Indexing Infrastructure for Semantics Full-text Search

by

Fatemeh Lashkari

Master of Science in Computer Science, University of Gothenburg, 2012
Bachelor of Software Engineering, SUT, 2009

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy

In the Graduate Academic Unit of In the Graduate Academic Unit of Computer Science

Supervisor(s):	Ali A. Ghorbani, PhD, Computer Science
	Ebrahim Bagheri, PhD, Computer Science
Examining Board:	Bruce Spencer, PhD, Computer Science
	Arash Habibi Lashkari, PhD, Computer Science
	Donglei Du, PhD, Business Administration
External Examiner:	Masoud Makrehchi, PhD, ECE, UOIT

This dissertation is accepted by

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

July, 2019

Abstract

The increasing effectiveness and wide spread use of automated entity linking platforms has enabled search techniques to adopt semantic-enabled methods such as sense disambiguation, intent determination, and instance identification within the search process. Researchers have already delved into the possibility of integrating semantic information into practical search engines, a paradigm known as *semantic full-text search*. However, the practical and efficient incorporation of semantic information within search indices is still an open challenge. In this thesis, we proposed two indexing approaches for building efficient and effective semantic full-text indices.

In the first approach, we remain faithful to the traditional form of building search indices where the index key of the index is guaranteed to be present in each of the indexed documents. As such, we will assume that the documents related to each of keyword, semantic entity, semantic type, do in fact explicitly contain this information. For this reason, the first proposed indexing mechanism is referred to *Explicit Semantic Full-text Index*. We propose various representation data structures and their effective integration strategies

for building the explicit semantic full-text index. Furthermore, we introduce algorithms for performing query processing tasks such as Boolean and rank union and intersection on the proposed indices.

In the second approach, we relax the traditional condition of search indices and allow documents associated with an index key to be *semantically similar* to the index key as opposed to explicitly including the key. We refer to this indexing strategy as the *Implicit Semantic Full-text Index*. We propose a mechanism to embedd keyword, semantic entity, semantic type information within a homogeneous representation space and hence be indexed in the same indexing data structure. Based on our experiments, we find that when neural embeddings are used to build inverted indices; hence, relaxing the requirement to explicitly observe the posting list key in the indexed document, (a) *retrieval efficiency* will increase compared to a standard inverted index, hence reducing the index size and query processing time, and at the same time (b) retrieval effectiveness retains competitive performance compared to the baseline in terms of retrieving a reasonable number of relevant documents from the indexed corpus.

Dedication

This thesis is dedicated to:

- Majid, my lovely husband, the greatest friend of mine, who was emotionally next to me in every step of this thesis and provided me with unconditional support.
- Zahra, my lovely daughter, who is indeed a treasure for us.
- And my parents who have been always a source of inspiration and encouragement for me.

Acknowledgements

I would like to express my gratitude to people who contributed to this thesis and my life as a PhD student.

My first and foremost acknowledgement goes to my advisor, Dr. Ali Ghorbani, who always offered wise advices and gave me the freedom to pursue my research interests. Thanks for always being there and for providing me insightful comments, incredible support and encouragement.

My special gratitude goes to Dr. Ebrahim Bagheri, as I was fortunate to work under his eminent guidance. Thanks for everything you have done for me. You taught me how to do research, from idea to presentation, and provided me with valuable comments and advices about every aspect of my research. Not only that, your encouragement and friendship was always a source of energy during these years; I summarize it by saying you were a great friend and teacher who was hard to find, difficult to leave, and impossible to forget.

I really appreciate the time and dedication my Advisory committee devoted to reviewing my thesis: Dr. Weichang Du and Dr. Bruce Spencer.

I am overwhelmed with gratitude by the support of all those who helped me

in different ways so that I may accomplish my goals. I would like to thank my friend Dr. Rasoul Shahsavarifar from whom I asked for help when I needed someone to discuss an idea with or I needed someone to share my feeling. I express my deepest thanks to my wonderful friends, most notably Andisheh Keikha, Dr. Mehdi Bashari and my colleagues in Laboratory for Systems, Software, and Semantics (LS3).

I would also like to thank my mother for their unconditional support for my education. Although I was away from her while doing my thesis, her help and support motivated me to go forward all through my studies. I would also want to thank my brother who was my childhood friend, and later my great teachers. Thanks for helping me to find my path and sending me your love and care over miles of distance.

Finally, I would like to thank my lovely husband, Majid, who was there for me all through this long journey and supported me through the ups and downs. His unconditional care and support helped me to be best of myself both in my research work and in my life.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Resource Description Framework	RDF
Information Retrieval	IR
Term Frequency	TF
Term FrequencyInverse Document Frequency	$TF - IDF$
Text Retrieval Conference	$TREC$
Mean Average Precision	MAP
Normalized Discounted Cumulative Gain	$nDCG@K$
Document identifier	$docId$
Term-At-A-Time	$TAAT$
Document-At-A-Time	$DAAT$
Score-At-A-Time	$SAAT$
Knowledge Base	KB
Freebase Annotations of ClueWeb Corpora	$FACC1$
Skip-gram	SG
Continuous bag-of-words	$CBOW$
Paragraph Vectors	PV
Distributed Memory Model	$PV - DM$
Word Movers Distance	WMD
Locality-sensitive hashing	LSH
Negative Sampling	NS
Hierarchical Softmax	HS
Query Process Time	QPT

Confidence Value	CV
Type identifier	$typeId$
Subtype identifier	$subtypeId$
Supertype identifier	$supertypeId$
set-vs-set	svs
out-of-vocabulary	OOV

Chapter 1

Introduction

In the complex dynamics of the World Wide Web, current search engines tend to retrieve relevant documents by counting occurrences of query terms in the document with variety of possible term frequency features (e.g. $BM25$). However, considering documents as bag of words, loses keyword ordering and interterm association at the time of indexing; and the lack of semantic description of keywords can lead to incorrect retrieval of ambiguous keywords [128]. In addition, keyword-based search has exhibited limitations particularly in dealing with more complex queries. Fernandez et al. [81] have discussed this problem by pointing to the limitations of keyword-based search engines when complex queries are encountered. For instance, in the two queries "books about recommender systems" versus "systems that recommend books" keyword-based search would not suffice in distinguishing between the two queries. Consequently, similar results are retrieved despite

the difference in the meaning between the two. While the first query requires a list of books about recommender systems, the second one requests information on a list of systems which recommend books. It is evident that additional information need to be taken into consideration to be able to effectively process such queries. For instance, the literature has already reported work on the semantic interpretation of search queries where important ontological concepts/entities within the query or the document collection are identified through *entity linking* [101, 30]. Such works address the very problem that was noticed in this example where the first query will be linked to the semantic concept representing recommender systems while the second one will not. To tackle these challenges associated with keyword-based search, the research community has explored incorporation of additional semantic information into the retrieval process, often referred to as *semantic search*. This type of search engine aims to improve search performance and accuracy by taking into account the intent and contextual meaning of keywords in the corpus and the query [210, 168]. The most relevant and state-of-the-art semantic search systems can be categorized into two groups, namely *Entity search* and *Semantic Web search engines* (e.g., Swoogle [71] and SemSearch [131]). In the former, entities, which are concepts represented in well-adopted knowledge bases such as DBpedia and Freebase, are indexed and searched instead of pure keywords [91] , while in the latter, semantic information such as Resource Description Framework (RDF) triples are identified and retrieved from Web documents or knowledge graphs that are shared on the

Web [194, 6].

1.1 Motivation

Despite improving the effectiveness of search results compared to a keyword-based search, semantic search has exhibited limitations particularly in dealing with a range of query types. Bast et al. [16] use the following query to show this problem: "astronaut walk on the moon". To answer this query, a semantic search engine would retrieve a list of documents containing the word "astronaut" or instances of astronaut (e.g., Neil Armstrong, Buzz Aldrin). The knowledge base index is also thoroughly searched for the word "moon". Problems arise when the knowledge base fails to provide information on the keyword "walk". In the above example, using semantic information instead of the keywords does not prove to be helpful in linking the "astronaut" entity with the "moon" entity and the integration can only happen if the keyword "walk" is considered to be a keyword as opposed to an entity. Therefore, the integration of the keyword information and semantic information becomes an essential component in processing search queries. This type of semantic search engine is referred to *semantic full-text search* [194, 18], in order to distinguish it from other types of semantic search. The integration of these two types of information determines the efficiency and effectiveness of the performance of the semantic full-text search engine due to the need for joining information from two different sets of indices, which can be costly [16, 18, 21].

The proposed integration approaches of keyword and semantic information can be divided into two categories. In the first category, well-known data structures used for full-text search (inverted indices) and semantic search (triple stores) are modified in an effort to incorporate semantic information [21] or add textual information to the semantic index. As far as this method is concerned, it is not considered a viable solution for semantic search for a number of reasons including the fact that it tends to be time consuming, and can also lose semantic information when dealing with complicated queries [16]. In the second category, the data structure of text indices and semantic indices are modified so as to make a connection between both indices. Early and efficient semantic full-text search engines including Mimir and Broccoli [194, 15, 136] are in this category. For instance, the work in [136] proposes to maintain separate indices for semantic entities as well as keywords that are observed in the document corpus.

According to Navarro et al. [120] and Bast et al. [23], the two factors impacting any Information Retrieval (IR) system, particularly semantic search engines, include *managing huge amounts of data* and providing *very precise results for queries* quickly. Both of these two factors are significantly influenced by the indexing data structure that is used for storing the information that will be later retrieved. This is even more so true for semantic full-text search engines where much more information needs to be stored and considered for retrieval. Adopting existing index structures that have already been built for keyword-based search can be a constructive approach. However,

existing data structures such as inverted indices cannot be directly used for semantic full-text search for several reasons including the following:

- The information to be stored in a semantic index is not confined to only textual information that have been traditionally stored in inverted indices. A semantic search index needs to be well equipped to efficiently and effectively retrieve and index additional types of information. Unlike keywords, whose occurrence and frequency are the most important information that need to be indexed, semantic entities and types carry additional information have additional information that need to be incorporated into the index and hence complicates the direct adoption of data structures such as inverted indices. For instance, entities identified within a document are often accompanied by a confidence value that show how confident the entity linking system was in identifying and linking this entity. Such information would need to also be stored in the index. These types of additional information are currently not included in traditional index data structures and need to be considered for semantic search.

- In addition, the amount of information that needs to be stored in the index for semantic information is more than that required for a keyword. For example, the surface form of an entity might be a phrase that consists of more than one keyword. Therefore, the index would not only require the starting position of the entity but also additional

information pertaining to the finishing position for that entity.

Based on the above points, the central research theme governing this thesis is to investigate how to build semantic full-text indices to decrease query process time and index size (improving efficiency) while increasing the number of semantically relevant results (effectiveness) of the given query. In our work, we view semantic full-text search as a process that considers entity information, type relationship and textual keyword information in tandem in order to answer an annotated query. This necessitates the development of a semantic full-text index that does not only store these three types of information but also integrates them so that complex queries can be answered by considering a wealth of information from the three distinct perspectives. Let us provide a concrete example to motivate our work by considering the following query: "books about recommender systems written by Dietmar Jannach". When processing this query from a semantic search perspective, we view three types of information in the query: i) Entities that can be linked to external knowledge bases and are automatically identifiable using entity linking systems. These would include entities such as Recommender Systems[1]. ii) Type information that would inform the search engine about the entities present in the query and the additional information available in the knowledge base. For instance, the fact that "Dietmar Jannach" is a Person or that he is a Scientist from Germany. iii) keyword information that include the terms that have been mentioned in the query but cannot be related to

[1] `https://en.wikipedia.org/wiki/Recommender_system`

any entities or types in the knowledge base, e.g., written.

In order to be able to index these three types of information, we propose two approaches for building semantic full-text indices based on semantic full-text search perspectives and neural embedding perspectives.

- From a semantic full-text search perspective, we investigate how the required underlying indexing data structures for semantic full-text search engines can be adopted and represented efficiently and effectively. The proposed semantic full-text index, maintains three types of indices including (i) textual indices that store keyword-document associations; (ii) entity indices, which consist of semantic entity-document relationships; and (iii) semantic entity type indices, which store entity type hierarchies. The integration of these three types of indices provides the infrastructure to search documents not only based on document-keyword relevance but also based on keyword semantics.

- From a neural embedding perspective, we propose to embed keywords and semantic information into the same embedding space. This means these heterogeneous information are turned into homogeneous information by using neural embeddings. Neural embeddings attempt to learn unique and dense yet accurate representations of objects based on the contexts they appear in. By embedding different information types within the same space, we can use a single inverted index to store such information. This inverted index is built based on the information in

the embedding space with respect to semantic similarity between documents, keywords and entities. Therefore, each posting list consists of the most semantically related documents to the index key, unlike traditional posting lists which consist of all those documents that explicitly contain the index key.

1.2 Contributions

A semantic search engine retrieves documents on the basis of the similarity of entities and keywords that are observed within the document and query spaces [194]. In order to be able to measure similarity, a combination of entities, keywords and entity types need to be properly indexed. The achieve our goal, we need to identify the most suitable indexing method that allow us to efficiently and effectively store and retrieve these three types of information. As mentioned earlier, to achieve this goal, we propose two strategies for building a semantic full-text index, which retrieves semantically related documents, based on two perspectives: semantic full-text search and neural embeddings.

In the former perspective,we explore the adoption of various data structures that have already been adopted in the literature for building different types of indices. The prevalent approaches that deal with designing the data structure for semantic full-text indices are generally divided into three categories: The work in the first category changes the structure of the posting

list [136, 206, 44] while the second uses more than one index for indexing semantic information and then combines the results at query process time [194, 136, 43]. In the last category, the structure of the inverted index is modified to provide the required functionality [15, 21]. In our work, the first and second approaches will be combined to present an efficient data structure for building the required semantic index. Furthermore, this approach needs to cover an integration of keywords, entities and types. There are different ways to combine the required information for such an index to answer semantic queries efficiently and effectively. For instance, ESTER [21] adds semantic information to a context as artificial words but in Broccoli [15] two indices are considered: one for indexing relations (ontology) and the other one for keywords. ESTER defines the *occurs-with* relation between entities and keywords that occurred in the same context, and this type of relations is added to the relation index, which is used during the query process to show association between keywords and entities. The data structure of Broccoli index is HYB [35]. Furthermore, the Entity Engine [136] uses two types of posting lists to integrate semantic entity information and keywords. The co-occurrence between keywords and entities is defined based on their position in the documents, which allows the Entity Engine to implicitly relate these two posting lists. To this end, we explore three main data structures, namely *HashMaps*, *Treaps*, and, *Wavelet Trees* as our indices. These three data structures are espoused for our purpose due to the following reasons:

1. The modification of inverted index data structure, as often imple-

mented in the form of a HashMap, has provided reasonable results in earlier works for keyword-based search tasks[120, 117].

2. Treaps have the ability to process ranked queries faster than the standard inverted index along with using less space [120].

3. Wavelet trees support positional index proximity search and process ranked intersection queries faster than Block-Max index [72], which is a variation of the inverted index structure[120].

We refer to a semantic full-text index which is built based on this approach as an *explicit semantic full-text index*. The following steps are the main contributions of building the explicit semantic full-text index:

- We systematically explore the possibility of building a semantic full-text index by using one or a combination of these three data structures. The adopted data structure or combination thereof would need to support the indexing of three types of information, keywords, entities and types; we propose that using three sub-indices, namely Keyword, Entity, and, Type Indices, provide efficient and effective search and fast integration of information across the three types of information.

- We study possible integration approaches between the adopted indexing data structures to process queries by integrating information of Keyword, Entity, and, Type indices in an efficient and effective way.

It is worth noting that the central reason behind our decision for not adopting well-known index data structures, such as forward indexing[66], signature file [76, 77], and, suffix array [144], is due to their limitations, e.g., their support for only Boolean queries and slower query time compared to basic inverted indices, just to name a few [76, 224].

To build a semantic full-text index based on neural embeddings, we explore the possibility of folding keyword, entity, and, type indices into a single index that incorporates keyword, entity, and, type information, collectively. Our proposed idea is to turn these three types of heterogeneous information into homogeneous information by embedding them in the same embedding space based on neural embedding approaches; hence, the created homogeneous information can be indexed with a single inverted index. In other words, we integrate these three types of indices to one index to prevent increasing query processing time of a semantic full-text index. We refer to a semantic full-text index which is built based on this approach as an *implicit semantic full-text index*, since we use neural embeddings for building this index. The contributions of building the implicit semantic full-text index are, succinctly, as follows:

- We systematically show how keywords, semantic entities, entity types and the documents that contain these contents can be embedded within the same space and hence become homogeneous to be indexed within a single inverted index.

- Our proposed work explores how inverted index built based on the information in the embedding space is constructed according to the concept of semantic similarity between documents and keywords. Therefore, unlike traditional inverted indices, the documents in each posting list are not guaranteed to explicitly contain the index key but are rather guaranteed to be, semantically-speaking, the nearest neighbors of the index key in the embedding space.

1.3 Thesis Overview

The rest of the paper is organized as follows:

Chapter 2 reviews the fundamental concepts of information retrieval and the state-of-the-art on semantic search. Then we present an overview of neural methods for information retrieval. We discuss practical and well know word embedding algorithms and document similarity measures.

Chapter 3 provides the details of our proposed approaches for building semantic full-text index. First, we explain indexing and document retrieval methods for the explicit semantic full-text index. Then, we describe how the implicit semantic full-text index is built based on the joint embedding of text and semantic information.

In Chapter 4, we introduce our evaluation methodology, evaluation corpora and report on our obtained experimental results. We also offer discussion on the efficiency and effectiveness of the proposed approaches.

Chapter 5 will include concluding remarks with recommendations and suggestions for further research.

Chapter 2

Background and Related Work

This chapter provides the required background to set the stage for the rest of the chapters in this thesis. First, the fundamental steps and techniques used in information retrieval are described in Section 2.1. Then, it provides an overview over indexing methods such as index data structures, compression and query process strategies related to the posting list structures. This is followed by a review of existing approaches for building semantic index in the literature. Finally, it surveys neural network methods in the information retrieval community for improving retrieval performance. Also, we provide a summary of word embedding and document embedding algorithms.

2.1 Information Retrieval

Information Retrieval (IR) is a broad research area that has been defined as [177] a field concerned with the structure, analysis, organization, storage, searching, and retrieval of information. The most practical application of information retrieval is computer-based search. The main focus of information retrieval has been on identifying and returning sorted documents based on their relevance score to queries. Relevance is a fundamental but loose concept in information retrieval; Croft et al. [56] defined a relevant document to contain the information that a person was looking for when she submitted a query to a search engine. IR systems are composed of some primary components such as document processing, query processing and retrieval of relevant documents, which is presented in Figure 2.1. Another fundamental feature of IR is evaluation, which is described in Section 2.1.4.

2.1.1 Document Processing

Document processing is an important component of an IR system since it is in charge of collecting documents and changing them in a way to support efficient search and lookup. Generally, document processing consists of three steps: (i) text acquisition, (ii) text transformation, and (iii) indexing/index creation[56]. These steps are described in the following.

Text acquisition involves finding new documents and updating existing ones; this process is called web crawling in web search.

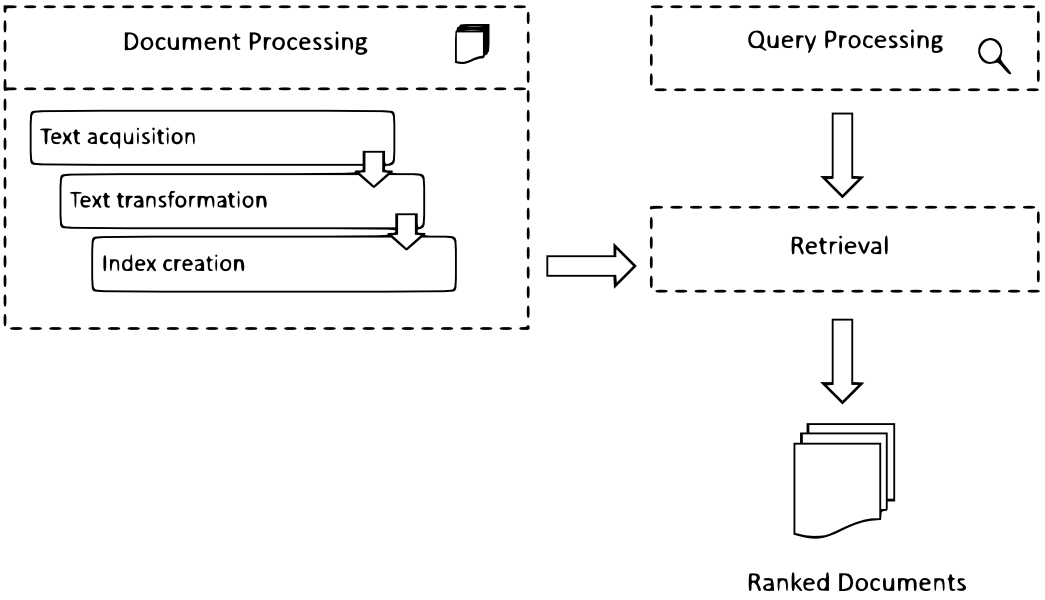

Figure 2.1: High level building blocks of IR system.

Text transformation. converts documents into basic indexing units. To achieve this goal the following linguistic transformations (but not exclusively) are often undertaken:

- Tokenization converts the input text into a sequence of tokens.
- Stop words elimination removes common words that have very little effect on identifying relevant documents of queries (e.g. the, a, and).
- Stemming derives the stem of the words; e.g., "laugh" for "laughter", "laughing" and "laughed". This process may be improve the retrieval effectiveness.

Indexing will maintain index keywords and other information about the keywords (e.g. term frequency, position) and documents (e.g. number of

keywords, title) in an efficient data structure. The index structure should be a time and space efficient structure that enables storage and update of documents and keywords, as well as looking up information about them. The index structures and strategies for improving efficiency will be discussed in Section 2.2.

2.1.2 Query Processing

Query processing discovers users information needs. The simplest query processing steps is the same steps that are performed for document processing such as: tokenization, stop words elimination, and stemming. Spell checking and query expansion are other query processing steps which can impact retrieval performance. For instance, around 10-15% of Web search queries contain spelling errors [59] which can be easily captured by using query logs, document collections, and trusted dictionaries. Also, in many cases, queries do not represents users needed information such as a concept with different keywords (e.g. mobile and cell phone) or a keyword with different concepts (e.g. "Python" can be a snack or a program language). Query expansion approaches solve this type of issues based on local and global query expansion methods. In the former, the words that are related to the topic of the query are added to the list of query terms, and in the latter, each query term is expanded with related words from a thesaurus. There are different strategies for processing a query which can impact the efficiency and effectiveness of an IR system.

2.1.3 Retrieval

Retrieval is concerned with ranking documents with respect to a query based on a relevance model. In this section, we just summarize three standard and popular retrieval models among many existing models: *Vector Space Model*, *BM25*, and *Language Models*.

Vector Space Model

The Vector Space Model [178] was proposed based on Luhn's similarity criterion [141], which recommends a statistical approach for searching information. In this model, queries and documents are defined as n-dimensional vectors in a common vector space. For instance, a document d and a query q for a collection of n keywords are represented as:

$$\vec{d} = (d_1, d_2, d_n),$$

$$\vec{q} = (q_1, q_2, q_n)$$

where d_i and q_i are the weights of ith keyword for the document and the query, respectively. Among various proposed keyword-weighting schemes; *term frequencyinverse document frequency* $(TF - IDF)$ is one of the most popular weighting factor in information retrieval. It shows how significant a word is to a document in a collection of documents. Search engines often use $(TF - IDF)$ as a tool for measuring the relevance of documents to queries. The $TF - IDF$ value for a keyword t and document d is computed as:

$$(TF - IDF)_{(t,d)} = TF_{t,d}.IDF_t \tag{2.1}$$

The $TF_{t,d}$ term in Eqaution 2.1 represents the frequency of keyword t in document d, and is usually computed as:

$$TF_{t,d} = \frac{freq(t,d)}{\sum_{i=1}^{n} freq(t_i,d)} \tag{2.2}$$

The IDF_t component represents the discriminating power of a keyword in the whole collection. It is typically defined as:

$$IDF_t = \log \frac{N}{df_t} \tag{2.3}$$

Here N is the number of documents in the document collection and df_t is number of documents that contain keyword t (also referred to as document frequency). This value will be high for a rare keyword, which proposes that a rare keyword carries a lot of information.

After creating document and query vectors, the similarity of each document to the query can be computed using vector similarity measures, for instance, with a *cosine similarity* function.

$$\cos(\vec{d},\vec{q}) = \frac{\vec{d}\cdot\vec{q}}{\|\vec{d}\|\|\vec{q}\|} = \frac{\sum_{i=1}^{n} d_i q_i}{\sqrt{\sum_{i=1}^{n} d_i^2}\sqrt{\sum_{i=1}^{n} q_i^2}} \tag{2.4}$$

Recently more advanced vector representation models based on neural embeddings have shown to improve retrieval effectiveness.

BM25

BM25 is defined based on the *Probability Ranking Principle* which was proposed by Robertson [172]. In this method, documents are ranked based on the probability of relevance of a document to a query without considering any inter-relationship between the query terms. This model is an effective and popular retrieval model which respects the binary independence model. BM25 formulation for a query Q which consist of terms: q_1, q_2, q_n and document d with length $|d|$ is:

$$\text{score}(d, Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i, d) \cdot (k_1 + 1)}{f(q_i, d) + k_1 \cdot \left(1 - b + b \cdot \frac{|d|}{\text{avgdl}}\right)} \tag{2.5}$$

Where the *avgl* is the average length of documents in the collection and two free parameters, k_1 and b control keyword saturation and document length normalization components, respectively. These values are usually chosen as $k_1 \in [1.2, 2.0]$ and $b = 0.75$ which were proposed by Robertson et al. [171] for statistical models for IR. Equation 2.5 involves *IDF't* which is defined based on the following formula for BM25.

$$\text{IDF}(q_i) = \log \frac{N - df_t + 0.5}{df_t + 0.5} \tag{2.6}$$

Language Models

The language modeling approach considers a document to be relevant, if the query could be generated from the document. This would happen if the query terms occur in the document. " A language model is a function that puts a probability measure over strings drawn from some vocabulary" [183].

One simple kind of language model over an alphabet σ is:

$$\sum_{x \in \sigma} P(x) = 1 \tag{2.7}$$

Probabilities over string $S = (s_1, s_2, s_3)$ can be defined by decomposing the probability of S into the probability of each keyword conditioned on earlier keywords. So, $P(S)$ is:

$$P(s_1, s_2, s_3) = P(s_1)P(s_2|s_1)P(s_3|s_1 s_2) \tag{2.8}$$

If we assume keywords are independent, then $P(S)$ is:

$$P(s_1, s_2, s_3) = P(s_1)P(s_2)P(s_3) \tag{2.9}$$

This model is called *unigram language model* which is also known as *bag of words* model in IR. In most of the IR tasks, the probability of a keyword does not depend on surrounding keywords, so, unigram language model is often sufficient for building probabilities over context.

To design language models in IR, we assume that the document d is only a sample of text and is seen as a fine-grained topic. Then a language model from this sample is estimated to calculate the probability of observing any sequence of keywords. Moreover, documents are ranked based on their probability of creating the query.

In this section, we describe the *query likelihood model* [166] which is a basic

and popular language modeling approach in IR. The basic idea of query likelihood is defined based on the probability of relevance of document d to the query q, i.e., $P(d|q)$. But since queries are of much shorter than documents and can not be good representatives for vocabulary words, the Bayes'rule is used for estimating the probability as follows:

$$P(d|q) = \frac{P(d|q)P(d)}{P(q)} = P(d|q) \tag{2.10}$$

In this equation, $P(q)$ can be ignored since it is the same for all documents. The prior probability of d, i.e., $P(d)$ can be ignored, as it is often treated as uniform across all documents. Hence, we simply ranked documents base on $P(q|d)$.

2.1.4 Evaluation

One of the fundamental aspects of information retrieval research is systematic evaluation of performance. ***Efficiency*** and ***effectiveness*** are two main evaluation aspects for any practical IR system. Consequently, evaluating efficiency and effectiveness of an IR system is essential for any real-word IR system. Effectiveness represents the degree of user satisfaction based on the quality of the retrieved results. Efficiency, on the other hand, illustrates to what extent an IR system is able to perform optimally in regards to speed and memory usage [56].

The standard evaluation method for the effectiveness of an IR system de-

pends on the notion of relevant and non-relevant documents [145]. General approach for evaluating effectiveness of IR systems is comparing their experimental results against a standard test collection. Test collections often consist of several queries and their corresponding relevance labels which are usually created by human annotators; which is called *relevance judgement results*. The most well-known initiative that provides test collections for variety of IR tasks is *Text Retrieval Conference* (TREC).

Beside test collections, several evaluation measures exist that quantify the performance of retrieval systems. Two main categories of evaluation measures are unranked-based measures (e.g. precision and recall) and rank-based measures (e.g. P@K and MAP)[145].

In the first category, *recall* and *precision* are the most basic measures for evaluating different IR tasks. Recall is the fraction of relevant items that are retrieved, and precision is the fraction of retrieved items that are relevant. F-measure combines precision (P) and recall (R) by taking the harmonic average of these two measures:

$$F_1 = \left(\frac{\mathrm{R}^{-1} + \mathrm{P}^{-1}}{2} \right)^{-1} = \frac{2 \cdot \mathrm{P} \cdot \mathrm{R}}{\mathrm{P} + \mathrm{R}} \tag{2.11}$$

These evaluation measures are commonly used for IR-related classification tasks (e.g. named entity recognition and entity linking). However, they are rarely used for ranking problems, since they do not consider the order of the retrieval results.

In the second category, the quality of the results depends on their positions in a ranked list. For instance, $P@K$ and $R@K$ are extensions of precision and recall, respectively. They compute these two values at a given rank position K. Combining precision at different levels of recall is called *Average Precision* (AP); averaging of AP over all queries defines other evaluation measure known as *Mean Average Precision* (MAP). This measurement is used particularly when relevance judgements are binary. Non-binary relevance judgements are evaluated with *Normalized Discounted Cumulative Gain* ($nDCG@K$) [111]. The main idea of this method is based on how much information is gained when a user views a document. This method calculates position-based penalty between results and relevance judgements since both lists are ordered based on relevancy.

Note that the real evaluation of an IR system depends on the concept of user utility. The main utility for this type of system is user satisfaction which needs to be quantified based on the relevance results, speed, space and user interface of an IR system [145]. For instance, studies show how improvements in formal retrieval effectiveness do not always mean a better system for users [104, 105, 202, 203]. But, user interfaces for IR and human factors (e.g. usability testing) are outside the scope of this thesis. More information can be found about these topics in [187, 13, 122]. Accordingly, the evaluation metrics in this thesis are defined based on effectiveness (relevance) and efficiency (speed and space).

2.2 Indexing

To search over large collections of textual data, we need efficient indices which can improve effectiveness of retrieval, speed and memory usage. Since, the overall performance of an index is dependent on the performance of its data structure [103, 120, 20].

In this section we present three popular and efficient data structures for creating indices. We then provide a summary of the main compression methods for improving index efficiency. At the end, we review three main strategies for processing queries in regard to indexing data structures.

2.2.1 Indexing Structures

Indexing plays a very important role in IR system performance specially for semantic search since the size of the input data and statistics needed for search can quickly become overwhelming [194, 17, 19]. As such, there is need to find efficient data structures that have the capability to retrieve results efficiently and effectively.

In this section, we introduce three main data structures: inverted index, treap and wavelet tree that have been widely used for building indices. We compare these data structures and evaluate their appropriateness to serve as a data structure for indexing.

2.2.1.1 Inverted Indices

The inverted index is an efficient index data structure [23, 117] that allows fast search and plays a pivotal role in IR [121, 117]. The Inverted index structure can function as a map where each key in the map corresponds to a keyword in the corpus and the corresponding value is a list of *postings*, conveniently called *posting lists*. Every posting in a posting list references a specific document where the keyword has appeared in, and stores information such as the document identifier ($docId$), the frequency of the keyword in that document (TF), the exact position of the keyword in the document, and the document length, among others. The distinct set of keywords present in the corpus being indexed is often referred to as the vocabulary. The Inverted index will have one entry for each item in the *vocabulary*. Figure 2.2 provides an overview of the structure of a positional inverted index whose postings contain $docId$ and TF and the keyword position in the document. For example, the keyword *africa* appears in three documents identified by 3, 108 and 205 and is located in positions 1, 99 and 467 of the document that is identified by the identifier 205. Researchers have shown that HashMaps are an efficient method for implementing an Inverted Index [103] and therefore we adopt such implementation in our work.

There are currently two approaches for ordering postings of a posting list in an inverted index depending on whether ranked or Boolean retrieval needs to be supported. The purpose of ranked retrieval is to retrieve documents that are believed to be most *relevant* to a query. In this context, relevancy is

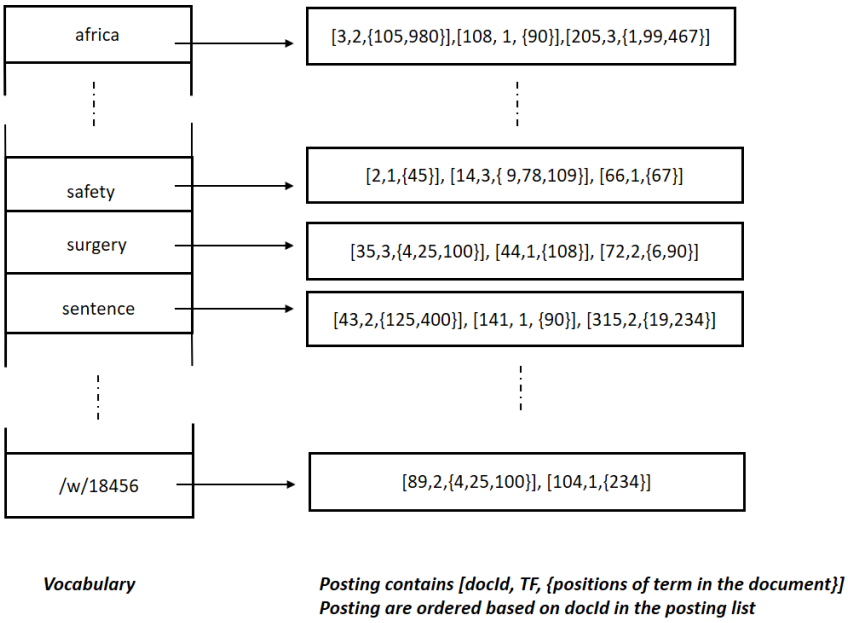

Figure 2.2: Structure of a simple inverted index. An posting list contains $docId$ in ascending order.

defined in accordance to a particular set of criterion (e.g. TF-IDF, BM25). Therefore, for ranked retrieval, postings in a posting list are sorted in descending order based on their relevancy. On the other hand, Boolean retrieval, also known as exact match querying, intends to find all the documents where the query terms appear in regardless of their relevancy measures. In this case, the postings in the posting lists are ranked based on increasing $docIds$. In Figure 2.2 postings are sorted based on ascending order of $docIds$.

2.2.1.2 Wavelet Trees

The wavelet tree data structure was proposed in [94] as a data structure to represent compressed suffix arrays [94, 144] and has since been adopted as an important component of the FM-index family [83]. A Wavelet Tree represents a sequence $S[1,n] = s_1, s_2, \cdots, s_n$ over an alphabet $\Sigma = [1, \cdots, \sigma]$ where $s_i \in \Sigma$. It represents at most $n \log \sigma + O(n)$ bits of space which is not larger than the needed space to represent S in plain form ($n\lceil log\sigma \rceil$ bits) [63], and can be constructed in $O(n \log q)$ time, where $q \leq min(n, \sigma)$ [89]. Wavelet tree is a binary balanced tree with σ leaves over Σ. It is created by continuously partitioning Σ into two subsets until each subset is just a symbol of Σ. The root node of a Wavelet Tree is $S[1,n]$ which is represented with a bitmap $B_{root}[1,n]$ in a way that, if $S[i] \leq \frac{(1+\sigma)}{2}$ then $B_{root}[i] = 0$, else $B_{root}[i] = 1$. The left child of the root is a Wavelet Tree for all $S[i]$ whose $B_{root}[i] = 0$ over the alphabet $[1, \cdots, \lfloor \frac{(1+\sigma)}{2} \rfloor]$ and the right child of the root is a Wavelet Tree for all $S[i]$ whose $B_{root}[i] = 1$ over the alphabet $[1 + \lfloor \frac{(1+\sigma)}{2} \rfloor, \cdots, \sigma]$. Figure 2.3 presents a wavelet tree for the sequence $S =$ '$2, 30, 59, 65, 15, 44, 15, 99, 17, 26, 2, 44$' where $\Sigma =$'$2, 15, 17, 26, 30, 44, 59, 65, 99$', so $n = 12$ and $\sigma = 9$. The Wavelet Tree returns any sequence element $S[i]$ in $O(\log \sigma)$, and provides answers to rank and select queries in $O(\log \sigma)$ time. These queries are defined as:

$rank_x(S, i)$ = number of occurrences of symbol x in $S[1, i]$
$select_x(S, i)$ = position of the i^{th} occurrence of symbol x in S

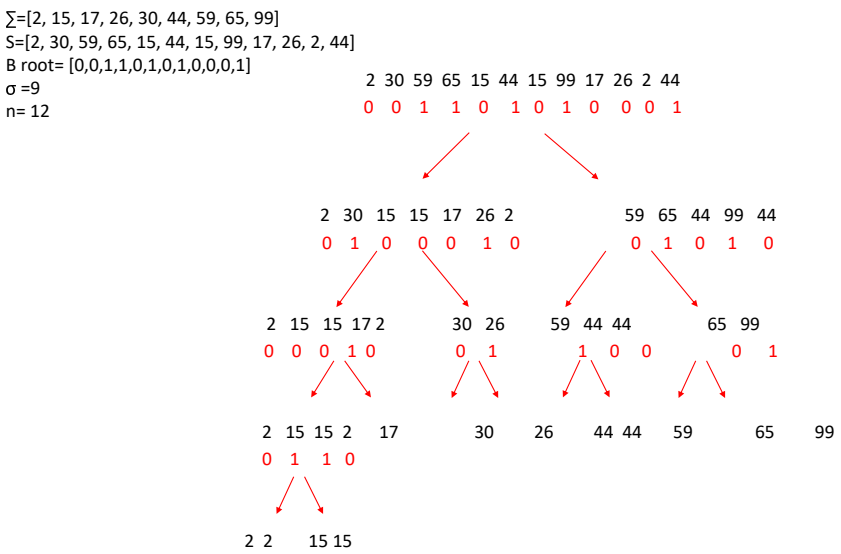

Figure 2.3: A wavelet tree on S = '2, 30, 59, 65, 15, 44, 15, 99, 17, 26, 2, 44'. The tree stores only the topology and the bitmaps (B[i]).

In order to answer $S[i]$, one needs to start from the root by examining $B_{root}[i]$. If it is 0, $S[i]$ will be the left side, otherwise it will be on the right side. In the first case, this process is continued recursively on the left child; otherwise, it is continued on the right child until we arrive at a leaf node. The label of this leaf will be $S[i]$. Note that the value of i is changed on the left (or right) child and therefore, the new position of i needs to be determined. In the case of the left child, the number of 0s in B_{root} up to position i is the new position of i in the left child. For the right child, the new position for i is the number of 1s in B_{root} up to position i.

Furthermore, the $select_x(S, i)$ query tracks a position at a leaf whose label

is x to find out where it is on the root bitmap. Therefore, it is the inverse process of the above approach. We start at a given leaf at position i. If the leaf is the left child of its parent v, the i^{th} occurrence of a 0 in its bitmap B_v is the new position i at v. If the leaf is the right child, then the new position i is the position of the i^{th} occurrence of a 1 in B_v. This procedure is continued from v until we reach the root, where we discover the final position.

The approach for answering $rank_x(S, i)$ is similar to $S[i]$. The only difference is that the path is chosen according to the bits of x instead of looking at $B_{root}[i]$. We go to the left child if x is in the first half of the alphabet, otherwise we go to the right child. When a leaf is reached, the value for i is the answer.

Wavelet Trees are versatile data structures, which can be represented in three different ways. The more basic way is through *sequence of values*, The *sequence of values represents* the values s_i of a sequence $S = s_1, s_2, \cdots, s_n$ with Wavelet Tree on S. The main operations supported in this approach are access ($S[i]$), rank, and select. The second less obvious way to represent the Wavelet Tree is *ordering* which involves the stable ordering of s_i in S. Here, the smallest symbol of S is placed in the first leaf of the Wavelet Tree and all occurrences of this symbol are ordered based on their original position in that leaf. In this respect, tracking a position downwards in the Wavelet Tree is the deciding factor on where it will end up after being sorted. The same pattern is visible when tracking a position upwards in the Wavelet Tree, which indicates where each symbol is positioned in the sequence. The lesser

general structure is referred to as a *grid of points*, which uses Wavelet Tree as a representation of $n \times n$ grid with n points in a way that two points do not share the same row or column [158].

2.2.1.3 Treaps

A Treap is a combination of Binary Search Tree and the Heap where each node has a key and an attribute, which is randomly assigned to a key (priority). Key values are ordered in the Treap in a way to satisfy the binary search tree property. The priority value of each node is greater or equal to the priority of its children to support the Heap order, which is determined by the structure of the tree. Therefore, the priority value of the root is the maximum-priority value in the a treap. Figure 2.4 displays the treap representation for the given posting list, which consists of $docIds$ and TFs. A key within a treap can be searched for just like a binary search tree, and at the same time, it can be used as a binary heap. Treaps have shown to use less space and perform fast ranked union and ranked intersection for Keyword indices [121].

2.2.1.4 Comparative Analysis of the Data Structures

Table 2.1 illustrates a set of features that are important for choosing a data structure appropriate for indexing search data. These features are categorized into two main groups: memory usage and index process time for index construction and search. The first three rows in Table 2.1 present the fea-

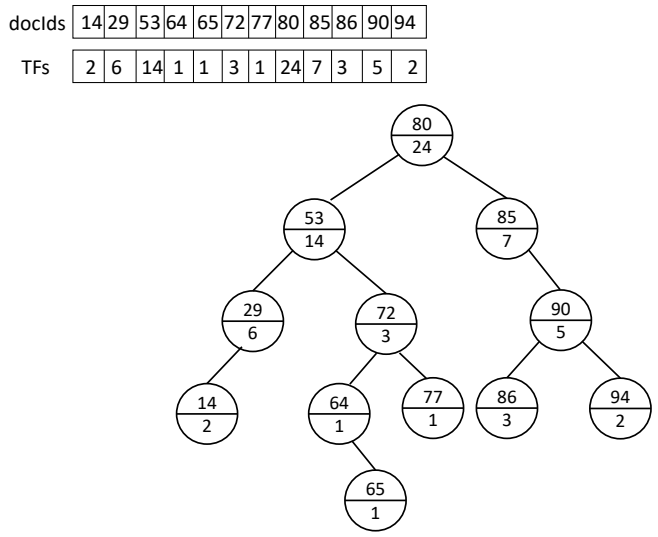

Figure 2.4: An example of treap representation. Key values (upper value inside nodes) are sorted inorder and priority values (lower value inside the nodes) are sorted top to bottom.

tures that are required for analyzing memory usage and the remaining rows address query process time.

The dual sorted inverted list feature in the first row of the table shows whether the data structure supports retrieving search results for both ranked and Boolean retrievals simultaneously compared to cases when two types of sorted inverted indices are needed, one for each type of retrieval. Wavelet Trees and Treaps can carry out the functions of the dual sorted inverted list feature while Inverted Indices do not possess the ability to support the aforementioned feature. From a memory usage perspective, these three data structures can be ordered as (1) Treap (2) Wavelet tree, and, (3) inverted index, respectively [121].

The second set of features is used for analyzing the time required to construct the index and to search. In light of the fact that one of the influential factors on query process time is query length, there is a direct relation between increasing the length of the query and query process time. Among the analyzed indices, Wavelet Tree has exhibited less sensitivity to query length [117].

The data structure employed for constructing the index has a direct relation to query process time and memory usage. Based on our knowledge, the inverted index is the only type of data structure used for indexing semantic information. However, within the keyword domain many different types of indexes have been considered. Forward Index [66], Signature File [224, 209], and Suffix Trees are data structures that are considered as alternatives to the inverted index. These data structures are not very suitable for use in indexing semantic information. For example, Signature Files and Bitmaps offer faster query processing time compared to Inverted Indices under certain circumstances but they show worse performance compared to the Inverted Index as they use an excessive amount of space to provide the same set of functionality [224]. Furthermore, applications that use Inverted Indices generally have better performance than Signature Files and Bitmaps, depending on the index size and query process time [209]. In addition, Zobel et al. have demonstrated that Signature Files are more complicated than Inverted Indices to process. For instance, Signature Files are much larger, slower and building them is more expensive due to a variety of parameters that need

	Wavelet Tree (WT)	Inverted Index (INV)	Treap
Dual Sorted Inverted List	Yes, but it is slower than INV [117].	No, inverted lists are sorted based on term frequency or *docId* [118]	Yes, Treap simultaneously sorts postings based on term frequency (priority) and *docId* (node value) [120].
In Memory Index	More memory demanding compared to INV [120].	It is an efficient data structure for in-memory index and indexing on hard disk [103].	It is just implemented as in-memory index [120].
Memory Usage	Uses same space of one compress INV [117].	Uses more space compared to Treap [120].	Uses 13% less than the Wavelet Tree and 10% below of the size of the corpus [120].
Query Process Time	Boolean intersection is faster than Block-Max (a type of INV)[120]. Slower than INV but uses less space during processing query [9].	WT process time for phrase s search is less than INV [63].	It is faster (up to twofold) for up to $k = 20$ on ranked intersections and up to $k = 100$ on ranked unions compared to INV and WT [120].
Query Process Time is Dependent to Query Length	In intersection query process time improves for long queries but is increased for Union [120].	Yes, it is increased along with an increase in query length [120].	Yes, it increases for query length > 4. It also increases sharply when k is increased for top-k retrieval [120].

Table 2.1: A comparison of the properties of Wavelet Trees, Inverted Indices, and, Treaps

to be determined, including analysis of the data and tuning for anticipated queries [224]. The need for too much additional space can be considered to be a negative aspect especially in the case of semantic information as there are much more information to be stored compared to keyword-based information.

Several researchers have proposed to change the structure of the posting list within the Inverted Index to decrease both processing time and memory usage. Konow et al. [121] followed through with this method with the intention of improving the efficiency of the Inverted Index. They proposed a new way of representing the Inverted Index based on the Treap data structure in order to improve the efficiency (query process time and memory usage) of the Inverted Index data structure. Treap was used to represent $docId$ and TF ordering of a posting list so as to efficiently carry out ranked intersection and union algorithms. The study by Konow et al. also revealed that their particular index uses 18% less space than the state-of-the-art Inverted Index in addition to decreasing the query process time. The same index structure has been used in our work for the first time to index semantic information. A thorough review of the literature reveals that Wavelet Trees and Treaps have not been considered previously for indexing semantic information and Inverted Indices have been the primary data structure in this domain. We systematically evaluate Inverted Indices, Wavelet Trees and Treaps for the purpose of indexing semantic information.

2.2.2 Compression

Compression of the posting lists of an index is necessary for efficient retrieval [208, 38]. The central goal of compression is to encode common data elements with short codes and uncommon data elements with longer codes. As mentioned in Section 2.2.1, posting lists are fundamentally lists of numbers, and some of those numbers are more frequent than others. If the frequent numbers are encoded with short codes and the infrequent numbers with longer codes, the index can be stored in smaller space and reduce the time required to evaluate the query since reading compressed information from the index is faster than uncompressed information. Therefore, we can use compression to further improve efficiency of indices. However, choosing the compression technique that can store data in the smallest amount of space is not sufficient for compressing an index. Because, query process needs to have posting list information in decompressed form. Therefore, efficient compression techniques reduce index size and decompression time.

The goal of all compression techniques that are presented in this section, is using as little space as possible for storing small numbers in posting lists (such as keyword frequency keyword positions). Therefore, all the coding techniques are considered in this section assume that small numbers are more likely to occur than large ones. The assumptions are that many words occur between one to three times in a document and only a small number of words occur more than 10 times and document identifiers in posting lists do not have any entropy that can be used for compression. However, postings are

typically ordered by document identifiers in a posting list which allows us to encode them by the differences between adjacent document identifiers. For instance, if document identifiers of a posting list are:

$5, 9, 23, 35, 40, 51$

They can be encoded as:

$5, 4, 14, 12, 5, 11$

This encoded list starts with 5 which shows the first document identifier is 5. The next entry is 4 which specifies the second document identifier is 4 more than the first document identifier ($5 + 4 = 9$). This type of encoding is called *delta encoding* which does not actually save any space on its own but creates ordered lists of small numbers. This process would be considered as a transformer of an ordered list of numbers to a list of small numbers which is practical for applying compression techniques. Note that if the difference between adjacent document identifiers is large, the delta encoding cannot be useful for compression as well as for a posting list with small difference between document identifiers. This means, posting lists for frequent keywords compress better than infrequent keywords.

As discussed later in Section 3.1.2 for ranked unions, posting lists are sorted by decreasing weight so delta encoding can be used for keyword weight; however, compression of document identifiers based on this encoding is not possible. The advantage of sorting based on weight is that the length of the encoding is not long since there are many equal weights in a posting list. Therefore, the corresponding document identifiers can be sorted increasingly

to encode them based on delta encoding [12, 222].

Konow et al. [119] introduced a new compressed representation for posting lists to makes ranked intersections and (exact) ranked unions without producing the full Boolean result first. They use treaps for their representation since it represents a left-to-right and a top-to-bottom ordering at the same time. They ordered document identifiers based on the left-to-right order to support fast Boolean operations and keyword frequency are ordered in the top-to-bottom ordering to provide thresholding of results for the intersection process simultaneously.

Consequently, we need to use little space for storing small numbers in posting lists by finding practical coding methods. In the next section we review some popular coding methods for storing statistics in posting lists. We describe some practical *bit-aligned* encoding where the codes can be broken after any bit position. We then discuss *byte-aligned* encoding where size of each code word is a byte. The last described encoding method divides posting lists into blocks and then encodes each block separately based on delta encoding.

2.2.2.1 Bit-Aligned Encoding

-Unary Code

One of the simplest codes is the unary code which encodes numbers by using a single symbol, for example, this code 11110 shows number 4 in unary form, because it consist of 4 1s followed by a 0. To have unambiguous code, the 0s are placed at the end of the code. This code is very efficient for small

numbers but it would be very expensive for large numbers.

-Elias-γ Code

The Elias-γ code uses unary and binary codes. It represents the number k by computing the two following quantities:

$k_1 = \lfloor \log_2 k \rfloor$

$k_2 = k - 2^{\lfloor \log_2 k \rfloor}$

The k_1 (unary code) indicates how many bits are required to code k and the k_2 which follows k_1. Figure 2.5 shows some examples of Elias-γ coding.

Number (k)	k_1	k_2	Code
1	0	0	0
2	1	0	1 0 0
15	3	7	1 1 1 0 1 1 1
225	7	127	1 1 1 1 1 1 1 01 1 1 1 1 1 1

Figure 2.5: Examples of Elias-γ code.

Elias-γ code requires $2\lfloor \log_2 k \rfloor + 1$ bits which consists of $\lfloor \log_2 k \rfloor + 1$ bits for k_1 and $\lfloor \log_2 k \rfloor$ bits for k_2. This coding improves unary code but it is not practical for large numbers because it requires twice as many bits as k can be represented with binary digits ($\log_2 k$).

-Elias-δ codes

Elias-δ code solves the problem of the Elias-γ codes by changing the encoding method of the k_2. It splits k_1 into k_{1a} and k_{1b} and then encodes k_{1a} in unary, k_{1b} in binary, and k_2 in binary.

$k_{1a} = \lfloor \log_2(k_1 + 1) \rfloor$

$k_{1b} = (k_1 + 1) - 2^{\lfloor \log_2(k_1+1) \rfloor}$

This code is unambiguous because k_{1a} indicates the length of k_{1b} and length of k_2 is indicated with k_{1b}. The encoding of 1,2 and, 15 in Elias-δ is presented in Figure 2.6.

Number (k)	k_1	k_2	k_{1a}	k_{1b}	Code
1	0	0	0	0	0
2	1	0	1	0	1 0 0 0
15	3	7	2	0	1 1 0 0 0 1 1 1
225	7	127	3	0	1 1 1 0 0 0 0 1 1 1 1 1 1 1

Figure 2.6: Examples of Elias-δ code.

Elias-δ increases efficiency of encoding larger numbers by sacrificing some efficiency for small numbers. In order to encode an arbitrary integer k in Elias-δ code, we require L bits, where L is:

$L = 2(\log_2(1 + \lfloor \log_2 k \rfloor)) + 1 + \lfloor \log_2 k \rfloor + 1 = \lfloor \log_2 k \rfloor + 2\lfloor \log_2(1 + \lfloor \log_2 k \rfloor) \rfloor + 2$

Elias codes are not generally used because many bit-operations are required for decoding.

2.2.2.2 Byte-Aligned Encoding

Bit-aligned codes are not fast in practice since processors (and programming languages) are built to handle bytes efficiently not bits. Therefore, encoding and decoding based on byte would be faster than bit in practice. This is the reason that *Byte-aligned* [182] or *word-aligned* codes [216, 60] are presented. The popular method for byte-aligned compression is *variable byte length* coding which is commonly known as *v-byte*. This coding is very similar to UTF-8 encoding in text. This method presents small numbers with short codes and longer numbers with longer codes. The length of the shortest v-byte code is one byte since each code is a series of bytes. For instance, the number of bits used for encoding 1 in v-byte is eight times of the encoding for this number in Elias-γ. But in general, the difference in memory usage is not quite so dramatic in v-byte.

In the v-byte code, the low seven bits of each byte indicate numeric data in binary and the high bit is a terminator bit. For numbers larger than 127, the first byte stores the least seven bits and the next byte is represented by the next seven bits. This process is iterated until all bits have been stored. The high bit is 0 if the actual byte is the last of the code, otherwise; it is set to 1. Figure 2.7 presents some examples of encoding numbers based on the v-byte code.

Number (k)	Binary Code	Hexadecimal
1	1 000001	81
2	1 000010	82
128	0 0000001 1 0000000	01 80

Figure 2.7: Examples of v-byte code.

2.2.2.3 Block-Based Encoding

The problem of the delta encoding for query processing approaches is that extracting posting lists data is optimal if they are traversed from the beginning since random access to these data can increase the speed of query processing. To solve this problem, posting lists can be divided into blocks that would be encoded separately based on delta encoding [61, 180]. Therefore, to improve compression and decompression speed, *block-based encodings* is presented based on the general idea of packing many small integers in a computer word and encoding large numbers in the number of needed bits. This type of encoding is known as block-based encodings. The popular block-based encodings are *Simple9* [3], *Simple16* [220] and *PforDelta* [115, 227]. For instance, Simple9 packs as many numbers as possible in 32 bits (one word).

Recently, a group of researches [188, 132, 201] have used SIMD operations available on modern CPUs to provide highly space-efficient and fast-decoding

and encoding methods. For instance, *SIMD-BP128* [132] is encoding which uses 128-bit SIMD and is based on adaption of Simple9.

2.2.3 Query Processing Strategies

Strategies for processing a query are classified into three categories, *Term-At-A-Time* ($TAAT$), *Document-At-A-Time* ($DAAT$) and *Score-At-A-Time* ($SAAT$) approaches [119]. These strategies are defined for different index organizations. $TAAT$ [36, 164] processes query terms one by one, from shortest to longest posting lists. The first posting list is considered as a candidate answer set, then the answer set is refined by traversing the next posting list. The postings of each posting list are sorted by decreasing weight, (e.g., $TF - IDF$, $BM25$). $TAAT$ is mainly used for processing ranked unions which process approximate ranked union queries efficiently by pruning the posting list based on heuristic thresholds [119]. On the other hand, $DAAT$ [117, 120, 62] processes all posting lists of query terms in parallel, looking for the same document in all posting lists. To satisfy $DATT$ postings in a posting list must be sorted by increasing $docId$. This strategy is popular for Boolean intersections and unions. $DAAT$ is a very efficient strategy for processing exact ranked intersection [36, 72] because the final score of the top-k candidates are known since each document is processed completely. The last query process strategy is a combination of $TAAT$ and $DAAT$ which is called $SAAT$ [4, 223]. This strategy is practical for impact-sorted indexes [5] which precomputes the actual score contributions of each keyword and quantizes

them into impact scores [139]. The posting lists of impact-sorted index are divided into segments. $SAAT$ processes all segments of query keywords posting lists at the same time with a set of accumulator variables. This strategy is used for top-k rank retrieval queries like $DAAT$ although their approaches and indexing strategy are completely different. The tradeoff between effectiveness and efficiency of $DAAT$ and $SAAT$ has been studied by Crane et al. [54].

2.3 Semantic Search

Semantic search intends to retrieve information using additional information beyond pure keywords. As mentioned in Section 1, many queries cannot be effectively satisfied using keywords alone because it is difficult to capture user information needs and their needs solely based on keywords. Semantic search can be classified based on the type of input data (text, knowledge bases, combination of these) and the type of search (keyword, structured, natural language). In this thesis, we classify them based on input data, as we focus on indexing methods of semantic search. Our work falls within the scope of *full-text semantic search* as it considers text and semantic information simultaneously.

In this section, we briefly explain knowledge bases and RDF semantic search. Then, we provide an overview of components of common IR systems for semantic full-text search, and review the techniques related to document

processing, query processing, and retrieval.

2.3.1 Knowledge Bases

The main resource of data for a semantic search is the knowledge base(KB) which is a structured repository of entities (e.g., people, things and locations), their attributes and their relationships to other entities. A large number of knowledge bases have been built, such as Freebase [32], DBpedia [10, 130], YAGO [190, 191] and Wikidata [75, 205]. Knowledge bases are often stored as RDF triples. Each RDF triple is in the form of ¡*subject*, *predicate*, *object*¿, where the subject is a URI (link to another entity), and the object is a URI or a literal value. The predicate specifies the relation between subject and object and is represented by a URI. Consider for example the following RDF triples about Neil Armstrong from DBpedia:

< dbpedia: Neil Armstrong dbo: birthDate '1930-08-05' >

< dbpedia: Neil Armstrong dbo: mission dbpedia: Apollo_11 >

where the predicates birthDate" and mission" take literal and URI values, respectively.

2.3.2 Semantic RDF Search

Semantic search that is primarily and solely focused on retrieving RDF triples is called *semantic RDF search* [150, 176, 80, 181, 99, 69]. It indexes RDF triples to answer either hybrid or structure queries. In this respect, manag-

ing triples can be categorized into three different perspectives based on RDF, including a relational perspective, an entity perspective, and, a graph-based perspective [161, 204, 6]. The database community has focused on the first perspective regarding RDF management by considering an RDF graph as relational data and applying techniques such as storing, indexing, and answering queries related to relational data. Techniques developed under this perspective generally aim to support the full SPARQL language [1]. SPARQL is recursive acronym for: *SPARQL Protocol and RDF Query Language*, which can retrieve and change data stored in RDF. SPARQL is adapted from SQL, the standard query language for databases. In the relational perspective, RDFs are represented either vertically or horizontally. In cases when RDFs are represented vertically, RDF data is inserted into a single table over the relation schema (subject, predicate, object). In the horizontal representation, RDF data is stored in numerous smaller tables. RDF data presented in each table are conceptually related, which is an indication that they contain related predicates. The triple predicate values are interpreted as column names of each table and their subject values are considered as rows in the table. Therefore, for each (subject, predicate, object) triple, the object value is considered as the cell value of the table.

Blanco et al. [29] have introduced an effective and efficient entity search over RDF data by defining a horizontal index, which represents RDF resources

[1]Query Language for RDF, W3C Recommendation, http: //www.w3.org/TR/rdf-sparql-query.

using only three fields. The fields are assigned to tokens from values, properties, and subject URI. This is followed by defining a vertical index in order to create a field for each property occurring in the data to combine these indices in order to support entity search related to RDFs. They reduce the number of fields of vertical index by categorizing properties into the three different levels of important, neutral, and unimportant in an effort to improve the performance of the ranking process. The most renowned relational perspective index is SIREn [68], which is the index of the Sindice search engine [142]. SIREn uses an Inverted index and builds its index on Lucene by extending the inverted list component.

The IR community has introduced the entity perspective for managing RDFs. Resources in the RDF graph are interpreted as entities in the way text documents are specified by a set of keywords in common IR settings. In such settings, each entity is specified by a set of attribute-value pairs [142]. A prominent example of this group of RDF indices for web data search is Semplore [206] which translates triples into documents, fields, and keywords so as to support the encoding of the RDF graph and be able to answer hybrid queries (keyword-based tree-shaped queries) over RDF graphs. For instance, relation names are indexed as keywords, subjects are stored as documents, and objects of a relation are stored in the position list. Semplore, as an information retrieval engine for querying semantic web data, supports hybrid queries and is designed based on the inverted index philosophy. Semplore consists of an ontology index, an entity index, and a textual index. The

ontology index stores the ontology graph of the dataset, including hierarchical concepts and relations. Entity index is tasked with the responsibility of indexing the relationship between entities. The textual index handles all triples with the text predicate in order to provide keyword searches for the system.

The graph-based perspective focuses on supporting navigation in the RDF graph by considering subjects and objects as nodes of the graph, and directed, labeled edges as the predicates. General graph-based indexing methods are comprised of suffix array, structural index, tree labeling schema, and distance-based indexing. Yongming et al. [142] offer a comprehensive review of graph-based indexing methods in their paper.

2.3.3 Semantic Full-Text Search

Semantic full-text search is a form of semantic search that works on both on text and knowledge bases content simultaneously. This type of search offers indexing and search over full text and external semantic knowledge bases. In this section, we explain the main components of semantic full-text search, and review the techniques related to document processing, query understanding, and retrieval.

2.3.3.1 Document Processing

Document processing in semantic full-text search is mainly about extracting meaning from documents and indexing this type of information in addition

to textual keyword information. For this purpose, documents are represented with additional semantic information as follows:

Entity recognition identifies named entities in documents and labels each of them with the most likely type/class; [34, 87, 167].

Entity disambiguation maps entity mentions to their corresponding entries in a KB by assuming that named entities have already been recognized [37, 98].

Entity linking combines these two steps and focuses on identifying annotated entities in the text and connects them to entities in a KB. Preliminary works on entity linking focused on contextual similarity between the document and the candidate referent entities [58, 57]. Later, two concepts of relatedness and commonness were introduced as main features for entity linking [155]. Relatedness shows the semantic similarity between two entities by using their incoming and outgoing links in the KB. The two well-known entity linking systems are DBpedia Spotlight [148] and TAGME [84, 86]. The former uses the Vector Space Model to disambiguate named entities and the latter uses a voting scheme for a relatedness score to find collective agreement for the link targets. The knowledge base entities extracted through the entity linking process are used to index documents in the semantic index.

2.3.3.2 Query Understanding

The process of identifying the underlying intent of the queries, based on a particular representation" [55] is called query understanding. One of the

main approaches for query understanding focuses on the topical classification of query content. Most of the methods in this area predict the category of web queries by training a classifier [39, 135, 185]. Another approach is to divide each query into phrases and consider each phrase as a separate concept which is known as query segmentation [27, 97, 107, 112, 170, 196]. Rizzo et al. [170] consider a segment's frequency and mutual information between the subsequences of a segment. Bergsma and Wang [27] used decision-boundary, context and dependency features for query segmentation based on a supervised learning method. Hagen et al. [97] proposed a method based on n-gram frequencies and Wikipedia titles to perform query segmentation.

Named entity recognition in queries is also employed for understanding query targets by categorizing recognized named entities in a query into predefined group (e.g. people, book) [196, 2]. For instance, Guo et al. [196] used probabilistic methods and a weakly supervised learning algorithm (WD-LDA) to identify the intent of the query but such works can only detect mentioned entities and do not perform disambiguation or linking.

Entity linking for queries has been recently used for query understanding [147, 30, 96]. Meij et al. [147] proposed an important work for semantic linking in short texts by first finding high recall and then improving precision by using machine learning. Entity recognition and disambiguation are two main challenges [40] of entity linking in queries because of finding multiple query interpretations.

2.3.3.3 Retrieval

Full text semantic search techniques answer users query by returning direct answer or ranked list of entities, snippets and documents rather than just retrieving a ranked list of documents which is offered by a search engine. Full text semantic search can be classified into either *entity retrieval* [19, 213, 198] or *document retrieval* categories depending on the expected results. The former, entity retrieval, returns entities in the form of search results, hence, the response to the query "astronaut walk on the moon" would inevitably be "Neil Armstrong", "Buzz Aldrin", and the reference to the other astronauts who have set foot on the moon. However, document retrieval models retrieve documents by matching a particular user query or keyword against a set of free-text records. For instance, the result of the above query would contain a list of documents that include the keyword "walk", and entity "moon", "astronaut" or instances of the astronaut type.

Document Retrieval

Bhagdev et al. have introduced Hybrid search [28], which is a document retrieval-based system that performs search by combining keyword-based search and semantic search. This is aimed at simultaneously benefiting from the generality and extensibility of keyword-based search and the disambiguation ability of semantic search when metadata is available. Metadata is defined as information associated with a document that describe its context (e.g. author, title) and content (annotating parts of a document). Hybrid search uses a standard keyword-based engine (e.g., Solr) to index documents

and allow annotation information to be stored in the form of RDF triples. To answer a query, different components of the query (keywords and metadata) are identified. To retrieve the results, terms of the query are sent to the traditional IR system and keywords are matched with the origin of annotations in documents. Using SPARQL [184], the metadata are converted into a full-featured, structured query language. Ultimately, all the results retrieved from different queries are merged and ranked.

Mimir [194, 33] is an integrated semantic search framework, which has been designed as a web application. Mimir uses positional inverted indices and direct indices to store text, document structure, linguistic annotations, and, formal semantic knowledge in order to retrieve related documents. Mimir considers the positional inverted index to find the documents containing co-occurrences of a query term, which can include keywords, entities, and, annotations. The posting of these indices consists of a $docId$ and a position and they are sorted based on $docId$ in a posting list. $DocIds$ are consistent between positional inverted index and direct index. Direct index reverses the inverted index, maps a document to a term, and returns terms of a document with their term frequency. Indexing annotations is used to index the structural and generated annotations by storing annotation IDs and their starting positions. Mimir can combine conventional full text Boolean retrieval, structural annotation graph search, and, SPARQL-based concept search in order to retrieve more semantically related documents by taking into account document structure.

However, most of the recent studies [35, 64, 74, 134, 168, 140, 210, 211] in this area have been focused on improving query process and retrieval not indexing of a semantic search especially after the Freebase Annotations of ClueWeb Corpora (FACC1) [88] were released in 2013. For instance, Dalton et al. [64] used FACC1 annotations for ad hoc document retrieval for the first time. They took entity annotations of both documents and queries as input to improve document retrieval based on a query expansion technique. Their approach used entities in a knowledge base for enriching the query with different features extracted from entities. They used Galago [2], an open source search engine to build their index based on an inverted index. They indexed each document using different fields: bidirectional reference from words to entities in the KB, and indirectly to Freebase types and Wikipedia categories. Rui Li et al. [134] improved the performance of retrieval by presenting a query expansion technique with knowledge graph based on the Markov random fields model. They combined distribution of original query terms, documents and two expanded alternatives, i.e. entities and properties. Ensan and Bagheri [74] and Raviv et al. [168] have recently extended language models to represent entity annotations of document and queries. These studies did not report their query process time since combination of semantic information and text in formation is time consuming. In this thesis we focus on improving efficiency (query process time and memory usage) and effectiveness of semantic search.

[2] `http://www.lemurproject.org/galago.php`

Entity Retrieval

Using entities in retrieval is a research direction that has been well studied. Since, many information can be discovered around entities. For example, different application domains, including question answering [127], enterprise search [14], and web search [157] use entities. In this section, we just summarize studies on entity retrieval that try to improve efficiency and effectiveness of semantic search in regards to the indexing part.

Chakrabarti et al. [43] present four indices to support proximity search in type-annotated corpora. Referred to as *stem* index, the first of the four indices maps stemmed terms to common posting lists. The stem index is a common search index, which stores all stemmed terms in the corpus. The second index called the *aType* Index stores each occurrence of an entity (e.g. Neil Armstrong) in a document and all its types (e.g., Astronaut, Person) with the same entity offset. The third index is the reachable index which takes two atypes or an atype and a token, in order to determine if atype is an ancestor or child of a token or other forms of atype in $O(1)$ time based on atype taxonomy. This index has the ability to recognize that "Person" is an ancestor of "Astronaut". The last of the indices is called the *forward index*, which answers ($docId$, term-offset) queries and creates snippets that specify tokens of a query in search responses. This index stores all terms that can be found in each document based on their term frequency. It should be pointed out that the main focus of this index is to enhance the optimization of proximity queries and perform local proximity instead of global proximity

[43].

In line with Chakrabarti's work, Bast et al. introduced a new compact index data structure known as HYB [23]. According to Bast et al., HYB supports very fast prefix searches, which enable other advanced queries including query expansion, faceted search, fast error tolerant search, database-style *select* and *join*, and, fast semantic search (ESTER) [21, 20, 22]. HYB is designed based on the philosophy of inverted index in a way that each posting in HYB contains the $docId$, termId, position and weight. The termId is the main reason for having a fast prefix search by applying HYB since termId is assigned to the term, based on lexicographical order. HYB vocabulary contains term ranges, also known as blocks, instead of unambiguous terms. The study conducted by Bast et al. reveals that compared to a state-of the-art compressed inverted index, HYB exhibits superiority in that it uses the same amount of space while processing the query ten times faster. Bast et al. also succeeded in creating a semantic search engine (Broccoli) based on the HYB structure. The relationship between terms and entities is defined by the *occurs-with* relation that identifies which terms and entities occur in the same context. The concept of context is determined by the syntactic analysis of the document sentences and extraction of syntactic dependency relations. Extracting context is carried out automatically at indexing time. The entity engine [21, 137] and dual inversion index [47] present index data structures in order to support Entity Index with efficient query response time. Dual inverted index uses two indices, *document-inverted* index and

entity-inverted index, for efficient and scalable parallel query processing. The query is processed based on the two concepts of *context matching* and global *aggregation across* the entire collection. Through context matching, the occurrence of entities and terms in the desired context pattern is measured (e.g., co-occurred in a window with length of ten terms). Although, the entity-inverted index uses more space than document-inverted index, it has, however, proven to be practical for 1020 types with a focus on improving query process time. From the database perspective, this work can be regarded as an *aggregate join query*.

The novel document inverted index for annotations proposed by Chakrabarti et al. [42] can handle in excess of a billion Web pages, more than $200,000$ types, over $1,500,000$ entities, and hundreds of entity annotations per page. The memory usage annotation index is comprised of two parts that include the Entity Index and Type Index designed based on the Inverted Index philosophy. The Entity Index postings store $docId$, left and right positions and extra information such as confidence values while type posting lists contain a block for every document to store entities of the key type that occurred in that documents This is referred to as the general style Snippet Interleaved Postings (SIP) coding. SIP is designed to prevent the repetition of the occurrence of entity id in posting lists by defining a shorter version for each entity in a particular posting. SIPs inline information from the snippets into specially designed posting lists in order to avoid disk seeks. In lieu of the apparent advantages of SIP, there is no reference to its retrieval process and

how Type Index and Entity Index relate to one another throughout the query processing task. Chakrabarti et al. are in fact adamant that SIP's memory usage are more efficient than public-domain indexing systems such as Lucene [44, 42].

2.4 Neural Models for Information Retrieval

The information retrieval community has recently become engaged in using neural methods for improving retrieval performance. The objective is to use neural methods as a relevance estimation function to interrelate document and query spaces. Neural models have primarily been used for determining relevance even for cases when query and document collections do not share the same vocabulary set. For example, several researchers have used neural models to estimate relevancy by jointly learning representations for queries and documents [95] whereas some other researchers have used neural models for inexact matching by comparing the query with the document directly in the embedding space [157, 108] or through the semantic expansion of the query [219, 70, 125]. In this context, neural word embeddings have been aggregated through a variety of approaches, such as averaging the embeddings and non-linear combinations of word vectors (e.g., Fisher Kernel Framework [51]) [129]. The majority of these works are focused on proposing more accurate relevance *ranking* methods and as such differ from our work, which is primarily for indexing relevant documents. The main difference lies in

the fact that ranking methods use a set of documents retrieved by a base retrieval method and re-rank them based on some relevance function, while our work serves as the underlying indexing mechanism for maintaining the list of relevant documents that can then be used in ranking methods.

Many information retrieval techniques are based on Language models, which are concerned with modeling the distribution of sequences of words in a corpus or a natural language. Researchers have shown that the probability distribution over sequences of words can be effectively capture through a neural model leading to the so-called *neural language models* [90]. In neural language models, the input words are modeled as vectors whose values are gradually trained using the error back-propagation algorithm in order to maximize the training set log-likelihood of the terms that have been seen together in neighboring sequences. While earlier ideas for building neural language model was based on feed-forward neural network architectures [25], later models focused on recurrent neural networks as the underlying architecture as they provide the means to capture sequentially extended dependencies [152, 153]. Several authors have also proposed that Long Short-Term Memory-based neural networks would be a suitable representation for learning neural language models as they are robust in learning long term dependencies [193, 175]. From an application perspective, there has been work that explores the possibility of learning neural multimodal language models that can condition text on images and also images on text for bi-directional retrieval. The work by Djuric et al [73] introduces a hierarchical neural lan-

guage model that consists of two neural networks where one is used to model document sequences and one to learn word sequences within documents. This model is relevant to our work as it dynamically learns embedding representations for both words and documents simultaneously, which relates to the aspect of our work that learn homogeneous representations for terms, entities, types and documents.

Other areas in information retrieval such as entity retrieval that is concerned with finding the most relevant entity from the knowledge graph to an input query [110, 106] and entity disambiguation that focuses on finding the correct sense of an ambiguous phrase [78, 214, 156] have used neural representations for more efficient retrieval performance. Other application areas such as query reformulation including both query rewriting [93] and query expansion [82], which are used to increase retrieval efficiency, have employed neural representations, and more specifically neural embeddings. One of the main advantages of applying neural models in these contexts is the possibility of overcoming vocabulary mismatch where the embedding representations allow for soft matching between the query and search spaces, be it documents, entities or disambiguation options. This has been the advantage observed in our work as well where the embedding representation of terms, entities, types and documents go beyond the hard matching of exact terms and entities within documents and similarity/relevance is calculated based on the similarity of the learnt embeddings.

Given our work considers the integration of multiple information types into

the same embedding space, it is important to cover related work that have attempted to train joint embedding models as well. One of the earlier works to jointly embed two types of information was the work by Chen et al [45], which jointly learns embedding representations at character and word levels in the Chinese language. The authors showed this was important due to the compositionality of the structural components of words in the Chinese language. Wang et al [207] and Yamada et al [215] considered embedding words and entities into the same continuous vector space for the sake of named entity disambiguation. The joint embedding model considers both the knowledge base hierarchical structure as well as the word co-occurrence patterns. Toutanova et al. [199] also learn a similar joint embedding between words and knowledge base entities but instead in the context of the knowledge base completion task. Our work focuses on a similar problem but with attention specifically towards learning a joint embedding representation for terms, entities, types and documents in the context of building semantic inverted indices, which to our knowledge has not been attempted in the past.

2.4.1 Word Embedding Algorithms

In traditional information retrieval techniques, the relation between a term and a document is often defined based on some measure of relevance such as $TF - IDF$. On this basis many retrieval models focus on building a vector space based on such measures of relevance [221]. In the vector space, each term/word is represented as a vector whose dimensionality is equal to the

vocabulary size. A clear limitation of such an approach is *curse of dimensionality* [24, 109]. As such, more recent models learn dense yet meaningful vector representations for words in a given corpus. These dense vectors are often known as *word embeddings* (distributed word representations) [151, 154] and if learnt based on a neural network model would be referred to as *neural embeddings*. Word embeddings have been used for a long time in academia. For instance, the Neural Network Language Model [25], that was made back in 2003, learns word embeddings and a statistical language model simultaneously. Existing works have proposed changes on this model but word embeddings have taken off since Mikolov et al. [151] proposed two simple log-linear models which considerably reduced time complexity and outperformed all prior complex architectures. Their models, which are called *Word2Vec*, can be used for bigger corpora while presenting more accurate embeddings for all kind of NLP models. Many researchers proposed improvements on these models, but Word2Vec remains to be a strong baseline and default source of word embeddings in publications since 2013.

After Word2Vec, GloVe [163] was proposed which is now also express as a second well known word embedding algorithm. These algorithms do not have many similarities on the surface, but at present they both achieve similar results on most tasks. Because, GloVe and Word2Vec perform under the same statement that words with similar contexts have similar meaning. It has been shown that neural embeddings can maintain syntactic and semantic relations between words [154, 221].

2.4.1.1 Word2Vec

Word2Vec [151] has two variations namely *skip-gram* (SG) and *continuous bag-of-words* ($CBOW$). In both of model variations, word embeddings are learnt by using a three-layer neural network with one hidden layer. Given a sequence of words, the $CBOW$ model is trained such that each word can be predicted based on its context, defined as the words surrounding the target word, while the SG model is trained to predict the surrounding words of the target word. It has been shown in practice that SG models have better representation power when the corpus is small while $CBOW$ is more suitable for larger datasets and is faster compared to the SG model [151, 154].
In terms of a more concrete formalization, the $CBOW$ model predicts a target word by maximizing the log conditional probability of the target word given the context words occurring within a fixed-length window around it. For instance, if we assume a given sequence of training words $w_1, w_2, .., w_T$ and define the context as c words to the right and left of the target word, the following objective function would need to be maximized.

$$\frac{1}{T}\sum_{t=1}^{T}\log p(w_t|\sum_{-c\leq j\leq c, j\neq 0} w_{t+j}) \tag{2.12}$$

This shows the $CBOW$ model sums up the context vectors to predict the target word. In contrast, within the SG model, vectors are trained to predict the context words by maximizing the classification of a word based on its context.

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-c\leq j\leq c, j\neq 0}\log p(w_{t+j}|w_t) \tag{2.13}$$

The probability p is defined in both models as a *Softmax* function, where u_w is a context embedding vector for w and v_w is a target embedding vector. The following definition is used for $CBOW$; the target and context vectors would be placed in reverse order for the SG model.

$$p(w_c|w_t) = \frac{exp(v_{w_c}^T u_{w_t})}{\sum_{w=1}^{W} exp(v_w^T u_{w_t})} \tag{2.14}$$

Mikolov et al. [154] proposed two alternatives for the Softmax function because computing the gradient has a complexity proportional to the vocabulary size W which can be too expensive. The first one is *Hierarchical Softmax*, which is the efficient approximation of the Softmax. It represents the output layer as a binary tree whose leaves are W words and each node of it represents the relative probabilities of its child nodes. Therefore, the complexity of hierarchical softmax is $O(\log_2 W)$. Mikolov et al. [154] use a binary Huffman tree to assign short codes to the frequent words to have fast training. The second alternative is *Negative Sampling* which assigns high probabilities to relevant words, and low probabilities to noise words. Therefore, the loss function is scaled only with the number of noise words which are much lower than W. They show that the practical value of noise words for small training datasets and large datasets are in the range of 5-20 and 2-5, respectively.

2.4.1.2 GloVe

Pennington et al. [163] separated training word embedding models in to two groups: global matrix factorization methods such as latent semantic analysis [67] and local context window methods such as Word2Vec. They claim both of them suffer serious drawbacks. Context window methods do well on the analogy task, however, they cannot use global word co-occurrence statistics. But, they proposed the model that utilizes global co-occurrence counts while simultaneously capturing the same vector space semantic structure as Word2Vec.

The main measure in this model for similarity is co-occurrence probability. This concept can be better understood with an example. Assume two related words w_1= king and w_2= queen; the relationship between them can be measured by looking at their co-occurrence with a set of m probe words, $\frac{P_{w_1 m}}{P_{w_2 m}}$. Consider w_1 and w_2 are a semantic axis, then the word $m = men$ for the specified ratio will be large, since it is located at the positive end of the scale. While the fraction will be small for a word like *women* specifying that it is semantically at the other end of the scale. The value for unrelated words (e.g., $m = fashion$) and highly related words (e.g., $m = kingdom$) to both scales will be one. GloVe proposes a formalism that describes the above phenomenon and trains the embeddings to simulate such a structure. Accordingly, GloVe takes advantage of global statistics directly while Word2Vec indirectly achieves these statistics by sequentially scanning the corpus.

2.4.2 Document Similarity

This section provides summary of the state-of-the-art in *Semantic Textual Similarity*. This section only reviews unsupervised algorithms that use word embeddings to compute semantic similarities between sentences, paragraphs or documents.

The oldest and simplest model is *Vector Space Model* which was mentioned in Section 2.1.3. Recently new models based on embedding algorithms are proposed for computing semantic similarity between pieces of text. The main observation in the embedding algorithms shows that there are two trends in this research direction. Some of them insist on using deep neural architectures to learn complex patterns (e.g., Sent2Vec). But, their training step is very expensive. In contrast, training shallow algorithms is cheap which provides the possibility of being trained on larger datasets however, they are not as powerful as algorithm based on deep neural architectures. Arora et al. [7] recently showed in many cases, simple weighted embedding centroids outperform these more powerful models.

2.4.2.1 Paragraph Embeddings

While word embeddings are able to efficiently and accurately learn embeddings for document words, it is often desirable to learn vector representations for larger portions of documents such as paragraphs. For instance, it would be quite useful to have a vector representation for a paragraph that has appeared in a given document. To this end, Le and Mikolov have introduced the

notion of *Paragraph Vectors*(PV) [129] as an extension to *Word2Vec* to learn relationships between words and paragraphs. In this model, paragraphs are embedded in the same vector space as words and hence their vectors are comparable. PV models can capture similarities between paragraphs by learning the embedding space in such a way that similar paragraphs are placed near each other in the space. Similar to the neural embeddings of words, PV models are trained based on completely unlabeled paragraph collections and rely solely on word and paragraph positional closeness. PV models can be trained based on two variations. The first method is the *Distributed Memory Model* ($PV - DM$), which assumes each paragraph to be a unique token and uses the word vectors in the paragraph context to learn a vector for the paragraph. The second method is the *Distributed Bag-of-Words* ($PV - DBOW$) model, which is trained by maximizing the likelihood of words that are randomly sampled from the paragraph and ignores the order of words in the paragraph [129]. The generated output of both paragraph models is an embedding space that consists of dense vector representations of words and paragraphs, which are directly comparable and hence homogeneous in nature. The PV-DM has been empirically shown to have superior performance [129, 228]. As explained in subsequent sections, we employ the PV-DM model to learn embeddings for terms, entities and types within the same embedding space as to make them comparable.

2.4.2.2 Word Movers Distance

Word Movers Distance (WMD) [123] can be considered to be the state of the art method for measuring document distances based on word embeddings. This method proposes a distance function that directly uses two sets of word embeddings, without computing intermediate representations.

The WMD similarity function is defined based on the well known *Earth Movers Distance* transportation optimization problem. This problem is a popular mathematical construct which can be used to compare two probability distributions [173].

In Earth Movers Distance problem; we assume several suppliers with specified amount of goods should provide what consumers need where each customer has a limited capacity. The cost of transporting a single unit of goods for each supplier to consumer is known. The goal of this problem is to find a least expensive transform of goods that satisfies the consumers' request.

Hence, WMD encodes each document as a set of word embeddings, each with a distinct weight (e.g. $TF-IDF$ or $BM25$) similar to the vector space model. The goal of WMD is to optimally move the words of one document to the second document. If the distance between two documents is high, it means documents are semantically different. The reverse happens when the documents are similar.

2.4.2.3 Sent2Vec

Pagliardini et al. proposed *Sent2Vec*[162] as an algorithm for embedding sentences. However, their model is general enough to generate embeddings for paragraphs and documents. They create document embeddings by averaging their component word embeddings, but those word embeddings are trained in a way to generate meaningful average document representations. Sent2Vec employs deep neural architectures with thousands of internal weights to train. The key advantage of Sent2Vec over deep neural models is that there is no need for creating sentence embeddings from their component word embeddings. Since, this model just needs to compute a vector centroid (constant operation) to achieve this goal.

In practice, this model is almost identical to the CBOW model from Word2Vec. This model is a window based embedding algorithm that scans sequentially through the corpus. The target word is the middle one in the window and the rest are the context.

Main difference between CBOW and Sent2Vec are:

- In CBOW the context window size can be any arbitrary size while in Sent2Vec windows can only be clear semantic units (e.g. sentences, paragraphs and full documents). The Sent2Vec points out that the embeddings are optimized if the centroid of a set contains the relevant information for representing the meaning of the set.

- In Sent2Vec, contexts also contain word n-grams since many concepts

are often expressed by multi-word phrases (e.g. scientific and technical text)

- Word2Vec improves generality by performing random word sub-sampling as a regularization step. In Sent2Vec, random words are deleted however, sub-sampling is performed on the context after it extracts all the n-grams.

2.4.3 Approximate Nearest Neighbor Search

K-nearest neighbour algorithms are a class of non-parametric methods that are used for text classification and regression [218, 217]. They find the nearest points in distance to a given query point and are a simple and generalized form of nearest neighbor search, which also is called similarity search, proximity search, or close item search. Applying nearest neighborhood to high-dimensional feature space deteriorates performance because the distance between the nearest neighbors can be high. Therefore, research has concentrated on finding approximations of the exact nearest neighbors [109, 11]. Finding approximate nearest neighbors has be presented in a variety of algorithms such as using kd-trees [26] or M-trees [49] by ending the search early or building graphs from the dataset, where each data point is presented with a vertex and its true nearest neighbors are adjacent to the vertex. Other methods use hashing such as *Locality-sensitive hashing* (LSH) [109] to project data points into a lower-dimensional space. In our work, approximate near-

est neighbor search is used to find the top-k approximate nearest documents to a query term. This retrieval method of documents is based on the assumption that the top-k nearest documents in embedding space will be the most related set of documents within the corpus to the query term.

Chapter 3

Proposed Approaches

3.1 Explicit Semantic Full-Text Index

A semantic full-text search engine retrieves documents on the basis of the similarity of the entities and the keywords that are observed within the document and the query [194]. In order to be able to determine similarity, a combination of entities, keywords and entity types need to be properly indexed. The objective of explicit semantic full-text index is to identify the most suitable indexing data structures that allow us to efficiently store and retrieve these three types of information. We also need to customize one or a collection of data structures that provide the possibility for performing Boolean retrieval and ranked retrieval in an efficient way. Figure 3.1 provides an overview of how we process query such as "books about recommender systems written by Dietmar Jannach" using the explicit semantic

full-text index.

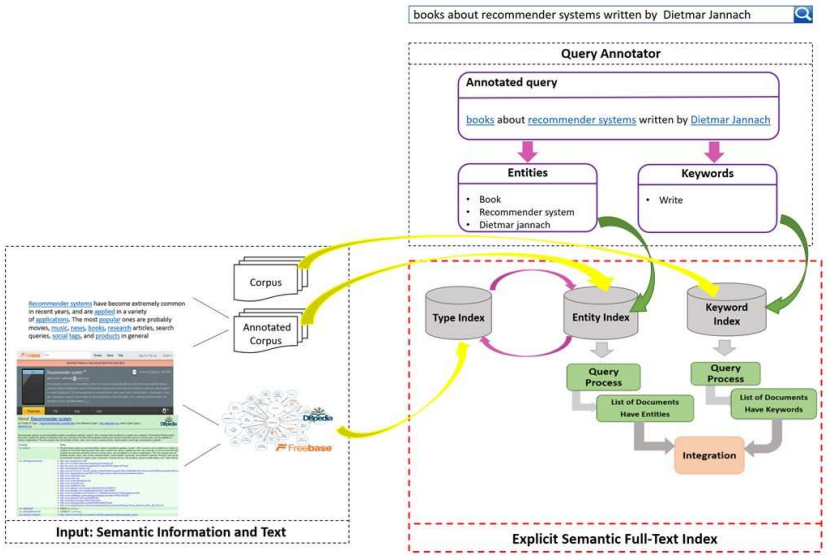

Figure 3.1: The workflow of the explicit semantic full-text index.

In order to be able to integrate keyword, entity and, type information, we build three separate indices, namely the Keyword Index, Entity Index, and, Type Index. The reason for considering these three indices is that the information that needs to be stored for each keyword, entity and, type are not the same. For example, a type posting would need to store super-types, sub-types and type instances, while an entity posting would only consist of information such as $docIds$, entity frequencies and confidence values. Table 3.1 presents the information that needs to be stored in each keyword, entity and, type posting.

We will briefly introduce the types of information that are stored in each index.

Index	Posting
Keyword	Document Identifier (docId) Term Frequency (TF)
Entity	Document Identifier (docId) Term Frequency (TF) Confidence Value (CV)
Type	List of its SuperTypes List of its SubTypes List of its Entities/Instances

Table 3.1: Information which are stored in each keyword, entity and, type posting.

Keyword Index

The commonly used text index is referred to as the Keyword Index in our work. The posting lists of our Keyword Index contain one posting per distinct document in which the keyword has occurred. The Keyword Index stores the $docId$ and the corresponding TF of the keyword in that document. This index needs to be flexible enough for any kind of retrieval algorithm especially ranked retrieval algorithms. It often stores term frequency information that allows for the computation of TF-IDF, BM25 or any other scoring function for ranking. However, the complete weight computation needs to be done at query time, which increases query process time. Keyword Index can answer common queries (e.g., "what is a telescope") on its own without the need for consulting the other two indices.

Entity Index

The Entity Index stores information about the entities that have been detected in a document through the use of semantic annotation or entity linking systems. Each posting list of the Entity Index stores information related to a given entity including the documents where it has been observed, the frequency of that entity in the document and also the confidence value (CV) of the entity in that document produced by the entity linking system.

Type Index

The Type Index stores structural information about entity types in order to enable reasoning on entities and their types during the retrieval process, e.g., to perform *query expansion*. Such an index would enable us to retrieve documents that do not have the keywords "moon" and "astronaut" but include the entity "Neil Armstrong" by considering the mentioned entities and their types. Within the Type Index, we store super-types, sub-types and instances of all of the types in our knowledge base. Thus, the posting list of each type is composed of three lists for each of the sub-types, super-types and instances.

Integrating Keyword, Entity and Type Indices

Defining relations between the Keyword, Entity and Type indices is important since their integration provides the means to perform additional reasoning for optimally connecting the query and document spaces. The relation and flow of information between these indices are shown in red dashed lines in Figure 3.1. The Entity Index serves as an interface between the Keyword Index and the Type Index since there are no explicit relations between

keywords and Types.

Let us review the case of query expansion to show how the integration of the three indices can support semantic interpretation of queries that were not possible before. Given an input query that consists of at least one entity, the Type index can be consulted to find the type, super-types, sub-types and/or instances related to the entity. The identification of the types in the type hierarchy that relate to the mentioned entities in the query would allow us to expand the query by adding semantically related entities into the query. For instance, for a query such as "first astronaut to walk on Mars", two entities can be spotted in the query, namely "astronaut" and "Mars". The Type Index would inform us that "Astronaut" is a sub-type of "Person" and that it has several instances such as "Yuri Gagarin" and "Neil Armstrong". It would further tell us that "Mars" is of type "Planet". The result for such a query would be the intersection of "Mars" and "Astronaut". However, given this would likely be an empty set, one can use the extended type information from the Type Index to expand the query. For instance, based on the Type Index, other instances of the "Planet" type could be added to the query, e.g., "Moon", that could lead to a reasonable result.

We will present different integration strategies for combining the Entity and Keyword Indices in Section 3.1.3. These integration strategies are categorized into two groups: homogeneous semantic full-text indices and heterogeneous semantic full-text indices. In the former, the data structure of the Entity Index and Keyword Index is the same; hence, the keyword homogeneous

semantic full-text index. On the other hand, integrating heterogeneous semantic full-text indices is used when the data structure of the Entity Index is not the same as that of the Keyword Index. Integration approaches in this group consist of list-based integration and non-list-based integration.

3.1.1 Explicit Semantic Full-Text Index Data Structures

As discussed earlier and based on the characteristics provided in Table 2.1, Wavelet Trees, Treaps and HashMaps are adopted as the data structures for constructing the three types of indices. We will discuss how these three data structures can be efficiently adopted as an indexing data structure.
In the following sections, we show how the three indices can be represented by each of the three data structures.

3.1.1.1 Treap Indices

Treap has recently been adopted as a new representation format for storing the inverted index that leads to decreased query process time and memory usage compared to the other state-of-art inverted indices [121]. Using Treap as a posting list data structure gives the opportunity to store postings based on $docId$ and weight (e.g. TF, $TF-IDF$) at the same time. This structure provides the ability to support ranked queries efficiently.

Treap-based Keyword Index

Here, the structure of the Keyword Index is based on the inverted index philosophy by representing each posting list as a Treap [121]. Each posting is considered to be a node of the Treap in such a way that $docId$ is the node key value and TF is the node priority value. Figure 2.4 represents a posting list of a term. Therefore, postings in a posting list are sorted incrementally and stored based on $docId$ while a heap ordering on TF is maintained.

Treap-based Entity Index

The Entity Index can be built using a Treap in a similar way to the Keyword Index with the exception that each Entity posting consists of $docId$, TF and CV as mentioned in Table 3.1. Given that each node of the Treap can only consist of one value pair, i.e., key value and priority value, we need to make some modifications so that additional information can be stored. To this end, we combine the TF and CV values to produce a priority value for the nodes in the Treap. The combination approach needs to be able to encode and decode quickly since the time of encoding and decoding affects the query process time. The priority value is built by concatenating the value of TF, a "0" character and the value of CV multiplied by 10^3. For instance, if TF is 10 and CV is 1 then their combination would be 10**0**1000. To decode this sequence, we start from the right side of this sequence and move until we find the first zero after seeing a number between 1 and 9. Then the numbers on the right side of the zero provide the CV value and the left side is the TF value.

Treap-based Type Index

Adopting the Treap structure for the Type Index is not possible since the posting lists of a Type Index consist of three independent lists: super-types, sub-types and entities. Treaps can only represent a list of pairs (key, priority) or two lists that can be transformed into a list of pairs. Furthermore, maintaining a heap ordering on the information within the Type Index does not have any meaning, which is required by the Treap. Therefore, the Treap structure cannot be considered as a data structure for our Type Index.

3.1.1.2 Wavelet Tree Indices

The Wavelet Tree data structure has already been widely used for keyword-based indexing in inverted indices, document retrieval indices, and, full-text indices [117, 89, 158, 159, 92]. However, it has not yet been explored to represent Type and Entity information. We consider the Wavelet Tree as one candidate data structure in our work.

Wavelet Tree-based Keyword Index

The structure of the Keyword Index based on Wavelet Trees can be adopted from the dual sorted inverted list [117, 159]. According to this, postings of the keyword posting lists are sorted by decreasing TF. A posting list associated with a keyword t is converted to two lists: a list of $docIdsS_t[1, df_t]$ and a list of TFs $W_t[1, df_t]$ by keeping their order. The number of documents where the keyword t appears is shown with df_t. This process is done for all keywords in the corpus, then all lists S_t are concatenated into a unique list $S[1, n]$ which is represented by a Wavelet Tree. Here, n is the number of

distinct keywords in the corpus. The starting position of each S_t is marked in a bitvector $V[1, n]$ to know the boundary of each $S_t \in S$. Given the fact that there are many equal values in a list W_t, in order to save space, only the non-zero differences between these values are stored in the list $W_t[1, m]$ where $m \leq df_t$. All lists W_t are concatenated into a unique list $W[1, M], M \leq n$. Then places of nonzero differences are specified in a bitmap $W'[1, n]$, which is aligned with S. So, the TF value of t for a document in position i of S is extracted using $W[rank_1(W', i)]$ instead of $W[i]$. Query $rank_1(W', i)$ returns the number of $1s$ in W' till reaching position i, which is the number of non-zero differences.

Wavelet Tree-based Entity Index

The structure of the Entity Index is similar to that of the Keyword Index. Accordingly, postings of an entity posting list are sorted in descending order based on TF and each posting list is transferred to two lists, $docIds$ and the combination of TFs and CVs (weights). The combination of TF and CV values of each posting is achieved using the strategy used for Treaps. The list of docIds S and list of weight W are created through the technique used for the Keyword Index. However, the value of weights is significantly larger than keyword TFs and the number of repeated weights among them is fewer in each posting list. To make this approach more practical, we define an arbitrary start value of "104" based on our scheme instead of just using 1 for the weight, because it is the first possible value of the weight of an entity since weight is the combination value of TF and CV. The base weight is

specified based on the possible minimum value for the TF and CV in our corpus. The reason we do not consider two lists for storing TF and CV is that extracting each of these values individually needs more time compared to the proposed approach.

Wavelet Tree-based Type Index

The Type Index is presented with three Wavelet Trees, namely Subtypes Wavelet Tree, Supertypes Wavelet Tree and Entities (instances of a type) Wavelet Tree. Let Sb_t be a list of all subtype identifiers (subtypeIds) of a type t. We propose to concatenate all lists Sb_t for all types in the corpus into a unique list $Sb[1, l]$. To know the boundary of a list Sb_t as suggested by Vlimki et al. [181], we insert a 0 at the end of each Sb_t in $Sb[1, l + T]$, where T is the number of available types observed at least once in the corpus. The reason for choosing 0 as a symbol to determine the boundary of each Sb_t in Sb is that the subtypeIds are larger than 0. The related information of a type with type identifier (typeId) i is between position p and q in Sb where $p = select_0(Sb, i)$ and $q = select_0(Sb, i + 1)$. For example, all subtypeIds between second and third 0 in Sb belong to the type whose typeId is three. We need to insert 0 for an empty list Sb_t because this gives us the ability to count the number of 0s in Sb with the rank function to know the typeId of a subtype in Sb. The Subtype Wavelet Tree presents the list Sb containing symbols from the alphabet $[0, H]$ where H is the number of available subtypes in the corpus. The Supertypes Wavelet Tree and Entities Wavelet Tree are created in the same way we create the Subtypes Wavelet Tree.

```
function getTypeId(x, S)
    num ← rank_x(S, S_lenght)                    ▷ x is a super-typeId/subtypeId/entityId
    i ← 1                                        ▷ S is a super-types/subtypes/entities sequence

    while i ≤ num do
        p ← select_x(S, i)
        typeIds ← rank_0(S, p) + 1               ▷ All types of x is stored in typeIds
        i ← i + 1
    end while
    return typeIds
end function
```

Figure 3.2: Function getTypeId returns all types of entities observed in the input, which are in the corpus.

These three Wavelet Trees support all necessary properties of the Type Index as discussed earlier in this section. For instance, to find all types that contain the given input (subtype/ super-type/ entity), each subsequence of subtypes, super-types and entities between two 0s can be checked by calling the Wavelet Tree select query. Whenever the result of select is not null, we use the Wavelet Tree rank function to extract the typeId by counting all 0s till that position plus 1. Function getTypeId in Figure 3.2 gives the pseudocode for this task of the Wavelet Tree Type Index.

3.1.1.3 HashMap Indices

The method for building the three HashMap Indices are similar to each other and the only difference between them is the structure of their posting list as mentioned earlier in this section. The keys and values of a HashMap index are entries of the inverted index vocabulary and their related posting lists, respectively. For instance, the entities of the corpus are the keys of the

HashMap-based Entity Index and the entity posting list of each entity is considered as the value of that key. Each posting list can be seen as a list of postings, where each posting belongs to a distinct document where the entity appears in.

3.1.2 Query Processing

Query processing of explicit semantic full-text index is composed of retrieving results of the Keyword, Entity and, Type Indices, and integrating the results of the Entity and Keyword Indices to retrieve the final results. Processing ranked queries and Boolean intersection queries on Treap and Wavelet Tree Indices are performed based on the DAAT. Retrieval algorithms of Boolean intersection queries and ranked intersection queries on HashMap Indices are implemented based on the TAAT while ranked union queries are processed based on the DAAT approach. In the following sections, we propose to use efficient query process algorithms for ranked queries and Boolean intersection queries.

3.1.2.1 Treap Query Processing

Processing ranked intersection and ranked union queries on the Treap has been introduced by Konow et al. [121]. In these query processing algorithms, the posting lists are traversed synchronously to find documents containing all or some of the query keywords, and in order to calculate the final weight of documents. For this purpose, a priority queue is used to store the top-k

results and a dynamic lower boundary (θ) is adopted after the priority queue size reaches k. This provides the ability to skip documents with a weight less than θ. The value of θ is updated whenever the size of the priority queue reaches $k+1$ and the document with the minimum weight is removed from the priority queue. Furthermore, the decreasing priority value (TF property) of Treaps offers the upper bound U for documents in the subtrees since the TF of the parent is larger than the TFs of its children. Because of the upper bound, it is possible to determine when to stop moving down the Treap ($U < \theta$). Treaps have been shown to be more efficient than Wavelet Trees and some implementations of the Inverted Index [121]. The ranked query process algorithms of Konow et al. [121] have been adopted in our work. We modify the rank intersection algorithm by removing θ, U and k from the algorithm to support processing Boolean intersection queries based on the DAAT approach. Consequently, all common documents of query posting lists are identified by simultaneously traversing Treaps.

3.1.2.2 Wavelet Tree Query Process

The work in [117, 159, 160] use the Wavelet Tree structure to represent the posting lists sorted by decreasing TF. This data structure supports ordering by increasing $docIds$ implicitly and efficiently with the help of its leaves ordered by ascending $docIds$ [121]. The authors implement approximate ranked unions and (exact) ranked intersections for ranked union queries and ranked intersection queries, respectively based on the TAAT and DAAT-like

approaches. The ranked intersection of a Wavelet Tree is even faster than a well-known Inverted Index (Block-Max) [121]. The main idea of the Wavelet Tree query processing is subdividing the universe of $docIds$ recursively to identify the final documents efficiently. The ranked query process of Wavelet Trees face the problem of not knowing frequencies until reaching an individual document, which causes efficiency issues in the Wavelet Tree [121]. The Wavelet Tree represents the input sequence, which is constructed by concatenating all posting lists S_t that consist of only the $docIds$. TF values are stored in a separate sequence W aligned with docIds sequence [117, 159].

The ranked union query can be processed based on Persin's algorithm [164] by defining a function to retrieve the k^{th} value of an interval, which gives the prefix of S_t where TF values are greater than the threshold, which is computed by an exponential search on W. The results are ordered by increasing $docIds$ in order to merge them with the set of accumulators which are ordered based on $docId$. Ranked union queries are processed rapidly by early stop, so we cannot be sure the reported weight of each document is fully scored/evaluated [36].

According to [117, 159], the ranked intersection query process is based on finding the query terms intervals $Sq[sq, eq]$ and then sorting them from shortest to longest. The process starts from the root and descends simultaneously to the left and right and updates the start and end points of all intervals until one of the intervals is empty or it reaches the leaves, which are the result documents of the intersection. The reason that the process stops going

down during the process is that there cannot be any common document from that subtree of the Wavelet Tree. The start and end point of each interval is updated based on $[rank_c(S, (sq1)) + 1, rank_c(S, eq)]$ and the value of c is 0 when it is going to the left child, and $c = 1$ on the right child. The result of a ranked intersection query is sorted based on increasing $docId$ because the leaves of the Wavelet Tree are ordered from the first to last symbol of the alphabet. The TFs of the result documents are calculated, and then the top-k result documents with the highest weights are returned.

The Boolean intersection and the Boolean union queries are also processed by adopting ranked intersection algorithms. The traversal of a Wavelet Tree in Boolean union queries is stopped whenever all intervals are empty instead of just one interval being empty. Our ranked intersection and Boolean queries are processed based on [117]. We define our ranked union query process algorithm by changing two parts of the Boolean union query process algorithm. First, the interval of each query term is modified to the interval $[sq, sq + k]$ if $sq+k < se$. This means that the length of all intervals is less than or equal to k. The reason for changing the end point of each query term intervals is that $docIds$ in each sequence S_t are sorted in a descending order by their TF and we need to find the top-k documents. Therefore, we union between just the top-k $docIds$ of each query interval and retrieve the top-k documents among them with the help of a min-priority queue with size k. This algorithm is an approximate ranked union since all documents and their weights are not considered.

3.1.2.3 HashMap Query Processing

The implemented processing algorithms for ranked intersection and Boolean intersection queries are based on the TAAT approach. Postings in the posting lists are sorted in an ascending order by their $docIds$. To process Boolean intersection, we sort the posting lists of query terms based on their length and then use the set-vs-set (svs) algorithm [14] for multi-lists to identify the common documents. The svs approach intersects multiple list by intersecting two of the shortest lists then intersecting the result with the next shortest one, and so on. The svs approach is the best approach for retrieving results of intersection queries compared to other approaches which are evaluated in the work by Barbey et al. [14]. The process of ranked intersection queries returns documents by a post processing step for finding the top-k documents of those retrieved from Boolean intersection. Retrieving results of ranked intersection queries based on the result of the Boolean intersection queries has already been used in earlier works [121]. The WAND algorithm (Weak AND, or Weighted AND) [36, 165] has been implemented to retrieve the results of ranked union queries. WAND retrieves the results based on the DAAT approach.

3.1.2.4 Type Index Query Processing

The Type Index processing algorithm only relates to the data structure of the Type Index and it does not depend on the type of the queries. The input and the output of the Type Index are entities of a query and a list of related

concepts (types, entities, super-types, and/or subtypes). As explained, the Type Index can be either implemented with Wavelet Tree or HashMap. Treap is not used as a data structure for the Type Index based on the discussion in Section 3.1.1.

Query Processing on HashMap-based Type Index

The HashMap Type Index maps each type to its posting list, which consists of three lists: super-types, sub-types and entities of that type. The first step of Type Index query process is to find related types of each query entity. We use a map to store the type(s) of each entity so retrieving the type of a concept is achieved quickly in $O(1)$.

Query Processing on Wavelet Tree-based Type Index

The combination of super-type Wavelet Tree, sub-type Wavelet Tree and entities Wavelet Tree is called the Wavelet Tree Type Index as described in Section 3.1.2.3. This Type Index does not need a map for assigning the query entities to the types. The GETTYPEID function (see Figure 3.2) is used to find the related types (T) of a query entity.

To retrieve the list of related entities, sub-types and super-types of each T, we give each of the entities sequence (Se), subtypes sequence (Sb) and super-types sequence (Sp) as the input to the GETRELATEDLIST function in Figure 3.3, respectively. In this function, the start and end positions of the input sequence are found, then all identifiers between these two positions are returned. For example, the result of GETRELATEDLIST (Se, t) is a list of entities whose type is t and the result of GETRELATEDLIST (Sb, t) is a list

```
function GetRelatedList(wt, tId)
    start ← select_0(wt, (tId − 1))                    ▷ tId is a typeId
    end ← select_0(wt, tId) − 1                        ▷ wt is a wavelet tree
    i ← 1

    while start < end do
        results ← wt[i]
    end while
    return results                                     ▷ All entities of tId is stored in results
end function
```

Figure 3.3: Function GETRELATEDLIST returns the subtypes/supertypes/entities of the input type (tId) according to the Wavelet Tree Type.

of subtypes of type t.

3.1.3 Integration of Entity and Keyword Indices

The retrieved documents from the Entity and Keyword indices need to be integrated to produce the final result. Our proposed integration approaches can be categorized into two groups: *Homogenous Integration* and *Heterogeneous Integration*. These approaches are used to process ranked queries and Boolean intersection queries.

To process a query such as "astronaut walk on the Moon", a relation between the Keyword Index and Entity Index needs to be found as discussed in Chapter 1. We assume the query is annotated so in this example "walk" is a keyword, "astronaut" and "moon" are entities. The query can simplistically[1] be processed based on the following steps: (1) look for occurrences

[1] Just to note that the query process enumerated here is not optimal and is just provided to discuss the integration of different indices.

of "astronaut" and/or "moon" in documents within the Entity Index. (2) Retrieve all instances, subtypes and super-types related to query entities(s) from the Type Index if the results of the Entity Index are not adequate (e.g. less than k for top-k queries). (3) Search the Keyword Index to return documents that contain the keyword "walk". (4) Integrate the results of the Keyword Index and Entity Index based on the $docIds$. (5) If the number of documents in final results is less than k, then we can add result documents of the Entity Index assuming that entities are more relevant for search than pure keywords.

In the above process, Step (4) is both challenging and expensive because the separate result document sets from Steps (1) and (3) are often large so integrating them is expensive [16]. Therefore, the method of integrating the result documents of the Entity Index and Keyword Index has a direct impact on the efficiency of query processing.

3.1.3.1 Homogenous Integration

In the homogenous integration approach, we integrate the result documents of the Keyword Index and Entity Index based on two approaches, namely *list-based* and *non-list-based* approaches. In the former, the results of the Entity and Keyword Indices are combined based on list intersection and list union algorithms. The integration of the Entity and Keyword Indices using the non-list-based approach is done during the query process of the Keyword Index (step 3); thus Step (4) is removed for processing a query based on

this integration approach. The integration approach of the Treap semantic full-text index and Wavelet Tree semantic full-text index (processing ranked queries) are implemented based on the non-list-based approach.

Homogenous Treap Integration

There is no need for explicitly integrating the Treap-based Keyword and Entity Indices since the data structure of both indices is the Treap and the Treap retrieval algorithm [121] processes all Treaps of the query keywords in parallel. Thus, the Entity posting lists and keyword posting lists are processed simultaneously to retrieve the final result. Therefore, we do not need extra time for integrating the Treap-based keyword and Treap-based Entity Indices. This can be considered to be an important advantage of the Treap data structure over the other data structures.

Homogenous HashMap Integration

The outputs of the HashMap-based Entity and Keyword Indices are integrated based on the list intersection (*svs*) and list union (*Merge*) algorithms. The results of the ranked intersection queries can be less than k since two lists are intersected with size k. In this situation, we use Step 5 to return up to k documents. In contrast, the number of retrieved documents of ranked union queries can be larger than k; in this case, the top-k among them are retrieved.

Homogenous Wavelet Tree Integration

In the Wavelet Tree-based Indices, the results of the ranked queries are integrated implicitly based on the non-list-based approach. This means that

the integration procedure is done during processing the Wavelet Tree-based Keyword Index. The integration procedure for processing ranked intersection queries (top-k) on the Wavelet Tree Indices includes the following steps:

1. Retrieve the result documents of processing ranked intersection queries on the Entity index (Section 6.1). The number of retrieved documents is k.

2. Find the intervals $r_q = [sq, eq]$ of the query terms Q with the help of the start position for $q \in Q$ in S. $eq = select_1(Sp, kId)$ and $sq = select_1(Sp, (kId - 1))$ where kId is the keyword identifier.

3. Search each document d from the retrieved documents of the Entity index in all intervals of the query keywords by checking the occurrences of d in each interval $[sq, eq]$, *is-occurred* $= rank_d(S, eq) - rank_d(S, sq)$. If *is-occurred* is greater than zero in all intervals, then d is inserted into the final document set (R).

4. Calculate the TF values of the documents in R on the Wavelet Tree-based Keyword Index.

5. If the number of documents is less than k after Step 3, add documents, which are retrieved in Step 1 but are not in R, retaining their order until the number of the final result documents reaches k.

6. Calculate the weight of each document d in the final result documents by adding the TF of d in the Keyword Index and weight of d in the

Entity Index.

The process of integrating the Entity and Keyword Indices for ranked union queries requires changing Step 3 of the above procedure to accept d as the result document if the *is-occurred* of d is greater than zero for at least one interval of the query keywords.

We do not use the above procedure to retrieve the results of Boolean intersection queries because the number of documents that are returned from processing this type of query on the Entity Index is significantly larger than k. Consequently, the above procedure is not efficient for integrating retrieval results of Boolean intersection queries on the Wavelet Tree-based Indices. Therefore, we integrate the documents of processing Boolean intersection queries on the Entity index and Keyword Index by using the intersection algorithm for two lists, i.e. *svs*.

3.1.3.2 Heterogeneous Integration

Besides the integration of Entity and Keyword Indices that are of the same data structure type, it is also possible to integrate two indices when they are built from two different data structure types. For instance, the Keyword Index can be built using a Wavelet Tree while the Entity Index is constructed using a Treap. On this basis, six combinations of the Treap, Wavelet Tree and HashMap data structures are defined for their integration. We will show in Section 4.1.4 that one of the heterogeneous integrations is more efficient than the other indices for processing ranked intersection queries. The integration

approaches for this type of index is implemented using the *svs* and *merging* algorithms. We use both non-list-based and list-based approaches to find the most efficient integration method for heterogeneous indices.

List-based Integration

In the list-based integration approach, the retrieval results of the query evaluation on the Entity and Keyword Indices are considered as a list of $docId$ along with their TFs. To evaluate ranked intersection queries and Boolean intersection queries, these two lists are integrated (intersected) using the *svs* algorithm. If the number of final results is less than k for ranked intersection queries, we use the same method that is explained for integrating HashMap-based indices (Section 3.1.3.1) to retrieve k documents. For performing ranked union for each index, we merge the two result lists along with updating the TF of the documents if they occurred in both lists. The result of merging two result lists is stored in the min-priority queue to return only the top-k $docIds$ and their associated TF.

Non-list-based Integration

Treap is a more efficient data structure for implementing the Keyword Index compared to Wavelet Tree and HashMaps. The results reported by Konow et al. [121] also reinforce our findings, presented in Chapter 4.1.4, confirming that Treap is an efficient data structure for the homogeneous indices. Therefore, we decided to use Treaps for integrating Keyword and Entity Indices when the data structure of one of these two indices is a Treap. This non-list-based method is called *Treap Integration*. The first step is to evaluate a query

on an index whose data structure is a Wavelet Tree or HashMap. Second, a new Treap is built for the retrieved results based on the $docId$ and TF of the retrieved documents. Note that the time for building a Treap is $O(\log n)$ where n is the number of nodes in the Treap. The final step is to evaluate the query on all of the Treaps based on the algorithm, which is explained in Section 3.2.1. The heterogeneous index is converted into a homogeneous Treap that can be used for integration as explained earlier.

We define a non-list-based method when the data structure of either the Entity or Keyword Indices is not a Treap. In such cases, the integration is performed using the Wavelet Tree, since we are interested in evaluating a query without using list-based algorithms. To evaluate ranked intersection queries, first, these queries are processed as Boolean intersection queries on the index which is built using HashMaps and the result documents are sorted based on descending TF. So the retrieved documents with higher TF are processed earlier than other documents. Then, we check the existence of each retrieved document on each query interval $St[s, e]$ of the Wavelet Tree-based index with query $\textit{is-occurred} = rank_d(St, e) - rank_d(St, s)$. If *is-occurred* is greater than zero for all query intervals, then d is one of the final documents. The final documents are stored in a min-priority queue with size k. Query processing is stopped as soon as the min-priority queue is full. The only difference between ranked intersection and Boolean intersection algorithms is that in Boolean intersection, we do not store the final documents in a min-priority queue since all common documents need to be retrieved.

Ranked union queries are processed by retrieving the result of ranked union queries on the HashMap Index. If the data structure of the Entity Index is a HashMap, then the weight of the document is updated. Otherwise, we find the documents, which are the result of a ranked union query of the HashMap index, which occurs at least in one of the query intervals of the Wavelet Tree-based Index and update their TF. If the number of the documents in the final results is less than k, then we process ranked union queries on the Wavelet Tree and add new documents to the final results.

The result of ranked queries of the non-list-based method might not have the exact score since we stop as soon as k documents are found. The summary of a query evaluation based on non-list-based of heterogeneous integration is presented in Table 3.2.

Index 1	Index 2	Retrieval steps
Treap	Wavelet Tree /HashMap	1) Query on Wavelet Tree/HashMap 2)Build a Treap from the results of *Step 1* 3)Evaluate the on all Treaps
Wavelet Tree	HashMap	1) Query evaluation on the HashMap 2) Check the results of *Step 1* in the Wavelet Tree 3) Only process ranked union queries if the data structure of the Entity Index is a Wavelet Tree. Evaluate the ranked union queries on the Wavelet Tree.

Table 3.2: The query evaluation procedure for non-list integration

3.2 Implicit Semantic Full-Text Index

As mentioned earlier, traditional information retrieval models primarily focus on keyword-based measures to find relevance between query terms and the documents in a collection. With the emergence of knowledge graphs and semantic-based ontologies, it has been shown that retrieval performance can be improved if semantic information is taken into account to determine relevance. Most existing work focus on developing semantics-enabled relevance models for retrieval, which have shown significant improvement over keyword-based retrieval models. With the positive impact of semantic information in retrieval, it is important to consider actual implementation practicality of these approaches. For instance, the SELM model proposed in [74] requires the pairwise calculation of the semantic similarity of entities available in the query and all documents of the collection. This model has shown strong retrieval effectiveness; however, little is known about its practical implementation details and how it can be operationalized. As such, our work is focused on building efficient indexing infrastructure for facilitating semantic information retrieval.

To this end, earlier works have adopted two strategies to be able to index semantic information. The first group of work have attempted to modify the structure of the inverted index to allow for the indexing of multiple heterogeneous information types, including keywords, entities and types [21, 18]. The second set of work adopt separate but interrelated indices to store

different information types [195, 126, 48]. Both of these works suffer in terms of query processing time where the first approach takes longer to find relevant documents as it has to distinguish between the heterogeneous information stored within the same index while the second approach requires multitude of index calls as well as the integration of the results from multiple indices. The main objective of our work is to develop a single index that can store keyword, entity and type information without the deficiencies of these already existing approaches.

The core idea of our work is based on three fundamental premises as follows:

1. In order to be able to index information within an inverted index without sacrificing query processing time, the index keys need to be of homogeneous nature. Therefore, we need to develop representations of documents, keywords, entities and semantic types, that will form the index keys, such that they are homogeneous and comparable. For this purpose, we propose to use the joint embedding of these four different heterogeneous information types within the same embedding space. The embedded representations of these information will be comparable and hence provide the homogeneity required by inverted index keys.

2. Each posting within the inverted index needs to store one additional measure of relevance for the indexed document to the related index key. The measure of relevance in semantic-based information retrieval is often based on some form of semantic similarity or relatedness [140,

102, 179]. Given the fact that we embed documents within the same embedding space as keywords, entities and types, it will be possible to calculate the similarity between each document and any of the three types of information through vector similarity, e.g. cosine similarity and Euclidean distance. This is possible because the documents, keywords, entities, and types are embedded within the same space by jointly embedding them.

3. Finally, traditionally within an inverted index, documents listed in the posting list of a given index key are guaranteed to consist of at least one mention of the key. In our work, however, we relax this requirement and do not require that all documents in the posting list have to necessarily contain the index key. Instead, we require that each posting list should include the top-k approximate nearest neighbors of the index key. Based on this requirement, it is possible that a document is listed in the posting list of a certain index key even if the document does not have the index key in it; but is, semantically speaking, more similar to the index key compared to those that actually contain the index key.

Based on these premises, Figure 3.4 presents the workflow of our work which consists of several steps: In the first step, we jointly learn embeddings for keywords, entities, types and documents within the same embedding space. Therefore, the vector representation of all this information is homogeneous and comparable; hence, they can all, if necessary, serve as keys for the same

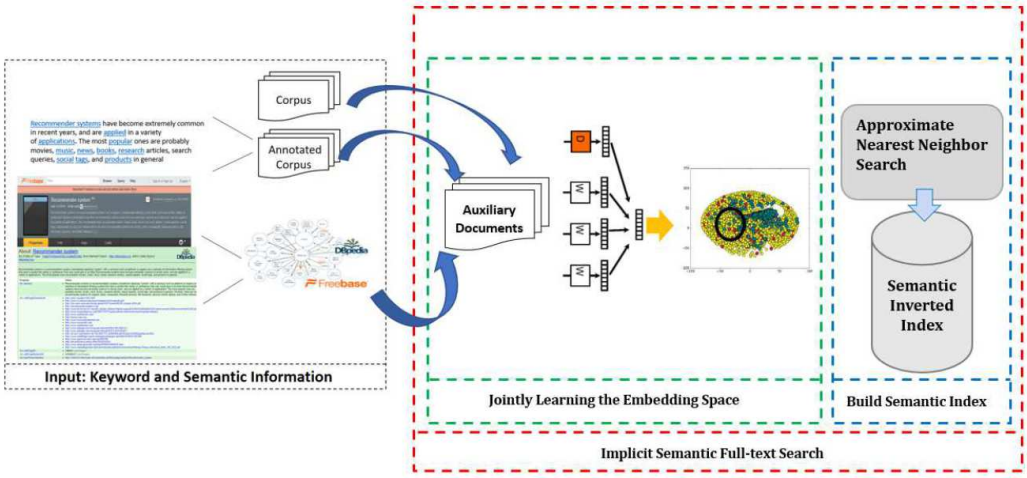

Figure 3.4: The workflow of the implicit semantic full-text index.

inverted index. For each keyword, entity and type observed in the document collection, we populate its posting list in the inverted index. In order to populate the posting list, we identify the top-k most similar documents that are represented as data points within the joint embedding space and add them to the posting list ordered by their degree of vector similarity. The top-k most similar documents are retrieved and identified using approximate nearest neighbor search. This process will result in an inverted index whose posting lists have the most k postings and each posting refers to a document that might not necessarily consist of the key of that entry in the inverted index but is guaranteed to be among the top-k most similar documents to the key. We provide the details of this process in the following sections.

3.2.1 Jointly Learning the Embedding Space

In order to jointly learn a vector representation for keywords, entities, types and documents, the first step is to identify and link textual documents with knowledge graph entities. In order to achieve this, we perform entity linking [85, 146] on each document in the corpus, as a result of which a set of relevant entities for the content of that document are identified and linked to some phrases in the document. We will later explain that we have used Freebase and DBpedia entities in our work. Once entities are identified for each document of the corpus, it is possible to find the type of the entity by traversing the type hierarchy within the knowledge graph. Depending on whether the links in the knowledge graph are traversed towards the children or the parents, super-types and sub-types of the immediate type of the entity can be retrieved. On this basis, we retrieve entity type information for the entities in each document. Now, given the annotated document, it would be possible to use paragraph vector models to jointly learn embeddings based on the whole collection. The main reason why we use paragraph vectors and not word vectors is the need to learn joint embeddings for all the types of information that is present including keywords, entities, types and documents. The use of paragraph vectors provides us with vector representations for all these elements in the same embedding space; hence, making them comparable to each other. As such it would be possible to compute the distance between documents, keywords, entities and types based on their vector representation without having to be concerned with them being different elements

because they are embedded systematically within the same space. However, before this can be done, we need to address the challenge that relates to the paragraph vector context window size.

As mentioned earlier, neural embedding models such as paragraph vector models define the context of a keyword in the form of a number of keywords seen before and after the keyword of interest. While this will work efficiently when dealing with keywords, it will not be directly applicable when additional information that did not originally appear in the document need to be considered. In this specific case, each annotated document now consists of an additional set of entities and their types that were not originally a part of the document and hence would not be included in the embedding process unless they are added to the document. In order to add them to the document, there are two considerations that need to be made: i) the position where the entities and types are added: This is important because the position where the entities and types are added will determine their neighboring keywords and hence form their context based on which the vector representations of the entities and types are trained. One approach would be to include the entities and entity types immediately after the phrase that is linked to the entity by the entity linking system. For instance, a document such as "Gensim is a robust open-source vector space modeling and topic modeling" is converted to "Gensim $/m/708mx$ $/m/126mx$ is a robust open-source $/m/1278q$ vector space$/m/498444q$ $/m/09731q$ $/m/171mx$ modeling and topic modeling $/m/393mx$" which now includes Freebase identifiers. This leads to the

second consideration: ii) once additional entity, and type information are added to the original document, keyword contexts are now different than they originally were. For instance, for a window size of three, the context for the keyword "Gensim" would have been "is robust open-source" (assuming articles are ignored), whereas the context of the same keyword in the revised document would be "is $/m/708mx$ $/m/126mx$".

To address these two issues, we generate multiple auxiliary documents for each of the original documents. In each auxiliary document, one of the annotated terms is replaced with its corresponding entity or entity type. For instance, one of the auxiliary documents generated for our earlier example would be "$/m/708mx$ is a robust open-source vector space modeling and topic modeling" where "Gensim" is replaced by its Freebase identifier. Another alternative auxiliary document would be " $/m/126mx$ is a robust open-source vector space modeling and topic modeling" where "Gensim" is replaced by its entity type. This way, entities and entity types are incorporated into the documents while respecting the context window size and also preserving the keyword neighborhood of the original document collection. It should be noted that the inclusion of auxiliary documents does not negatively skew the balance of the keyword co-occurrences because while the frequency of co-occurrences between keywords that appear in the same context increases as the number of auxiliary documents increases, the overall frequency of co-occurrences between all other co-occurring keywords also increases. This means that the frequency of all co-occurring keywords increases

Index Key	Similar Keywords and Entities
Obama	President, Mother, News, Iowa, Barack_Obama, African_Americans, American, Family
President	Chairman, Obama, United_States_Capitol, President_of_the_United_States, African_Americans
Game	Atari, Battle, Poker, Computer, Japanese, Anaheim, Korean, PlayStation_2
Poker	Casino, Tournament, Game, Internet, Texas, PlayStation_2
Music	Rock, Band, Song, Interview, California, Uranus

Table 3.3: Sample query terms and their most similar neighbors in the joint embedding space.

similarly due to the inclusion of additional auxiliary documents and as such while the frequency counts will have larger values, they are proportionally approximately similar to when auxiliary documents were not included.

Now, given the newly developed document collection that includes both the original documents as well as the newly added auxiliary documents that include entities and types, we learn vector representations using the Paragraph Vector model on this document collection. The learned vector representations will include embeddings for keywords, entities, types and documents as all of these are present as tokens in the newly created document collection. Essentially based on our new document collection, the embedding model does not distinguish between keywords, entities and types as they are all placed in the documents as tokens. Therefore, the paragraph vector model learns vector representations for documents and tokens consisting of keywords, entities and types. The learnt vectors for all four types of information are in the same space as required.

For the sake of depicting a few examples of how the keywords, entities and types are embedded within the same space, Table 3.3 provides some sample query terms and their nearest neighbors (including keywords, entities and types) in the embedding space. Our general observation from the derived embeddings was that narrow domain keywords, e.g., Obama and Poker, tend to have much more semantically similar neighbors compared to more general keywords such as "Game" and "Music".

3.2.2 Building Semantic Inverted Indices

One of the limitations of earlier work in building inverted indices in keywords of including both textual and semantic information is the *heterogeneity* of the index keys. We have used *PV* [129] to address this issue by embedding keyword, entity, and type information within the same embedding space; therefore, all these three types of information can be used as index keys in the inverted index. As such, it is possible to simply build an inverted index that would consist of one posting list for each keyword, entity, and type that has been observed in the document collection. According to the traditional method for populating the posting list related to each index key, the posting list consists of one posting per those documents where the index key has been observed at least once. As such, all documents of the posting list are guaranteed to contain at least one mention of the index key. In our work, we relax this requirement and allow documents to be listed in the posting list even if they do not explicitly contain the index key. The primary reason

for this is based on the empirical observations of relevance. The relevance judgements provided by human experts in the TREC collection, in some cases, contain relevant documents that do not include the query term that is being searched. For this reason, while the presence of the query term is a strong indicator of relevance, it does not necessarily mean that all the other documents that do not have the query term are irrelevant. There are cases where the document is related to the query terms but it does not contain the query terms explicitly. For instance, for a query such as "famous conspiracies", a document that talks about the Apollo moon landing would be relevant even if the keywords "famous" and "conspiracies" do not appear in the document.

The relaxation of the need to explicitly observe the index key allows us to benefit from the semantics embedded in the vector representation of documents, keywords, entities, and types. On this basis, we populate the posting list related to a given index key based on the similarity of the index key with the documents in the document collection.

Several studies have measured distance in the embedding space based on the Euclidean distance [124, 100, 186, 46]. For example, Trieu et al. [200] proposed a new method for news classification by using pre-trained embeddings based on Twitter content. The authors measure the semantic distance between two embedding vectors using three direct distance metrices L1, Euclidean distance and cosine similarity. Their experiments demonstrated that the semantic distance between two vectors based on Euclidean distance pro-

vides the best accuracy. Furthermore, most approximate nearest neighbor search algorithms support Euclidean distance; therefore, in our work, we adopt the inverse of the Euclidean distance between the vector representations of the index key and the document as the *measure of relevance*. For an index key $k = (k_1, ..., k_n)$ and a document $d = (d_1, ..., d_n)$ that are embedded in the same n-dimensional space, the relevance of d for k, $rel(k, d)$, is calculated as follows:

$$rel(k, d) = \epsilon + \left(\sqrt{(k_1 - d_1)^2 + ... + (k_n - d_n)^2}\right)^{-1} \tag{3.1}$$

Based on this vector-based measure, the relevance of each document to the index key can be computed. This will range from the most relevant, which would have a relevance of approximately infinity to the least similar which would have a relevance of near zero. For each index key, all documents in the corpus can be inversely ordered and included in the relevant posting list. Now, given the fact that a significant number of documents in the corpus are completely unrelated to an index key, it would not be reasonable to include all documents in each of the posting lists. For this reason, the top-k most similar documents can be selected to be included in each posting list.

Based on this strategy, the length of the posting list would depend on the size of the chosen k. In order to find the top-k most similar documents, it is possible to perform *approximate nearest neighbor search* [109] that significantly reduces the computational time of the similarity calculations. In our

work, we use LSH and random projections [109]. The approximate nearest neighbor search for every index key finds and retrieves k documents that are most relevant to the index based on vector similarity. As mentioned earlier, the size of each posting list can be at most k and the postings in each posting list are not guaranteed to contain the index key but are, with an accurate *approximation*, the most similar documents to the index key based on the learnt embeddings.

At the end we need to address how we handle two main limitations of word embeddings; word disambiguation and out-of-vocabulary (OOV) words. Our proposed model provides better model for representing distinct meanings of a word into a single vector. Because we built an embedded space of context, annotated context and KB and several studies [138, 79] show this combination improves the vector representation of ambiguated words. In addition to solve the OOV, we can aggregate the vector representation of subwords; for instance, the vector representation of the *university of new Brunswick* can be created by averaging word vectors of *university* and *new Brunswick*. If the OOV can not be subdivided to see words of our model, we can add the vector representation of its composing characters of the word. This method of creating vector representation of a word has shown improvement on Chines language and acceptable for English [45, 31].

3.3 Summary

In this chapter we have described the proposed indexing methods: the explicit semantic full-text index and the implicit semantic full-text index. In the former, we integrate textual and semantic information during query process time, while in the latter these information are integrated before building the index.

The explicit semantic full-text index aims to identify the most appropriate indexing data structures that can store and retrieve textual and semantic information efficiently and effectively. It consists of three indices: Keyword, Entity and Type Indices to store keywords, entities and types respectively. These indices are constructed by using three well known index data structures: Treap, Wavelet Tree and HashMap. To integrate these three indices during query process time, we offered different integration methods with regards to the data structures, and utilize the list-based integration approaches. The implicit semantic index utilizes neural models for integrating textual and semantic information in the same embedded space. The created space represents semantic relations between keywords, entities, types and documents. We then used approximate nearest neighbor search to find top-k related documents for all keywords, entities and types. Since, implicit semantic full-text index only stores the most k semantic similar documents for each index key. In the next chapter, we have described implementation details and evaluation methodologies for both of our proposed semantic full-text indices.

Chapter 4

Evaluation

Within the context of IR, evaluation is most generally performed through experiments on a standard test collection (e.g., TREC). Two core necessarycharacteristics of an IR system are its *Efficiency* and *Effectiveness*. The performance of the retrieval systems is measured by considering index storage space (index size) and query processing time. While, Effectiveness is evaluated by looking at how many relevant documents are retrieved for any given query. We explain more about different methods of evaluation for measuring Efficiency and Effectiveness in Section 2.1.4. In this section, we describe our experimental setup, implementation details and our finding for explicit semantic full-text index and implicit semantic full-text index respectively.

4.1 Explicit Semantic Full-Text Index

In this section, we first describe our experimental setup and implementation details. Then, our findings about efficiency and effectiveness of the explicit semantic full-text index are presented. However the main focus of evaluation is on finding the most efficient and practical data structure based on:

1. the relation between indexing data structures and query process time;
2. the impact of different integration models on query process time;
3. the effect of query expansion using semantic information on query process time.

4.1.1 Experimental Setup

We choose 5 million random documents of the English-language Web pages from the ClueWeb09 corpus ($ClueWeb09\_English\_1$), which contains $7,910,158$ different keywords in the vocabulary. Freebase annotations of the ClueWeb Corpora (FACC1) [88] are used as our annotation text to extract entities and types of the selected documents. There are $1,112,566$ entities and $1,302$ types in the selected documents. We use the *Million query track 2009 queries* dataset since they are annotated based on Freebase to evaluate the query process times. The queries have different number of terms and entities from a range of 1 to 8 terms (entities/keywords).

We use the abbreviated name for all indices, which are presented in Table 4.1. The first column and first row of the table specify the data structure of the Entity Index and Keyword Index, respectively. As an example of how this table can be interpreted, TH is the abbreviation of a heterogeneous index, which consists of the Treap-based Entity Index and HashMap-based Keyword Index.

		Keyword Index		
		Treap	HashMap	Wavelet Tree
Entity Index	Treap	TT	TH	TW
	HashMap	HT	HH	HW
	Wavelet Tree	WT	WH	WW

Table 4.1: The Abbreviation of Indices

4.1.2 Implementation

Explicit semantic full-text index consists of three indices: Keyword Index, Entity Index and Type Index; mentioned in Section 3.1. To build an efficient and effective explicit semantic full-text index, we implement three versions of it by considering Treap, Wavelet Tree and HahsMap as data structures for Keyword, Entity and Type Index. The justification for selecting these data structures is mentioned in Section 2.2.1.

Our experiments were performed on a 2.1 GHz Six -Core Intel 2 x Xeon E5-2620V2 CPU running Ubuntu 12.04 with 64.00 GB of RAM. All imple-

mentations are done in Java. We define Treap and Wavelet Tree structures in Java to provide all required utilities which are explained in Sections 3.1.1.1 and 3.1.1.2 for these two types of indices respectively. The functionality for these data structures are defined based on what they need to do for processing queries regarding the statements in Section 3.1.2.1 for Treap and Section 3.1.2.2 for Wavelet Tree. We use HashMap from Java library to implement the inverted index[1].

The data flow diagram illustrated in Figure 4.1 represents how the explicit semantic full-text index is constructed and how it would process queries. To build the explicit semantic full-text index we required a document collection that was annotated. We also need to have access to the KB properties which is used for annotating documents and queries. In this section we briefly describe each process of the data flow diagram. ***Text Transformation***is one of the main steps for performing document processing for an IR system (refer to Section 2.1.1). This process in our implementation consists of:

- Removing stop words which are specified with lumer indri
- Stemming based on porter stemmer of snowballstem [2]
- Parsing HTML page with jsoup (version 1.9.2) [3].
- Tokenizing text by extracting tokens from text and make sure all tokens

[1]https://docs.oracle.com/javase/8/docs/api/?java/util/HashMap.html
[2]http://snowballstem.org/
[3]https://jsoup.org/

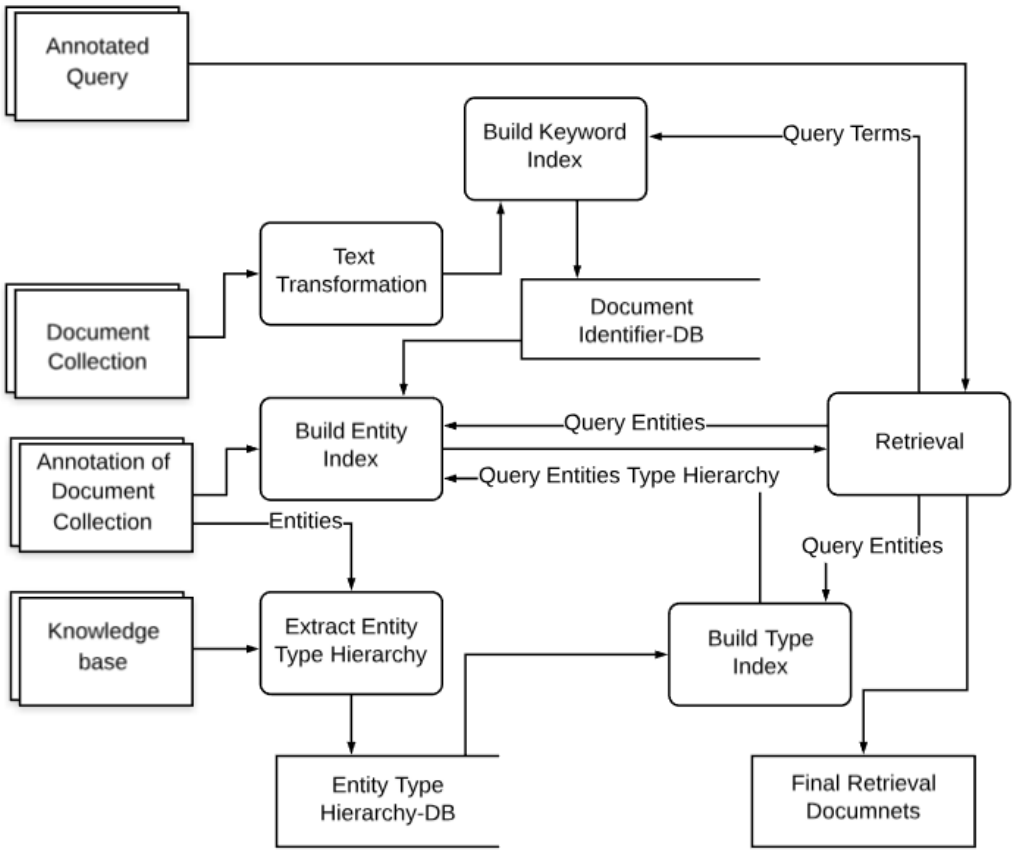

Figure 4.1: The data flow diagram of explicit semantic full-text index.

are valid keywords (index vocabulary), we use OpenNLP [4] to perform this task.

The output of this process is given to the ***Build Keyword Index*** process to build the Keyword Index with regards to the data structure of posting list based on the strategies we described in Section 3.1.1.

In ***Build Entity Index*** process, we scan *the annotation of document collec-*

[4]http://www.java2s.com/Code/Jar/o/Downloadopennlptoolsjar.htm

tion to store all mentioned entities of the corpus. As mentioned in Table 3.1; for each entity we need to store $docId$, TF and CV. In Implementation of the Treap-based Entity Index, we need to convert these three values to two values, because Treap can only save two values: key value and priority value, in each node. So, in our implementation the TF and CV are combined based on the method explained in Section 3.1.1.1 since both of them are used as weight parameter of retrieval algorithms. Therefore, all required information of Entity Index can be stored in a Treap. We use this combination technique for these two values when implementing Wavelet Tree-based Entity Index because storing these values separately with Wavelet Tree requires building more Wavelet Trees which consequently consumes more space and time.

Extract Entity Type Hierarchy process has two inputs: KB and annotation of document collection. It uses these information to create the required information of the proposed Type Index (refer to Section 3.1). To fulfil the goal of this process, we performed the following tasks:

- Find all properties in KB whose object or subject is the mentioned entities of the corpus when their properties are *is-a*.
- Store the extracted RDFs in a table; it is called *Entity Type Hierarchy* table.
- Extract the relation between entities and their hierarchical types (e.g. super-type and sub-type) based on the RDFs were stored in the Entity Type Hierarchy table.

The Entity Type Hierarchy table contains required information for building Type Index. Accordingly, the only input of the ***Build Type Index*** process is this table. Type Index stores these structural information about entity types to provide capability of reasoning on entities and their types during the retrieval process, e.g., to make *query expansion*.

The ***Retrieval*** process aims to answer queries and retrieve final documents. This process consists of four tasks:

- Sending query terms and query entities to the Keyword Index and Entity Index respectively.
- Applying a query process algorithm on these indices and retrieve the results.
- Integrating the results of these two indices to retrieve final results documents.
- performing query expansion if necessary by running the query entities against the type.

Query processing algorithms of Keyword Index and Entity Index depend on the data structure of the index and type of the queries (e.g., top-k or Boolean). We implement all query processing algorithms that are related to these indices, presented and described in Sections 3.1.2.1and 3.1.2.2 for Treap and Wavelet Tree, respectively. To retrieve result for the HashMap based index, we use the *list* algorithm for answering ranked and Boolean queries taking into account the explanation in Section 3.1.2.2.

We implement query processing algorithms of Wavelet tree-based Type Index and HashMap-based TypeIndex based on the explanation in Section 3.1.2.4. Note that, Treap-based Type Index is not applicable for the proposed Type index as explained in Section 3.1.1.

After processing queries of all indices, the results of Entity and Keyword Indices are *integrated* to retrieve the final results. To find best data structure for building explicit semantic full-text index, we need to consider all possible permutations of these two indices with respect to their data structures. Moreover, integration of the two lists algorithms are studied to show the effect of the integration process regardless of the index data structure. We implement all *Homogenous Integration* and *Heterogeneous Integration* algorithms based on the description in Section 3.1.3).

Explicit semantic full-text index is applicable to any document collection which is annotated and any KB as the input. This index can answer any term query (refer to Section 2.3) as well as annotated query. However, most users looking for top-k intersection retrieval results. Fortunately, this index also has the capability for answering Boolean intersection, Boolean union and top-k union queries.

4.1.3 Efficiency of the Explicit Semantic Full-Text Index

As mentioned earlier, an IR system is evaluated based on efficiency and effectiveness metrics. In an indexing method, efficiency metrics are concerned with storage space (index size) and query processing time. To evaluate the efficiency of explicit semantic full-text index; first, the space complexity of Treap, Wavelet Tree and HashMap are compared in big-O notation. As mentioned in Section 3.1.3 the integration of textual information and semantic information is performed during processing queries. Therefore, the main focus of evaluation of the explicit semantic full-text index is on measuring efficiency based on query process time. As such, we compare the query process time of ranked queries and Boolean intersection queries for all homogeneous indices and heterogeneous indices. In all figures presented in this section, the number of entities and keywords of the queries are clearly mentioned to show their impact on query process performance. These results are compared based on sensitivity to query length along with the effect of modifying the number of entities and keywords in the queries. Also, we evaluated the effect of top-k results on query process time for ranked queries by changing the value of k from 10 to 20 for all the indices.

4.1.3.1 Memory Usage of the Explicit Semantic Full-Text Index

Traditionally posting lists were stored on disk, so, reducing index size means reducing transfer time and improving query process time. The availability of large main memories solves this issue because now the whole inverted index can be stored in the main memory of one or several machines [117]. But, index size is still a very important efficiency feature of an IR system. Because reducing memory usage of an index means providing ability for a single machine to store larger collections which is very essential for limited-memory devices (e.g., cellphone). Furthermore, an index with smaller size means reducing the number of required machines to store the index, saving energy and decreasing the query process time since the amount of communication between machines is reduced.

In order to perform efficiency evaluation of the proposed indices, we compare the memory usage of Treap, Wavelet Tree and HashMap based on their space complexity. We do not perform any experimental evaluation on index size. We only add to the number of indices in the proposed approach for building the explicit semantic full-text index. Thus, the approximate index size of this approach is just $3 \times O(datastructurespace)$.

To compute space complexity of Treap, Wavelet Tree and HashMap -based Indices; we assume:

- the number of documents is d.
- all posting lists contain n documents (upper bound).

- size of vocabulary in each index is v.

In the indexing context, the order of these three parameters is $n \leq d < v$.

Space Complexity for Treap-based Indices

To compute the space complexity of Treap based indices we need to compute the space complexity representation of: i) Treap topology which is the data structure of each posting list, ii) each node which consists of pointers, docIds and TF values. Representation of a Treap topology is similar to the representation of any general tree. Hence, there are $\Theta(\frac{4^n}{n^{\frac{3}{2}}})$ general trees of n nodes. So, we need $\log_2(\frac{4^n}{n^{\frac{3}{2}}}) = 2n\text{-}\Theta(\log n)$ bits to represent any such tree. It has been proven that the compact representation of tree [174, 8] uses $2n + O(n)$ bits to represent a tree that can perform many tree operations efficiently (e.g., taking the first child and computing postorder of a node). Furthermore, each node of a Treap has:i) three pointers, ii)a key value, and iii)an attribute value. Hence, the memory usage of a node is constant. Accordingly, the space complexity of representing a node is $O(C)$ and space complexity for representing n nodes is $O(n)$. Therefore, the space complexity of a Treap- based Index is:

$$v \times 2(n + O(n)) \tag{4.1}$$

Space Complexity for Wavelet Tree-based Indices

As mentioned in Section 3.1.1.2, to represent Wavelet Tree-based Keyword Index and Wavelet Tree-based Entity Index, each posting list is divided into

two lists of docIds $S_i[1,n]$ and a list of TFs $W_i[1,n]$ without changing their order. DocIds lists S_i of all index vocabulary are concatenated to generate input sequence of Wavelet Tree $S[1,(v \times n)]$. All W_i lists are concatenated based on the order of S_i lists in S to generate the $W[1,(v \times n)]$ sequence. Furthermore, we use a bitmap to store the starting position of each S_i in S. Therefore, to compute the space complexity of Wavelet Tree-based Keyword Index and Wavelet Tree-based Entity Index, we need to consider the number of bits for: i) storing in a Wavelet Tree, ii) representation of its topology, iii) storing TFs, and iv) storing the starting position of each S_i.

We assume a Wavelet Tree, where the length of its input sequence is m and its alphabet size is d. The Wavelet Tree height, number of internal nodes and leaves are $\lceil \log d \rceil$,$d-1$ and d respectively.

By Traversing this Wavelet Tree level by level, it is not hard to see that, exactly $v \times n$ bits are stored at each level. So, at most the total number of bits it stores is $(v \times n)\lceil \log d \rceil$(upper bound) since the last level has at most $(v \times n)$ bits. Claude et al. [50] proved, if we only use one bitmap to represent the topology of a wavelet tree, then there is no need to use pointers. Thus, the total space to efficiently implement the Wavelet Tree become $(v \times n)\lceil \log d \rceil + O((v \times n)) \log d$ [158]. Storing the start position of S_i in S with a bitmap requires $O((v \times n))$ bits. Consequently, the space complexity of the Wavelet Tree-based Keyword Index and Wavelet Tree-

based Entity Index is:

$$(v \times n)\lceil \log d \rceil + O((v \times n)) \log d + O((v \times n)) \tag{4.2}$$

In the implementation of Wavelet Tree-based Type Index, we build three Wavelet Trees for indexing Subtypes, Supertypes and Entities (instances of a type) separately. To build the Subtypes Wavelet Tree, we assume, $Sb_t[1, n]$ is a list of all subtypeIds of a type t. Then, all lists Sb_i for all types in the corpus are concatenated into a unique list $Sb[1, ((v \times n) + v)]$. To know the boundary of a list Sb_i in Sb, we insert a 0 at the end of each Sb_i in Sb. Therefore, the space complexity of Subtypes Wavelet Tree based on the above statements is:

$$((v \times n) + v)\lceil logv \rceil + O((v \times n) + v) \log v \tag{4.3}$$

We implement Supertypes Wavelet Tree and Entities Wavelet Tree the same way as the Subtypes Wavelet Tree is built. Hence, the space complexity of them is equivalent to the space complexity of Subtypes Wavelet Tree (refer to Equation4.3).

Space Complexity of HashMap-based Indices

Needless to say, the space complexity of HashMap is $O(v)$[52]. But, the exact space complexity of a HashMap depends on i)the hashing function, and ii)the type of the keys and the values. For instance, in Java 7, HashMap uses an inner array of Entry. An entry has:

- a reference to a next entry
- a precomputed hash (integer)
- a reference to the key
- a reference to the value

Assuming, a HashMap contains v elements and its inner array has a capacity/size C. The space complexity of this HashMap in Java 7 is approximately:

$$sizeOf(integer) \times v + sizeOf(reference) \times (3 * v + C) \tag{4.4}$$

Consequently, the space complexity of Treap, Wavelet Tree and HashMap is *linear*. This agrees with our expectations, since all of them are well known index data structures; as mentioned in Section 2.2. Overall, Wavelet Tree has the worst memory usage compared to Treap and HashMap because its space complexity in big-O is $O((v \times n)$ while HashMap is $O(v)$ and Treap is $O(n)$. It is clear that the, memory usage of Treap is less than the others.

4.1.3.2 Query Process Time of Homogeneous Indices

As will be shown in this section, while query process time increases as query length is increased, the growth rate of the query process time is different between the various homogeneous indices. The number of entities and keywords of queries do not affect the query process time of these indices. For example, the query process time is very similar for queries with three entities and one

keyword and queries with one entity and three keywords. We will explain more about the effect of these features for ranked and Boolean intersection queries.

Ranked Intersection

The ranked intersection query process time of the three homogeneous indices, namely HH, TT and WW, are presented for $k = 10$ and $k = 20$ in Figure 4.2. The query process time increases when the query length increases independently of the number of keywords and entities of the query. The sensitivity of the homogeneous indices to query length ranging from least sensitive to most sensitive is TT, HH and WT. Treap is less sensitive to query length, which is contrary to the results mentioned in previous work [121]. The reason for our observation may be that we employ these data structures in an integrated approach where information in the Keyword and Entity Indices are integrated to prepare the final search result; while in [121] the data structures are tested solely on a Keyword Index.

The query process time of TT, HH and WW increases when the value of k increases. WW is more sensitive to query length than the other indices. Figure 4.2 shows that the query process time of TT is more sensitive to k than HH and WW as noted in [16, 20] given the fact that the rate of increase of the query process time by changing k from 10 to 20 in TT is larger than HH and WW. Also, TT is still the fastest choice for retrieving ranked intersection results among the other indices. The main reason, which

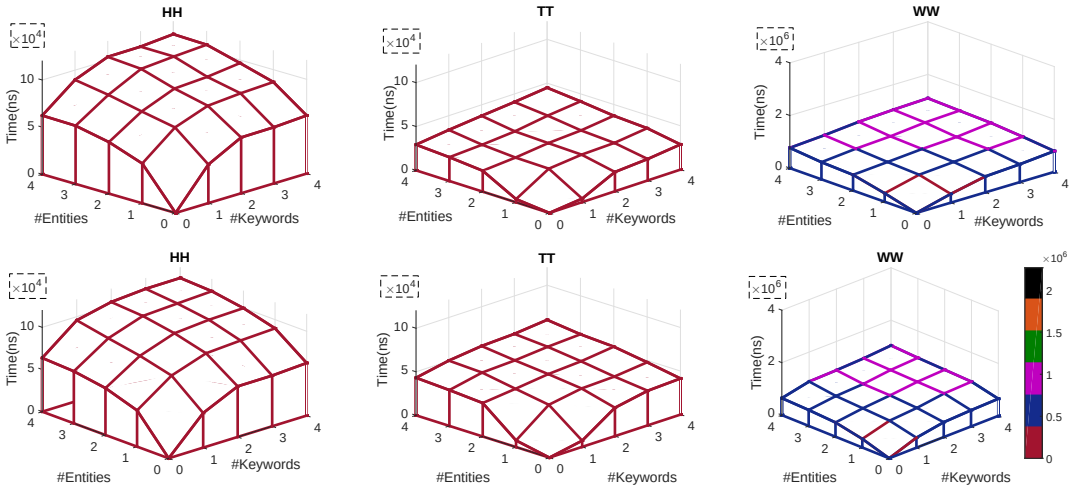

Figure 4.2: Time performance for ranked intersection for varying number of entities and keywords for k = 10 (top row) and k = 20 (bottom row).

causes the difference between TT and HH is their ranked retrieval algorithms. The Treap ranked intersection algorithm just finds the top-k documents by skipping documents whose weights are lower than the threshold (calculated based on the current top-k candidate set) [121] which is completely different from retrieving the k^{th} highest documents after performing a full Boolean intersection (Section 6.1). Also, this algorithm explains why there is only a small difference between the query process time of HH for $k = 10$ and $k = 20$.

Ranked Union

The results of the query process time of ranked union queries when $k = 10$ and $k = 20$ are presented for HH, TT and WW in Figure 4.3. HH is the most efficient data structure among all indices for processing ranked union queries. Also, these results show the WAND algorithm is a practical algo-

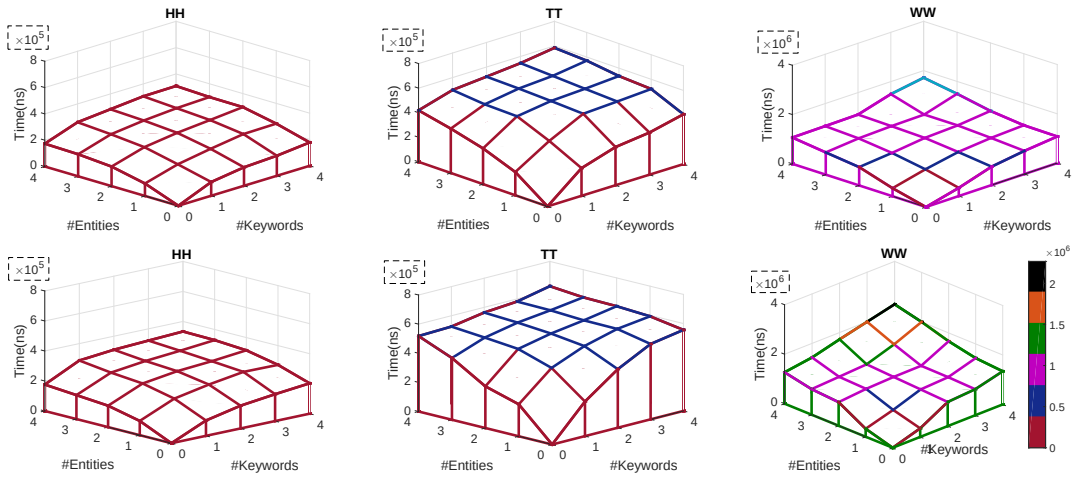

Figure 4.3: Time performance for ranked unions for varying number of entities and keywords in the query and using k = 10 (top row) and k = 20 (bottom row).

rithm for processing this type of query. The ranked union query process time of HH is around one third of that for TT. Also, it is less sensitive to query length compared to TT and WW. Changing the value of k from 10 to 20 has the least effect on the query process time of HH. However, query process time for WW significantly increases with the increase in the value of k.

Boolean Intersection

Figure 4.4 shows the query process time of Boolean intersection queries for all homogeneous indices. The query process time of TT is less than those of the other indices. WW has the worst query process time on Boolean intersection since the method of calculating and retrieving TF of a document in Wavelet Tree is much more time consuming compared to the other data structures.

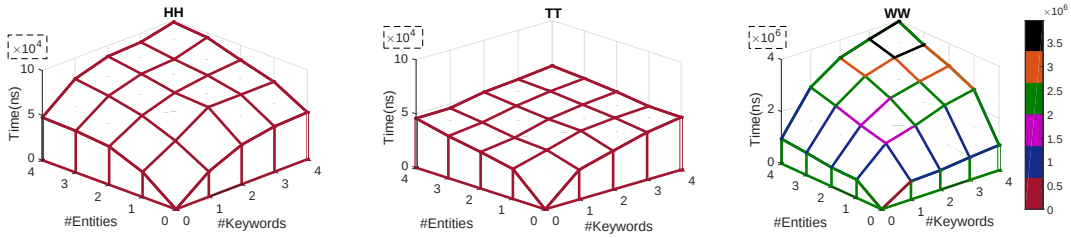

Figure 4.4: Boolean intersection process time for queries, which contain zero to four keywords and entities so the query length can be between 1 to 8.

Boolean intersection processing time is higher than ranked intersection for all data structures except the HashMap structure because of the HashMap ranked intersection algorithm as explained earlier in the ranked intersection section.

The Boolean intersection time of WW is much higher than the query process time of its ranked intersection and ranked union counterparts. The main reason is that the Wavelet Tree ranked query algorithm follows the strategy of *early stop* during query processing. Also, the number of results is significantly larger than k and therefore more weights need to be computed to prepare the results of Boolean intersection queries.

4.1.3.3 Query Process Time of Heterogeneous Indices

To find the most efficient data structure for our Semantic Hybrid Index among Treap, Wavelet Tree and HashMap, we evaluate the query process time of all heterogeneous indices as presented in Table 4.1. The query process times of Boolean intersection, ranked intersection and ranked union when k is 20 are shown in this section. All the results are presented based on the

list-based approach, which retrieves the final result documents more rapidly than the non-list-based approach according to the results in Section 4.1.3.4.

Ranked Intersection

Figure 4.5 shows the query process time of all heterogeneous indices for ranked intersection. The indices built by combining Treap and HashMap (HT, TH) are more efficient in terms of processing time than other heterogeneous indices. The query process times of HT and TH increase when the length of queries increase. However, TH processes the ranked intersection queries faster than HT; thus the most efficient heterogeneous index for processing ranked intersection queries is TH.

The next most efficient group of heterogeneous indices is built by combining Wavelet Tree and Treap. The effect of increasing the number of entities in queries on the query process time for WT is very small compared to the consequences of increasing the number of terms in queries. Therefore, the query process time of WT is more sensitive to the number of terms in queries. Instead, the query process time of TW is more sensitive to the number of entities in the queries. These two results show the effect of Wavelet Tree index on the query process time. TW does not retrieve results as fast as WT since the search time of a Wavelet Tree is $O(\log \sigma)$ where σ is the size of the alphabet and the alphabet size of the Wavelet Tree-based Keyword Index is larger than the size of the alphabet in the Wavelet Tree-based Entity Index. The query process time of ranked intersection for WH and HW is much larger

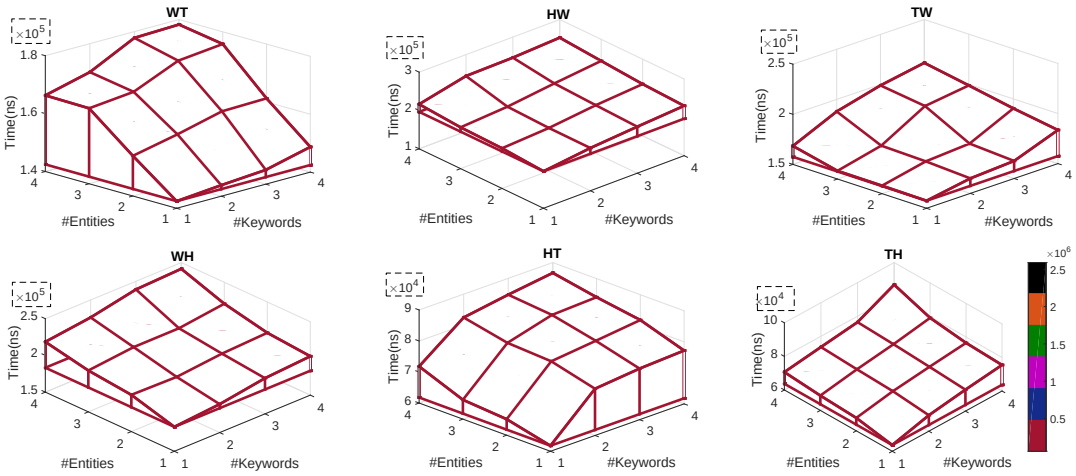

Figure 4.5: The query process time of ranked intersection for all heterogeneous indices.

than that of other heterogeneous indices. The query process time of these two heterogeneous indices is sensitive to the query length. However, the effect of the number of entities is larger than that of the number of keywords in HW. Also, the increase in the number of keywords in queries has more effect on query process time of WH compared to the increase in the number of entities. This is due to the data structure of the Keyword Index and Entity Index in HW and WH, where Wavelet Tree is the most sensitive index to the query length according to the result of Section 4.1.3.2. The query process time of HW is larger than that of WH because the size of the alphabet for Wavelet Tree as a Keyword Index is larger than that of the Entity Index and query process time of Wavelet Tree has a direct relation with alphabet size. For the same reason, the query process time of TW is larger than that of WT.

Ranked Union

The results of evaluating query process time of ranked union for all heterogeneous indices are presented in Figure 4.6. The order of efficiency of all indices for ranked union is the same as ranked intersection. All heterogeneous indices are more sensitive to the query length compared to ranked intersection according to the results presented in Figures 4.6 and 4.7. The query process time of ranked union for HashMap is less than Treap based on Figure 4.3. This observation is also confirmed by the results of HT and TH in Figure 4.6. Thus the growth of the number of keywords in queries does not increase the query process time of TH as much as the growth of the number of entities in the queries. In contrast, the growth of the number of keywords compared to the growth of number of entities in the queries has more impact on query process time of HT. In conclusion, TH is the most efficient heterogeneous index in terms query process time for ranked union queries.

Boolean Intersection

Figure 4.7 presents the query process time of Boolean intersection queries for all heterogeneous indices. The most efficient heterogeneous index for processing Boolean intersection queries is the same as ranked queries. Therefore, TH is the most efficient heterogeneous index for all types of queries. The query process time of HT is larger than TH similar to the query process time of ranked queries. The sensitivity of TH to the number of keywords in the query is more than entities while HT is more sensitive to the number of entities in the query. These relations confirm that HashMap is more sensitive

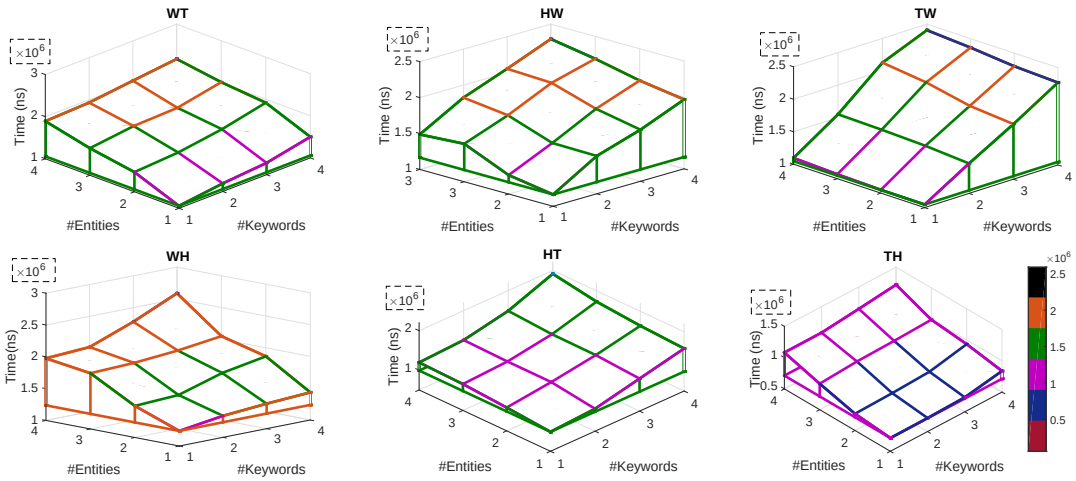

Figure 4.6: The query process time of ranked union for all heterogeneous semantic hybrid indices.

to the query length compared to Treap according to Section 4.1.3.3.

WH and HW process Boolean intersection queries faster than TW and WT, which is in contrast to the observations for ranked queries. HW is always more efficient than WH because the Wavelet Tree search time depends on the size of alphabet as discussed earlier. The query process time of HW and WH increases when the length of the queries increases independently of the number of keywords and entities in the queries.

The combination of Treap and Wavelet Tree creates the heterogeneous indices with the largest query process time for processing Boolean intersection queries. The effect of increasing the number of entities and keywords on query process time is opposite for WT and TW because of the Wavelet Tree index since the process time of the Wavelet Tree Index is larger than Treap Index according to the result of Figure 4.4. The query process time of WT

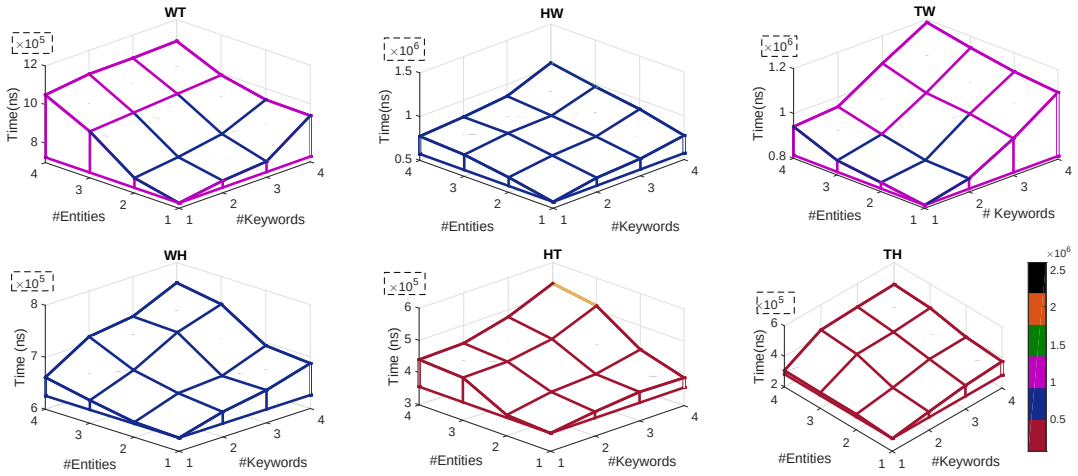

Figure 4.7: The query process time of Boolean intersection for all heterogeneous indices.

increases rapidly when the number of entities in the query increases; in contrast the query process time of TW increases quickly when the number of keywords in the query increases.

4.1.3.4 Comparing Homogeneous and Heterogeneous Indices

To find the most efficient indexing data structure, we compare the query process time of the most efficient heterogeneous index (list-based TH) with all homogeneous indices for ranked union queries, ranked intersection queries and Boolean intersection queries. Figure 4.8 presents the difference between the query process times of all homogeneous indices and list-based TH for processing ranked intersection queries. The results of these comparisons show the TT is the most efficient data structure for processing this type of queries; however, the query process time of list-based TH is significantly smaller than

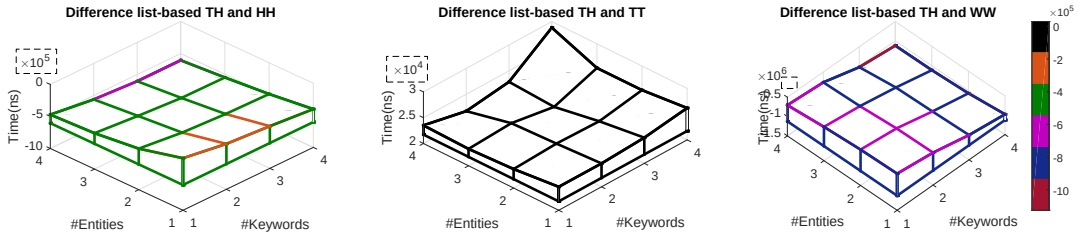

Figure 4.8: The difference (delta) between the query process time of the ranked intersection of homogeneous indices and list-based HT (the most efficient heterogeneous index).

that of HH and WW. The difference between query process times of TT and list-based TH is relatively small compared to the others; and the absolute difference between the query process times of HH and WW increases when the query length increases.

The differences between query process times of all homogeneous indices and the list-based TH for processing ranked union queries are presented in Figure 4.9. The list-based TH is not as efficient as TT and HH; also, the difference becomes more noticeable with the increase in query length. The difference between query process times of ranked union of list-based TH and HH increases faster than that of list-based TH and TT when the query length increases. This means that the most efficient data structure for processing ranked union queries is HH.

Figure 4.10 shows the difference between the query process time of list-based TH and all homogenous indices for processing Boolean intersection queries.

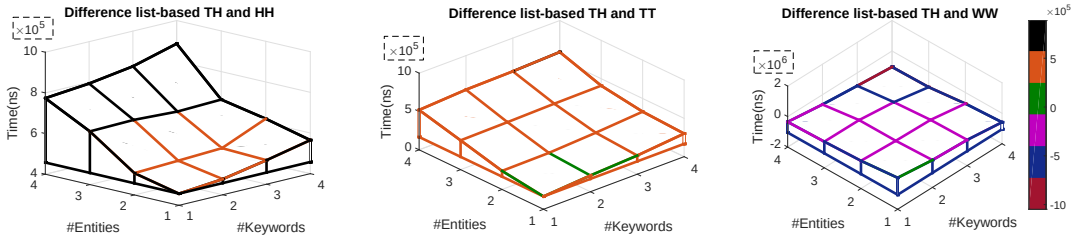

Figure 4.9: The difference (delta) between query process time of ranked union of homogeneous indices and list-based TH (the most efficient heterogeneous index).

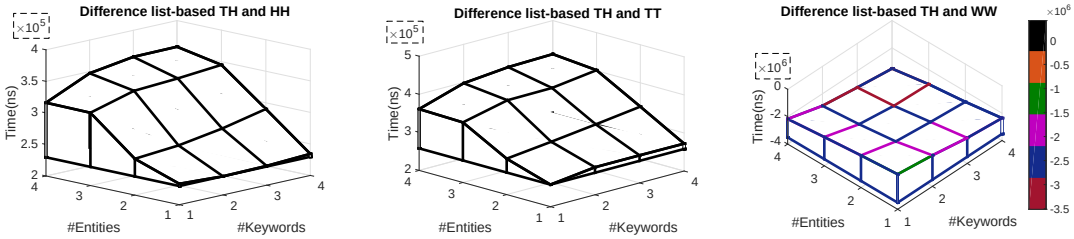

Figure 4.10: The difference (delta) between query process time of Boolean intersection queries of homogeneous indices and list-based TH (the most efficient heterogeneous index).

TT and HH process Boolean intersection queries faster than list-based TH. Based on these results, it can be concluded that the most efficient data structure for processing Boolean intersection queries is TT.

4.1.3.5 Effect of Integration Methods on Query Process Time

Integrating the Entity and Keyword Indices to retrieve the results of a query is achieved based on two approaches: list-based and non-list-based techniques. The effect of integration approaches on homogeneous indices is shown

in Figures 4.2, 4.3 and 4.4 for ranked queries and Boolean intersection queries. The *Treap integration* approach is more efficient for ranked intersection and Boolean intersection since the TT processes queries faster than other homogeneous indices. However, the list-based approach for ranked union queries retrieves results faster than the TT approach. The largest query process time for all types of queries belongs to WW independent of the type of the integration approach according to the results of Section 4.1.3.2. For instance, Figure 4.4 shows that the query process time of Boolean intersection queries for WW is significantly larger than that of other homogeneous indices.

To find the most efficient approach for integrating heterogeneous indices, we present the difference (delta) between query process times when either the list-based and non-list-based approaches are applied for processing ranked union queries, ranked intersection queries and Boolean intersection queries for all heterogeneous indices as shown in Figures 4.11, 4.12 and 4.13. We compute the difference by subtracting the query process time of the non-list-based approach from the list-based approach. The differences of all indices in Figures 4.11, 4.12 and 4.13 are calculated based on this method. We present only the results of ranked queries when $k = 20$ since we want to compare the effect of list-based and non-list-based approaches regardless of the value of k and under worst-case scenario.

Figure 4.11 presents the difference between the query process time of ranked intersection queries based on both the non-list-based and list-based approaches for all heterogeneous indices. The query process time of these indices based

on the list-based approach is significantly less than that of the non-list-based approach. The significant difference between these two approaches occurs when the data structure of one of the indices is Wavelet Tree, especially when it is the data structure of the Keyword Index because the search time of the Wavelet Tree has a direct relation with the size of the alphabet (see Section 4.1.3.3).

The significant difference between TH and HT indices mainly results from building a Treap during processing the query when the data structure of one of the indices is Treap in the non-list-based approach. The difference between all heterogeneous indices increases as the query length increases. Note that, increasing the number of entities or keywords does not affect query process time as long as the lengths of the queries are not changed for TH, HT, WH and WT. Finally, the largest difference between the non-list-based and list-based approaches belongs to TW and the smallest difference for these two approaches belongs to HT.

We show the difference between the query process time of ranked union queries based on non-list-based and list-based approaches in Figure 4.12. The difference between non-list-based and list-based approaches increases along with the increase in the query length for HW, TW, WH and HT. The increase of query time of TW and TH is significantly higher than those of HW and WH. In contrast, the difference between WT and TH is increased when the number of entities and keywords in the queries increases, respectively. The smallest difference belongs to the combination of the HashMap and Wavelet

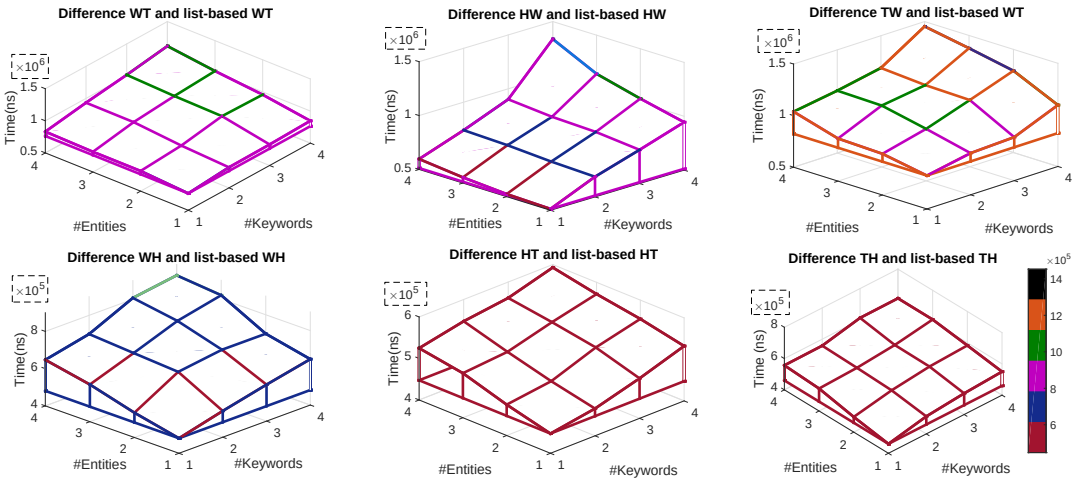

Figure 4.11: The difference (delta) between the process times of ranked intersection queries ($k = 20$) based on the non-list-based and list-based approaches for all heterogeneous indices.

Tree, which means the integration approach is not the issue of the higher query process time for HW and WH. Also, these results show the query process time for the non-list-based approach with the help of a Treap for ranked union is significantly larger than the list-based approach.

Figure 4.13 shows the difference between the non-list-based and list-based approaches for processing Boolean intersection queries. These results show the largest difference between non-lists based and list-based integration occurs for processing Boolean intersection queries when one of the index data structures is a Treap. The difference between non-list based and list-based integration of HW is smaller than zero when the number of entities is less than or equal to the number of keywords in the query.

The differences between WT, TW, HT and TH increase sharply when the

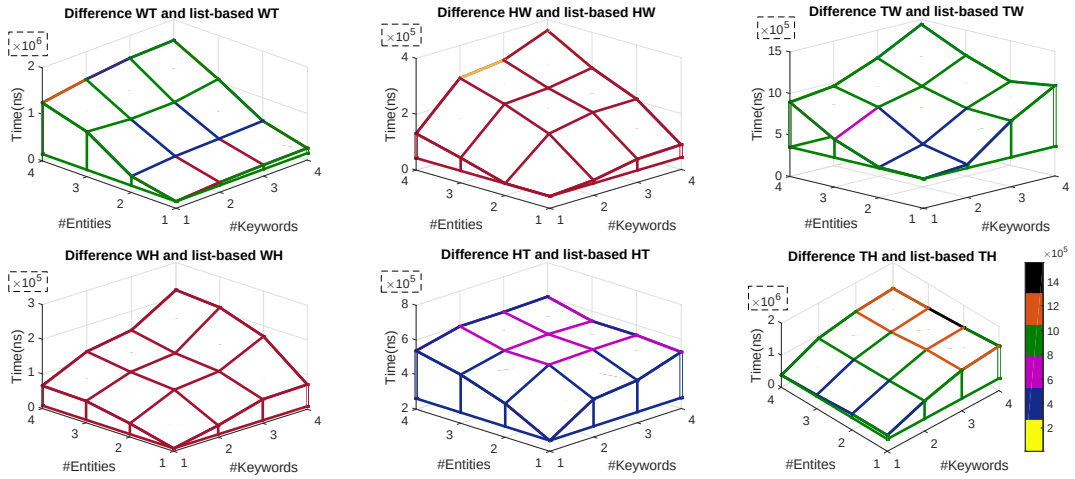

Figure 4.12: The difference (delta) between the process times of ranked union queries ($k = 20$) based on the non-list-based and list-based approaches for all heterogeneous indices.

length of the query increases. The different number of entities and keywords in queries, as long as the length of the queries is fixed, does not have any effect on the difference between the integration approach for HT and TH. However, the difference between non-list-based and list-based approaches increases significantly when the number of keywords and entities is increased in queries of WT and TW, respectively due to the Wavelet Tree data structure. By studying the difference between the query process time of the integration approaches, we conclude that the list-based approach is more efficient than the non-list-based approach especially for the TW and WT indices. Since these two indices have the largest difference for all types of queries. In contrast, the smallest difference belongs to the WH and HW indices for all types of queries and shows the integration process is not the main reason for having

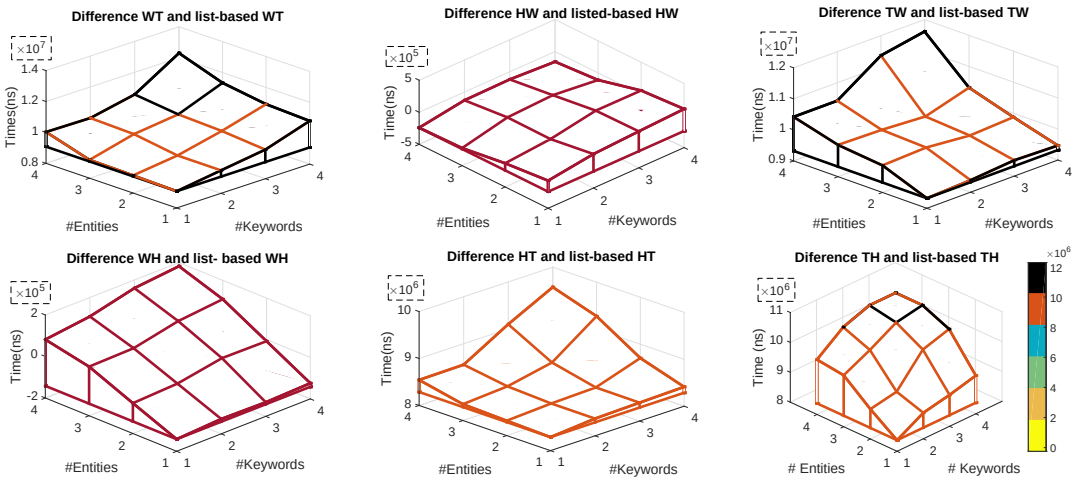

Figure 4.13: The difference (delta) between the process times of Boolean intersection queries ($k = 20$) based on the non-list-based and list-based approaches for all heterogeneous indices.

the largest query process time based on results in Section 4.1.3.3.

4.1.3.6 Query Expansion

To efficiently support query expansion, we need to identify the data structure that would provide the fastest lookup function for the Type Index. The main function of query expansion is done using the lookup function where the type of each query entity is found and then the posting list(s) of the query type(s) must be retrieved for expanding the query. Based on the discussions in Section 3.1.1.1, the Treap data structure does not have the ability to be considered as a Type Index data structure. Therefore, the lookup times of the HashMap-based Type Index and Wavelet Tree-based Type Index are compared to find the most efficient Type Index structure. The lookup times

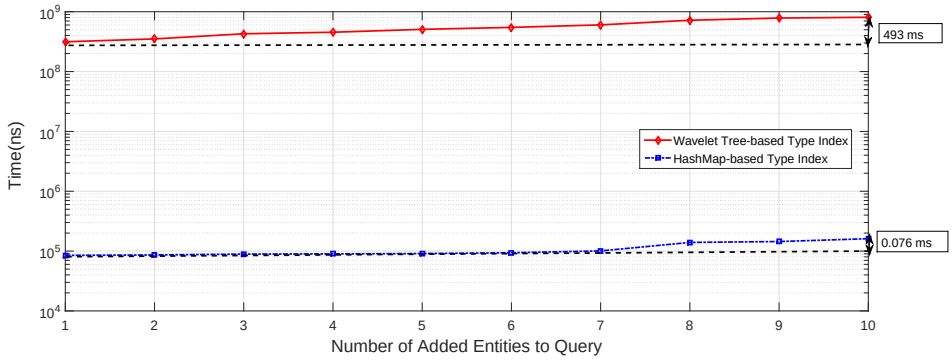

Figure 4.14: Entity Type lookup in HashMap-based Type Index compared to the Wavelet Tree-based Type Index.

of these indices are shown in Figure 4.14 for varying numbers (1 to 10) of entities within queries. The amount of time added to the query process time is very stable for the HashMap data structure. Although the lookup time of Wavelet Tree data structure is much higher than HashMap, the difference between adding one entity or 10 entities to the query based on the Wavelet tree-based Type Index is just 493 milliseconds.

4.1.4 Effectiveness of the Explicit Semantic Full-Text Index

Effectiveness can be measured in terms of metrics such as MAP and $nDCG@K$, which primarily focus on whether relevant documents are placed at the top of the retrieved list or not, stated in Section2.1.4. Intuitively, a better retrieved list would be the one that consists of relevant documents to the query being placed higher in the list. Therefore, these metrics are sensitive not only to the

relevance of the documents but also to their ranking. In retrieval systems, the *indexing mechanism* is responsible for making sure that all relevant documents are available for retrieval while *ranking algorithms*, which work based on the content inside the index, are responsible for putting the available content in order. This implies that the best metric to evaluate the effectiveness of an indexing method is the availability of all relevant documents that it would return for a given query while the best metric for evaluating a ranking method would be to see how well the retrieved documents by the index are in order. Furthermore, explicit semantic full-text index is represented by Treap, Wavelet Tree and HashMap which can be referred to, as *deterministic* index data structures. Meaning that these data structures do not lose any relevant document regard-less of the retrieval method. This is due to the fact that the posting list of an index key consists of all documents containing at least one occurrence of that index key. On the contrary, the implicit semantic full-text index only stores semantically relevant documents in the posting list of the index key. Consequently, evaluation of explicit semantic full-text index does not require us to measure the effectiveness of these three index data structures.

4.1.5 Final Results Synopsis

In this section, we summarize the empirical experiment results of our findings for the most efficient data structure for the Keyword, Entity Index and Type Index. The most efficient hybrid index for processing ranked intersection

queries and Boolean intersection queries is TT, which integrates the Entity Index and Keyword Index based on a non-list-based approach (integrated through the Treap data structure). However, the efficient hybrid index for processing ranked union queries is HH, which uses the list-based approach for integration. The list-based TH index is the most efficient heterogeneous index for processing ranked queries and Boolean intersection queries. However, it is not as efficient as TT for processing ranked intersection queries. TT and HH process ranked union and Boolean intersection queries faster than TH. The evaluation of the relation between query process time and query length determined that Treap is less sensitive to query length while it has a significant effect on Wavelet Tree query process time. We summarize all our findings in Table 4.2.

Based on our observations, it seems that the selection of HashMap-based Type Index would be the most efficient for semantically processing queries given the results.

4.2 Implicit Semantic Full-Text Index

In this section, we systematically evaluate our proposed indexing approach based on several research questions (RQ) as follows:

$RQ1$. How does the proposed indexing approach compare with the traditional inverted index from an *efficiency* perspective, i.e., memory usage and query processing time?

	Ranked Intersection	Ranked Union	Boolean Intersection	Integration Approach
The Most Efficient Index	TT	TT	HH	N/A
Heterogeneous Indices	TH	TH	TH	List-based
Homogeneous Indices	TT	TT	HH	*Hashmap integration* is the most efficient integration approach for ranked union. *Treap integration* is the most efficient approach for ranked and Boolean intersection

Table 4.2: The synopsis of our findings.

*RQ*2. How does the proposed indexing approach perform when contrasted with the traditional inverted index from an *effectiveness* point of view, i.e., the number of retrieved relevant results?

*RQ*3. How do the parameters of our proposed indexing approach such as embedding space dimensions, context window size and the size of the posting lists, which is dependent on the value for k in top-k approximate nearest neighbors impact *efficiency* and *effectiveness*?

4.2.1 Experimental Setup

In our experiments, we benefited from three widely adopted document collections within the information retrieval community: i) TREC *Robust04*, which is a small news dataset; ii) *ClueWeb09-B*, is a large Web collections from which we chose 1, 2 and 5 million random documents from among the first 50 million English pages of the corpus; and iii) Pooled Baselines documents from ClueWeb09-B where the top-100 related retrieval documents are extracted from three widely cited retrieval baselines, namely EQFE [65], the RM [127] and SDM [149]. We divide the ClueWeb09-B document collection into three document collections to evaluate the effect of document collection size on efficiency and effectiveness.

We use Freebase annotations of the ClueWeb Corpora (FACC1) as the semantic annotations of the ClueWeb09 documents. We use TagMe [85] to perform annotations for TREC Robust04 since there are no public annotations for this collection. In order to do so, we created a locally installed

Collection	Documents	Vocabulary Size	TREC Topics (Queries)	Max Length of Queries	Max Length of Annotated Queries
Robust04	528,155	782,799	301-450, 601-700	4	7
ClueWeb09-B-1m	1,073,009	5,910,302	1-200	5	8
ClueWeb09-B-2m	2,186,082	7,791,876	1-200	5	8
ClueWeb09-B-5m	5,006,963	13,666,170	1-200	5	8
Pooled Baselines	249,334	1,870,151	1-200	5	8

Table 4.3: Details of the TREC collections used in our experiments.

version of TagMe on our local server and ran each document through the service, which produced a set of entity links to Wikipedia entries. This way, the set of Wikipedia entities appearing in each document would be automatically derived. To prune unreliable entities, we set TagMe's confidence value to the recommended value of 0.1. The motivations for choosing the TagMe annotation engine is a study [53], which shows that TagMe is among the better performing annotation engines for different types of documents, e.g., Web pages and Tweets. Also, TagMe is an open source and provides publicly accessible API. The datasets and the used topics (queries) in our experiments are summarized in Table 4.3. The selected topics needed to also be semantically annotated for which we use TagMe to annotate the related TREC queries for each document corpus. Figure 4.15 shows a visualization of the document collections based on the embedding of the keywords, entities and documents in the embedding space, developed using the *t-Distributed Stochastic Neighbor Embedding* (t-SNE) technique [143].

To evaluate the effect of other parameters of the PV model, we selected three dimensions; 300, 400 and 500 and two context window sizes; 5 and 10. Thus, several variations of our proposed index are built based on the above mentioned parameter set. To distinguish between variations, which were built based on different context window sizes and different sampling methods, we defined simple abbreviations to refer to each variation as presented in Table 4.4. For instance, the abbreviation *W5* refers to the variation of our implicit

[5] `https://lvdmaaten.github.io/tsne/`

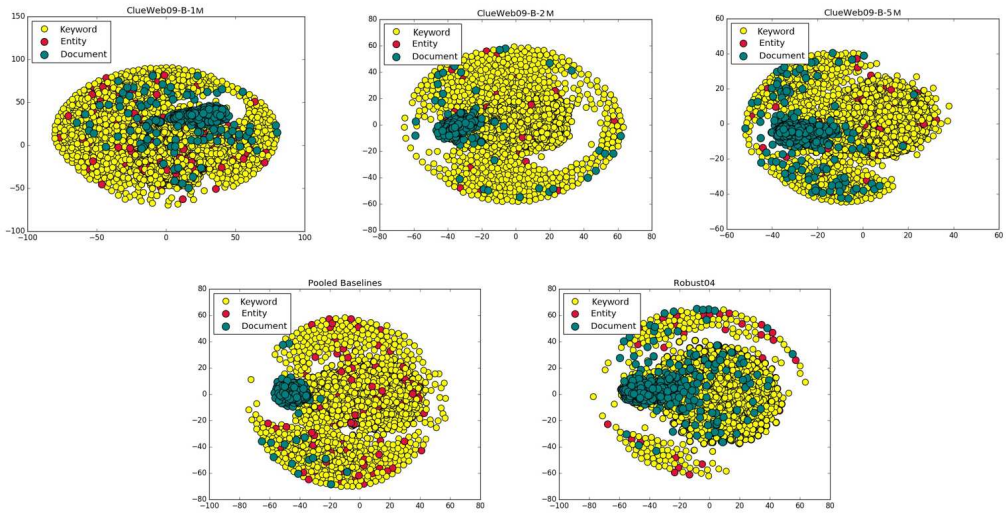

Figure 4.15: The joint embedding space of all document collections for keywords, entities, types and documents is visualized in scatter plots with t-SNE.[5]

semantic full-text index with paragraph vector model trained with a context window size of 5 and based on Negative Sampling and likewise *W10-HS* refers to the variation that has a context window size of 10 with Hierarchical Softmax sampling. Moreover, in order to show the impact of k, three values for k were experimented. We refer to these values of k as $k_1 = 0.05\%$, $k_2 = 0.1\%$ and $k_3 = 0.2\%$, which are percentage of the number of documents in the document collection. In addition, the ratio of the average length of posting list for baseline indices to the average length of posting list of the proposed semantic indices is less than 15 for all selected document collections. The posting list size ratio chosen based on k_1, k_2 and k_3 for baseline indices to the proposed semantic indices for all document collections is presented in

Sampling Method	Window Size	
	5	10
Hierarchical Softmax	W5-HS	W10-HS
Negative Sampling	W5	W10

Table 4.4: Abbreviations used to refer to the different variations of our work.

Collection	Posting List Size Ratio based on:		
	k_1	k_2	k_3
Robust04	1	2	4
ClueWeb09-B-1m	2	4	8
ClueWeb09-B-2m	4	8	15
ClueWeb09-B-5m	2	4	8
Pooled Baselines	4	8	15

Table 4.5: The ratio of the average length of baseline Indri index posting list to the average length of the proposed indexing approach for different values of k.

Table 4.5. The actual values for k depending on the document collections are presented in Table 4.6.

In terms of the comparative baseline used in our experiments, we used Indri[6] with its default parameter settings. Indri [189] is a widely adopted information retrieval toolkit developed to simplify evaluation over standard text collections from evaluation forums, e.g., *TREC*, *CLEF*, *NTCIR*. For the sake of comparison, we used Indri to index the various text corpora that are listed in Table 4.3 and also to process the queries listed in the same table. The

[6] `http://www.lemurproject.org/indri.php`

Collection	Length of Posting List		
	k_1 (0.05%)	k_2 (0.1%)	k_3 (0.2%)
Robust04	2,700	5,400	5,000
ClueWeb09-B-1m	5,000	10,000	20,000
ClueWeb09-B-2m	10,000	20,000	40,000
ClueWeb09-B-5m	26,000	52,000	104,000
Pooled Baselines	1,250	2,500	5,000

Table 4.6: The length of the posting list depending on the value for k.

obtained results for the baseline are based on this installation of Indri.

4.2.2 Implementation

Our experiments were performed on a 2x2.7GHz Eight-Core Intel Xeon Processor with 20MB Cache–E5-2680 with 256GB of memory running Ubuntu 16. We use Python for implementations of implicit semantic full-text index. Figure 4.16 illustrates the data flow diagram of implicit semantic full-text index to give an overview of this index implementation. The Input data of the implicit semantic full-text index is the same as the explicit semantic full-text index. But all input data is integrated before building index in contrast to explicit semantic full-text index. To integrate input data, we learn embeddings for keywords, entities, entities types and documents within the same embedding space using PV models. We use *Gensim* [169] for building PV models, which follows the model introduced by Mikolov et al. [129]. In our experiments, both sampling techniques, namely Hierarchical Softmax and

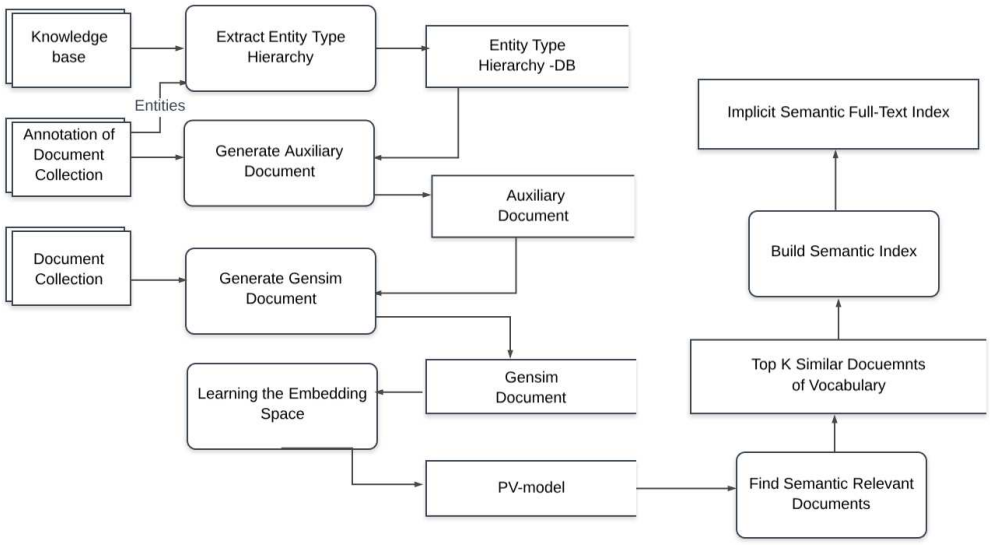

Figure 4.16: The data flow diagram of implicit semantic full-text index.

Nagetive sampling are considered and are implemented according to [154].

In order to build PV models, all input data needs to be prepared. The ***Extract Entity Type Hierarchy*** process provides entity type by extracting type hierarchy of annotated entities of document collection from KB. Accordingly, KB is parsed to find all types, sub-types and super-types of annotated entities of the document collection. These values are then stored in a MySQL table which is named *Entity Type Hierarchy*, providing fast and easy access to them whenever needed.

We recall from Section 2.4.2.1 that in PV models, the context of a keyword is the number of keywords seen before and after it. Hence, we cannot directly use PV models due to the fact that our input data is more than a document context. Therefore, we need a new document context which consists of an-

notated entities and their types alongside with document context. The new document is called *auxiliary document*(Section 3.2.1). The ***Generate Auxiliary Document*** process creates multiple auxiliary documents for each of the documents in the corpus by replacing each annotated token with one of its corresponding entity or entity type.

Furthermore, we need to perform text transformation on auxiliary documents like any document collection before building the index. The process of text transformation consists of:

- Stemming the context of auxiliary documents with *Porter Stemmer* of Gensim preprocessing library [7],
- removing stopwords of Indri [8] and Gensim preprocessing library.
- Htmlparser[9] is used for extracting text on HTML pages.
- Using Gensim preprocessing library to perform some basic tasks, e.g., removing punctuations and converting all words to, lower case.

Furthermore, the auxiliary documents needed to be converted to documents appropriate for Gensim. Since, the maximum length of a document in Genism is fixed documents longer $10,000$ tokens should be converted to a set of documents within the acceptable length by keeping document identifiers. To perform this work, we defined the ***Generate Gensim Documents***. The

[7]https://radimrehurek.com/gensim/parsing/preprocessing.html
[8]http://www.lemurproject.org/stopwords/stoplist.dft
[9]https://docs.python.org/2/library/htmlparser.html

output of this process is given to Gensim for embedding keywords, entities, entities types and document to build PV models.

To evaluate the effect of other parameters of the PV model, several variations of our proposed index are built based on the mentioned parameter set in Section 4.2.1. The abbreviation of all variation of PV model are presented in Table 4.4.

The objective of the ***Top-k Similar Documents of Vocabulary*** process is building posting lists for implicit semantic full-text index. To identify the top-k most similar documents that are represented as data points within the embedding space. We use approximate nearest neighbor search to identify the top-k most similar documents. These documents are sorted in each posting list based on their degree of vector similarity. For the purpose of performing approximate nearest neighbor search on our learnt embedding space, we employed Annoy[10], which is a C++ library with Python bindings. It uses the Binary tree as a data structure for returning k nearest neighbors in $O(logn)$ where each node is a random split. Annoy divides the search space m times and builds m binary trees, thus a forest of trees is created. The value of m is a tradeoff between precision and performance of this algorithm. We chosen to have $2,000$ trees in the binary tree forest of Annoy. To evaluate the efficiency and effectiveness of the implicit semantic index, queries and their annotation need to be combined since this index does not distinguish between keyword, entity and type. As mentioned in the previous

[10]`https://github.com/spotify/annoy`

section, we use Indri for building the implicit semantic full-text index and for processing each query. Thus, the queries need to be converted to Indri queries.

Implicit semantic full-text index is applicable to any document collection which is annotated with concepts from any KB as the input. This index can answer any keyword query (refer to Section 2.3) as well as entity bearing query.

4.2.3 Efficiency of the Implicit Semantic Full-Text Index

As mentioned earlier, efficiency is one of the main desirable features of an IR system [41]. In this context, efficiency evaluates the performance of the retrieval systems by considering index storage space (index size) and query processing time (QPT). As mentioned in Section 2.1.4; striking the right balance between efficiency and effectiveness is important for the usefulness and usability of an information retrieval system.

In the first research question, we evaluate the efficiency of the proposed indexing mechanism. The efficiency of an index is often evaluated based on two measures: the index size and its QPT. Therefore, one can compare the efficiency of two comparable indexing mechanisms by comparing their index size and QPT when applied on the same textual corpora and for the same query set. In order to perform comparative analysis of the performance of

Collection	Configuration		
	Window Size	Posting List Size	Embedding Dimension
Robust04	W5	k_3	D500
ClueWeb09-B-1M	W10	k_3	D500
ClueWeb09-B-2M	W10	k_3	D500
ClueWeb09-B-5M	W10	k_3	D500
Pooled Baselines	W5	k_3	D500

Table 4.7: The configurations of our proposed approach used in *RQs* 1 and 2.

our proposed index, we indexed the corpora listed in Table 4.8 both using our proposed indexing method as well as an installation of Indri. The configurations used in *RQs* 1 and 2 are shown in Table 4.7. For instance, for the Robust04 collection, we used a window size of 5 with Negative Sampling, the length of posting list to be k_3 according to Table 4.6 and an embedding dimension of 500. We will discuss later how the values in Table 4.7 are decided in RQ 3. We measured the amount of storage space required for each of the indices (index size) as well as the amount of time required by each of the indices to process the queries related to each collection. The results are reported in Table 4.8.

As seen in the table, our proposed indexing approach has a better efficiency compared to the baseline Indri index in terms of both index size and QPT. The improved efficiency of our approach can be consistently observed across all five corpora and for different query sets; hence, it shows *robust* performance regardless of the corpus size, corpus characteristics and the input

Collection	Indri		Proposed Approach		Efficiency Difference	
	Size	QPT	Size	QPT	Δ Size	Δ QPT
Robust04	904.2 MB	39.3s	552.8 MB	26.9s	+38.86%	+31.55%
ClueWeb09-B-1M	3.2 GB	1m54.3s	1.05 GB	54.7s	+67.18%	+52.14%
ClueWeb09-B-2M	10.7 GB	5m20.3s	2.06 GB	2m23.6s	+80.74%	+55.16%
ClueWeb09-B-5M	18.9 GB	11m33.3s	12.46 GB	4m54.5s	+34.07%	+57.52%
Pooled Baselines	1.9 GB	45.1s	962.4 MB	32.1s	+49.34%	+28.82%

Table 4.8: Comparative analysis of the efficiency of our proposed indexing strategy compared to Indri.

query set. The improved performance of our proposed indexing approach is primarily due to two characteristics of our proposed approach: i) the method does not enforce that all documents that have the index key to be present in the related posting list, and ii) the size of the posting lists depends on the size of k that is selected in the top-k approximate nearest neighbor search technique. We will later show in RQ 3 how the size of k and other parameters related to learning the embedding model impact efficiency and effectiveness. Theoretically speaking, there is a clear dependency between index size and QPT whereby smaller indices result in faster response time to queries as there are less information in the index to be processed for each query. This is observable in Table 4.8 as well. However, it is quite likely that smaller indices do not consist of all the relevant documents and hence cannot retrieve all the possible relevant documents for an input query. Therefore, it is important to explore whether the improvement in index size and QPT provided by our approach translates into poorer/better performance in terms of retrieving relevant documents or not. As such, the next research question ($RQ2$) explores the effectiveness of our proposed approach.

4.2.4 Effectiveness of the Implicit Semantic Full-Text Index

The effectiveness of any index method is measured by studying the **availability of all relevant documents** to retrieve; as explained in Section

Collection	Number of relevant documents retrieved by Indri (percentage of relevant documents)	Number of relevant documents retrieved by our approach (percentage of relevant documents)	Effectiveness Difference
Robust04	10,131	13,178	+23.12%
ClueWeb09-B-1M	179	157	-12.29%
ClueWeb09-B-2M	1,028	998	-2.91%
ClueWeb09-B-5M	1,587	1,471	-7.30%
Pooled Baselines	6,309	6,080	-3.62%

Table 4.9: The number of relevant documents retrieved by each index.

4.1.4. Hence, we evaluate the effectiveness of implicit semantic full-text index by measuring whether all relevant documents are accessible to retrieve or not. As mentioned earlier in Section 3.2.2, posting lists of the implicit semantic full-text index have the top-k most similar documents to the index key. As well, there is no guaranty these documents certainly contain index key in their context. For these reasons, measuring effectiveness of this indexing method is essential to be sure implicit semantic full -text index stores all relevant documents for an index key. To achieve this goal, the effectiveness is evaluated based on the ***number of relevant documents retrieved*** for the queries related to the document corpora. Consequently, in our comparisons with the baseline, we compute how many relevant documents are returned by the baseline (Indri) compared to the number of relevant documents returned by implicit semantic full-text index.

We would like to provide further insight as to why we compare our work

with Indri and not with any of the state of the art semantics-based retrieval models such as [74, 65, 212]. There are two primary reasons for this: (1) as mentioned earlier, our proposed method is an *indexing mechanism* whose objective to index and maintain the maximum number of relevant documents while the mentioned state of the art techniques are *retrieval methods* that focus on ranking and hence are less focused on maintaining comprehensiveness and are primarily engaged in making sure that the most relevant documents are placed at the top of the retrieved list. Therefore, the objective of our work and these methods is different. (2) The mentioned techniques operate primarily by re-ranking results obtained from a keyword-based retrieval system such as [127, 149] and therefore, do not maintain or provide a separate list of relevant documents. As such, the maximum number of relevant documents retrieved by these methods is equivalent to the number of relevant documents retrieved by the baseline keyword-based techniques already implemented in Indri. For this reason, and given the fact that the focus of our work is maximizing the coverage of relevant documents in the index and not on ranking the relevant documents, we compare our work with Indri and not ranking methods.

The results reported in Table 4.9 show the number of relevant documents retrieved based on Indri and our proposed approach. For instance, as indicated in the table, Indri is able to retrieve $6,309$ relevant documents based on the Pooled Baselines collection, while our proposed approach has been able to return $6,080$ relevant documents. It should be noted that both re-

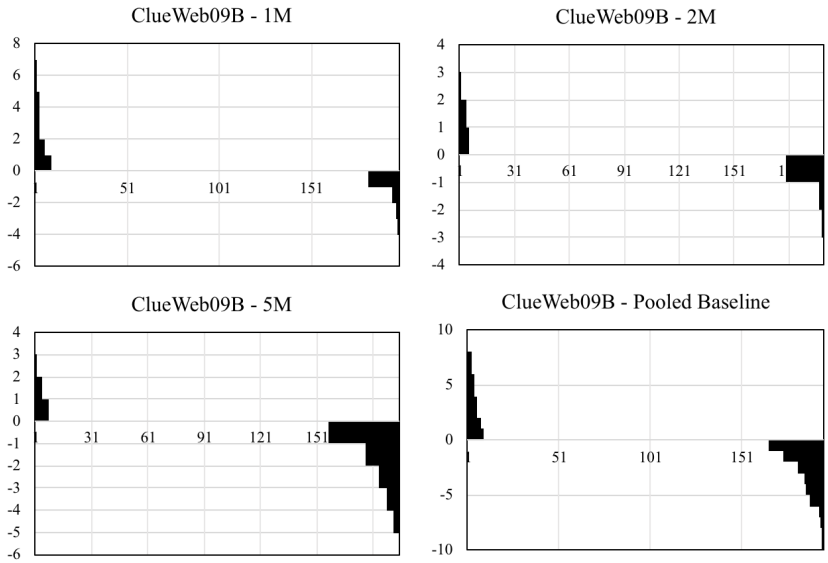

Figure 4.17: The comparative performance of the effectiveness of our proposed approach against Indri on a per query basis on ClueWeb09.

call and precision metrics will have comparable performance based on the number of relevant retrieved documents by each method. The reason is that recall is defined as the number of relevant documents retrieved in the context of all relevant documents. The number of relevant documents per query is constant and the same for both methods and hence is a constant value. On the other hand, precision is the number of relevant documents in the list of all retrieved documents. In this case, since our retrieval happens based on top-k most similar documents, the size of the retrieved set is also a constant value. Therefore, the behavior of recall and precision is similar and primarily dependent on the number of relevant retrieved documents.

There are several observations that can be made based on the results in Table

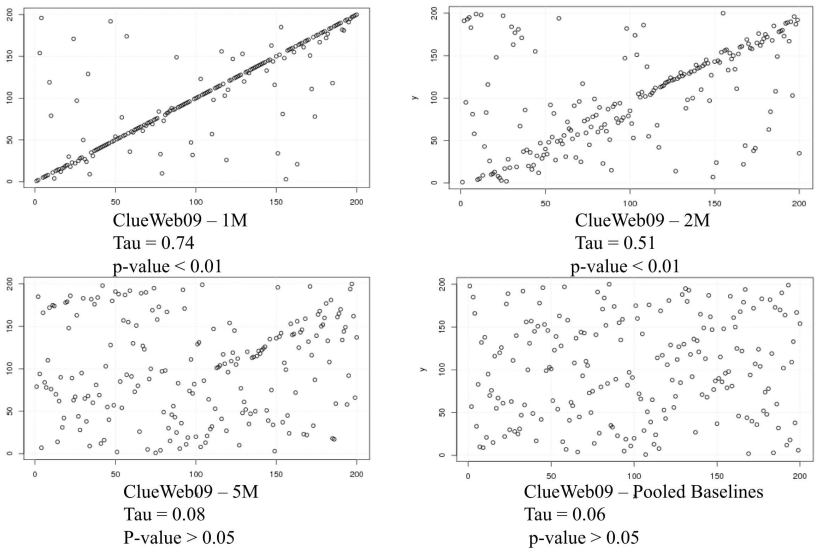

Figure 4.18: Kendall's rank correlation between the ranked list of queries based on the number of relevant documents retrieved for our approach compared to Indri.

4.9. The first observation indicates that both approaches retrieve a reasonable number of relevant documents for all five document collections. The second observation is that while both approaches are close in the percentage of relevant documents that they can retrieve, they differ in their performance depending on the collection. For the variations of the ClueWeb9B collection, the Indri index returns a higher number of relevant documents while on the Robust04 collection, our approach provides better results.

We now further compare the performance of our proposed approach with Indri on a per query basis. It is important to see whether the comparative effectiveness results reported in Table 4.9 are also consistently observed in each query. For this reason, we compare the number of relevant retrieved doc-

uments by our approach compared to Indri in Figure 4.17 (for ClueWeb09). The y-axis of the diagrams is the difference between the number of relevant documents retrieved by our approach and Indri. Therefore, positive lines in the chart denote those queries for which our approach retrieved more relevant documents and the negative values are those queries for which Indri retrieves more relevant documents. The queries are sorted in descending order for better visual understanding. The blank queries are those queries that had the same number of relevant documents by both our approach and indri. As seen in Figure 4.17, for the majority of the queries, the number of relevant documents retrieved by both approaches are the same and there are only few queries that have slightly different performance.

Now, in order to better understand the behavior of each index, we measure Kendall's rank correlation coefficient based on the queries for each approach sorted by the number of relevant documents retrieved by our approach compared to Indri. The higher the rank correlation is, the more similar the performance of the approaches would be. Figure 4.18 visualizes the rank correlations. The figure shows that when the number of relevant documents to queries are low (ClueWeb09 - 1M and 2M) that the performance of our proposed approach and Indri as well as the rank of the queries are quite correlated; however, as more relevant documents become available the correlation of the two approaches drops and the correlation is no longer statistically significant (ClueWeb09 - 5M and Pooled Baselines). This observation needs to be interpreted in the context of the findings of Figure 4.17. As seen in Figure

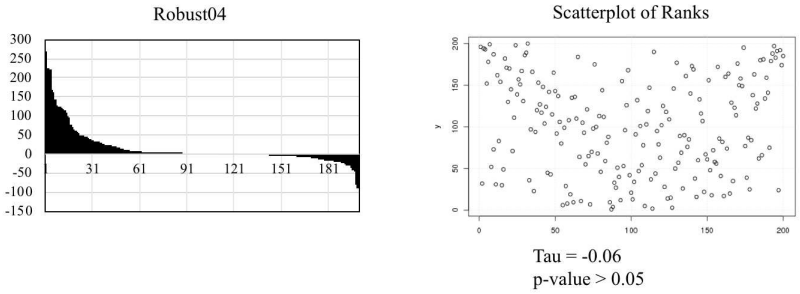

Figure 4.19: The comparative performance of the effectiveness (left) and Kendall's rank correlation (right) for our approach compared to Indri on the Robust04 document collection.

4.17, the two approaches have a similar performance on the query level on all three variations of the ClueWeb09 document collection (blank space in the middle of the diagram showing queries that were tied in terms of number of relevant documents retrieved by each of the approaches), the divergence of the rank correlation of queries on ClueWeb09 - 5M and Pooled Baselines means that our approach is retrieving a different set of documents compared to Indri for the queries. In other words, while the two approaches retrieve similar number of relevant documents for each query, the relevant documents that are retrieved are not necessarily overlapping in both approaches and become complimentary as the size of the document collection grows. This is also similarly observed for the Robust04 document collection in Figure 4.19. We additionally explored whether the retrieval effectiveness exhibited by our proposed approach is due solely to the characteristics of the neural embedding technique or the additional inclusion of type and entity information also played a role. In order to examine this, we used the best configuration ob-

	ClueWeb09-B			Pooled Baselines	Robust04
	1M	2M	5M		
Kendall's Tau	0.539	0.427	0.425	0.019	0.044

Table 4.10: Kendall's rank correlation between the ranked list of queries baesd on the number of relevant documents retrieved by our approach compared to when neural embeddings are learnt solely based on keywords.

tained in our previous experiments shown in Table 4.7, to train an embedding model based solely on the textual content of the document collections without the inclusion of type and entity information. The trained embedding model was then used to build the index. Given the index, we retrieved the related documents to each query and then ranked the queries based on the number of relevant documents retrieved. The ranked list of queries was then compared to the ranked list of queries obtained from our proposed approach based on Kendall's rank correlation measure. A highly correlated set of ranked queries would show that our proposed approach and the index based on embeddings trained solely on textual content are similar and as such the inclusion of entity and type information does not play an important role in the process. The findings are reported in Table 4.10. Based on the correlations reported in this table and compared to the rank correlations observed between our approach and the Indri indices (shown in Figures 4.18 and 4.19), it can be seen that the correlation between our proposed approach and Indri is higher than when compared to the embeddings trained solely based on textual content, indicating that type and entity information included in our approach

do in fact play a substantial role in retrieval effectiveness. Furthermore, it is important to point out that not only does the inclusion of the type and entity information impact retrieval effectiveness, but also enables other upstream document ranking models, which work based on entities and types, such as [74, 102], to be built on top of our proposed approach. This is something that is not possible based on Indri indices or embeddings trained solely based on textual content.

We further explore our observations based on Kendall's rank correlation by identifying the hardest and easiest queries for our approach and Indri. We define hard queries for some method (method being our approach or Indri) to be those queries that have the least number of relevant documents retrieved for them by that method. Conversely, we define easy queries to be those that have the most number of relevant retrievals by that method (our approach or Indri). Table 4.12 (easiest query shown on the top row) and Table 4.11 (hardest query placed at the top row) show the sorted list of queries for our approach and Indri. As seen in the tables, the two approaches have a similar set of queries identified as easy queries but their hard queries do not have as much overlap. This reinforces our observations based on Kendall's rank correlation and the difference in retrieval effectiveness that while overall the two approaches retrieve similar number of relevant documents but their effectiveness is complementary to each other, showing that our approach can retrieve relevant documents that would otherwise not be retrieved by Indri. As indicated earlier, the tradeoff between effectiveness and efficiency is also

an important consideration in designing information retrieval systems. RQs 1 and 2 explore these two aspects independently and hence it is important to analyze them in tandem. Within the Robust04 dataset, our proposed approach provides improvement in terms of both effectiveness and efficiency. Furthermore, on the Pooled Baselines collection, our approach provides significant improvement in terms of efficiency (index size and QPT) over the baseline; however, it shows a similar performance in terms of effectiveness. It is clear that in such a case, our proposed approach would be favored in a competitive information retrieval systems. Finally, on the ClueWeb09B variations, regardless of the size of the corpus, the Indri index provides better effectiveness while our proposed approach provides better efficiency. The clear tradeoff between effectiveness and efficiency can be seen in the ClueWeb09B collections. We believe that our proposed approach is suitable for cases where: 1) QPT is of significant importance because our work is able to provide at least 50% speedup for the ClueWeb09B collection; and 2) storage space is of importance for storing the index. Given the abundance of memory, this might not seem to be an important consideration; however, when caching or embedded systems considerations are taken into account, a smaller index could be of more help in efficiently retrieving relevant documents. It is important to point out that recent studies [197] have shown that for two competitive retrieval systems, QPT can be the determining factor for overall user satisfaction even when one of the retrieval systems is providing slightly weaker retrieval effectiveness. As such, the speedup and space utilization pro-

vided by our approach can be a strong advantage when considering that its effectiveness is still competitive to the baseline on the ClueWeb09B collection and competitive or better for the Pooled Baselines and Robust04 collections.

4.2.5 Impact of Model Parameters on Effectiveness and Efficiency

The impact of the model parameters can be considered from the perspective of the embedding model variations as well as the impact of the size of the posting lists. We systematically explore the impact of these parameters in this section.

4.2.5.1 Impact of Embedding Parameters

The performance of our proposed indexing strategy depends on how well the learnt embedding model can capture the semantics and relationship between keywords, entities, types and documents. Research has already shown that the parameters of the embedding space training can impact the quality of the embedding [226]. As such, we have empirically evaluated the impact of these parameters on the performance of our proposed index. There are primarily three parameters within the training process: (1) sampling strategy; (2) context window size; and (3) embedding dimension. As mentioned earlier, the results reported in research questions *RQs* 1 and 2 are based on the findings of this section with the best performing trained models.

We first explore whether and how much the sampling strategies impact the performance of the proposed index. There are two main types of sampling strategies namely, Negative Sampling and Hierarchical Softmax. In order to study the impact of the sampling strategy, we systematically studied the various combinations of parameter values for context window size and embedding dimension in combination with the sampling strategies. Due to space constraint, we only report the values for the parameter values k_3, $D500$ and context window sizes of 5 and 10 but note that the other variations show similar behavior. Our observations with regards to the impact of both effectiveness and efficiency regardless of the dimensionality of the embeddings and the context window size was that both sampling strategies show very competitive performance for effectiveness and efficiency. Table 4.13 summarizes the number of relevant documents that are retrieved based on the proposed approach depending on whether Negative Sampling or Hierarchical Softmax was employed. As seen in the table, the number of relevant documents retrieved by each of the sampling strategies is very close to each other with Negative Sampling having a slight edge over Hierarchical Softmax. On the other hand and for efficiency as shown in Figure 4.20, the performance of the proposed indexing strategy is very similar for both Negative Sampling and Hierarchical Softmax. Our conclusion based on the observations made in the experiments is that the sampling strategy does not impact the performance of the indexing mechanism and as such is not an issue of consideration. In the rest of our experiments, we adopted Negative Sampling due to its slightly

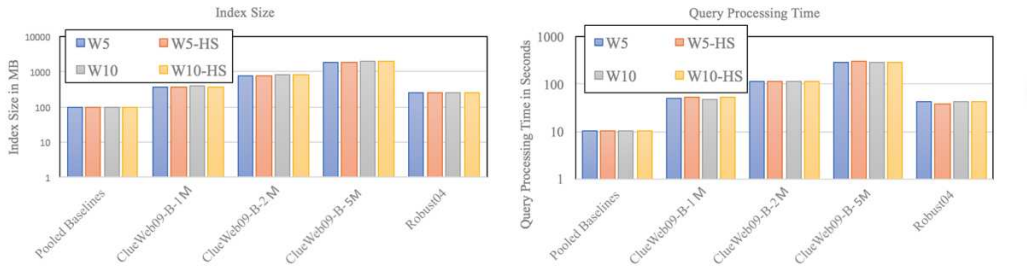

Figure 4.20: Impact of sampling strategies on retrieval efficiency.

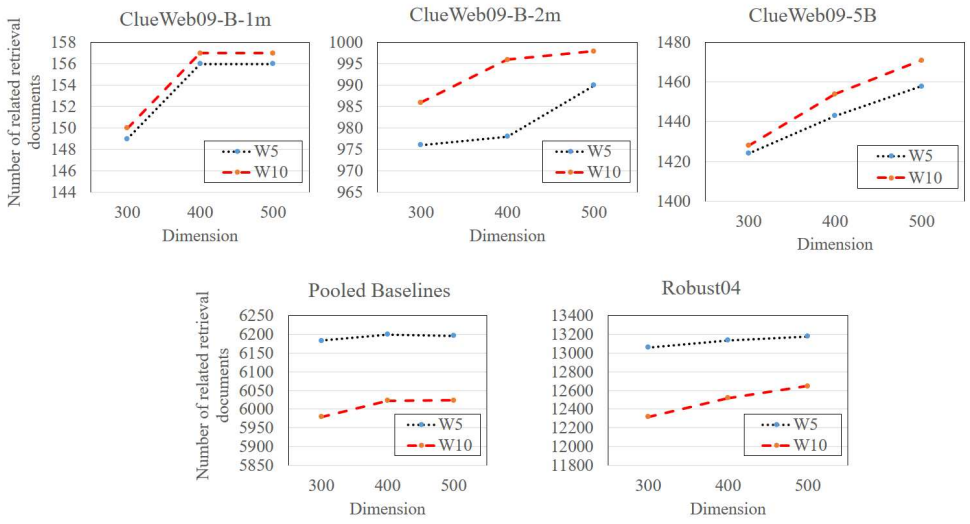

Figure 4.21: Impact of context window size and embedding dimension on retrieval effectiveness.

better performance on retrieval effectiveness.

Furthermore, several researchers [129, 226] have already shown that neural embeddings can be sensitive to context window size as it is this parameter that determines which keywords, entities and types are considered adjacent in practice and hence would end up having similar vector representations.

	ClueWeb09 - 1M		ClueWeb09 - 2M		ClueWeb09 - 5M		Pooled-Baselines		Robust04	
	Ours	Indri	Ours	Indri	Ours	Indri	Ours	Indri	Ours	Indri
1	**rice**	er tv show	**bobcat**	income tax return online	**income tax return online**	the current	**to be or not to be that is the question**	arkadelphia health club	R&D drug prices	Educational Standards
2	**satellite**	the wall	milwaukee journal sentinel	sewing instructions	**to be or not to be that is the question**	defender	**memory**	angular cheilitis	poaching, wildlife preserves	Falkland petroleum exploration
3	rincon puerto rico	raffles	**earn money at home**	kiwi	**earn money at home**	wilson antenna	**earn money at home**	the current	obesity medical treatment	journalist risks
4	iowa food stamp program	titan	**rice**	**bobcat**	rincon puerto rico	avp	rincon puerto rico	defender	food/drug laws	ocean remote sensing
5	rock art	earn money at home	rincon puerto rico	milwaukee journal sentinel	fybromyalgia	**income tax return online**	map	**to be or not to be that is the question**	Great Britain health care	blood-alcohol fatalities
6	credit report	**rice**	getting organized	**earn money at home**	map	**to be or not to be that is the question**	kcs	**memory**	supercritical fluids	drug legalization benefits
7	elliptical trainer	**satellite**	to be or not to be that is the question	**rice**	ps 2 games	**earn money at home**	djs	**earn money at home**	**sick building syndrome**	**sick building syndrome**

Table 4.11: The list of *hard queries* for our approach compared to Indri. Shared queries between two methods are denoted by **bold**.

	ClueWeb09 - 1M		ClueWeb09 - 2M		ClueWeb09 - 5M		Pooled Baselines		Robust04	
	Ours	Indri	Ours	Indri	Ours	Indri	Ours	Indri	Ours	Indri
1	**dangers of asbestos**	**dangers of asbestos**	**inuyasha**	**inuyasha**	**inuyasha**	**inuyasha**	**fact on uranus**	**fact on uranus**	**Implant Dentistry**	computer viruses
2	**cheap internet**	**voyager**	**obama family tree**	**obama family tree**	**dangers of asbestos**	**dangers of asbestos**	**inuyasha**	**inuyasha**	Radio Waves and Brain Cancer	**encryption equipment export**
3	**voyager**	**cheap internet**	**fact on uranus**	**fact on uranus**	**fact on uranus**	**worm**	**tornadoes**	**tornadoes**	**Income Tax Evasion**	tax evasion indicted
4	**interview thank you**	**cell phones**	**tornadoes**	**tornadoes**	**obama family tree**	**voyager**	**dangers of asbestos**	**dangers of asbestos**	**encryption equipment export**	**Implant Dentistry**
5	**cell phones**	used car parts	**figs**	**figs**	**worm**	**fact on uranus**	**espn sports**	**espn sports**	Poliomyelitis and Post-Polio	food stamps increase
6	**bellevue**	**bellevue**	**tangible personal property tax**	**south africa**	tornadoes	**obama family tree**	**euclid**	**diabetes education**	Hubble Telescope Achievements	**Income Tax Evasion**
7	lower heart rate	**interview thank you**	**south africa**	**tangible personal property tax**	**voyager**	south africa	**diabetes education**	**euclid**	Endangered Species (Mammals)	exotic animals import

Table 4.12: The list of *easy queries* for our approach compared to Indri. Shared queries between two methods are denoted by **bold**.

Collection	Number of relevant documents based on				
	Judgement qrel file	Context window size 5		Context window size 10	
		NS	HS	NS	HS
Robust04	17412	13178	13129	12647	12651
ClueWeb09-B-1M	212	156	130	157	148
ClueWeb09-B-2M	1050	990	984	998	989
ClueWeb09-B-5M	1666	1459	1436	1471	1458
Pooled Baselines	6390	6196	6085	6024	5964

Table 4.13: Impact of sampling strategies: Negative Sampling(NS) and Hierarchical Softmax(HS) on retrieval effectiveness (k_3 and $D500$).

A larger context window size might result in creating semantic association between keywords, entities and types that are not in fact related, while a smaller context window size might miss to make correct associations between related keywords available in the corpus. In order to evaluate the impact of context window size, we selected the sampling strategy to be Negative Sampling, as discussed in the previous paragraph, and systematically varied the dimension size between 300, 400 and 500. Based on these settings, we evaluated whether a change in context window size between 5 and 10 impacted retrieval effectiveness and efficiency. Figure 4.21 reports on our findings of the impact of context window size on retrieval effectiveness. We find that the characteristics of the document collection influences the optimal size of the context window. In our five document collections, the three variations of the ClueWeb09-B corpus favored a larger context window size while the

Robust04 and Pooled Baselines collections were inclined to better retrieval effectiveness with a smaller context window size. This can be explained based on the characteristics of the document collection as mentioned earlier. The difference between the ClueWeb document collections and the Robust04 and Pooled Baselines collections is on *size of the collection*. Our findings indicate that larger context window sizes will show better retrieval effectiveness performance when used on collections that have a large document corpus size; on the contrary, smaller-sized collections will show better effectiveness with a small context window size.

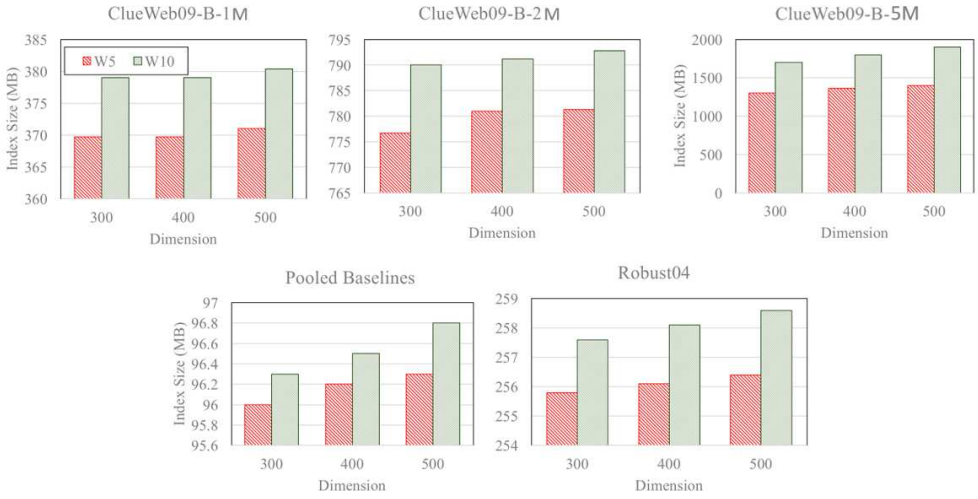

Figure 4.22: Impact of context window size and embedding dimension on retrieval efficiency (index size).

From the perspective of retrieval efficiency, both in terms of index size shown in Figure 4.22 and QPT shown in Figure 4.23, the impact of context window size is consistent. Larger context window size results in both an increase in in-

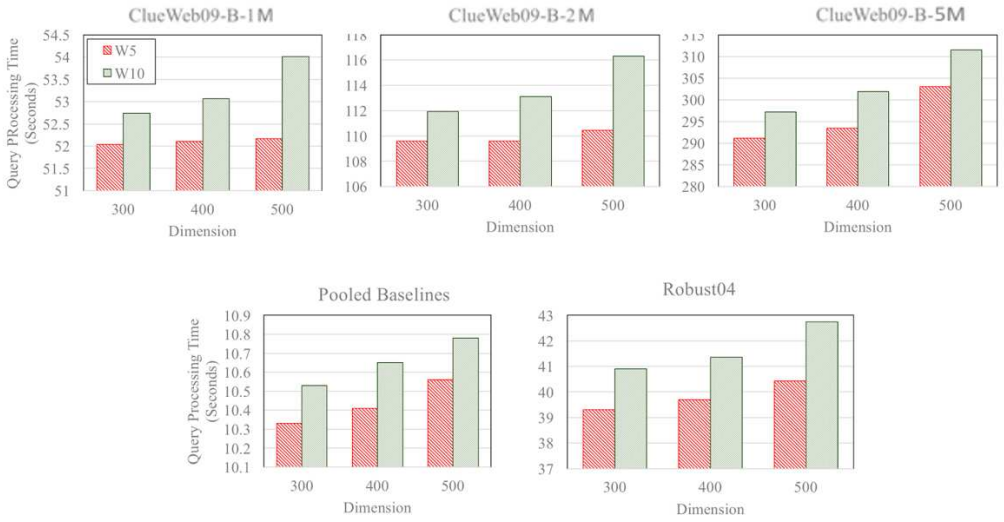

Figure 4.23: Impact of context window size and embedding dimension on retrieval efficiency (QPT).

dex size as well as increased QPT. This finding is expected as a larger context window size will lead to a larger number of similar vectors in the embedding space and therefore there will be much more similar keywords, entities and types per given index key in the proposed index leading to increased index size and QPT. There is clearly a tradeoff between effectiveness and efficiency as shown in the contrast between Figure 4.21 and Figures 4.22 and 4.23. Therefore, the choice of the best context window size will depend on the requirements of the retrieval system. In our experiments reported in *RQs* 1 and 2, the choice of the size of the context window selected and reported in Table 4.7 was motivated by the tradeoff between efficiency and effectiveness and giving higher importance to effectiveness. It should be noted that in a resource-constrained environment or scenarios where extremely fast response

time is required, one might prefer to prioritize efficiency over effectiveness.

We also explore the impact of embedding dimensions on both effectiveness and efficiency. The dimension of the vectors in the embedding space is often related to its representation power where a larger size vector has the ability to capture and represent a wider variance of information. With this in mind, if the number of data points that need to be represented in the embedding space are not too high, a larger embedding dimension is not required in practice. Our findings are inline with this theoretical foundation. Figure 4.21 shows that for the larger document collections, namely the 2 million and 5 million ClueWeb corpora, the increase in dimension size leads to improved retrieval effectiveness while in the other collections, an increase in the dimension size does not necessarily lead to meaningful improvements in effectiveness. Similar to context window size, larger embedding dimensions lead to increased index size and QPT; hence reinforcing the importance of deciding on the dimension of the embeddings depending on the requirements of the retrieval system. As mentioned in the previous paragraph, the choices for the embedding dimension reported in Table 4.7 were motivated by prioritizing effectiveness over efficiency in our work.

In our proposed indexing strategy, the size of the posting lists is determined based on the number of nearest neighbors that are retrieved for the posting list key. As such the number of postings in a posting list can range up to the maximum number of documents in the corpus ranked based on their vector similarity to the posting list key. This is in contrast to the size of the posting

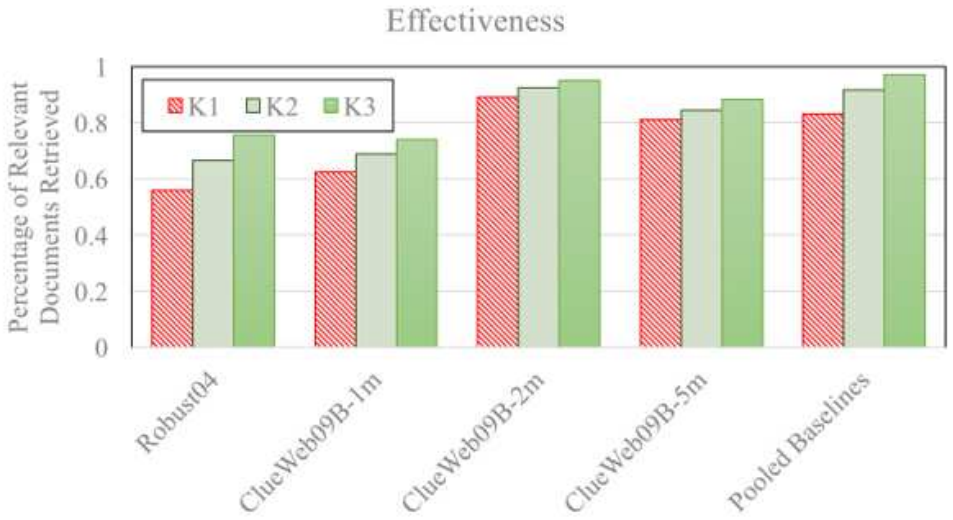

Figure 4.24: Impact of nearest neighborhood radius on retrieval effectiveness.

lists in inverted indices where the number of postings is determined solely based on the number of documents that include the posting list key. As such, we investigate the impact of the posting list size on retrieval effectiveness and efficiency. To this end, we systematically changed the size of the posting lists based on k_1, k_2 and k_3 as shown in Table 4.6. The values for k indicate the radius size of the nearest neighborhood search and determines how many most similar data points to the posting list key need to be retrieved and placed in the posting list. In these experiments, the parameter values used for training the embedding model was based on the settings shown in Table 4.7.

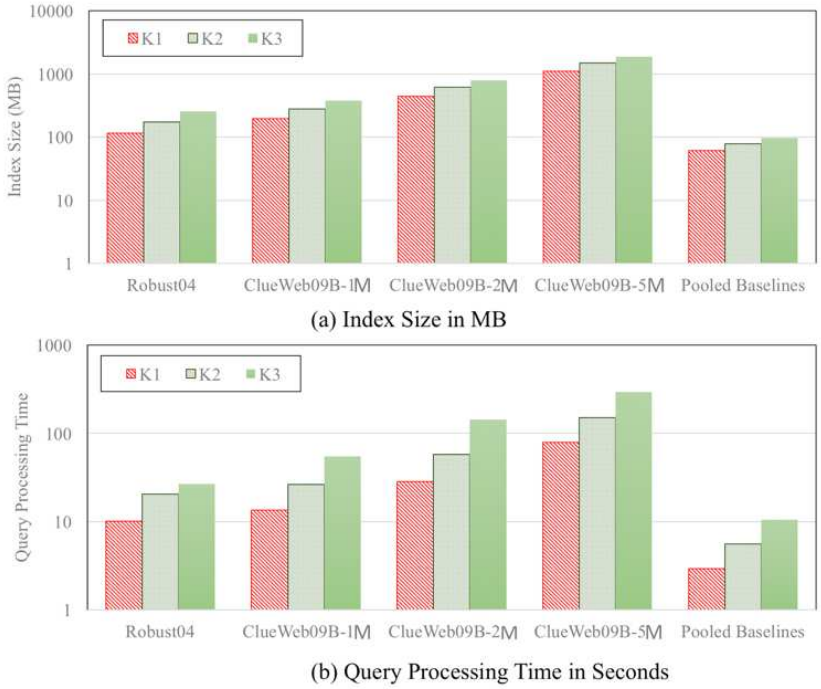

Figure 4.25: Impact of nearest neighborhood radius on retrieval efficiency.

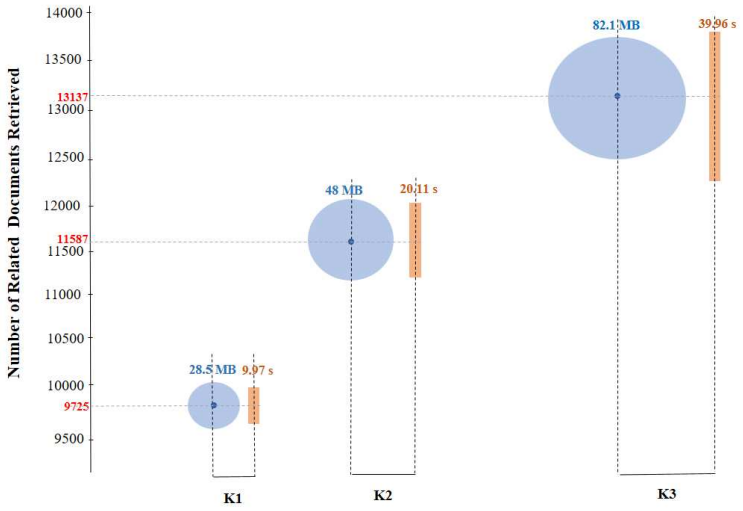

Figure 4.26: The tradeoff between effectiveness and efficiency based on different neighborhood radius sizes on Robust04.

4.2.5.2 Impact of Neighborhood Radius

Figure 4.24 shows that the increase in the size of the neighborhood radius positively impacts retrieval effectiveness. This is an expected outcome given the fact that a larger posting list has a higher likelihood of containing relevant documents. In other words, if all available documents in the corpus are included in a posting list, the chances of retrieving all relevant documents would be perfect. However, as shown for the other model parameters, this comes at the cost of retrieval efficiency as shown in Figure 4.25. The larger the neighborhood radius is, the larger the index size would become as a result of a larger posting list and the slower the retrieval process will be. The relative importance of retrieval effectiveness versus retrieval efficiency would

be an important consideration in determining the best size of neighborhood radius. To clearly visualize the tradeoff, Figure 4.26 shows how changing the value of k impacts retrieval effectiveness and efficiency in the Robust04 collection.

As seen in Figure 4.26, a smaller size for k (k_1) leads to the retrieval of only 55.8% of the relevant documents while a larger k (k_3), which is four times larger than k1, retrieves 75.4%. The first observation is that the increase in the neighborhood radius does not linearly increase retrieval effectiveness. Second, the increase in the size of the posting list from k_1 to k_3 leads to a decreased retrieval time by four orders of magnitude, which is a considerable slow down on retrieval efficiency considering that retrieval time has significant impact on user satisfaction. There have been some studies that have suggested that users are primarily interested in the retrieved results that are presented to them at the very top of the result list. In such circumstances, adopting a small k such as k_1 might be a reasonable decision as retrieval time would be improved significantly. However, on the other hand in critical domains, e.g., patent search, where retrieving all possible relevant documents is of utmost importance and retrieval time is less important, adopting a larger k such as k_3 would be a better choice.

4.2.6 Final Results Synopsis

Our experiments have shown that it is possible to build an effective and efficient index based on the similarity of the vectors jointly learnt for keywords,

entities, types and documents. We found that there is a tradeoff between retrieval effectiveness and retrieval efficiency and model parameters and corpus characteristics can influence this tradeoff. Our main findings are:

- Our proposed indexing strategy shows noticeable improvement in terms of retrieval efficiency, i.e., index size and QPT, as shown in Table 4.8 across all document collections regardless of the characteristics of the document collection;

- From a retrieval effectiveness perspective, our proposed approach shows competitive or lower performance compared to the baseline inverted index on the ClueWeb'09 document collection while it shows improved performance on the Robust04 document collection as shown in Table 4.9.

- We found that the parameters used to train the joint embedding space can influence retrieval effectiveness and efficiency. First, an increased embedding dimension size can improve retrieval effectiveness on large document collections while higher dimension sizes do not necessarily lead to better performance for smaller collections as reported in Figure 4.21. Furthermore, as expected, larger embedding dimension sizes negatively impact retrieval efficiency. The length of the context window parameter can influence retrieval effectiveness but has inverse effect depending on the size of the document collection.

- Finally, our experiments showed that the neighborhood radius size impacts the tradeoff between retrieval effectiveness and efficiency. A larger neighborhood radius will result in a higher number of relevant retrieved documents but at the same increases index size and QPT. The inverse relation between retrieval effectiveness and efficiency is not linear and improved effectiveness could result in $4x$ worst performance on efficiency.

It is important to note that the tradeoff between retrieval efficiency and effectiveness depends on the objectives of the target domain for which the retrieval system is being designed and the characteristics of the document collection. The findings of our experiments assist in choosing the right parameters for an efficient and effective embedding-based index that does not necessarily operate based on explicit keyword-document association and works according to the similarity of the embedding representation of keywords, entities, types and documents.

4.3 Summary

This chapter discussed how the empirical experiments to evaluate the explicit semantic full-text index and implicit semantic index were performed. It then described the process of building these indices. We also discussed the efficiency and effectiveness of these two indices based on standard IR evaluation.

To identify the efficient data structure for building the explicit semantic full-text index; we implemented various prototypes of this index with Treap, Wavelet Tree and HashMap data structures in Java. We created Treap and Wavelet Tree data structures in Java while we use HashMap from an existing Java library. Moreover, all homogenous and heterogeneous integration approaches for these indices are implemented to identify the efficient approach for integrating Keyword Index and Entity Index. Based on our experimental results, the most efficient hybrid index for processing ranked intersection queries and Boolean intersection queries is TT, where we used homogenous Treap integration to retrieve final results. The efficient hybrid index for processing ranked union queries is HH, where the list-based integration approach is used. While the heterogeneous indices are not as efficient as hybrid indices. We also measured the relation between query process time and query length as properties for determining the efficient index data structure for the explicit semantic full-text index. Our experimental results illustrated that Treap indices are less sensitive to query length in contrast to Wavelet Tree indices since their query process time significantly increased when the length of the query became longer. HashMap-based Type Index was found to be the most efficient data structure for building Type Index based on our experimental results.

The implicit semantic full-text index was built in Python. To implement the joint embedded space of keywords, entities, types and documents Gensim was used. After building the PV model Annoy was used as the approximate

nearest neighbor search which was applied on the created joint embedded space to find top-k similar documents for each index key. In our empirical experiments to evaluate the retrieval efficiency and effectiveness of our index, we compared the implicit semantic full-text index with the baseline inverted index (Indri). Our experimental results illustrated that the retrieval efficiency of the implicit semantic full-text index improved significantly for all chosen document collections compared to the Indri (refer to Table 4.8). While, the retrieval effectiveness of our index showed competitive or lower performance compared to Indri as shown in Table 4.9. Furthermore, to create jointly embedded space, we need to consider the trade off between retrieval efficiency and retrieval effectiveness. Our experimental results illustrated that increasing neighborhood radius of approximate nearest neighbor search and PV model parameters: dimension size and context size, improved the effectiveness while it had inverse outcome on efficiency. It is also important to mention that, the relation between efficiency and effectiveness is not linear in the implicit semantic full-text index.

Chapter 5

Concluding Remarks

In this chapter, we will first summarize our main findings and draw conclusions for the thesis. Then we provide an outlook on related future work for building more efficient semantic full-text indices.

5.1 Technical Developments

In this thesis, we have systematically explored the possibility of building a semantic full-text index by applying efficient data structures proposed for keyword-based index; we refer to this index as the explicit semantic full-text index. Moreover, we investigated integration of textual and semantic information before indexing them. The integration is performed by applying neural embedding to create one space which represents all of these information. So, we can use the existing keyword-based indexing method for building

our index which is referred to as the implicit semantic full-text index.

In the explicit semantic full-text search approach, we studied the possibility of building indexing data structures for semantic search, which consists of the Keyword, Entity and Type Indices to support the process of retrieving both keyword-based and entity-bearing queries. These three indices store information related to keywords, entities and types derived from external knowledge bases.

In order to find an efficient indexing data structure, we explored using similar as well as dissimilar data structures for each of these three indices. Our experiments shows the most efficient explicit semantic full-text index for processing ranked intersection queries and Boolean intersection queries is homogeneous Treap indices (TT). On the other hand, homogeneous HashMap indices (HH) are the efficient explicit semantic index for processing ranked union queries. In addition, we integrate the results of the various index types using two approaches, namely list-based and non-list-based approaches. We have found that the list-based approach is more efficient compared to the non-list-based approach for heterogeneous indices.

We reported the HashMap Type Index is the efficient data structure for indexing hierarchical relations between entities of the explicit semantic full-text index.

Furthermore, computing the memory usage of these three data structures in

big-O notation shows that the Treap data structure requires the least amount of space to store index compared to HashMap and Wavelet Tree.

In the implicit semantic full-text indexing approach, we explored the possibility of building inverted indices for semantic full-text search engines not based on keywords occurrence in documents but based on the similarity of the vector representation of keywords, entities, types and, documents jointly learnt based on neural embeddings. To this end, we relaxed the main requirement of inverted indices, which require each posting related to a posting list to include the posting list key. Instead, we include postings in the posting list based on the degree of similarity of the document to the posting list key within the embedding space. We employ approximate nearest neighbour search to populate the proposed inverted index structure based on the jointly learnt embedding space.

We have evaluated our work based on publicly available and well accepted document collections and their related standard TREC query sets (topics) and compared its retrieval effectiveness and efficiency with a state of the art retrieval system. Summarily, we found that our approach shows noticeable improvement over the baseline in terms of retrieval efficiency and has better retrieval effectiveness on document collections Robust04 and Pooled Baseline. Furthermore, we systematically explored how different model parameters impact retrieval effectiveness and efficiency and methodically present our findings.

5.2 Future Work

We believe that our work has the potential to be extended in the following aspects:

- A potential venue for future work of this thesis will be combining the explicit and implicit semantic full-text indices to explore the effect of using different data structures on efficiency and effectiveness of the semantic full-text index. Using Treap data structure in building semantic index of implicit semantic full-text index instead of inverted index can improve efficiency based on our findings in the explicit semantic full-text index.

- In building the explicit semantic full-text index, we plan to add the ability of proximity search by storing positions of keywords and entities in the corpus. Therefore, we need to find the most efficient data structure to provide the ability to store positions along with other information of postings. For instance, Treap does not have this ability because each node can store only two values (TF, CV). Therefore, we need to identify a data structure for positional Keyword Index and positional Entity Index based on Wavelet Tree and HashMaps or heterogeneous semantic full-text indices to identify the most efficient data structure to support proximity search in semantic search.

- In building the implicit semantic full-text index, the paragraph vector models can be improve by applying two suggested strategies: vector

norms [1] and Content Tree Word Embedding [113]. The first strategy has shown to be related to both word frequency and document structures. This way, the effectiveness of our proposed index could be improved by integrating the language model and paragraph models because existing work have shown increased retrieval performance by integrating these two approaches [225, 90]. The Content Tree Word Embedding approach is a framework for representing documents while using word embedding feature learning. They update each word vector based on its location in the content tree. Therefore, they combine global pre-trained word embedding and local context. Based on empirical experiments, this approach outperforms local context-based approaches (e.g. Doc2Vec and bag-of-words) and two global pre-trained approaches, Word2Vec and Glove [163].

- We are also interested in exploring an alternative method for finding top-k documents related to each term based on the Word Movers Distance (WMD) [192] instead of the Euclidean distance that is currently used. WMD is a document-level similarity function which calculates the minimum traveling distance from the matching words of one document to another. Since, this technique was shown to be effective in document classification [114] it might also have the potential to improve retrieval. Furthemore, Kim et al. [116] have proposed word embeddings based on WMD for calculating similarity between query and documents when no exact matches are found between a query and

a document. This method showed an increase in mean average precision compared to a BM25 baseline and hence is another sign of the potential usefulness of WMD.

- Exploring maintenance aspects of the proposed indexing methods for systems which require efficient index updates such as systems that receive new documents at high rates or when context of documents change regularly is an important area to be further explored. We are interested in exploring three main strategies, namely, in-place update, re-merge update and re-build index [133] within the context of our work. A practical in-place update strategy stores new documents in a temporary buffer whenever the buffer is full, the new documents are sequentially integrated into the main index. The re-merge update strategy, on the other hand, utilizes algorithms for efficient index construction by dividing the document collection into blocks and building an index for each block. Then, sub-indices are merged for building the final index. The cost of re-merge strategy is higher than in-place update because of copying the whole index at each update, while its cost is lower than the re-build strategy.

Bibliography

[1] Qingyao Ai, Liu Yang, Jiafeng Guo, and W. Bruce Croft, *Analysis of the paragraph vector model for information retrieval*, Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval (New York, NY, USA), ICTIR '16, ACM, 2016, pp. 133–142.

[2] Areej Alasiry, Mark Levene, and Alexandra Poulovassilis, *Detecting candidate named entities in search queries*, Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval, ACM, 2012, pp. 1049–1050.

[3] Vo Ngoc Anh and Alistair Moffat, *Index compression using fixed binary codewords*, Proceedings of the 15th Australasian Database Conference - Volume 27 (Darlinghurst, Australia, Australia), ADC '04, Australian Computer Society, Inc., 2004, pp. 61–67.

[4] ———, *Simplified similarity scoring using term ranks*, Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 2005, pp. 226–233.

[5] ———, *Pruned query evaluation using pre-computed impacts*, Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 2006, pp. 372–379.

[6] Marcelo Arenas, Bernardo Cuenca Grau, Evgeny Kharlamov, Šarūnas Marciuška, and Dmitriy Zheleznyakov, *Faceted search over rdf-based knowledge graphs*, Web Semantics: Science, Services and Agents on the World Wide Web **37** (2016), 55–74.

[7] Sanjeev Arora, Yingyu Liang, and Tengyu Ma, *A simple but tough-to-beat baseline for sentence embeddings*, (2016).

[8] Diego Arroyuelo, Rodrigo Cánovas, Gonzalo Navarro, and Kunihiko Sadakane, *Succinct trees in practice*, Proceedings of the Meeting on Algorithm Engineering & Expermiments, Society for Industrial and Applied Mathematics, 2010, pp. 84–97.

[9] Diego Arroyuelo, Senén González, and Mauricio Oyarzún, *Compressed self-indices supporting conjunctive queries on document collections*, International Symposium on String Processing and Information Retrieval, Springer, 2010, pp. 43–54.

[10] Sören Auer, Christian Bizer, Georgi Kobilarov, Jens Lehmann, Richard Cyganiak, and Zachary Ives, *Dbpedia: A nucleus for a web of open data*, The semantic web, Springer, 2007, pp. 722–735.

[11] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull, *Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms*, Similarity Search and Applications - 10th International Conference, SISAP 2017, Munich, Germany, October 4-6, 2017, Proceedings, 2017, pp. 34–49.

[12] Ricardo Baeza-Yates, Alistair Moffat, and Gonzalo Navarro, *Searching large text collections*, Handbook of massive data sets, Springer, 2002, pp. 195–243.

[13] Ricardo Baeza-Yates and Berthier Ribeiro-Neto, *Modern information retrieval*, Addison Wesley, Harlow, 1999.

[14] Jérémy Barbay, Alejandro López-Ortiz, Tyler Lu, and Alejandro Salinger, *An experimental investigation of set intersection algorithms for text searching*, Journal of Experimental Algorithmics (JEA) **14** (2009), 7.

[15] Hannah Bast, Florian Bäurle, Bjorn Buchhold, and Elmar Haussmann, *Broccoli: Semantic full-text search at your fingertips*, arXiv preprint arXiv:1207.2615 (2012).

[16] Hannah Bast, Florian Bäurle, Björn Buchhold, and Elmar Haussmann, *A case for semantic full-text search*, Proceedings of the 1st Joint International Workshop on Entity-Oriented and Semantic Search, ACM, 2012, p. 4.

[17] Hannah Bast, Florian Bäurle, Bjorn Buchhold, and Elmar Haussmann, *Semantic full-text search with broccoli*, Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval (New York, NY, USA), SIGIR '14, ACM, 2014, pp. 1265–1266.

[18] Hannah Bast and Björn Buchhold, *An index for efficient semantic full-text search*, Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management (New York, NY, USA), CIKM '13, ACM, 2013, pp. 369–378.

[19] Hannah Bast, Björn Buchhold, Elmar Haussmann, et al., *Semantic search on text and knowledge bases*, Foundations and Trends® in Information Retrieval **10** (2016), no. 2-3, 119–271.

[20] Hannah Bast and Marjan Celikik, *Fast construction of the hyb index*, ACM Transactions on Information Systems (TOIS) **29** (2011), no. 3, 16.

[21] Holger Bast, Alexandru Chitea, Fabian Suchanek, and Ingmar Weber, *Ester: Efficient search on text, entities, and relations*, Proceedings of the 30th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '07, ACM, 2007, pp. 671–678.

[22] Holger Bast, Fabian Suchanek, and Ingmar Weber, *Semantic full-text search with ester: scalable, easy, fast*, 2008 IEEE International Conference on Data Mining Workshops, IEEE, 2008, pp. 959–962.

[23] Holger Bast and Ingmar Weber, *Type less, find more: fast autocompletion search with a succinct index*, Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 2006, pp. 364–371.

[24] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin, *A neural probabilistic language model*, J. Mach. Learn. Res. **3** (2003), 1137–1155.

[25] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin, *A neural probabilistic language model*, Journal of machine learning research **3** (2003), no. Feb, 1137–1155.

[26] Jon Louis Bentley, *Multidimensional binary search trees used for associative searching*, Commun. ACM **18** (1975), no. 9, 509–517.

[27] Shane Bergsma and Qin Iris Wang, *Learning noun phrase query segmentation*, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), 2007.

[28] Ravish Bhagdev, Sam Chapman, Fabio Ciravegna, Vitaveska Lanfranchi, and Daniela Petrelli, *Hybrid search: Effectively combining*

keywords and semantic searches, European Semantic Web Conference, Springer, 2008, pp. 554–568.

[29] Roi Blanco, Peter Mika, and Sebastiano Vigna, *Effective and efficient entity search in rdf data*, International Semantic Web Conference, Springer, 2011, pp. 83–97.

[30] Roi Blanco, Giuseppe Ottaviano, and Edgar Meij, *Fast and space-efficient entity linking for queries*, Proceedings of the Eighth ACM International Conference on Web Search and Data Mining, ACM, 2015, pp. 179–188.

[31] Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov, *Enriching word vectors with subword information*, Transactions of the Association for Computational Linguistics **5** (2017), 135–146.

[32] Kurt Bollacker, Colin Evans, Praveen Paritosh, Tim Sturge, and Jamie Taylor, *Freebase: a collaboratively created graph database for structuring human knowledge*, Proceedings of the 2008 ACM SIGMOD international conference on Management of data, AcM, 2008, pp. 1247–1250.

[33] Kalina Bontcheva, Valentin Tablan, and Hamish Cunningham, *Semantic search over documents and ontologies*, Bridging Between Information Retrieval and Databases, Springer, 2014, pp. 31–53.

[34] Andrew Borthwick, John Sterling, Eugene Agichtein, and Ralph Grishman, *Exploiting diverse knowledge sources via maximum entropy in named entity recognition*, Sixth Workshop on Very Large Corpora, 1998.

[35] Wladmir C Brandão, Rodrygo LT Santos, Nivio Ziviani, Edleno S de Moura, and Altigran S da Silva, *Learning to expand queries using entities*, Journal of the Association for Information Science and Technology **65** (2014), no. 9, 1870–1883.

[36] Andrei Z Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien, *Efficient query evaluation using a two-level retrieval process*, Proceedings of the twelfth international conference on Information and knowledge management, ACM, 2003, pp. 426–434.

[37] Razvan Bunescu and Marius Paşca, *Using encyclopedic knowledge for named entity disambiguation*, 11th conference of the European Chapter of the Association for Computational Linguistics, 2006.

[38] Stefan Büttcher and Charles L. A. Clarke, *Index compression is good, especially for random access*, Proceedings of the Sixteenth ACM Conference on Information and Knowledge Management, CIKM 2007, Lisbon, Portugal, November 6-10, 2007, 2007, pp. 761–770.

[39] Huanhuan Cao, Derek Hao Hu, Dou Shen, Daxin Jiang, Jian-Tao Sun, Enhong Chen, and Qiang Yang, *Context-aware query classification*,

Proceedings of the 32nd international ACM SIGIR conference on Research and development in information retrieval, ACM, 2009, pp. 3–10.

[40] David Carmel, Ming-Wei Chang, Evgeniy Gabrilovich, Bo-June Paul Hsu, and Kuansan Wang, *Erd'14: entity recognition and disambiguation challenge*, ACM SIGIR Forum, vol. 48, ACM, 2014, pp. 63–77.

[41] Matteo Catena, Craig Macdonald, and Iadh Ounis, *On inverted index compression for search engine efficiency*, Advances in Information Retrieval - 36th European Conference on IR Research, ECIR 2014, Amsterdam, The Netherlands, April 13-16, 2014. Proceedings, 2014, pp. 359–371.

[42] Soumen Chakrabarti, Sasidhar Kasturi, Bharath Balakrishnan, Ganesh Ramakrishnan, and Rohit Saraf, *Compressed data structures for annotated web search*, Proceedings of the 21st international conference on World Wide Web, ACM, 2012, pp. 121–130.

[43] Soumen Chakrabarti, Kriti Puniyani, and Sujatha Das, *Optimizing scoring functions and indexes for proximity search in type-annotated corpora*, Proceedings of the 15th international conference on World Wide Web, ACM, 2006, pp. 717–726.

[44] Soumen Chakrabarti, Devshree Sane, and Ganesh Ramakrishnan, *Web-scale entity-relation search architecture*, Proceedings of the 20th

international conference companion on World wide web, ACM, 2011, pp. 21–22.

[45] Xinxiong Chen, Lei Xu, Zhiyuan Liu, Maosong Sun, and Huan-Bo Luan, *Joint learning of character and word embeddings.*, IJCAI, 2015, pp. 1236–1242.

[46] Zhigang Chen, Wei Lin, Qian Chen, Xiaoping Chen, Si Wei, Hui Jiang, and Xiaodan Zhu, *Revisiting word embedding for contrasting meaning*, Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing of the Asian Federation of Natural Language Processing, ACL 2015, July 26-31, 2015, Beijing, China, Volume 1: Long Papers, 2015, pp. 106–115.

[47] Tao Cheng and Kevin Chen-Chuan Chang, *Beyond pages: supporting efficient, scalable entity search with dual-inversion index*, Proceedings of the 13th International Conference on Extending Database Technology, ACM, 2010, pp. 15–26.

[48] ———, *Beyond pages: Supporting efficient, scalable entity search with dual-inversion index*, Proceedings of the 13th International Conference on Extending Database Technology (New York, NY, USA), EDBT '10, ACM, 2010, pp. 15–26.

[49] Paolo Ciaccia, Marco Patella, and Pavel Zezula, *M-tree: An efficient access method for similarity search in metric spaces*, Proceedings of the 23rd International Conference on Very Large Data Bases (San Francisco, CA, USA), VLDB '97, Morgan Kaufmann Publishers Inc., 1997, pp. 426–435.

[50] Francisco Claude and Gonzalo Navarro, *Practical rank/select queries over arbitrary sequences*, International Symposium on String Processing and Information Retrieval, Springer, 2008, pp. 176–187.

[51] Stéphane Clinchant and Florent Perronnin, *Aggregating continuous word embeddings for information retrieval*, Proceedings of the Workshop on Continuous Vector Space Models and their Compositionality, 2013, pp. 100–109.

[52] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to algorithms*, MIT press, 2009.

[53] Marco Cornolti, Paolo Ferragina, and Massimiliano Ciaramita, *A framework for benchmarking entity-annotation systems*, 22nd International World Wide Web Conference, WWW '13, Rio de Janeiro, Brazil, May 13-17, 2013 (Daniel Schwabe, Virgílio A. F. Almeida, Hartmut Glaser, Ricardo A. Baeza-Yates, and Sue B. Moon, eds.), International World Wide Web Conferences Steering Committee / ACM, 2013, pp. 249–260.

[54] Matt Crane, Andrew Trotman, and Richard O'Keefe, *Maintaining discriminatory power in quantized indexes*, Proceedings of the 22nd ACM international conference on Information & Knowledge Management, ACM, 2013, pp. 1221–1224.

[55] W Bruce Croft, Michael Bendersky, Hang Li, and Gu Xu, *Query representation and understanding workshop.*, SIGIR Forum, vol. 44, 2010, pp. 48–53.

[56] W. Bruce Croft, Donald Metzler, and Trevor Strohman, *Search engines - information retrieval in practice*, Pearson Education, 2009.

[57] Andras Csomai and Rada Mihalcea, *Linking documents to encyclopedic knowledge*, IEEE Intelligent Systems **23** (2008), no. 5.

[58] Silviu Cucerzan, *Large-scale named entity disambiguation based on wikipedia data*, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL), 2007.

[59] Silviu Cucerzan and Eric Brill, *Spelling correction as an iterative process that exploits the collective knowledge of web users*, Proceedings of the 2004 Conference on Empirical Methods in Natural Language Processing, 2004.

[60] J. Shane Culpepper and Alistair Moffat, *Enhanced byte codes with restricted prefix properties*, String Processing and Information Retrieval,

12th International Conference, SPIRE 2005, Buenos Aires, Argentina, November 2-4, 2005, Proceedings, 2005, pp. 1–12.

[61] ———, *Compact set representation for information retrieval*, String Processing and Information Retrieval, 14th International Symposium, SPIRE 2007, Santiago, Chile, October 29-31, 2007, Proceedings, 2007, pp. 137–148.

[62] J Shane Culpepper and Alistair Moffat, *Compact set representation for information retrieval*, International Symposium on String Processing and Information Retrieval, Springer, 2007, pp. 137–148.

[63] J Shane Culpepper, Gonzalo Navarro, Simon J Puglisi, and Andrew Turpin, *Top-k ranked document search in general text databases*, European Symposium on Algorithms, Springer, 2010, pp. 194–205.

[64] Jeffrey Dalton, Laura Dietz, and James Allan, *Entity query feature expansion using knowledge base links*, Proceedings of the 37th international ACM SIGIR conference on Research & development in information retrieval, ACM, 2014, pp. 365–374.

[65] ———, *Entity query feature expansion using knowledge base links*, Proceedings of the 37th International ACM SIGIR Conference on Research & Development in Information Retrieval (New York, NY, USA), SIGIR '14, ACM, 2014, pp. 365–374.

[66] Edward Kai FUNG Dang, Robert Wing Pong Luk, and James Allan, *Fast forward index methods for pseudo-relevance feedback retrieval*, ACM Transactions on Information Systems (TOIS) **33** (2015), no. 4, 19.

[67] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman, *Indexing by latent semantic analysis*, Journal of the American society for information science **41** (1990), no. 6, 391–407.

[68] Renaud Delbru, Stephane Campinas, and Giovanni Tummarello, *Searching web data: An entity retrieval and high-performance indexing model*, Web Semantics: Science, Services and Agents on the World Wide Web **10** (2012), 33–58.

[69] Andrea Dessi and Maurizio Atzori, *A machine-learning approach to ranking rdf properties*, Future Generation Computer Systems **54** (2016), 366–377.

[70] Fernando Diaz, Bhaskar Mitra, and Nick Craswell, *Query expansion with locally-trained word embeddings*, Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics, ACL 2016, August 7-12, 2016, Berlin, Germany, Volume 1: Long Papers, 2016.

[71] Li Ding, Tim Finin, Anupam Joshi, Rong Pan, R Scott Cost, Yun Peng, Pavan Reddivari, Vishal Doshi, and Joel Sachs, *Swoogle: a search and*

metadata engine for the semantic web, Proceedings of the thirteenth ACM international conference on Information and knowledge management, ACM, 2004, pp. 652–659.

[72] Shuai Ding and Torsten Suel, *Faster top-k document retrieval using block-max indexes*, Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval, ACM, 2011, pp. 993–1002.

[73] Nemanja Djuric, Hao Wu, Vladan Radosavljevic, Mihajlo Grbovic, and Narayan Bhamidipati, *Hierarchical neural language models for joint representation of streaming documents and their content*, Proceedings of the 24th international conference on world wide web, International World Wide Web Conferences Steering Committee, 2015, pp. 248–255.

[74] Faezeh Ensan and Ebrahim Bagheri, *Document retrieval model through semantic linking*, Proceedings of the Tenth ACM International Conference on Web Search and Data Mining (New York, NY, USA), WSDM '17, ACM, 2017, pp. 181–190.

[75] Fredo Erxleben, Michael Günther, Markus Krötzsch, Julian Mendez, and Denny Vrandečić, *Introducing wikidata to the linked data web*, International Semantic Web Conference, Springer, 2014, pp. 50–65.

[76] Chris Faloutsos and Stavros Christodoulakis, *Signature files: An access method for documents and its analytical performance evaluation*, ACM Transactions on Information Systems (TOIS) **2** (1984), no. 4, 267–288.

[77] Christos Faloutsos and Stavros Christodoulakis, *Description and performance analysis of signature file methods for office filing*, ACM Transactions on Information Systems (TOIS) **5** (1987), no. 3, 237–257.

[78] Wei Fang, Jianwen Zhang, Dilin Wang, Zheng Chen, and Ming Li, *Entity disambiguation by knowledge and text jointly embedding*, Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11-12, 2016, 2016, pp. 260–269.

[79] Wei Fang, Jianwen Zhang, Dilin Wang, Zheng Chen, and Ming Li, *Entity disambiguation by knowledge and text jointly embedding*, Proceedings of the 20th SIGNLL conference on computational natural language learning, 2016, pp. 260–269.

[80] David C Faye, Olivier Curé, and Guillaume Blin, *A survey of rdf storage approaches*, Arima Journal **15** (2012), 11–35.

[81] Miriam Fernandez, Ivan Cantador, Vanesa Lopez, David Vallet, Pablo Castells, and Enrico Motta, *Semantically enhanced information retrieval: An ontology-based approach*, Web semantics: Science, services and agents on the world wide web **9** (2011), no. 4, 434–452.

[82] Francis C Fernández-Reyes, Jorge Hermosillo-Valadez, and Manuel Montes-y Gómez, *A prospect-guided global query expansion strategy using word embeddings*, Information Processing & Management **54** (2018), no. 1, 1–13.

[83] Paolo Ferragina and Giovanni Manzini, *Indexing compressed text*, Journal of the ACM (JACM) **52** (2005), no. 4, 552–581.

[84] Paolo Ferragina and Ugo Scaiella, *Tagme: on-the-fly annotation of short text fragments (by wikipedia entities)*, Proceedings of the 19th ACM international conference on Information and knowledge management, ACM, 2010, pp. 1625–1628.

[85] ———, *Tagme: On-the-fly annotation of short text fragments (by wikipedia entities)*, Proceedings of the 19th ACM International Conference on Information and Knowledge Management (New York, NY, USA), CIKM '10, ACM, 2010, pp. 1625–1628.

[86] ———, *Fast and accurate annotation of short texts with wikipedia pages*, IEEE software **29** (2012), no. 1, 70–75.

[87] Jenny Rose Finkel, Trond Grenager, and Christopher Manning, *Incorporating non-local information into information extraction systems by gibbs sampling*, Proceedings of the 43rd annual meeting on association for computational linguistics, Association for Computational Linguistics, 2005, pp. 363–370.

[88] Evgeniy Gabrilovich, Michael Ringgaard, and Amarnag Subramanya, *Facc1: Freebase annotation of clueweb corpora, version 1 (release date 2013-06-26, format version 1, correction level 0)*, Note: http://lemurproject. org/clueweb09/FACC1/Cited by **5** (2013).

[89] Travis Gagie, Gonzalo Navarro, and Simon J Puglisi, *New algorithms on wavelet trees and applications to information retrieval*, Theoretical Computer Science **426** (2012), 25–41.

[90] Debasis Ganguly, Dwaipayan Roy, Mandar Mitra, and Gareth J.F. Jones, *Word embedding based generalized language model for information retrieval*, Proceedings of the 38th International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '15, ACM, 2015, pp. 795–798.

[91] Fausto Giunchiglia, Uladzimir Kharkevich, and Ilya Zaihrayeu, *Concept search*, European Semantic Web Conference, Springer, 2009, pp. 429–444.

[92] Simon Gog and Matthias Petri, *Compact indexes for flexible top-k retrieval*, Annual Symposium on Combinatorial Pattern Matching, Springer, 2015, pp. 207–218.

[93] Mihajlo Grbovic, Nemanja Djuric, Vladan Radosavljevic, Fabrizio Silvestri, and Narayan Bhamidipati, *Context- and content-aware embeddings for query rewriting in sponsored search*, Proceedings of the 38th

International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '15, ACM, 2015, pp. 383–392.

[94] Roberto Grossi, Ankur Gupta, and Jeffrey Scott Vitter, *High-order entropy-compressed text indexes*, Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms, Society for Industrial and Applied Mathematics, 2003, pp. 841–850.

[95] Jiafeng Guo, Yixing Fan, Qingyao Ai, and W. Bruce Croft, *A deep relevance matching model for ad-hoc retrieval*, Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (New York, NY, USA), CIKM '16, ACM, 2016, pp. 55–64.

[96] Stephen Guo, Ming-Wei Chang, and Emre Kiciman, *To link or not to link? a study on end-to-end tweet entity linking*, Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2013, pp. 1020–1030.

[97] Matthias Hagen, Martin Potthast, Benno Stein, and Christof Bräutigam, *Query segmentation revisited*, Proceedings of the 20th international conference on World wide web, ACM, 2011, pp. 97–106.

[98] Xianpei Han and Jun Zhao, *Named entity disambiguation by leveraging wikipedia semantic knowledge*, Proceedings of the 18th ACM conference on Information and knowledge management, ACM, 2009, pp. 215–224.

[99] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker, *Yars2: A federated repository for querying graph structured data from the web*, The Semantic Web, Springer, 2007, pp. 211–224.

[100] Tatsunori B Hashimoto, David Alvarez-Melis, and Tommi S Jaakkola, *Word embeddings as metric recovery in semantic spaces*, Transactions of the Association for Computational Linguistics **4** (2016), 273–286.

[101] Faegheh Hasibi, Krisztian Balog, and Svein Erik Bratsberg, *Entity linking in queries: tasks and evaluation*, Proceedings of the 2015 International Conference on The Theory of Information Retrieval, ACM, 2015, pp. 171–180.

[102] ———, *Exploiting entity linking in queries for entity retrieval*, Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval (New York, NY, USA), ICTIR '16, ACM, 2016, pp. 209–218.

[103] Steffen Heinz and Justin Zobel, *Efficient single-pass index construction for text databases*, Journal of the American Society for Information Science and Technology **54** (2003), no. 8, 713–729.

[104] William R. Hersh, Andrew Turpin, Susan Price, Benjamin Chan, Dale Kraemer, Lynetta Sacherek, and Daniel Olson, *Do batch and user evaluation give the same results?*, Proc. SIGIR, 2000, pp. 17–24.

[105] William R. Hersh, Andrew Turpin, Lynetta Sacherek, Daniel Olson, Susan Price, Benjamin Chan, and Dale Kraemer, *Further analysis of whether batch and user evaluations give the same results with a question-answering task*, Proc. TREC, 2000.

[106] Zhiting Hu, Poyao Huang, Yuntian Deng, Yingkai Gao, and Eric P Xing, *Entity hierarchy embedding.*, ACL (1), 2015, pp. 1292–1300.

[107] Jian Huang, Jianfeng Gao, Jiangbo Miao, Xiaolong Li, Kuansan Wang, Fritz Behr, and C Lee Giles, *Exploring web scale language models for search query processing*, Proceedings of the 19th international conference on World wide web, ACM, 2010, pp. 451–460.

[108] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck, *Learning deep structured semantic models for web search using clickthrough data*, Proceedings of the 22Nd ACM International Conference on Information & Knowledge Management (New York, NY, USA), CIKM '13, ACM, 2013, pp. 2333–2338.

[109] Piotr Indyk and Rajeev Motwani, *Approximate nearest neighbors: Towards removing the curse of dimensionality*, Proceedings of the Thir-

tieth Annual ACM Symposium on Theory of Computing (New York, NY, USA), STOC '98, ACM, 1998, pp. 604–613.

[110] Shoaib Jameel, Zied Bouraoui, and Steven Schockaert, *Member: Max-margin based embeddings for entity retrieval*, Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '17, ACM, 2017, pp. 783–792.

[111] Kalervo Järvelin and Jaana Kekäläinen, *Cumulated gain-based evaluation of ir techniques*, ACM Transactions on Information Systems (TOIS) **20** (2002), no. 4, 422–446.

[112] Rosie Jones, Benjamin Rey, Omid Madani, and Wiley Greiner, *Generating query substitutions*, Proceedings of the 15th international conference on World Wide Web, ACM, 2006, pp. 387–396.

[113] Mehran Kamkarhaghighi and Masoud Makrehchi, *Content tree word embedding for document representation*, Expert Systems with Applications **90** (2017), 241–249.

[114] Tom Kenter and Maarten de Rijke, *Short text similarity with word embeddings*, Proceedings of the 24th ACM International on Conference on Information and Knowledge Management (New York, NY, USA), CIKM '15, ACM, 2015, pp. 1411–1420.

[115] Martin L Kersten, A Boncz Peter, and Sándor Héman, *Super-scalar database compression between ram and cpu-cache*, MS Thesis, Centrum voor Wiskunde en Informatica (CWI, Citeseer, 2005.

[116] Sun Kim, Nicolas Fiorini, W John Wilbur, and Zhiyong Lu, *Bridging the gap: Incorporating a semantic similarity measure for effectively mapping pubmed queries to documents*, Journal of biomedical informatics **75** (2017), 122–127.

[117] Roberto Konow and Gonzalo Navarro, *Dual-sorted inverted lists in practice*, International Symposium on String Processing and Information Retrieval, Springer, 2012, pp. 295–306.

[118] ———, *Dual-sorted inverted lists in practice*, International Symposium on String Processing and Information Retrieval, Springer, 2012, pp. 295–306.

[119] Roberto Konow, Gonzalo Navarro, Charles L. A. Clarke, and Alejandro López-Ortíz, *Inverted treaps*, ACM Trans. Inf. Syst. **35** (2017), no. 3, 22:1–22:45.

[120] Roberto Konow, Gonzalo Navarro, Charles L.A. Clarke, and Alejandro López-Ortíz, *Faster and smaller inverted indices with treaps*, Proceedings of the 36th International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '13, ACM, 2013, pp. 193–202.

[121] Roberto Konow, Gonzalo Navarro, Charles LA Clarke, and Alejandro López-Ortíz, *Faster and smaller inverted indices with treaps*, Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval, ACM, 2013, pp. 193–202.

[122] Robert R. Korfhage, *Information storage and retrieval*, Wiley, 1997.

[123] Matt Kusner, Yu Sun, Nicholas Kolkin, and Kilian Weinberger, *From word embeddings to document distances*, International Conference on Machine Learning, 2015, pp. 957–966.

[124] Matt J. Kusner, Yu Sun, Nicholas I. Kolkin, and Kilian Q. Weinberger, *From word embeddings to document distances*, Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015, 2015, pp. 957–966.

[125] Saar Kuzi, Anna Shtok, and Oren Kurland, *Query expansion using word embeddings*, Proceedings of the 25th ACM International on Conference on Information and Knowledge Management (New York, NY, USA), CIKM '16, ACM, 2016, pp. 1929–1932.

[126] Fatemeh Lashkari, Faezeh Ensan, Ebrahim Bagheri, and Ali A. Ghorbani, *Efficient indexing for semantic search*, Expert Syst. Appl. **73** (2017), 92–114.

[127] Victor Lavrenko and W. Bruce Croft, *Relevance based language models*, Proceedings of the 24th Annual International ACM SIGIR Conference

on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '01, ACM, 2001, pp. 120–127.

[128] Quoc Le and Tomas Mikolov, *Distributed representations of sentences and documents*, Proceedings of the 31st International Conference on Machine Learning (ICML-14), 2014, pp. 1188–1196.

[129] Quoc V. Le and Tomas Mikolov, *Distributed representations of sentences and documents*, Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014, 2014, pp. 1188–1196.

[130] Jens Lehmann, Robert Isele, Max Jakob, Anja Jentzsch, Dimitris Kontokostas, Pablo N Mendes, Sebastian Hellmann, Mohamed Morsey, Patrick Van Kleef, Sören Auer, et al., *Dbpedia–a large-scale, multilingual knowledge base extracted from wikipedia*, Semantic Web **6** (2015), no. 2, 167–195.

[131] Yuangui Lei, Victoria Uren, and Enrico Motta, *Semsearch: A search engine for the semantic web*, International Conference on Knowledge Engineering and Knowledge Management, Springer, 2006, pp. 238–245.

[132] Daniel Lemire and Leonid Boytsov, *Decoding billions of integers per second through vectorization*, Softw., Pract. Exper. **45** (2015), no. 1, 1–29.

[133] Nicholas Lester, Justin Zobel, and Hugh E Williams, *In-place versus re-build versus re-merge: index maintenance strategies for text retrieval systems*, Proceedings of the 27th Australasian conference on Computer science-Volume 26, Australian Computer Society, Inc., 2004, pp. 15–23.

[134] Rui Li, Linxue Hao, Xiaozhao Zhao, Peng Zhang, Dawei Song, and Yuexian Hou, *A query expansion approach using entity distribution based on markov random fields*, Asia Information Retrieval Symposium, Springer, 2015, pp. 387–393.

[135] Xiao Li, Ye-Yi Wang, and Alex Acero, *Learning query intent from regularized click graphs*, Proceedings of the 31st annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 2008, pp. 339–346.

[136] Xiaonan Li, Chengkai Li, and Cong Yu, *Entityengine: Answering entity-relationship queries using shallow semantics*, Proceedings of the 19th ACM international conference on Information and knowledge management, ACM, 2010, pp. 1925–1926.

[137] ———, *Structured querying of annotation-rich web text with shallow semantics*, Tech. report, Citeseer, 2010.

[138] Yuezhang Li, Ronghuo Zheng, Tian Tian, Zhiting Hu, Rahul Iyer, and Katia Sycara, *Joint embedding of hierarchical categories and entities*

for concept categorization and dataless classification, arXiv preprint arXiv:1607.07956 (2016).

[139] Jimmy Lin and Andrew Trotman, *Anytime ranking for impact-ordered indexes*, Proceedings of the 2015 International Conference on The Theory of Information Retrieval, ACM, 2015, pp. 301–304.

[140] Xitong Liu and Hui Fang, *Latent entity space: a novel retrieval approach for entity-bearing queries*, Information Retrieval Journal **18** (2015), no. 6, 473–503.

[141] Hans Peter Luhn, *A statistical approach to mechanized encoding and searching of literary information*, IBM Journal of research and development **1** (1957), no. 4, 309–317.

[142] Yongming Luo, François Picalausa, George HL Fletcher, Jan Hidders, and Stijn Vansummeren, *Storing and indexing massive rdf datasets*, Semantic search over the web, Springer, 2012, pp. 31–60.

[143] Laurens van der Maaten and Geoffrey Hinton, *Visualizing data using t-sne*, Journal of machine learning research **9** (2008), no. Nov, 2579–2605.

[144] Udi Manber and Gene Myers, *Suffix arrays: a new method for on-line string searches*, siam Journal on Computing **22** (1993), no. 5, 935–948.

[145] Christopher D Manning and P Raghavan, *H. schu tze. 2008. introduction to information retrieval*.

[146] Edgar Meij, Krisztian Balog, and Daan Odijk, *Entity linking and retrieval for semantic search*, Proceedings of the 7th ACM International Conference on Web Search and Data Mining (New York, NY, USA), WSDM '14, ACM, 2014, pp. 683–684.

[147] Edgar Meij, Wouter Weerkamp, and Maarten De Rijke, *Adding semantics to microblog posts*, Proceedings of the fifth ACM international conference on Web search and data mining, ACM, 2012, pp. 563–572.

[148] Pablo N Mendes, Max Jakob, Andrés García-Silva, and Christian Bizer, *Dbpedia spotlight: shedding light on the web of documents*, Proceedings of the 7th international conference on semantic systems, ACM, 2011, pp. 1–8.

[149] Donald Metzler and W. Bruce Croft, *A markov random field model for term dependencies*, Proceedings of the 28th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '05, ACM, 2005, pp. 472–479.

[150] Peter Mika, *Distributed indexing for semantic search*, Proceedings of the 3rd International Semantic Search Workshop, ACM, 2010, p. 3.

[151] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean, *Efficient estimation of word representations in vector space*, http://arxiv.org/abs/1301.3781 (2013).

[152] Tomáš Mikolov, Martin Karafiát, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur, *Recurrent neural network based language model*, Eleventh Annual Conference of the International Speech Communication Association, 2010.

[153] Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur, *Extensions of recurrent neural network language model*, Acoustics, Speech and Signal Processing (ICASSP), 2011 IEEE International Conference on, IEEE, 2011, pp. 5528–5531.

[154] Tomas Mikolov, Ilya Sutskever, Kai Chen, Gregory S. Corrado, and Jeffrey Dean, *Distributed representations of words and phrases and their compositionality*, Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States., 2013, pp. 3111–3119.

[155] David Milne and Ian H Witten, *Learning to link with wikipedia*, Proceedings of the 17th ACM conference on Information and knowledge management, ACM, 2008, pp. 509–518.

[156] Jose G. Moreno, Romaric Besançon, Romain Beaumont, Eva D'hondt, Anne-Laure Ligozat, Sophie Rosset, Xavier Tannier, and Brigitte Grau, *Combining word and entity embeddings for entity linking*, The Semantic Web - 14th International Conference, ESWC 2017, Portorož, Slovenia, May 28 - June 1, 2017, Proceedings, Part I, 2017, pp. 337–352.

[157] Eric Nalisnick, Bhaskar Mitra, Nick Craswell, and Rich Caruana, *Improving document ranking with dual word embeddings*, Proceedings of the 25th International Conference Companion on World Wide Web (Republic and Canton of Geneva, Switzerland), WWW '16 Companion, International World Wide Web Conferences Steering Committee, 2016, pp. 83–84.

[158] Gonzalo Navarro, *Wavelet trees for all*, Journal of Discrete Algorithms **25** (2014), 2–20.

[159] Gonzalo Navarro and Simon J Puglisi, *Dual-sorted inverted lists*, International Symposium on String Processing and Information Retrieval, Springer, 2010, pp. 309–321.

[160] Gonzalo Navarro, Simon J Puglisi, and Daniel Valenzuela, *General document retrieval in compact space*, Journal of Experimental Algorithmics (JEA) **19** (2015), 2–3.

[161] Eyal Oren, Renaud Delbru, Michele Catasta, Richard Cyganiak, Holger Stenzhorn, and Giovanni Tummarello, *Sindice. com: a document-oriented lookup index for open linked data*, International Journal of Metadata, Semantics and Ontologies **3** (2008), no. 1, 37–52.

[162] Matteo Pagliardini, Prakhar Gupta, and Martin Jaggi, *Unsupervised learning of sentence embeddings using compositional n-gram features*, arXiv preprint arXiv:1703.02507 (2017).

[163] Jeffrey Pennington, Richard Socher, and Christopher Manning, *Glove: Global vectors for word representation*, Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP), 2014, pp. 1532–1543.

[164] Michael Persin, Justin Zobel, and Ron Sacks-Davis, *Filtered document retrieval with frequency-sorted indexes*, JASIS **47** (1996), no. 10, 749–764.

[165] Matthias Petri, J Shane Culpepper, and Alistair Moffat, *Exploring the magic of wand*, Proceedings of the 18th Australasian Document Computing Symposium, ACM, 2013, pp. 58–65.

[166] Jay M Ponte and W Bruce Croft, *A language modeling approach to information retrieval*, Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 1998, pp. 275–281.

[167] Roman Prokofyev, Gianluca Demartini, and Philippe Cudré-Mauroux, *Effective named entity recognition for idiosyncratic web collections*, Proceedings of the 23rd international conference on World wide web, ACM, 2014, pp. 397–408.

[168] Hadas Raviv, Oren Kurland, and David Carmel, *Document retrieval using entity-based language models*, Proceedings of the 39th International

ACM SIGIR conference on Research and Development in Information Retrieval, ACM, 2016, pp. 65–74.

[169] Radim Řehůřek and Petr Sojka, *Software Framework for Topic Modelling with Large Corpora*, Proceedings of the LREC 2010 Workshop on New Challenges for NLP Frameworks (Valletta, Malta), ELRA, May 2010, `http://is.muni.cz/publication/884893/en`, pp. 45–50 (English).

[170] Knut Magne Risvik, Tomasz Mikolajewski, and Peter Boros, *Query segmentation for web search.*, WWW (Posters), 2003.

[171] Stephen Robertson and Hugo Zaragoza, *The probabilistic relevance framework: Bm25 and beyond*, Found. Trends Inf. Retr. **3** (2009), no. 4, 333–389.

[172] Stephen E Robertson, *The probability ranking principle in ir*, Journal of documentation **33** (1977), no. 4, 294–304.

[173] Yossi Rubner, Carlo Tomasi, and Leonidas J Guibas, *A metric for distributions with applications to image databases*, Computer Vision, 1998. Sixth International Conference on, IEEE, 1998, pp. 59–66.

[174] Kunihiko Sadakane and Gonzalo Navarro, *Fully-functional succinct trees*, Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2010, pp. 134–149.

[175] Haşim Sak, Andrew Senior, and Françoise Beaufays, *Long short-term memory recurrent neural network architectures for large scale acoustic modeling*, Fifteenth annual conference of the international speech communication association, 2014.

[176] Sherif Sakr and Ghazi Al-Naymat, *Relational processing of rdf queries: a survey*, ACM SIGMOD Record **38** (2010), no. 4, 23–28.

[177] Gerard Salton, *Automatic information organization and retrieval*, (1968).

[178] Gerard Salton and Michael J McGill, *Introduction to modern information retrieval*, (1986).

[179] David Sánchez, Montserrat Batet, David Isern, and Aida Valls, *Ontology-based semantic similarity: A new feature-based approach*, Expert Systems with Applications **39** (2012), no. 9, 7718–7728.

[180] Peter Sanders and Frederik Transier, *Intersection in integer inverted indices*, Proceedings of the Nine Workshop on Algorithm Engineering and Experiments, ALENEX 2007, New Orleans, Louisiana, USA, January 6, 2007, 2007.

[181] K Saruladha, G Aghila, and Sathish Kumar Penchala, *Design of new indexing techniques based on ontology for information retrieval systems*, International Conference on Advances in Information and Communication Technologies, Springer, 2010, pp. 287–291.

[182] Falk Scholer, Hugh E. Williams, John Yiannis, and Justin Zobel, *Compression of inverted indexes for fast query evaluation*, Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '02, ACM, 2002, pp. 222–229.

[183] Hinrich Schütze, Christopher D Manning, and Prabhakar Raghavan, *Introduction to information retrieval*, vol. 39, Cambridge University Press, 2008.

[184] Andy Seaborne, Geetha Manjunath, Chris Bizer, John Breslin, Souripriya Das, Ian Davis, Steve Harris, Kingsley Idehen, Olivier Corby, Kjetil Kjernsmo, et al., *Sparql/update: A language for updating rdf graphs*, W3c member submission **15** (2008).

[185] Dou Shen, Jian-Tao Sun, Qiang Yang, and Zheng Chen, *Building bridges for web query classification*, Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval, ACM, 2006, pp. 131–138.

[186] Vered Shwartz and Ido Dagan, *Cogalex-v shared task: Lexnet - integrated path-based and distributional method for the identification of semantic relations*, Proceedings of the 5th Workshop on Cognitive Aspects of the Lexicon, CogALex@COLING 2016, Osaka, Japan, December 12, 2016, 2016, pp. 80–85.

[187] Amanda Spink and Charles Cole (eds.), *New directions in cognitive information retrieval*, Springer, 2005.

[188] Alexander A. Stepanov, Anil R. Gangolli, Daniel E. Rose, Ryan J. Ernst, and Paramjit S. Oberoi, *Simd-based decoding of posting lists*, Proceedings of the 20th ACM International Conference on Information and Knowledge Management (New York, NY, USA), CIKM '11, ACM, 2011, pp. 317–326.

[189] Trevor Strohman, Donald Metzler, Howard Turtle, and W Bruce Croft, *Indri: A language model-based search engine for complex queries*, Proceedings of the International Conference on Intelligent Analysis, vol. 2, Amherst, MA, USA, 2005, pp. 2–6.

[190] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum, *Yago: a core of semantic knowledge*, Proceedings of the 16th international conference on World Wide Web, ACM, 2007, pp. 697–706.

[191] ———, *Yago: A large ontology from wikipedia and wordnet*, Web Semantics: Science, Services and Agents on the World Wide Web **6** (2008), no. 3, 203–217.

[192] Kohei Sugawara, Hayato Kobayashi, and Masajiro Iwasaki, *On approximately searching for similar word embeddings.*, ACL (1), 2016.

[193] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney, *Lstm neural networks for language modeling*, Thirteenth annual conference of the international speech communication association, 2012.

[194] Valentin Tablan, Kalina Bontcheva, Ian Roberts, and Hamish Cunningham, *Mímir: An open-source semantic search framework for interactive information seeking and discovery*, Web Semantics: Science, Services and Agents on the World Wide Web **30** (2015), 52–68.

[195] Valentin Tablan, Kalina Bontcheva, Ian Roberts, and Hamish Cunningham, *Mímir: An open-source semantic search framework for interactive information seeking and discovery*, J. Web Sem. **30** (2015), 52–68.

[196] Bin Tan and Fuchun Peng, *Unsupervised query segmentation using generative language models and wikipedia*, Proceedings of the 17th international conference on World Wide Web, ACM, 2008, pp. 347–356.

[197] Jaime Teevan, *Search, re-search*, 4 2017, Keynote at the 39th European Conference on Information Retrieval, Aberdeen, Scotland [Accessed: 2018 01 21].

[198] Alberto Tonon, Michele Catasta, Roman Prokofyev, Gianluca Demartini, Karl Aberer, and Philippe Cudré-Mauroux, *Contextualized ranking of entity types based on knowledge graphs*, Web Semantics: Science, Services and Agents on the World Wide Web **37** (2016), 170–183.

[199] Kristina Toutanova, Danqi Chen, Patrick Pantel, Hoifung Poon, Pallavi Choudhury, and Michael Gamon, *Representing text for joint embedding of text and knowledge bases.*, EMNLP, vol. 15, 2015, pp. 1499–1509.

[200] Lap Q. Trieu, Huy Q. Tran, and Minh-Triet Tran, *News classification from social media using twitter-based doc2vec model and automatic query expansion*, Proceedings of the Eighth International Symposium on Information and Communication Technology (New York, NY, USA), SoICT 2017, ACM, 2017, pp. 460–467.

[201] Andrew Trotman, *Compression, simd, and postings lists*, Proceedings of the 2014 Australasian Document Computing Symposium (New York, NY, USA), ADCS '14, ACM, 2014, pp. 50:50–50:57.

[202] Andrew Turpin and William R. Hersh, *Why batch and user evaluations do not give the same results*, Proc. SIGIR, 2001, pp. 225–231.

[203] ———, *User interface effects in past batch versus user experiments*, Proc. SIGIR, 2002, pp. 431–432.

[204] Ruben Verborgh, Miel Vander Sande, Olaf Hartig, Joachim Van Herwegen, Laurens De Vocht, Ben De Meester, Gerald Haesendonck, and Pieter Colpaert, *Triple pattern fragments: A low-cost knowledge graph interface for the web*, Web Semantics: Science, Services and Agents on the World Wide Web **37** (2016), 184–206.

[205] Denny Vrandečić and Markus Krötzsch, *Wikidata: a free collaborative knowledgebase*, Communications of the ACM **57** (2014), no. 10, 78–85.

[206] Haofen Wang, Qiaoling Liu, Thomas Penin, Linyun Fu, Lei Zhang, Thanh Tran, Yong Yu, and Yue Pan, *Semplore: A scalable ir approach to search the web of data*, Web Semantics: Science, Services and Agents on the World Wide Web **7** (2009), no. 3, 177–188.

[207] Zhen Wang, Jianwen Zhang, Jianlin Feng, and Zheng Chen, *Knowledge graph and text jointly embedding.*, EMNLP, vol. 14, 2014, pp. 1591–1601.

[208] Hugh E. Williams and Justin Zobel, *Compressing integers for fast file access*, Comput. J. **42** (1999), no. 3, 193–201.

[209] Ian H Witten, Alistair Moffat, and Timothy C Bell, *Managing gigabytes: compressing and indexing documents and images*, Morgan Kaufmann, 1999.

[210] Chenyan Xiong and Jamie Callan, *Esdrank: Connecting query and documents through external semi-structured data*, Proceedings of the 24th ACM International on Conference on Information and Knowledge Management, ACM, 2015, pp. 951–960.

[211] ———, *Query expansion with freebase*, Proceedings of the 2015 international conference on the theory of information retrieval, ACM, 2015, pp. 111–120.

[212] Chenyan Xiong, Jamie Callan, and Tie-Yan Liu, *Word-entity duet representations for document ranking*, Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '17, ACM, 2017, pp. 763–772.

[213] Mohamed Yahya, Denilson Barbosa, Klaus Berberich, Qiuyue Wang, and Gerhard Weikum, *Relationship queries on extended knowledge graphs*, Proceedings of the Ninth ACM International Conference on Web Search and Data Mining, ACM, 2016, pp. 605–614.

[214] Ikuya Yamada, Hiroyuki Shindo, Hideaki Takeda, and Yoshiyasu Takefuji, *Joint learning of the embedding of words and entities for named entity disambiguation*, Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning, CoNLL 2016, Berlin, Germany, August 11-12, 2016, 2016, pp. 250–259.

[215] Ikuya Yamada, Hiroyuki Shindo, Hideaki Takeda, and Yoshiyasu Takefuji, *Joint learning of the embedding of words and entities for named entity disambiguation*, The SIGNLL Conference on Computational Natural Language Learning (CoNLL), 2016.

[216] Hao Yan, Shuai Ding, and Torsten Suel, *Inverted index compression and query processing with optimized document ordering*, Proceedings of the 18th International Conference on World Wide Web (New York, NY, USA), WWW '09, ACM, 2009, pp. 401–410.

[217] Yiming Yang, *An evaluation of statistical approaches to text categorization*, Information retrieval **1** (1999), no. 1-2, 69–90.

[218] Yiming Yang and Xin Liu, *A re-examination of text categorization methods*, Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '99, ACM, 1999, pp. 42–49.

[219] Hamed Zamani and W. Bruce Croft, *Embedding-based query language models*, Proceedings of the 2016 ACM International Conference on the Theory of Information Retrieval (New York, NY, USA), ICTIR '16, ACM, 2016, pp. 147–156.

[220] Jiangong Zhang, Xiaohui Long, and Torsten Suel, *Performance of compressed inverted list caching in search engines*, Proceedings of the 17th International Conference on World Wide Web (New York, NY, USA), WWW '08, ACM, 2008, pp. 387–396.

[221] Ye Zhang, Md. Mustafizur Rahman, Alex Braylan, Brandon Dang, Heng-Lu Chang, Henna Kim, Quinten McNamara, Aaron Angert, Edward Banner, Vivek Khetan, Tyler McDonnell, An Thanh Nguyen, Dan Xu, Byron C. Wallace, and Matthew Lease, *Neural information retrieval: A literature review*, http://arxiv.org/abs/1611.06792 (2016).

[222] Justin Zobel and Alistair Moffat, *Inverted files for text search engines*, ACM Comput. Surv. **38** (2006), no. 2.

[223] ———, *Inverted files for text search engines*, ACM computing surveys (CSUR) **38** (2006), no. 2, 6.

[224] Justin Zobel, Alistair Moffat, and Kotagiri Ramamohanarao, *Inverted files versus signature files for text indexing*, ACM Transactions on Database Systems (TODS) **23** (1998), no. 4, 453–490.

[225] Bin Zou, Vasileios Lampos, Shangsong Liang, Zhaochun Ren, Emine Yilmaz, and Ingemar Cox, *A concept language model for ad-hoc retrieval*, Proceedings of the 26th International Conference on World Wide Web Companion (Republic and Canton of Geneva, Switzerland), WWW '17 Companion, International World Wide Web Conferences Steering Committee, 2017, pp. 885–886.

[226] Guido Zuccon, Bevan Koopman, Peter Bruza, and Leif Azzopardi, *Integrating and evaluating neural word embeddings in information retrieval*, Proceedings of the 20th Australasian Document Computing Symposium (New York, NY, USA), ADCS '15, ACM, 2015, pp. 12:1–12:8.

[227] Marcin Zukowski, Sándor Héman, Niels Nes, and Peter A. Boncz, *Super-scalar RAM-CPU cache compression*, Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA, 2006, p. 59.

[228] Stefan Zwicklbauer, Christin Seifert, and Michael Granitzer, *Robust and collective entity disambiguation through semantic embeddings*, Pro-

ceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval (New York, NY, USA), SIGIR '16, ACM, 2016, pp. 425–434.

Vita

Candidate's full name: Fatemeh Lashkari
University attended (with dates and degrees obtained):

University of Gothenburg, Sweden
Master of Science in Computer Science, 2012

Sharif University of Technology, Iran
Bachelor of Software Engineering, 2009

Publications:

- Fatemeh Lashkari, Ebrahim Bagheri, and Ali A. Ghorbani, Neural embedding-based indices for semantic search, Information Processing & Management 56 (2019), 733-755.
- Fatemeh Lashkari, Faezeh Ensan, Ebrahim Bagheri, and Ali A. Ghorbani, Efficient indexing for semantic search, Expert Syst. Appl. 73 (2017), 92-114.