Tübingen-Focused Web Search System

Farha Baig **Arkadii Bessonov** **Lalit Chaudhary** **Ali Gharaee**
Maximilian Selzer **Stephen Tafferner**

July 21, 2025

Introduction

The Tübingen-Focused Web Search System is a custom search engine designed to surface high-quality, English-language web content relevant to Tübingen. Combining a tailored crawling strategy, hybrid BM25 and dense vector indexing, and a generative assistant, it delivers semantically rich and visually intuitive search results. [1]

Crawling

The crawler in the `Crawler` module goes well beyond a basic URL fetcher by combining robust error handling, relevance-aware metrics, and asynchronous request batching.
We implement a sophisticated HTTP error-detection system in `statusCodeManagement`. Whenever a request to a URL fails, we evaluate the "badness" of the response and, if warranted, add the URL to `main.disallowedURLCache`. Simultaneously, an Unbiased Time Exponential Moving Average (UTEMA) is maintained, as described in [1], over status-code weights that is assigned per error type. If this moving average exceeds a configurable threshold—and recent calls to that domain have clustered in time—the entire domain is marked in `main.disallowedDomainsCache`, preventing further crawls there. This time-weighted average gives exponentially less influence to older errors (tunable via the parameter β), making it particularly sensitive to recent spikes in failures that often indicate temporary blocks.
The discovered URLs are stored in a frontier structure (`main.frontier`). On visiting a page, we parse its HTML or XML, extract all hyperlinks, and enqueue them via `frontierManagement.frontierWrite`. For each fetched URL, its human-readable is cached in text and, if present, its title in `main.cachedUrls`. To guide crawl relevance—especially given the Tübingen-focused seeds in `seed.py`, two graph-based metrics are computed:

- **linkingDepth**: the minimum number of inter-domain hops from any seed to the current domain.
- **domainLinkingDepth**: the shortest in-domain link chain from the first-reached page on a domain to the current URL.

To maximize throughput without triggering site bans, our crawler issues up to N parallel HTTP requests (configurable) using asynchronous calls. Crucially, `frontierManagement.manageFrontierRead` ensures that concurrently fetched URLs originate from distinct domains, distributing load and reducing block risk. For more information on the crawler and its structure see the README.md file in the crawler - folder.

Optimization Approach

To boost crawling throughput, we introduced several low-level and infrastructural improvements. First, the multiprocessing pipeline was overhauled to use thread-safe queues with explicit locking and graceful shutdown hooks, ensuring that all database I/O completes reliably even under high load. Second, we replaced heavyweight full-page parsers with targeted regular expressions for link, title, and metadata extraction, cutting parsing time by more than half.

Quality Enhancements

Beyond raw speed, two strategies were applied to improve the relevance and diversity of the collected content. First, a domain-diversity policy was enforced: domains that exceeded 1,000 total crawled pages (for example, wg-gesucht, GitHub, and dai-tuebingen) are automatically excluded to avoid overrepresentation. This change noticeably broadened our topical coverage. Next, we refined our seed list by ranking all candidate URLs by relevance, selecting the top 6,000, removing domain duplicates, and assembling a balanced set of 3,500 seeds. The result is a seed set that drives the crawler toward both high-relevance and high-diversity

[1] You can find the GitHub repository at `https://github.com/StephenTaf/Modern-Search-Engines-Project` and database at `https://huggingface.co/datasets/lalit3c/Tuebingen-Crawled/blob/main/crawlerDB.db`

content, underpinning better search results for our Tübingen-focused queries.

Metric

The metric `metric.py` computes a `tueEngScore` for each crawled page, quantifying its relevance to Tübingen-focused English content. It combines three signals: URL (Tübingen-related keywords), text (English language and frequency of city or academic terms), and incoming links, into a weighted sum scaled by a linear penalty based on link distance. An `OfflineMetric()` variant recalculates scores post-crawl, incorporating outgoing links and applying steeper decay for deeper pages.

Indexing

Data Preprocessing Pipeline

Our preprocessing pipeline integrates data from multiple crawling sessions by merging outputs from various crawled databases reflecting the distributed nature of the crawling process.

URL Normalization Strategy

To improve duplicate detection, we normalize URLs by removing protocol prefixes, query parameters, and trailing slashes. The original URLs are retained for reference, while the normalized form is used as the deduplication key.

Dual-Phase Duplicate Removal

Deduplication is performed in two steps: first, we exclude URLs already present in historical datasets; second, we remove duplicates within the new data. This ensures both cross-session consistency and internal uniqueness.

Language Detection

We combine `langdetect` and `polyglot` to classify English content, using a 15% confidence threshold to accommodate multilingual pages. This approach favors recall, minimizing the risk of discarding relevant English material due to low-confidence predictions.

BM25 Indexing System

Database Schema Design

We use a normalized four-table schema to support efficient updates and modular storage:

- **`bm25_doc_stats`**: Document metadata (doc_id, doc_length, processing_timestamp)
- **`bm25_term_freq`**: Term frequencies per document (doc_id, term, frequency)
- **`bm25_term_stats`**: Global term statistics (term, document_frequency, total_frequency, idf_score)
- **`bm25_corpus_stats`**: Corpus-level metrics (average_document_length, total_documents)

This separation enables efficient incremental updates. Precomputing IDF scores and corpus statistics reduces runtime overhead.

Text Processing Pipeline

Documents are indexed by combining title and content, applying Unicode normalization (e.g., "Tübingen" variants), enforcing a 1M character limit, and converting text to lowercase.
We use spaCy's **en_core_web_sm** model for lemmatization, stop word removal, and filtering of punctuation and non-alphabetic tokens. Lemmatization improves recall by consolidating word variants, while spaCy's tokenizer outperforms simple splitting.

Scalable Processing Architecture

The system processes data in adaptive batches (default size: 5,000), with parallel execution triggered for batches of 50 or more documents. Worker count scales with available CPU cores.
Only unprocessed documents are selected using LEFT JOINs, ensuring idempotent incremental updates. Performance is enhanced through bulk database operations, index-aware lookups, and cached IDF values.

Dense Vector Indexing System

Inspired by the ColBERT [2] model's high-granularity relevance matching, we adopt a dense reranking approach with granularity level. While ColBERT operates at the token level:

$$\{\mathbf{v}_{t_1}, \mathbf{v}_{t_2}, \ldots, \mathbf{v}_{t_n}\} \quad \text{(token embeddings)}$$

this generates excessive vectors for practical deployment. To balance granularity with efficiency, we use *window-level embeddings*:

- Split documents into fixed-size overlapping chunks: $w_{\text{size}} = k$, $w_{\text{overlap}} = m$
- Embed query $\mathbf{q}$ and all chunks $\{\mathbf{d}_{c_1}, \mathbf{d}_{c_2}, \ldots, \mathbf{d}_{c_k}\}$
- Compute document-query similarity as maximum chunk similarity:

$$\text{sim}(d, q) = \max_i \left(s(\mathbf{d}_{c_i}, \mathbf{q}) \right)$$

This approach provides three key advantages:

1. **Granularity**: Locates relevant passages in long documents
2. **Memory Efficiency**: Fewer vectors than token-level methods
3. **Computational Efficiency**: Avoids full-document embedding

Embedding Generation

For embedding generation, we fine-tuned ModernBERT [3] specifically for retrieval tasks. Training methodology and parameters are detailed in Section 'Model Training'.
To optimize for storage during indexing as well as fast inference during retrieval we adopt following strategies:

1. Similar to BM25 indexing, we process only the unindexed documents instead of doing a force reindex with every corpus update
2. **Precomputation**: Generate document chunk embeddings $\{\mathbf{d}_{c_i}\}$ during indexing in batches
3. Optimized database I/O with batches and transactions for bulk updates and inserts
4. Dynamic index management (drop during insertion, rebuild after) to prevent performance degradation during insertion
5. **Real-time Processing**: Compute only query embedding $\mathbf{q}$ during search
6. **Similarity Matching**: Calculate $\max(s(\mathbf{d}_{c_i}, \mathbf{q}))$ through batched vector operations during runtime

Model Training

Our training implementation is available in the `embedder_training` directory of the main repository. We fine-tuned ModernBERT with an added classification head using a cosine similarity objective function. We used the `sentence-transformers/gooaq` [4] dataset from HuggingFace, processing it as follows:

- **Hard Negative Mining**: Employed the pretrained `mrl-en-v1` embedder to identify challenging negatives
- **Sampling Strategy**: For each query-positive pair (q, p^+):
  - Retrieved top-k candidates from passage pool using `mrl-en-v1`
  - Selected hardest negatives with top similarity scores
- **Efficiency Consideration**: Avoided multi-stage training by leveraging existing pretrained embedder

Table 1: *Training setup parameters*

Parameter	**Value**
Base model	ModernBERT
Training pairs	1.5 million $(q, p^+, \{p_i^-\})$
Hardware	NVIDIA H100 GPU
Batch size	1024
Training time	∼4 hours

This single-stage training approach achieved competitive performance while optimizing computational resources, avoiding the need for iterative negative mining cycles. Our training setup parameters are represented in Table 1.

Query processing

In our pipeline, we adopt a hybrid reranking approach with several enhancements tailored for relevance to Tübingen and user experience:

- **Query Augmentation**: For every incoming query, we append the keyword "tübingen" if it is not already present. This increases the probability of retrieving content relevant to the city and region.
- **Document Retrieval**: We retrieve the top 1000 candidate documents using a BM25 scoring function.
- **Embedding-Based Reranking**:
  - We load precomputed embeddings of these top 1000 document chunks from DuckDB.
  - During search, we compute only the query embedding $\mathbf{q}$.
  - We calculate a similarity score for each document using $\max(s(\mathbf{d}_{c_i}, \mathbf{q}))$, where $\mathbf{d}_{c_i}$ denotes the i-th chunk of a document. This is done efficiently via batched vector operations.
- **Relevance Scoring**: We combine BM25 and embedding-based similarity using a weighted average:

  $$\text{score} = 0.15 \cdot \text{BM25} + 0.85 \cdot \text{Similarity}$$

  The weight of 0.15 for BM25 was empirically determined based on testing with a small set of representative queries. To account for the fact that BM25 scores and embedding similarities are not on the same scale, we normalize each set of scores to the [0,1] range within each pool of retrieved documents (i.e. the top 1000). This ensures fair combination by aligning both scores onto a common scale, preventing one from dominating due to magnitude alone. The resulting weighted average balances lexical precision (BM25) with semantic generalization (embedding similarity), improving retrieval robustness across diverse query types.
- **Title Boosting**: If the most similar chunk is the first chunk of a document (typically containing the title or introduction), we apply a small boost to the final score. This heuristic assumes that high similarity in the opening section correlates with stronger overall document relevance.
- **Domain Diversity Filtering**: We observed that the top results often come from multiple pages within the same domain, which, while relevant,

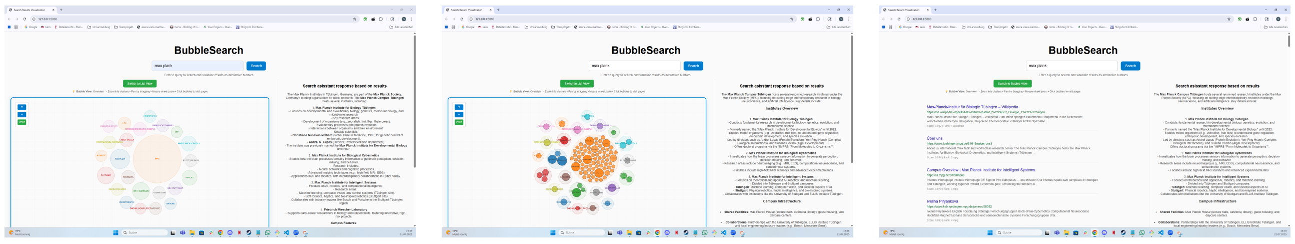
(a) *Cluster-level force simulation* **(b)** *Bubble view of result page* **(c)** *List view of result page*

Figure 1: *User interface views: (a) Cluster-level simulation; (b) Full bubble view; (c) List view.*

can reduce the diversity of search results. To address this, we:

- Group documents by domain.
- Categorize documents into *high* and *medium* relevance groups based on reranking scores.
- In the final ranking, retain only one document per domain in the top results, prioritizing diversity. Additional documents from the same domain are appended afterward in order of relevance.

Search assistant

Inspired by Google's AI-powered search summaries, we implemented a generative interface that synthesizes answers from the most relevant text windows identified by our reranking approach. The pipeline operates as follows:

1. **Relevant Context Extraction**:
   - Retrieve top-k document chunks: $\mathcal{C} = \{c_1, c_2, \ldots, c_k\}$
   - Select windows with maximum similarity: $\text{sim}(c_i, q) > \tau$
2. **Generative Answer Synthesis**:

   $$\text{Answer} = \text{Qwen3-235B}(\texttt{[prompt\&docs][query]})$$

3. **API Integration**:
   - Model: Qwen3-235B via HTTP API
   - Context window: 128K tokens
   - Input format: Structured with query-context pairs

This architecture provides three key benefits:

- **Grounded Responses**: Answers directly reference retrieved evidence
- **Efficiency**: Leverages pre-filtered high-relevance chunks
- **Freshness**: Decouples generative model from document storage

Search Result Presentation

The website begins with a minimalistic landing page that displays the name of our search engine, **BubbleSearch**, and a search input field. Once a query is entered, the system retrieves a list of 100 documents, each associated with a score and domain.
Instead of displaying the results as a simple list, we opted for a more visual and interactive approach by using the (`D3.js`) library's force simulation to generate a bubble chart. This chart visually represents the 100 documents, grouped by domain.
To create the visualization, the documents are first aggregated by domain. For each domain, we compute a total score by summing the scores of all its documents. Each domain is then represented as a cluster—a circle whose radius is scaled according to this total score.
In the first force simulation, these domain clusters are positioned using forces that draw them toward the center while maintaining spacing (via collision forces) for better readability. Once the cluster positions are fixed (see Figure 1a),a second force simulation is ran within each cluster. This positions the individual documents, which are also represented as circles, drawn toward their respective cluster centers while avoiding overlap. The radius of each document circle is scaled according to its individual score (see Figure 1b).
To enhance usability, the top 10 documents are rendered with higher opacity. Hovering over a document highlights it and displays a brief preview of its content. Clicking on a document redirects the user to its associated URL.
In addition to the bubble view, we provide a traditional list view where documents are ordered by their score (see Figure 1c). Users can toggle between the two views. On the right side of the screen—regardless of the selected view—we display a response generated by our search assistant. This response summarizes the content of the top 10 results into a coherent and informative paragraph.

References

[1] Michael Menth and Frederik Hauser. “On moving averages, histograms and time-dependentrates for online measurement”. In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. 2017, pp. 103–114.

[2] Omar Khattab and Matei Zaharia. *ColBERT: Efficient and Effective Passage Search via Contextualized Late Interaction over BERT*. 2020. arXiv: `2004.12832 [cs.IR]`. URL: `https://arxiv.org/abs/2004.12832`.

[3] Benjamin Warner et al. *Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference*. 2024. arXiv: `2412.13663 [cs.CL]`. URL: `https://arxiv.org/abs/2412.13663`.

[4] Daniel Khashabi et al. “GooAQ: Open question answering with diverse answer types”. In: *arXiv preprint arXiv:2104.08727* (2021).