An Efficient and Fast Search Engine for Academic Papers using Hierarchical Navigable Small Worlds

Balraj Gill, Dale McDiarmid, Lorenzo Baldini,
Luwei Wang, Saad Sharif, Vincenzo Incutti

Contents

1 Useful Links

The Search Engine can be accessed via the following link: `http://ec2-34-245-10-248.eu-west-1.compute.amazonaws.com/?q=test&size=n_10_n`. The repository containing the code is openly accessible at `https://github.com/saadsharif/ttds-group`.

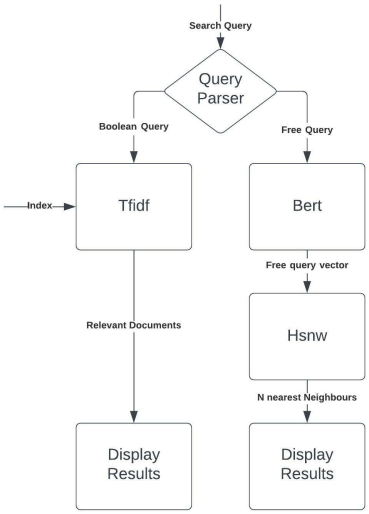

Figure 1: Flow chart of the system

2 Corpus and Data processing

2.1 The arXiv corpus

The corpus we have adopted in our project is composed of academic papers from arXiv[6], an open-access repository of research papers treating mostly scientific subjects such as computer science, physics, mathematics and economics. The documents in such a corpus classify as large according to the project specification, thus we limited ourselves to papers submitted to the website from 2021 onwards. While new papers are scraped every 6 hours (see 2.2 Live Data Collection), we have a total of 181,188 documents as of March 21st 2022. These were obtained via an Amazon S3 bucket by using the bulk access service that arXiv provides.

2.2 Live Data Collection

To ensure fresh data is always available to the user, we employ a **scraper** agent to periodically fetch new papers from arXiv. While not needed at our scale, the scraper is designed to be fully **scalable**, with a central scheduler spawning workers to fetch data for a specified time period. We currently run it with only one worker – enough for the volume of papers released in a day. It interfaces with the Indexing/Querying engine through an HTTP endpoint, making it fully **distributed**, **independent** of the IR engine, and easily **extensible**, for example to scrape other archives.

2.3 Data Pre-processing

As the first step in the data cleaning process, the raw text extracted from the PDF files was stripped of newlines and unnecessary strings such as random single letter characters, while taking care to keep emails, links and references. Then, tokenisation and stemming are performed before indexing. The same process is performed on queries before evaluation to ensure the resulting terms are identical for lookup in the term dictionary. Specifically:

1. Text is tokenised using the `re.split` function with the regular expression `\W+`, which splits on all non-word characters. For each document or query, this produces a list of tokens. This represents an aggressive tokenisation technique that doesn't preserve characters such as hyphens (see alternative tokenisation approaches considered in Section 10);

2. Each token produced in step (1) is then

   (a) Case folded using `lower()`. This is crucially performed before stop word removal to ensure case variants are not required in the stop word list, e.g. "And", "and";

   (b) Dropped if it matches a pre-defined stop word list [8];

   (c) Stemmed using the Porter stemming algorithm - via the package `Stemmer` [15]. Profiling indicated this to be the most performant stemmer.

All fields passed during indexing are concatenated into a single string. **Positions are calculated once tokenisation and stemming have been performed**. Note that this can result in false positives for phrase and proximity matching, e.g. the query "the cat jumped over the dog" becomes "cat jumped dog" with positions 1,2,3. This phrase could potentially be matched against "a cat jumped under a dog". Exact phrase matching may also result in false positives due to stemming.

Tokenization also preserves the original token for a term (i.e. its first occurrence) for use in the query expansion module. These unstemmed forms are, however, not indexed.

3 Index

In order to service boolean and proximity queries we maintain an inverted index. This section discusses the structure of this index, its storage format on disk and how queries are executed. Free text queries, defined as those with the absence of any Boolean operator, are routed to HNSW vector matching (Section 7). However, the structures described here are still utilised for ancillary operations such as filtering and faceting.

3.1 Assumptions

1. Documents are provided in JSON format. For bulk indexing, ndjson (newline-delimited JSON [7]) is supported. All communications occur through a REST interface.

2. Indexing is single-threaded. Only one thread can update the index at any time. During a batch update, the index structures are available only for reading. Other indexing threads will be blocked from writing until the previous batch completes.

3.2 Document Ids

All documents receive a unique internal document identifier. This is a monotonically increasing integer starting at 1. Users must provide a unique external identifier that can be any arbitrary string. A bi-directional, in-memory mapping, is in turn maintained between the two identifiers. This internal document identifier ensures that postings for a term are sorted. This aids query evaluation and allows efficient algorithms such as Linear merge to be utilised (see Query Evaluation, Section 4.2). All of the document structures described below use this internal document id. As a result, query evaluation also returns internal document ids, which are mapped back to the original ids before being returned. The dictionary used to store these mappings represents a low memory overhead and allows document ids to be arbitrary.

3.3 Index Structure

3.3.1 Misc Structures

The following data structures are peripheral to actual search but used during or after query evaluation:

1. A LMDB Lightning database [16][17] for documents themselves. Documents are persisted as JSON with their internal id used as the primary key. This structure is only used to return the original documents in results once hits have been identified.

2. Bi-directional in memory dictionary of the internal to external id i.e. a lookup can be performed in any direction. This structure is used to ensure indexed documents are not duplicated and have unique ids. All internal query evaluation occurs using internal identifiers. These are in turn converted back to external ids when returning results to the user.

3.3.2 Internal Store

In order to ensure the index is not memory bound, most search index structures are overloaded to disk. We use direct IO and rely on File System caching for performance. The principal storage structure is a persistent HashMap Store. This store is backed by a file. Any string value can be used as the key (usually a term or internal document id). These keys are held in memory as well as on-disk - allowing them to be loaded into memory on restart. Insertion order of the keys is preserved. The value of this hashmap represents an offset to the position on the disk. This offset is held in memory with the true value on disk only until accessed. When accessing the value, the Store internally accesses the file, seeks to the relevant offset and returns the bytes stored. Writing to the map is an append operation, where the key is kept in memory and the value written to disk with the offset recorded. This Hashmap does not support updates i.e. insertions are immutable.

The key is stored on disk as a string with the value persisted as bytes. The Store itself is agnostic of the value type and can be used to store different information. Decoding and encoding of the value to and from the byte representation are left to higher-level abstractions. Note that reading from the store is thread-safe but we assume single-threaded writes. No locking is used for the latter, so we also rely on higher-level abstractions to ensure concurrent writes do not occur.

1. **Doc values** - a file containing a mapping from the internal document id to the values for a field. This is for short string fields only e.g. authors and is used for faceting as well doc id lookups. Stored in doc id order. This uses the internal store for persistence. A cache is also implemented to accelerate doc id lookups. Exists per Segment (3.3.3). Currently, we store doc values for the authors and subject fields. This is visually shown in Fig. 2 (same principle for authors):

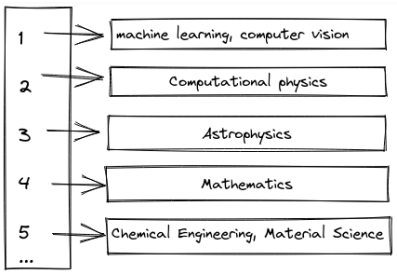

Figure 2: Every document has an internal doc id. Each doc id is mapped to the subjects of the corresponding paper.

2. **Postings** - a file containing a mapping from a term to a list of the containing documents in order of document id. In addition, the collection frequency of the term is encoded along with the frequency of the term for each document (document frequency). This uses the persistent hashmap described above, where the term represents the key. Keys/terms are inserted in the order they occur. However, prior to persistence to disk they are sorted in lexicographical order. This allows for later merging (see Section 3.3.4)to occur efficiently. A postings file exists per Segment (3.3.3). This file allows boolean queries which do not require positions to be evaluated at a lower disk access cost - i.e. all logic except proximity/phrase queries. Prior to persistence to disk, skips lists are generated for the document ids. These are persisted along with the above information and used to accelerate intersections. A postings entry for a term is shown in Fig. 3 as persisted on disk. Note that the unstemmed form of the term is stored in the posting value. This information is currently only used for building suggestions.

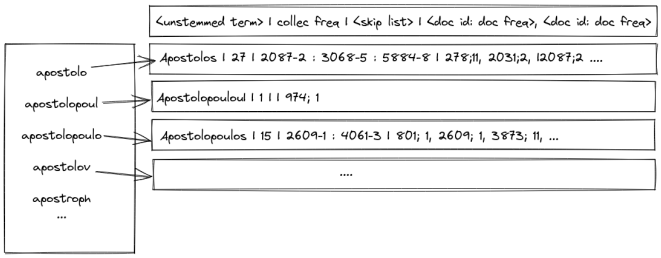

Figure 3: Posting entry for a term.

3. **Positions** - a file containing a mapping from a term to a list of the containing documents and the respective positions of the term. This is very similar to the postings file described above, except positions of the terms for each document are also encoded. A positions file is thus considerably larger than a postings file for the same term. We additionally encode skips lists for the positions of each document. Positions are their respective skip lists are used for proximity queries.

4. **Suggestion Trie** - a trie structure can be built from the postings on demand to service suggestion queries. This is held in memory only.

3.3.3 Segments

When indexing a list of documents we are left with a set of terms with postings, positions and doc values after the tokenization and stemming process. Updating an inverted index is potentially expensive, as term information will need to be read and updated. This process may be viable if the entire index can be held in memory. This, however, is not viable on larger datasets forcing us to persist postings and positions on disk - note the persistent hashmap

only keeps our terms in memory as keys. Reading, updating and rewriting information back to disk becomes prohibitively expensive. To address this issue we ensure new terms and their postings are written immutably and merged with the existing data later. This introduces the concept of segments.

A segment represents an isolated index for the documents, and contains its own postings, doc values and positions. Initially, this information is stored entirely in memory – this uses in-memory representations of the structures described above. When the number of documents exceeds a configurable limit (10,000 by default), or an explicit "flush" operation is initiated, the structures are written to disk. A new segment is created, to which new documents will be added. Indexing is single-threaded guaranteeing only one "in memory" segment can exist at any one time. The flushing of a segment involves the sorting of the terms to ensure they are lexicographical order and the use of the persistent hash map for the structures described earlier. Additionally, skip lists are generated at this stage prior to disk persistence. By limiting the buffer size, we limit the amount of memory usage. Note that the in-memory segment can be searched like any other segment.

Our index maintains a file pointer to each of the current segments. This index is extremely lightweight, rarely exceeding several MB, and is persisted on disk using pickle [18]. A restart of the server loads this file, restoring the segment pointers and thus allowing searches. Prior to any server shutdown, any in-memory segments are flushed. The current segment list is held in order of its creation. This is an important optimization as it ensures any searches executed across the segments will result in the documents being in the order of document id (postings are stored in order of document id within each segment). This permits fast intersections and unions on the results.

At query time, the query is executed across all of the segments linearly. Each of these segments has their own postings, positions and doc value files: they are in effect their own isolated index. Results are collected from each of the segments (in document id order) and combined before sorting is applied. Any facets are computed from the final list - again requiring a read from each segment for each of the fields on which a facet is requested. This process is viable for smaller segment counts. However, for larger numbers of segments, this requires many file accesses and reads e.g. at minimum with no faceting, `number of terms * number of segments`. This slows downs queries considerably. To reduce the number of segments, and in turn, the number of file reads, and accelerate queries we introduce a merge process.

This process is visualized in Fig. 4.

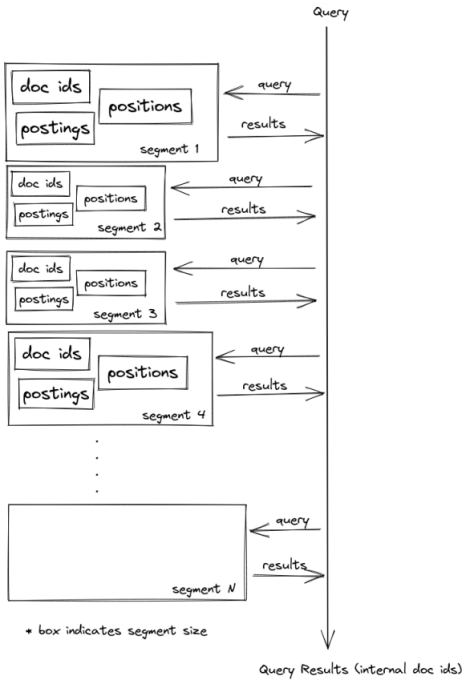

Figure 4: The index is made up of multiple segments, which can be viewed as indexes of their own. A query is run on all segments and the results merged.

3.3.4 Merging

Merging addresses the challenge of an ever-increasing number of segments and its potential to negatively impact query performance. When a merge is initiated, the two smallest adjacent segments (by document count) are merged together. The resulting merged segment replaces their reference in the index. This requires a short lightweight lock, during which queries cannot execute, as the segment list is updated in the index. The original segments are in turn updated on disk. This process is shown in Fig. 5. Merging can continually be called (via its API endpoint) until there is a single segment - the most optimal index. We call this process "optimizing".

Several earlier design choices optimize merge speed. Specifically:

1. Terms are inserted into the postings and position files in lexicographical order. This allows postings and position files to be merged with a linear merge - worst case `O(m+n)`.

2. Postings are stored in document id order. This allows postings for a term to be simply concatenated.

3. Document ids are held in order of doc id in doc value files. This allows doc value fields to be concatenated. The merged result will be a doc value file in order of document id. This process represents a simple Logarithmic merging [9] technique. Only one merge can occur at any one time.

4 Search

4.1 Query Parsing

Queries are parsed using a Parsing Expression Grammar(PEG) [10] that allows for both boolean and free-text queries. This grammar is implemented using the library `pyparsing` [11]. This avoids the need to write error-prone query parsing code whilst providing a formal definition of the grammar and allowing arbitrarily complex boolean expressions. The parsed query tree is evaluated recursively depth-first, with the base case of the recursive leaf's requiring term lookups against the index. The following search expressions are currently supported by the grammar and parser. Each expression type is a node type in the parsed grammar tree. All operators return a list of ScoredPosting, each representing a scored document (score of 0 if scoring is disabled).

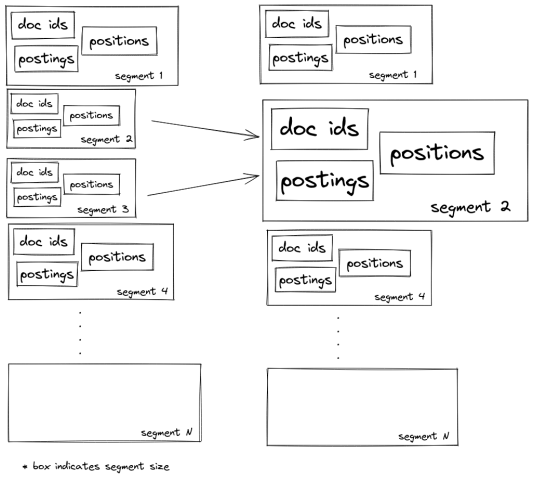

Figure 5: The two smallest adjacent segments are merged to tackle the increasing number of segments affecting query performance.

4.2 Query Evaluation

4.2.1 Natural Language

A natural language is defined as a query with no boolean, proximity or phrase operators. They must be absent from the entire query. The query must also be greater than 1 term (this is a term query). The query text is first processed by the BERT model, converting it to a vector. This is used to request 10,000 hits from HNSW. This list is subsequently filtered to docs with a cosine distance from the query, greater than a user-specified value (default 0.2). We use an `ef` (number of neighbors for each vector) of 50, finding this gave a reasonable compromise between performance and accuracy. Results are returned in order of least distance. This distance is subtracted from 1 to give a final doc score.

4.2.2 Term

This represents either a single term query or a leaf node in a more complex tree. Terms are looked up against the instance of the `Index` class via `get_term`. This returns the associated term information as an instance of `TermPostings`. This class exposes an iterator over the postings, each representing the term/doc information using the `Postings` class - including the positions. As `Postings` are iterated the associated documents are scored using TF-IDF, producing a `ScoredPosting` (effectively wrapping a `Postings` instance). A list of `ScoredPosting` is returned for use by higher-level operators, e.g. AND. Note that we allow scoring to be disabled. In this case, the score is 0. When accessing term information either postings or positions file is accessed. This depends on the higher-level operator e.g. `AND`, `Phrase` etc. **Queries that do not require positions use the postings file only to reduce the amount of data to read and decode**.

4.2.3 AND

AND operators join two nodes within the tree, e.g. A AND B. A and B can be any other expression type, e.g. a term, phrase or another AND etc. The left and right sides are recursively evaluated, resulting in two lists of `ScoredPosting`. Our indexing strategy ensures these two lists are sorted by doc id. This allows a linear merge to be performed on the two lists, resulting in an intersected list that is returned. The AND operator additionally accepts a conditional. This is a function that is used to confirm if a doc id should be added to the intersected list. By default, this function simply returns true. However, this function can be used to perform more complex conditional requirements and is used by Phrases (Section. 4.2.7) and Proximity (Section 4.2.8) Matching. To accelerate intersections we use skip lists stored with the postings.

4.2.4 OR

Similar to AND, OR operators join two nodes in the tree (e.g. A OR B) and recursively evaluate each side. The two resulting lists of `ScoredPosting` are processed using an adapted linear merge that performs a union (removing duplicates) instead of an intersection. The doc id order is preserved in the resulting list which is returned.

4.2.5 NOT

The NOT operator receives one node from the tree, e.g. NOT A. It first requests an evaluation recursively to obtain a list of scored doc ids in ascending order. It then iterates from 1 to the max doc id in the index (`max(doc_id)`), recording any doc ids that are less than the value of the current position in the list - which starts at 0. The list position is advanced once the iterator value equals the current value in the list. This process continues until all values in the list are exhausted, at which point the remaining doc ids from the value to `max(doc_id)` are recorded. NOT can thus be performed in linear time.

4.2.6 Parenthesis

Parenthesis are supported around boolean queries to explicitly state execution order e.g. `(nuclear AND physics)OR (astronomy AND fusion)`. Each query within a parenthesis will form a subtree for execution before being merged using the combining operator.

4.2.7 Phrase

Phrases are expressed using quotations, e.g. "The Cat jumped". The operator receives this as a single node in the tree. The phrase is in turn re-written as an AND query, e.g. "The Cat jumped" $\rightarrow$ "Cat AND jump" (after tokenisation/stemming). This AND query is evaluated but with an additional conditional function that must evaluate to true. During intersection, this function is invoked to confirm an item should be added to the list. The function receives the two `ScoredPosting` instances being confirmed. These represent the doc id information for the two terms. These instances also provide access to the underlying `Postings`, which contain the positions of terms in the docs. The conditional function uses these positions to confirm if the terms occur in order, i.e. the position of the right term is one after the position of the left term. If this occurs anywhere in the two lists, true is returned immediately. This is achieved using an adapted linear merge that moves through the two lists concurrently in `O(m+n)` time, relying on the fact that the positions are sorted in ascending order. A return value of true or false confirms whether the doc should be added to the AND intersection.

For phrase queries with more than two terms, the terms are evaluated in pairs from right to left. This exploits the usual AND recursive evaluation behaviour. For example, "A B C" $\rightarrow$ A AND B AND C, which is evaluated as A AND (B AND C), i.e. B AND C are evaluated first, with the phrase conditional, before the result is used for an AND with A. This relies on the positions of the left term being returned for the new `ScoredPosting` instance resulting from the intersection.

When terms are fetched for this query, the positions file is utilised. The intersection of AND utilises the skip lists. Furthermore, we use skip lists for the positions to accelerate the comparisons.

4.2.8 Proximity

Proximity matching such as *`#10(income, taxes)`* use exactly the same logic as phrase matching relying on the sorted positions, relying on passing a conditional function `_proximity_match` to the AND function (which returns true as soon as a match is found), except:

1. The distance between the terms is set to a value of N instead of 1, i.e. in the above example, this is 10. **This value is inclusive**.
2. The order of the terms does not matter, i.e. it is not directional.
3. Skip lists for positions not currently exploited.

4.2.9 Mixed

If a query has no boolean, proximity or phrase operator it is processed using vector scoring via HNSW. However, if only a component of the query is free of these operators, but the wider query is not a natural language, then an OR is used for those terms with no operator. For example, for `nuclear physics AND thermodynamics` this will be interpreted as `nuclear OR physics AND thermodynamics`.

4.3 Scoring

Scoring is enabled by default but can be disabled at query time. Disabling scoring causes documents to be returned by document id in ascending order.

Operators are scored as follows:

1. Term - Simple TF-IDF function for the matching doc id/term pair i.e. `score = (1 + math.log10(doc_posting .frequency))* math.log10(self._index.number_of_docs/term_postings.doc_frequency)` .
2. AND - Sum - a new `ScoredPosting` is created (using the left positions) with the sum of two `ScoredPosting` instances which satisfy the intersection as the score.
3. OR - the score of the `ScoredPosting`. If the doc exists in both lists, a sum is performed similarly to AND.
4. NOT - Constant score of 1 to all docs.
5. Proximity/Phrase - Currently scored the same as an AND.
6. Natural Language - scored with Cosine similarity vs HNSW.

4.4 Faceting & Filtering

Doc values are used to provide faceting functioning as shown below. These facets can also be clicked, applying a filter to the result set on the value.

Doc values provide a mapping from a document id to the value of a field. This is limited to specific fields - currently `authors` and `subjects` but easily configurable. Once the complete set of results is collated for a query, the values for the requested facet fields (see API 8) are read from the current segments. The count of each unique value is then returned. To accelerate this process, values are held in an in-memory cache per field in each segment. When a document's field value is read, this is pushed into the cache. Subsequent requests for the same document id and field from future queries will utilize this cache prior to requesting a disk read. This cache is only cleared when a segment is merged.

To allow filtering on facets, the field values must also be indexed. To ensure the value is associated with the field, these values are prefixed with the field name. For example, the field `subject` with the value `Materials Science` will be indexed as:

- `Materi`
- `Scienc`
- `subject:Materials_Science`

When a filter is specified for a query, this is appended to the query text with an `AND` clause and final term. For example, the query `graphite` with a filter for `Materials_Science` on the `subject` field causes the final query `graphite AND subject:Materials_Science to be executed`.

Filtering for natural language queries requires 2 phrase execution. The query text is first converted to a vector and executed against HNSW. The filters are in turn converted to a simple `AND` query - similar to the format described above, but with no query text. This query is in turn executed against the inverted index. The results of these two queries are finally intersected to give the filtered list. Note that faceting always occurs after filtering, thus ensuring counts are reflected of the filtered results.

4.5 Other Search functions

The following additional functionality is supported:

1. Ability to pagination through results via `offset` and `max_results` parameters in the search request body.

4.6 Document Results

Once all results have been finalized and faceting completed, the internal document ids are used to look up the JSON body for the top N documents requested from the embedded LMDB database. The fields returned for the top N hits, can be limited to a specific set of fields via a `fields` parameter in the search request body. This reduces network transfer to the UI and improves rendering time.
The top N results are returned to the UI in JSON, along with facets and the total hits.

5 Query Autocompletion

To optimise user experience, our system provides query autocompletion. Contrary to general-purpose autocompletion models, we share the same dictionary as the inverted index. This allows suggestions to be sorted by their relevance – using both absolute frequency and tf-idf. For fast and efficient run-time, the dictionary is stored as a trie, in particular we use a Matching Algorithm with Recursively Implemented StorAge (MARISA) trie. Tries make it efficient to perform prefix searches (finding words beginning with a given prefix). For multi-word queries, a position query is performed with the most relevant terms found in the trie, to further select the one which best matches the other terms in the query. For optimal performance, the trie is only build once at the end of a bulk indexing, rather than after every call to the `add_documents` API. This means that queries and suggestions will be out-of-sync for a few minutes when the index is first built. However, this is not a problem during live-data collection, as the volume of new documents added is small enough that the build is immediate.

6 BERT

Keyword-based search approaches usually struggle with several problems. Firstly, it is possible that the same word in a query may have different meaning. They also do not perform well on a relative long document such as an academic paper. As a result, we implemented semantic search using BERT to support the vector-based search and improve the retrieval performance. To enable vector search, we need to create dense vectors from queries and documents with fixed length. BERT can embed each token in an input text into a vector so that in its neural network, each layer contains tokens in the text with corresponding extracted feature vectors. To attain a document-level vector, we need pooling method that averages all token vectors in an input text. Since there are many hidden layers in the neural network that we could use for extracting feature vectors, we ran various experiments and found the second to last layer to perform best in semantic search.

The available library we used for building our BERT module is Hugging Face. It provides several pre-trained models such as bert-base, bert-large, longformer and distilbert. In the BERT module class, we also provided different model choices for final performance testing. Firstly, we tried bert-base and bert-large models. However, the total number of tokens in an academic document exceeds the maximum number of inputs for the model. Hence, what we did is to truncate the whole paper into several paragraphs that reach the upper bound of input length and average their feature vectors. However, we realized that a simple pooling operation loses globally meaningful semantic information. This led us to try the longformer model which has a maximum input size of of 4096 tokens. Nevertheless, we later found that it would take too long for the engine to respond in real time, and therefore we changed to another small model, `distilbert`, which is faster while preserving the performance. The final change is to further improve the relevance of retrieved documents. Since our HNSW is searching based on cosine similarity score, we used a fine-tuned BERT model trained on cosine similarity which is S-BERT. This model can improve our performance while maintaining its speed.

7 Hierarchical Navigable Small World

7.1 How it works

In order to achieve fast results with larger amount of documents, we can use the graph based Hierarchical Navigable Small World, HNSW [19]. This algorithm is used in conjunction with vector representations of both documents and queries to perform a fast vector space search. HNSW is modelled from other approximate nearest neighbour algorithms, whilst improving upon them.

Abstractly, the fast retrieval method works by using an indexed HNSW model which stores multiple layers that are searched through with increasing density, where the sparsity of a layer is an advantage that allows to traverse a layer quickly. Within a layer, we only find a local minimum, before using this as an entry point in the denser layer below to find better local minima. It is not always the case that the k nearest neighbours are the best selection, as we may find that clusters of vectors in the same space may be isolated from outlying vectors or other clusters. This isolation of vectors would make traversal and searches slower or impossible to achieve desirable results. The heuristic selection of neighbours is used to prevent such difficulties, by choosing diverse neighbours that are in different directions.

7.2 The library

A basic algorithm was implemented [21] but to make it efficient was out of scope of the project and would have taken a long time to build. Instead a library was used [20], which implemented the algorithm using C and Python bindings. This had to be heavily tested to find the right parameters to make it efficient to use in the context of our project.

8 API

All interactions with the index occur through a JSON based REST API using Flask [12].

- `/flush` - flushes the current in-memory segment to disk.
- `/optimize` - initiates a merge between the two smallest adjacent segments. Blocks if a merging is occurring. Reports the previous and new segment count. Can be repeatedly called until the number of segments is 1 for an optimal index. Should be executed when indexing is complete.
- `/index` - Indexes a single document via a POST. The document should be sent in the request body in JSON format. A special field vector should be present for the vector for HNSW.
- `/bulk_index` - Indexes a batch of documents via a POST. Documents should be sent in ndjson format in the body. A special field vector should be present for the vector for HNSW.
- `/suggest` - Provides suggestions based on query text.
- `/build_suggest` - Builds the suggestion trie using the current segments - see Suggestions.

Further details can be found here on deployment and request specifications.

9 Safety & Performance

9.0.1 Threading

We support concurrent querying but only single-threaded indexing. A number of read-write locks are used to achieve this. Our read-write lock allows concurrent reads but only single-threaded writes. Writes must wait for all reads to complete before executing and reads are blocked whilst a write occurs. Note, the use of these locks. They do not prevent concurrent querying and indexing - only protecting key state changes. Specifically:

- `write_lock` - Ensures all operations which modify the index state (e.g. pointers to segments) such as indexing, saving on exit and loading on start are single=threaded. Attempts to perform concurrent state-changing operations are blocked. Does not block reads.

- `merge_lock` - Ensures merging is single-threaded. No impact on queries.
- `segment_update_lock` -

Per segment we maintain:

- `indexing_lock` - Ensures indexing remains single-threaded per segment. Blocks indexing during flushing as terms in the buffer are iterated over and written to disk. Does not block queries ever.
- `flush_lock` - Used momentarily once flushing is complete to switch queries to the disk structure instead of buffers that are cleared and not re-used.

9.0.2 Optimizations

In order to optimize query performance the following optimizations proved critical:

- Use of the ujson [13] library for decoding and encoding documents. Other libraries such as SimdJSON but provided minimal gain;
- Use of skip lists to improve intersections and proximity matches;
- Doc value caches for faceting;
- Separation of postings and positions into separate files. This specifically helps non-proximity queries by reducing the volume of data for reading and decoding;
- Moving to a minimal representation on disk. JSON representations proved a bottleneck on decoding at query time. This requires further improvement;
- Ensuring terms were stored in lexicographical order and ensuring segment order was maintained. This ensures queries across segments do not need to resort documents for intersections and unions, since they are inherently in order;
- Positions and postings were originally delta encoded. This provided no performance benefit. Although data size was reduced on disk, this did not offset the cost of reversing the encoding at query time.

10 Future Improvements

More efficient/scalable/robust alternatives have not been implemented in the project due to time constraints but were considered [14].

11 Individual Contributions

11.1 Luwei Wang, s2161733

Researched BERT models and its implementation methods. Was responsible for building document embedding module, i.e. BERT Module and writing report of corresponding BERT Module part 6.

11.2 Saad Sharif, s1745977

Researched multiple vector space search algorithms, implemented the whole HNSW algorithm from scratch [21], that was not used for final system, and tested it, wrote the HNSW description in report.

11.3 Lorenzo Baldini, s1853050

Designed and implemented the UI and frontend. Adapted the scraper script for live data collection. Implemented query autocompletion.

11.4 Dale McDiarmid, s0127267

Implemented the inverted index and built an efficient way to store and retrieve documents. Implemented support for all boolean, phrase and proximity queries. Worked on the API stack and scaffolding for the UI.

11.5 Vincenzo Incutti, s1819881

Data collection via arXiv bulk access service; Comparison of multiple PDF processing libraries for text extraction; Efficient vector generation & comparative analysis of different BERT models; First implementation of live indexing scraper; Report editing.

11.6 Balraj Gill, s1743204

Wrote the script for cleaning up raw web scraped data, tested and evaluated the hnsw library to find optimal parameters and configured it to be easily added to the system. Helped with optimizing python code for final system, helped with Report Editing.

References

[1] Doug Turnbull, 2019. *What problem does BERT hope to solve for search?* https://opensourceconnections.com/blog/2019/12/18/bert-and-search-relevance-part2-dense-vs-sparse/

[2] Dharti Dhami, 2020. *Understanding BERT — Word Embeddings.* https://medium.com/@dhartidhami/understanding-bert-word-embeddings-7dc4d2ea54ca

[3] Kostas Stathoulopoulos, 2020. *How to Build a Semantic Search Engine With Transformers and Faiss.* https://towardsdatascience.com/how-to-build-a-semantic-search-engine-with-transformers-and-faiss-dcbea307a0e8

[4] Reddit, 2020. *How well does average word vectors using BERT embeddings work for document classification tasks?* https://www.reddit.com/r/LanguageTechnology/comments/jcearm/how_well_does_average_word_vectors_using_bert/

[5] Shay Palachy, 2020. *Document Embedding Techniques.* https://www.topbots.com/document-embedding-techniques/

[6] *ArXiv.* https://arxiv.org/

[7] *ndjson.* http://ndjson.org/

[8] *Stop Words list* http://members.unine.ch/jacques.savoy/clef/englishST.txt

[9] *Logarithmic Merging*, Information Retrieval: Implementing and Evaluating Search Engines, by Stefan Büttcher, Charles L. A. Clarke, Gordon V. Cormack, pg. 240 https://mitmecsept.files.wordpress.com/2018/05/stefan-bc3bcttcher-charles-l-a-clarke-gordon-v-cormack-information-retrieval-implementing-and-evaluati
pdf

[10] *Parsing Expression GrammAr (PEG)* https://en.wikipedia.org/wiki/Parsing_expression_grammar

[11] *pyparsing* https://github.com/pyparsing/pyparsing

[12] *Flask* https://flask.palletsprojects.com/en/2.0.x/

[13] *ujson* https://pypi.org/project/ujson/

[14] *Future Improvements* https://github.com/saadsharif/ttds-group/blob/main/api/indexing.md

[15] *Stemmer* https://pypi.org/project/PyStemmer/

[16] *Lightning Memory-Mapped Database* https://en.wikipedia.org/wiki/Lightning_Memory-Mapped_Database

[17] *lmbd package* https://lmdb.readthedocs.io/en/release/

[18] *pickle* https://docs.python.org/3/library/pickle.html

[19] *Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs* https://arxiv.org/pdf/1603.09320.pdf

[20] *hsnwlib* https://github.com/nmslib/hnswlib

[21] *HNSW implementation that is unused* https://github.com/saadsharif/ttds-group/blob/main/deprecated/hnsw.py