EXPLOITING CLUSTER-SKIPPING INVERTED INDEX STRUCTURE FOR SEMANTIC PLACE RETRIEVAL

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY

ENES RECEP ÇINAR

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

NOVEMBER 2022

Approval of the thesis:

EXPLOITING CLUSTER-SKIPPING INVERTED INDEX STRUCTURE FOR SEMANTIC PLACE RETRIEVAL

submitted by **ENES RECEP ÇINAR** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Engineering Department, Middle East Technical University** by,

Prof. Dr. Halil Kalıpçılar
Dean, Graduate School of **Natural and Applied Sciences** _____

Prof. Dr. Halit Oğuztüzün
Head of Department, **Computer Engineering** _____

Assoc. Prof. Dr. İsmail Sengör Altıngövde
Supervisor, **Computer Engineering, METU** _____

Examining Committee Members:

Prof. Dr. Pınar Karagöz
Computer Engineering, METU _____

Assoc. Prof. Dr. İsmail Sengör Altıngövde
Computer Engineering, METU _____

Assist. Prof. Dr. Engin Demir
Computer Engineering, Hacettepe University _____

Date:30.11.2022

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Surname: Enes Recep Çınar

Signature :

ABSTRACT

EXPLOITING CLUSTER-SKIPPING INVERTED INDEX STRUCTURE FOR SEMANTIC PLACE RETRIEVAL

Çınar, Enes Recep
M.S., Department of Computer Engineering
Supervisor: Assoc. Prof. Dr. İsmail Sengör Altıngövde

November 2022, 53 pages

Semantic place retrieval is a popular research problem that aims to search over knowledge graphs using both text and location information. While handling such queries, it is crucial to appropriately balance the relevance and spatial distance of places to the user's query, to satisfy the user's information needs. Furthermore, given modern users' expectations, it is also critical to return results in a short time, which implies the necessity of using advanced index structures in the underlying retrieval system.

In this work, our contribution toward improving the efficiency of semantic place retrieval is two-fold. First, we show that by applying some ad hoc yet intuitive restrictions on the depth of search on the knowledge graph, it is possible to adopt several well-known index structures, so-called geo-textual indices that are introduced for processing the spatial keyword queries, for the semantic place retrieval scenario. Secondly, as a novel solution to the semantic place retrieval problem, we adapt the idea of cluster-skipping inverted index (CS-IIS), which has been originally proposed for retrieval over topically clustered document collections. In our adaptation, we also use an early-stopping technique based on the textual and spatial scores of the spatial grids that are being processed.

Our exhaustive experiments lead to several interesting findings. We show that while some of the earlier geo-textual indices in the literature yield high efficiency in terms of in memory processing time, they may cause a large number of direct disk accesses. In contrast, our approach based on CS-IIS requires a few direct disk accesses (which is equal to the number of terms in the query) and hence, performs considerably better than the baseline approaches in terms of the total query processing time.

Keywords: Semantic place retrieval, indexing, query processing

ÖZ

ANLAMSAL MEKAN GETİRİMİ İÇİN KÜME ATLAMALI TERS DİZİN YAPISINDAN YARARLANMA

Çınar, Enes Recep
Yüksek Lisans, Bilgisayar Mühendisliği Bölümü
Tez Yöneticisi: Doç. Dr. İsmail Sengör Altıngövde

Kasım 2022 , 53 sayfa

Anlamsal mekan getirimi, hem metin hem de konum bilgisini kullanarak bilgi grafları üzerinde arama yapmayı amaçlayan popüler bir araştırma problemidir. Bu tür sorguları ele alırken, kullanıcının bilgi gereksinimini karşılamak için, mekanların alakalılık düzeyini ve mekansal uzaklığını kullanıcının sorgusuna uygun şekilde dengelemek çok önemlidir. Ayrıca, modern kullanıcıların beklentileri göz önüne alındığında, sonuçların kısa sürede sağlanması ise kritiktir, bu da altta çalışan bilgi getirimi sistemlerinde gelişmiş indeks yapılarının kullanılması gerekliliğini beraberinde getirir.

Bu çalışmada, anlamsal mekan getiriminin verimliliğini artırmaya yönelik iki yönlü katkımız bulunmaktadır. İlk olarak, bilgi grafında arama derinliğine bazı sezgisel kısıtlamalar uygulayarak, mekansal kelime sorgularını işlemek için önerilen ve coğrafi metin indeksleri olarak adlandırılan birkaç iyi bilinen indeks yapısını elimizdeki problem için kullanmanın mümkün olduğunu gösteriyoruz. İkinci olarak, anlamsal mekan edinimi sorununa özgün bir çözüm olarak, aslında konuya göre kümelenmiş metinler üzerinde bilgi getirme için önerilmiş bir yapı olan küme atlamalı ters çevrilmiş indeks (CS-IIS) fikrini uyarlıyoruz. Uyarlamamızda, işlenmekte olan mekansal bölgelerin

metinsel ve mekansal skorlarına dayalı bir erken durdurma tekniği de kullanıyoruz.

Kapsamlı deneylerimiz bir çok ilginç bulguya yol açıyor. Literatürdeki bilinen bazı mekansal metin indekslerinin bellek içi işlem süresi açısından yüksek verimlilik sağlarken, çok sayıda doğrudan disk erişimine neden olabileceğini gösteriyoruz. Buna karşılık, CS-IIS'ye dayalı yaklaşımımız, az sayıda doğrudan disk erişimi gerektirir (bu da sorgudaki terimlerin sayısına eşittir) ve bu nedenle, toplam sorgu işleme süresi açısından bahsedilen temel yaklaşımlardan önemli ölçüde daha iyi performans göstermektedir.

Anahtar Kelimeler: Anlamsal mekan getirimi, indeksleme, sorgu işleme

To my dear wife and family...

ACKNOWLEDGMENTS

I'd love to thank my supervisor, Assoc. Prof. Dr. İsmail Sengör Altıngövde for his guidance and help throughout the whole MSc period.

Also, I'd like to thank my Examining Committee Members Prof. Dr. Pınar Karagöz and Assist. Prof. Dr. Engin Demir for their feedback.

Finally, I'd like to thank my wife for her full support, patience, and all of her feedback.

TABLE OF CONTENTS

LIST OF TABLES

TABLES

LIST OF FIGURES

FIGURES

LIST OF ABBREVIATIONS

CS-IIS	Cluster-Skipping Inverted Index Structure
RDF	Resource Description Framework
kNN	k Nearest Neighbor
MBR	Minimum Bounding Rectangle
S2I	Spatial Inverted Index
TS	Text primary Spatio-Textual Index
SKIF	Spatial Keyword Inverted File
TQSP	Top-k Tightest Qualified Semantic Places
BSP	Basic Semantic Place
SPP	Semantic Place Search with Pruning
BFS	Breadth First Search
YAGO	Yet Another Great Ontology
DIR-Tree	Document Similarity Enhanced Inverted File R-tree
IR-Tree	Inverted File R-Tree

CHAPTER 1

INTRODUCTION

1.1 Motivation and Problem Definition

Due to the wide usage of smartphones in our daily lives, location-based keyword search is becoming increasingly popular among users, who may like to search for nearby *cafes* selling *Colombian coffee* during a work-break, or *museums* including *Renaissance paintings* during a touristic trip. In [3], the authors show that 18.6% of the queries have geographical entries. This percentage is even higher for mobile devices, which is around 53% [4].

Location-based queries are at the core of several applications, such as Google Maps, Foursquare, and Booking.com. Furthermore, as several web documents are also being geo-tagged [5] in these days, such queries are also submitted to general search engines. Traditional search systems based on plain inverted indexing cannot produce an answer for these types of queries. Therefore, addressing location-based keyword queries is an attractive research direction.

Recent works [2] [6] [7] define a particular type of this problem, so-called semantic place retrieval, where place search is conducted over knowledge graphs that store various facts about places. In such a graph, places and other (non-place) entities are represented as nodes, and relationships between pairs of nodes are represented as edges, as illustrated in Figure 1.1. In the figure, each node is associated with terms extracted from the textual description of the entity. In contrast, relationship types between nodes, which are rather obvious, are not shown for brevity. Note that, in Figure 1.1, as our primary goal is retrieving places, such nodes are distinguished and shown as squares, while other nodes are shown as circles.

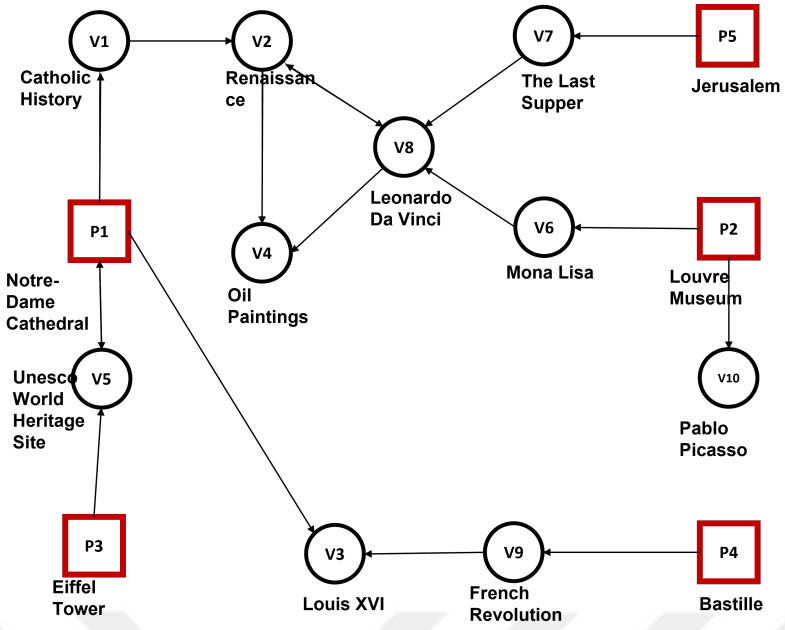

Figure 1.1: Example knowledge graph for place retrieval scenario

Assume a user at the given location, issues the query *Renaissance Paintings* on the graph shown in Figure 1.1. Our goal is to score and rank the places so that the user can get a relevant and spatially close list of places as the result set. Thus, while computing such scores, we focus on both relevances to the query (i.e., textual score) and the spatial distance of the places to the user (i.e., spatial score). To this end, we employ the scoring function presented in [2]. In particular, while computing relevance, we make use of the number of nodes that should be traversed to reach the query keywords from a given place. Applying "Ranked AND" query processing logic, a place is considered as a result candidate only if all query keywords are reachable from this place. For our example query, node P1 (*Notre-Dame Cathedral*) is a candidate place as the keywords *Renaissance* and *Paintings* are reachable from this place by following paths of length 2 and 3, respectively. Obviously, a place should have a higher textual score if the paths from this place to query keywords are shorter. Then, the spatial distance score is simply computed as the Euclidean distance between the coordinates of the query point and candidate place. The final score of a place is a linear combination of the textual and spatial scores.

To address the semantic place retrieval problem described above, in [2], the authors

propose to dynamically traverse the knowledge graph to compute the aforementioned textual score for each place, in the ascending order of spatial distance of the places to the query. While the closest places to the query can be easily obtained using an R-tree, the graph traversal is an expensive step, and indeed, our experiments reveal that even with some optimizations proposed in the latter work, it is prohibitively costly to be applied on-the-fly.

In this thesis, we address the semantic place retrieval problem from an alternative perspective. We hypothesize that while computing the textual score, it would not be helpful to consider nodes that are more than a few edges away from the place being considered. Thus, we propose to limit the depth of graph traversal to a predefined value. This intuitive choice allows us to avoid conducting an online graph traversal and instead, use pre-built index files while computing the textual score.

1.2 Contributions and Novelties

In this work, we have a number of contributions listed as follows:

- We apply two well-known geo-indexing techniques (namely, IR-Tree and DIR-Tree [5]) to the semantic place retrieval problem. To this end, we extended the implementation of these approaches provided in a public code base [1] of an earlier work [8] to support Ranked AND query processing logic.

- We adapt cluster-skipping inverted index structure (CS-IIS) [9] [10] [11], normally used for query document matching in cluster-based search, to improve the efficiency of semantic place retrieval over knowledge graphs. Our query processing strategy over CS-IIS involves an early-stopping strategy (similar to that in [12]) based on both textual and spatial scores of spatial grids.

- Our CS-IIS based solution is also implemented within the aforementioned code base to be shared with the research community.

[1] https://web.archive.org/web/20220227042457/http://lisi.io/spatial-keyword%20code.zip

1.3 The Outline of the Thesis

In Chapter 2, we review the existing methods in the literature. In particular, we discuss the grid-based and R-Tree based methods and explain how they differ from our work. In Chapter 3, we first show how some of the existing indexing structures can be used for the problem of semantic place retrieval on the knowledge graphs, and then present our approach using CS-IIS [9] with an early-stopping strategy. Chapters 4 and 5 provide the experimental setup and the results of our evaluations, respectively. Finally, in Chapter 6, we conclude and point to future work directions.

CHAPTER 2

RELATED WORK

In this thesis, we focus on the problem of place search on knowledge graphs, which is closely related to the location-based keyword search. In[8], the authors state that there are three widely employed types of such queries, which can be expressed in our context as follows:

1. **Boolean kNN Query** aims to retrieve a set of objects that are nearest to the query location such that every object contains all of the query keywords.

2. **Ranked AND Query** aims to retrieve a set of objects that are sorted by the score, which is a combination of spatial distance to the query point and textual score. The latter is based on the path distance of each query keyword to the place on the knowledge graph and due to the conjunctive processing logic, all keywords should be reachable from the place. In the ranking queries, documents are assigned a matching score according to their similarity to the query, using the vector space model [13].

3. **Range Query** aims to retrieve a set of places with maximum selected distance to the query location and as before, all of the query keywords should be reachable from the places.

In this thesis, we focus on Ranked AND queries and discuss earlier works on related indexing schemes in this chapter. For computing the textual score of a place with respect to a keyword query, earlier studies typically employ an inverted index, which is reviewed in Section 2.1. For computing the spatial scores, various indexing schemes have been proposed in the literature, which broadly fall into categories of

R-Tree based indexes (e.g., [14] [5]) or grid-based indexes (e.g., [15] [1] [9]), which are reviewed in Sections 2.2 and 2.3, respectively.

2.1 Inverted Files

Inverted indexes [16] are widely accepted and used in the literature. An inverted index contains a posting list that contains the document id and the frequency information for each term in the vocabulary of the document collection (e.g., see Table 2.1).

In Ranked AND processing using an inverted index, for a query with, say, three keywords, we find the intersection of the corresponding three postings lists and score the documents that are in the intersection. Then, we rank these documents based on their textual scores.

To address the semantic place retrieval problem, a straightforward approach is using an inverted index and computing the spatial score for each place in the intersection, as will be discussed in Section 3.2. However, there are several other indexing schemes in the literature that make use of an inverted index to compute both spatial and textual scores, as we review in the rest of this chapter.

Table 2.1: Example postings lists in an inverted file for place retrieval

Term	**Lists**
renaissance	<P1,2> <P2,3> <P3,4> <P5,3>
davinci	<P1,3> <P2,2> <P5,2>
paintings	<P1,2> <P2,3> <P3,4> <P5,3>

2.2 R-Tree Based Indexes

2.2.1 R-Tree

R-Tree [17] is a kind of tree index structure that is used to store and access spatial information such as coordinates and other polygons. R-tree is a balanced search tree,

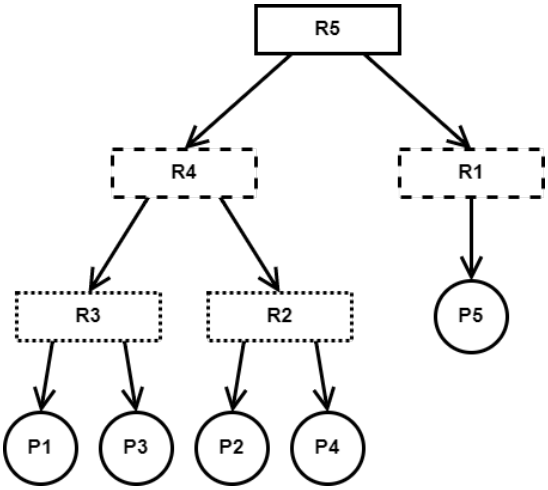

Figure 2.1: R-Tree example

which means all of the leaf nodes are either at the same level or they have one level difference at most, as shown in Figure 2.1.

The main concept behind R-tree is to group related objects and represent them in the next higher level of the tree with their minimum bounding rectangles (e.g., Figure 2.2). So, if a query term does not intersect with an MBR it means that it won't intersect with the contained objects.

2.2.2 S2I

One of the R-Tree-based works, S2I [14] employs two different indexing strategies for frequent and infrequent terms. Since, according to Zipf's Law [18], a large number of terms are used infrequently, their posting list is expected to be relatively small in terms of document size. So, the authors used inverted files for these types of terms. On the other hand, the posting list size of the frequent terms is larger. To search and reach the documents more efficiently, authors used an R-Tree variant, aggregated R-Tree. This indexing structure is similar to the IR-Tree [5] like every frequent term has its own IR-Tree structure. In the aggregated R-Tree, every leaf node contains the index entries. Non-leaf nodes contain non-spatial information (in our case it is a textual score), which is maximum in its child nodes.

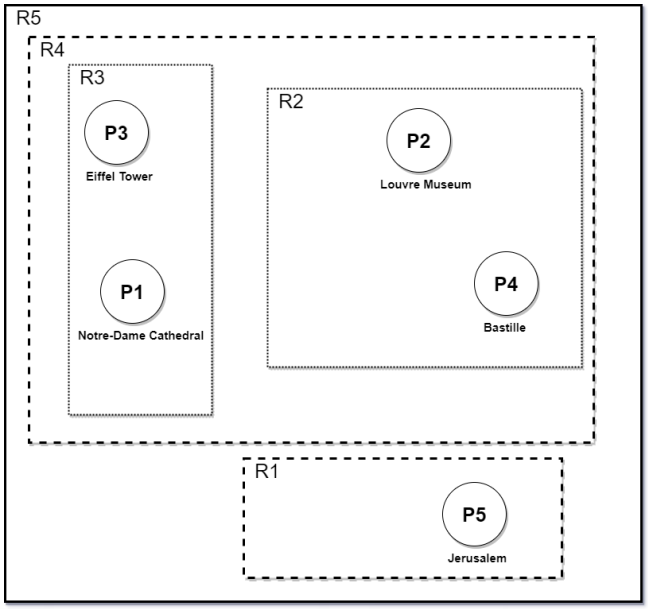

Figure 2.2: Example place MBRs to be used in an R-Tree

2.3 Grid and Cluster Based Indexes

Cluster-Based Retrieval systems aim to categorize documents in a collection based on, say, their topical similarity or spatial locations. In the first round of the two-stage retrieval process, the clusters should be examined for each query to find which ones are worth checking. The selected ones should be examined in the second round to construct the resulting set of documents (e.g., [12] [10] [9]).

Using clusters for query processing improves the efficiency and effectiveness of the document retrieval system according to the work in [19]. It is also emphasized that cluster-based retrieval in query processing improves the query processing time [10]. In this section, we focus on spatial location-based clustering systems, namely grid-based indexing structures. These so-called grid-based retrieval systems aim to group documents with their spatial information.

2.3.1 TS

In [1], the authors focus on the problem of searching geo-tagged documents and discuss several indexing schemes for the efficient processing of such queries. One par-

ticular indexing scheme they use is the so-called TS, which is similar to ours in that the documents in the posting lists are clustered spatially based on the grid cells to which they belong. As shown in Figure 2.3 (adopted from [1]), they assume that each document is associated with geographical regions that overlap with one or more grids. Then, an inverted index is created where posting lists group documents with respect to corresponding grids (e.g., see Table 2.2). However, our CS-IIS based approach differs from the latter work in several ways. First, their work only addresses the queries with explicit grid-based geographic constraints, so the index is used to identify grids that are exactly matching to the query. In contrast, in our approach, a keyword query has a given coordinate and we do employ a two-stage query processing where we first score grids with respect to spatial distance and textual similarity simultaneously, and then score the places in these grids, in the order of their ranking. Furthermore, we employ an early-stopping strategy that allows us to terminate query processing when a grid, which can not include any places to appear in top-k results, is encountered.

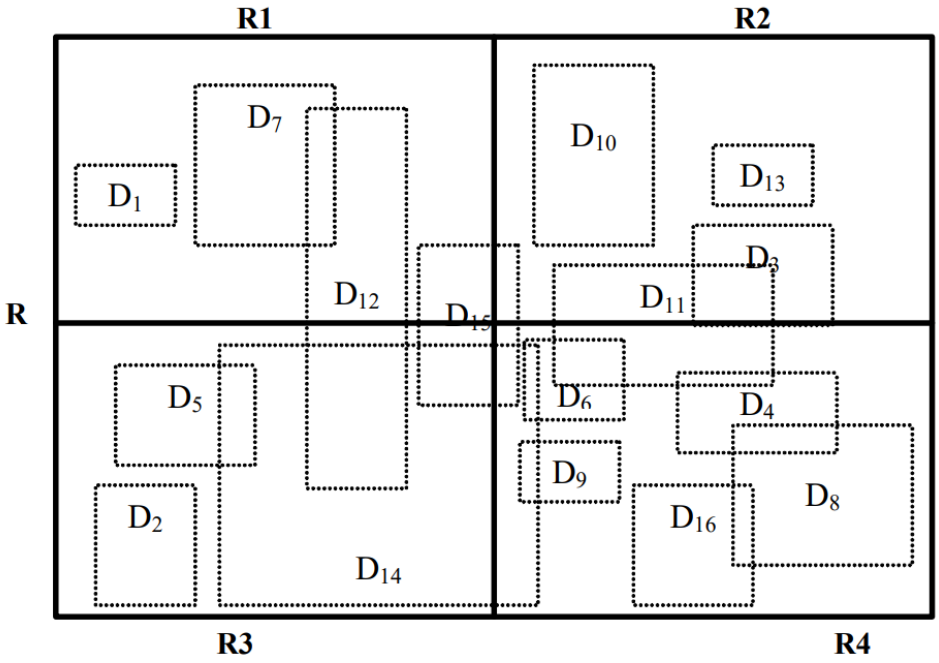

Figure 2.3: Example geo-tagged documents and overlapping grids for TS index (adopted from [1])

Table 2.2: Posting list example for TS (adopted from [1])

painting	$R1(D1, D12); R2(D10); R3(D5, D14); R4(D4)$

2.3.2 SKIF

Another work that exploits grids in the indexing structure [15] employs two different inverted index files for the grids and textual part, respectively. In Table 2.3 illustrating their index, the type column shows whether the index entry belongs to a grid or textual part. In the grid index entries, each grid id is mapped to a list of documents in this grid, while in textual entries, as usual, a term is mapped to a list of documents. During query processing, first, the grid cells that overlap with the query location are determined. Then posting lists corresponding to overlapping grid cells and query keywords are processed altogether to obtain document scores. Clearly, as in the previous case of TS [1], this indexing scheme also assumes that queries explicitly specify the grids to be searched. In contrast, the problem we address in this thesis requires computing the spatial distance between the query and documents (or places) for the final ranking.

Table 2.3: TS example (adopted from [1])

term *t*	f_t	**type**	**Spatial-Keyword Inverted List for *t***
soccer	5	1	<1, 1.00><2, 1.00><3, 1.00><4, 1.00><5, 1.00>
league	5	1	<1, 0.80><3, 1.00><4, 1.00><5, 1.00><6, 1.00>
c1	1	0	<1, 1.00>
c2	2	0	<1, 0.55><2, 1.00>
c4	1	0	<4, 0.25>
c5	3	0	<3, 1.00><4, 0.06><5, 1.00>

CHAPTER 3

METHODOLOGY

In this chapter, we first review the baseline approaches proposed for semantic place retrieval over a knowledge graph. In Section 3.1, we summarize the approaches proposed in [2] that require on-the-fly graph traversal and argue that the latter requirement can be relaxed if the search is limited to a fixed depth on the graph. Based on the latter assumption, we describe how some well-known index schemes from the literature namely, an inverted index and IR/DIR-Trees [5], can be employed in this scenario, respectively, in Sections 3.2 and 3.3. Finally, in Section 3.4, we propose adapting CS-IIS [9] [10] with an early-stopping strategy to handle this problem.

3.1 Graph Traversal Baseline Approaches

In this thesis, we focus on the semantic place retrieval over knowledge graphs, which are typically represented using Resource Description Framework (RDF) [20]. In a nutshell, RDF is a data model representing the data in three parts: the subject node, the object node, and a directed edge between these nodes. In case of a knowledge graph, nodes represent entities while edges represent relationships between them. Figure 3.1 shows an example knowledge graph including places, other related entities and relationships. In Table 3.1, we provide some triples illustrating how the latter knowledge graph would be expressed in RDF.

Table 3.1: Example RDF triples

<Renaissance>	<has>	<Oil Paintings>
<The Last Supper>	<owner>	<Leonardo Da Vinci>

Typically, RDF data is processed with complex query languages, such as SPARQL [21]. But these languages are hard to understand for end users. So, in a recent work [2], the authors focus on searching over RDF data with simple keywords rather than SPARQL. In their approach, RDF data is converted to a graph data structure, and the aim is to find a minimum sub-graph containing all the query keywords.

In [2], for the semantic place retrieval scenario, kSP queries are introduced to locate closest place entities to the query location so that RDF sub-graphs rooted at these place entities contain all the query keywords. Specifically, a kSP query has three parameters: query location, query keywords, and k, the number of elements of the result set. In this setup, a *qualified semantic place* is defined as the root of a tree covering all the query keywords, and a kSP returns top-k *tightest qualified semantic places* (TQSP). The places are scored using a function that computes the spatial score based on the distance between a place and query location, and the textual score based on the coverage of query keywords in the tree rooted at this place.

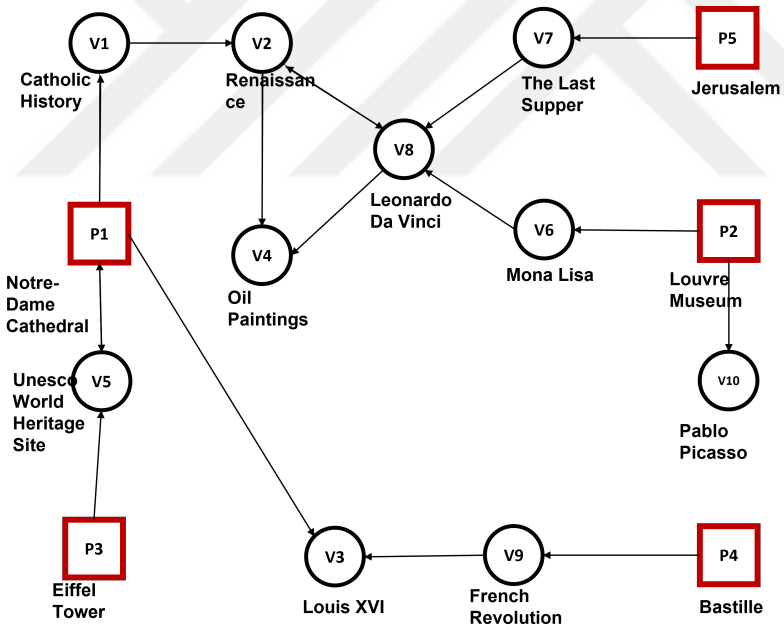

Figure 3.1: Example knowledge graph (nodes for place entities and other entity types are shown as squares and circles, respectively)

To compute the textual score, the previous work [2] introduces the notion of *looseness*. Looseness is based on the length of the shortest paths between the place entity

(i.e., root of the tree) and the nodes including query keywords. This notion is intended to capture how relevant a node (and hence, the keyword) is to the place. For example, the looseness score between *P2 Louvre Museum* and *V8 Leonardo Da Vinci* is 2. Note that, since the graph traversal proceeds in the Breadth First Search manner; even if the same node can be reached multiple times, only its first occurrence is considered. Thus, for the query *Renaissance Paintings*, the tree rooted at the place entity *P1* will have the looseness score (and hence, textual score) of 5, as the shortest path to keyword "Renaissance" is 2 (at node *V2*) while the keyword "paintings" is at a path length of 3 (at node *V4*).

3.1.1 BSP

To address the top-k semantic place retrieval problem, the first approach proposed in [2] is called the basic semantic place retrieval algorithm (BSP). The authors claim that the TSQP computation is far more expensive than spatial distance calculation; hence, the BSP doesn't follow a text-first search approach. Instead, it prioritizes spatial search using an R-Tree [17].

The BSP method (shown in Algorithm 1) works as follows: Place entities are extracted from the R-Tree [17], R, one by one, in the ascending order of the place's distance to the query location (line 7 in Algorithm 1). For every place entity p, the total looseness score, $L(T_p)$, of the tree T_p rooted at p is calculated by calling the $GetSemanticPlace$ procedure (line 12). If all of the query keywords $q.\psi$ are found in the nodes that are reachable from the place entity, the latter algorithm returns the $L(T_p)$ value (and otherwise, $L(T_p)$ is set to infinity). Then, the overall score f of p is computed as a linear combination of the spatial score, $S(q, p)$, and the textual (looseness) score, $L(T_p)$. The places are added to a min-heap H_k of size k, only when the computed score is lower than the k-th element's score θ (after the heap is populated with the first k places). Note that, the condition in lines 8–10 simply discards the places with a spatial score greater than θ without computing their textual score.

The GetSemanticPlace procedure (shown in Algorithm 2) works as follows: Starting from a place p, the graph G is traversed in the BFS manner, and for each visited node v, we check whether it contains any of the query keywords. Note that, during the

Algorithm 1 BSP Algorithm (adopted from [2])

1: MinHeap $H_k = \emptyset$, ordered by $f(L(T_p), S(q,p))$
2: **for** keyword $w_i \in q.\psi$ **do**
3: Load posting list of w_i from I
4: **end for**
5: Construct $M_{q.\psi}$
6: $\theta \leftarrow \infty$
7: **while** $e = GetNext(R, q)$ **do**
8: **if** $S(q,e) \geq \theta$ **then**
9: break
10: **end if**
11: **if** e refers to a place p **then**
12: $L(T_p) = GetSemanticPlace(q.\psi, p, G, M_{q.\psi})$
13: **if** $L(T_p) = \infty$ **then**
14: continue
15: **end if**
16: Compute score f using $L(T_p)$ and $S(q,p)$
17: **if** $f < \theta$ **then**
18: $H_k.add(p, f)$
19: Update θ
20: **end if**
21: **end if**
22: **end while**
23: return H_k

Algorithm 2 GetSemanticPlace (adopted from [2])

1: $T_p = \emptyset$
2: $L(T_p) = 1$
3: $B \leftarrow q.\psi$
4: **while** $v = BFS(G, p)$ and $B \neq \emptyset$ **do**
5: Add v to T_p
6: $v.\psi_q \leftarrow M_{q.\psi}.get(v)$
7: **if** $B \cap v.\psi_q \neq \emptyset$ **then**
8: $L(T_p)$ += $|B \cap v.\psi_q| \times d(p, v)$
9: $B \leftarrow B \backslash v.\psi_q$
10: **end if**
11: **end while**
12: **if** $B \neq \emptyset$ **then**
13: $L(Tp) = +\infty$ and $Tp = NULL$
14: **end if**
15: return $L(T_p)$ and T_p

latter check, for efficiency purposes, the authors of [2] first read postings lists of each query term in the memory and then converts them into a map $M_{q.\psi}$, which basically maps a node v to the contained query terms, $v.\psi_q$. If a node v includes one (or, more) query keyword, path length $d(p, v)$ is added to $L(T_p)$, and the found keywords are removed from the current list of searched keywords, denoted as B (lines 8 and 9). At the end of BFS, if the current query keywords set B is not empty, it means that some of the query keywords are not reachable from the place p. Then, since our querying logic is conjunctive (i.e., Ranked AND), we return the looseness as ∞.

3.1.2 SPP

SPP [2] is an improved version of the aforementioned BSP algorithm. Specifically, while computing the textual score, a pruning strategy, called dynamic bound based pruning, is applied. Thus, SPP approach employs an optimized version of GetSemanticPlace procedure, shown in Algorithm 3.

As described in the previous section, while traversing the knowledge graph, the looseness $L(T_p)$ is updated each time a new query keyword is encountered. Since the BFS starts searching from the closest nodes, it is guaranteed that there is no better looseness for the already found keywords in the remaining (unvisited) nodes in the graph. The looseness threshold can be calculated as $L_w(T_p) = \theta - S(q, p)$, where θ is the worst score of the MinHeap H_k. So, if the current looseness $L_B(T_p)$ is larger than the threshold $L_w(T_p)$ we can say that no better score will be calculated for the corresponding place p. So, when we encounter such a place, we apply dynamic bound-based pruning and stop traversing the rest of the graph for that place.

Algorithm 3 GetSemanticPlace for SPP (adopted from [2])

$T_p = \emptyset$
$L_B(T_p) = 1$
$B \leftarrow q.\psi$
Compute the looseness threshold $L_w(T_p)$
while $v = BFS(G, p)$ and $B \neq \emptyset$ **do**
    Add v to T_p
    Compute the dynamic bound $L_B(T_p)$
    if $L_B(T_p) \geq L_w(T_p)$ **then** ▷ Pruning Rule
        return ∞ and $T_p \leftarrow null$
    end if
    $v.\psi_q \leftarrow M_{q.\psi}.get(v)$
    if $B \cap v.\psi \neq \emptyset$ **then**
        $L(Tp)$ += $|B \cap v.\psi| \times d(p, v)$
        $B \leftarrow B \backslash v.\psi$
    end if
end while
if $B \neq \emptyset$ **then**
    $L(Tp) = +\infty$ and $Tp = NULL$
end if
return $L_B(Tp)$ and Tp

3.1.3 Limiting Traversal Depth

In our preliminary experiments, we observed that the aforementioned approach BSP and its variant SPP suffer from the extensive cost of on-the-fly BFS traversal of the knowledge graph. Indeed, due to a similar observation, the authors of [2] have also proposed another variant (so-called SP algorithm) involving an offline pre-processing stage that determines the terms appearing in the nodes within an α-Radius neighborhood of each place, to prune the graph traversal. While this optimization may allow early elimination of some places, for others, an on-the-fly graph traversal, which is the most expensive component of the algorithm, is still required.

In this thesis, we take a further step to eliminate the need for online graph traversal completely. Our approach is based on the intuition that a place's relevance to a keyword would considerably drop if this keyword is associated with a node that is several vertices far away from this place (i.e., implying a high looseness). Thus, rather than conducting an online BFS, we suggest setting a heuristic depth limit, DL, for BFS and traversing the graph offline to determine all the terms that are reachable from each place within a search depth of DL. In other words, we represent each place with a vector of terms that are reachable in length-DL paths (see the example in the next section). This assumption allows us to directly employ or adapt various indexing schemes for the semantic place retrieval problem, as discussed in the following sections.

An astute reader would question whether imposing such a depth limit considerably affects the result quality, or not. In a preliminary experiment, we observed that top-10 query result lists with and without a depth limit for BFS are sufficiently similar, yielding a Jaccard similarity score of 72% (see Chapter 4 for the details of the experimental setup) when DL is set to 4. Further note that, here we set the depth limit value as 4 in an ad hoc manner, and higher values would yield even higher overlap with the results obtained by BFS without any limit. Therefore, in the rest of this thesis, we essentially focus on how setting such a limit makes it possible to adapt various indexing schemes to our problem and how their efficiency compare to each other.

3.2 Full Index

Pre-processing the knowledge graph as suggested in Section 3.1.3 would allow representing each place with a term vector including the keywords on the nodes within a path distance of DL. That is, each place node *p* is represented with a vector of <*t,l*> pairs where the *t* is the term, and the *l* is the looseness of the closest node that includes the term *t*, where $l \leq DL$.

For instance, assuming depth limit DL is set to 2, the term vector for the place *P2 (Louvre Museum)* in Figure 3.1 is as follows:

$$P2 \rightarrow <mona, 1><lisa, 1><pablo, 1><picasso, 1><leonardo, 2><davinci, 2>$$

Given such term vectors per place, a straightforward strategy is constructing a typical inverted index (referred to as Full Index here) that maps each term *t* to a list of <*p,l*> pairs. For instance, assuming depth limit DL is set to 4, the posting list for the term "renaissance" is as follows:

$$renaissance \rightarrow <P1, 2><P2, 3><P3, 4><P5, 3>$$

Note that, in this index, each posting may also include the coordinates of the place to ease the spatial distance computation.

This index allows us to apply document-at-a-time (DaaT) query processing strategy [22]. To generate the Ranked AND result set, we find the intersection of each query keyword. For each intersecting place, we calculate a score using the textual score and the spatial score as in Equation 3.1.

$$Score = \alpha \times L(T_p) + (1 - \alpha) \times S(q, p) \tag{3.1}$$

As before, $L(T_p)$ denotes the textual score, i.e., the looseness of the tree rooted at the place p, and $S(q, p)$ denotes the spatial distance between the place and query.

DAAT query processing over the Full Index proceeds as follows (Algorithm 4): Postings lists for each query keyword $w \in q.\psi$ are read. Every pointer shows the initial position of the corresponding query keyword's (w_i) list. The place ids that each pointer shows are checked. If all of them point to the same place id p, the looseness l

Algorithm 4 DAAT Query Processing using Full Index

```
1: keywords ← q.ψ
2: pointers ← 0
3: MinHeap H_k = ∅, ordered by f(L(T_p), S(q, p))
4: θ ← 0
5: for keyword w_i in q.ψ do
6:     w_i.PL ← readPostingsList(w_i)
7: end for
8: while no posting list is finished do
9:     if all pointers point to the same place id p then
10:        for keyword w_i in q.ψ do
11:            L(T_p) += w_i.l
12:        end for
13:        Compute score f using L(T_p) and S(q, p)
14:        if H_k.size < k then
15:            H_k.add(p)
16:            θ ← H_k.peek()
17:        else if H_k.size() == k AND θ >= f then
18:            H_k.add(p)
19:            H_k.pop()
20:            θ ← H_k.peek()
21:        end if
22:        for pointer in pointers do pointer += 1 end for
23:    else
24:        for pointer in pointers do pointer ← max(pointers) end for
25:    end if
26: end while
27: return H_k
```

for each keyword is added to the total looseness score $L(T_p)$. Then the total score f is calculated using looseness score $L(T_p)$ and the spatial score $S(q, p)$. If the number of places in the MinHeap, H_k, is less than the target result set size, k, the place node p is added to the H_k, and the θ is updated with the worst score in H_k. Otherwise, if

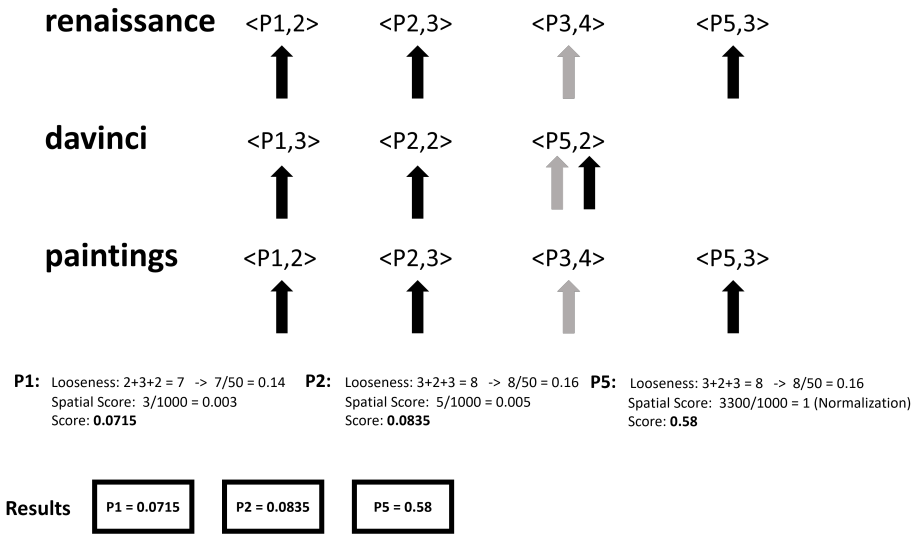

Figure 3.2: DAAT processing over full index for a toy scenario

the number of places in the MinHeap is equal to k, then the place p is added to the H_k only if its score f is less than θ, and in this case the node with the worst score is popped from the H_k and θ is again updated. The algorithm continues until one of the posting lists are consumed.

For example, in Figure 3.2 a query is processed with 3 query terms such as *renaissance, da vinci, and paintings*. As the first column of dark arrows shows, all the pointers point to the initial position and point to the same place id, $P1$. Since we apply document-at-a-time processing, a score is calculated for the place $P1$. This score is the combination of the spatial and the textual score. First, the spatial score is computed as the distance between the query point and the P1, which is, say, 3 km (for the sake of simplicity, the coordinates are not shown explicitly). The spatial score is normalized with 1000 km, following the practice in [2]. The textual score is calculated as the sum of the looseness values in the postings of these three terms, i.e., 2, 3, and 2, respectively. Total looseness 7 is normalized with a maximum looseness value, which is set to 50, again as in [2]. The final score for $P1$ is calculated as $(0.14 + 0.003)/2 = 0.0715$. The same calculations are done with the other places in the intersection, namely, P2 and P5, as shown with the darker arrows. The final result is shown in Figure 3.2.

Algorithm 5 Application of AND logic

```
results ← ∅
for keyword in query keywords do
    for place in keyword.postingList do result.get(place) + = 1
    end for
end for
return results(f → f.value == len(queryKeywords))        ▷ Checks if all the keywords are found for a place
```

3.3 IR-Tree

The approach using a Full Index presented in the previous section prioritizes text search, as it first intersects postings lists and then computes the spatial score for each place in the intersection. In the literature, there are alternative hybrid indexing schemes that guide the search by prioritizing the spatial and textual scores simultaneously, such as the IR-Tree and its variant DIR-Tree [5]. Earlier works have shown that both indexing schemes are feasible approaches for spatial keyword search (e.g., see [8]). Of course, these approaches have their own disadvantages, too. For instance, IR-Tree is shown to be more efficient when the number of query keywords is small [23]. Furthermore, the inverted files can be easily distributed across several machines, while this may not be easily possible for the R-trees [24].

In this thesis, we adapt IR- and DIR-tree geo-indexing schemes for the problem of semantic place retrieval. To this end, we modified a public code repository[1] for an earlier work [8] in the following ways:

- **Query evaluation logic:** In this thesis, following the application scenario in [2], we employ Ranked AND query evaluation logic, while the codebase had the implemention for the Ranked OR logic. Therefore, we modified the codebase to enforce the existence of all query keywords in every scored (and visited) node of the IR-Tree.

  Algorithm 5 shows the application of the Ranked AND logic during the IR-Tree traversal. For the sake of simplicity, other scoring details are not included.

[1] http://web.archive.org/web/20220227042457/http://lisi.io/spatial-keyword%20code.zip

- **Scoring functions:** We updated the scoring function of the codebase as in Equation 3.1. We also reflected the aforementioned normalization of the scores, which normalizes the spatial distance with 1000 km and total looseness with 50 as in the previous work [2].
- **Index content:** The original codebase creates postings lists that include term frequencies in the postings, while in our case, the postings should include the looseness values (as described in the previous section). Note that using looseness values required us to modify the score comparisons at various places in the code, as lower looseness and lower overall scores indicate better query results in our case.

3.3.1 How does an IR-Tree Work?

In this section, we illustrate how an IR-Tree would operate in our setup. As described in [5], an IR-tree is based on an R-tree but extends the latter so that every tree node has its own inverted index. Every IR-Tree node is represented with $< w.pl, n.p >$ where $w.pl$ shows the posting list of the query terms and the $n.p$ shows the node's location. Posting list elements are in the form of $< p.id, l >$ where $p.id$ is the place id, and the l is the looseness. For the leaf nodes, the posting lists store the actual looseness values of the places. The non-leaf nodes store the minimum looseness value for each child (sub-tree) and each query term, allowing to compute the best-possible (upper bound) score for its children.

Table 3.2: Typical postings lists for a toy scenario

Word/Looseness	**P1**	**P2**	**P3**	**P4**	**P5**
renaissance	2	3	4	-	3
painting	2	3	-	-	3
davinci	3	2	-	-	2

Let's consider a toy scenario with a query including three keywords, *renaissance, painting, daVinci*. In Table 3.2, we show the postings lists in a typical inverted index (as described in Section 3.2, for each term, we store the place and looseness (assuming

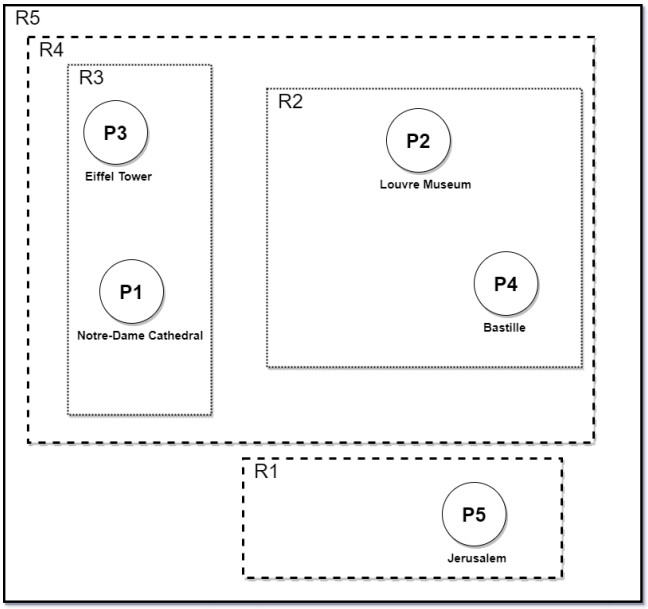

Figure 3.3: Places and MBRs to be used in our toy R-Tree

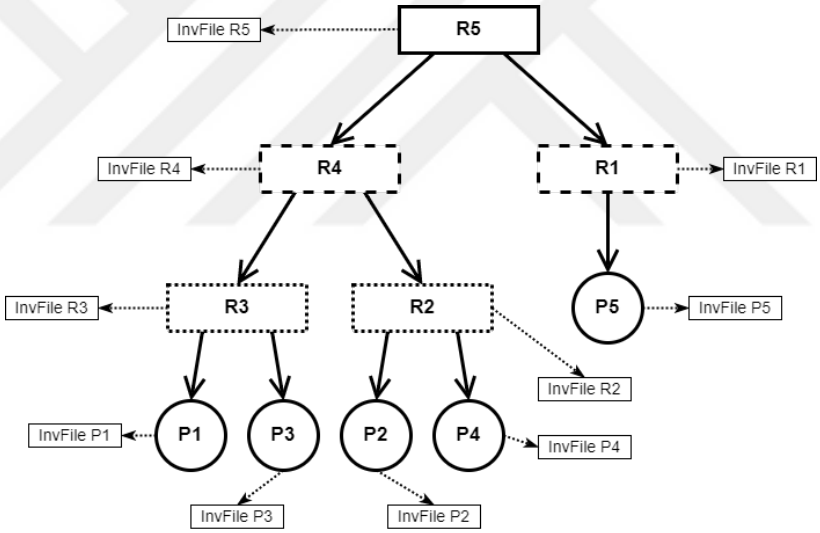

Figure 3.4: IR-Tree structure showing the inverted files for each node

a depth limit of 4) for the knowledge graph in Figure 3.1). Further assume that the places in this scenario are associated with the R-tree MBRs shown in Figure 3.3. Then the resulting IR-tree is shown in Figure 3.4.

As depicted in Figure 3.4, every node in the IR-tree stores an inverted file. The postings lists of these indexes at each node (R1 to R5) are shown in Table 3.3. For the leaf nodes R1, R2, and R3, the postings lists store the actual (place, looseness) pairs.

For the non-leaf nodes, the minimum (best) looseness is stored for each sub-tree. For instance, R4 points to nodes R2 and R3, so for the term *renaissance*, R4 has postings for R2 and R3. Furthermore, the minimum looseness for the term *renaissance* in the R3 is 2 (corresponding to place $P2$), and hence, the R4 has the posting with the looseness value 2 for R3.

Table 3.3: Postings lists for the IR-Tree nodes (R1 to R5)

Term/IR-TreeNode	**R1**	**R2**	**R3**	**R4**	**R5**
renaissance	<P5,3>	<P2,3>	<P1,2><P3,4>	<R2,3><R3,2>	<R1,3><R4,2>
davinci	<P5,2>	<P2,2>	<P1,3>	<R2,2><R3,3>	<R1,2><R4,2>
painting	<P5,3>	<P2,3>	<P1,2>	<R2,3><R3,2>	<R1,3><R4,2>

The query processing using an IR-tree (as shown in Algorithm 6) works as follows: the root of the tree is loaded into the memory. The root is added into a MinHeap H_k which is sorted by the score $f(D(q, p_L), L(q, p_L))$. The algorithm continues until the H_k becomes $\emptyset$ or k places are extracted.

The first element e of the H_k is popped. If the e is a leaf entry, then the distance $D(q, p_L)$ between the place of this leaf entry and the query point is calculated. Then a score $f(D(q, p_L), L(q, p_L))$ is calculated by using distance $D(q, p_L)$ and the looseness $L(q, p_L)$. Given this score, the place is added to the result queue r_K. If the result queue r_K reaches the size of k, the result queue r_K is returned, and the algorithm is finished.

But if the e is a non-leaf node, the posting lists p_L of the query keywords are fetched to memory. The total looseness $L(q, p_L)$ of these posting lists for each child of the node is calculated. Then for each child, a pseudo score $S'(D(q, p_L), L(q, p_L))$ by node distance $D'(q, p_L)$ to the query point and looseness $L(q, p_L))$ of the child is calculated. The children are added to the MinHeap H_k.

Algorithm 6 Query Processing using IR-Tree (adopted from [5])

```
1: MinHeap H_k = ∅, ordered by f(L(q,p), D(q,p))
2: r_K ← ∅
3: Put root of the tree to H_k
4: while H_k.size() != 0 do
5:     e ← pop(H_k)
6:     if e.isLeafEntry() then
7:         if k > r_K.size() then
8:             r_K.add(< e.id, f(D(q,p_L), L(q,p_L)) >)
9:             if r_K.size() == k then
10:                break
11:            end if
12:        end if
13:    else
14:        node ←readNode(e.id)
15:        p_L ←readPostingListsForEachKeyword()
16:        if p_L = ∅ then
17:            continue
18:        end if
19:        for child in node.children do
20:            if L(q,p_L) = null then
21:                continue
22:            end if
23:            child.score ← S'(D(q,p_L), L(q,p_L))
24:            H_k.add(child)
25:        end for
26:    end if
27: end while
```

3.4 DIR Tree

DIR-tree [5] is a variant of the IR-tree indexing scheme. IR-tree only takes into account spatial information of the documents while computing minimum bounding

rectangles (MBRs). In contrast, DIR-Tree also considers document similarity in addition to spatial information. In other words, DIR-tree creates MBRs of the IR-Tree by optimizing the objectives of minimizing the MBR area and maximizing text similarities. In our case, this corresponds to minimizing looseness, which means increasing the text relevancy.

3.5 CS-IIS

Cluster-Skipping Inverted Index (CS-IIS) is a cluster-based indexing scheme [9] [10] with a two-level architecture, which we adapt here for the place retrieval scenario. In our adaptation, the first level maps each term to the grids including places with that term. Furthermore, we store the minimum looseness value of the places in the grid for the given term. Thus, postings in this level are in the form of <*Grid Id, Looseness*>. We emphasize that a grid G's postings appear in a term's postings list if and only if a place inside the grid G contains this term. The second level of the index typically stores the actual places and the looseness value of the place for each term.

As before, assume a toy scenario with places and respective grids shown in Figure 3.5. Then, example postings lists for grids and places are shown in Tables 3.4 and 3.5, respectively (recall that place and hence associated grid descriptions include all the terms on the nodes that are within a length-DL path, where DL is 4.). Also note that, we show the postings lists separately in the latter tables only for the sake of clarity; in the actual CS-IIS organization, the postings for the grids and places are intermixed, so that places are associated with the corresponding grids can be directly accessed, as illustrated in Figure 3.6.

Even if the additional grid posting lists increase the index size slightly, in our evaluations we show that it dramatically improves the performance due to the capability of pruning many grids without processing the places inside them. Also, if memory is limited and index files need to be kept on disk, the CS-IIS allows the queries to be processed by only one direct disk access per query term [9], as will be shown in the experimental evaluation section.

Note that, we also tailor an early-stopping approach that terminates the query pro-

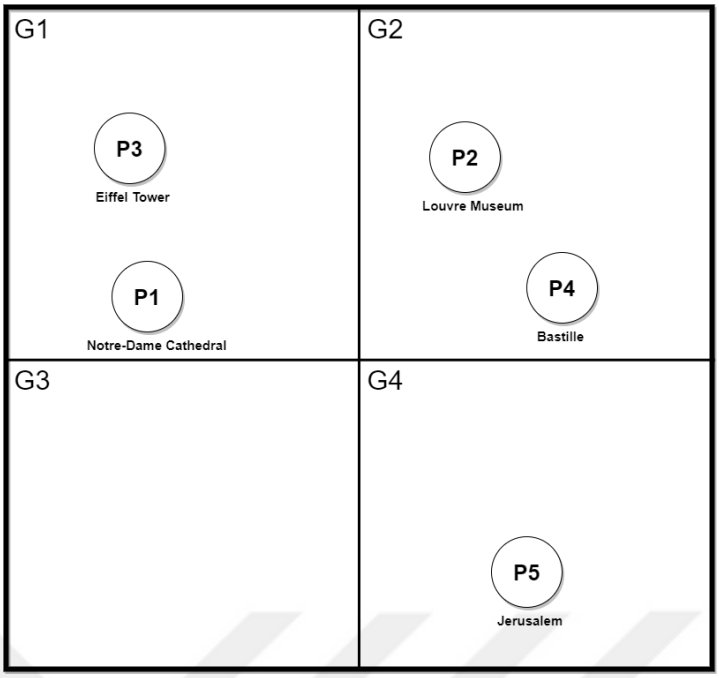

Figure 3.5: Places and respective grids for CS-IIS

Table 3.4: CS-IIS: Example postings for the grid-level

Place	**Keywords**
renaissance	<G1,2> <G2,3> <G4,3>
davinci	<G1,3> <G2,2> <G4,2>
paintings	<G1,2> <G2,3> <G4,3>

Table 3.5: CS-IIS: Example postings for the place-level

Place	**Keywords**
renaissance	<P1,2> <P2,3> <P3,4> <P5,3>
davinci	<P1,3> <P2,2> <P5,2>
paintings	<P1,2> <P2,3> <P3,4> <P5,3>

cessing when the upperbound score of a grid is worse than the worst score of the current top-k places, which is described in detail later. While our approach is similar to that in [12], their work is focused on Ranked OR processing of queries over typical document collections and they aim to obtain the best results within a given time budget. In contrary, our scoring function involves both textual and spatial components

and we focus on a Ranked AND query processing logic, for which the performance of early-stopping on CS-IIS has not been explored before.

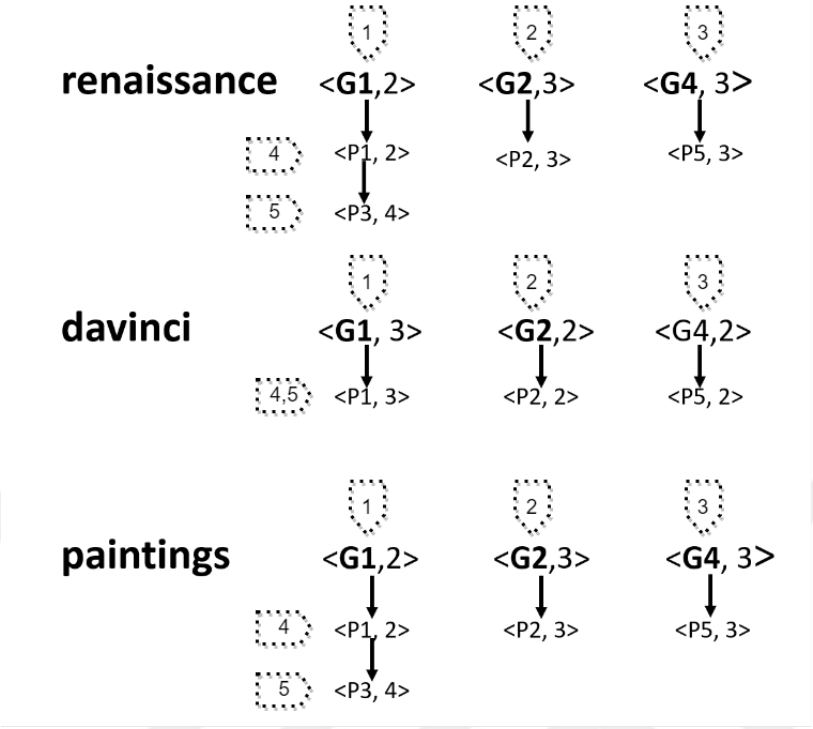

Figure 3.6: Query processing illustrated for CS-IIS

Our query processing algorithm with early-stopping using the CS-IIS operates as follows (Algorithm 7): First, a result queue of places, r_K, and a heap of grids, H_k, are created and set to $\emptyset$. θ stores the worst score of the r_K. Initially, all the postings lists for the query keywords $q.\psi$ are fetched into the memory.

In the first round of the algorithm, grids in the intersection are determined and scored. The process begins with the first grid of each query keyword in $q.\psi$. If all the pointers ptr point to the same grid id G (see step 1 in Figure 3.6, dotted numbers show the step numbers), the grid is added to the grid heap H_k. Then every pointer ptr is moved to the right by one (see step 2 in Figure 3.6). If the pointers point to different grid ids, then the grid with the maximum id number is found with binary search and every pointer ptr is moved to that position. In our example, the ids are again the same, i.e., G2, which is also scored. This process continues until one of the pointers reaches the end of the grid list. After the creation of the H_k, the first round finishes and the second round of the algorithm begins.

During the second round, for each grid in the H_k, an early-stopping opportunity is checked. If the worst score θ of the MinHeap (r_K) is smaller than the grid upper bound score f' then the algorithm is stopped and the result queue r_K is returned. If the θ is not smaller than the grid's upper bound score, a similar scoring process with Ranked AND logic is applied, but this time, over the places in the current grid that is being processed, as illustrated is steps 4 and 5 in Figure 3.6. As before, this process continues until one of the pointers reaches the end of the place postings list.

Algorithm 7 Query Processing with Early-Stopping using CS-IIS

MinHeap $H_k = \emptyset$, ordered by $f(L(G_E), S(q, G_E))$
$\theta \leftarrow \infty$
MinHeap $r_K \leftarrow \emptyset$, ordered by $f(L(T_p), S(q, p))$
for $q.\psi$ **do** $readIndexEntry()$ **end for**
while no grid-level posting list is finished **do**
    if Each ptr points to the same $G_E.id$ **then**
        $L(G_E) \leftarrow 0$
        for each G_E **do** $L(G_E)$ += $G_E.l$ **end for**
        Compute grid score f' using $L(G_E)$ and $S(q, G_E)$
        $H_K.add(G_E)$
        for ptr in pointers **do** p += 1 **end for**
    else
        for ptr in pointers **do** $p \leftarrow max(pointers)$ **end for**
    end if
end while
while $H_k \neq \emptyset$ **do**
    $G_E \leftarrow H_K.pop()$
    if $f' \geq \theta$ **return** r_K ▷ Early-stopping
    $p \leftarrow 0$, $L_r \leftarrow 0$
    if Each ptr points to the same $p.id$ **then**
        for p in places **do** $L(T_p)$ += $p.l$ **end for**
        Compute score f using $L(T_p)$ and $S(q, p)$
        if $r_K.size() < k$ **then**
            $r_K.add(p)$
        else if $r_K.size() \geq k$ **and** $\theta \geq f$ **then**
            $r_K.add(p)$
            $r_K.pop()$
        end if
        $\theta \leftarrow r_K.peek()$
    else
        $p \leftarrow max(p.id)$
    end if
end while

CHAPTER 4

EXPERIMENTAL SETUP

4.1 Dataset

We created a dataset based on YAGO 3 (Yet Another Great Ontology) RDF knowledge base [25] for our semantic place retrieval scenario. This knowledge base is built upon various resources, such as Wikipedia, Wordnet, etc. The YAGO version employed here has more than 17 million triplets, and 357,603 place nodes [26].

As discussed before, RDF models the data as triples: A subject node, an object node, and a directed edge to show the relationship between the subject and the object. An object can be another node or a literal value, such as the birth date of a person. Following the procedure in [2], we created our dataset as follows. We preprocessed the dataset by removing the punctuation so that *National Library* and *National_Library* or *Polatlı_(Ankara)* and *Polatlı Ankara* become identical. Since we only consider looseness as the relevancy metric we removed the edges that show the relationship between nodes. Also, we have marked the nodes with the geo-location information as places.

As discussed in Chapter 3, for index-based approaches, we first traversed the graph until the depth of 4 starting from each place node to determine the textual representation of each place and then created the required inverted index files in which postings list for each term including pairs of <*placeId, looseness*> (and some other information, based on the index type).

4.2 Query Sets

Since using randomly generated keywords and locations reduce the finding of any meaningful results for a Ranked AND query, we followed the practice in an earlier work [2]. To generate a query, we chose a random place p from the place nodes in our dataset. Then, from the term vector of p (as described in Section 3.2) we picked n query keywords to obtain $q.\psi$, where $1 \leq n \leq 8$. In the end, we ended up with a query set including 1112 queries (see Table 4.1 for the details of our query set).

Table 4.1: Number of queries for each query length value

Number of query keywords	**Number of queries**
1	207
2	133
3	143
4	142
5	133
6	145
7	112
8	97

4.3 Meta Information For the Index Files

We created six different index files corresponding to the methods described in Chapter 3. We provide their size (on disk) in Table 4.2.

4.4 Parameters and Platform

We have experimented with a range of parameters to expand our experiments, as summarized in Table 4.3. For the query result set, we have defined k as either 10 or 20.

Table 4.2: Index sizes

Index	Size (GB)
CS-IIS 180	30
CS-IIS 36	28
CS-IIS 300	32
DIR-Tree	25
IR-Tree	26
Full Index	28

The number of query keywords $q.\psi$ per query is between 1 and 8.

To set the balance between the score of spatial distance and looseness, we used α. We set α to 0.5, meaning that spatial distance and looseness are equally important.

Another important parameter is the number of grids that are used while creating the CS-IIS index (as described in Section 3.5). In our experiments, we experiment with three different values, namely, 36, 180, and 360, implying that we have 36×36, 180×180 and 300×300 grids, respectively.

All our experiments are conducted on a computer with two Intel Intel Xeon E5-2630 CPUs, 256 GB of RAM, and 4 TB HDD, running Ubuntu Linux v14.04 and Java 13.

Table 4.3: Parameters

Parameter	Values
Result set size (k)	$\{10, 20\}$
$\|q.\psi\|$	$\{1, 2, 3, 4, 5, 6, 7, 8\}$
α	0.5
Disk page size	64000
IR-tree fanout	1024
Number of grids (for CS-IIS)	$\{36 \times 36, 180 \times 180, 300 \times 300\}$

4.5 Evaluation Metrics

We employ and report the following evaluation metrics:

- **In Memory Processing Time:** This captures the entire processing time (elapsed time) in the memory. It starts from the beginning of the query processing and then lasts until the top-k query result is computed, but excludes the disk access time .
- **Total Query Processing Time:** Total time means the duration between the beginning and end of the query processing, which includes all the time spent in memory and disk access.
- **Simulated Total Query Processing Time:** Simulated time is calculated using the in memory time, accessed page count, direct IO count, disk seek time and disk transfer time, which can be seen in the Equation 4.2. The disk parameters employed in Equation 4.1 are shown in Table 4.4.

$$ReadTime = 1/(TransferSpeed(Bytes/Sec)/PageSize) \tag{4.1}$$

$$SimulatedTime = MemoryTime + (SeekTime * IOCount) + (ReadTime * PageCount) \tag{4.2}$$

Table 4.4: Disk parameters for simulated total query processing time

Parameter	Values
Transfer Speed(Mb/Sec)	190
Page Size	64000
Seek Time(ms)	10

- **Direct Disk Access Count:** IO Count which indicates the number of direct disk access during the query processing.
- **Document Posting Count & Grid Posting Count:** For IR-Tree, DIR-Tree, and Full index, Document Posting Count is the number of processed document

postings for each query. For CS-IIS, Grid Posting Count indicates the number of processed grid postings for each query.

While measuring the time-related metrics, we re-started the server after experimenting with each different method, to avoid any caching effects between experiments.

CHAPTER 5

EXPERIMENTAL RESULTS

5.1 Time Efficiency Comparison

5.1.1 In Memory Processing Time

This metric captures the elapsed time for all in memory calculations. So, memory time excludes the time spent reading from disk, loading index to memory, etc. In particular, memory time measurement starts after reading index entries to the memory for the Full and CS-IIS indexes. These index structures access all the posting lists for the query terms at the beginning of the query processing. For IR-Tree and DIR-Tree indexes, memory time measurement starts after loading the root node to the memory and continues until the end of the query processing but excludes the time for fetching the other tree nodes from the disk.

Table 5.1 presents the in memory processing times for the baseline and proposed indexing schemes for $k = 10$. We report the findings for each query length and their average in detail. Our findings reveal that among the baseline indexing schemes, both geo-indexing approaches (IR-Tree and DIR-Tree) outperform the typical Full Index, and the IR-tree is the best-performing baseline for all query lengths. Among the CS-IIS variants, we see that CS-IIS 180, which employs 180×180 grids is better than the other two variants involving 300×300 and 36×36 grids, implying that having a larger or smaller number of grids would deteriorate the performance for this dataset.

When we compare IR-Tree and CS-IIS 180, we see that the latter provides a speed-up of 1.10x over the former indexing scheme. In particular, Table 5.1 shows that the average processing time using IR-Tree is 36.99 ms, while it takes 33.71 ms using

Table 5.1: In memory processing time (ms) for top-10 results (speed-up is reported wrt. IR-Tree baseline for the average case)

No of Query Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	134.42	22.50	23.52	13.98	13.74	15.91
2	286.80	33.23	35.32	20.10	27.44	27.20
3	201.59	33.65	38.31	24.56	31.54	32.26
4	382.11	36.68	40.54	34.94	38.99	40.35
5	224.49	43.25	46.18	40.23	42.34	49.08
6	246.85	43.26	44.81	45.00	52.26	50.91
7	278.83	45.81	48.32	50.86	50.71	57.62
8	451.60	50.26	52.94	60.57	75.35	68.64
Average	**260.56**	**36.99**	**39.56**	**33.71**	**38.43**	**39.82**
Speed-up	**0.14**	**1.00**	**0.94**	**1.10**	**0.96**	**0.93**

CS-IIS 180, indicating a relative efficiency improvement of around 9%. Furthermore, when we focus on shorter queries with 1, 2 or 3 keywords, the relative gains are more emphasized, reaching up to 38%, 40%, and 27%, respectively. Given that users typically tend to issue shorter queries (e.g., average query length is around 2-3 terms for web search engines [27], [28])), the gains provided by the CS-IIS based approach seem remarkable.

At this point, an astute reader may question the appropriateness of the parameters for the IR-Tree. In our dataset, we have 357,603 places, which are indexed using an IR-Tree with a fanout of 1024 and a page size of 64000 bytes. These parameters imply that the resulting IR-Tree has only one level, i.e., it can quickly access the leaf level without the requirement of traversing some interior nodes and fetching them from the disk. Thus, we think that our efficiency comparison of IR-tree and CS-IIS presented in this section is fair.

Finally, in Table 5.2, we report the performance of SPP [2], the baseline approach that employs on-the-fly graph traversal, as described in Section 3.1.2. Our preliminary experiments showed that the SPP algorithm, without any depth limit for BFS, runs for several minutes for certain queries and is inferior to all the aforementioned approaches. Hence, in Table 5.2, we report the run time figures for SPP when the

Table 5.2: In memory processing time (ms) for top-10 results using SPP method [2]

No of Keywords	1	2	3	4	5	6	7	8	**Average**
In memory time(ms)	55,377.85	44,071.32	47,783.69	47,021.73	43,468.37	34,701.26	49,291.54	24,689.04	**44,571.33**

Table 5.3: In memory processing time (ms) for top-20 results (speed-up is reported wrt. IR-Tree baseline for the average case)

Number of Keywords	**Baseline**			**Proposed**		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	135.17	23.01	24.45	14.36	14.70	16.71
2	284.42	32.78	33.88	20.36	28.13	27.21
3	195.53	37.38	41.42	26.63	33.34	34.42
4	384.26	38.58	42.86	35.06	37.88	41.19
5	225.98	42.37	46.11	40.89	43.95	49.73
6	258.80	43.53	46.05	45.28	50.92	50.66
7	286.12	45.16	48.58	49.34	50.04	60.35
8	456.10	49.40	51.93	61.97	74.01	72.10
Average	**262.77**	**37.54**	**40.34**	**34.18**	**38.62**	**40.97**
Speed-up	**0.14**	**1.00**	**0.93**	**1.10**	**0.97**	**0.92**

BFS depth limit DL is set to 4. Note that, with such a limitation, the performance of SPP is expected to be closer to that of SP, the more advanced version of SPP proposed in [2]. Furthermore, we think that the SP algorithm of [2] is similar[1] to using an IR-Tree when the $\alpha-$radius parameter for the former method is also set to 4, i.e., the depth limit DL used while constructing the IR-tree. Because of these reasons, we only consider SPP as a baseline here, but not SP. Our findings in Table 5.2 justifies our motivation in this thesis: even with a depth limit for BFS, the SPP method is several order of magnitudes slower than the index-based approaches. Hence, in the rest of this chapter, we only focus on the performance of the index-based approaches.

In Table 5.3, we present the efficiency figures when the result set size, k, is set as 20. The trends are similar to those in Table 5.1, as the CS-IIS (with 180×180 grids) is again the best-performing approach.

[1] Indeed, SP algorithm employs an IR-Tree like structure for pruning, but still attempts to traverse the graph on-the-fly without any depth limit, and hence, it is likely to be inferior to IR-Tree in terms of run-time efficiency.

Table 5.4: Total processing time (ms) for top-10 results (speed-up is reported wrt. IR-Tree baseline for the average case)

Number of Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	319.99	1,069.08	1,150.57	213.54	195.74	218.07
2	648.36	1,312.68	1,391.29	408.76	372.52	411.81
3	674.00	1,278.27	1,478.15	534.89	484.76	539.76
4	950.03	1,359.04	1,439.51	649.27	567.31	646.65
5	924.89	1,461.06	1,607.87	803.99	697.05	795.93
6	1,007.08	1,317.12	1,461.44	875.49	769.32	864.02
7	1,159.47	1,436.78	1,482.36	1,025.84	874.98	993.54
8	1,427.99	1,437.78	1,613.75	1,154.02	990.32	1,105.14
Average	**828.39**	**1,310.57**	**1,427.44**	**654.64**	**573.98**	**646.17**
Speed-up	**1.58**	**1.00**	**0.92**	**2.00**	**2.28**	**2.03**

5.1.2 Total Query Processing Time

In this section, we compare the total query processing time for the indexing schemes in two ways: by reporting the actual measurements and simulated results.

For the actual Total Query Processing Time, we measure the total time from the beginning of the query processing to the end of it. To make our measurements more reliable, we restarted our server before running each experiment (for a given indexing scheme and parameter combination). By doing so, we aim to reduce the caching effects (of disk pages) among different experiments.

While the aforementioned approach may reduce the caching effect across experiments, it is still vulnerable to inter-query caching, e.g., when the same keyword appears in more than one query. As a remedy, we calculate the Simulated Total Query Processing time, which essentially computes the disk access times using disk parameters and disk access statistics that are logged during the processing of each query, as in Formula 5.2.

$$ReadTime = 1/(TransferSpeed(Bytes/Sec)/PageSize) \tag{5.1}$$

$$SimulatedTime = MemoryTime + (SeekTime * IOCount) + (ReadTime * PageCount) \quad (5.2)$$

As can be seen in Tables 5.4 and 5.5, CS-IIS indexes clearly outperform IR-Tree and DIR-Tree in terms of both measured and simulated total query processing time. Interestingly, even the Full Index is better than the IR-Tree and DIR-Tree indexing schemes. The reason for the inferior performance of the latter approaches is their large number of direct disk I/Os (as will be discussed in the next section). For a query of n keywords, both Full Index and CS-IIS variants make only n direct I/Os, while tree-based indexes may need to access several leaf nodes until the top-k result is constructed.

We also see the parallelism between Total Query Processing Time and Simulated Query Processing Time results reported in the respective Tables 5.4 and 5.5. While the presented values differ (as the actual disk access times may differ from the analytical results due to disk workload during the experiments), the trends are similar. Overall, our findings regarding memory and total query processing times reveal that our approach using the CS-IIS scheme is more efficient than all its competitors in addressing the semantic place retrieval problem.

5.2 Direct Disk Access Count

Direct Disk Access count is the number of direct I/O operations, i.e., the number of times a disk seek is required to access a position on the disk. This is another measure that the CS-IIS is an obvious winner. The Full Index and CS-IIS make a disk access per query term to get its posting list, while IR-Tree and DIR-Tree need to access several tree nodes. Tables 5.6 and 5.7 show that the difference in access patterns is remarkable, as Full Index and CS-IIS makes an order of magnitude smaller number of accesses. Furthermore, the tree-based indexes need to make an even larger number of I/Os, as shown in the case of retrieving top-20 results, while the number of I/O operations for Full Index and CS-IIS is fixed, i.e., equal to the number of query keywords.

Table 5.5: Simulated total processing time (ms) for top-10 results (speed-up is reported wrt. IR-Tree baseline for the average case)

Number of Keywords	**Baseline**			**Proposed**		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	166.10	218.97	203.17	47.15	45.52	50.41
2	361.80	349.63	314.66	98.83	102.60	109.31
3	316.16	453.23	393.92	144.88	146.36	157.82
4	530.13	571.22	522.94	190.35	187.29	202.40
5	415.31	760.12	670.45	240.75	233.52	258.45
6	465.37	769.38	705.45	274.20	271.15	289.92
7	538.21	937.29	852.09	323.05	310.36	341.55
8	745.77	1,076.29	975.78	369.37	369.88	390.80
Average	**412.16**	**593.33**	**536.01**	**192.84**	**190.29**	**205.81**
Speed-up	**1.44**	**1.00**	**1.11**	**3.08**	**3.12**	**2.88**

Table 5.6: Direct disk access count comparison for top-10 results

Number of Keywords	**Baseline**			**Proposed**
	Full Index	IR-Tree	DIR-Tree	CS-IIS
1	1.00	17.48	18.91	1.00
2	2.00	27.23	30.41	2.00
3	3.00	34.85	40.14	3.00
4	4.00	47.04	51.34	4.00
5	5.00	60.68	69.07	5.00
6	6.00	64.06	70.10	6.00
7	7.00	78.00	86.00	7.00
8	8.00	89.54	99.00	8.00
Average	**4.11**	**48.28**	**53.57**	**4.11**

5.3 Other Metrics

5.3.1 Disk Page Count

Disk Page Count shows how many pages are read from the disk during the execution of query processing. This metric differs from the Direct Disk Access Count since during an I/O operation many disk pages (say, corresponding to a long posting list)

Table 5.7: Direct disk access count comparison for top-20 results

Number of Keywords	Baseline			Proposed
	Full Index	IR-Tree	DIR-Tree	CS-IIS
1	1.00	19.75	21.03	1.00
2	2.00	29.10	32.28	2.00
3	3.00	40.70	46.97	3.00
4	4.00	50.28	54.79	4.00
5	5.00	64.02	74.39	5.00
6	6.00	71.45	77.68	6.00
7	7.00	80.36	88.14	7.00
8	8.00	100.39	108.09	8.00
Average	**4.11**	**52.63**	**58.14**	**4.11**

can be read. Therefore, we separately report the disk page count for our experiments.

Tables 5.8 and 5.9 show that the baseline approaches IR-Tree and DIR-Tree have fewer Disk Page Counts on average, which is only 48.28 for the IR-Tree. CS-IIS has a Direct Disk Access Count the same as the number of query terms, but on the other hand, CS-IIS reads the postings list of a query term as a whole, making the disk page count 350.55 on average for the CS-IIS 180 index. IR-Tree and DIR-Tree reach the disk for every node they encounter, which typically takes a single page. This increases their Direct Disk Access Count but reduces the Total Disk Page Count they read.

Disk Page Count is insufficient to make IR-Tree and DIR-Tree successful in terms of the disk operation metrics. Even if they have a fewer Disk Page Count, for modern disks the transfer speed is considerably faster than the seek time, making the Direct Disk Access Count a more important metric. In particular, for the disk in our experimental setup, the transfer speed is measured as 190 Mb/sec, implying a latency 0.3 ms for reading a page with 64000 bytes. In contrary, average seek time is 10 ms, which is an order of magnitude slower.

5.3.2 Total Byte Size

Total Byte size measures how many bytes are read from the disk during the algorithm evaluation. Obviously, this is correlated with the previous metric, number of disk

Table 5.8: Total disk page count comparison for top-10 results

Number of Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	64.38	17.48	18.91	68.80	64.65	72.75
2	163.30	27.23	30.41	174.38	163.74	184.39
3	251.06	34.85	40.14	268.13	251.81	283.71
4	320.68	47.04	51.34	342.62	321.54	362.32
5	418.07	60.68	69.07	446.87	419.13	473.11
6	470.60	64.06	70.10	502.30	471.69	531.43
7	562.23	78.00	86.00	600.27	563.04	635.12
8	636.12	89.54	99.00	679.25	636.89	718.93
Average	**328.23**	**48.28**	**53.57**	**350.55**	**328.96**	**370.91**

Table 5.9: Total disk page count comparison for top-20 results

Number of Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	64.38	19.75	21.03	68.80	64.65	72.75
2	163.30	29.10	32.28	174.38	163.74	184.39
3	251.06	40.70	46.97	268.13	251.81	283.71
4	320.68	50.28	54.79	342.62	321.54	362.32
5	418.07	64.02	74.39	446.87	419.13	473.11
6	470.60	71.45	77.68	502.30	471.69	531.43
7	562.23	80.36	88.14	600.27	563.04	635.12
8	636.12	100.39	108.09	679.25	636.89	718.93
Average	**328.23**	**52.63**	**58.14**	**350.55**	**328.96**	**370.91**

Table 5.10: Total byte size (MB) comparison for top-10 results

Number of Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	4.08	0.30	0.33	4.36	4.10	4.62
2	10.39	0.43	0.48	11.09	10.42	11.74
3	15.98	0.52	0.61	17.06	16.02	18.06
4	20.40	0.61	0.68	21.79	20.44	23.07
5	26.60	0.79	0.94	28.43	26.66	30.14
6	29.93	0.79	0.88	31.95	29.99	33.83
7	35.77	0.93	1.05	38.18	35.80	40.45
8	40.47	1.04	1.21	43.20	40.50	45.77
Average	**20.88**	**0.64**	**0.72**	**22.30**	**20.92**	**23.61**

pages read. Hence, IR-Tree and DIR-Tree are again better than the CS-IIS and Full indexes for similar reasons explained in Section 5.3.1. The IR-Tree and DIR-Tree have read fewer bytes since their query processing algorithm reads posting lists corresponding per node, as described in Section 3.3. Naturally, the posting lists per node are shorter compared to the full postings lists of terms that are fetched by CS-IIS. As shown in Tables 5.10 and 5.11, IR-Tree reads 0.64 MB on the average. This value is 22.30 MB for the CS-IIS 180. Even if the total byte size is significantly larger, the total query processing time, including the disk reading time, is smaller for the CS-IIS. This difference has a good reason. While CS-IIS reads the posting lists as a whole, the IR-Tree divides them into smaller chunks for each node, increasing the number of direct disk accesses, each of which involves an expensive disk seek.

5.3.3 Number of Processed Document Postings

As implied by its name, this measure captures the number of document postings that are processed during query evaluation. In other words, the total number of $< place, looseness >$ pairs processed for each query is logged, and we report the average value over all the queries. For IR-Tree and DIR-Tree, whenever a new node is accessed during processing, the size of each posting list of each query keyword is added to this count. As shown in Table 5.12, the counts for other indexes are sig-

Table 5.11: Total byte size (MB) comparison for top-20 results

Number of Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	4.08	0.34	0.36	4.36	4.10	4.62
2	10.39	0.46	0.51	11.09	10.42	11.74
3	15.98	0.60	0.71	17.06	16.02	18.06
4	20.40	0.65	0.72	21.79	20.44	23.07
5	26.60	0.83	1.00	28.43	26.66	30.14
6	29.93	0.87	0.97	31.95	29.99	33.83
7	35.77	0.95	1.08	38.18	35.80	40.45
8	40.47	1.14	1.28	43.20	40.50	45.77
Average	**20.87**	**0.68**	**0.77**	**22.29**	**20.91**	**23.61**

nificantly smaller than that for the Full Index. We explain this as follows. IR-Tree and DIR-Tree stop execution when they reach the top-k results. CS-IIS also employs an early-stopping technique. But Full Index doesn't apply such a methodology, and hence, should process all the document postings. In particular, the IR-Tree has processed 15,979.92, and the CS-IIS 180 processed 18,618.87 postings on the average, as shown in Table 5.12. In contrast, the Full Index processed 570,108.36 postings on the average.

Table 5.12: Number of processed document postings for top-10 results

Number of Keywords	Baseline			Proposed		
	Full Index	**IR-Tree**	**DIR-Tree**	**CS-IIS 180**	**CS-IIS 36**	**CS-IIS 300**
1	111,727.15	2,339.20	2,814.54	2,601.88	5,783.40	2,396.07
2	283,963.91	7,366.18	8,638.73	4,113.61	18,154.51	3,081.14
3	436,647.61	11,623.57	14,171.77	6,992.34	27,197.15	5,748.10
4	557,293.48	14,917.60	17,017.41	15,001.85	35,904.11	12,146.91
5	726,474.56	21,823.23	26,558.64	24,441.05	64,548.53	20,853.19
6	817,373.36	22,771.50	26,025.19	31,706.26	70,956.28	26,870.41
7	975,775.44	28,042.07	32,417.37	33,828.61	64,555.07	29,355.53
8	1,103,735.94	32,785.70	38,919.05	50,014.96	100,475.18	45,152.69
Average	**570,108.36**	**15,979.92**	**18,782.77**	**18,618.87**	**43,569.39**	**15,998.14**

Table 5.13: Number of processed grid postings for CS-IIS for top-10 results

Number of Keywords	CS-IIS 180	CS-IIS 36	CS-IIS 300
1	69.26	14.90	122.37
2	8.66	3.82	13.47
3	10.42	3.90	18.38
4	19.31	5.32	33.18
5	35.57	9.08	62.37
6	23.64	5.70	42.23
7	25.36	4.83	51.61
8	29.61	6.19	57.82
Average	**30.21**	**7.27**	**54.20**

5.3.4 Number of Processed Grid Postings

Processed grid count is the number of grids processed by our approach on CS-IIS until the early termination. This is an important metric that clearly shows the benefit of employing grid structure in the indexing scheme. As explained in Chapter 2.3, the main advantage of the grid-based indexing schemes is eliminating some grids even before processing.

As shown in Table 5.13, CS-IIS variants for the different numbers of grids (i.e., ranging from 36×36 to 300×300) process significantly smaller numbers of grid postings, i.e., between 7.27 and 54.20, on the average. We think that this finding is essentially due to the Ranked AND query processing applied in our setting.

In addition to the number of processed grid postings, we also report its ratio to the total number of grid postings that appear in the grid-intersection set of a query's postings lists, to emphasize the impact of our early-stopping. Table 5.14 reveals that on average, the ratio is 18% for both CS-IIS 180 and CS-IIS 36, and 19% for CS-IIS 300. These figures imply that the early-stopping approach applied during the query processing with CS-IIS is quite effective.

Table 5.14: Ratio of the processed grid postings for CS-IIS for top-10 results

Number of Keywords	CS-IIS 180	CS-IIS 36	CS-IIS 300
1	0.32	0.32	0.32
2	0.10	0.12	0.10
3	0.10	0.11	0.11
4	0.14	0.14	0.14
5	0.15	0.15	0.16
6	0.22	0.19	0.23
7	0.16	0.13	0.18
8	0.21	0.17	0.23
Average	**0.18**	**0.18**	**0.19**

5.4 Summary of Experimental Findings

Our extensive experiments presented in this chapter demonstrate that our adaptation of CS-IIS with early-stopping is the best-performing approach for the semantic place retrieval problem. Specifically, our approach significantly outperforms the most efficient baseline using an IR-Tree in terms of the total query processing time (using either measured or simulated time figures). This is due to the small number of direct disk accesses required by the CS-IIS, while the number of accesses made by the IR-Tree is an order of magnitude larger. Since disk seeks are more costly than data reading, CS-IIS keeps its advantage although it fetches a larger number of disk pages in total.

Note that, even if the disk access cost is totally neglected (i.e., assuming all indexes are kept in memory), our approach can still outperform its competitor, as CS-IIS (using 180×180 grids) provides a speed-up of 1.10x over the IR-tree based approach in terms of the in memory query processing time. Our gains are even more emphasized for queries with a small number of keywords, which is more likely in practical search applications. Experiments also show that the early-stopping strategy significantly reduces the number of grids considered during the query processing, which contributes to the superior in memory processing efficiency of our approach with CS-IIS.

CHAPTER 6

CONCLUSIONS

In this work, we present an alternative approach to address the problem of semantic place retrieval, which aims to retrieve top-k places on a knowledge graph using both textual and spatial scores.

Our work makes several important contributions to scientific literature. First, we implemented and analyzed a baseline approach that conducts an on-the-fly traversal of the knowledge graph to answer the semantic place queries. We observed that such a traversal is prohibitively costly in terms of in memory time and total query processing time measurements. As a remedy, we hypothesized that query keywords that are located at very distant nodes from the place may not be helpful, and hence, enforced a pre-defined limit for the depth of graph search, i.e., we did not follow the paths beyond this depth limit.

Setting a limit on search depth allowed us to determine all the terms related to a place during an offline pre-processing stage. Thus, as our second contribution, we applied two well-known geo-indexing approaches, IR-Tree and DIR-Tree [5], to this problem. To this end, we extended a public codebase from an earlier work [8]. Our extensive experiments showed that these approaches yield high efficiency in terms of in memory query processing time, but they cause a large number of direct disk accesses, making them inferior to even using a simple Full Index in terms of total query processing time.

As a third contribution, we adapted the cluster-skipping inverted index structure (CS-IIS) [9] that has been proposed for search over clustered text document collections. In the postings lists of CS-IIS, instead of topical clusters, we grouped postings for

places according to the geographical grids they appear in. We also tailored an early-stopping technique that terminates query processing when it is guaranteed that unprocessed grids cannot include a place that can get into the top-k results. Our experiments demonstrated that CS-IIS needs just a few direct disk accesses (i..e, equal to the number of query keywords) and hence, it performs better than all the baseline approaches. In particular, CS-IIS (with 180×180 grids) is 9% better than IR-Tree for the in memory execution time. When the simulated total query processing time is considered, CS-IIS is 64% better than IR-Tree.

There are several exciting future work directions. It seems promising to create a hybrid of the CS-IIS and IR-tree, to have the best of both approaches. We also plan to extend our research by comparing CS-IIS with other indexing approaches from the literature. Finally, we plan to investigate the performance of other query types, such as Ranked OR queries and range queries.

REFERENCES

[1] S. Vaid, C. B. Jones, H. Joho, and M. Sanderson, "Spatio-textual indexing for geographical search on the web," in *International Symposium on Spatial and Temporal Databases*, pp. 218–235, Springer, 2005.

[2] D. Wu, H. Zhou, J. Shi, and N. Mamoulis, "Top-k relevant semantic place retrieval on spatiotemporal rdf data," *The VLDB Journal*, vol. 29, no. 4, pp. 893–917, 2020.

[3] M. Sanderson and J. Kohler, "Analyzing geographic queries," in *SIGIR Workshop on Geographic Information Retrieval*, vol. 2, pp. 8–10, 2004.

[4] "Microsoft: 53 percent of mobile searches have local intent." `https://searchengineland.com/microsoft-53-percent-of-mobile-searches-have-local-intent-55556`, 2010. Accessed: 2022-12-21.

[5] G. Cong, C. S. Jensen, and D. Wu, "Efficient retrieval of the top-k most relevant spatial web objects," *Proceedings of the VLDB Endowment*, vol. 2, no. 1, pp. 337–348, 2009.

[6] Z. Cai, G. Kalamatianos, G. J. Fakas, N. Mamoulis, and D. Papadias, "Diversified spatial keyword search on RDF data," *VLDB J.*, vol. 29, no. 5, pp. 1171–1189, 2020.

[7] J. Shi, D. Wu, and N. Mamoulis, "Top-k relevant semantic place retrieval on spatial rdf data," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1977–1990, 2016.

[8] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: An experimental evaluation," *Proceedings of the VLDB Endowment*, vol. 6, no. 3, pp. 217–228, 2013.

[9] I. S. Altingovde, E. Demir, F. Can, and O. Ulusoy, "Incremental cluster-based retrieval using compressed cluster-skipping inverted files," *ACM Trans. Inf. Syst.*, vol. 26, jun 2008.

[10] F. Can, I. S. Altingövde, and E. Demir, "Efficiency and effectiveness of query processing in cluster-based retrieval," *Information Systems*, vol. 29, no. 8, pp. 697–717, 2004.

[11] I. S. Altingövde, E. Demir, F. Can, and Ö. Ulusoy, "Site-based dynamic pruning for query processing in search engines," in *Proceedings of the 31st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 861–862, 2008.

[12] J. Mackenzie, M. Petri, and A. Moffat, "Anytime ranking on document-ordered indexes," *ACM Transactions on Information Systems (TOIS)*, vol. 40, no. 1, pp. 1–32, 2021.

[13] G. Salton and M. J. McGill, *Introduction to modern information retrieval*. mcgraw-hill, 1983.

[14] J. B. Rocha-Junior, O. Gkorgkas, S. Jonassen, and K. Nørvåg, "Efficient processing of top-k spatial keyword queries," in *International Symposium on Spatial and Temporal Databases*, pp. 205–222, Springer, 2011.

[15] A. Khodaei, C. Shahabi, and C. Li, "Hybrid indexing and seamless ranking of spatial and textual features of web documents," in *International Conference on Database and Expert Systems Applications*, pp. 450–466, Springer, 2010.

[16] J. Zobel and A. Moffat, "Inverted files for text search engines," *ACM Computing Surveys (CSUR)*, vol. 38, no. 2, pp. 6–es, 2006.

[17] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pp. 47–57, 1984.

[18] G. K. Zipf, *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio Books, 2016.

[19] N. Jardine and C. J. van Rijsbergen, "The use of hierarchic clustering in information retrieval," *Information Storage and Retrieval*, vol. 7, no. 5, pp. 217–240, 1971.

[20] "Rdf." `https://www.w3.org/RDF/`, 2014. Accessed: 2022-12-17.

[21] "Sparql 1.1 protocol." `https://www.w3.org/TR/sparql11-protocol/`, 2013. Accessed: 2022-12-17.

[22] B. Croft, D. Metzler, and T. Strohman, *Search engines: Information retrieval in practice*. Pearson Education, 2011.

[23] J. Mackenzie, F. M. Choudhury, and J. S. Culpepper, "Efficient location-aware web search," in *Proceedings of the 20th Australasian Document Computing Symposium*, pp. 1–8, 2015.

[24] B. Schnitzer and S. T. Leutenegger, "Master-client r-trees: a new parallel r-tree architecture," in *Proceedings. Eleventh International Conference on Scientific and Statistical Database Management*, pp. 68–77, IEEE, 1999.

[25] "YAGO 3." `https://yago-knowledge.org/downloads/yago-3`, 2022. Accessed: 2022-08-14.

[26] F. Mahdisoltani, J. Biega, and F. Suchanek, "Yago3: A knowledge base from multilingual wikipedias," in *7th Biennial Conference on Innovative Data Systems Research*, CIDR Conference, 2014.

[27] "Google keyword study." `https://backlinko.com/google-keyword-study`, 2020. Accessed: 2022-11-29.

[28] G. Pass, A. Chowdhury, and C. Torgeson, "A picture of search," in *Proceedings of the 1st International Conference on Scalable Information Systems (InfoScale)*, 2006.