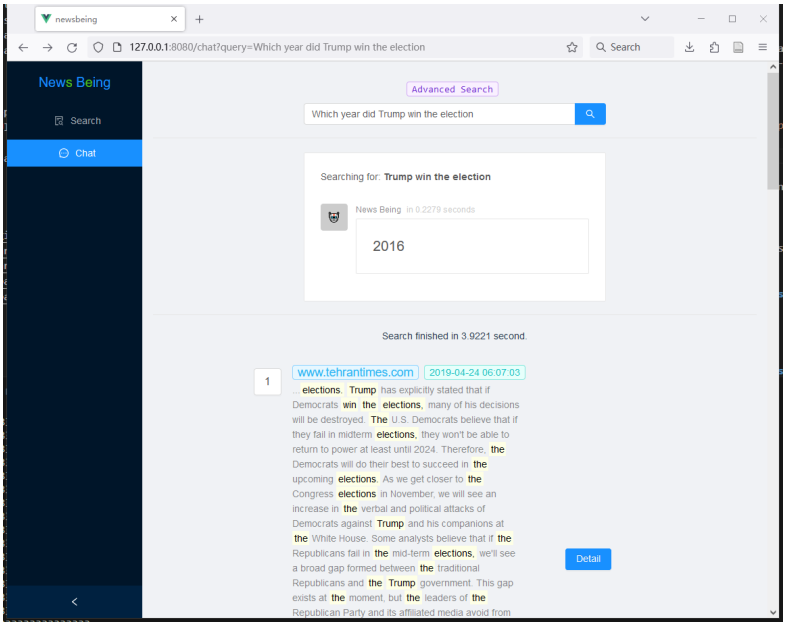

Figure 12: Chat Search: `Which year did Trump win the election?`

5.5 Poor result rejection

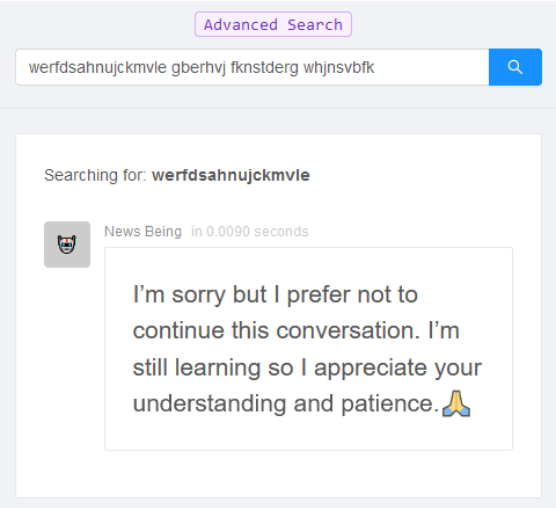

Figure 13: Ms. News Being doesn't want to answer this question.

5.6 Performance Evaluation

Here we present several comparisons on key design choices.

5.6.1 Reducing Latency

Table 1 shows latency comparisons when adopting different mechanisms. In this experiment, we use the Boolean query "(car AND buy AND NOT rent)", and the NewsBeing search engine produces 82902 results. In general, building Sqlite Index and LRU Cache are good design pratices, for they greatly reduce the end-to-end latency. Rewriting part of logic in C++ does not improve much performance for us.

Sqlite Index	In LRU Cache	Language	Time
No	No	Python	188s
Yes	No	Python	1.78s
Yes	No	Python, C++	1.66s
Yes	Yes	Python	<0.0001s

Table 1: Latency Results

5.6.2 Reducing Disk Space

Table 2 shows the disk space allocated before and after index compression. In general, the Polyline Encoding saves about 37.5% on the size of index. At the same time, the compression brings some overhead to queries due to additional decompression phase, but the overall latency isn't much affected.

Compression	Preprocess(s)	Boolean Query(s)	Ranked Query(s)	Total Size(GB)	Index Size(GB)
No	2160	0.78	0.32	63	18.71
Yes	2250	0.756	0.38	56	11.68

Table 2: Total and Index Size

5.6.3 Improving Relevance

Figure 14 and 15 shows the generated results before and after Relevance Improvement. The latter method produces results that contain both "USA" and "President", while the former produces results that only contain "USA" because of high *tf*.

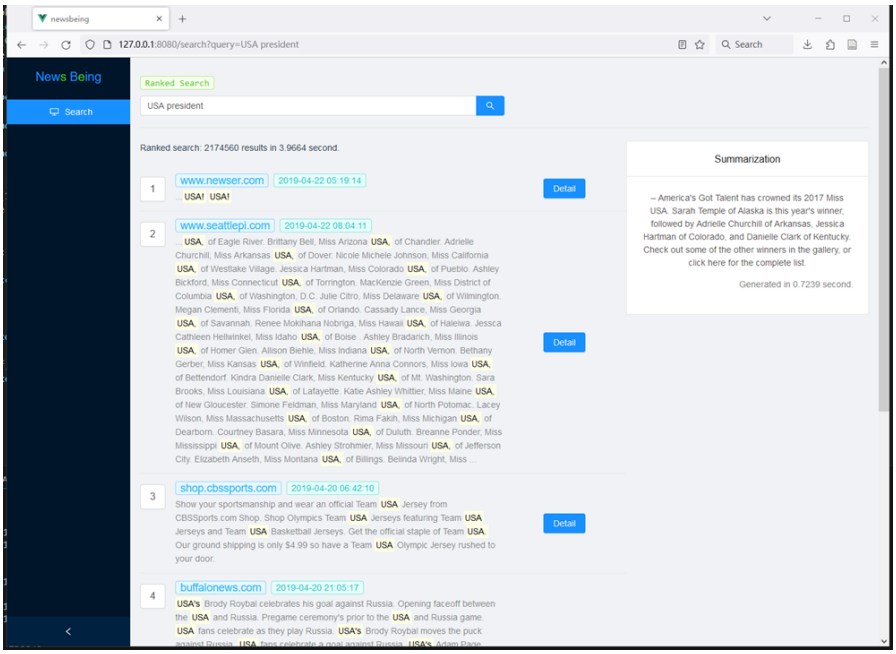

Figure 14: Before Relevance Improvement

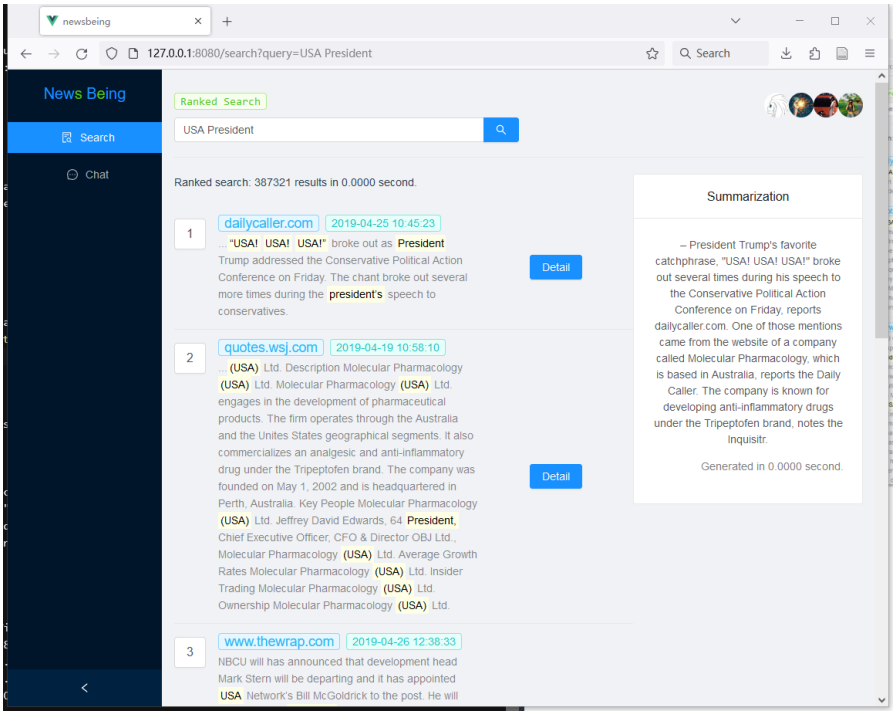

Figure 15: After Relevance Improvement