CS5604 (Information Retrieval) F2023
Team 2 Project

Final Report

Ujjwal Maheshwari      Aseem Khandelwal      Jason Banuelos      Nikhil Ram

Harsha Bhamidipati

December 14, 2023

Instructed by Professor Edward A. Fox
Guided by Satvik Chekuri

Virginia Polytechnic Institute and State University
Blacksburg, VA 24061
Abstract

Theses and dissertations represent significant bodies of work accomplished by others, often containing remarkable contributions. The advent of electronic theses and dissertations (ETDs) aimed to simplify the storage and accessibility of these documents. However, their true value is realized when accompanied by an effective system for searching and retrieving specific documents. Our project involved building an Information Retrieval System that supports searching, ranking, browsing and recommendations for a large collection of ETDs. We divided the main goal into two modules - Search and Recommendation. Search is accomplished using Elasticsearch. An overview of the tool is given in the report, along with goals and the implementation process. A recommendation module will provide relevant recommendations for a user, built by experimenting with multiple algorithms in order to obtain the best results. The user manual has been provided for the reference of other groups. The developer manual includes how the project was developed, including architecture, data flow, module overviews, etc. The final report provides an overview of the tasks undertaken, how we planned to achieve our goals, milestones and our timelines. By the project’s conclusion, we successfully scaled the system to manage 500K ETDs. Our efforts resulted in enhancements, particularly in bulk indexing and achieving faster response times for searches. Additionally, we refined the existing index schema and implemented a logging mechanism within Elasticsearch to accommodate logs from all collaborating teams.
Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>List of Tables</td>
<td>5</td>
</tr>
<tr>
<td>List of Figures</td>
<td>6</td>
</tr>
<tr>
<td>List of Listings</td>
<td>7</td>
</tr>
<tr>
<td>1 Overview</td>
<td>8</td>
</tr>
<tr>
<td>1.1 Project Effort</td>
<td>9</td>
</tr>
<tr>
<td>2 Literature Review</td>
<td>10</td>
</tr>
<tr>
<td>2.1 Search</td>
<td>10</td>
</tr>
<tr>
<td>2.2 Elasticsearch Field Data Types</td>
<td>11</td>
</tr>
<tr>
<td>2.3 Recommendations</td>
<td>12</td>
</tr>
<tr>
<td>2.3.1 Primary Recommendation System Types</td>
<td>12</td>
</tr>
<tr>
<td>2.4 Elasticsearch Scalability: Optimizations for Large-Scale Text Search and Analysis</td>
<td>14</td>
</tr>
<tr>
<td>2.4.1 Shardng and Replication</td>
<td>14</td>
</tr>
<tr>
<td>2.4.2 Index Lifecycle Management</td>
<td>14</td>
</tr>
<tr>
<td>2.4.3 Field Data Types</td>
<td>15</td>
</tr>
<tr>
<td>2.4.4 Avoid Large Text Fields if Possible</td>
<td>15</td>
</tr>
<tr>
<td>2.4.5 Nested Objects</td>
<td>15</td>
</tr>
<tr>
<td>2.4.6 Dense Vector Fields</td>
<td>15</td>
</tr>
<tr>
<td>2.4.7 Indexing Performance</td>
<td>16</td>
</tr>
<tr>
<td>3 Requirements</td>
<td>17</td>
</tr>
<tr>
<td>4 Design</td>
<td>18</td>
</tr>
<tr>
<td>4.1 Methodology</td>
<td>18</td>
</tr>
<tr>
<td>4.2 Approach</td>
<td>19</td>
</tr>
<tr>
<td>4.2.1 Search</td>
<td>20</td>
</tr>
<tr>
<td>4.2.1.1 Types of Documents</td>
<td>21</td>
</tr>
<tr>
<td>4.2.1.2 Ranking</td>
<td>21</td>
</tr>
<tr>
<td>4.2.1.3 Search Experiments</td>
<td>21</td>
</tr>
</tbody>
</table>
4.2.2 User Management ........................................... 23
4.2.3 Logging ....................................................... 23
4.2.4 Recommendations ......................................... 24
  4.2.4.1 Models ................................................... 25
4.2.5 Data Schema ................................................ 26
4.3 Indexing Data .................................................. 29
4.4 Handling Updates ............................................. 29
  4.4.1 Data Update ................................................ 29

5 Implementation .................................................... 30
  5.1 Overview ...................................................... 30
  5.2 Tasks, Outcomes and Schedule ............................... 30
    5.2.1 Search .................................................... 30
    5.2.2 Recommendation ......................................... 31

6 Future Work ...................................................... 33

7 User Manual ...................................................... 34
  7.1 Search ........................................................ 34
  7.2 Experiments .................................................. 35
    7.2.1 Create a Search Experiment ............................. 35
    7.2.2 Run Search Experiment .................................. 36
    7.2.3 Delete Search Experiment ................................ 37
    7.2.4 List User Related Search Experiments .................. 38
    7.2.5 Search Experiment Over a collection ................. 38
  7.3 Logging ....................................................... 39
    7.3.1 Saving Logs .............................................. 39
    7.3.2 Retrieve Logs ............................................ 40
  7.4 Recommendation ............................................. 40
    7.4.1 Get Recommendations for User ......................... 40
    7.4.2 Update Recommendations for User ..................... 41

8 Developer Manual ................................................ 42
8.1 Search ................................................................. 42
  8.1.1 Embeddings ...................................................... 42
  8.1.2 Indexing .......................................................... 42
    8.1.2.1 APIs .......................................................... 43
    8.1.2.2 Filtering ...................................................... 44
    8.1.2.3 Creation of Embeddings .................................. 44
    8.1.2.4 Execution of the Script .................................. 44
    8.1.2.5 Future Work ................................................ 45
  8.1.3 Security ........................................................ 46
  8.1.4 Searching ....................................................... 46
    8.1.4.1 Traditional Search with BM25 .......................... 46
    8.1.4.2 k-Nearest Neighbor Search ................................. 46
    8.1.4.3 Hybrid Search with RRF .................................. 47
  8.2 Logging ............................................................ 49
    8.2.0.1 Logging Service and Kafka Usage ....................... 49
    8.2.1 Creating New API with Logging Service Functions .......... 50
  8.3 Recommendation ................................................... 52
    8.3.1 Initialization and Pre-compute ............................ 52
    8.3.2 APIs .......................................................... 52
    8.3.3 Future work .................................................. 52

Bibliography ............................................................ 55
List of Tables

1  Schema for ETD Index ......................................................... 26
2  Schema for Chapters Index ............................................... 26
3  Schema for Logging Index ............................................... 26
4  Search Tasks ................................................................. 31
5  Recommendation Tasks ...................................................... 32
List of Figures

1. Flow of Data ................................................................. 9
2. Methodology ................................................................. 19
3. System Design ............................................................... 19
4. Search Service Architecture ............................................... 20
5. Experimenter Flow ........................................................ 22
6. Recommendation System .................................................. 25
7. Search API Request Body .................................................. 34
8. Search API Response ....................................................... 35
9. Create Experiment Request Body ....................................... 36
10. Run Experiment Request Body ......................................... 36
11. Run Experiment Response Body ....................................... 37
12. Delete Experiment Request Body ...................................... 37
13. Delete Experiment Response Body ..................................... 37
14. Delete Experiment Response Body ..................................... 38
15. Search on collection Request Body .................................... 39
16. Recommendation System ................................................ 42
17. Indexing Script Diagram ................................................ 43
18. ETD Metadata Response .................................................. 43
19. Indexing Script Options .................................................. 45
20. Elasticsearch BM25 Search Query ..................................... 46
21. Elasticsearch kNN Search Query ....................................... 47
22. Elasticsearch Hybrid Search Query .................................... 48
23. Apply RRF on BM25 and kNN Results ................................. 49
### List of Listings

1. Sharding configuration in Elasticsearch .................................................. 14
2. Index lifecycle management in Elasticsearch .............................................. 14
3. Optimal indexing performance in Elasticsearch .......................................... 16
4. Elasticsearch index schema definition for log_index .................................. 28
5. Request body to get recommendations for a user_id .................................... 40
6. Response body of get recommendations API .............................................. 40
7. Request body to get recommendations for a user_id .................................... 41
8. Log Service Structure .............................................................................. 49
9. Sample Kafka Producer code .................................................................... 50
10. Example Usage.......................................................................................... 51
1 Overview

To truly utilize electronic theses and dissertations (ETDs), it is crucial to have the ability to search for and retrieve them. It’s vital to have a system that can accomplish this and does so effectively. Our work builds upon a foundation provided by the previous work in the offerings of CS 5604. The previous team [17] successfully worked with a substantial collection of over 50,000 ETDs and developed a reliable system. Our goal is to expand this system’s capacity to handle a much larger dataset of around 500,000 ETDs.

Our project’s goal is to create a system for storing and retrieving information that can index and search the content of ETDs with the guidance of Dr. Edward Fox and our Subject Matter Expert Satvik Chekuri. Our tasks involve taking in pre-processed data provided by other teams, indexing this data using Elasticsearch, improving search performance and ranking, and finally building a recommendation system.

We worked with the other teams in the course to ingest data and provide search and recommendation as shown in Fig. 1. Team 3 ran their object detection models and put the detected object metadata and topic modelling data into the database. Team 4 summarized chapters and generated classification data which was stored in the database. While signing up, the user will choose their categories and topics of interest, which will be stored in the database. All of this data will be used to power the search and recommendation services. As per our discussions with Team 5 and Team 6, the product does not have any logging setup. Since search logs are an essential source of data for recommendations, we stored user queries in an Elasticsearch index as an alternative to logging.

The first goal is to index data using Elasticsearch so that users can do searches over the entire corpus of ETDs. By doing searches on a variety of figures and objects, in addition to the text of the thousands of ETDs, we hope to enhance the search results and the system’s usefulness. To further improve search results, we are utilizing Reciprocal Rank Fusion on Elasticsearch.

We also planned to develop a recommendation system that recommends ETDs that are closely related to the document the user is reading. Generally, when a user clicks on one of the ETDs, it is likely that they would be interested in the documents that are closely related. Item-item clustering based on ETD title and abstract will be used to determine which ETDs are closely related. If 2 users have similar interests, it is likely that they would be interested in the same set of ETDs. User-User clustering based on their ProQuest categories selected while signing up and other data (e.g., selected topics) will be used to determine which users are closely related. Finally, the generation of User-Item relations, initially planned to be implemented through the LightFM [14] model, remained unattained by the project’s conclusion.
1.1 Project Effort

From a high level, we have accomplished the following tasks as part of our project effort:

1. Able to consume ETD data (metadata and chapter) through /v2/digitalobjects/all API endpoint.
2. Able to make changes to index and learn more about it.
3. Incorporated E5-large-v2 model \[26\] to generate embeddings and use in kNN search queries.
4. Fixed bugs in the existing Flask server which is used to serve search results to the user interface.
5. Implemented RRF.
6. Created new indexing script to consume ETD data from new database containing 500K ETDs.
7. Indexed metadata of 500K ETDs present in the database. This set can be searched over by using BM25, kNN or combined search using RRF.
8. Designed and implemented recommendation system based on Item-Item clustering.
2 Literature Review

2.1 Search

Elasticsearch is an open-source search engine built on top of Apache Lucene, a full-text search-engine library [3]. Elasticsearch is written in Java and uses Lucene internally for all its indexing and searching. It aims to make full-text search easy by hiding the complexities of Lucene behind a simple, coherent, RESTful API. Elasticsearch indexes the contents of each document to make them searchable. In Elasticsearch, you index, search, sort, and filter documents, unlike rows of columnar data in relational databases. It is a fundamentally different way of dealing with the data and is one of the reasons Elasticsearch can perform a complex full-text search.

Since our work this semester is to further build upon an existing system, the Team 2 report from CS 5604 Fall 2022 [17] was our main reference. The report does a good job of explaining all of the work done along with various examples in the User Manual and the Developer Manual. The 2019 report by the ELS Team [27] was also useful. For a theoretical understanding of the information retrieval domain, the recommended textbook “Introduction to Information Retrieval” was imperative [19]. In addition, the book “Text Data Mining and Analysis” [28] provided us with insights on searching and indexing.

To gain more understanding about Elasticsearch, we referred to the official documentation available on their website [3]. In the practical implementation phase, Hugging Face played a crucial role in helping us deploy the E5-large-v2 [26] model into our index and search setup. This allowed us to explore various approaches to retrieving information using Elasticsearch, focusing specifically on text, lexical, and semantic search. We reviewed the classic lexical search with BM25, semantic search with kNN, and hybrid search using Reciprocal Rank Fusion (RRF).

The ranking of documents for relevance in Elasticsearch, using the Lucene implementation of the BM25 model, follows a classic approach to text search. This model, primarily employed for lexical search, operates on the basis of exact term matches. To facilitate this, Elasticsearch employs text analysis, executed by an analyzer [5] – a set of rules governing the extraction of relevant tokens. A crucial component of the analyzer is the tokenizer, which breaks a stream of characters into individual tokens (typically words). Ultimately, BM25 assesses document relevance based on term frequency and importance.

While lexical search is fast and straightforward, its limitations become evident with vocabulary mismatch – a common issue where different people may name the same thing differently [12]. This motivates the exploration of scoring models that incorporate semantic knowledge. Transformer-based models, adept at processing natural language, offer a dense, context-aware vector representation of text, empowering Semantic Search. After converting data into meaningful vector values, the k-nearest neighbor (kNN) search algorithm is utilized to find vector representations similar to a query vector [6].

Dense vector search brings advantages such as enabling semantic search, scalability for large datasets, and flexibility with various data types. However, challenges include selecting the right embedding model, fine-tuning for domain-specific datasets, and the computational expense of indexing high-dimensional vectors.
When it comes to search, there is no universal solution. Each of these retrieval methods has its strengths but also its challenges. Depending on the use case, the best option may change. Often the best results across retrieval methods can be complementary. Hence, to improve relevance, we looked at combining the strengths of each method.

There are multiple ways to implement a hybrid search, including linear combination, giving a weight to each score, and RRF, where specifying a weight is not necessary. Ranking search results by combining multiple ranking methods like BM25 and kNN can lead to better results. However, merging these result sets effectively, especially without precise knowledge of score distributions, is challenging. The RRF algorithm offers an efficient alternative supported by academic research [8]. RRF excels in seamlessly merging result sets based on ETD position and existence, reducing dependence on ranking scores. This approach enhances search result quality, making it a valuable tool in information retrieval [4].

The RRF score is derived as:

$$RRFscore(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

- $D$ - set of ETDs
- $R$ - set of rankings as permutation on $1..|D|
- $k$ - determines the weight of lower-ranked ETDs, typically set to 60 by default.

Documents that consistently rank higher across multiple rankings will have higher RRF scores, indicating a higher degree of relevance or importance within the set "D". In other words, if both methods agree on which documents are relevant, those documents are more likely to be truly relevant. Conversely, if they disagree on a document’s relevance, it may be less relevant. This technique helps in combining rankings from various sources or criteria into a single unified score for each document.

2.2 Elasticsearch Field Data Types

Each field in an Elasticsearch index has a data type associated to it. Below are some of the fields that are commonly used in our indexes.

1. Nested – The nested type is a specialised version of the object data type. In this way, arrays of objects can be indexed so that they can be queried independently.

2. Keyword – The keyword field type is used for structured content. They are used to index and search for text values and are also used to store metadata about the documents they are associated with. The keyword datatype can come in handy for cases where a user will be querying for exact matches.

3. Text – The text field type is used for a full-text search.
2.3 Recommendations

Aberdeen et al. [1] introduce a fairly straightforward but powerful idea to implement personalized recommendations accurately and at scale and to incorporate user feedback continuously. It uses the real-world use case of GMail and its implementation of Priority Box to highlight the model. We see that in a real-world use case, user behaviour varies vastly and it is important to provide recommendations catering to the user, and to avoid generalizations. Generalizations are the basis of all models, no matter how big or small. To counter this, the 2022 team planned to build a smaller model (here, logistic regression) which is trained only for the particular user and on their interactions. The results of this simple model and the global model are combined through a learned weight. These results have shown to be superior to the individual models.

Amazon.com, an industry leader in recommendations and personalization, uses item-to-item collaborative filtering as a primary component of their recommendation system [20]. Instead of searching for related users, the approach proposed compiling a list of similar items, and recommending items from the list. The algorithm was more scalable than traditional collaborative filtering. In later years, they experimented with adding in a component of time to ensure more accurate recommendations, and suggest the same. New items tend to have a cold start problem as there is not much data, and require an explore/exploit process to solve it.

In one of their older papers, Amazon further describes item-to-item collaborative filtering [15]. The algorithm matches purchased and rated items of the user to similar items, then combines those similar items into a list of recommendations. Similarity is computed using the cosine similarity measure. Given a similar-items table, the algorithm then performs similarity matching with the purchased/rated item, and recommends the most similar item from the table.

Netflix, a relatively new player in the online content delivery domain, and thus in the recommendation domain, has proven to be renowned for its recommendation system. To ensure recently watched content holds more significance and thus more weight during recommendation, Netflix’s recommendation engine considers the order of item association using sequential models which were initially developed for Natural Language Processing tasks [21]. Heterogeneous features like time of day or device being used by the user are certain features that also add great statistical significance.

Naumov et al. [16] propose a deep learning model for recommendations and personalization. The model utilizes Matrix Factorization (MF) and Factorization Machines (FM) to produce a hybrid filtering method. It takes in sparse and dense embeddings as input and produces a Click-Through-Rate (CTR) for an item. The CTR is a probability of the user clicking on the item. Items can be ranked based on the CTR to show recommendations to the user.

LightFM [14] is a Python implementation of a number of popular recommendation algorithms for both implicit and explicit feedback. LightFM makes it possible to incorporate both item and user metadata into the traditional matrix factorization algorithms. It represents each user and item as the sum of the latent representations of their features, thus allowing recommendations to generalise to new items (via item features) and to new users (via user features).

2.3.1 Primary Recommendation System Types

1. Content-based filtering
Content-based filtering uses item features to recommend other items similar to what the user likes, based on their previous actions or explicit feedback [20].

(a) Search-based methods

Search or content-based methods treat the recommendation problem as a search for related items. Given the user’s clicked items, the algorithm constructs a search query to find other popular items by the same author or with similar keywords or subjects [20].

(b) Cluster-based methods

To find ETDs that are similar to the clicked item, cluster models divide the database into many segments and treat the task as a cluster search problem. The algorithm’s goal is to assign the item to the segment containing the most similar ETDs. It then uses the clicks in the segment to generate recommendations. $k$-means clustering, for example, assigns database items to $k$ clusters by iteratively assigning each item to a cluster and minimizing the sum of the distances between the items and the centroid of each cluster [19].

The segments typically are created using a clustering or other unsupervised learning algorithm, although some applications use manually determined segments. Using a similarity metric, a clustering algorithm groups the most similar users together to form groups. Once the algorithm generates the cluster, it computes the item’s similarity to vectors that summarize each group, then chooses the group with the strongest similarity, and assigns the item to that group, accordingly. Cosine similarity is an example of a similarity measure, which calculates the cosine of the angle between the two vectors as a measure of their similarity [19].

2. Collaborative filtering

(a) Bag-of-Items

All the items are treated as items in a bag. The order of picking the item is not considered while modeling. Hence, more recent browsing history will have the same impact as earlier history.

(b) Traditional

User-based method where a user$_1$ is matched based on similarity to user$_2$ and the items purchased by user$_2$ are then recommended to user$_1$.

(c) Item-based filtering [15]

Rather than matching the user to similar customers, item-to-item collaborative filtering matches each of the user’s purchased and rated items to similar items, then combines those similar items into a recommendation list.

Given a similar-items table, the algorithm finds items similar to each of the user’s purchases and ratings, aggregates those items, and then recommends the most popular or correlated items. This computation is very quick, depending only on the number of items the user purchased or rated.

(d) Sequential – Transformers

Similar to predicting the next word in a sentence problem, transformers are used here to predict the next item in the bag. Here time and order are important and are maintained.
2.4 Elasticsearch Scalability: Optimizations for Large-Scale Text Search and Analysis

To enhance our Elasticsearch schema for scalability, we are considering multiple factors. The optimization of Elasticsearch performance involves a mix of schema design, index settings, and operational configurations [11]. Below are the tools and techniques that could be used to achieve scalability.

2.4.1 Sharding and Replication

Sharding in Elasticsearch [13] is a technique for dividing large indexes into smaller, manageable chunks so that they can be processed distributedly across multiple nodes. This enables Elasticsearch to handle a large volume of data and higher query loads, allowing for parallel processing of queries and aggregation across multiple shards. Shards can also be replicated across nodes, providing high availability, data redundancy, and fault tolerance. Listing 1 demonstrates a sample sharding implementation on an index.

```
PUT /document_index
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 2
  },
  "mappings": {
    // ... mappings
  }
}
```

Listing 1: Sharding configuration in Elasticsearch

2.4.2 Index Lifecycle Management

Elasticsearch 8.0 introduces a high-level index configuration feature for controlling index lifecycles [10]. It allows the creation of a new index when the existing index surpasses a certain threshold in terms of size or the number of documents. Additionally, it offers the capability to create indexes based on fixed time intervals, following a predefined time frame. This feature also empowers Elasticsearch to automatically delete stale indexes, ensuring compliance with data retention standards. Listing 2 demonstrates the implementation of sample index lifecycle management.

```
PUT _ilm/policy/etd_policy
{
  "policy": {
    "phases": {
      "hot": {
        "actions": {
          "rollover": {
            "max_size": "50GB",
            "max_age": "30d"
          }
        }
      }
    }
  }
}
```

Listing 2: Index lifecycle management configuration in Elasticsearch
2.4.3 Field Data Types

Selecting appropriate field data types in Elasticsearch is crucial for optimizing storage and enhancing performance [9]. Field data types influence storage efficiency by determining the amount of disk space required, and they significantly impact query and indexing performance. For instance, employing the ‘keyword’ type for fields not requiring full-text search can expedite query processing, while ‘text’ types are suitable for enabling efficient full-text search capabilities. Furthermore, the choice of field data types affects filtering, aggregation, sorting, and overall relevance to specific use cases, simplifying index mapping management and ensuring data is correctly interpreted and indexed. Consequently, making informed decisions about field data types is vital to achieve storage efficiency and maximize Elasticsearch’s query and indexing capabilities.

2.4.4 Avoid Large Text Fields if Possible

Large text fields can be expensive to index and search. If possible, large texts could be summarized or truncated, or could be stored as a representative subset of the text using text summarization techniques.

2.4.5 Nested Objects

Nested objects can cause the index to grow significantly. Unnecessary nested objects should be removed or measures should be taken to flatten them.

2.4.6 Dense Vector Fields

While dense vector fields are powerful for similarity searches, they can be quite large and may impact performance. We can test and evaluate the dimensionality (i.e., dims) and ensure it’s set to an optimal level for our use case.
2.4.7 Indexing Performance

Adjusting the refresh_interval and number_of_replicas settings during bulk indexing can improve indexing performance [11]. We need to be cautious of hitting the keyword field limit (defaults to 1000), especially if our index has a large number of fields. You might need to adjust the index.mapping.total_fields.limit setting. Listing 3 demonstrates sample optimal indexing performance configuration practice.

```
PUT /etd/_settings
{
   "settings": {
      "refresh_interval": "-1",
      "number_of_replicas": 0
   }
   "mappings": {
      // ... mappings
   }
}
```

Listing 3: Optimal indexing performance in Elasticsearch
3 Requirements

The following is the list of requirements for Team 2:

- Index ETDs and digital objects related to the ETDs. Support search over the indexed objects. Leverage prior work and improve search efficiency by refining and ranking the results.
- Redefine existing index schema to efficiently index ETD metadata, as well as chapter and object data.
- Create an endpoint to enable curators to index collections of ETDs. Subsequently, use Kafka queues to index data asynchronously.
- Improve search capability by the use of Elasticsearch 8.0 search modules.
- Test with 500k ETD data to check scalability and performance.
- Generate personalized recommendations based on user search data, clustering based on user profile data, and document-document clustering.
- Build log index to store user activity logs.
- Leverage log data to provide recommendations.
- Back-end support through APIs of the interfaces for search and recommendation.
- Support search and recommendation experiments.
4 Design

4.1 Methodology

Figure 2 shows different user personas, the various tasks we need to accomplish for each persona and the related services.
Figure 2: Workflow diagram showing users, goals, tasks, and services.

4.2 Approach

The high level design for our team requirements is shown in Figure 3.

Figure 3: High-level design for the interaction between services and other databases.
Team 2 is responsible for overseeing three containerized services: Elasticsearch, the search service, and the recommendation service. These services play a crucial role in collaborating with Team 5’s APIs to bolster our ETD search system. Our workflow begins when we receive user API requests via the API gateway following user authentication. To provide comprehensive functionality, we leverage APIs created by Team 5 to access user information, ETD documents, and chapters that we subsequently index for seamless search operations.

4.2.1 Search

The search service is responsible for indexing and searching over all the ETD metadata, figures, chapters, and other digital objects. The service runs as a container and interacts with Front-end, Elasticsearch, and Database containers as shown in Figure 4. The service queries the database using the APIs provided by Team 5 and indexes the ETDs and other digital objects into Elasticsearch. We use an explicit mapping which is discussed in Section 2.2 to map the data into the index. An API endpoint, developed by [17], is used to receive queries from the front-end. We then create a search query and use an Elasticsearch client to search over the Elasticsearch index. Conventional methods like inverted index and ML based models like kNN are used to power the search. We will use different metadata fields like author, university, major, etc. to sort and filter the search results. By default we will sort the documents based on relevance which is measured by the estimated relevance score provided by Elasticsearch [2].

Figure 4: Search service architecture and its interactions. Taken from [17].
4.2.1.1 Types of Documents

We index the following items into Elasticsearch.

1. ETD metadata which has text fields like author, abstract, title, etc. (See the complete list of fields in Section 4.2.5.)

2. Chapter summaries, classifications, and other chapter-related objects received as part of Team 4’s summarization and classification pipeline (used for searching over chapters) and topics and objects metadata from Team 3’s object detection pipeline.

4.2.1.2 Ranking

In any search system, there is a degree of uncertainty in ranking documents accurately, as different queries and user preferences may influence the effectiveness of the initial rankings. This uncertainty can result in potentially relevant documents being buried beneath less relevant ones. By revisiting the ranking of search results, we aim to improve the precision and relevance of the documents presented to the user. Re-ranking becomes especially critical when dealing with large and diverse datasets, where the initial ranking might not always capture the nuances of user intent.

We’ll use Reciprocal Rank Fusion [7] to boost the ranking of search results from Elasticsearch. Instead of relying solely on Elasticsearch’s default ranking, we’ll combine rankings from different methods to get more accurate and contextually relevant results. Reciprocal Rank Fusion involves calculating reciprocal ranks across various rankings. We’ll experiment with different methods and re-rank documents based on their consistent performance across these methods. This way, we’ll create a refined ranking list that increases the chances of presenting the most relevant information.

4.2.1.3 Search Experiments

In addition to the default functionalities provided for users to search the ETDs repository, we have implemented an existing interface that empowers researchers to design and carry out search-related experiments. These experiments will allow the experimenter to index custom vectors for each ETD. Following the generation of such data, experimenters possess the capability to perform a hybrid search, which combines the k-nearest neighbors (kNN) algorithm with keyword-based searching, across the cataloged documents. Subsequently, experimenters can assess the relevance scores assigned to the retrieved articles. The process for planning and executing an experiment is depicted in Figure 5. The experiments conducted within this research are customized for individual users, and the relevant metadata for these trials is maintained in an Elasticsearch index.

In addition to the initial four experiments, we have introduced an additional experiment, denoted as "e5-large-v2". These experiments are accessible to all users having the experimenter role. The default experiments encompass the utilization of text embeddings derived from the abstract and title of a set of 1000 ETDs. These five distinct text embedding models are employed for generating embeddings in the context of the default experiments:
1. all-distilroberta-v1 [22]
2. all-mpnet-base-v2 [25]
3. all-MiniLM-L12-v2 [24]
4. LaBSE [23]
5. e5-large-v2 [26]

Figure 5: Flow diagram depicting experimenter actions and corresponding interactions with Elasticsearch.
In prior research, embedding vectors were supplied by the researcher in the form of a CSV file to facilitate search-related experiments. This process is proposed to be streamlined by automatically generating an ETD collection based on a specified search term, on which the experimental analysis can be conducted. The experimenter will specify a search term and the desired size of the ETD collection for experimentation. Subsequently, a collection will be procured from an Elasticsearch database, employing a hybrid search methodology that integrates traditional search mechanisms with kNN. Following this retrieval, the experimenter can then perform experiments with the above mentioned collection. Additionally, we will be providing different methods for curators to index data including Kafka topics and APIs.

4.2.2 User Management

Team 5 is tasked with managing an API gateway, serving as the primary point of contact for all requests originating from the user interface. Notably, this gateway assumes dual roles: authentication server and reverse proxy.

In its capacity as an authentication server, the API Gateway interfaces with the PostGres database’s Users table to validate user-provided credentials, specifically usernames and passwords. Upon successful authentication, the API Gateway generates a JWT token, incorporating the user’s user_id. This token is subsequently injected into the HTTP request header before forwarding the request to Team 2’s search container.

Once the search service receives the request, it undertakes the critical task of validating and parsing the JWT token present in the HTTP request header to extract the username. If necessary, the search service may also make an external API call to retrieve information about the user’s preferred topics and categories of interest.

The obtained username is then used to log the user’s activities within the logs index. Furthermore, the recommendation service leverages the user’s specified topics and categories of interest to offer personalized suggestions and recommendations.

4.2.3 Logging

We implemented a generic Logging system, designed to record various user interactions as well as individual team logs. These interactions include:

1. Document search activities, noting when users perform a search with a query.
2. Accessing ETDs to view their contents.
3. Chapter search operations, recording when users search for specific chapters.
4. Clicking on particular chapters as part of a chapter search.
5. The creation of experiment indexes by users.
6. Deletion of experiment indexes by users.
7. Searches made using experiment indexes created by the users themselves.
8. Auto save logs coming from the Kafka log channel.

The logging service is implemented with methods to save different type of logs and access them. Any API call made to Team-2-search will internally call an appropriate log method to either perform a save or retrieve log operation.

In addition, the Team-2-search container service will provide a dedicated Kafka channel called "log" to enable other teams and clients to directly send a log dump which will later be processed and saved in the Elasticsearch database. Alternatively it also hosts an API endpoint (/logs API) for the same job. This API can be invoked by the recommendation service to retrieve historical data on users' past queries, ETDs they have accessed, and their areas of interest in topics and categories.

Other teams can use the generic log save API (/application/log API) to save any service-level logs for their own debugging purposes, fostering a streamlined approach to troubleshooting and analysis. By leveraging this API, teams gain the flexibility to capture and retain specific log data, enabling efficient identification and resolution of issues unique to their respective services. To differentiate between logs of different teams and services, a unique log type can be chosen.

The recent upgrade to our logging system introduces a powerful feature: customizable meta field. Logs can now seamlessly accommodate an array of additional information tailored to specific needs. This enhancement not only provides adaptability but also significantly amplifies search capabilities. The meta fields’ customization empowers users to capture diverse data points, while the increased searchability allows for more efficient querying and troubleshooting. This upgrade reflects our commitment to delivering a logging system that is not only robust but also adaptable to the unique requirements of our users.

Please note that our database contains permanent storage for user activity and interests through tables such as User_classes, User_queries_clicks, and User_topics. We assume that Team 5’s API gateway will be responsible for writing data to these tables.

These logs, systematically collected through the API and Kafka topic, will serve as training data for the recommendation system’s future enhancements and refinements. The rich and diverse data captured from various service-level logs will provide essential insights into user behavior, preferences, and system interactions. Leveraging this aggregated information, the recommendation system can optimize its algorithms, fine-tune personalized suggestions, and improve overall user experiences.

4.2.4 Recommendations

The recommendation module will recommend similar ETDs or ETDs of potential interest to the user. The recommendation system will cluster users and ETDs for User-User and Item-Item recommendations and will use user click history for User-Item recommendations. Click history, here, refers to the ETDs that the user has clicked on. The system will recommend documents similar to ones the user has clicked on and may use collaborative filtering to recommend documents that other, similar users have clicked on.

Since user data wouldn’t be available till the end stages of the semester, we came up with a new design of a recommendation system which uses only Item-Item relations to provide recommendations. The system is shown in Figure 6.
The system has 2 stages:

1. Initialization and Pre-computation: During this stage, we employ K-Means clustering to categorize ETDs into a predefined number of clusters. As K-Means operates solely on numerical data, we utilize the vector representations of ETD metadata stored in the kNN Elasticsearch index as input for the K-Means algorithm. Upon computing the clusters, we calculate the cosine similarity within each cluster and store the results.

2. Recommendations: Initially, every user receives ten random recommendations. When a user views an ETD, it indicates their interest in that particular ETD and suggests a potential interest in ETDs similar to the one currently viewed. The recommendation system takes both the user ID and ETD ID as input. Subsequently, the system retrieves the five nearest neighbors to the viewed ETD from pre-computed scores. The existing recommendations in the list are then shifted five positions down, making room for the top five nearest neighbors, which now become the user’s primary recommendations. This process repeats as the user continues to explore and read different ETDs. At any given time, the UI can query the recommendation server for the current list of recommendations for a particular user.

4.2.4.1 Models

We plan to use the hybrid latent representation recommendation model LightFM [14] for User-Item recommendations. For Item-Item recommendations, we will cluster based on the Title/Abstract of the ETDs, and for User-User recommendations, we will cluster users based on the "Interested Topics" and/or ProQuest categories filled by users in their profiles.
### 4.2.5 Data Schema

Tables 1, 2 and 3 show the mapping of database fields to their respective Elasticsearch data types. Since there was a change in the database schema since [17], the database fields and Elasticsearch types have changed.

#### Table 1: Schema for ETD Index

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Elasticsearch Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id (primary_key)</td>
<td>int</td>
<td>keyword</td>
</tr>
<tr>
<td>etds_id (primary_key)</td>
<td>int</td>
<td>keyword</td>
</tr>
<tr>
<td>title</td>
<td>varchar(255)</td>
<td>search as you type</td>
</tr>
<tr>
<td>author</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>contributors</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>abstract_additions</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>degree</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>degree_level</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>abstract</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>institution</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>year</td>
<td>int</td>
<td>integer</td>
</tr>
<tr>
<td>discipline</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>rights</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>keywords</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
</tbody>
</table>

#### Table 2: Schema for Chapters Index

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Column Type</th>
<th>Elasticsearch Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id (primary_key)</td>
<td>int</td>
<td>keyword</td>
</tr>
<tr>
<td>title</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>topics</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>categories</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>summary</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>etd_id</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
<tr>
<td>etd_title</td>
<td>varchar(255)</td>
<td>text</td>
</tr>
</tbody>
</table>

Tables 1 and 2 present the various attributes of an ETD or chapter, their corresponding data type in a SQL database, and their data type in Elasticsearch. Since most of the fields are text based and should be available for a free text search, their data type is text.

#### Table 3: Schema for Logging Index

<table>
<thead>
<tr>
<th>Column Name</th>
<th>Elasticsearch Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>id (primary_key)</td>
<td>keyword</td>
</tr>
<tr>
<td>type</td>
<td>text</td>
</tr>
</tbody>
</table>
Table 3 outlines the implementation of a new generic logging index designed to capture and organize various types of log data within our system. The primary goal of this initiative is to enhance our logging capabilities and improve our ability to monitor and analyze system activities. This log will be saved on the Elasticsearch container’s storage.

Following are the explanation of fields used in constructing the Elasticsearch index schema for log_index as shown in Listing 4.

- **ID Field (Primary Key):** We have implemented a primary key field labeled "id" with a data type "keyword." This field serves as a unique identifier for each log entry.
- **Type Field:** The "type" field, set as "text" is introduced to categorize log entries between user and system logs.
- **User ID Field:** For user identification, we have included the "user_id" field with a data type "keyword". This field captures the user responsible for the logged activity.
- **User Activity Field:** The "user_activity" field, set as "keyword", records the specific user actions, including delete, update, add, search and user login.
- **Timestamp Field:** To track the timing of each log entry, we utilize the "timestamp" field with a data type of "date". This field stores the date and time when the activity occurred.
- **Query Type Field:** We have introduced the "query_type" field as "text" to specify the type of query or operation associated with the log entry.
- **API Endpoint Field:** The "api_endpoint" field, set as "text", records the API endpoint or resource involved in the logged activity.
- **ETD ID Field:** For tracking modifications related to ETD data, we introduce the "etd_id" field as "keyword".
- **CHAPTER ID Field:** For tracking modifications related to Chapter data, we introduce the "chapter_id" field as "keyword".
- **Meta Object Field:** To accommodate structured metadata, we include the "meta" field with a data type of "object". This field allows the storage of additional contextual information related to each log entry.
Drawing upon the insights gained from our comprehensive literature review on Elasticsearch scalability practices, we have proactively applied these methodologies to enhance the performance and efficiency of our logging index.

```plaintext
PUT /log_index
{
  "settings": {
    "number_of_shards": 5, // number of primary shards as needed
    "number_of_replicas": 1, // number of replicas for high availability
    "index.lifecycle.name": "log_lifecycle_policy", // index lifecycle policy
    "index.lifecyclerollover_alias": "log_alias" // rollover alias
  },
  "mappings": {
    "_source": {
      "enabled": true
    },
    "properties": {
      "id": {
        "type": "keyword"
      },
      "type": {
        "type": "text"
      },
      "user_id": {
        "type": "keyword"
      },
      "user_activity": {
        "type": "keyword"
      },
      "timestamp": {
        "type": "date"
      },
      "query_type": {
        "type": "text"
      },
      "api_endpoint": {
        "type": "text"
      },
      "etd_id": {
        "type": "keyword"
      },
      "etd_title": {
        "type": "text"
      },
      "meta": {
        "type": "object"
      }
    }
  }
}
```

Listing 4: Elasticsearch index schema definition for log_index.
4.3 Indexing Data

For data to be searchable, it needs to be extracted from the database and indexed into the correct Elasticsearch index.

In the current setup, the indexing is done manually by using a Python script created by [17].

We plan to streamline indexing of data in 2 stages. In the first stage, we will develop an API which can be used to index data. This should only be accessible to curators. The curators should be able to reference a set of collection IDs from the Collections table and specify the index into which the data needs to be indexed. Additionally, we could provide an input field that accepts a list of ETD IDs corresponding to the entries in the ETD_metadata table that require indexing. By leveraging the information stored in the Collections table, we can establish a more robust and database-driven approach for handling the identification of collections in the indexing process.

In the second stage, we would like to move to asynchronous indexing of data through a messaging queue such as Kafka. Using Kafka has 2 main advantages. First, in case the search service is down, the data in the queue can be indexed later. Second, the system can automatically index data as an ETD is added to the system. The curator won’t need to manually trigger the indexing process.

4.4 Handling Updates

The present system will expect two types of updates: data and schema updates of the Elasticsearch indexes.

4.4.1 Data Update

**Description:** In the current implementation, there is no defined way to update data that has already been indexed.

**Proposals:**

- Provide a DELETE endpoint (only accessible to curator) in the search service to remove all data related to an ETD from all indexes in Elasticsearch. Subsequently, the curator may use the API mentioned in Section 4.5 to index the updated data as a new entry into Elasticsearch.

- For all indexing operations, check if the ETD already exists in Elasticsearch. If it does, remove the existing ETD data and index the new data. This method might cause accidental overwrites of data.
5 Implementation

5.1 Overview

The goal of the project is to implement a system which can facilitate searching over a huge collection of ETDs and digital objects. We also leverage the user data to provide recommendations. Our approach involves building containerized microservices for each task within the following services:

- A search service which will index, search for documents, and log user interactions.
- A recommendation service which consumes user logs and ETD data to train a model which can be used to provide recommendations.

A detailed description of the individual tasks for each service is shown in the tables below.

5.2 Tasks, Outcomes and Schedule

5.2.1 Search

We use Elasticsearch (v8.9.1) to index and search over the ETD metadata and chapters.

We were using the Python Flask server created in [17] to push ETD data into Elasticsearch. The search service uses the Python requests (v2.28.1) [18] library to query /v2/digitalobjects/all GET etd API to receive ETD metadata and the chapter objects related to the ETD. We used the embedding language model E5-large-v2 to create dense vectors for abstract and title text, and index them into Elasticsearch.

During the semester, a new database schema was created and implemented. This database was subsequently populated with metadata for 500K ETDs. Team 5 developed new APIs to facilitate both read and write operations with the new database. We utilized the team2/etd-metadata and team2/etd-metadata/{etd-id} GET APIs to retrieve ETD metadata. Additionally, we filtered, processed, generated dense vectors, and indexed the data into pertinent Elasticsearch indexes. As per the suggestion provided by Dr. Fox, we included keywords into the dense vector creation.

Upon scaling the Elasticsearch index to accommodate 500K ETDs, the observed performance impacts varied across different search methods:

1. **Traditional Search with BM25**: Minimal performance impact was noted, with the response time increasing from an average of 300ms to 3-4 seconds.

2. **kNN Search**: The performance of kNN search was notably impacted. Initially, searching with kNN resulted in timeouts. To address this, we increased the timeout from 30 seconds to 150 seconds, allowing us to retrieve search results, albeit with significant delays. To further enhance performance, we adjusted the parameter $k$ from 100 to 10, resulting in a reduced response time of approximately 16 seconds.
3. **Hybrid Search with RRF:** RRF was implemented, utilizing both BM25 and kNN under the "combined" search method. To assess its efficacy, we analyzed the RRF result set by manually comparing it with BM25 and kNN result sets. Presently, the response time for the combined search method is heavily influenced by the performance of the kNN search method. Consequently, we observed a response time similar to that of kNN.

Table 4 shows the various individual tasks along with their dates of completion and status.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Due Date</th>
<th>Assignee</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review previous year team’s work</td>
<td>Sep 5</td>
<td>All</td>
<td>Completed</td>
</tr>
<tr>
<td>Elasticsearch setup &amp; config</td>
<td>Sep 5</td>
<td>All</td>
<td>Completed</td>
</tr>
<tr>
<td>Indexing ETD data</td>
<td>Sep 15</td>
<td>Nikhil, Ujjwal, Mahesh</td>
<td>Completed</td>
</tr>
<tr>
<td>Get last year’s APIs up and running</td>
<td>Sep 5</td>
<td>All</td>
<td>Completed</td>
</tr>
<tr>
<td>Finalize the schema for Elasticsearch</td>
<td>Sep 20</td>
<td>Mahesh</td>
<td>Completed</td>
</tr>
<tr>
<td>Index and support search over chapters</td>
<td>Sep 15</td>
<td>Ujjwal, Nikhil</td>
<td>Completed</td>
</tr>
<tr>
<td>Keyword match based searching</td>
<td>Oct 6</td>
<td>Nikhil, Ujjwal</td>
<td>Completed</td>
</tr>
<tr>
<td>Implement kNN based search</td>
<td>Oct 27</td>
<td>Ujjwal, Mahesh</td>
<td>Completed</td>
</tr>
<tr>
<td>Implement RRF and compare with different ranking methods</td>
<td>Oct 27</td>
<td>Aseem</td>
<td>Completed</td>
</tr>
<tr>
<td>Build indexing endpoint for curators</td>
<td>Nov 7</td>
<td>Ujjwal</td>
<td>Future Work</td>
</tr>
<tr>
<td>Implement new logging schema and endpoints</td>
<td>Nov 11</td>
<td>Nikhil, Mahesh</td>
<td>Completed</td>
</tr>
<tr>
<td>Evaluate RRF efficiency with Hit Rate and MRR</td>
<td>Nov 15</td>
<td>Aseem</td>
<td>Future Work</td>
</tr>
<tr>
<td>Implement Kafka Producer and consumer services</td>
<td>Nov 15</td>
<td>Nikhil</td>
<td>Completed</td>
</tr>
<tr>
<td>Implement hybrid search with RRF</td>
<td>Nov 15</td>
<td>Aseem</td>
<td>Completed</td>
</tr>
<tr>
<td>Customize search module for experimentation</td>
<td>Nov 15</td>
<td>Harsha</td>
<td>Future Work</td>
</tr>
<tr>
<td>Develop test suites</td>
<td>Nov 25</td>
<td>Aseem</td>
<td>Future Work</td>
</tr>
<tr>
<td>Customize recommendation module for experiment with variables</td>
<td>Nov 15</td>
<td>Harsha</td>
<td>Future Work</td>
</tr>
<tr>
<td>Scaling Elasticsearch</td>
<td>Nov 25</td>
<td>All</td>
<td>Completed</td>
</tr>
</tbody>
</table>

5.2.2 **Recommendation**

Originally, we were planning on improving the recommendation system built in [17]. The previous team’s recommendation system was a single User-Item autoencoder model that provided recommendations based on user clicks. However, we decided to completely replace the system with a new ensemble system that uses an Item-Item and User-User model as well as a different model for User-Item recommendations.

For Item-Item clustering, we will be leveraging the kNN search feature present in Elasticsearch.
This will help us avoid writing the business logic to keep track and store ETD neighbors data. ETDs are clustered using K-means clustering on the embeddings from the Elasticsearch kNN index. The nearest neighbors of each ETD are pre-computed within each cluster using cosine similarity, and when a user clicks on an ETD in the user interface, the 5 nearest neighbors of the ETD are returned to update a recommendation list on the front-end. Each user starts with 10 random ETDs in their recommendation list, and each time they click on an ETD, the 5 nearest neighbors of that ETD are pushed to the top of the list, pushing the existing recommendations down.

For User-User clustering, we intended to have the front-end show new users a list of ProQuest categories and ask users to select ones they are interested in. Users would be clustered based on their selected categories. For User-Item clustering, we intended to train a LightFM [14] model on user clicks stored in Elasticsearch.

Table 5 shows the timeline of tasks.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>Due Date</th>
<th>Assignee</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>Review past work; plan improvements to previous team’s system</td>
<td>Oct 6</td>
<td>Jason, Ujjwal</td>
<td>Completed</td>
</tr>
<tr>
<td>Scrap previous system, plan new ensemble system</td>
<td>Oct 17</td>
<td>Jason, Ujjwal</td>
<td>Completed</td>
</tr>
<tr>
<td>Modify clustering code from one of Satvik’s previous projects to retrieve ETDs from our index for training</td>
<td>Oct 31</td>
<td>Jason, Ujjwal</td>
<td>Completed</td>
</tr>
<tr>
<td>Leverage kNN embeddings from Elasticsearch</td>
<td>Nov 9</td>
<td>Jason, Ujjwal</td>
<td>Completed</td>
</tr>
<tr>
<td>Cluster documents based on Title/Abstract</td>
<td>Nov 17</td>
<td>Ujjwal</td>
<td>Completed</td>
</tr>
<tr>
<td>Cluster users based on interested topics in profile data from front-end</td>
<td>Nov 28</td>
<td>Jason, Harsha, Ujjwal</td>
<td>Future Work</td>
</tr>
<tr>
<td>Recommendation experiments</td>
<td>Nov 28</td>
<td>Harsha</td>
<td>Future Work</td>
</tr>
</tbody>
</table>
6 Future Work

The following items can be worked on by future groups.

1. Optimize the performance of the kNN search method to achieve faster results with 500K ETDs.

2. Evaluate the efficiency of RRF using Hit Rate and Mean Reciprocal Rank (MRR).

3. Establish test suites to be executed during CI/CD pipeline runs, ensuring code quality.

4. Create an API for the front end to retrieve recommended ETDs from the recommendation model when the user clicks on a new ETD.

5. For recommendations, implement a User-Item model using LightFM [14] and implement a User-User model by clustering based on ProQuest categories and topics generated from the topic modeling system, each selected by the user when they register at the front end.

6. Form initial user recommendation list by using a query that is constructed from a user’s selected topics and categories.

7. Create APIs for further training the recommendation models when new users join the system and new ETDs are added to the library.

8. Create indexing APIs for curators.

9. Design and implement a log rotation strategy for the Elasticsearch log index.
7 User Manual

7.1 Search

When users initiate a search, they can specify parameters along with the search query. These parameters include:

- **Search field** – Users can provide any search phrase, and the query is executed against the title and abstract of the document set.

- **Search Method** – By default, the search utilizes Elasticsearch with the BM25 algorithm. However, users have the option to choose other algorithms such as kNN or a hybrid approach that combines both BM25 and kNN.

Unfortunately, the UI screens were not completely set up as part of Team 6 efforts, so we’ll illustrate how the search functionality works using API calls.

To initiate a search, users can make API calls with the following request body structure (see Fig. 7):

```
POST ▼ https://search.endeavour.cs.vt.edu/api/v1/search

Params | Authorization | Headers (B) | Body | Pre-request Script | Tests | Settings
--- | --- | --- | --- | --- | --- | ---

| none | form-data | x-www-form-urlencoded | raw | binary | JSON |

```1
2 "field": "keyword",
3 "method": "traditional",
4 "query": "machine learning",
5 "user":{
6 "email": "brucemayne@gmail.com",
7 "familyName": "Wayne",
8 "givenName": "Bruce",
9 "googleID": "118226734211629172868",
10 "imageURL": "https://lh3.googleusercontent.com/a/ALM6mu1AqqcFw-EKm0qV7VhKx7nCtole-oUnKMSG8-=-s96-c",
11 "name": "Bruce Wayne"
```

Figure 7: Search API Request Body

The response to the API call will retrieve documents based on the user’s query and specified parameters. Each result will display the document’s abstract and title, along with additional details such as authors, publisher, and year of publication (see Fig. 8). Currently, the system supports ETD documents and chapter objects.
7.2 Experiments

7.2.1 Create a Search Experiment

Creates a search experiment with the chosen parameters.

**POST** https://search.endeavour.cs.vt.edu/api/v1/experiment/create

The request body is shown in Fig. 9.
Figure 9: Creating Search Experiment API Request Body

Responses

- **200**: Experiment added
- **400**: Experiment already exists

7.2.2 Run Search Experiment

Conducts a search experiment with a provided query and provides a list of ETDs along with their corresponding search scores, using either default or custom settings.


The request body is shown in Fig. 10.

```
1   "k": 10,
2   "knn_weight": "0.50",
3   "name": "experiment_all-mpnet-base-v2_default",
4   "query": "machine learning",
5   "query_vector": ""
```

Figure 10: Run Search Experiment API Request Body

Responses
• **200**: Returns a list of ETDs and the scores associated as shown in Fig. 11.

![Figure 11: Run Search Experiment API Response Body](image)

### 7.2.3 Delete Search Experiment

Deletes a search experiment with the given name.

**POST** [https://search.endeavour.cs.vt.edu/api/v1/experiment/delete](https://search.endeavour.cs.vt.edu/api/v1/experiment/delete).

The request body is shown in Fig. 12.

```json
{
  "name": "test",
  "user": "Harsha Bhamidipati"
}
```

![Figure 12: Delete Search Experiment API Request Body](image)

**Responses**

• **200**: Deleted Experiment.

The response body is shown in Fig. 13.

![Figure 13: Delete Search Experiment API Response Body](image)
7.2.4 List User Related Search Experiments

List all search experiments with the given username.

**GET** https://search.endeavour.cs.vt.edu/api/v1/experiment/<username> .

**Responses**

- **200**: Lists default and user specific experiments.

The response body is shown in Fig. 14.

![Figure 14: List user specific Experiments API Response Body](image)

7.2.5 Search Experiment Over a collection

Performs Search experiment after forming a collection.


The request body is shown in Fig. 15.
Responses

- **200**: List of documents with scores.

### 7.3 Logging

#### 7.3.1 Saving Logs

The search APIs automatically internally trigger the corresponding log service to save logs seamlessly. Furthermore, teams and services can utilize the "/application/log" API as the primary method for log saving. Teams can send any amount of meta data in the request body, which will be saved in log index's meta field after processing. The API will expect "type" and "user_activity" in the request body and "User" in the header of the API call. It is essential to adhere to the specified log format to maintain consistency and maximize effectiveness. Additionally, for the efficient handling of bulk log data from authorized sources, a dedicated "logs" Kafka topic has been established.

Comprehensive example and sample log format is available below in the CURL code, serving as a helpful reference for users to accurately structure and input log data.

```bash
curl --location 'https://search.endeavour.cs.vt.edu/api/v1/application/log' \
    --header 'User: UNIQUE_NAME_OF_SERVICE' \
    --header 'Content-Type: application/json' \
    --data '{
        "data" : "Your error message",
        "type" : "INFO",
        "user_activity": "application_log",
        "meta_field1": "data1",
        "meta_field2": "data2",
        "meta_field3": "data3"
    }
'
```

Figure 15: Search experiment over a collection API Request Body

---

1. "size": 50,
2. "search_method": "traditional",
3. "name": "test_search_on_collection",
4. "query": "machine learning"
Alternatively, data sent to the "logs" topic of Kafka will save the bulk logging data in the Elasticsearch container’s storage. Any team can choose to use this service to log their data by choosing unique user_id to retrieve it later via a GET API using the same user_id. For further code implementation about the same, refer to Listing 9.

7.3.2 Retrieve Logs

To retrieve their personalized log data, users can employ a simple GET request directed to the designated endpoint: "GET https://search.endeavour.cs.vt.edu/api/v1/logs". It is imperative to include a specific "UserId" within the request header, serving as the unique identifier for the desired logs. This meticulous inclusion of the user’s identification enables the API to fetch logs tailored precisely to the individual user, facilitating seamless access for analysis, debugging, or review of personalized log data. This straightforward API call streamlines the process, ensuring swift access to logs pertinent to a specific user for various diagnostic or analytical purposes.

```
curl --location 'https://search.endeavour.cs.vt.edu/api/v1/logs' \
    --header 'User: user_id' \
    --header 'Content-Type: application/json'
```

7.4 Recommendation

This section explains how the User Interface team can use the recommendation system.

The recommendation system has 2 APIs.

7.4.1 Get Recommendations for User

Endpoint to get recommendations for a particular user.

**POST** https://search.endeavour.cs.vt.edu/api/v1/recommendations .

The request body is as shown in Listing 5.

```
1 { 
 2   "user_id": 1
3 }
```

Listing 5: Request body to get recommendations for a user_id.

It returns an array of recommendations as shown in Listing 6.

```
1 { 
 2   "recommendations": [
 3     337596,
 4     337597,
 5     337598,
 6     337599,
 7     337600,
 8     337601,
```
7.4.2 Update Recommendations for User

Endpoint to update recommendations for a particular user.


The request body is as shown in Listing 7.

```json
{
  "user_id": 1,
  "etd_id": 337636
}
```

Listing 7: Request body to get recommendations for a user_id.

Responses:

- **200**: Recommendations updated for the user.
8  Developer Manual

Most of the work done as part of this report is a continuation of the work done in [17]. Readers are encouraged to go over the developer manual in [17] to better understand the setup of the Elasticsearch client and the code related to various search methods.

8.1  Search

8.1.1  Embeddings

The previous system supported the indexing of embeddings via upload of the vectors in a CSV file. With the help of SMEs, we decided to use e5-large-v2 [26] as the embedding model for the system. The model can be used to create a dense vector of 1024 dimension of any string. Dense vectors need to be created at 2 places:

1. When metadata needs to be indexed into the kNN index.
2. When a kNN search needs to be executed, the query string needs to be converted into a dense vector.

It can be used as shown in Figure 16.

```python
model = SentenceTransformer('intfloat/e5-large-v2')
vector = model.encode(vector_str, normalize_embeddings=True)
```

Figure 16: Usage of e5-large-v2.

8.1.2  Indexing

There was a change in the database schema during the semester. Team 5 created new APIs to support read and write operations on this database. To index data from this database we had to write a new indexing script. Figure 17 shows the flow of the indexing script.
8.1.2.1 APIs

There are 2 API’s being called:

1. GET https://mainapi.endeavour.cs.vt.edu/team2/etd-metadata: This API returns the IDs of all the ETDs that exist in the ETD-Metadata table.

2. https://mainapi.endeavour.cs.vt.edu/team2/etd-metadata/{etd-id}: This API returns the metadata of 1 particular ETD. A sample response is shown in Figure 18.

Both of the APIs require an API key to be accessed.
8.1.2.2 Filtering

To ensure data quality, we have applied the following filters in our script:

1. The title should not be null.
2. The length of the title should not be less than 5 characters.
3. The abstract should not be null.
4. The length of the abstract should not be less than 10 characters.

If the ETD metadata passes these checks, we will move on to the next stage.

8.1.2.3 Creation of Embeddings

To create the embeddings for ETD metadata, we want to include as much useful data as possible. Once the metadata passes the filtering stage, we form a string by concatenating the title and abstract with a space in between them. The keywords field shown in Figure 18 could be useful in giving more information about the ETD. Hence, if the keyword field is not null, we include the keywords field in the final string. Then, as per Section 8.1.1, we create the embeddings.

8.1.2.4 Execution of the Script

There are a few things which need to be set in the indexing script before executing it.

The API for retrieving all ETD IDs provides the IDs in a random order and lacks pagination, creating a challenge for parallel processing of ID lists. To address this issue, we have implemented code to store the list of IDs in a text file. This ensures that we have a stable and predefined list of IDs to facilitate more efficient processing. The script accepts options as shown in Figure 19.
There are 2 steps required to execute the script:

1. The aim of this step is to get the list of IDs into the etd_ids.txt file. To do this, execute the command "python3 scripts/index_etds_new_apis.py getids".

2. Since we have the IDs stored, we can now use them to run parallel batches of indexing. For example, if you want to index 500K ETDs in 2 parallel executions of 250K each, you can give the command "python3 scripts/index_etds_new_apis.py -o 1 -e 250000 index" followed by "python3 scripts/index_etds_new_apis.py -o 250000 -e 500000 index"

In our complete indexing run, the script took 45 hours and 45 minutes to process 500,000 ETDs. Out of these, 182,689 ETDs did not pass the title/abstract filtering outlined in Section 8.1.2.2.

8.1.2.5 Future Work

- Reduce execution time of the script.
- Tune the filtering parameters. Explore changes to the number of character limit for title and abstract. Do any of the other metadata fields need filters or transformations?
- Consider creating 2 scripts instead of one. The first script will get the ETD IDs and the second script would index the data. This approach might make it easier to use the scripts.
8.1.3 Security

To enable security in Elasticsearch, the "xpack.security.enabled" environment variable should be configured with the value "true." As we are utilizing Docker containers for our operations, it is essential to set this variable within the configuration tab of Rancher specifically for the Elasticsearch container. Once the configuration is set, restart the container.

To use Elasticsearch with authentication, we need to generate usernames and passwords for various tasks. In this semester, Team 5 helped us generate usernames and passwords. Once generated, the username and password need to be set as environment variables in the search container. This should be done through the config tab in Rancher.

The username and password should be provided to the developers working on search and indexing. Developers need to add these credentials to their .env file to perform any development activities with Elasticsearch.

8.1.4 Searching

8.1.4.1 Traditional Search with BM25

A search query, or query, is a request for information about data in Elasticsearch data streams or indices. A search consists of one or more queries that are combined and sent to Elasticsearch. Documents that match a search’s queries are returned in the hits, or search results, of the response. Figure 20 shows the Python client code to search using Elasticsearch default method, which employs the BM25 model.

```python
search_query = {
    "query": {
        "query_string": {
            "query": query
        }
    }
}

response = es_client.search(index=index, body=search_query,
                            source=out_fields, source_excludes=exclude_fields, size=20)
```

Figure 20: Elasticsearch BM25 Search Query

8.1.4.2 k-Nearest Neighbor Search

A kNN search query must include the vector of the search query (query vector), dense vector field in the index (field), the number of nearest neighbors returned as top hits (k), and number of candidates to be considered per shard (num candidates). We set k to 10, but this can be tweaked for
experimentation. Figure 21 shows the kNN search query, and its implementation using the Python ES client.

```python
search_query = {
    "field": "vector",
    "query_vector": model.encode(query),
    "k": 18,
    "num_candidates": 100
}
response = es_client.search(
    index=index, knn=search_query, source=out_fields, size=20)
```

Figure 21: Elasticsearch kNN Search Query

8.1.4.3 Hybrid Search with RRF

In the previous system, a combined search was achieved using linear combination, assigning weights to BM25 and kNN search methods. We have improved this to use RRF, where you don’t assign weights manually to the search methods. In this approach, we perform a search query with BM25 and kNN, obtaining their results. We then apply RRF (see Fig. 23) to get the final combined ranking. The weight of lower-ranked ETDs (k) is set to 60, but this can be tweaked for experimentation. Figure 22 demonstrates the hybrid search query with RRF and its implementation using the Python ES client.
Figure 22: Elasticsearch Hybrid Search Query with RRF

```python
# Traditional search query
traditional_search_query = {
    "query": {
        "match": {
            "abstract": {
                "query": query
            }
        }
    }
}

traditional_response = es_client.search(index=index, body=traditional_search_query, size=20)
traditional_hits = traditional_response['hits']['hits']
traditional_ranks = {hit['_id']: idx + 1 for idx, hit in enumerate(traditional_hits)}

# kNN search query
knn_search_query = {
    "knn": {
        "field": "vector",
        "query_vector": model.encode(query),
        "k": 20,
        "num_candidates": 100
    }
}

knn_response = es_client.search(index=index, body=knn_search_query, size=20)
knn_hits = knn_response['hits']['hits']
knn_ranks = {hit['_id']: idx + 1 for idx, hit in enumerate(knn_hits)}

# Apply RRF
k = 60
rrf_results = apply_rrf(traditional_ranks, knn_ranks, k)

# Uncomment this method if you want to get document IDs
# rrf_ids = [doc_id for doc_id, _ in rrf_results]
# return rrf_ids

# Fetch documents based on RRF scores
documents = []
for doc_id, _ in rrf_results:
    # Fetch the document by its ID (to compare results and efficiency)
    doc = es_client.get(index=index, id=doc_id)["_source"]
    documents.append(doc)

return documents
```
8.2 Logging

8.2.0.1 Logging Service and Kafka Usage

The structure of the logging service, invoked by the "/application/log" API, is shown in Listing 8.

```python
def apply_rff(traditional_ranks, knn_ranks, k):
    rff_scores = {}
    for doc_id, rank in traditional_ranks.items():
        rff_scores[doc_id] = 1.0 / (k + rank)
    for doc_id, rank in knn_ranks.items():
        rff_scores[doc_id] = rff_scores.get(doc_id, 0) + 1.0 / (k + rank)
    return sorted(rff_scores.items(), key=lambda x: x[1], reverse=True)
```

Listing 8: Log Service Structure

This function call to the logging service captures essential log details, including the Elasticsearch client ("es_client"), log type ("type"), user identifier ("user_id"), user activity ("user_activity"), API endpoint ("api_endpoint"), and additional metadata ("meta") provided in the "body" parameter. Specific parameters like "query", "etd_id", and "chapter_id" can be included when relevant. This structured approach ensures comprehensive log data collection for analysis and debugging purposes.

The Listing 9 code snippet demonstrates how log data in the specified JSON format can be pushed to the "logs" Kafka topic using a Kafka producer. Data sent to this topic will save the bulk logging data in the Elasticsearch container storage. Any team can choose to use this service to log their data by choosing a unique user_id to retrieve it later via a GET API using the same user_id. Adjust the log_data variables to represent actual log data according to the specified structure. The producer sends an array of JSON log data to the Kafka topic, which can then be consumed
and processed accordingly. Adjust the KafkaProducer initialization with your specific Kafka broker details.

```python
from kafka import KafkaProducer
import json

# Example log data in the specified format
log_data_1 = {
    "type": "log_type_1",
    "user_id": "user_id_1",
    "user_activity": "user_activity_1",
    "api_endpoint": "/your_endpoint/log",
    "query": "Machine Learning in Medical",
    "etd_id": 49834,
    "chapter_id": 39382,
    "meta": {
        "key1": "value1",
        "key2": "value2"
    }
}

log_data_2 = {
    # Define your second log data in the same format...
}

# Creating a Kafka producer
producer = KafkaProducer(bootstrap_servers=['your_kafka_broker_host:port'])

# Pushing logs to the Kafka topic "logs"
producer.send('logs', json.dumps([log_data_1, log_data_2]).encode('utf-8'))
```

Listing 9: Sample Kafka Producer code

8.2.1 Creating New API with Logging Service Functions

Any new API created should be implemented with the logging service so that API interaction could be logged. To utilize the provided functions (log() and get_logs()) for logging functionality:

1. API Endpoint Implementation:
   (a) Define API endpoints in your application, each corresponding to specific functionalities (e.g., logging, log retrieval).
   (b) Implement these endpoints using a framework (e.g., Flask, Django), incorporating the provided functions within their respective routes.

2. Logging Service Integration:
(a) Upon receiving requests at the designated logging endpoint, invoke the log() function to store logs based on the received data.

(b) Extract necessary parameters from the incoming requests (e.g., user ID, activity, endpoint, etc.), and pass these parameters to the log() function to save logs to the Elasticsearch cluster.

(c) Use the get_logs() function when handling requests for log retrieval, passing the appropriate user ID to retrieve logs associated with that user.

# Example usage within API routes

```python
# Example usage within API routes
import json
from flask import request
from flask import Blueprint
import services.logging_service as logging_service

# Endpoint to make new service and save its logs
@blueprint.route('/sample/create', methods=['POST'])
def run_sample_process():
    user_id = request.headers.get('User')
    vector_file = request.files['file'].stream
    body = request.form
    # Run your api processes
    # Logs the data after the process is done
    logging_service.log(
        es_client=es_client,
        type="INFO",
        user_id=user_id,
        user_activity="POST",
        api_endpoint="/sample/create",
        etd_id=body.etd_id,
        chapter_id=None,
        meta=body
    )
    response = {'message': "OK"}
    return response

# Endpoint to retrieve logs
@blueprint.route('/sample/logs', methods=['GET'])
def get_logs():
    user_id = request.headers.get('User')
    return json.dumps(logging_service.get_logs(user_id, es_client))
```

Listing 10: Example Usage.

Listing 10 illustrates a basic approach. Ensure that you handle errors, perform data validation, and implement appropriate security measures within your API routes. Replace the es_client variable with the Elasticsearch client instance. Adjust the routes, parameters, and functionality based
on your specific application requirements and framework. This conceptual approach showcases how the provided methods can be integrated into an API structure for logging and log retrieval functionality within a larger application context.

8.3 Recommendation

The design and high-level functioning of the recommendation system is explained in Section 4.2.4. This section will detail the specific technical implementation details of the system.

The recommendation system is divided into 2 parts.

8.3.1 Initialization and Pre-compute

The initialization and pre-compute is triggered as soon as the Flask server is started. The steps involved are:

1. Get all the data from the kNN Elasticsearch index.
2. Cluster the ETDs into a fixed number of clusters.
3. Compute the cosine similarity scores within each of the clusters.

The code for all this is in service/clustering.py.

8.3.2 APIs

Please refer to Section 7.4 for the request body and response body for the recommendation APIs.

1. **Get recommendations**: If the current request is the initial recommendation request for a specific user_id, the code allocates 10 random recommendations from the initial response obtained in step 1 of Section 8.3.1. In the event that recommendations have been previously requested for the user_id, they will be stored in the Hash Map "user_recommendations" within the "service/recommendations.py" file. In such cases, the code simply returns the value associated with the user_id.

2. **Update recommendations**: This endpoint receives a user_id and etd_id as input. For the etd_id, the 5 nearest neighbors within its cluster is calculated. The existing recommendations in the list are then shifted five positions down, making room for the top five nearest neighbors, which now become the user’s primary recommendations.

8.3.3 Future work

- The initialization is done every time the Flask server starts. We might need some sort of mechanism to choose whether we want to trigger the initialization (environment variable, setting).
• Currently, everything is being stored in memory. It would be better to store the ETD-ETD relations and User-User relations in the database tables ETD-ETD_neighbors and User-user_neighbors, respectively.

• The number of clusters is controlled by a variable "number_of_clusters" in service/cluster-ing.py. It would be better to put this in a configuration file.

• The number of ETDs to be processed is controlled by a variable "number_of_etds_to_process" in service/cluster-ing.py. It would be better to put this in a configuration file.

• When the recommendations are returned, only the ETD IDs are being returned. The ideal design would be to return the relevant metadata fields like title and author along with the ID.
References


