Unsupervised Dendrogram Text Index and Search using Hierarchical Clustering

Michel Abboud
E.C.E. department
Lebanese American University
Byblos, Lebanon
michel.abboud@lau.edu

Charbel Aoun[1]
Institut Catholique d'Arts et Métiers
ICAM, School of Engineering
Toulouse, France
charbel.aoun@icam.fr

Joe Tekli ✉
E.C.E. department
Lebanese American University
Byblos, Lebanon
joe.tekli@lau.edu.lb

***Abstract*—Text data retrieving depends initially on the data indexing, emphasizing the role of different indexing techniques used for query processing. In this context, traditional Information Retrieval (IR) techniques based on the inverted index share a number of challenges including limited scalability, accuracy, and semantic awareness. In this study, we provide an improved solution by introducing a new Machine Learning (ML) indexing technique based on hierarchical clustering. Our Unsupervised Dendrogram Index and Search (UDIS) approach introduces a dendrogram-like tree index structure that allows linking data items according to their clustering in the document corpus, providing data co-occurrence and semantic context in index building and querying. UDIS aims at improving the IR imbedding with a semantic tree structure, allowing improved quality and seamless integration of semantic-aware search results. We have empirically evaluated the impact of UDIS index size and query execution, and their relationship with search quality. Experiments show improved results compared with unigram and bigram AND-and-OR inverted indices.**

Keywords*—*Text indexing, Text querying, Information retrieval, Unsupervised learning, Data clustering, Dendrogram.

I. Introduction

With its exponential growth and massive volume, data generation has taken huge leaps in the last couple of years as it reached the zettabyte range, and it is continuously rising [21]. This phenomenon, also known as Big Data, introduces various challenges in terms of storage, security, quality, and retrieval of data. Information retrieval (IR) is defined as the process by which a collection of data is represented, stored, and searched for knowledge discovery as a response to a user request (query). One key application that demonstrates the importance of information retrieval is search engines such as Google, and Bing, among other platforms. Due to the rapid increase and diversity of Web data, it is becoming more challenging for IR solutions to provide users with information that satisfies their needs [6]. Retrieving data from documents depends initially on how the data is indexed, emphasizing the role of different indexing techniques used for efficient query processing. Traditional IR techniques are based on the inverted index structure which perform one-by-one query processing, and share a number of challenges including limited scalability, accuracy, and semantic awareness. In this study, we address the afore mentioned challenges, and provide an improved solution by introducing a new Machine Learning (ML) indexing technique based on hierarchical clustering. Our Unsupervised Dendrogram Index and Search (UDIS) approach introduces a dendrogram-like tree index structure that allows to link data items according to their clustering in the document corpus, providing data co-occurrence and semantic context in index building and querying. We aim to improve the IR imbedding within a semantic tree structure, allowing improved quality and seamless integration of semantic-aware search results. We have empirically evaluated the impact of UDIS index size and query execution, and their relationship with search quality. Experiments show improved results compared with unigram and bigram AND-and-OR inverted indices.

Section 2 briefly describes the related works. Section 3 presents our proposal. Section 4 describes the experimental evaluation, before concluding in Section 5 with future works.

II. Related Works

In Information Retrieval (IR) and database (DB) systems, handling a user query search request is an important task that needs to be fast and accurate, to allow a good user experience. To accomplish this, indexing structures are used as a backbone to almost any IR/DB engine [16]. This section briefly describes legacy and ML-based indexing techniques.

A. Legacy Text Indexing Techniaues

Most search engines use an inverted index to store their documents for IR. As the name suggests, inverted indices reverse the document structure by storing for each word in a collection of documents the set of documents it appears in. The granularity of the position of a word stored in the inverted index can vary depending on the application needed. Usually, the size of the inverted index ranges from 5 to 100% of the size of the indexed documents, depending on the way the index is stored in compressed or uncompressed form and the preprocessing techniques applied on the documents prior to indexing. For instance, stemming or lemmatizing, removing stop-words, and removing punctuation will help in reducing the size of the index [3]. Querying the inverted index is done by searching the indexed documents and returning the documents where each word of the query occurrs. Then, a Boolean operation is applied to the results (AND or OR) to get the final result of documents.

Other legacy text indices have been developed, each with its set of properties and applications. B-tree indices consist of nodes and branches connected from top to bottom. The top-most node is called the root and the bottom-most nodes are called the leaves which are usually pointers to physical data locations [12]. B-tree indices usually have a low update, delete, and insert cost and can be seen as structures that map keys to positions within a sorted array [8]. B-trees are good for range and equality comparisons. Hash indices apply a hash function to a certain key to find where the data is stored in the disk and is best suited when working with one key at a time [4]. Bitmap indices are built using a two-dimensional array and usually index a certain column in a table. The number of columns in the bitmap is equal to the number of values that can be taken by the column to be indexed in the table plus one column to indicate the row [20]. For example, if the column to be indexed is “Student” and the values are “Yes” or “No” and there are ten rows in the table, then in the bitmap we will have three columns (Row, Yes, No) and ten rows. Next, for each row in the original table, we check its value and set respectively the values of the columns in the index to either 0 or 1 which is the reason behind the “bit” in the bitmap naming [10]. The bitmap index is best suited for columns with low cardinality [23]. Multiple variations of each technique have been developed, including B+-tree, B*-tree, hybrid B-Tree [14]. In the case of B-tree, different hashing functions can be used to reduce collisions and other issues for hash indices [7], and different compression techniques for the bitmap indices can be used [23]. Moreover, multiple hybrid techniques have emerged to solve problems in a particular indexing method, for instance, combining B-Tree and Hash map to reduce the number of hash map lookups [7]. Since each index is best suited in certain situations, choosing a proper

[1] C. Aoun is co-affiliated with the Lab-STICC, CNRS UMR, ENSTA (Ecole Nationale Supérieure de Techniques Avancées), Brest, France

979-8-3315-8747-5/25/$31.00 ©2025 IEEE

indexing technique depending on the query types presented to a DB greatly affects the performance and execution of the queries [1]. Query optimizers (QO) are usually used to get index recommendations and decide what to do with an index whether it should be created, removed, or recreated. More recent algorithms utilize Genetic Algorithms (GA) and ML algorithms to improve the indexing process. In [1], an artificial neural network (ANN) classifier is proposed as an algorithm trained to choose the most suitable index based on certain patterns learned by the ANN from a given dataset. Results show improved efficiency.

B. Semanic-Aware Text Indexing

Many recent studies have attempted to include semantic meaning in the text indexing process [5], where users who are not familiar with the data usually use terms that are syntactically different from the data which frequently leads to irrelevant results [16]. For instance, consider user query "car", and a data collection about automobiles where there is no semantic knowledge connecting "car" with "automobile". In this case, the legacy indexing structure looks at the syntactic difference only and considers these words as different, without returning any results to the user. Including semantics comes with its challenges such as performing word sense disambiguation [18], semantic query reformulation [15], and search result clustering [17]. Multiple approaches suggest incorporating semantic knowledge at the query processing level, through the expansion of the original query with a knowledge source (e.g., Wordnet [11], or a domain ontology [5]), and the usage of query relaxation, rewriting, disambiguation and refinement techniques [18]. However, these approaches generally suffer from reduced quality, speed, and limited user involvement [16]. To partly solve some of these issues, recent studies suggest integrating semantics at the most basic data indexing level [15, 16]. To accomplish this task, the authors suggest creating a combination of two graph representations of the input, one representing the textual data, and another represents the knowledge base graph of the terms in the corpus. Semantic-aware search is subsequently performance on the integrated graph structure using optimized graph-search algorithms. In [19], the authors introduce an indexing algorithm that uses spectral graph theory and semantic spanning forests built from semantic relations extracted from different thesauri (namely Wordnet and Wikipedia) to create a dense semantically enriched representation of a document, then when queried, the same processing technique is applied to the query to generate its semantic representation and then the similarity between the documents and the user query is computed using the Hausdorff distance. The semantic indexing solutions in [19] were compared with traditional indexing techniques and showed promising results.

C. Machine Learning Indexing Techniques

More recently, few Machine Learning (ML) indexing techniques have been proposed in the literature, e.g., [8, 9, 22]. With this category of techniques, an index structure is seen as a model that can be trained to predict results. For instance, a key input of the position of an object in a sorted array can describe a B-tree, and can be viewed as a hash map. In addition, a bitmap or bloom filter can be seen as a model trained to output whether a record exists or not. The ML model can dynamically learn the distribution of the data which might be challenging for traditional indices. For instance, in the case of hash functions, a model can be used to learn a hashing function that fits the data with the lowest conflicts possible which is applicable in both hash map and bloom filter indices. For instance, the authors in [8] have trained a two-layer fully connected neural network, and compared it with a B-tree, resulting in a 70% faster indexing and less memory consumption. Furthermore, the authors in [8] propose three important models: the learning index framework (LIF), the recursive-model indices (RMI), and hybrid indices. The LIF is based on Tensorflow to learn simple models and generate index configurations accordingly. The RMI is a hierarchy of models where at each level a model predicts which of the next level model, it thinks knows better the needed location with a minimal error. The hybrid method merges between the RMI and B-tree indices, replacing a model with a B-tree whenever its performance is worse than a B-tree. Experiments showed improved results compared with the traditional indices: i) the learned B-tree and bloom filter indices were faster in terms of build time and lookup and had smaller sizes compared with their legacy counterparts, and ii) the learned hashing function in the hash map index was able to greatly reduce the conflict number [8]. In [9], the authors introduce SmartIX, another ML approach to automatically index DBs using reinforcement learning. The method is based on an agent which job is to choose an index for the DB. The agent is a module of five components. The first stage is to transform the current index of the DB into the agent state to test its performance on a benchmark (TCP-H) in the second step, and then a learning algorithm will be rewarded based on the results. Finally, an exploration function will decide what to do next: whether to further investigate the new information, or look at other actions. Empirical results in [9] show improved query time and index storage size, compared with legacy indexing solutions. In [22], the authors use a Convolutional Neural Network (CNN) to implement a semantic index for biomedical documents. They first generate the feature representation of the documents using Wikipedia categories and Metamap, then the output of this step is fed to a CNN with a ReLU activation function and 50% dropout in all the layers. Next, the output of the CNN is fed to a sequence of two classifiers: the first one predicts one out of a chosen number of independent categories, and the second is more specific predicting a subcategory inside the previously chosen category. Experiments showed accuracy results which are on a par with and sometimes surpassing existing methods.

III. MOTIVATION

When used with full text search, the traditional inverted index and querying techniques showcase many challenges, including: i) lack of semantics, ii) lack of query coherence where queries are processed as separate individual terms, and iii) drawbacks of intersection and union queries. We illustrate these issues below.

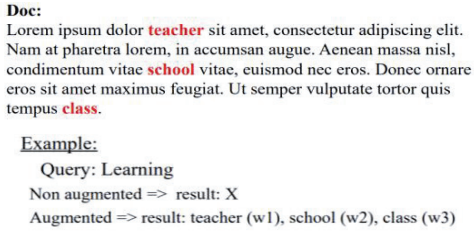


Figure 1. Sample document and query result

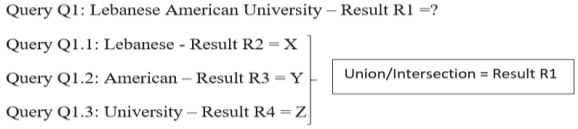


Figure 2. Sample keyword queries

Lack of semantics: Figure 1 illustrates the problem. We notice that the non-augmented index cannot return related words

and only looks for exact matches which can affect retrieval accuracy and might not satisfy the users' needs.

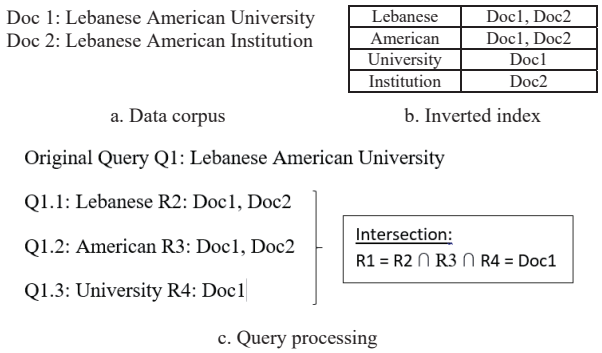


Figure 3. Query processing using intersection

Lack of query coherence: the query is processed as separate individual term-based queries, thus it does not consider the whole initial query as one single unit of information which might lead to loss of information, returning incomplete results. Figure 2 illustrate an example, where the target query is processed as three separate queries. The results are then joined using set theory through intersection or union which do not always cover the whole initial query.

Drawbacks of intersection and union: Using intersection to join the results of multiple sub-queries can produce limited information about the results. For instance, consider the documents and their inverted index in Figure 3. We notice that the result is only Doc1 even though Doc2's content are also related to Q1. Based on the results, we can deduce that only Doc1 contains all the keywords in the query, but we cannot know whether there exist other documents that are partly related to Q1 having a certain number of keywords occurring in Q1. For instance, obtaining as a result: R1 = <Doc1 (100%), Doc2 (75%)> would be more informative and complete rather than getting only Doc1.

Using union to combine individual sub-query results into one list might include partly irrelevant/noisy results. Consider for instance the documents and inverted index in Figure 4. We notice that even though Doc2 is not quite relevant to Q1, yet it is returned as a result. Similarly, consider a large text having the word "Institution" in it only once, then it will also be returned as part of the result and will thus negatively affect the precision of the query execution.

Potential solutions: using N-grams and weighed terms. Considering N-grams to highlight the combinations of query terms as individual units of information might solve part of the previously stated challenges, however, it will also drastically increase the size of the index, creating an index scalability problem. Consider for instance the indices in Table 1 presenting two index tables: unigram only and unigram+bigram. We notice that the number of entries of the index increases from 3 to 5 if we add only consecutive words, showing the importance of having these words with each other in the same order. However, if there is a need to add all combinations or other n-grams, then the number of index entries will significantly increase accordingly. Note that weights can be added to highlight the relevance of each index entry in describing the indexed document(s). This process will need more storage space as well as more creation and processing time (cf. empirical evaluation in Section 5).

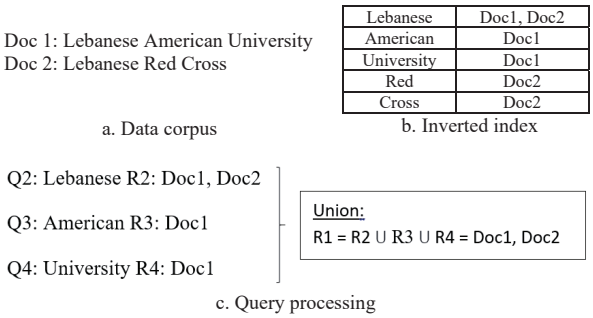


Figure 4. Query processing using union

The above mentioned challenges have motivated us to design a new unsupervised index solution allowing to process a user query holistically, rather than individual terms, aiming to consider the query's semantics in the indexing structure and improve the traditional inverted index technique by leveraging the power of ML in this field.

Table 1. Inverted indices: unigram and bigram

a. Unigram

Lebanese	Doc1, Doc2, Doc3
American	Doc1, Doc2, Doc4
University	Doc1

b. Unigram + Bigram

Lebanese	Doc1, Doc2, Doc3
American	Doc1, Doc2, Doc4
University	Doc2
Lebanese American	Doc1, Doc2
American University	Doc2

IV. PROPOSAL

The architecture of our Unsupervised Dendrogram Index and Search (UDIS) is show in Figure 5. It consists of two components: i) offline index building and ii) online index search. On the one hand, the offline component includes parsing, preprocessing the documents, clustering the documents, and creating the hierarchical dendrogram clustering structure.

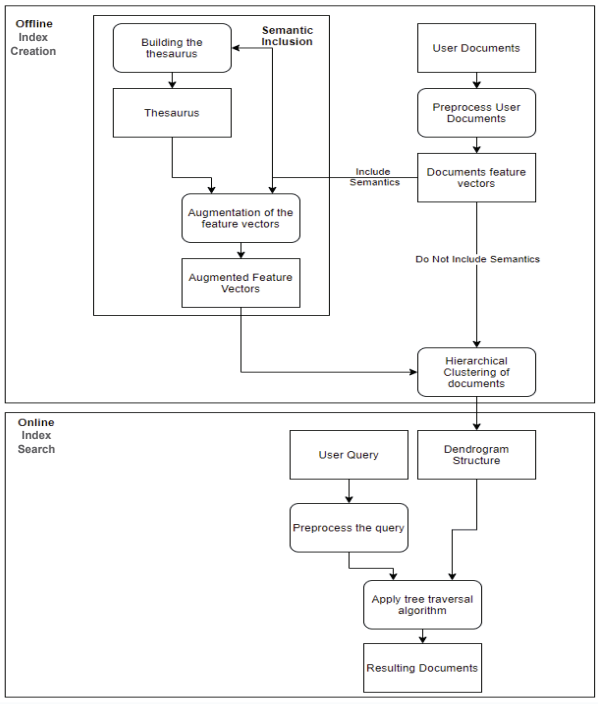


Figure 5. Overall architecture of USSI

To perform semantic augmentation, two additional offline steps are added to include semantics in the structure: i) building the thesaurus and ii) augmenting the feature vectors prior to clustering the documents. On the other hand, the online component performs real-time execution of the hierarchical structure traversal algorithm, accepting a user query as input and producing the relevant result documents as output. The input query is preprocessed similarly to the offline processing pipeline, and the results are adapted to user-chosen thresholds.

A. Pre-Processing and Clustering

Figure 6 describes our preprocessing pipeline, allowing to generate the document TF-IDF feature vectors which will be augmented through an inputted thesaurus or directly fed to the hierarchical clustering algorithm. Data serialization is achieved using the python *pdftotext* library which converts input documents into a text string. The lemmatization and stop words are handled using the python NLTK library. Generating the feature vectors is achieved using the sklearn TF-IDF vectorizer and are saved in a Pandas dataframe whose columns represent the feature words, rows represent each document, and each row 'i' – column 'j' pair contains the weight of the word 'j' in the document 'i' based on the TF-IDF concept. Next, the dataframe of feature vectors is used to create a dendrogram based on the cosine distance similarity through the SciPy hierarchical clustering library. The outputted SciPy linkage matrix is used to build the dendrogram index structure (section IV.D).

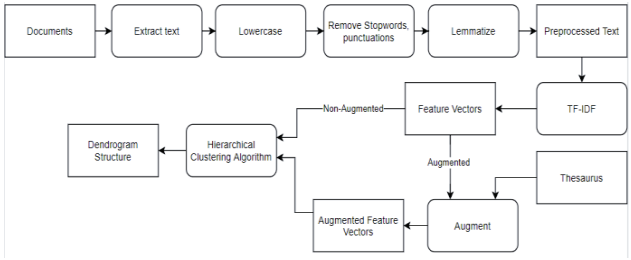


Figure 6. Document preprocessing and clustering

B. Thesaurus Building

Our thesaurus building component is based on the distributional thesaurus generation algorithm we previously developed in [13]. It accepts as input: a text corpus C, as well as input parameters designating the co-occurrence *window size* and the number of *top-ranked terms* needed to identify related terms. The output thesaurus consists of the list of distinct terms from C, where every term t_i is associated a co-occurrence vector $\overrightarrow{V_{Occ}} = < occf(t_i, t_j), occf(t_i, t_k), \ldots >$ providing the co-occurrence frequencies of the top terms co-occurring with t_i in C. The user can choose to generate the thesaurus form the document corpus accepted as input for topic extraction, or can be generated based on an external corpus. The latter needs to be chosen to describe the target documents at hand, since the effectiveness of the thesaurus depends on the lexical coverage of its reference corpus.

C. Feature Vector Augmentation

We use the generated thesaurus to augment the feature vectors and enrich them with new semantically related words. Consider t_i a word in our corpus and Y the set of words (y_1, y_2, y_3, …) related to t_i in the thesaurus, and consider occ_{i-j} the number of times t_i and y_j co-occur together in the reference corpus. To get the normalized semantic relatedness, we divide nb_{i-j} by the maximum number of occurrences *MaxOcc* between any two words in the thesaurus, denoted R_i. Next, we update the original weight of t_i in the feature:

$$\text{Weight}(t_i) = \text{Weight}(t_{i\,old}) + \sum_{Y_i \,in\, Y} \text{Weight}(Y_{i\,old}) * R_i * \text{alpha} \quad (1)$$

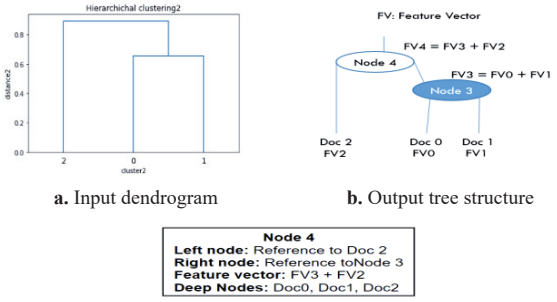
where *alpha* $\in$ [0, 1] is used to specify the effect of the co-occurrence on the augmented weights.

$$\text{Weight}(Y_i) = \sum_{t_i \,in\, C} \text{Weight}(X_{i\,old}) * R_i * \text{alpha} \quad (2)$$

To implement this behavior, we copy the original dataframe of feature vectors to make sure we have the old values of t_i and then we loop over our features, check their related words, and perform the augmentation on the columns in the dataframe.

D. Dendrogram-based Index Structure

Following feature vector augmentation, we represent the output of the hierarchical clustering algorithm as a dendrogram tree-like structure that contains the needed information to apply our traversal algorithm (described in Section IV.E). Tree nodes represent documents and centers of each dendrogram component, having the following attributes: left node, right node, feature vector, deep nodes. At each center, the feature vector is calculated as the sum of its children feature vectors, where deeper nodes are processed recursively based their descendent nodes (Figure 7).



a. Input dendrogram **b.** Output tree structure

c. Node attributes

Figure 7. Index tree representation

E. Index Traversal Algorithm

While the construction of the index tree structure is done bottom-up, its traversal is done from the top-down. The input is a user query that contains a free text. This text will be pre-processed in the same way as indexed documents, then a feature vector is generated while making sure it has the same dimensions as our queried data and will be fed to the algorithm. Next, the user feature vector will be inserted at the root of the index tree, and then it starts comparing it with the feature vectors of the left and right child nodes using the cosine distance similarity (other similarity measures can be used). Two thresholds can be set by the user to control the traversal algorithm and the search results:

- Relevance threshold $Thresh_{Rel} \in$ [0,1]: Used to specify whether we need to search the node with smaller similarity only if the small similarity is higher than or equal to the similarity of its parent multiplied by the $Thresh_{Rel}$. Behavior at the extremities: i) maximum (=1): does not consider searching the node with smaller similarity to the query, ii) minimum (= 0): always considers all the children's nodes and searches them.

- Specificity threshold $Thresh_{Spec} \in [0,1]$: used to specify when to stop searching deeper in the index tree. If a given node has a cosine similarity value with the query $\geq Thresh_{Spec}$, then we proceed to this node and continue searching deeper inside the index tree, else we stop. Behavior at the extremities: i) maximum (=1), no navigation in the index tree – returns all the documents as results (most generic), ii) minimum (= 0), navigates the index tree to reach the most similar leaf node – returns only a minimal number of documents as result (most specific), iii) default (=-1), the threshold will be dynamically computed while traversing the index structure and returns the node with the highest similarity.

Algorithm IndexTreeTraversal
Input: User query q; root node; $Thresh_{Rel}$; $Thresh_{Spec}$
Output: Query result

```
1   Result = []
2   ParentToSearch = []
3   If (Thresh_Spec == -1) Then Initiatlize Thresh_Spec = 0
4     Step 1: For each node n_i ∈ First Level Children
5     Step 2: Compute Sim(q, n_i)
6     Step 3: If (min(Sim) ≥ Thresh_Rel × Thresh_Spec) Then append n_i to ParentNode
7     Step 4: If (max(Sim) ≥ Thresh_Spec) Then
8                 If n_i is internal node Then
9                       If ThreshSpec == -1 Then ThreshSpec = max(Sim)
10                      Repeat from Step 1
11                Else If n_i is leaf node Then Add n_i to Result      // Target reached
12              Else Add all leaf nodes to Result                      // Parent node is target
13    If (ParentToSearch is !Empty) Then Apply Search on each node in
14    ParentToSearch
      Return Result
```

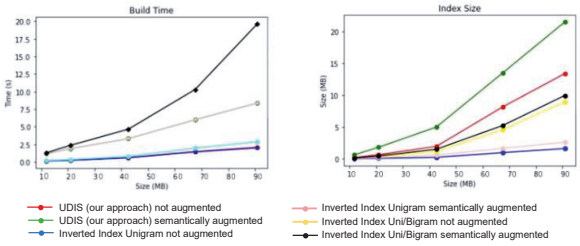
Figure 8. Index Tree Traversal Algorithm

The algorithm's pseudo-code is shown in Figure 8. It checks the highest and smallest similarity nodes: if the highest similarity node is $\geq Thresh_{Spec}$, then we navigate to it and continue the same procedure (lines 1-6). Meanwhile, if the smallest similarity node is $\geq Thres_{Spec} \times Thresh_{Rel}$, then this node is added as a parent to search later since it might also have relevant results (lines 7-12). Special cases such as two nodes having the same similarity are quite rare; however if such a case occurs and both have similarities $\geq Thresh_{Spec}$, then the algorithm navigates to one node and adds the other as a parent to search later (line13). $Thresh_{Rel}$ is chosen by the user, while $Thresh_{Spec}$ takes the value of the highest similarity in each traversal level of the algorithm. The output is a set of documents with their similarity scores ranked in descending order of their similarities with respect to the user query.

V. Empirical Evaluation

A. Experimental Process and Metrics

To evaluate the quality and performance of our approach, we compare it with multiple variations of inverted indices: i) unigram non-augmented inverted index, ii) unigram semantically-augmented inverted index, iii) uni/bi-gram non-augmented inverted index, and iv) uni/bi-gram semantically-augmented inverted index. We utilize the following metrics: 1) *index building time*: we vary the size of the input dataset, 2) *index size*: we vary the size of the input dataset, 3) *query execution time*: we vary the number of query terms and threshold values, and 4) *query result quality*: we calculate precision, recall, F-value, and MAP values of the returned documents while varying the number of query terms.



a. Index build time **b.** Index size

Figure 9. Index build time and index size

B. Experimental Results

1) Index Build Time and Size

The time and size complexity of building our index are linear in the size of the document dataset being indexed, $O(n \times |D|)$ where n is the number of documents being index and $|D|$ is the maximum document size. To evaluate the latter, we use 400 documents downloaded from the United Nations' Manara platform[2] as a test dataset. Results in Figure 9.a shows that small subset (< 40 MB) requires almost the same time as building the unigram inverted index in both non-augmented and augmented variations. In addition, creating a uni/bigram non-augmented inverted index slightly takes more time since it needs to traverse the two term features in the document. However, a significant leap is observed in build time when creating a uni/bigram augmented inverted index since additional processing is required to generate all the term combinations needed from the thesaurus, and since the inverted index was not originally created to include semantics.

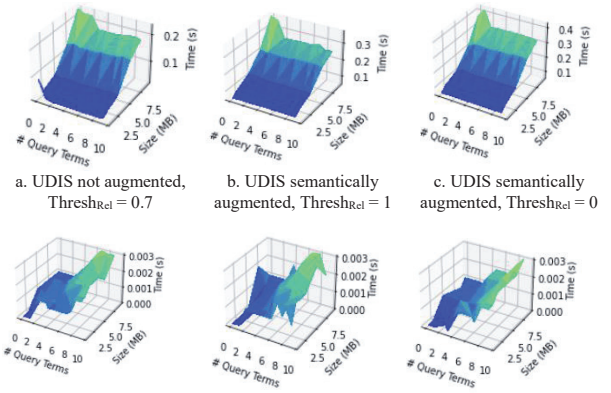
Figure 9.b shows a comparison between the different sizes of the indices were both our approaches require more space than the inverted index techniques. This is because our approach does not only save the syntactic features of the documents but also stores the structural relations built after clustering all the documents and building the tree structure where each parent node contains its unique feature vector equal to the addition of its children feature vectors and the respective references to their nodes. This additionally stored information is required to perform index traversal during the query evaluation phase.

2) Query Execution time

To evaluate query execution time, we created a 10 term-long query from random terms collected from our UN documents dataset, and then for each batch of documents, we queried all the indices 10 times starting with a 1-term query until reaching the whole 10-term query. We vary $Thresh_{Rel}$ between 0.7, 1 (returns only the best node), and 0 (returns all the leaf nodes). $Thresh_{Spec}$ is computed dynamically by our index tree traversal algorithm (Figure 8). For each of the inverted indices, we utilize both AND- and-OR operators. Results are shown in Figure 10. Here, we make the following observations. First, the time needed to execute a query for a given dataset size is almost constant and is not heavily affected by the number of query terms since in all cases the query is transformed into a feature vector that matches the dataset feature vector. Second, the query execution time increases with the size of the dataset where bigger index tree is be used, and thus the time needed to navigate through it will increase. Third, we notice a time increase when $Thresh_{Rel}$ decreases, allowing to further investigate lower similar nodes in the index tree traversal algorithm. Regarding the inverted index approaches, we notice a minor increase in execution time when

[2] https://manara.unescwa.org/home

the number of query terms and when the size of the dataset increase, reflecting index data retrieval speed.



a. UDIS not augmented, $Thresh_{Rel} = 0.7$

b. UDIS semantically augmented, $Thresh_{Rel} = 1$

c. UDIS semantically augmented, $Thresh_{Rel} = 0$

d. Inverted index, Unigram, not augmented, AND

e. Inverted index, Unigram, augmented, AND

f. Inverted index, Unigram, augmented, OR

Figure 10. Query execution time

2) Query Result Relevance

In this experiment, we used a smaller dataset composed of 17 UN documents each having a different semantic meaning (i.e., targeting each one of the 17 UN's Sustainable Development Goals – SDGs). Next, we created three sets of queries with their corresponding correct answers to evaluate three different conditions, each set is made of four queries starting with a query of 2-3-4-5 terms respectively. A 1-term query will not be of any significance since both traditional techniques and our approach and will lead to the same answer. Average results are reported in Table 2. First, selecting a $Thresh_{Rel} = 0$ returns all the documents sorted by their corresponding weights and acts as an inverted index OR approach. Second, even though we observe high recall (R) and mean average precision (MAP) values for the OR inverted index approach, its weakness is exposed through the low prevision (PR) and F-value metrics since it returns many irrelevant results with only a good ranking of these results based on the TF-IDF weights used for both approaches. Third, the AND method is prone to missing relevant results, where it performed well in the 2nd set of queries but failed in the others. Fourth, by changing $Thresh_{Rel}$, the users can tune the results according to their needs. To sum up, results in Table 2 reflect the improved quality of our indexing technique compared with existing inverted index solutions.

Table 2. Query relevance results (average PR, R, F-value, and MAP)

	Method	PR	R	F-Value	MAP
0	Our Approach 0.7	1.000000	0.535363	0.633333	0.535363
1	Our Approach 0	0.559829	0.987179	0.598088	0.962179
2	Our Approach 0.5	0.847222	0.730128	0.719907	0.730128
3	Our Approach 1	0.750000	0.867735	0.710560	0.717603
4	InvIndex Unigram AND	0.500000	0.371795	0.395833	0.371795
5	InvIndex Unigram OR	0.559829	0.987179	0.598088	0.970976
6	InvIndex Uni/Bigram AND	0.500000	0.371795	0.395833	0.371795
7	InvIndex Uni/Bigram OR	0.559829	0.987179	0.598088	0.970976

VI. Conclusion

In this study, we describe a new text indexing solution by introducing a Machine Learning (ML) indexing technique based on hierarchical clustering. We introduce a dendrogram-like tree index structure which allows linking data items according to their clustering in the document corpus, providing data co-occurrence and semantic context in index building and querying. We have empirically evaluated the impact of index size and query execution, and their relationship with search result quality. Results show improved quality compared with unigram and bigram AND-and-OR inverted index solutions.

We are currently extending our empirical evaluation on larger datasets to evaluate the quality and performance of our approach. We are also investigating the parallel execution of the algorithm, allowing to navigate the index tree structure using multi-threading, aiming to optimize query execution speed. In the near future, we plan to examine the use of supervised machine learning techniques [2] in combination with our unsupervised approach in the creation of the index. In the long run, we plan to investigate the creation of an ensemble indexing technique, combining both the inverted index and our ML-based index, to optimize both time performance and result quality.

References

[1] Albert A., et. al., *Intelligent Indexing—Boosting Performance in Database Applications by Recognizing Index Patterns.* Electronics, 2020. 9(9), 1348.

[2] Attieh J. and Tekli J., *Supervised Term-Category Feature Weighting for Improved Text Classification.* Knowledge Based Systems 2023. 261:110215.

[3] Bruch S., et al., *Efficient Inverted Indexes for Approximate Retrieval over Learned Sparse Representations.* Inter. ACM SIGIR Conf. on Research and Development in Information Retrieval (SIGIR'24) 2024. pp. 152-162.

[4] Chen M., et al., *GPHash: An Efficient Hash Index for GPU with Byte-Granularity Persistent Memory.* USENIX Conference on File and Storage Technologies (FAST'25) 2025. pp. 203-220.

[5] Dash S. and RaoS., *ECG Arrhythmia Detection Using Choi-Williams Time-Frequency Distribution and Artificial Neural Network.* Inter. J. of Advanced Research in Computer and Communication Engineering, 2016. 2278-1021.

[6] Fote F., et al., *Big Data Storage and Analysis for Smart Farming.* International Conference on Cloud Computing and Artificial Intelligence: Technologies and Applications (CloudTech'20) 2020. pp. 1-8.

[7] Hu Y., et al., *A novel hashing-inverted index for secure content-based retrieval with massive encrypted speeches.* MM Syst., 2024. 30(1): 22.

[8] Kraska T., et al., *The Case for Learned Index Structures.* ACM SIGMOD Conference (SIGMOD'18), 2018. pp. 489-504.

[9] Licks G., et al., *SmartIX: A database indexing agent based on reinforcement learning.* Applied Intelligence, 2020. 50(8): 2575-2588.

[10] Liu S., et al., *ACER: Accelerating Complex Event Recognition via Two-Phase Filtering under Range Bitmap-Based Indexes.* ACM SIGKDD Conf. on Knowledge Discovery and Data Mining (KDD'24), 2024, 1933-1943.

[11] Miller G.A. and Fellbaum C., *WordNet Then and Now.* Language Resources and Evaluation, 2007. 41(2): 209-214.

[12] Qader M., Cheng S., and Hristidis V., *A Comparative Study of Secondary Indexing Techniques in LSM-based NoSQL Databases.* ACM SIGMOD Conference (SIGMOD'18), 2018. pp. 551-566.

[13] Sarkissian S. and Tekli J., *Unsupervised Topical Organization of Documents using Corpus-based Text Analysis.* Inter. ACM Conf. on Management of Emergent Digital EcoSystems (MEDES'21), 2021. pp. 87-94.

[14] Sun J., et al., *A blockchain-based multi-keyword rank search scheme for B+ tree inverted index.* Computer Standards & Interfaces, 2025. 93: 103968.

[15] Tekli J., et al., *Full-fledged Semantic Indexing and Querying Model Designed for Seamless Integration in Legacy RDBMS.* Data and Knowledge Engineering, 2018. 117: 133-173.

[16] Tekli J., et al., *SemIndex+: A Semantic Indexing Scheme for Structured, Unstructured, and Partly Structured Data.* Elsevier Knowledge-Based Systems, 2019. 164: 378-403.

[17] Tekli J., *An Overview of Cluster-based Image Search Result Organization: Background, Techniques, and Ongoing Challenges.* Knowl. Inf. Syst., 2022. 64(3): 589-642.

[18] Tekli J., Tekli G., and Chbeir R., *Combining offline and on-the-fly disambiguation to perform semantic-aware XML querying.* Computer Science and Information Systems, 2023. 20(1): 423-457.

[19] Tsatsaronis G., Varlamis I., and Nørvåg K., *SemaFor: semantic document indexing using semantic forests.* Inter. Conf. on Information and Knowledge Management (CIKM'12), 2012. pp. 1692-1696.

[20] Wang J. and Athanassoulis M., *CUBIT: Concurrent Updatable Bitmap Indexing.* Proceedings of the VLDB Endowment, 2024. 18(2): 399-412.

[21] Yan S., Wang J., and Liang J., *Big Data Storage and Analysis System for Space Application.* Inter. Conf. on Computer Supported Cooperative Work in Design (CSCWD'24), 2024. pp. 1764-1769.

[22] Yan Y., et al., *Semantic indexing with deep learning: a case study.* Big Data Analytics, 2016. https://doi.org/10.1186/s41044-016-0007-z.

[23] Yildiz B., *Optimizing bitmap index encoding for high performance queries.* Concurrency & Computation: Practice & Experience, 2021. 33(18) (2021).