M. Sc. Engineering Thesis

Spatio Temporal Keyword Search for Nearest Neighbor Queries

by

Saif-Ul-Islam Khan

Submitted to

Department of Computer Science and Engineering

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE & ENGINEERING

Department of Computer Science and Engineering

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Dhaka-1000, Bangladesh

April, 2014

The thesis titled "**Spatio Temporal Keyword Search for Nearest Neighbor Queries**", submitted by Saif-Ul-Islam Khan, Roll No. 0411052003P, Session April 2011, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory in partial fulfillment of the requirements for the degree of Master of Science in Computer Science and Engineering and approved as to its style and contents. Examination held on April 26, 2014.

Board of Examiners

1. _____

Dr. Mohammed Eunus Ali — Chairman
Associate Professor — (Supervisor)
Department of CSE, BUET, Dhaka 1000.

2. _____

Dr. Mohammad Mahfuzul Islam — Member
Professor & Head — (Ex-officio)
Department of CSE, BUET, Dhaka 1000.

3. _____

Dr. Md. Mostofa Akbar — Member
Professor
Department of CSE, BUET, Dhaka 1000.

4. _____

Dr. Md. Yusuf Sarwar Uddin — Member
Assistant Professor
Department of CSE, BUET, Dhaka 1000.

5. _____

Dr. Mohammad Nurul Huda — Member
Professor — (External)
Department of CSE & MSCSE Coordinator
United International University,Dhaka-1209.

Candidate's Declaration

This is to certify that the work presented in this thesis entitled "**Spatio Temporal Keyword Search for Nearest Neighbor Queries**" is the outcome of the investigation carried out by me under the supervision of Associate Professor Dr. Mohammed Eunus Ali in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), Dhaka. It is also declared that neither this thesis nor any part thereof has been submitted or is being currently submitted anywhere else for the award of any degree or diploma.

Saif-Ul-Islam Khan
Candidate

Contents

List of Figures

List of Tables

Acknowledgments

I thank Allah for giving me the ability to work on the thesis. My deepest gratitude goes to my supervisor, Associate Prof. Dr. Mohammed Eunus Ali, who was the only person behind motivating me to work in this hybrid indexing field. Starting from the idea upto writing he was with me. Indeed, he played the most instrumental role in my thesis completion and I shall forever treasure this entire period that I have spent working with him.

I earnestly thank my thesis committee members, Prof. Dr. Mohammad Mahfuzul Islam, Prof. Dr. Md. Mostofa Akbar, Prof. Dr. Mohammad Nurul Huda and Assistant Prof. Dr. Md. Yusuf Sarwar Uddin, whose suggestions were very beneficial for this work.

I would deeply like to thank my ever-caring wife, my loving parents and parents in law for consistently encouraging me, with great patience, at every moment of this wonderful and challenging journey of completing my thesis. I am also grateful to my mother from whom I often used to take help while writing this book. Finally, I would like to record my deep appreciation and respect with my heavy heart to the very recently departed soul of my elder sister, Dr. Tabassum F. Khan who in spite of her long enduring struggle with cancer in her mid-forties, helped me immensely in preparing this book. She left this world throwing us in the sea of bereavement only on the other day 13 Apr 2014. May Allah keep her soul in peace.

Abstract

The widespread availability and the technological advancements of geo-positioning devices enable users to generate a huge volume of geo-tagged objects everyday. These objects include points of interest (e.g., restaurants), photos, and buying/selling items. To describe such an object, which is commonly referred as a spatial objet, users often use textual information or keywords along with the geographic location of the entity. Based on these geo-tagged objects, a large variety of location based services has been emerged. For example, a user often issues a query like *"find the Italian restaurant nearest to my location"* to a location based service provider (LSP), and the LSP returns the Italian hotel that is nearest to the user's location as an answer. Due to the popularity of keyword search, this field leads much work on querying spatial keyword (SK) search. However, there are many new applications which require incorporating time along with location and textual information, e.g., *"find the Italian restaurant nearest to my location which opens at 10pm today"*. We term this type of query as an *spatio-temporal keyword* (STK) query. A straightforward way of answering STK queries using existing spatial keyword search technique requires retrieving objects that are not temporally relevant to the query time. To solve this issue, in this paper, we introduce a new index structure that hierarchically organizes time along with location and keywords, and develop an efficient algorithm for processing STK queries. We also extend our work to handle time uncertainty. An extensive experimental study shows the efficiency and effectiveness of our proposed techniques.

Chapter 1

Introduction

The proliferation of location aware portable devices (e.g., GPS enabled mobile phones) enables users to generate a large amount of geo-tagged objects. These objects include different point of interests (POIs), saleable items, photos, etc. Each object is defined using textual information (keywords), location information (latitude and longitude), and other meta-data (e.g., time, image, etc.). These data sets provide a new platform for answering a large variety of location based services (LBSs) like Google Map, Foursquare. For example, a user may be interested in finding the nearest Italian restaurant from her location. To find the nearest spatial object (e.g., restaurant) that matches with user interest (e.g., Italian restaurant), the user provides her location and keywords (e.g., Italian) to the location based service provider (LSP). The LSP returns the nearest object whose description/keywords matches with the query the search keywords. This type of query is known as a spatial keyword query.

Existing works on spatial keyword search focus on developing hybrid index structures that organize both locations and keywords in a hierarchical tree [7, 8, 10]. These approaches essentially merge location index $R-$tree with keyword index inverted file. These approaches are suitable for processing different types of spatial keyword queries such as kNN query and top-k NN query. A kNN query finds the k objects nearest to the user's location, where each object's text description contains query keywords; whereas a top-k NN query returns the k objects with the highest scores measured as a combination of the distance from the user location and the relevance of their text description with the given query keywords.

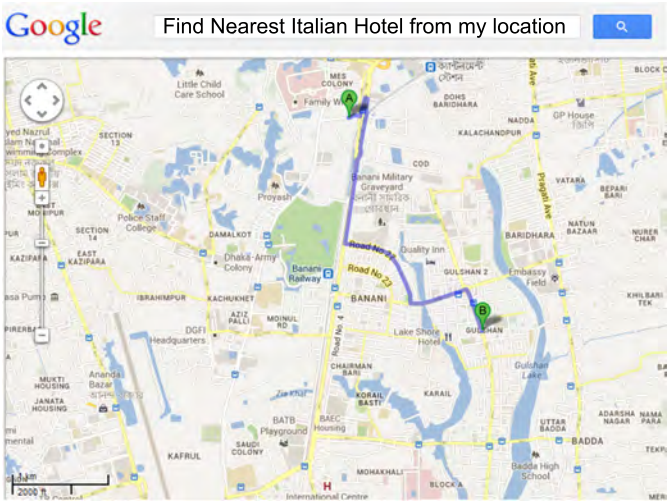


Figure 1.1: Google Map

Emergence of location based information has contributed many popular application. For example location base service (LBS) like Google Map, Foursquare, location based marketing (LBM) and location based advertisement (LBA) like facebook. With LBM or LBA it becomes easier for the marketer or advertiser to send relevant product information or advertise to nearest users' mobile. Again users can also search for nearest shop having their desired products by utilizing their location information and text information. Google Maps, Facebook allow the users to provide a list of keyword or text information that a spatial objects contain and returns the nearest spatial objects. Such queries require the processing of text relevancy and location proximity to prune the search space. Users very often are interested in a number nearest objects than a single nearest object. This draws attention on development of *Top-k* retrieval of objects. Here k is the desired number of objects in ordered of their increasing distance from user location.

We envision a new set of applications that require incorporating *time* along with spatial and textual information. Let us consider the following scenario. Farmers in rural area of Bangladesh produce a plenty of seasonal fruits and vegetables. In most of the cases, these farmers need to sale these items as quickly as possible to avoid being rotten as they do not have any access to cold storage. Due to lack of infrastructure, these farmers cannot afford to carry these items to distant urban area where the demand is more. Middle-men exploit these shortcomings of farmers, and deliberately reduce the buying price. Hence,

Figure 1.2: Location based marketing

farmers did not get any profit, and even in some cases incurred huge loses, of their hard-earned investment. It has become a serious concern of farmers in recent years [1, 2].

To solve the problem, we envision a novel concept of future marketing for farmers, which can be described as follows. After planting and harvesting the crops, farmers generally know the probable *time* when their products will be ready for sale. Thus, farmers can plan ahead and start selling their products in advance. This allows a farmer to utilize a time span to market their product to potential buyers. Also, buyers can plan to buy appropriate items based on their near future demand. Thereby, for such a future marketing platform, a large number of buyers may continuously search for sellers to get their desired products at their expected time. Similarly sellers may also search for suitable buyers to sell their products in time. In such cases, a seller publishes an item location, description, and a time range denoting the expected availability duration of the item. A buyer issues a query that contains its location, item description, and the time when the buyer needs the item. As we can see, to answer such queries, we need to manage time along with spatial (location) and textual (description) attributes of data objects (items).

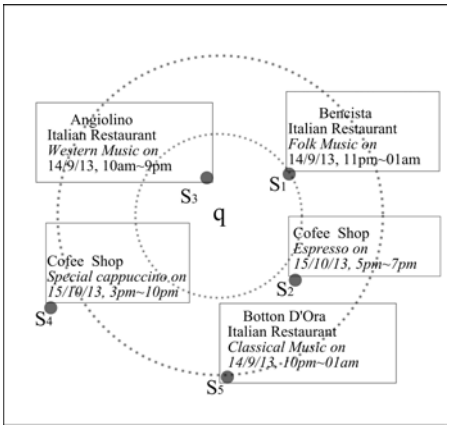


Figure 1.3: Location of spatio-temporal objects

Besides future marketing, another class of application requires handling location, time and keyword together. For example, a user may want to search for the nearest restaurant of her choice (e.g., Italian) that is open or hosts an event of her choice (e.g., music event) at a specific time. To answer such queries, the LSP stores list of restaurants with their locations, descriptions, and time for opening/hosting different events.

In all of the above scenarios, since a query involves location, time, and keywords, we term it as a spatio-temporal Keyword (STK) query. When a user is interested on finding the nearest object with respect to his location, we call this as an STK nearest neighbor (STK-NN) query. Generalization of the STK-NN query is termed as an STK-kNN query where the user is interested in finding the k nearest object.

Figure 1.3 represents an example of an STK-NN query, where the point location of the spatio-temporal objects S_1, S_2, S_3, S_4, and S_5 are tagged with text and time. Once a user located at a point location q submits a query *"find the Italian restaurant that is nearest my location and has a music event during 11pm - 1am today"*. This query returns a restaurant that is nearest to the user's location and matches keywords (*"Italian", "Restaurant", "Music"*) and time (11pm - 1am today). In this example S_1 and S_5 are reported as first and second nearest object respectively. Though S_3 matches it's textual description and is the nearest object among all but it is not retrieved as it's time does not match the query's time.

To solve an STK-kNN query, a straightforward approach is to use existing

hybrid indexing techniques where location and text are indexed and time is filtered out at the final level. However, this approach requires high processing time and I/O cost as time is not considered as a while retrieving objects by traversing the index. To resolve this problem, we develop a new hybrid index structure called *spatio-temporal inverted file-R (STIR) tree* that includes time dimension into hierarchical structure embedded with spatial and textual attributes. We incorporate the time in the STIR-tree in such a way that lower level contains the fine granularity of time and upper level contains the coarse granularity of time while constructing STIR-tree. Our pruning starts from the beginning of tree traversal by keyword and *time* together and thereby reduce processing time and I/O costs significantly. Based on our index structure we propose an efficient algorithm that finds kNN spatio- temporal objects.

(a) An example of Time Uncertainty (b) An example of Nearest Time

Figure 1.4: Seller and buyer community

We also propose a variant of the STK-kNN query that incorporates time *uncertainty* in the query evaluation. In our motivating example, farmers may not be able to fix a specific date/time for the availability of a product, however, they can predict a date/time range when the product will be ready for sell (e.g., a seller/farmer publishes that popular seasonal fruit "Lichi" will be available anytime between "01 Feb 2014 to 07 Feb 2014"). As we can see that the value for time data is uncertain in this case, and thus the query answer will have some associate probabilities. We propose an efficient algorithm for processing STK-kNN queries with time uncertainty. In addition to all of the above queries, we have also proposed another variant that finds the closest match with respect

a give query time instead of considering the location proximity.

In summary, the contribution of this paper are as follows.

- We formulate the problem of spatio-temporal keyword search for the nearest neighbor (STK-NN) query.
- We develop time indexing and an efficient hybrid index structure that integrates location, keyword, and time.
- We present efficient algorithm for answering STK-kNN.
- We extend our STK-kNN query to support time uncertainty.
- We conduct an extensive experimental study which shows that our approach outperforms the straightforward approach significantly.

The structure of this book is as follows: In Chapter 2 we discuss the preliminaries. In Chapter 3 we provide an overview of the related work. In Chapter 4 we define our problem definition. In Chapter 5 we describe our hybrid indexing. In Chapter 6 we show our three types of query processing and their algorithm for spatio-temporal keyword search for nearest neighbor. Chapter 7 reports our experimental result and finally Chapter 8 concludes the book.

Chapter 2

Preliminaries

Today data is housed and managed via a database management system (DBMS). Traditional role of DBMS is simple but effective warehouse of business and accounting data. Data residing in these databases is simple, consisting of number, names, addresses, product information etc. These DBMS are very efficient for the task they were designed for. For example, a query like "List the top ten customer, in terms of sales, in the year 2014" will be very efficiently answered by a DBMS even if the database has to scan through a large customer database. Such commands are called "queries". The database will not scan all the customer, it will use an index. On the other hand, a relatively simple query such as "List all the customer who resides within fifty miles of the company headquarters" will confound the database. To process this query, database will have to transform the company headquarters' and customers' addresses into suitable reference system, possibly latitude and longitude, in which distance can be computed and compared. Then database has to scan through the entire customer list, compute the distance between the company and the customer, and if this distance less than fifty miles, save the customer's name. It will not be able to use an index to narrow down the search, because traditional database are incapable of ordering multidimensional coordinate data. Therefore the need for database demands for handling spatial data and spatial queries.

In this chapter we will discuss spatial databases, spatial data (objects), spatial index, spatial queries, text search, keyword index and hybrid index structure based on spatial data and text.

2.1 Spatial Databases

Spatial data is known as geospatial data that identifies the geographic location of features and boundaries on Earth, such as natural or constructed features, oceans, and more. Spatial data is usually stored as coordinates and topology, and is data that can be mapped. Spatial data is often accessed, manipulated or analyzed through Geographic Information Systems (GIS). A spatial database system is a database system that offers additional support to spatial data or objects such as points, lines, and polygons. A spatial database system provides support for spatial objects in its data model and query language, and employs spatial indexes for processing spatial queries efficiently. In this section, we focus on the main concepts of spatial database systems employed in Geographical Information Systems (GIS).

2.1.1 Spatial Queries

Before describing spatial queries, we state an assumption about the underlying space and present the most important spatial relationships employed in spatial database systems.

- **Underlying space.** Unless explicitly mentioned, an Euclidean space is implicitly assumed. Therefore, a point (object) is represented by a pair of real numbers, and the distance between two points is defined by the Euclidean distance.

- **Spatial relationships.** The spatial algebra offers a set of spatial relationships among the objects. The most important are: 1) topological such as adjacent, inside, and disjoint; 2) directional such as above, below, and left; and 3) metric such as distance.

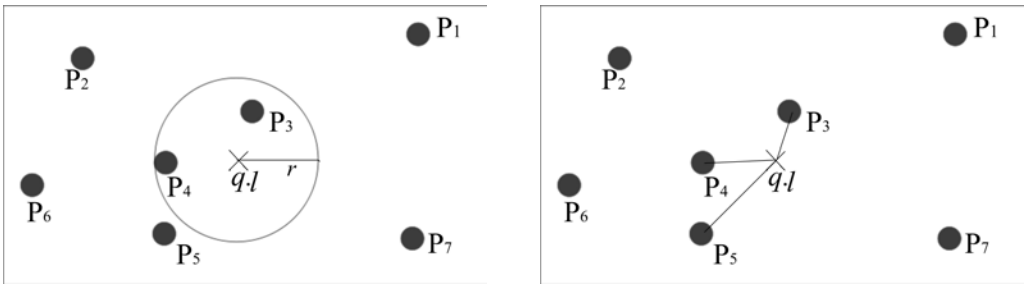
The most important query types supported by spatial database systems are spatial selection and spatial join. Given a dataset, a spatial selection returns the set of objects that satisfy a predicate. The predicate can contain one or more spatial relationships. For example, "Find all restaurants within a radius 100m from my current location". A spatial join, on the other hand, compares two or more datasets through a predicate on their spatial attributes. For example, "Find all pairs of farms and rivers that intersect".

There are two major spatial queries , such as:

Range query. Given a query location $q.l$ (point) and a distance r (radius) from $q.l$, the range query returns all objects p whose distance to $q.l$ is smaller or equals r, $dist(p, q.l) \leq r$.

Nearest neighbor queries are also very popular in spatial database systems and is defined as:

Nearest neighbor query(k-NN). Given a query location $q.l$ and an integer k, the k-NN query returns the k objects nearest to $q.l$ (shortest distance).



(a) Range query (b) Nearest neighbor (3-NN)

Figure 2.1: Example of spatial queries

Example. Figure 2.1 shows an example of range and nearest neighbor queries. In Figure 2.1a, the range query returns the object *p3* and *p4* that are within a distance r to *q.l*. In Figure 2.1b, the nearest neighbor query (3-NN) returns the three objects *p3, p4,* and *p5* nearest to the query location *q.l*.

2.1.2 Spatial Indexes

In this section, we describe one spatial indexes used in the thesis: *R-tree*. In addition, we present how to perform spatial queries employing these indexes.

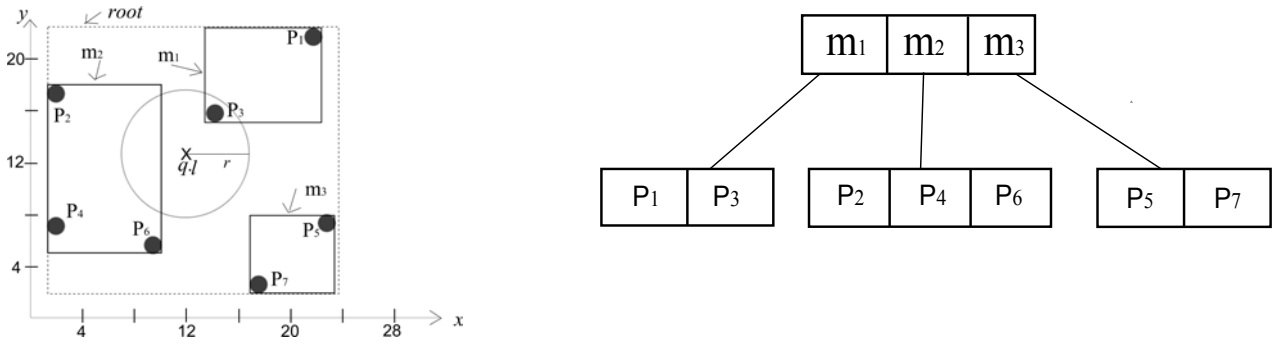
R-tree

One of the most influential access methods in the area of Spatial Data Management is the *R-tree* structure proposed by Guttman in 1984 [13]. It is a hierarchical data structure based on B^+-trees, used for the dynamic organization of a set of d-dimensional geometric objects. The original R-tree was designed for efficiently retrieving geometric objects contained within a given

query range. Every object in the R-tree is represented by a minimum bounding d-dimensional rectangle (MBR). Data objects are grouped into larger MBRs forming the leaf nodes of the tree. Leaf nodes are grouped into larger internal nodes. The process continues recursively until the last group of nodes that form the root of the tree. The root represents an MBR that encloses all objects and nodes indexed by the tree, and each node corresponds to the MBR that bounds its children (Figure 2.2a). A range query can be answered efficiently by traversing the tree starting from the root and ending at the leaves, accessing only nodes whose MBR intersect with the query range. At the leaf level of the tree, the actual geometric objects are retrieved and tested for true containment in the query.
An R-tree of order *(m,M)* has the following characteristics:

- The root node of the tree contains at least two entries, unless it is a leaf (in this case, it may contain zero or a single entry).
- Internal nodes(Figure 2.3a) can store between $m \leq M/2$ and M child entries. Each entry is of the form (*p, mbr*), where p is a pointer to a children node and *mbr* is the MBR that spatially encloses all entries contained in the sub-tree rooted at this child.
- Each leaf node (unless it is the root) as shown in Figure 2.3b can store between $m \leq M/2$ and M entries. Each entry is of the form (*oid, mbr*), where *oid* is an object identifier and *mbr* is the MBR that spatially encloses this object.
- The R-tree is a height-balanced structure, i.e., all leaves appear at the same level of the tree.

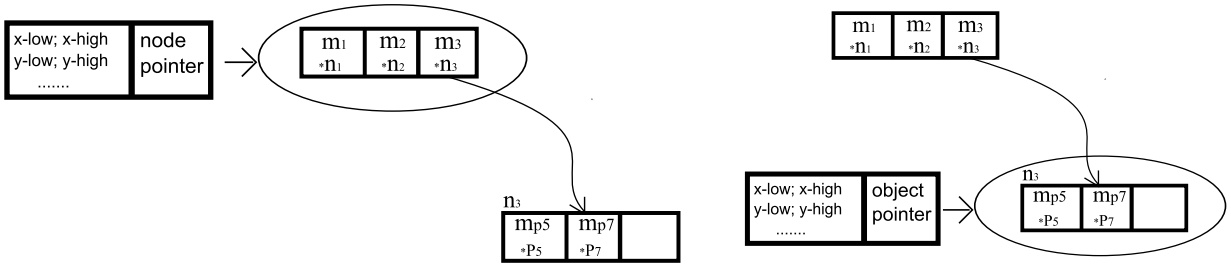
Figure 2.2 shows an example of a spatial area (Figure 2.2a) whose the objects are indexed in an R-tree (Figure 2.2b). In Figure 2.2b, the root is an intermediary node that contains three intermediary-entries pointing to the leaf-nodes n_1, n_2, and n_3. The intermediary-entry $(m_1, *n_1)$ has an MBR m_1 that encloses all objects in the node n_1, where $*n_1$ represents a pointer to n_1. The leaf-node n_1 contains two leaf-entries (m_{p_1}, p_1) and (m_{p_3}, p_3) where m_{p_1} is a MBR that bounds the spatial object p_1 and $*p_1$ is a pointer (identification) to p_1 in the



(a) Spatial query on R-tree

(b) R-tree

Figure 2.2: R-tree examples



(a) Structure of a non leaf node

(b) Structure of a leaf node

Figure 2.3: Node examples

dataset.

Example. Figure 2.2a shows an example of a range query on the R-tree. The query is looking for the objects within radius r from the cross mark $q.l$ (query location). The query starts at the root node searching for all entries whose MBRs have a minimum distance to $q.l$ smaller or equals r. Two entries satisfy this constraints $(m_1, *n_1)$ and $(m_2, *n_2)$. Therefore, the leaf-nodes n_1 and n_2 are accessed searching for the leaf-entries whose the MBRs are within a distance at most r from $q.l$. After this filtering step, the object p_3 is found.

2.1.3 Cost

Most algorithms that traverse R-tree in a top-down manner use some form of depth-first or breadth first tree traversal. Finding a leaf node containing a query object q in a spatial index can be done in a depth first manner by recursively descending the tree structure. With this method, the recursion stack keeps

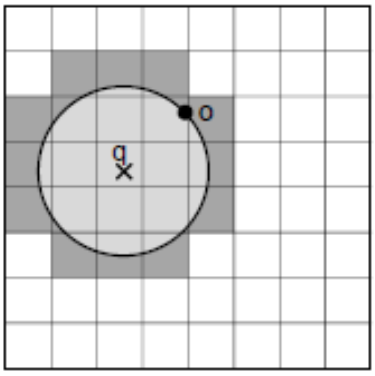


Figure 2.4: The circle around query object q depicts the search region after reporting o as next nearest object

track of what nodes have yet to be visited. Having reached a leaf, it needs to extend this technique to find the nearest object, as the leaf may not actually contain the nearest neighbor. The problem here is that it need to unwind the recursion to find the nearest object. Moreover, if we want to find the second nearest object, the solution becomes even tougher. With breadth-first traversal, the nodes of the tree are visited level by level, and a queue is used to keep track of nodes that have yet to be visited. However, with this technique, a lot of work has to be done before reaching a leaf node containing q. To resolve the problems with depth-first and breadth-first traversal, the incremental nearest neighbor algorithm employs what may be termed best-first traversal (BFS). When deciding what node to traverse next, it picks the node with the least distance in the set of all nodes that have yet to be visited. This means that instead of using a stack or a plain queue to keep track of the nodes to be visited, we use a priority queue where the distance from the query object is used as a key. The key feature of this solution is that the objects as well as the nodes are stored in the priority queue.

Best First Search using R-trees

If the spatial objects are stored external to the R-tree, such that leaf nodes contain only bounding rectangles for objects, then this adaptation leads to a considerably more efficient incremental algorithm. This enables the bounding rectangles to be used as pruning devices, thereby reducing the disk I/O needed to access the spatial descriptions of the objects. In Best First Search method,

the search starts from the root of the tree, and the child nodes are recursively accessed in the increasing order of their distances from the query point. The search process terminates as soon as the k nearest data points (kNN) are retrieved from the tree. The Best First Search method maintains a priority queue to store nodes to be explored through the search process. The nodes in the priority queue are sorted according to their minimum distance to the query point. During the search process, Best First Search method repeatedly dequeues top entry in the queue and enqueues its child nodes with their minimum distance into the queue. When a data point is dequeued, it is included in the result set.

Suppose that we want to find the three nearest neighbors to query point q in the R-tree given in Figure 2.2a, where the spatial objects are stored external to the R-tree. Below, we show the dequeue and enqueue process by Best First Search method for finding three nearest object. The process starts with enqueueing Root node and then executes the following step:

1. Dequeue *Root*, enqueue m_1, m_2, m_3
   Queue:$\{m_1, m_2, m_3\}$

2. Dequeue m_1, enqueue p_3, p_1
   Queue:$\{m_2, p_3, p_1\}$

3. Dequeue m_2, enqueue p_6, p_4, p_2
   Queue:$\{p_3, p_6, p_4, p_2, p_1\}$

4. Dequeue p_3, and report as 1st nearest object.
   Queue:$\{p_6, p_4, p_2, p_1\}$

5. Dequeue p_6, and report as 2nd nearest object.
   Queue:$\{p_4, p_2, p_1\}$

6. Dequeue p_4, and report as 3rd nearest object.

Finally p_3, p_6, and p_4 reported as 3 *NN* object.

Analysis

Let O be the k^{th} nearest neighbor of the query object q, and let r be the distance of O from q. The region within distance r from q is called the search

region. Since q is a point, the search region is a circle radius r. Figure 2.4 depicts this scenario. Observe that all objects inside the search region have already been reported by the algorithm (as the next nearest object), while all nodes intersecting the search region have been examined and their contents put on the priority queue. If n is a node that is completely inside the search region, all nodes and objects in the subtree rooted at n have already been taken off the queue. Thus all elements on the priority queue are contained in nodes intersecting the boundary of the search region (the dark shaded region in Figure 2.4). Any algorithm that uses a spatial index must visit all the nodes that intersect the search region; otherwise, it may miss some objects that are closer to the query object than O. Thus it is established that the algorithm visits the minimal number of nodes necessary for finding the k^{th} nearest neighbor. This can be characterized by saying that the algorithm is optimal with respect to the structure of the spatial index.

Generally, two steps are needed to derive performance measures for the incremental nearest neighbor algorithm. First, the expected area of the search region is determined. Then, based on the expected area of the search region and an assumed distribution of the locations and sizes of the leaf nodes, we can derive such measures as the expected number of leaf nodes accessed by the algorithm (i.e., intersected by the search region) or the expected number of objects in the priority queue. Henrich describes one such approach, which uses a number of simplifying assumptions [3]. In particular, it assumes N uniformly distributed data points in the two-dimensional interval [0; 1] X [0; 1], the leaf nodes are assumed to form a grid at the lowest level of the spatial index with average occupancy of c points, and the search region is assumed to be completely contained in the data space. Since we assume uniformly distributed points, the expected area of the search region is k/N and the expected area of the leaf node regions is c/N. The area of a circle of radius r is πr^2 , so for the search region we have $\pi r^2 = k/N$, which means that its radius is $r = \sqrt{k/\pi N}$. If the leaf node regions are squares, so their side length is $s = \sqrt{c/N}$. Henrich points out that the number of leaf node regions intersected by the boundary of the search region is the same as that intersected by the boundary of its circumscribed square [3]. Each of the four sides of the circumscribed square intersects $[2r/s] \leq 2r/s$ leaf node regions. Since each two adjacent sides intersect the same leaf node region

at a corner of the square, the expected number of leaf node regions intersected by the search region is bounded by

$$4(2r/s - 1) = 4\left(\frac{2\sqrt{k/(\pi N)}}{\sqrt{c/N}} - 1\right) = 4\left(2\sqrt{\frac{k}{\pi c}} - 1\right)$$

It is reasonable to assume that, on the average, half of the c points in these leaf nodes are inside the search region, while half are outside. Thus the expected number of points remaining in the priority queue (the points in the dark shaded region in Figure 2.4) is at most

$$\frac{c}{2}4\left(2\sqrt{\frac{k}{\pi c}} - 1\right) = 2c\left(2\sqrt{\frac{k}{\pi c}} - 1\right) = \frac{4}{\sqrt{\pi}}\sqrt{ck} - 2c \approx 2.26\sqrt{ck} - 2c$$

The number of points inside the search region (the light shaded region in Figure 2.4) is k. Thus the expected number of points in leaf nodes intersected by the search region is at most $k + 2.26\sqrt{ck}$-2c. Since each leaf node contains c points, the expected number of leaf nodes that were accessed to get these points is bounded by $k/c + 2.26\sqrt{k/c}$ - 2

To summarize, the expected number of leaf node accesses is $O(k + \sqrt{k})$ and the expected number of objects in the priority queue is $O(\sqrt{k})$.

2.2 Text Search

The technology underlying text search engines has advanced dramatically in the past decade. The development has led to a wide range of innovations in keyword indexing and query evaluation. Search engines are structurally similar to database systems. Documents are stored in a repository, and an index is maintained. Queries are evaluated by processing the index to identify matches which are then returned to the user. However, there are also many differences. Database systems must contend with arbitrarily complex queries, whereas the vast majority of queries to search engines are lists of terms and phrases. In a database system, a match is a record that meets a specified logical condition; in a search engine, a match is a document that is appropriate to the query according to statistical heuristics and may not even contain all of the query terms. Database systems return all matching records; search engines return a fixed number of matches, which are ranked by their statistical similarity.

2.2.1 Query Modes

In traditional databases, the primary method of searching is by key or record identifier. Such searching is rare in text databases. The dominant mode of text search is by its content in order to satisfy an information need. People search in a wide variety of ways. Perhaps the commonest mode of searching is to issue an initial query, scan a list of suggested answers, and follow pointers to specific documents. If this approach does not lead to discovery of useful documents, the user refines or modifies the query and may use advanced querying features such as restricting the search domain or forcing inclusion or omission of specific query terms. In this model of searching, an information need is represented by a query, and the user may issue several queries in pursuit of one information need. Users expect to be able to match documents according to any of the terms they contain. Another contrast with traditional databases is the notion of matching. A record matches an SQL query if the record satisfies a logical condition. A document matches an information need if the user perceives it to be relevant. But relevance is inexact, and a document may be relevant to an information need even though it contains none of the query terms or irrelevant even though it contains them all. Thus a measure is needed to identify the documents that are relevant to users query.

2.2.2 Similarity Measures

All current search engines use ranking to identify potential answers. In a ranked query , a statistical similarity measure is used to assess the closeness of each document to the textual query. The underlying principle is that the higher the similarity score awarded to a document, the greater the estimated likelihood that a human would judge it to be relevant. Most similarity measures use some composition of a small number of fundamental statistical values:

- $f_{d,t}$, the frequency of term t in document d;
- $f_{q,t}$, the frequency of term t in the query;
- f_t , the number of documents containing one or more occurrences of term t;
- F_t , the number of occurrences of term t in the collection;

- N , the number of documents in the collection;

- n , the number of indexed terms in the collection.

Following three monotonicity are enforced:

1. Less weight is given to terms that appear in many documents.

2. More weight is given to terms that appear many times in a document.

3. Less weight is given to documents that contain many terms.

The intention is to bias the score towards relevant documents by favoring terms that appear to be discriminatory and reducing the impact of terms that appear to be randomly distributed.

A simple and very common formula to calculate the similarity between a document d and the query q is given below:

$$w_{q,t} = \ln\left(1 + \frac{N}{f_t}\right) \quad w_{d,t} = \ln\left(1 + \frac{N}{f_{d,t}}\right)$$

$$W_d = \sqrt{\sum_t w_{d,t}^2} \quad W_q = \sqrt{\sum_t w_{q,t}^2}$$

$$S_{q,d} = \frac{\sum_t w_{d,t}.w_{q,t}}{W_d.W_q}$$

The quantity $w_{q,t}$ typically captures the property often described as the inverse document frequency of the term, or IDF, while $w_{d,t}$ captures the term frequency, or TF, hence the common description of similarity measures as TF x IDF formulations. An algorithm for exhaustive ranking using the similarity measure is shown below:

Algorithm 1 *To rank a document collection with regard to a query q and identity the top r matching documents:*

```
1: Calculate w_{q,t} for each query term t in q;
2: for each document d in the collection do
3:    Set S_d ← 0;
4:    for each query term t do
5:       Calculate or read w_{d,t}, and
6:       Set S_d ← S_d + w_{q,t} x w_{d,t};
7:    end for
8:    Set S_d ← S_d/W_d;
9:    Identify the r greatest S_d values and return the corresponding documents;
10: end for
```

Given a formulation, ranking a query against a collection of documents is in principle straightforward: each document is fetched in turn, and the similarity between it and the query calculated. The documents with the highest similarities can then be returned to the user.

2.2.3 Keyword Indexing

Fast query evaluation makes use of an index: a data structure that maps terms to the documents that contain them. For example, the index of a book maps a set of selected terms to page numbers. With an index, query processing can be restricted to documents that contain at least one of the query terms.

In applications involving document, the single most suitable structure is an inverted file [27]. An inverted file contains, for each term in the lexicon, an inverted list that stores a list of pointers to all occurrences of that term in the document, where each pointer is, in effect, the number of a document in which that term appears.

Baseline Inverted File.

An inverted file index consists two major components. The search keyword or vocabulary stores for each distinct word t,

- a count f_t of the documents containing t.

- a pointer to the start of the corresponding inverted list.

The second component of the index is a set of inverted lists where each list stores for the corresponding word t,

- the identifiers d of documents containing t, represented as ordinal document numbers.

- the associated set of frequencies $f_{d,t}$ of terms t in document d.

The lists are represented as sequences of $< d, f_{d,t} >$ pairs. There is a simple example on how to build inverted file index.

Document	Text
1	Material world is
2	Bounded by time and space
3	There is no time and space
4	In spiritual world

Table 2.1: Document set

Table 2.1 represents the document set. We build the corresponding word-level inverted file index as shown in Table 2.2. Together with an array of W_d values (stored separately as shown in Table 2.3), these components provide all the information required for both Boolean and ranked query evaluation.

Number	Term t	Times f_t	Inverted list for t
1	Bounded	1	$< 2, 1 >$
2	Material	1	$< 1, 1 >$
3	Space	2	$< 2, 1 > < 3, 1 >$
4	Spiritual	1	$< 4, 1 >$
5	Time	2	$< 2, 1 > < 3, 1 >$
6	World	2	$< 1, 1 > < 4, 1 >$

Table 2.2: Document level Inverted File

Notice that the word like *"By", "Is", "In", "No", "There"* are not considered as distinct vocabulary . They are considered as STOP WORDS. No inverted list is created for such type of stop words.

d	1	2	3	4
W_d	1.41	1.73	1.41	1.41

Table 2.3: Score of documents

Ranking using an inverted file is described in Algorithm 3. In this algorithm, the query terms are processed one at a time. Initially each document has a similarity of zero to the query; this is represented by creating an array A of N partial similarity scores referred to as accumulators, one for each document d. Then, for each term t, the accumulator A_d for each document d mentioned in ts inverted list is increased by the contribution of t to the similarity of d to the query. Once all query terms have been processed, similarity scores S_d are calculated by dividing each accumulator value by the corresponding value of W_d . Finally, the r largest similarities are identified, and the corresponding documents returned to the user.

Algorithm 2 *To use an inverted index to rank a document collection with regard to a query q and identify the top r matching documents:*

1: Allocate an accumulator A_d for each document d and set $A_d \leftarrow 0$;
2: **for each** query term t in q **do**
3: Calculate $w_{q,t}$, and fetch the inverted list for t;
4: **for each** pair $< d, f_{d,t} >$ in the inverted list **do**
5: Calculate $w_{d,t}$, and
6: Set $A_d \leftarrow A_d + w_{q,t}$ x $w_{d,t}$;
7: **end for**
8: **end for**
9: Read the array of W_d values;
10: **for each** $A_d > 0$ **do**
11: Set $S_d \leftarrow A_d/W_d$;
12: **end for**
13: Identify the r greatest S_d values and return the corresponding documents;

The cost of ranking via an index is far less than with the exhaustive Algorithm 2. Nonetheless, the costs are still significant. Disk space is required for the index at 20% $\sim$ 60% of the size of the data for an index of the type shown in Table 2.2; memory is required for an accumulator for each document and for

some or all of the vocabulary; CPU time is required for processing inverted lists and accumulators; and disk traffic is used to fetch inverted lists.

2.2.4 Hybrid Index Framework: The IR-Tree

The R-tree is the dominant index for spatial queries, and the inverted file is the most efficient index for text information retrieval. These were developed separately and for different kinds of queries. IR-tree utilizes both indexing structures in a combined fashion.

The IR-tree is essentially an R-tree, where each node of which is enriched with reference to an inverted file for the objects contained in the sub-tree rooted at the node. In the IR-tree, a leaf node N contains a number of entries of the form (O,$rectangle$,$O.\psi$), where O refers to an spatial object in database D, $rectangle$ is the bounding rectangle of object O, and $O.\psi$ is the k keyword of the spatial object O.

A non-leaf node R contains a number of entries of the form (N,$rectangle$,$N.\psi$) where N is the address of a child node of R, $rectangle$ is the Minimum Bounding Rectangle of all rectangles in entries of the child node, and $N.\psi$ is the identifier of an of an inverted list which is the union of all item in the lists of it's child nodes.

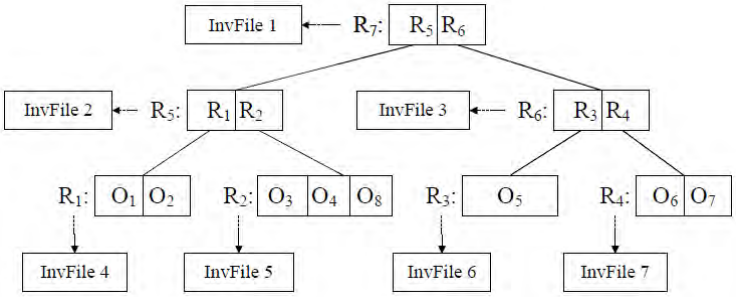


Figure 2.5: IR-tree.

In this chapter, we have discussed location indexing (R-tree), text indexing (*Inverted file*), hybrid indexing based on location and text (*IR-tree*). We have also highlighted spatial query, mainly Nearest Neighbor query and Range query. We have discussed the Similarity Measure that current search engine use to assess the closeness of each document to the textual query and inverted file.

Chapter 3

Related Work

In this chapter, we discuss the related works based on indexing technique and their query. Basing on the hybrid index structure, we classify the existing works into three categories.

1. Nearest Neighbor Queries.
2. Spatial Keyword Queries.
3. Tempo-Textual Queries.

3.1 Nearest Neighbor Queries

Searching for a nearby data object from a database based on the location has received considerable attention from database community. Existing technique for processing k queries assume that data objects are indexed, e.g. using an R-tree [13]. In order to find kNN from a query point, the tree can be traversed in a depth-first (DF) [18] or a best-first (BF) [15] manner. In the BF technique , the search starts from the root of the R-tree. Initially, all child nodes of the root nodes are stored in a priority queue. The entries in the priority queue are ordered based on the minimum distance between the query point and the minimum bounding rectangles (MBRs) of R nodes or data objects. In the next step, it removes the top element from the queue, which is the node representing the MBR or a data object with the minimum distance from the query point. If the removed element is a node, algorithm again inserts the child nodes or data objects of the removed node into the priority queue. On the other hand if the

dequeued item is a data object, then the corresponding object is reported as the next nearest neighbor. The above process continues until k data objects, i.e., k NNs are dequeued from the queue.

Finding a leaf node containing a query object q in a spatial index can be done in a depth first (DF) manner by recursively descending the tree structure [16]. With this method, the recursion stack keeps track of what nodes have yet to be visited. Having reached a leaf, it needs to extend this technique to find the nearest object, as the leaf may not actually contain the nearest neighbor. The problem DF technique is that it has to unwind the recursion to find the nearest object. Moreover, if it needs to find the second nearest object, the solution becomes even tougher.

3.2 Spatial Keyword Queries

There is more and more commercial and research interest in location-based web search, i.e. finding web content whose topic is related to a particular place or region. In this type of search, location information should be indexed as well as text information. However, the index of conventional text search engine is set-oriented, while location information is two-dimensional and in Euclidean space. This brings new research problems on how to efficiently represent the location attributes of web pages and how to combine two types of indexes and how to efficiently search location specific information. A straight forward approach is to treat textual information which represent spatial object as common keywords, and to retrieve spatial object in the same way to keyword matching. However, simple keyword matching neglects underlying spatial relationships.

Zhou *et al.* [26] tackled the problem of retrieving web documents relevant to a keyword query within a pre-specified spatial region. They proposed three approaches based on loose combination of of an *inverted files* and *R*-trees*. R* tree is another variant of R-tree. Inverted file is a structure to index keywords of a document which we have described elaborately in Chapter 2. They studied three different combining schemes:

1. Inverted file and R*-tree double index.

2. First inverted file then R*-tree.

3. First R*-tree then inverted file.

In the first scheme web pages are indexed separately twice, once by R*-tree and once by inverted files. All MBRs are indexed by an R*-tree. The difference from conventional R*-tree is that each leaf node of the MBR tree points to a page list whose scope includes this MBR, as shown in Figure 3.1. Inverted files are the same to conventional search engines. Thus it maintains two kinds of page lists whose entry is either an MBR or a keyword.

In the second scheme, first an inverted index file is built. Then, the file is modified by building a R*-tree to the set of objects MBR pointed to by each keyword in the file. The leaf node of R*-tree points to a page list of object ids whose entry contain the keyword and the MBR. This entry is called a geo-keyword. When a query is issued, the query keywords are filtered using the inverted index. Later, the R*-tree corresponding to each query keyword are used to filter the spatial part of the query. The intersection of object ids from the R*-trees produces the final answer set. This approach is highly insensitive to Spatial Keyword queries with the AND semantics as this approach does not take advantage of the association of keywords in space.

In third scheme, as shown in Figure 3.2b, an R*-tree is first built for all the objects MBR irrespective of keywords. An inverted index file is created for keywords that appear in the leaf node of the tree. Each keyword in this inverted index points to a page list of object ids whose entries contain both the MBR and the keyword, again referred to as geokeyword. When a query is issued, a set of leaf nodes that intersect with the query rectangle is retrieved first. Then using the retrieved nodes inverted file index, object ids satisfying the query keywords are obtained. In this approach, the leaf nodes point to inverted index lists that are usually small and causes generating of many candidate object ids during spatial filtering stage.

Zhou's' hybrid index structure prune the search space by spatial pruning and textual pruning separately and retrieve relevant documents within a given geographic region. Hariharan *et al.* [14] proposed an improved hybrid index structure called Keyword R (KR^*) tree. KR^* tree consist of an R*-tree and an inverted file for the nodes of the R*-tree. The nodes of KR*-tree are virtually augmented with the sets of keywords that appear in the subtrees rooted at that nodes. Thus, many keywords appear in the upper level nodes of the

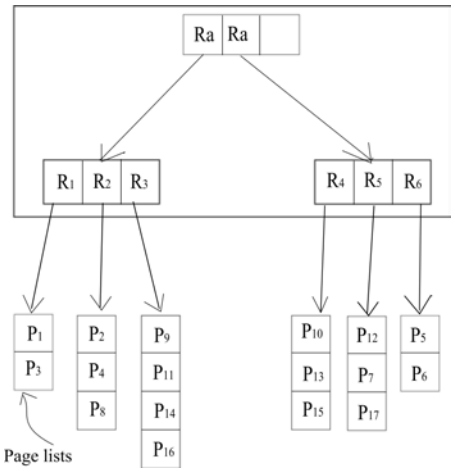
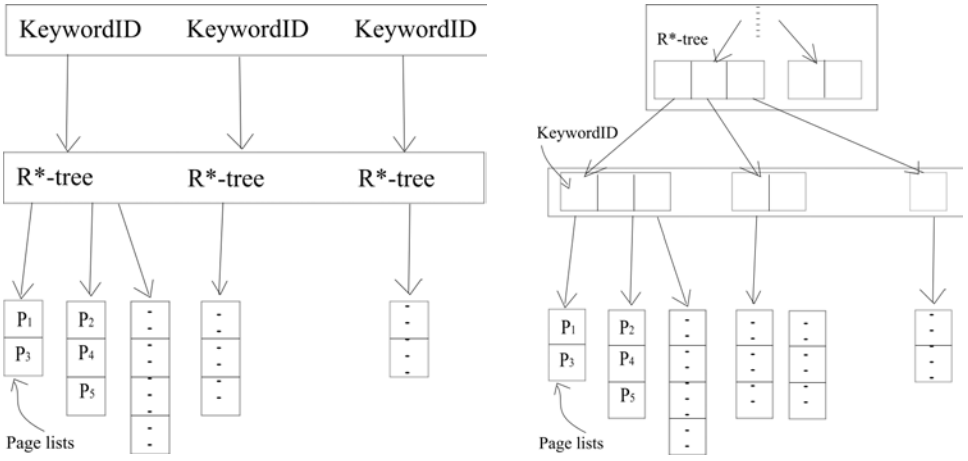


Figure 3.1: Inverted File and R^* tree double index



(a) Inverted File-R*-tree (b) R*-tree Inverted File

Figure 3.2: Zhou's index structure

tree and smaller number of keywords appear in the lower level nodes of the tree. At query time, the KR*-tree based algorithm finds the nodes that contain the query keywords and then uses these as candidates for subsequent search. This approach suffers from unnecessary overhead when there are many candidates. KR^* tree only good for spatial objects with a small number of keywords. Though his keyword query works on integrated manner but his work does not support nearest neighbor queries.

Consider a simple emergency GIS database presented in Table 3.1. The database has 10 objects whose spatial distribution and the corresponding KR*-tree nodes are shown in Figure 3.3a. Now, let us consider an SK query: $<$

OID	X	Y	keywords
1	x_1	y_1	{fire,medical}
2	x_2	y_2	{earthquake,medical}
3	x_3	y_3	{fire,medical}
4	x_4	y_4	{facility,medical}
5	x_5	y_5	{earthquake,hazard}
6	x_6	y_6	{earthquake,medical}
7	x_7	y_7	{fire,facility}
8	x_8	y_8	{earthquake,shelter}
9	x_9	y_9	{fire,shelter}
10	x_{10}	y_{10}	{earthquake,shelter}

Table 3.1: Example GIS database

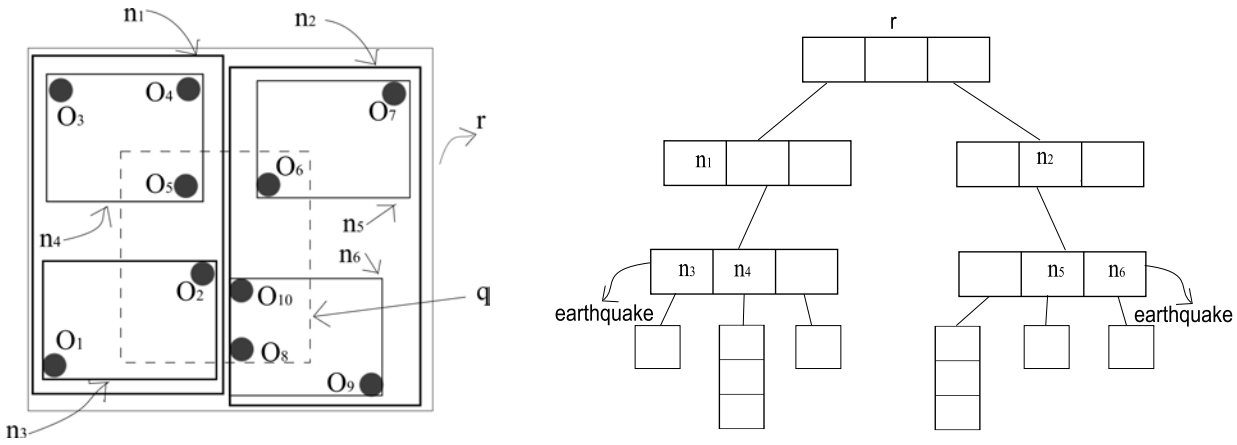
Node	keywords
r	{earthquake, medical, shelter,fire,facility, hazard}
n_1	{fire, medical, earthquake, facility, hazard}
n_2	{fire, medical, earthquake,shelter,facility}
n_3	{fire, medical,earthquake}
n_4	{fire, medical,earthquake, facility }
n_5	{medical, fire, earthquake,facility}
n_6	{earthquake,shelter,fire}

Table 3.2: Distribution of keywords in space

$q, \{earthquake, shelter >$ that asks for *earthquake shelter* in spatial region q. In order to answer this query KR^* tree performs better than previous three index structure.

Using R*-tree Inverted file: For the same example query, the R*-tree nodes $r, n_1, n_2, n_3, n_4, n_5, n_6$ are accessed to generate the candidate object set. Next, the inverted index list of leaf nodes n_3, n_4, n_5, n_6 are accessed to filter objects that contain the keywords earthquake and shelter to finally arrive at the answer set o_8, o_{10}.

Using Inverted file R*-tree: For the example query, first the R*-trees of the keywords earthquake and shelter are loaded to get the candidate object id lists.



(a) Objects Distributed in Space

(b) KR*-tree

Figure 3.3: Space diagram and KR* tree

Then, the intersection of the two candidate object id lists gives the answer set o_8, o_{10}.

Using KR*-tree: Table 3.2 shows the keywords that appear in space covered by all nodes of the KR^* tree. For the given query, the root is first accessed to get its children n_1 and n_2. Now, for each child node, its spatial intersection with the query region and the presence of query keywords in it are checked. In this case, n_1 does not contain both the keywords, but n_2 does, hence only node n_2 is opened. Applying the same principle for n_2ś children, we see that only n_6 satisfies the SK query. Hence only {r, n_2, n_6} are accessed to generate the candidate object set.

Yiu et al. [23] worked on spatial preference query that ranks objects based on the qualities of features in their spatial neighborhood and retrieves the spatial objects basing on their rank. In his work, spatial data are ranked by the user with respect to the appropriateness of their location, the qualities of other features (e.g., restaurants, cafes, hospital, market, etc.) within a distance range from them. However, here he did not consider the keywords to search those spatial objects.

Many applications require finding objects closest to a specified location that contains a set of keywords. For example, online yellow pages allow users to specify an address and a set of keywords. In return, the user obtains a list of businesses whose description contains these keywords, ordered by their distance from the specified address. This kind of application require spatial indexing

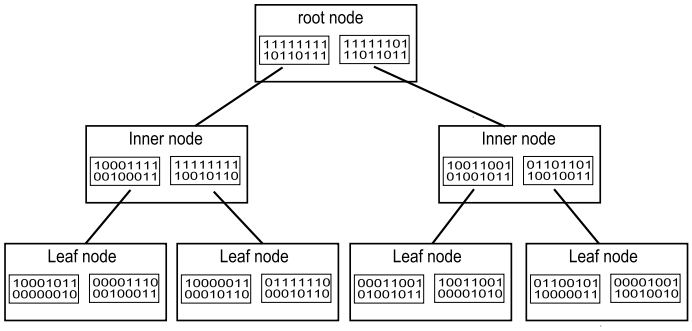


Figure 3.4: IR^2 tree

for finding the nearest object from query's precise location (point location) and keyword indexing to get keyword as an input to search the spatial object by it's description (textual information). Felipe for the first time combined these index in a single structure called Information Retrieval R (IR^2) tree [12]. IR^2 tree is an R-tree where a *signature* [11] is added instead of inverted file to each node of the IR^2 tree to denote the textual content of all spatial objects in the subtree rooted at that node.

A signature file is a technique applied to document retrieval. In this technique, the documents are stored sequentially in the "text file" and their abstraction are stored sequentially in the "signature file". The signature of a node of IR^2 tree (Figure 3.4) is the superimposition (OR-ing) of all the signatures of its entries. Thus a signature of a node is equivalent to a signature for all the documents in its subtree. When a query arrives, the signature file is scanned sequentially, and a large number of non qualifying documents are discarded.

A drawback of the IR^2-tree is that the same signature length is used for all levels which leads to more false positives in the higher levels, that have more 1s (since they are the superimpositions of the lower levels). To address this problem, they use varying signature lengths for different levels. This is achieved using multi-level superimposed coding, which reduces the number of false positives, particularly in non-leaf nodes. In this case, they use the optimal signature length for each level and superimpose the signatures of all objects in the subtree of each node, instead of the signatures of the children nodes as before. This variant is called Multi-level IR^2-tree (MIR^2 tree). A drawback of this MIR^2 tree, is that it significantly increases the complexity of the tree maintenance operations (Insert and Delete) since for each object inserted or

deleted, it need to recompute the signatures of all ancestor nodes by accessing all underlying objects and not just by superimposing the childrens signatures as before. In his proposed index structure performance get worst when the number of keyword increases. At times, signature files are not able to eliminate the objects not satisfying the query keywords (false hits). This result in loading and reading more objects, which is costly. However their work support nearest neighbor queries by keyword search for the first time in hybrid indexing field.

Work discussed so far is based on spatio object retrieval by keyword matching. In many cases user may interest on finding spatial objects that is textually relevant to user's search keyword. In this case, a new method is required to rank an object how much relevant they are to user's search keyword. Cong proposed a new kind of ranking query which is called location aware top-k text retrieval (LkT) query [10]. Their index approach called the Inverted File R (IR) tree which is essentially an R-tree extended with inverted files. Their query processing algorithm utilizes the location index information to estimate the spatial distance of a query to the objects in the nodes sub-tree, and it uses the text index to estimate the text relevancy scores for these objects. Thus their query returns objects ranked according to a linear interpolation function that combines normalized location proximity and text relevancy. If D be a spatial database and each spatial object S in D is defined as pair ($S.loc$, $S.doc$), where $S.loc$ is a location descriptor in multidimensional space and $S.doc$ is a document (e.g., a dining menu) that describes the object (e.g., an Italian restaurant). Document $S.doc$ is represented by a vector in which each dimension corresponds to a distinct term in the document. The value of a term in the vector is computed by a language model [22, 24] as follows:

$$\hat{p}(t \mid \theta_{S.doc}) = (1-\lambda)\frac{tf(t, S.doc)}{|S.doc|} + \lambda\frac{tf(t, Coll)}{|Coll|}$$

where *tf(t,S.doc)* is the number of occurrences of term t in document $S.doc$ and *tf(t,Coll)* is the count of term t in the document collection $Coll$ of D; *tf(t,S.doc)*/$|S.doc|$ is the maximum likelihood estimate of term t in document $S.doc$ and *tf(t,Coll)*/$|Coll|$ is the maximum likelihood estimate of term t in collection $Coll$; λ is a smoothing parameter of the Jelinek-Mercer smoothing method. Given a query Q and a document $S.doc$, the ranking function for the query likelihood language model is as follows:

$$P(Q.keywords|S.doc) = \prod_{t \in Q.keywords} \hat{p}(t \mid \theta_{S.doc})$$

To rank an spatial object S with respect to query object Q, they derived a ranking function as a linear interpolation of normalized factors.

$$D_{ST}(Q, O) = \lambda \frac{D_e(Q.loc, S.loc)}{maxD} + (1 - \lambda)(1 - \frac{P(Q.keywords|S.doc)}{maxP})$$

where $\lambda \in (0, 1)$ is a parameter used to balance spatial proximity and text relevancy; the Euclidian distance between Q and S, $D_e(Q.loc, S.loc)$, is normalized by $maxD$, which is the the maximum distance between two objects in D; and $maxP$ is used to normalize the probability score into the range from 0 to 1 and is computed by:

$$\prod_{t \in Q.keywords} \max_{S' \in D} \hat{p}(t|S'.doc)$$

This ranking function is used when their query algorithm reaches leaf nodes. During visiting the internal node of IR tree they use the following ranking function:

$$MIND_{ST}(Q, N) = \lambda \frac{MIND_e(Q.loc, N.mbr)}{maxD} + (1-\lambda)(1-\frac{P(Q.keywords|N.doc)}{maxP})$$

Here $MIND_{ST}(Q, N)$ is the minimum Euclidian distance between Q.loc and $N.mbr$. $N.doc$ is the pseudo document on node N and,

$$P(Q.keywords|N.doc) = \prod_{t \in Q.keywords} \hat{p}(t \mid \theta_{N.doc})$$

They also proposed another variant of IR tree called DIR tree . While IR tree considers only location information when generating it's MBRs, the DIR tree considers both location information and document similarity. LkT query is able to prune the search space by simultaneous use of both spatial proximity and text relevancy and result in retrieval of kNN spatial objects.

There are some major problems with IR tree. First, one inverted file needs to be stored. For web, total number of documents and keywords are very large, that results huge number of nodes in R-tree and also large inverted file

Location	Documents		keywords
		Soccer	League
Redonda beach	d_1	5	4
Culver city	d_2	3	0
Palma	d_3	1	1
Santa Monica	d_4	1	1
Downtown	d_5	2	2
Pasedena	d_6	0	2

Table 3.3: Documents describing the location

in each node. Addressing this problem, Khodaeil *et al.* [20] work on web pages containing documents that describe specific object for a specific location. They present a ranking method that considers both the spatial overlap of a document with a query and the frequency of the query keywords in the document in order to compute relevance score of the document to the query. To measure the textual relevancy they use a similarity measure (*tf-idf*) .

In most keyword queries, a similarity measure is determined by using the following important parameters:

where

- $f_{d,k}$, the frequency of keyword k in document d;
- $maxf_{d,k}$, the maximum value of $f_{d,k}$ over all the keywords in document d;
- $f_{d,k}$, normalized of $f_{d,k}$, which is $\frac{f_{d,k}}{max(f_{d,k})}$;
- f_k , the number of documents containing one or more occurrences of keyword k;

Using these values, three monotonicity observations are enforced (1) less weight is given to the terms that appear in many documents; (2) more weight is given to the terms that appear many times in a document; and (3) less weight is given to the documents that contain many terms. The first property is quantified by measuring the inverse of frequency of keyword k among the documents in the collection. This factor is called inverse document frequency

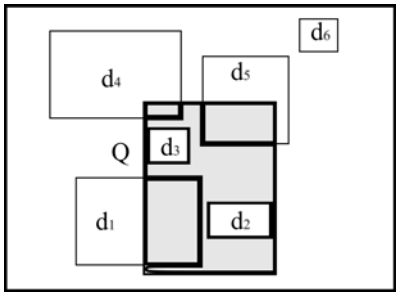
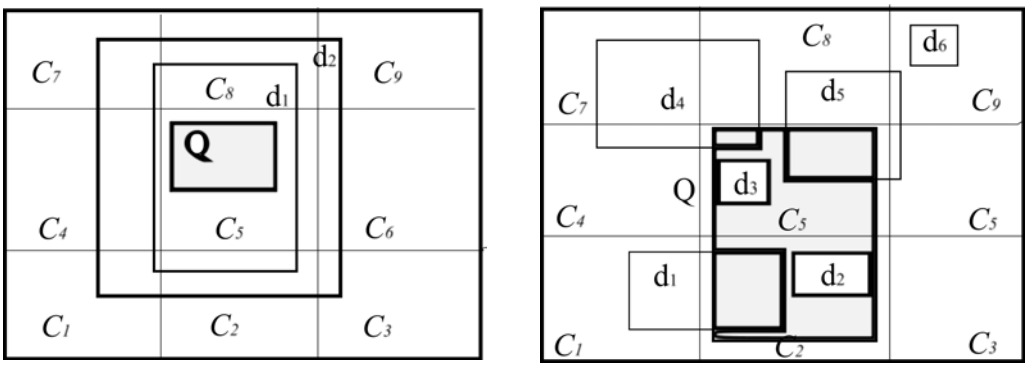


Figure 3.5: A spatial keyword query on documents with location information



(a) Property (2) and (3)

(b) Example of SK query considering cell

Figure 3.6: Space partitioned into grid cells

or the idf score. The second property is quantified by the raw frequency of keyword k inside a document d. This is called term frequency or tf score, and it describes how well that keyword describes the contents of the document. The third property is quantified by measuring the total number of keywords in the document. This factor is called document length.

In order to be able to use the analogous ideas used in the regular tf-idf score, they treat spatial data similar to textual data. For this, they partition the space into grid cells and assign unique identifiers to each cell. Therefore, each location in document can be associated with a set of cell identifiers. These cells are defined as the cells which overlap with the document location. With spatial tf-idf, the overlap of a cell with the document is analogous to the existence of a keyword in document with tf-idf. They use the overlap area between each cell and the document to provide a measure of how well that cell describes the document. Analogous to frequency of term t in document d, they define frequency of cell c in document d as $f_{d,c} = \frac{L_d \cap c}{c}$, which is the area of overlap

between the document location L_d and cell c divided by the area of cell c. Analogous to textual relevancy following parameter can be defined:

- $f_{d,c}$, the frequency of cell c in document d;
- $maxf_{d,c}$, the maximum value of $f_{d,c}$ over all the cells in document d;
- $\overline{f_{d,c}}$, normalized of $f_{d,c}$, which is $\frac{f_{d,c}}{max(f_{d,c})}$;
- f_c , the number of documents containing one or more occurrences of cell c;

So (1) less weight is given to cells that appear in many documents; (2) more weight is given to cells that overlap largely with a document; and (3) less weight is given to documents that contain many cells. For example in Figure 3.6a, cell C_9 better describes the document d_2 than C_3 as it largely overlaps document d_2. As per property 3, document d_1 should be weighted more than d_2 as d_1 contain less cells. Cell C_8 should be weighted less than cell C_9 as per property 1 as C_8 is appeared in document d_2 only. By combining these two relevancy they propose new index structure called spatial keyword inverted file (*SKIF*) and treat the spatial data similar to textual data and thus avoid using *R*-tree. *SKIF* structure is capable of indexing and searching both textual and spatial data in a similar and integrated manner. Their work did not consider *NN* queries.

LkT query returns ranked objects that are near to a query location and their textual descriptions match query keywords. However, it is found that a relevant object with nearby objects that are also relevant to the query is likely to be preferable over a relevant object without relevant nearby objects. Based on this findings, a new type of query, the Location-aware top-k Prestige-based Text retrieval *(LkPT)* query is proposed [7] that retrieves the top-k spatial web objects ranked according to both prestige-based relevance and location proximity. LkPT query takes into account both location proximity and prestige based text relevance (PR). PR score of an object is affected by the PR scores of its neighbors.

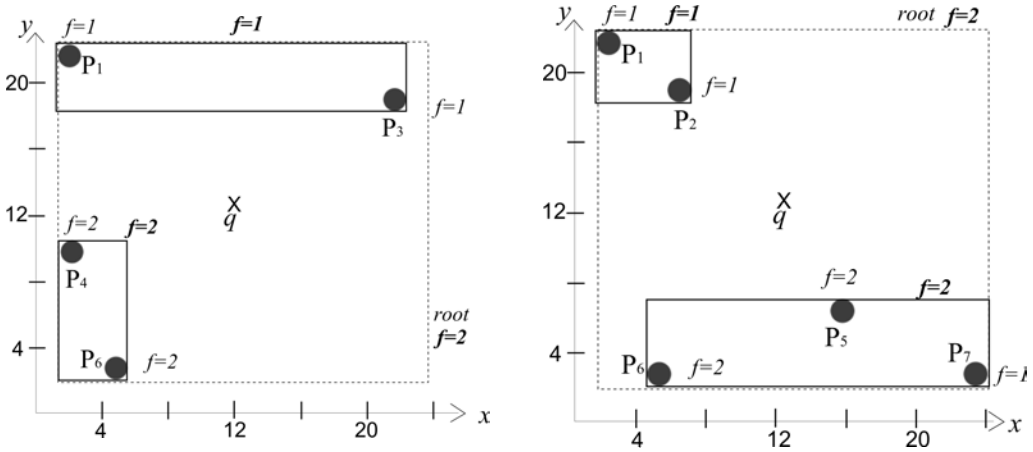
Hybrid index([7], [10], [14]) discussed so far augments the nodes of an R-tree with inverted indexes. The inverted index at each node refers to a pseudo-document that represents all the objects under the node. Therefore, in order to verify if a node is relevant for a set of query keywords, the current approaches

Object	keywords
P_1	{bar, pub, pop, pop}
P_2	{pub, rock}
P_3	{bar, rock }
P_4	{bar, bar, samba}
P_5	{pub, samba }
P_6	{pub, bar, pub, bar}
P_7	{pub, samba}

Table 3.4: Keywords describing the objects

Term	id	df_t	type	ptr	storage
bar	t_1	4	tree	$\hookrightarrow$	aR^{t_1}
pop	t_2	2	block	$\hookrightarrow$	$\langle P_1, P_5 \rangle$
pub	t_3	5	tree	$\hookrightarrow$	aR^{t_3}
rock	t_4	2	block	$\hookrightarrow$	$\langle P_2, P_3 \rangle$
samba	t_5	2	block	$\hookrightarrow$	$\langle P_4, P_7 \rangle$

Table 3.5: S2I



(a) aR^{t_1} (b) aR^{t_3}

Figure 3.7: Spatial Inverted index and the aR tree of terms t_1 and t_3

access the inverted index at each node to evaluate the similarity between the query keywords and the pseudo-document associated with the node. This process incurs in non-negligible processing cost that results in long response time. To increase the performance of top-k spatial keyword queries Rocha-Junior *et al.* [17] proposed a novel hybrid index called spatial inverted index (*S2I*) instead of employing single R-tree embedded with inverted indexes. *S2I* maps each keyword (term) to a distinct aggregated R-tree *(aR-tree)* that stores the objects with the given term. The *S2I* maps each term t to an *aR-tree* or to a block that stores the spatio-textual objects p that contain t. The most frequent terms are stored in *aR-trees*, one tree per term. The less frequent terms are stored in blocks in a file, one block per term. Similarly to a traditional inverted index, the *S2I* maps terms to objects that contain the term. Thus *S2I* consists of three components: vocabulary, blocks, and trees as shown in Table 3.5.

Vocabulary. The vocabulary stores, for each distance terms the number of objects in which the term appears (df_t), a flag indicating the type of storage used by the term (block or tree), and a pointer to a block or *aR-tree* that stores the objects containing the given term.

Blocks. Each block stores a set of objects, the size of a block is an application parameter. For each object, object identification $p.id$, the object location $p.l$, and the impact of term t in textual document $p.d$ ($\lambda_{t,p.d}$).

Trees. The aggregated R-tree of a term aR^t (Figure 3.7a,3.7b) follows the same structure of a traditional R-tree. A leaf-node stores information about the data objects: $p.id$, $p.l$, and $\lambda_{t,p.d}$. An intermediary-node stores for each entry (child node) a minimum bounding rectangle (MBR) that encloses the spatial location of all objects in the sub-tree. Differently from an R-tree, the nodes of an *aR-tree* store also an aggregated non-spatial value among the objects in its sub-tree. In this case, the aggregated value is the maximum impact of the term t among the objects in the sub-tree of the node.

In many cases user's need may not be easily satisfied by a single objects, but a group of objects can collectively better satisfy an user need. Considering this scenario Cao *et al.* [8] again used the IR-tree. But here they worked on retrieving group of object having lowest inter object distances between them and are close to query point. Their method collectively meet the query keywords. If D be a database consisting of m spatial objects then a spatial group keyword query $q = \langle q.\lambda, q.\psi \rangle$ where $q.\lambda$ is a location and $q.\psi$ represents a set of keywords will retrieve a group of objects χ, $\chi \subseteq D$, such that $\cup_{r \in \chi} r.\psi \supseteq q.\psi$ and such that the cost Cost(χ) is minimized. Their cost function has two weighted function as follows:

$$Cost(q, \chi) = \alpha C_1(q, \chi) + (1 - \alpha) C_2(\chi)$$

where $C_1(.)$ is dependent on the distance of the objects in χ to the query object and $C_2(.)$ characterizes the inter-object distances among the objects in χ. α is the parameter that is used to give preference on C_1 and C_2. Most existing works on spatial keyword queries retrieve single objects that are close to the query point and are relevant to the query keywords. In contrast, her work retrieve groups of objects that are close to the query point and collectively meet the keywords requirement.

The proliferation of geo-social network, such as Foursquare and Facebook

Places, enables users to generate location information and its corresponding descriptive tags. Using geo-social networks, users with similar interests can plan for social activities collaboratively. Zhang et al. [25] proposes a novel type of query, called tag-based top-k collaborative spatial (*TkCoS*) query, for users to make outdoor plans collaboratively. This type of queries retrieve groups of geographic objects that can satisfy a group of users requirements expressed in tags, while ensuring that the objects be within the minimum spatial distance from the users. To answer TkCoS queries efficiently, an hybrid index structure called spatial-tag R-tree (*STR-tree*) is proposed, which is an extension of the R-tree.

Many location-based service (LBS) users have direction-aware search requirement. For example, a user on the highway wants to find nearest gas stations or restaurants. She has a search requirement that the answers should be in the right front of her driving direction, if in a right-hand traffic country. To address this issue, Li and Xu work on direction-aware spatial keyword queries [21] called Desks and proposed direction-aware index structure to prune unnecessary directions.

3.3 Temporal and Textual Queries

Modern text analytics applications operate on large volumes of temporal text data such as Web archives, newspaper archives, blogs, wikis, and micro-blogs. In these settings, searching and mining needs to use constraints on the time dimension in addition to keyword constraints. Anand et al. [5] worked on Web archives, newspaper archives, blogs, wikis, and micro-blogs which consists of large volumes of temporal text data. Their queries that combine the content and temporal predicates are termed as time-travel queries. For efficient queries, they temporally partitioned inverted index which essentially meant that the time enriched inverted indexes are sliced along the time-axis (or partitioned vertically). Such index organization suffers from an index-size blowup incurred during the slicing process. Considering this later they proposed to shard or horizontally partition each index list along document identifiers, instead of time [6].

Again Khodaeil and Shahabi work on web pages [19]. But this time they considered handling the time with document's textual information. To index

temporal and textual attributes of document he proposed tempo-textual inverted index (T^2I^2). T^2I^2 index structure is capable of indexing and searching both textual and temporal data in a similar and integrated manner and results in retrieval of top-k document based on temporal and textual relevance score. Here, they did not consider location indexing.

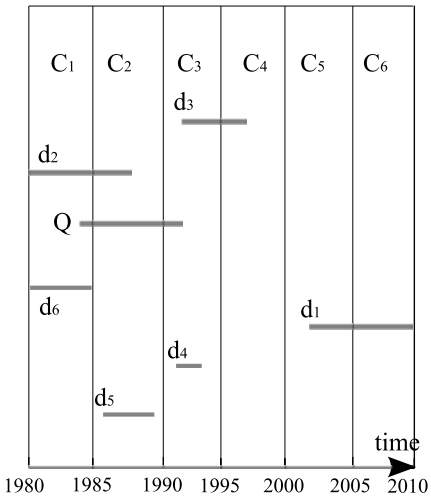


Figure 3.8: Temporal cells

In order to use the analogous ideas of regular tf-idf score, they treat temporal data similar to textual data. Hence they partition the time domain into consecutive cells and assign unique identifiers to each cell (Figure 3.8). Therefore, each time span in the document can be associated with a set of cell identifiers. These cells are defined as the cells which overlap with the document time span. With temporal tf-idf, the overlap of a cell with the document is analogous to the existence of a keyword in the document with tf-idf. These cell describes the temporal content of the document that comes from following three properties.

1. Less weight is given to cells that appear in many documents.

2. More weight is given to cells that overlap largely with a document.

3. Less weight is given to documents that contain many cells.

Thus in Figure 3.8 we find that cell C_6 better describes document d_1 than cell C_5 as per property 2. Document d_1 should be weighted less than document d_5 as per property 3. Cell C_3 should be weighted less than cell C_5 as per property 1.

Finally Chen et al. [9] provides an all around survey of hybrid indexing that comprised of location and keyword indexing only. So far we have discussed the related works on spatial keyword queries, most of them is based on location and keyword[6,7,9,11,13,17,20,21,25,26] and very few works[4,5,19] based on text and time. We focus on spatio-temporal keyword search, where we need to incorporate time in tandem with location and keywords and to the best of our knowledge no work has been done so far on our proposed area.

In this chapter, we have discussed current works related to hybrid index structure that mostly based on location indexing and keyword indexing and their queries. We also have highlighted few works on hybrid index structure based on text indexing and time indexing. Finally, we have mentioned our proposed index structure that based on location index, keyword index, and time index.

Chapter 4

Problem Definition

In this chapter, we first formulate our problem based on a real world application.

Let us consider our future marketing based location based services. The application hosts a list of sellable items/objects described by their locations, descriptions, and the time of item availability from sellers. A buyer submits an STK-kNN query represented as buyer's location, item description, and desired time when the buyer wants the item.

Seller	X	Y	Item	From	To
S_1	5	9	Potato, Onion	07/06/14 0900hrs	07/06/14 1500hrs
S_2	9	6	Onion,Garlic	12/06/14 1200hrs	12/06/14 1800hrs
S_3	2	17	Potato,Garlic	14/08/14 0600hrs	14/08/14 1200hrs
S_4	17	16	Onion,Potato,Garlic	30/06/14 0600hrs	12/07/14 0800hrs
S_5	23	21	Apple,Onion	14/08/14 0900hrs	14/08/14 1200hrs
S_6	22	11	Lemon,Cucumber	01/07/14 1500hrs	01/07/14 1800hrs
S_7	17	5	Onion,Potato	02/06/14 0600hrs	29/06/14 1800hrs
S_8	23	3	Cucumber,Potato	07/06/14 0900hrs	31/07/14 2100hrs
S_9	5	14	Garlic,Potato	22/08/14 0900hrs	15/09/14 0900hrs

Table 4.1: Sample data set of seller object

Table 4.1 shows location, textual description (keyword), and temporal description for a set S of seller objects as $\{S_1, S_2, ..., S_n\}$ in a spatial database D. Object S_i is defined as triple $(S_i.\lambda, S_i.\psi, S_i.t)$ where $S_i.\lambda$ denotes location, $S_i.\psi$ is a set of keywords, and $S_i.t$ is time stamp of the object. Similarly we can represent a STK-kNN query Q as $(Q.\lambda, Q.\psi, Q.t)$. Once buyer submit a STK-kNN query to location based service provider, it returns k nearest objects (with respect to buyer's location) that match the query's keyword and time.

In this paper, we develop an efficient technique to answer STK-kNN queries.

Moreover, since sellers may not certain about the item availability, time range given by the seller needs to be considered as uncertain data, where the item will be available at any point of time in the given time range. In such cases, the answer of an STK-kNN query will have a probability value assigned to the returned object. For example, a seller S_1 publishes an item as *"Lichi will available anytime between 03 Feb 2014 and 10 Feb 2014"*. Another seller S_2 with more confidence publishes his item as *"Lichi will be available between 03 Feb 2014 and 05 Feb 2014"*. Now if a buyer submit a query like *"Find nearest seller location having Lichi on 01 Feb 2014"*, then an STK-kNN query retrieves the k nearest objects each having with probability value. This probability value determines how reliable and relevant they are to the query's time. In this case, seller S_2 has more probability value than seller S_1 as S_2 predicts the item availability in a short time span. In this paper, we extend the STK-kNN query to handle time uncertainty.

Chapter 5

Our proposed STIR-tree

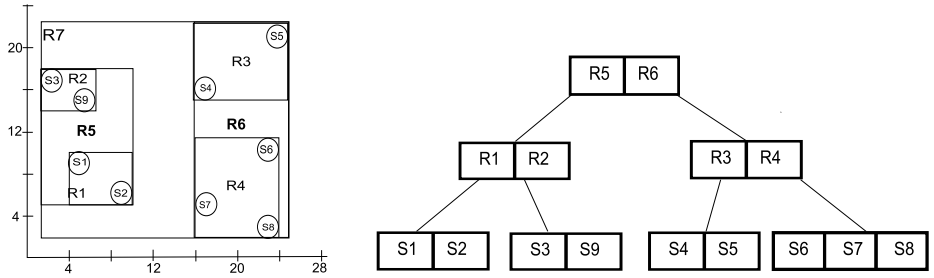
In this chapter, we first discuss location indexing and keyword indexing based spatial information and text information on Table 4.1 of Chapter 4. Then discuss our time indexing method and how to integrate time indexing with existing hybrid index structure based on location and keyword. We call our proposed hybrid index structure as STIR-tree.

5.1 Indexing Location

For location indexing we use the R-tree [13]. Every object in the R-tree is represented by a minimum bounding d dimensional rectangle (MBR). Data objects are grouped into larger MBRs forming the leaf nodes of the tree. Leaf nodes are grouped into larger internal node or non leaf node. The process continues recursively until the last group of nodes form the root of the tree. The root represents the MBR that encloses all objects and nodes indexed by the tree and each node corresponds to the MBR that bounds it's children. Figure 5.1(a) shows point location of the spatial objects taken from Table 4.1 and their bounding rectangle and Figure 5.1(b) shows corresponding R-tree.

5.2 Indexing Keywords

A keyword index is a mechanism for locating a given term in a document. In applications involving document, the most suitable structure is an inverted file [27]. Inverted file consist of lists, one per keyword (K), recording the iden-



(a) Objects and their bounding rectangles (b) R-tree formed by location of the objects

Figure 5.1: Our location indexing

tifier of the documents containing the keywords. This file consists of two major parts.

- Vocabulary list. The vocabulary list is a distinct keywords that describe the object. Vocabulary list can also be taken as keyword list.
- Posting list. A posting list for each distinct keyword K which is a identifier of the object that contain the respective keyword.

In our application since list of items (Column 4) describe their respective entity as shown in Table 4.1, our inverted file contains for each item K an inverted list as a sequence of $< l, f_{l,K} >$. Here l is the list identifier and $f_{l,K}$ is the frequency of the item that appears in that list. In the leaf node of R-tree, we index the keyword or item by list identifier and in the non leaf node we index the item by an identifier which is the union of all item in the lists of it's child node. Table 5.1 shows the inverted list of leaf nodes R_1, R_2, R_3, and R_4 and Table 5.2 shows the inverted list of non leaf nodes R_5, R_6, and R_7. For example, item Onion available at entity S_1, S_2, S_4, S_5, and S_7. Since leaf node R_1 bounds S_1 and S_2, R_3 bounds S_4 and S_5 and R_4 bounds S_6, S_7, and S_8, so the inverted list of Onion for leaf node R_4 is $< S_7, 1 >$. Here S_7 is the identity where Onion is available and 1 is the frequency that the item appears in the object. In this case, Onion appears once in S_7. Though R_3 bounds S_6 and S_8 but they are not listed in the inverted list as Onion is not available at S_6 and S_8. Similarly inverted list of Onion for leaf node R_3 is $< S_4, 1 >, < S_5, 1 >$.

Non leaf node R_5 and R_6 bounds leaf nodes R_1, R_2 and R_3, R_4 respectively. So the inverted list of Onion for R_6 is $< R_3, 2 >, < R_4, 1 >$. As inverted

R_4	R_3	R_2	R_1
Potato: $< S_7, 1 >, < S_8, 1 >$	Potato: $< S_4, 1 >$	Potato: $< S_3, 1 >, < S_9, 1 >$	Potato: $< S_1, 1 >$
Onion:$< S_7, 1 >$	Onion:$< S_4, 1 >, < S_5, 1 >$	Garlic:$< S_3, 1 >, < S_9, 1 >$	Onion:$< S_1, 1 >, < S_2, 1 >$
Lemon:$< S_6, 1 >$	Garlic:$< S_4, 1 >$		Garlic:$< S_2, 1 >$
Cucumber:$< S_6, 1 >, < S_8, 1 >$	Apple:$< S_5, 1 >$		

Table 5.1: Inverted list for leaf nodes R_1, R_2, R_3, and R_4

$Root(R_7)$	R_6	R_5
Potato: $< R_5, 2 >, < R_6, 2 >$	Potato: $< R_3, 1 >, < R_4, 2 >$	Potato: $< R_1, 1 >, < R_2, 2 >$
Onion:$< R_5, 1 >, < R_6, 2 >$	Onion:$< R_3, 2 >, < R_4, 1 >$	Onion:$< R_1, 2 >$
Garlic:$< R_5, 2 >, < R_6, 1 >$	Garlic:$< R_3, 1 >$	Garlic:$< R_1, 1 >, < R_2, 2 >$
Apple:$< R_6, 1 >$	Apple:$< R_3, 1 >$	
Lemon:$< R_6, 1 >$	Lemon:$< R_4, 1 >$	
Cucumber:$< R_6, 1 >$	Cucumber:$< R_4, 1 >$	

Table 5.2: Inverted list for non leaf nodes R_5, R_6, and R_7

list for Onion appears twice in leaf node R_3 and once in leaf node R_4, so the number 2 and 1 appears in R_6's inverted list. Similarly inverted list for non leaf node R_5 is $< R_1, 2 >$. Likewise inverted list of Onion for root node R_7 is $< R_5, 1 >, < R_6, 2 >$.

5.3 Time Indexing

In this section, we have developed a methodology that allows us to organize time hierarchically in the R-tree in tandem with location and keywords. The intuition behind our index is to exploit the natural time-granularity, i.e., *hour*, *day*, etc., and embed them in the hierarchical tree from leaf to root in order of fine (e.g., hour) to coarse (e.g., month) granularity of time.

In our approach, we model the time in 3 different forms, i.e. *Hour*, *Day*, and *Month*. The fine granular form is "hour" and the coarse granular form is "month". We further represent each hour, day, and month type of time-form into 24, 31, and 12 fixed cells, respectively, as shown in Figure 5.2. Thus, each cell of a day points to cells of hour of that particular day. Similarly, each cell of a month points to cells of day of that particular month. We embed the time in our index in such a way that leaf node contains the fine granularity of time and root node contains the coarse granularity of summarized time of all its child nodes.

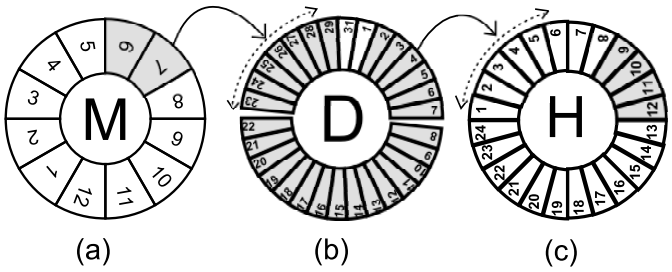


Figure 5.2: Time indexing (a)Month (b)Day (c)Hour

Let us explain the concept with our running example. In our example, Table 4.1 contains spatio-temporal objects S_5, S_6, S_7, and S_8. We see from the entry S_5 that onion and apple are available on 14 Aug 2014 from 0900 hours upto 1200 hours. So we can represent the time dimension of S_5 using the hour-type form, i.e., we set cell entries 9, 10, 11, and 12 in an hour-type time form. Similarly we can set entity S_6's time dimension using an hour-type form. On the other hand, the time dimension of entry S_7, from 02 Jun 2014, 0600 hours to 29 Jun 2014, 1800 hours, needs to be represented using a day-type time form. Here we set cells from 2 to 29 in day-type time form for representing S_7 time dimension. Similarly, the time dimension of entry S_8 ranges from 07 Jun 2014, 0900 hours to 31 Jul 2014, 2100 hours. Thus, we need to use the month-type time form to represent the time range, where we set 6 and 7 in month granular form. Similarly, we can represent time dimension of S_4.

Now, based on the above time representation, we can hierarchically organize these times in an R-tree that also organizes spatial and textual dimensions. Since, R_4 is bounding the location of spatial object S_6, S_7, and S_8 (in Figure 5.3), the time content of node R_4 is the aggregate time of its child nodes. Thus, in this example, time entry of node R_4 is represented using a month-type time form, where we set months 6 and 7 that covers the time dimensions of all of its child nodes. Similarly time entry of node R_3 is of month-type and we set its time contents as 6, 7, and 8 in month-type time form.

The time content of an internal node R_6 is obtained by aggregating the time contents of node R_3 and R_4 as shown in Figure 5.3. The above process continues in such a way that any non-leaf node contains the aggregate time of its child nodes' time. In this way, the root also gets its time entry, which covers the time dimensions of the entire data sets.

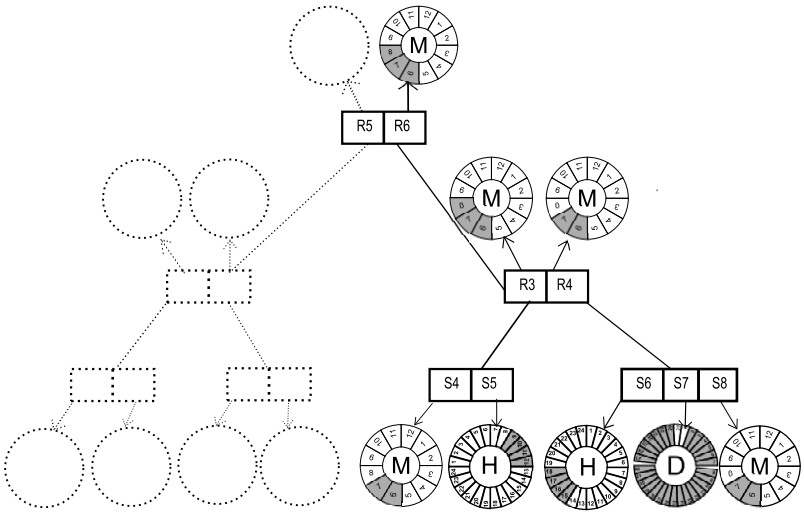


Figure 5.3: Time indexing in various node

5.4 STIR Tree

Figure A.2 represents our proposed hybrid index structure STIR-tree. Leaf node of STIR-tree contains a number of entries. Each entry is of the form ($S.mbr$, $S.\psi, S.t$), where $S.mbr$ is the bounding rectangle of the object S, $S.\psi$ is the keyword or item list of object S and $S.t$ is the time dimension.

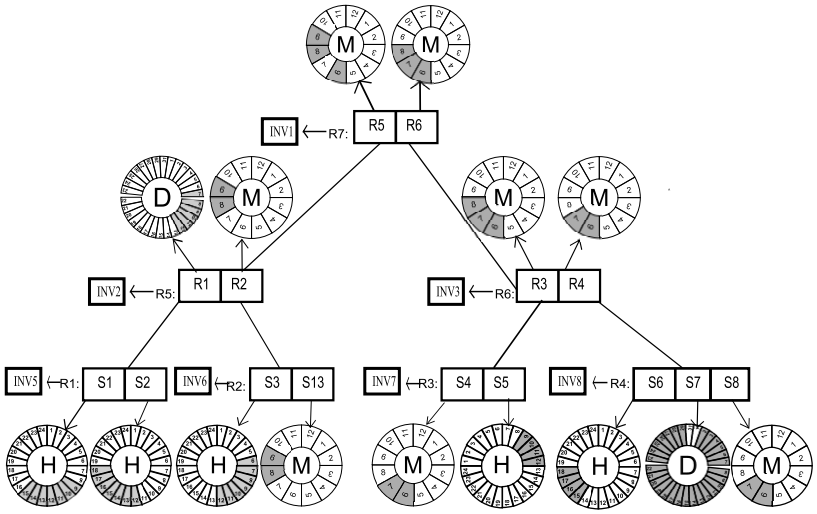


Figure 5.4: STIR tree

A non-leaf node contains child entries. Each child entries is of the form ($N.mbr, N.\psi, N.t$) where $N.mbr$ is the minimum bounding rectangle that spa-

tially encloses all entries contained in the sub-tree rooted at this node, $N.\psi$ is the identifier of an inverted list which is the union of all item in the lists of its child nodes and $N.t$ is the aggregate time that covers time dimensions of all of its child nodes.

In this chapter, we have discussed our location indexing using R-tree, keyword indexing using inverted file and time indexing. Base on our data structure, we organize the time hierarchically in R-tree with location and keyword from leaf to root in order of fine to coarse granularity of time.

Chapter 6

Query processing using STIR-tree

In this chapter, we first show algorithm for query processing of spatio-temporal keyword (STK) search of k nearest neighbor (STK-kNN) using our proposed hybrid index structure STIR-tree. We also discuss time uncertainty of each spatio-temporal object. To measure the uncertainty, we introduce probability math model in our second algorithm which we call STK-kNN with time uncertainty (STK-kTU). As a last step, we discuss another algorithm to find out time based nearest neighbor search (STK-kTNN). Finally, we show an application scenario based on our hybrid index structure.

6.1 Processing of spatio-temporal keyword queries for k nearest neighbor (STK-kNN)

To process spatio-temporal keyword search for k nearest neighbor (STK-kNN) queries using our proposed STIR-tree, we adopt the concept of best first search [16]. Best first search starts from the root node of a tree and the child nodes are recursively accessed in the increasing order of their distances from the query point. The search process terminates as soon as the k nearest objects are retrieved from the STIR-tree. As a pruning strategy a priority queue is maintained to keep only those nodes whose inverted list and temporal list matches the query's keyword and time. When deciding what node to visit next, it picks the node having least distance from set of all nodes in a priority queue.

We assume that each spatial data objects have different time range and number of keywords. We also consider that time dimension of each spatial object is certain. Data objects' location are indexed using an R-tree in the database. Keywords are indexed using inverted file and time is indexed as discussed in Chapter 5. Here we propose algorithm STK-kNN that follows the best first search technique to incrementally access only those data objects having desired keyword and time to find the Top k NN query. (discussed in Chapter 3)

Algorithm 3 shows the steps of finding k nearest neighbor for spatio-temporal keyword search. The algorithm takes the following parameter as input: a query object Q having attributes location, item and fixed time, index R of STIR-tree, and number of expected results k. We summarize the common notations used in this section in Table 6.1.

Notation	Description
$Q.\lambda$	Query's location
$Q.\psi$	Query's item/keyword
$Q.t$	Query's time
k	Number of expected result
$N.\psi$	Nodes's item/keyword list
$N.t$	Nodes's time
$DIST_e$	Euclidean distance between $Q.\lambda$ and object
$DIST_{min}$	Minimum distance between $Q.\lambda$ and object bounding rectangle (MBR)

Table 6.1: Notations used in algorithm

In Algorithm 3, a *priority queue* is maintained to store explored nodes through the search process. The nodes in the queue are sorted according to their minimum distance to the query point. During the search process, best first method repeatedly dequeues the top entry in the queue and enqueues its child nodes with their minimum distance into the queue. When the first data object is dequeued, it is reported as first nearest object. The whole process continues until number of expected result k is found or the priority queue is empty. Steps of Algorithm 3 is given below:

Algorithm 3 $STK - kNN(Query, R, k)$

1: INPUT: Query object Q, *Index* of STIR-tree, expected number of result k.
2: OUTPUT: Top-k spatio-temporal object.
3: *Queue* ←NewPriorityQueue();
4: **if** ($Q.\psi \cap N.\psi \neq \emptyset$ **and** $Q.t \cap N.t \neq \emptyset$) **then**
5: *Queue*.Enqueue($Index.RootNode, 0$); */ Root node is queued first along with distance 0*/*
6: **end if**
7: **while** (**not** *Queue*.IsEmpty()) **do**
8: *Element* ← *Queue*.Dequeue();
9: **if** (*Element* is an object) **then**
10: **if** (**not** *Queue*.IsEmpty()**and**$DIST_e(Query, Object) >$ *Queue*.First().Key) **then**
11: *Queue*.Enqueue(*Object,* $DIST_e$*(Query,Object)*);
12: **else**
13: Report *Element* as the nearest object;
14: **if** k nearest objects have been found **then**
15: **break**;
16: **end if**
17: **end if**
18: **else if** (*Element* is a leaf node) **then**
19: **for each** entry(*Object*) in leaf node *Element* **do**
20: **if** ($Q.\psi \cap N.\psi \neq \emptyset$ **and** $Q.t \cap N.t \neq \emptyset$) **then**
21: *Queue*.Enqueue(*Object,* $DIST_e$*(Query,Object)*);
22: **end if**
23: **end for**
24: **else**
25: **for each** entry(*Node*) in node *Element* **do**
26: **if** ($Q.\psi \cap N.\psi \neq \emptyset$ **and** $Q.t \cap N.t \neq \emptyset$) **then**
27: *Queue*.Enqueue(*Node,* $DIST_{min}$*(Query,Node)*);
28: **end if**
29: **end for**
30: **end if**
31: **end while**

Step of the Algorithm 3 is given below:

Step 1: A user first sends her query that contain keyword, time, and her point location. (Line 1)

Step 2: A *Queue* is initialized. (Line 3)

Step 3: The algorithm first starts with the root node and check whether root node contain query's keyword and time. (Line 4)

Step 4: If root node contains query's keyword and time, then algorithm enqueues it into *Queue* along with it's distance (*i.e. 0*) from the query's point location. (Line 5)

Step 5: Algorithm dequeue the top entry in the queue and keep it in the *Element*. (Line 8)

Step 6: In each iteration, *Element* is checked for whether it is a data object or node. If it is a node, then algorithm checks it's child nodes contain query's keyword and time. If any child node contains those then algorithm enqueues it into *Queue*. If the element is a data object then it is reported as first nearest spatio-temporal object. The whole process continues until the *Queue* is empty or number of expected result k nearest object is found. (Lines 7-31)

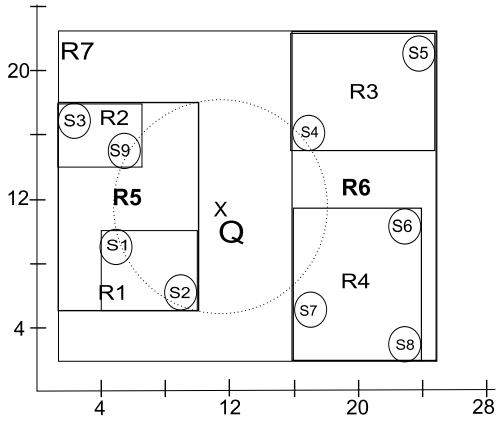


Figure 6.1: Query on STIR tree

In Figure 6.1 the query starts at the root node R_7. Assume R_5, R_6 matches the keyword and time, so they are kept in priority queue. Since R_5 is nearer to $Q.\lambda$ so it is popped up from the priority queue and checked it's child nodes' keyword and time. The process continues recursively until top-K spatio-temporal object is found. We are going to explain the algorithm with an example.

Example 1: Consider the query Q in Figure 6.1 where $Q.\lambda = (11, 11)$, $Q.\psi =$ *Onion*, $Q.t$ = 30/06/14 1500hrs and $k = 2$
The algorithm starts with enqueueing R_7 and executes the following step:

1. Dequeue R_7, enqueue R_5, R_6
   Queue:{(R_5,2,M{6,8,9}),(R_6,6,M{6,7,8})}

2. Dequeue R_5, enqueue Nothing [R_1 pruned by time, R_2 pruned by keyword and time]
   Queue:{(R_6,6,M{6,7,8})}

3. Dequeue R_6, enqueue R_3, R_4
   Queue:{(R_4,6,M{6,7}), (R_3,7.81,M{6,7,8})}

4. Dequeue R_4, enqueue S_7 [S_6, S_8 pruned by keyword]
   Queue:{(R_3,7.81,M{6,7,8}), (S_7,8.48,M{6,7})}

5. Dequeue R_3, enqueue S_4 [S_5 pruned by time]
   Queue:{(S_4,7.81,M{6,7}), (S_7,8.48,M{6,7})}

6. Dequeue S_4, and report as 1st nearest object.
   Queue:{(S_7,8.48,M{6,7})}
   item Dequeue S_7 and report as 2nd nearest object.

Finally S_4, S_7 reported as 2 *NN* object.

6.2 Time Uncertainty

Uncertainty is an inherent property in many database application that includes location based service [4]. In our envisioned application, there exists time uncertainty as items may be available at predicted time interval. For example, a seller may predict that his item "Brown rice" may be available from 01 Oct 2014 to 07 Oct 2014. Again when a buyer submit a search query like *"I may need brown rice from 02 Oct 2014 to 06 Oct 2014"*. From both the seller's and buyer's point of view, there exists an uncertainty about the exact time when an item is available or wanted. So in our application, each spatio-temporal object has time uncertainty. In this section, we first discuss time uncertainty

and propose one algorithm that handles time uncertainty of spatio-temporal objects.

Let S be a set of uncertain retrieved object. An uncertain object $S_i \in S$ is represented by a d dimensional uncertain range R_i and probability density function(pdf) $f_i(u)$ that satisfies $\int_{R_i} f_i(u)du = 1$ for $u \in R_i$. If $u \notin R_i$, then $f_i(u) = 0$. For uniform distribution, the pdf of S_i can be expressed as $f_i(u) = \frac{1}{Area(R_i)}$ for $u \in R_i$. For one dimensional object S_i, uncertainty region and the pdf can be represented as $R_i = [l_i, u_i]$ and $f_i(t) = \frac{1}{u_i - l_i}$, where l_i is the lower bound of time, u_i is the upper bound of time and $l_i < u_i$.

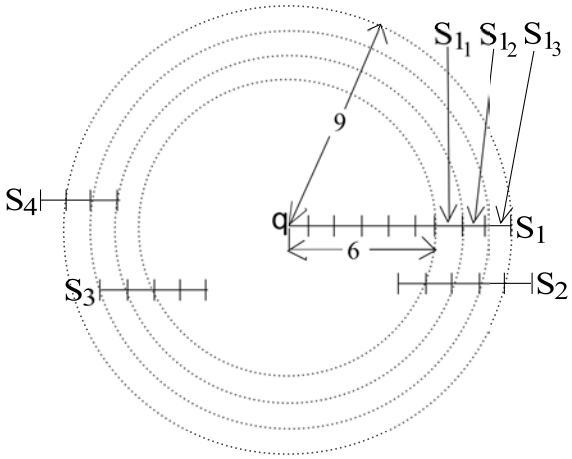


Figure 6.2: An example of time uncertainty

Suppose number of retrieved objects is four (i.e. S_1, S_2, S_3 and S_4) with various time interval(t_i) as shown in Figure 6.2 . Here the resolution metric is *hour*. If the uncertainty regions are R_1=(l_1, u_1), R_2=(l_2, u_2), R_3=(l_3, u_3), and R_4=(l_4, u_4) for objects S_1, S_2, S_3, and S_4 respectively, then nearest neighbor return all four (S_1, P_1), (S_2, P_2), (S_3, P_3) and (S_4, P_4) as probable nearest NN for query point q where P_1, P_2, P_3, and P_4 represents probability value of their respective spatial object.

The probability $P(S_i, q)$ of an object S_i being the most probable in terms of time to a query point(q) time can be computed as follows. For any point $t \in R_i$, we need to first find out the probability of S_i being at t and multiply it by the probability of all other objects being farther than t with respect to q, and then summing up these products for all t to compute $P(S_i, q)$. Thus $P(S_i, q)$ can be expressed as follows:

$$P(S_i, q) = \int_{t \in R_i} f_i(t)dt(\prod_{j \neq i} \int_{v \in R_j \wedge d(q,t) \leq d(q,v)} f_j(v)dv) \tag{6.1}$$

Figure 6.2 shows a query point having fixed time Q, and four objects S_1, S_2, S_3, and S_4. In this example, we assume a discrete space where the time interval of four objects are 3, 5, 4, and 3 units respectively. Time distance between S_1 and q is 6 units. Suppose that dashed circles $(q, 6)$, $(q, 7)$, $(q, 8)$, and $(q, 9)$ centered at q with radii 6, 7, 8, and 9 units respectively, divide the uncertain region R_1 of S_1 into three sub-regions S_{1_1}, S_{1_2}, and S_{1_3}. Based on Equation 6.1, $p(S_1, q)$ can be computed by summing: (1) the probability of S_1 being within the sub-region S_{1_1}, multiplied by the probabilities of S_2, S_3, and S_4 being outside the circular region (q, 7), (2) the probability of S_1 being within the sub-region S_{1_2}, multiplied by the probabilities of S_2, S_3, and S_4 being outside the circular region (q, 8), (3) the probability of S_1 being within the sub-region S_{1_3}, multiplied by the probabilities of S_2, S_3, and S_4 being outside the circular region (q, 9).

Now if q has a uncertain time instead of fixed time, then $P(S_i, q)$ can be expressed as follow:

$$P(S_i, q) = \sum_{q_i \in Q} \int_{t \in R_i} f_i(t)dt(\prod_{j \neq i} \int_{v \in R_j \wedge d(q_i,t) \leq d(q_i,v)} f_j(v)dv) \tag{6.2}$$

Now we are going to explain this with an example.

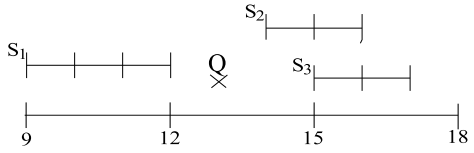


Figure 6.3: Q is point

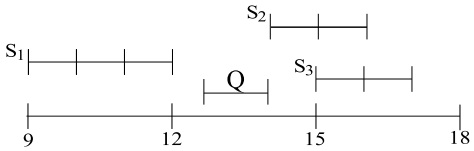


Figure 6.4: Q is range

Example 2: For simplicity let us assume that three NN objects as shown in Figure 6.3(a) S_1, S_2 and S_3 having time information "07/6/14, 0900∼1200

hours", "7/6/14, 1400∼1600 hours" and "7/6/14, 1500∼1700 hours". If Q's time is "07/6/14, 1300 hours". The score of S_1 with respect to S_2 and S_3 is $(\frac{1}{3} * \frac{1}{2} * 1 + \frac{1}{3} * 0 * \frac{1}{2} + \frac{1}{3} * 0 * 0)$ or 0.1667. Now the score of S_2 with repect to S_1 and S_3 is $(\frac{1}{2} * \frac{2}{3} * 1 + \frac{1}{2} * \frac{1}{3} * \frac{1}{2})$ or 0.416. Finally the score of S_3 with repect to S_1 and S_2 is $(\frac{1}{2} * \frac{1}{3} * 0 + \frac{1}{2} * 0 * 0)$ or 0.00.

Example 3: If Q's time is "07/6/14, 1300∼1400 hours" as shown in Figure 6.4(b) then the score of S_1 with respect to S_2 and S_3 is $\frac{1}{2}(\frac{1}{3} * 1 * 1 + \frac{1}{3} * \frac{1}{2} * 1 + \frac{1}{3} * 0 * \frac{1}{2}) + \frac{1}{2}(\frac{1}{3} * 0 * \frac{1}{2} + \frac{1}{3} * 0 * 0)$ or 0.25.

6.2.1 STK-kNN with Time Uncertainty

In previous section, we have shown STK-kNN query that takes time, keyword, and point location as input and retrieves k number of spatio-temporal objects where we assume that time dimension of each object is certain. In this section, we assume that each spatio-temporal object has time uncertainty at different degree. Our second algorithm STK-kNN with time uncertainty (STK-kTU) takes time (uncertain), keyword, and point location as input and retrieves k number of spatio-temporal objects and associated probabilities denoting the likelihood of the object being nearest to the query object. Very often user may be interested to find the list of nearest objects that fall in the given range from a fixed time. To realize this scenario, we introduce one additional parameter α in this algorithm (STK-kTU) which is used to find a range (lower time and upper time) from a fixed time. We also add probability function (Equation 1) to calculate probability value of each retrieved object with respect to query object's time.

Difference between algorithm STK-kTU and STK-kNN is that algorithm STK-kTU has one additional parameter α to check whether nodes' time content intersects lower time and upper time. It also has one computational cost to calculate time probability of each retrieved object with respect to other retrieved objects. Algorithm STK-kTU takes the following parameter as input: a query object Q having attributes location, item, and time, index R of STIR-tree, number of expected results k, and threshold α. Algorithm checks each node's time content. If node's time content intersects lower time and upper time then algorithm visits it's child nodes. Gradually it reaches leaf nodes and retrieves

k nearest neighbor spatio-temporal objects with respect to query's point location. As these spatio-temporal objects have time with uncertainty, so as a last step algorithm calculates the time probability of each of the k spatio-temporal objects with respect to k-1 objects. Step of the Algorithm STK-kTU is given below:

Step 1: A user first sends her query that contain keyword, time, threshold, and her point location. (Line 1)

Step 2: Initialize lower time and upper time. (Line 3,4)

Step 3: Initialize a list *S*. (Line 5)

Step 4: A *Queue* is initialized. (Line 6)

Step 5: The algorithm first starts with the root node and checks whether root node contain query's keyword and time that intersects lower time and upper time. (Line 7)

Step 5: If root node contains query's keyword and required time, then algorithm enqueues it into *Queue* along with it's distance. (Line 8)

Step 6: Algorithm dequeues the top entry in the queue and keep in the *Element*. (Line 11)

Step 7: In each iteration, *Element* is checked for whether it is a data object or node. If it is a node, then algorithm checks it's child nodes' whether they contain query's keyword and time that intersects lower and upper time. If any child node fulfill the condition then algorithm enqueues it into *Queue*. If the element is a data object then it is included in the list *S*. The whole process continues until the *Queue* is empty or number of expected result k object is found. (Lines 7-35)

Step 8: When S is filled up by k spatio-temporal objects then algorithm calculates the time probability score of each object in list S by equation 1 with respect to k-1 objects. (Line 8)

Algorithm 4 $STK - kTU(Query, R, k, \alpha)$

1: INPUT: Query object Q, *Index* of STIR-tree, expected number of result k, threshold α.
2: OUTPUT: Top-k spatio-temporal object with probability score.
3: $\alpha_l \leftarrow Q_t - \alpha$;
4: $\alpha_u \leftarrow Q_t + \alpha$;
5: $S \leftarrow \phi$;
6: $Queue \leftarrow$NewPriorityQueue();
7: **if** $(Q.\psi \cap N.\psi \neq \emptyset$ **and** $(\alpha_l, \alpha_u) \in N.t)$ **then**
8: $Queue$.Enqueue($Index.RootNode, 0$); */ *Root node is queued first along with distance 0*/*
9: **end if**
10: **while** (**not** $Queue$.IsEmpty()) **do**
11: $Element \leftarrow Queue$.Dequeue();
12: **if** (*Element* is an object) **then**
13: **if** (**not** $Queue$.IsEmpty()**and**$DIST_e(Query, Object) > Queue$.First().Key) **then**
14: $Queue$.Enqueue(*Object,* $DIST_e$*(Query,Object)*);
15: **else**
16: $S \leftarrow Element$;
17: **if** k nearest objects have been found **then**
18: Calculate time probability of each object in S;
19: **break**;
20: **end if**
21: **end if**
22: **else if** (*Element* is a leaf node) **then**
23: **for each** entry(*Object*) in leaf node *Element* **do**
24: **if** $(Q.\psi \cap N.\psi \neq \emptyset$ **and** $(\alpha_l, \alpha_u) \in N.t)$ **then**
25: $Queue$.Enqueue(*Object,* $DIST_e$*(Query,Object)*);
26: **end if**
27: **end for**
28: **else**
29: **for each** entry(*Node*) in node *Element* **do**
30: **if** $(Q.\psi \cap N.\psi \neq \emptyset$ **and** $(\alpha_l, \alpha_u) \in N.t)$ **then**
31: $Queue$.Enqueue(*Node,* $DIST_{min}$*(Query,Node)*);
32: **end if**
33: **end for**
34: **end if**
35: **end while**

6.2.2 Time based Nearest Neighbor with STIR-tree

In previous subsection, we have discussed STK-kNN query with time uncertainty. In this subsection, we discuss time based nearest neighbor queries. Very often a user may be interested in nearest time. For example, when a user submit a query like *"find brown rice around 18 Sep 2014"*. In such scenario, query retrieves a list of spatio-temporal object that is nearest to query's time. In this subsection, we propose Algorithm 5($STK - kTNN$) that takes the following parameter as input: a query object Q having attributes location, item and fixed time, index R of STIR-tree, and number of expected results k.

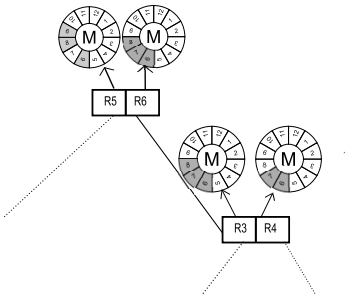


Figure 6.5: Nearest time query on STIR tree

Consider query point time $Q.t = 30/10/14$ 1500 hrs. In Figure 6.5 the query starts at the root node R_7. Consider R_5, R_6 matches the keyword, so they are kept in priority queue. Since R_5 has nearest available time (i.e. 9) than R_6 (i.e. 8) so it is popped up from the priority queue and checked for it's child nodes keyword and further nearest available granular time. The process continues recursively until top-K spatio-temporal object having nearest time is found.

The difference between algorithm STK-kTNN and STK-kTU is that algorithm STK-kTNN checks nearest neighbor time in each node of STIR-tree (Lines 4, 20, and 26) and it does not calculate time probability after retrieving k objects. Step of the Algorithm 5 is given below:

Step 1: A user first sends her query that contain keyword, time, and her point location. (Line 1)
Step 2: A *Queue* is initialized. (Line 3)
Step 3: The algorithm first starts with the root node and checks whether root node contain query's keyword and nearest time. (Line 4)

Step 4: If root node contains query's keyword and nearest time, then algorithm enqueues it into *Queue* along with it's distance (*i.e. 0*) from the query's point location. (Line 5)

Step 5: Algorithm dequeues the top entry in the queue and keep in the *Element*. (Line 8)

Step 6: In each iteration, *Element* is checked for whether it is a data object or node. If it is a node, then algorithm checks whether it's child nodes contain query's keyword and nearest time. If any child node contain query's keyword and nearest time, then algorithm enqueues it into *Queue*, otherwise that node is pruned. If the element is a data object then it is reported first and algorithm derives the time distance(threshold) between query's time and object's farthest time. Then algorithm retrieves other spatio-temporal objects that falls within the threshold. The whole process continues until the *Queue* is empty or number of expected result k object is found. (Lines 7-31)

Algorithm 5 $STK - kTNN(Query, R, k)$

```
1: INPUT: Query object Q, Index of STIR-tree, expected number of result k.
2: OUTPUT: Top-k spatio-temporal object.
3: Queue ←NewPriorityQueue();
4: if (Q.ψ ∩ N.ψ ≠ ∅ and Q.t nearest to N.t) then
5:   Queue.Enqueue(Index.RootNode, 0); */ Root node is queued first along
     with distance 0*/
6: end if
7: while (not Queue.IsEmpty()) do
8:   Element ← Queue.Dequeue();
9:   if (Element is an object) then
10:    if (not Queue.IsEmpty()and DIST_e(Query, Object) >
       Queue.First().Key) then
11:      Queue.Enqueue(Object, DIST_e(Query,Object));
12:    else
13:      Report Element as the nearest object and find it's nearest time;
14:      if k nearest objects have been found then
15:        break;
16:      end if
17:    end if
18:  else if (Element is a leaf node) then
19:    for each entry(Object) in leaf node Element do
20:      if (Q.ψ ∩ N.ψ ≠ ∅ and Q.t nearest to N.t) then
21:        Queue.Enqueue(Object, DIST_e(Query,Object));
22:      end if
23:    end for
24:  else
25:    for each entry(Node) in node Element do
26:      if (Q.ψ ∩ N.ψ ≠ ∅ and Q.t nearest to N.t) then
27:        Queue.Enqueue(Node, DIST_min(Query,Node));
28:      end if
29:    end for
30:  end if
31: end while
```

6.3 Spatio-Temporal Keyword Stream: An Application Scenario

Our proposed methodologies for spatio-temporal keyword search can handle *continuous stream* of seller and buyer requests. In our envisioned buyer-seller community application, a seller can post an item any time or a buyer can search for his desired item at any time. To support a continuous matching of buyers and sellers requests, we maintain two STIR-trees, one for sellers' posted items, an another for buyers' requested items. Initially, all sellers' items are indexed in a seller index tree (seller-tree). When a buyer sends a query for a desired item, the system searches for the desired item in the seller-tree. If it finds desired item, the buyer query is retrieved from the system. However, if there is no match of the buyer's desired item in the existing seller-tree, the buyer's query is saved in the buyer-tree if the buyer requests an item for a future time. Whenever, a new seller item appears in the system, the system first searches the item in the buyer-tree to see if there is a match with any of the pending queries saved in the buyer-tree. If the system finds a match, it reports the answer to both seller and buyer. Otherwise, the seller item is saved in seller-tree.

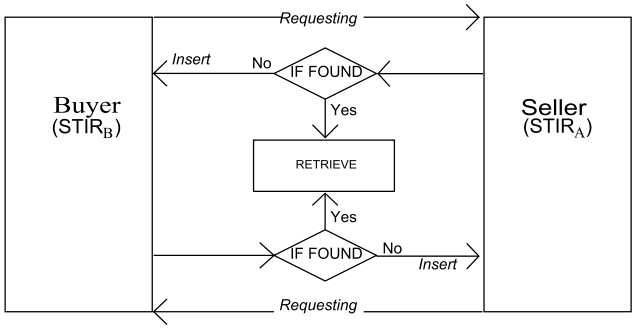


Figure 6.6: Flowchart of seller and buyer community

Figure 6.6 shows a flow-chart of the buyer-seller community application. The figure shows two indices, $STIR_A$ is the seller-tree and $STIR_B$ is the buyer-tree. Here a buyer searches for his desired item at a specified time. If the item is found in $STIR_B$, the application retrieves those seller with location as an ascending order of the distance from the query (buyer) location and delete that seller object from the seller-tree if there is no more item left. If the item is not found, the system inserts the requested item along with their other attributes

(i.e., location, keyword/item, and time) into the buyer-tree, $STIR_B$. Similarly when a seller object appears in the system, it also searches the buyer-tree for the desired item. If the suitable buyers are found that matches seller's query keyword and time, then it retrieves buyers' location as an ascending order of their distance from seller's location and delete that buyer object from $STIR_B$. If the item is not found then it inserts that seller along with other attributes into $STIR_A$.

In this chapter, we have have highlighted 3 types of queries mainly spatio-temporal keyword search for k nearest neighbor (STK-kNN), STK-kNN with time uncertainty, and time based nearest neighbor search (STK-kTNN). Finally, we have shown an application scenario based on our proposed hybrid index structure (*STIR tree*).

Chapter 7

Experimental Study

In this section, we discuss the performance of our STIR based spatio-temporal keyword search queries with an extensive set of experiments.

7.1 Experimental Setup

We use synthetic datasets and query sets in our experiment. We generate synthetic dataset with uniform distribution representing a wide range of real scenario. We vary the data size as 5000, 10,000, 15,000, and 20,000 point locations. We randomly choose the time-stamps with different granularity for each spatial object. For keyword, we randomly choose keyword/item from a list and embed them in data object.

We also generate synthetic query sets with uniform distribution and vary the query size as 500, 1000, 1500 and 2000 number. Here we embed the keyword by randomly choosing from a list of keyword, and embed the time by randomly choosing the time-stamps with different granularity. We vary the number of expected result k as 1, 2, 4, 8, 16, and 20. We also vary the time range/threshold(α) as 0, 5, 10, 15, and 20 for the experiment STK-kNN with time uncertainty. Necessary parameter, parameter range and their default value are shown in Table 7.1.

In our experiment, objects' point location are indexed using R-tree [13], keyword are indexed using inverted file [27], and time is indexed as described in Chapter 5 which results our index structure STIR-tree. We use the node capacity of 50 entries for the STIR-tree.

All the experiment are conducted on a system with Intel dual core 2.66 GHz processor and 2 GB of memory running Windows XP. We implement our hybrid index structure and search algorithm in $C^{\#}$.

7.2 Experimental Evaluation

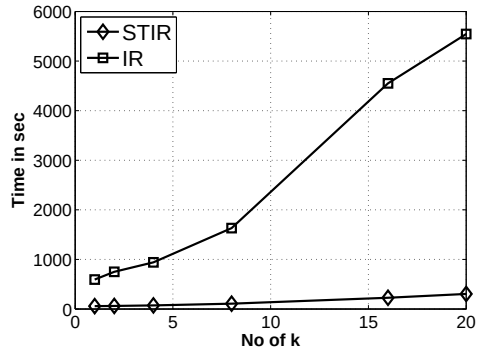
We evaluate our proposed STIR index based algorithms for three types of queries, i.e. spatio-temporal keyword search for nearest neighbor queries (STK-kNN), spatio-temporal keyword search for nearest neighbor queries with time uncertainty (STK-kTU), and spatio-temporal keyword search for time based nearest neighbor queries (STK-kTNN). We compare our approach with a naive approach. In the naive approach, we have used IR-tree [10] to index location and keywords of spatio-temporal object, and filter out time at the final refining stage while running the queries. In all experiments, we measure the query processing time and I/O cost as the efficiency measures of the algorithm.

Parameter	Range	Default
k	1, 2, 4, 8, 16, 20	4
Data set	5000, 10000, 15000, 20000	10000
Number of query (q)	500, 1000, 1500, 2000	1000
Time threshold (α)	0, 5, 10, 15, 20	10

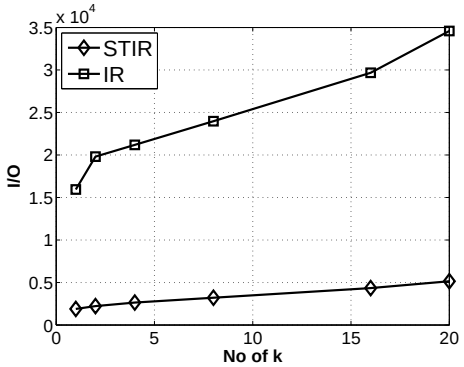
Table 7.1: Table of data set

7.3 Experimental Results

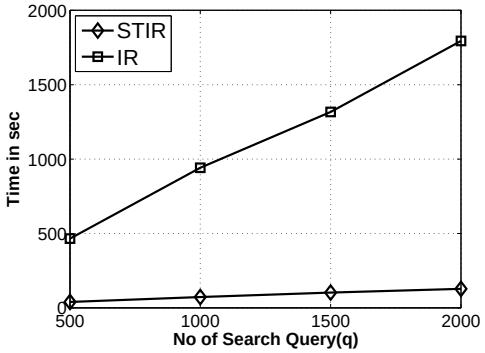
Here we elaborately discuss the performance of our proposed algorithm STK-kNN, STK-kTU, and STK-kTNN in sub-subsection 7.3.1, 7.3.2, and 7.3.3 respectively.



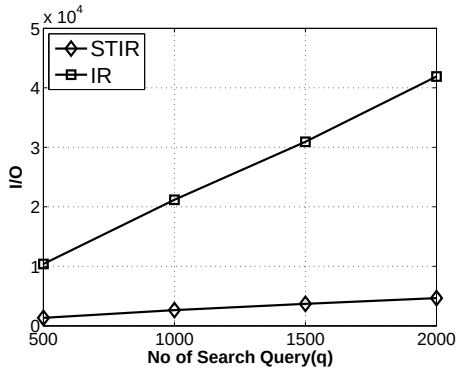
(a) k vs Time



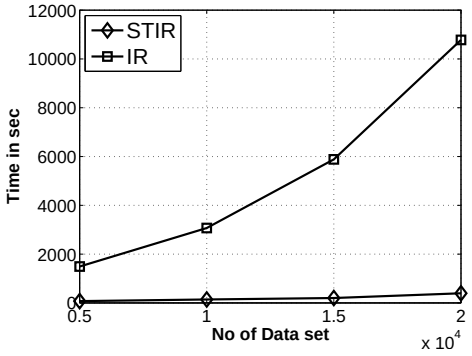
(b) k vs I/O



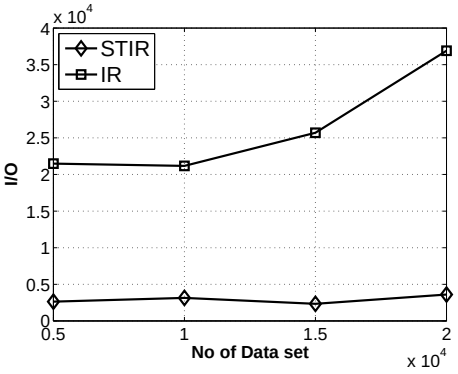
(c) Search query vs Time



(d) Search query vs I/O



(e) Dataset vs Time



(f) Dataset vs I/O

Figure 7.1: Experimental graph for STK-kNN

7.3.1 STK-kNN

In our first experiment, we vary the number of expected result k, number of search query q, and number of dataset to measure the performance of the algorithm. Here we consider that, time dimension of each spatial object is certain.

Effect of *k*: In this experiment, we vary the value of k using 1, 2, 4, 8, 16, and 20. Figure 7.1(a) and 7.1(b) show that our STIR structure outperforms than IR structure. At lower value of $k(<8)$ processing time and I/O cost is 10-15 times faster and requires 7-8 times less I/O cost than that of naive approach. At upper values of $k(>8)$ processing time is 15-20 times faster and it needs 6-7 times less I/O than that of naive approach. This is expected, since with IR approach it need to access more tree nodes and more objects. In contrast, STIR-tree uses time index to prune the whole subtree if node time list does not match the query time.

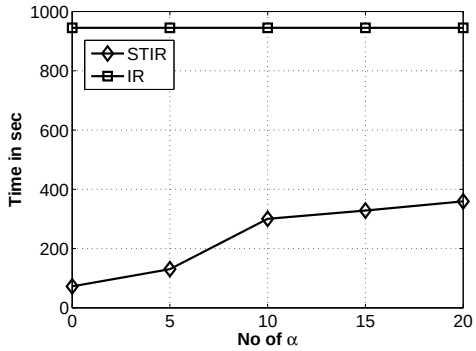
Effect of Number of Search Queries: We vary the number of search queries q in the range of 500 to 2000 with a step size of 500 units as shown in Figure 7.1(c) and 7.1(d). Query processing time is 11.8-14 times faster than IR approach when we use STIR-tree and it requires 7-9 times less I/O cost than that of IR approach. Thus we find that our algorithm outperforms for STIR approach by a greater margin for an increased value of *q*. With increase of number of search query there is a great possibility of having node with more irrelevant time which is pruned in STIR approach and decreases processing time and I/O cost significantly.

Effect of Number of Dataset: Figure 7.1(e) and 7.1(f) show the required processing time and I/O cost in STIR and IR approach. In this experiment, we vary the dataset size using 5K, 10K, 15K, and 20K with step size of 5000. Here query processing time for STIR approach is 20-27 times faster than naive approach and it requires around 6-11 times less I/O cost than that of naive approach.

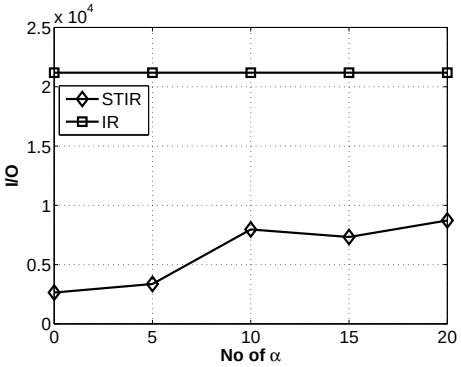
7.3.2 STK-kTU

In our second experiment, we first vary α. Then we keep α=10 while varying other parameter *k, q*, and *dataset* size to measure the performance of the algorithm. Here we consider that, time dimension of each spatial object is uncertain.

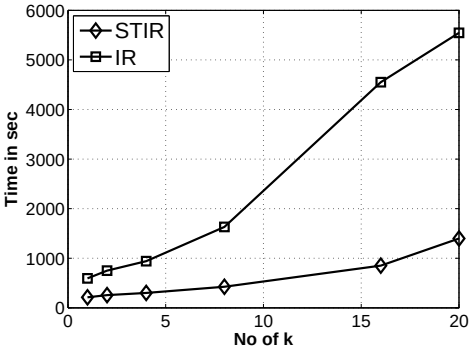
Effect of α: Figure 7.2(a) and 7.2(b) shows the required processing time and I/O cost in STIR by varying α. In this experiment, we vary time threshold α



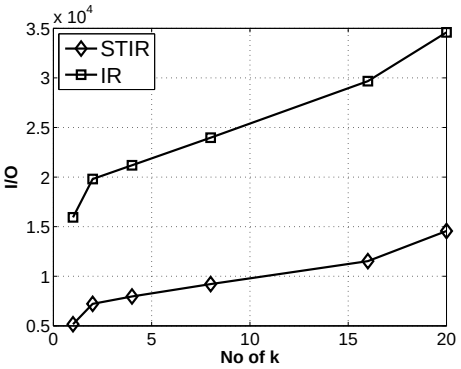
(a) Threshold vs Time



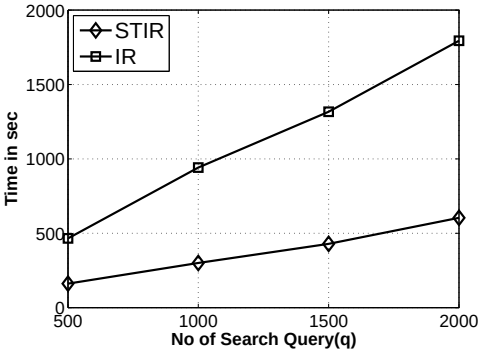
(b) Threshold vs I/O



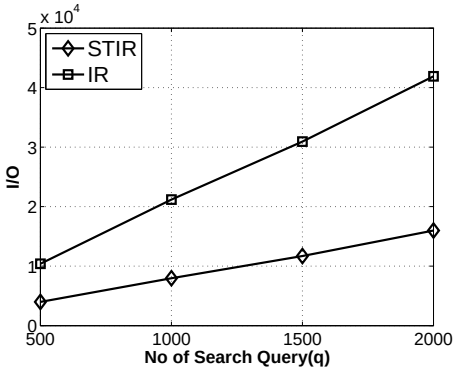
(c) k vs Time



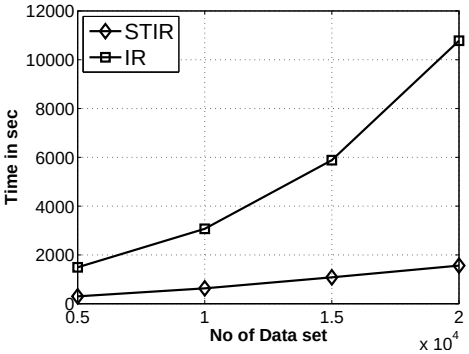
(d) k vs I/O



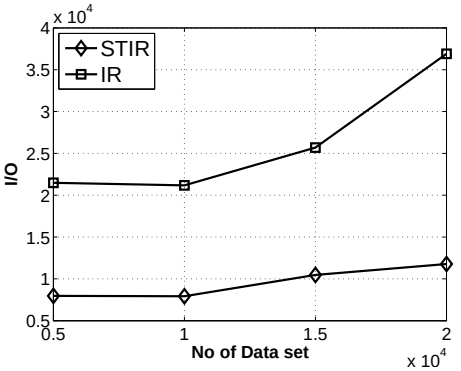
(e) Search query vs Time



(f) Search query vs I/O



(g) Dataset vs Time



(h) Dataset vs I/O

Figure 7.2: Experimental graph for STK-kTU

using 0, 5, 10, 15, and 20. From the graph, we find that as α increases processing time and I/O of STIR also increases. It is expected, as increasing α causes more node of STIR-tree to be opened for processing. In IR structure, since it does not consider time, so varying α does not change the processing time and I/O cost and it remains constant for whole range of α.

Effect of *k*: Here, we investigate the effect of varying the number of results k using 1, 2, 4, 8, 16, and 20. Figure 7.2(c) and 7.2(d) show query processing time and I/O cost increases for STIR approach in comparison to IR approach. We find that at lower value of $k(<8)$ query processing time is 2.8-3.8 times faster and require 2.6-3.1 times less I/O cost than that of IR approach, whereas at upper value of $k(>8)$ processing time is 3.9-5.3 times faster and it requires 2.3-2.5 times less I/O cost than that of IR approach when we take STIR approach. For STK-kTU, query processing time and I/O cost increases by 3.7-4.6 times and 2.7-3.2 times respectively than STK-kNN query due to time threshold α and computing probability score (Figure 7.1(a), 7.1(b)).

Effect of Number of Search Queries: Figure 7.2(e) and 7.2(f) show the results when we vary the number of search queries in the range of 500 to 2000 with a step size of 500. Here, STIR is on average 3 times faster and it needs on average 2.6 times less I/O cost than that of IR approach. We find that, for STK-kTU, query processing time and I/O cost increases than STK-kNN query on average 4 times and 3 times respectively (Figure 7.1(c), 7.1(d)). It is expected, as time threshold α causes more nodes of STIR to be opened for further processing and there also involve extra computational cost while calculating the probability of each spatio-temporal object. But there always remain a great margin between STIR and IR approach.

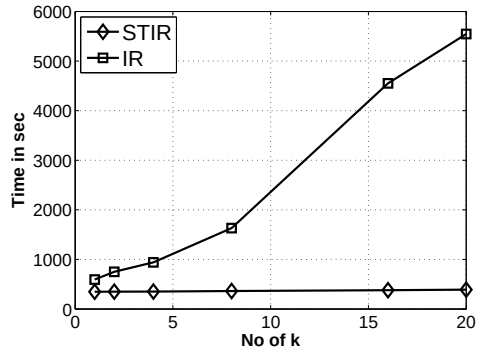
Effect of Number of Dataset: In this experiment, we vary the dataset size using 5K, 10K, 15K, and 20K with step size of 5000 (Figure 7.2(g), 7.2(h)). Here query processing time is 4.8-6.9 times faster and it needs around 2.4-3.1 times less I/O cost than that of IR approach when we use STIR approach. For STK-kTU, query processing time and I/O cost increases on average 3.9-5.4 times and 2.5-4.5 times respectively than that of STK-kNN query (Figure 7.1(e), 7.1(f)).

7.3.3 STK-kTNN

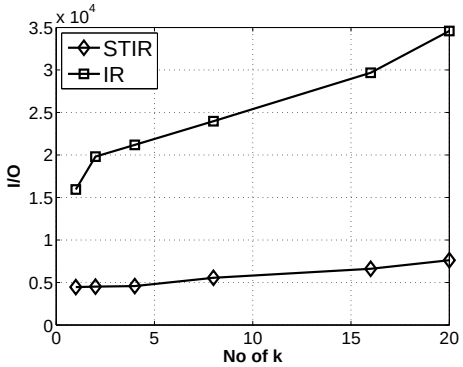
In our third and final experiment, we have considered nearest neighbor time in each node and finally retrieve the time based nearest object that falls within a threshold which is derived from a first retrieved spatio-temporal object. To measure the performance, we vary the parameter *k, q* and *dataset* size. Here we consider that, time dimension of each spatial object is certain.

Effect of *k*: Figure 7.3(a) and 7.3(b) show that processing time and I/O cost increases as in comparison to IR approach for all values of k. At lower value of $k(<8)$ query processing time is 1.7-4.5 times faster and for upper value of $k(>8)$ processing time is 12-14 times faster and it needs around 4 times less I/O cost for all values of k than IR approach. For STK-kTNN, query processing time and I/O cost increase 1.2-6 times and 1.4-2.3 times respectively than that of STK-kNN query (Figure 7.1(a), 7.1(b)). On the other hand, query processing time increases by 1.2-1.6 times for $k<8$ and decreases by 1.2-3.6 times for $k>8$ and I/O decreases by 1.15-2 times for all values of k than that of STK-kTU (Figure 7.2(c), 7.2(d)). It is expected, as STK-kTU approach, algorithm has to compute probability score for significant number of spatio-temporal objects for upper value of $k(>8)$, and for $k<8$, it has to calculate probability score for relatively less number of spatio-temporal objects. Again for STK-kTU approach, default value of α=10 causes more nodes to be explored than STK-kTNN, hence I/O cost decreases in third experiment.

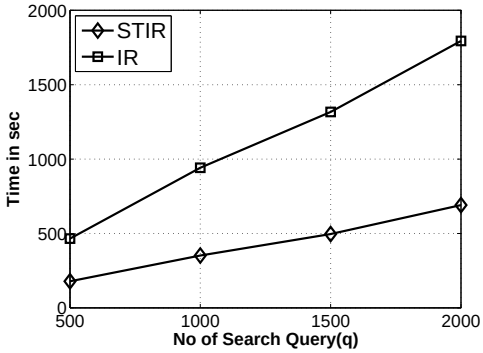
Effect of Number of Search Queries: Figure 7.3(c) and 7.3(d) shows the results when we vary the number of search queries in the range of 500 to 2000 with a step size of 500 units. Here, STIR is around 2.6 times faster and it requires 4.3-4.8 times less I/O cost than IR approach. For STK-kTNN query, processing time and I/O cost increase 4.5-5.4 times and 1.8 times respectively than STK-kNN query (Figure 7.1(c), 7.1(d)). Again query processing time increases 1.1-1.4 times and I/O cost decreases by 1.6-1.8 times than that of STK-kTU (Figure 7.2(e), 7.2(f)). It is expected as STK-kTNN algorithm takes more processing time while visiting the nodes and checking the nearest time. But for STK-kTU, opening α causes more node to visit than STK-kTNN.



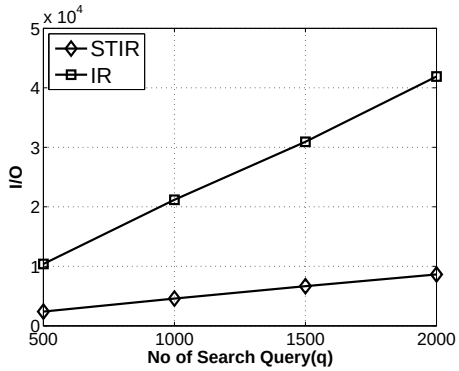
(a) k vs Time



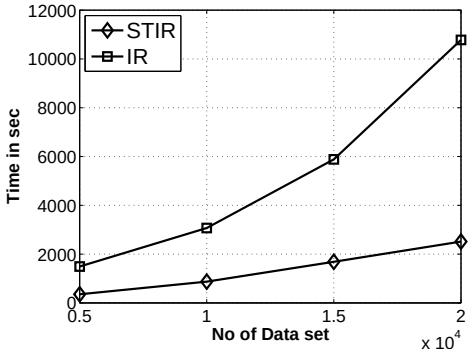
(b) k vs I/O



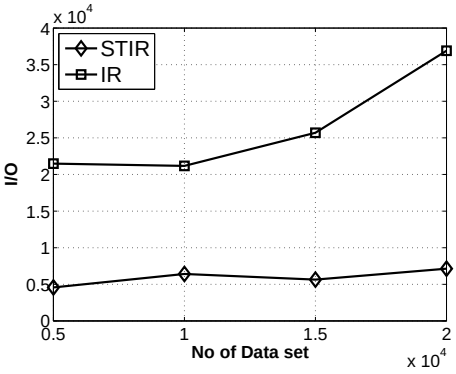
(c) Search query vs Time



(d) Search query vs I/O



(e) Dataset vs Time



(f) Dataset vs I/O

Figure 7.3: Experimental graph for STK-kTNN

Effect of Number of Dataset: Figure 7.3(e) and 7.3(f) show the required processing time and I/O cost in STIR and IR approach. Here, STIR is 3.5-4.3 times faster and it needs 3-4.7 times less I/O cost than IR approach. Its query processing time and I/O cost increase by 4.9-8.4 times and 1.9-2.4 times respectively than that of STK-kNN query (Figure 7.1(e), 7.1(f)). Again processing time increases by 1.2-1.6 times whereas I/O decreases around 1.6 times than

STK-kTU query (Figure 7.2(g), 7.2(h)). Similarly we can explain this result.

In this chapter, we have discussed the results that comes out as extensive experiment. We have evaluated our three types of query i.e. STK-kNN, STK-kNN with time uncertainty and STK with time based nearest neighbor(NN), and compare the results with the IR approach. In all experiments, we measure the query processing times and I/O costs as the efficiency measures of the algorithm.

Chapter 8

Conclusion

In this chapter, we summarize our work and highlight promising direction for future work.

8.1 Summary

We have introduced a new type of query, namely spatio-temporal keyword search for nearest neighbor (STK-NN) query. To process STK-NN queries efficiently, we developed a time indexing method and integrate with location and keyword index to form new hybrid index structure spatio-temporal information retrieval R-tree (STIR-tree). Our proposed hybrid index structure handles spatial, textual, and temporal features of data objects simultaneously. We have also extended our approach to handle time uncertainty and propose techniques to process STK-NN queries and time based nearest neighbors with time uncertainty. Our proposed algorithm for finding the nearest neighbor spatio-temporal object significantly outperforms than that of IR approach. We conduct an extensive experiment of query processing using STIR structure. Experimental studies proves its superior performance over state of the art technique in terms of both query processing time and I/O cost. For STK-kNN query, STIR approach is on average 16.6 times faster and access 7.6 times less I/O than that of naive approach. For STK-kTU query, STIR approach is on average 4.3 times faster and access 2.6 times less I/O than that of naive approach. For STK-kTNN query, STIR approach is on average 4.1 times faster and access 4 times less I/O than that of naive approach.

8.2 Future work

Our work on integrating time indexing with location and keyword indexing opens to a number of promising directions for future work. In this work, we have assumed that user's location is a point. In the future work, we will consider user's location as region instead of point thus considering user's privacy. Though we consider time uncertainty while processing our query but we can also consider the location as uncertain and formulate the new problem. While processing our queries we can consider the query location region and develop algorithms for other type of queries, e.g., range queries.

References

[1] The daily star news. *http://archive.thedailystar.net/newDesign/news-details.php?nid=29081, Accessed on 08 Oct 2013*, 2008.

[2] The daily star news. *http://www.thedailystar.net/beta2/news/potato-sells/, Accessed on 08 Oct 2013*, 2013.

[3] Henrich A. A distance-scan algorithm for spatial access structures. *In proceedings of the Second ACM Workshop on Geographic Information Systems*, December 1994.

[4] Mohammed Eunus Ali, Egemen Tanin, Rui Zhang, and Ramamohanarao Kotagiri. Probabilistic voronoi diagrams for probabilistic moving nearest neighbor queries. *Data and Knowledge Engineering*, 75:1–33, 2012.

[5] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Efficient temporal keyword queries over versioned text. *Proceedings of the 19th ACM international conference on Information and knowledge management*, pages 699–708, 2010.

[6] Avishek Anand, Srikanta Bedathur, Klaus Berberich, and Ralf Schenkel. Temporal index sharding for space-time efficiency in archive search. *Proceedings of the 34th international ACM SIGIR conference on Research and development in Information Retrieval*, pages 545–554, 2011.

[7] Xin Cao, Gao Cong, and Christian S. Jensen. Retrieving top-k prestige-based relevant spatial web objects. *Proceedings of the VLDB endowment*, 3(1):373–384, 2010.

[8] Xin Cao, Gao Cong, Christian S.Jensen, and Ben Chin Ooi. Collective spatial keyword querying. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 373–384, 2011.

[9] Lisi Chen, Gao Cong, Christian S. Jensen, and Dingming Wu. Spatial keyword query processing: An experimental evaluation. *Proceedings of the VLDB Endowment*, 6(3):217–228, 2013.

[10] Gao Cong, Christian S.Jensen, and Dinging Wu. Efficient retrieval of the top-k most relevant spatial web objects. *Proceedings of the VLDB Endowment*, 2(1):337–348, 2009.

[11] Chris Faloutsos and Stavros Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems (TOIS)*, 2(4):267–288, October 1984.

[12] Ian De Felipe, Vagelis Hristidis, and Naphtali Rishe. Keyword search on spatial databases. *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, pages 656–665, 2008.

[13] Antonm Guttman. R-trees:a dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, 14(2):47–57, 1984.

[14] Ramaswamy Hariharan, Bijit Hore, Chen Li, and Sharad Mehrotra. Processing spatial-keyword (sk) queries in geographic information retrieval(gir) systems. *In proceedings of the 19th international conference on Scientific and Statistical Database Management (SSDBM)*, pages 1–10, 2007.

[15] Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. *In Advances in Spatial Databases - Fourth international Symposium*, pages 83–95, 1995.

[16] Gisli R. Hjaltason and Hanan Samet. Distances browsing in spatial databases. *ACM Transactions on Database Systems (TODS)*, 24(2):265–318, October 1999.

[17] João B. Rocha Junior, Orestis Gkorgkas, Simon Jonassen, and Kjetil Nrvág. Efficient processing of top-k spatial keyword queries. *In Proceed-*

ings of the International Symposium on Spatial and Temporal Databases (SSTD), pages 205–222, 2011.

[18] Stephen Kelly, Nick Roussopoulas, and Frederic Vincent. Nearest neighbor queries. *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, pages 71–79, 1995.

[19] Ali Khodaei and Cyrus Shahabi. Temporal-textual retrieval:time and keyword search in web documents. *International Journal of Next-Generation Computing*, 3(3):289–310, 2012.

[20] Ali Khodaei, Cyrus Shahabi, and Chen Li. Hybrid indexing and seamless ranking of spatial and textual features of web documents. *Proceedings of the 21st international conference on Database and expert systems application: Part I, August 30 - September 03, 2010, Bilbao, Spain*, pages 450–466, 2010.

[21] Guoliang Li, Jianhua Feng, and Jing Xu. DESKS: Direction-aware spatial keyword search. *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, pages 474–485, 2012.

[22] 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*, pages 275–281, 1998.

[23] Man Lung Yiu, Xiangyuan Dai, Nikos Mamoulis, and Michail Vaitis. Top-k spatial preference queries. *Proceedings of the 2007 IEEE 23rd International Conference on Data Engineering*, pages 1076–1085, 2007.

[24] Chenqxianq Zhai and John Lafferty. A study of smoothing methods for language models applied to information retrieval. *ACM Transactions on Information Systems (TOIS)*, 22(2):179–214, 2004.

[25] Jinzeng Zhang, Xiaofeng Meng, Xuan Zhou, and Dongqi Liu. Co-spatial searcher:efficient tag-based collaborative spatial search on geo-social network. *Proceedings of the 17th international conference on Database Systems for Advanced Applications*, pages 560–575, 2012.

[26] Yinghua Zhou, Xing Xie, Chuang Wang, Yuchang Gong, and Wei-Ying Ma. Hybrid index structures for location-based web search. *Proceedings of the 14th ACM international conference on Information and knowledge management*, pages 155–162, 2005.

[27] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Computing Surveys (CSUR)*, 38(2):1–56, July 2006.

Appendix A

STIR Simulator

STIR simulator is a GUI-enabled software tool that allows the user to obtain hybrid index structure *STIR*. The tool has been entirely coded in $C^{\#}$. This section describes the various features of the tool. Subsection A.1 elaborates on how to use the tool. Subsection A.2 gives minimum system requirements of the tool.

A.1 Using the Simulator

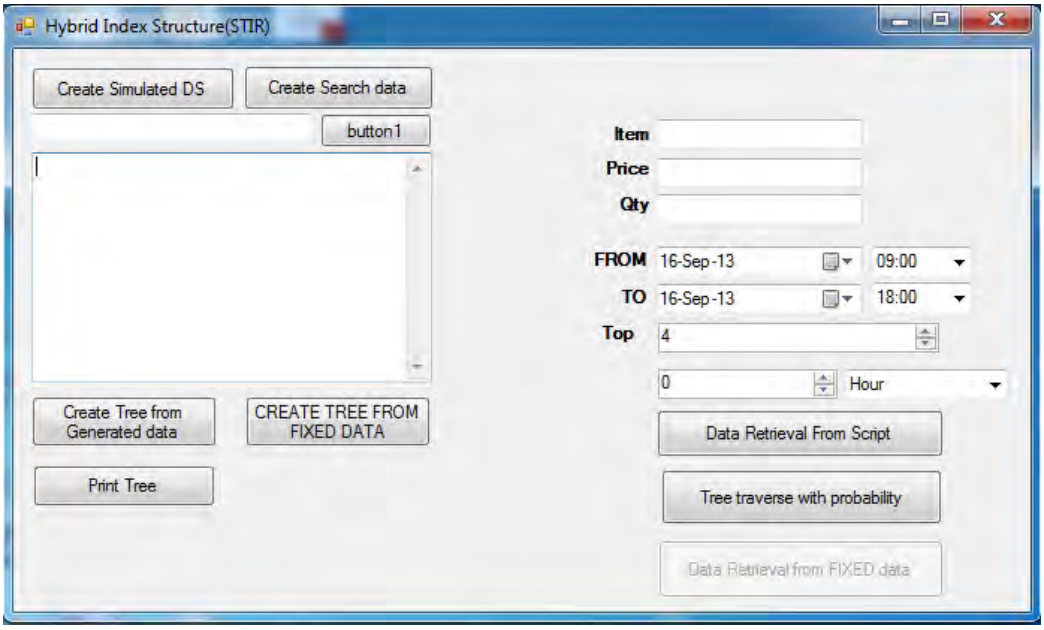


Figure A.1: STIR simulator

In order to execute the application the user has to STIR.exe application file of the software. The tool consist of several buttons with various function. "Create Simulated DS" button will create spatio-temporal dataset. "Create Search data" will create query set and "Data Retrieval from Script" will function as STK-kNN. Finally "Tree traverse with probability" will function as STK-kNN with time uncertainty or or STK-kNN with nearest neighbor time depending on the function calling.

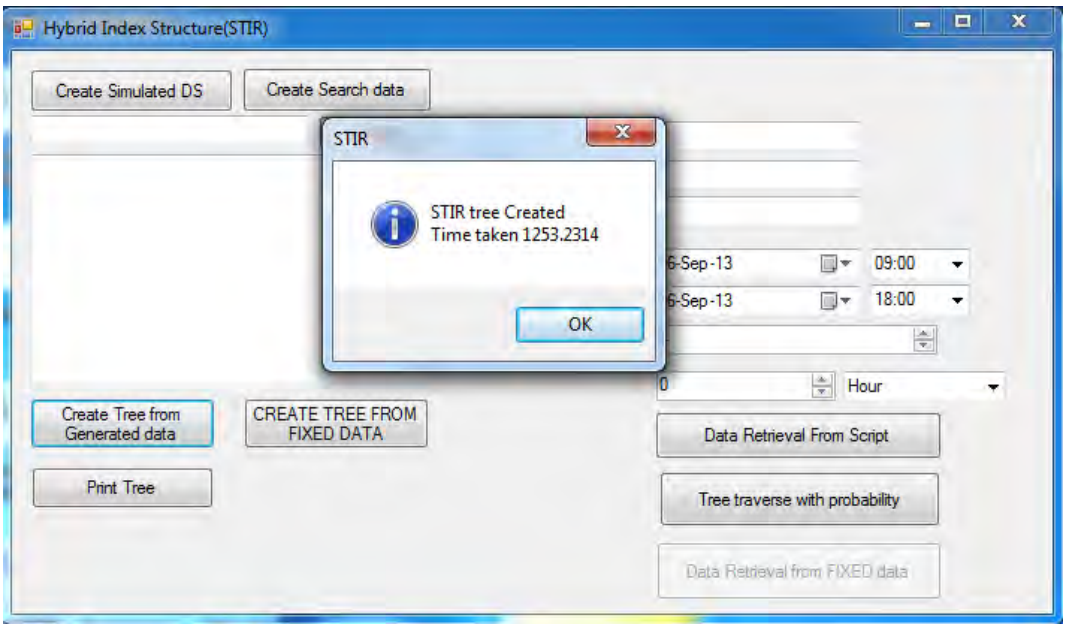


Figure A.2: A snapshot of creating STIR

A.1.1 Input

The user will have to select the number of expected result from dropdown list labeled as "Top" and create a .doc file with spatio-temporal dataset. To create data file (with .doc extension), user need to click "Create Simulated DS". By default, 10000 spatio-temporal dataset with uniformly distributed location will be created. Each data will be integrated with randomly chosen keyword and time from keyword and time list. Necessary spatio-temporal data is shown in Appendix B.

To create query set, user need to click "Create Search data" button which will create 1000 (by default) query set with uniformly distributed location. Each query set will be embedded with randomly chosen keyword and time. Necessary

query set is shown in Appendix C.

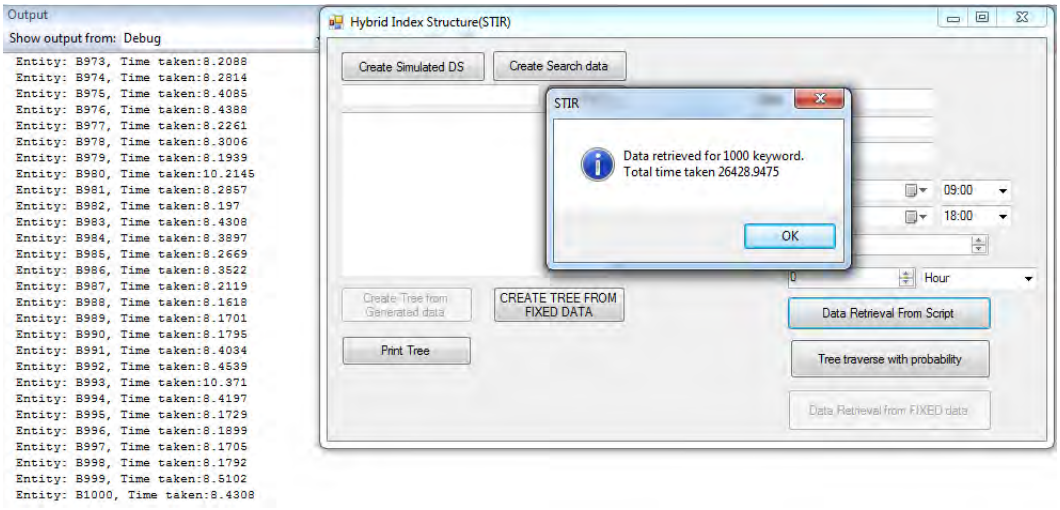


Figure A.3: A Snapshot of data retrieving using STK-kNN

A.1.2 Output

In order to create hybrid index structure, user need to click on "Create Tree from Generated data" as shown in Figure A.1. After clicking the button, hybrid index tree will be created and confirmation note come in the message box as shown in Figure A.2. STIR tree structure is shown in Appendix D.

To get the retrieved dataset user need to click "Data Retrieval from the Script". Software tool then will start retrieving data and measure total time and I/O cost for IR structure and STIR structure as shown in Figure A.3. Time and I/O will be recorded in separate file as shown in Appendix E. Retrieved dataset considering STK-kNN, STK-kNN with Time Uncertainty and STK-kNN with nearest neighbor time is shown in Appendix F, Appendix G, and Appendix H respectively.

A.2 System Requirements

STIR simulator is a windows based application that can be used in Windows based machines. For best user experience, we recommend the application to be used in a system configured with at least the following settings,

- Operating System: Windows XP.

- Processor:Intel dual core 2.66 GHz.
- RAM 2 GB.

Appendix B

Spatio-Temporal Data set

Our default data set size was 10000. We have shown our dataset (partial) as follow:

S1,1040,2000,1040,2000,0,0,Lemon:130:3;Pumpkin:150:6;,5/6/2012,0900,5/6/2012,1200
S2,612,688,612,688,0,0,Chilli:161:7;Carrot:180:6;,3/6/2012,1000,3/6/2012,1700
S3,1791,1667,1791,1667,0,0,Cabbage:192:7;Carrot:197:5;,27/3/2012,1600,27/3/2012,1700
S4,1336,1083,1336,1083,0,0,Cabbage:130:8;Cucumber:99:7;,7/1/2012,1600,7/1/2012,1700
S5,899,1629,899,1629,0,0,Lemon:108:4;Cucumber:183:8;,27/10/2012,0900,27/10/2012,1200
S6,19,425,19,425,0,0,Cabbage:196:4;Pumpkin:145:5;,3/10/2012,1800,3/10/2012,2000
S7,1104,291,1104,291,0,0,Tomatoe:113:9;Pumpkin:177:8;,13/6/2012,1100,13/6/2012,1500
S8,1205,1968,1205,1968,0,0,Lemon:54:8;Cucumber:97:7;,7/7/2012,1000,7/7/2012,1500
S9,134,1037,134,1037,0,0,Tomatoe:144:3;Pumpkin:72:4;,27/6/2012,0900,27/6/2012,1000
S10,972,290,972,290,0,0,Lemon:69:3;Pumpkin:69:4;,7/3/2012,1500,7/3/2012,1800
S11,931,618,931,618,0,0,Tomatoe:152:3;Carrot:129:4;,17/6/2012,1200,17/6/2012,1800
S12,1474,1456,1474,1456,0,0,Lemon:191:8;Cucumber:193:7;,3/7/2012,1000,3/7/2012,1100
S13,1672,349,1672,349,0,0,Cabbage:108:7;Carrot:122:8;,30/6/2012,1600,30/6/2012,1700
S14,166,1670,166,1670,0,0,Cabbage:55:4;Carrot:137:7;,9/1/2012,1000,9/1/2012,1800
S15,1316,1412,1316,1412,0,0,Lemon:97:4;Cucumber:188:5;,3/10/2012,1100,3/10/2012,2000
S16,1609,1032,1609,1032,0,0,Tomatoe:76:4;Cucumber:117:5;,19/1/2012,1800,19/1/2012,2000
S17,195,1382,195,1382,0,0,Tomatoe:182:9;Carrot:114:8;,1/7/2012,1100,1/7/2012,1700
S18,578,1945,578,1945,0,0,Tomatoe:117:3;Carrot:78:5;,13/6/2012,1100,13/6/2012,1700
S19,46,909,46,909,0,0,Tomatoe:126:9;Pumpkin:159:8;,21/1/2012,1000,21/1/2012,1100

S20,821,632,821,632,0,0,Lemon:121:4;Pumpkin:99:2;,1/6/2012,1500,1/6/2012,1800
S21,1577,818,1577,818,0,0,Chilli:85:5;Pumpkin:146:3;,21/7/2012,1100,21/7/2012,1700
....
S9973,8909,9692,8909,9692,0,0,Rice:136:4;Flour:122:7;,3/6/2012,1000,4/6/2012,2000
S9974,9271,9560,9271,9560,0,0,Potatoe:77:8;Pulse:188:6;,3/6/2012,1100,4/6/2012,2000
S9975,9367,8204,9367,8204,0,0,Flour:107:8;Potatoe:102:8;,3/6/2012,0900,4/6/2012,1200
S9976,9230,8371,9230,8371,0,0,Pulse:98:9;Salt:105:2;,3/6/2012,0900,3/6/2012,1200
S9977,8311,9285,8311,9285,0,0,Flour:52:7;Potatoe:126:3;,3/6/2012,0900,4/6/2012,1700
S9978,8714,8760,8714,8760,0,0,Pulse:148:3;Sugar:135:6;,4/6/2012,1000,4/6/2012,1200
S9979,8777,9304,8777,9304,0,0,Sugar:151:3;Salt:157:2;,3/6/2012,0900,4/6/2012,1500
S9980,9966,8007,9966,8007,0,0,Flour:99:3;Rice:190:4;,3/6/2012,1100,4/6/2012,1700
S9981,9048,9033,9048,9033,0,0,Salt:80:2;Potatoe:77:2;,3/6/2012,1200,4/6/2012,1800
S9982,8928,8737,8928,8737,0,0,Flour:138:9;Flour:196:8;,3/6/2012,1100,3/6/2012,1200
S9983,9827,8078,9827,8078,0,0,Flour:181:8;Flour:182:8;,3/6/2012,1600,4/6/2012,1700
S9984,8254,9191,8254,9191,0,0,Sugar:141:4;Salt:80:3;,3/6/2012,1600,4/6/2012,2000
S9985,9824,8070,9824,8070,0,0,Potatoe:106:2;Flour:140:4;,4/6/2012,1100,4/6/2012,1700
S9986,9208,9327,9208,9327,0,0,Potatoe:74:4;Pulse:167:2;,3/6/2012,1000,4/6/2012,1000
S9987,8220,8886,8220,8886,0,0,Rice:101:8;Pulse:65:5;,3/6/2012,0900,3/6/2012,1700
S9988,8361,8321,8361,8321,0,0,Flour:177:3;Pulse:89:7;,3/6/2012,0900,4/6/2012,2000
S9989,8568,9323,8568,9323,0,0,Sugar:130:8;Rice:111:7;,3/6/2012,1100,4/6/2012,1500
S9990,9007,9838,9007,9838,0,0,Flour:99:3;Rice:61:5;,3/6/2012,1000,4/6/2012,1800
S9991,8698,9951,8698,9951,0,0,Sugar:114:8;Potatoe:85:3;,3/6/2012,1200,4/6/2012,1800
S9992,9425,9017,9425,9017,0,0,Rice:189:3;Potatoe:79:5;,3/6/2012,0900,4/6/2012,1000
S9993,9278,9638,9278,9638,0,0,Rice:118:5;Flour:75:9;,3/6/2012,1700,4/6/2012,1800
S9994,9188,8314,9188,8314,0,0,Flour:62:8;Pulse:157:8;,3/6/2012,1100,4/6/2012,2000
S9995,8504,9570,8504,9570,0,0,Rice:134:7;Potatoe:55:2;,3/6/2012,1700,4/6/2012,1800
S9996,8144,9295,8144,9295,0,0,Flour:112:3;Sugar:150:2;,3/6/2012,1000,4/6/2012,1700
S9997,8999,8267,8999,8267,0,0,Pulse:155:6;Potatoe:101:6;,3/6/2012,1000,3/6/2012,1200
S9998,8947,8291,8947,8291,0,0,Sugar:148:2;Rice:153:4;,3/6/2012,1000,3/6/2012,2000
S9999,9544,9006,9544,9006,0,0,Sugar:127:2;Potatoe:92:7;,4/6/2012,1100,4/6/2012,1200
S10000,8498,8600,8498,8600,0,0,Potatoe:102:6;Flour:106:5;,3/6/2012,1700,4/6/2012,1800
End of file
18-Mar-13 11:25:23 AM

Appendix C

Search Query set

Our default number of search query set was 1000. We have shown search query set (partial) as follow:

B1,8,159,8,159,0,0,Pumpkin,101,2,4/3/2012,1200,9/3/2012,1600
B2,96,63,96,63,0,0,Cabbage,83,3,11/9/2012,1200,27/9/2012,1600
B3,29,162,29,162,0,0,Cabbage,86,5,20/4/2012,1600,20/4/2012,1700
B4,133,124,133,124,0,0,Pumpkin,185,9,16/11/2012,0900,25/11/2012,1500
B5,183,183,183,183,0,0,Lfinger,133,9,12/2/2012,1500,20/2/2012,1600
B6,79,141,79,141,0,0,Aubergine,169,8,15/6/2012,0900,16/6/2012,2000
B7,153,156,153,156,0,0,Aubergine,180,6,9/3/2012,0900,10/3/2012,1700
B8,190,65,190,65,0,0,Lemon,80,7,15/7/2012,0900,19/7/2012,1500
B9,183,186,183,186,0,0,Lfinger,51,8,15/7/2012,0900,22/7/2012,1200
B10,144,59,144,59,0,0,Spinach,131,5,20/2/2012,1100,22/2/2012,1200
B11,98,133,98,133,0,0,Lemon,152,3,2/7/2012,1000,25/7/2012,1000
B12,89,58,89,58,0,0,Aubergine,149,2,15/10/2012,0900,22/10/2012,1200
B13,62,193,62,193,0,0,Chilli,150,7,27/2/2012,1600,29/2/2012,1700
B14,61,190,61,190,0,0,Cabbage,82,8,18/11/2012,1000,22/11/2012,2000
B15,143,32,143,32,0,0,Lfinger,150,2,16/3/2012,1100,29/3/2012,1700
B16,4,51,4,51,0,0,Lemon,157,8,17/3/2012,0900,22/3/2012,1000
B17,198,9,198,9,0,0,Cabbage,171,5,1/3/2012,1000,28/3/2012,1200
B18,145,98,145,98,0,0,Aubergine,130,5,12/10/2012,1000,29/10/2012,1500
B19,70,196,70,196,0,0,Lfinger,146,4,1/2/2012,0900,7/2/2012,1700

B20,168,169,168,169,0,0,Aubergine,77,3,16/11/2012,1000,17/11/2012,1100
B21,170,154,170,154,0,0,Cabbage,160,2,14/2/2012,1000,22/2/2012,1000
B22,189,155,189,155,0,0,Cabbage,67,3,15/12/2012,1100,16/12/2012,1700
B23,106,40,106,40,0,0,Pumpkin,199,7,5/12/2012,1600,13/12/2012,1700
....
B974,995,938,995,938,0,0,Rice,186,6,23/11/2012,1800,27/12/2012,2000
B975,943,887,943,887,0,0,Salt,135,9,3/11/2012,1200,6/11/2012,1600
B976,864,960,864,960,0,0,Flour,85,8,4/12/2012,1500,28/12/2012,1600
B977,864,989,864,989,0,0,Rice,165,5,14/11/2012,0900,16/12/2012,1700
B978,974,950,974,950,0,0,Flour,180,4,8/11/2012,1100,26/11/2012,2000
B979,963,890,963,890,0,0,Sugar,104,4,10/11/2012,1100,22/12/2012,1500
B980,829,899,829,899,0,0,MilkPowder,84,8,2/11/2012,1000,15/11/2012,1800
B981,857,966,857,966,0,0,Salt,85,8,19/11/2012,1000,19/12/2012,1700
B982,876,873,876,873,0,0,Flour,61,6,8/11/2012,1700,29/12/2012,1800
B983,855,826,855,826,0,0,Salt,132,4,2/11/2012,1000,25/12/2012,1600
B984,948,883,948,883,0,0,Sugar,73,4,6/11/2012,0900,22/12/2012,1500
B985,843,989,843,989,0,0,Potatoe,193,8,15/12/2012,1500,27/12/2012,1800
B986,803,801,803,801,0,0,Rice,84,2,11/11/2012,1500,24/11/2012,1800
B987,957,864,957,864,0,0,Flour,199,9,11/12/2012,1600,13/12/2012,2000
B988,999,913,999,913,0,0,Salt,156,5,25/12/2012,1000,26/12/2012,1500
B989,877,892,877,892,0,0,Pulse,134,8,1/11/2012,1200,14/12/2012,1800
B990,892,938,892,938,0,0,Pulse,70,6,7/12/2012,1200,22/12/2012,1800
B991,921,808,921,808,0,0,Potatoe,140,9,12/11/2012,1200,16/11/2012,1800
B992,959,848,959,848,0,0,Sugar,101,7,7/11/2012,1200,12/12/2012,1800
B993,893,888,893,888,0,0,MilkPowder,63,5,8/11/2012,1800,20/12/2012,2000
B994,983,804,983,804,0,0,Salt,191,7,6/11/2012,1000,19/12/2012,1600
B995,927,837,927,837,0,0,Pulse,152,4,5/11/2012,0900,22/11/2012,1500
B996,841,985,841,985,0,0,Flour,99,2,3/11/2012,1000,27/12/2012,1500
B997,945,805,945,805,0,0,Sugar,113,2,6/11/2012,1100,13/12/2012,1700
B998,897,918,897,918,0,0,Pulse,132,5,8/11/2012,0900,13/12/2012,1000
B999,823,950,823,950,0,0,Rice,138,3,11/12/2012,1600,25/12/2012,1700
B1000,965,980,965,980,0,0,Salt,128,3,1/11/2012,1800,13/12/2012,2000
End of file
23-Mar-13 11:48:48 AM

Appendix D

Structure of STIR tree

Our STIR tree (partial) created from simulated data is shown below:

Simulated STIR Tree created on:18-Mar-13 11:31:32 AM

Object:(11, 162, 0), (11, 162, 0) Level:0 From:21-03-12 1000 Hrs To:21-03-12 1600 Hrs TimeTag:Hr TimeKeeper:10,16, Item:chilli:145:5,cucumber:176:8,
Object:(22, 378, 0), (22, 378, 0) Level:0 From:15-10-12 1100 Hrs To:15-10-12 1200 Hrs TimeTag:Hr TimeKeeper:11,12, Item:chilli:73:9,carrot:197:6,tomatoe:148:3,
Object:(25, 429, 0), (25, 429, 0) Level:0 From:15-10-12 1000 Hrs To:15-10-12 1800 Hrs TimeTag:Hr TimeKeeper:10,18, Item:chilli:158:6,carrot:78:2,
Object:(9, 66, 0), (9, 66, 0) Level:0 From:21-06-12 1200 Hrs To:21-06-12 1600 Hrs TimeTag:Hr TimeKeeper:12,16, Item:cabbage:61:7,cucumber:169:5,lemon:168:9,
Object:(9, 545, 0), (9, 545, 0) Level:0 From:27-03-12 1000 Hrs To:27-03-12 1200 Hrs TimeTag:Hr TimeKeeper:10,12, Item:chilli:168:5,pumpkin:121:5,
Object:(19, 425, 0), (19, 425, 0) Level:0 From:03-10-12 1800 Hrs To:03-10-12 2000 Hrs TimeTag:Hr TimeKeeper:18,20, Item:cabbage:196:4,pumpkin:145:5,cabbage:121:6,
Object:(2, 207, 0), (2, 207, 0) Level:0 From:03-03-12 1000 Hrs To:03-03-12 1100 Hrs TimeTag:Hr TimeKeeper:10,11, Item:tomatoe:162:6,carrot:137:8,
Object:(22, 71, 0), (22, 71, 0) Level:0 From:03-03-12 0900 Hrs To:03-03-12 1500 Hrs TimeTag:Hr TimeKeeper:9,15, Item:lemon:126:7,pumpkin:162:9,
Object:(6, 318, 0), (6, 318, 0) Level:0 From:19-10-12 1600 Hrs To:19-10-12 2000 Hrs TimeTag:Hr TimeKeeper:16,20, Item:tomatoe:164:5,cucumber:104:6,

Object:(24, 522, 0), (24, 522, 0) Level:0 From:19-10-12 0900 Hrs To:19-10-12 2000 Hrs TimeTag:Hr TimeKeeper:9,20, Item:chilli:131:7,pumpkin:52:5,
Object:(18, 26, 0), (18, 26, 0) Level:0 From:03-03-12 0900 Hrs To:03-03-12 1500 Hrs TimeTag:Hr TimeKeeper:9,15, Item:tomatoe:197:9,cucumber:90:8,cabbage:164:3,
Object:(16, 299, 0), (16, 299, 0) Level:0 From:13-01-12 0900 Hrs To:13-01-12 1000 Hrs TimeTag:Hr TimeKeeper:9,10, Item:tomatoe:124:6,pumpkin:142:8,lemon:73:2,

---

Node:(2, 26, 0), (25, 545, 0) Level:1 TimeTag:Dy TimeKeeper:3,13,15,19,21,27, EntryCount:13 Entries:[(11, 162, 0), (11, 162, 0)];[(22, 378, 0), (22, 378, 0)];[(25, 429, 0), (25, 429, 0)];[(9, 66, 0), (9, 66, 0)];[(9, 545, 0), (9, 545, 0)];[(19, 425, 0), (19, 425, 0)];[(2, 207, 0), (2, 207, 0)];[(22, 71, 0), (22, 71, 0)];[(6, 318, 0), (6, 318, 0)];[(14, 239, 0), (14, 239, 0)];[(24, 522, 0), (24, 522, 0)];[(18, 26, 0), (18, 26, 0)];[(16, 299, 0), (16, 299, 0)]; Inverted List:chilli:145:5:(11, 162, 0), (11, 162, 0),cucumber:176:8:(11, 162, 0), (11, 162, 0),chilli:73:9:(22, 378, 0), (22, 378, 0),carrot:197:6:(22, 378, 0), (22, 378, 0),tomatoe:148:3:(22, 378, 0), (22, 378, 0),pumpkin:76:9:(22, 378, 0), (22, 378, 0),chilli:158:6:(25, 429, 0), (25, 429, 0),carrot:78:2:(25, 429, 0), (25, 429, 0),cabbage:61:7:(9, 66, 0), (9, 66, 0),cucumber:169:5:(9, 66, 0), (9, 66, 0),lemon:168:9:(9, 66, 0), (9, 66, 0),cucumber:52:9:(9, 66, 0), (9, 66, 0),chilli:168:5:(9, 545, 0), (9, 545, 0),pumpkin:121:5:(9, 545, 0), (9, 545, 0),cabbage:196:4:(19, 425, 0), (19, 425, 0),pumpkin:145:5:(19, 425, 0), (19, 425, 0),cabbage:121:6:(19, 425, 0), (19, 425, 0),pumpkin:78:9:(19, 425, 0), (19, 425, 0),tomatoe:162:6:(2, 207, 0), (2, 207, 0),carrot:137:8:(2, 207, 0), (2, 207, 0),lemon:126:7:(22, 71, 0), (22, 71, 0),pumpkin:162:9:(22, 71, 0), (22, 71, 0),tomatoe:164:5:(6, 318, 0), (6, 318, 0),cucumber:104:6:(6, 318, 0), (6, 318, 0),lemon:108:9:(14, 239, 0), (14, 239, 0),cucumber:75:6:(14, 239, 0), (14, 239, 0),chilli:131:7:(24, 522, 0), (24, 522, 0),pumpkin:52:5:(24, 522, 0), (24, 522, 0),tomatoe:197:9:(18, 26, 0), (18, 26, 0),cucumber:90:8:(18, 26, 0), (18, 26, 0),cabbage:164:3:(18, 26, 0), (18, 26, 0),cucumber:88:7:(18, 26, 0), (18, 26, 0),tomatoe:124:6:(16, 299, 0), (16, 299, 0),pumpkin:142:8:(16, 299, 0), (16, 299, 0),lemon:73:2:(16, 299, 0), (16, 299, 0),pumpkin:141:2:(16, 299, 0), (16, 299, 0),

Level:2

Node:(467, 439, 0), (807, 2000, 0) Level:2 TimeTag:Mnth TimeKeeper:1,3,6,7,10,

EntryCount:12 Entries:[(567, 1511, 0), (807, 1636, 0)];[(469, 705, 0), (528, 1079, 0)];[(641, 458, 0), (661, 1168, 0)];[(693, 1658, 0), (796, 2000, 0)];[(531, 857, 0), (610, 1177, 0)];[(486, 1204, 0), (562, 1554, 0)];[(467, 1642, 0), (617, 1998, 0)];[(667, 444, 0), (694, 1084, 0)];[(564, 1236, 0), (684, 1510, 0)];[(605, 1665, 0), (736, 1969, 0)];[(467, 439, 0), (622, 688, 0)];[(537, 712, 0), (633, 840, 0)]; Inverted List:lemon:186:5:(567, 1511, 0), (807, 1636, 0),carrot:181:7:(567, 1511, 0), (807, 1636, 0),chilli:57:4:(567, 1511, 0), (807, 1636, 0),carrot:123:8:(567, 1511, 0), (807, 1636, 0),lemon:135:7:(567, 1511, 0), (807, 1636, 0),cucumber:153:5:(567, 1511, 0), (807, 1636, 0),cabbage:109:4:(567, 1511, 0), (807, 1636, 0),pumpkin:189:7:(567, 1511, 0), (807, 1636, 0),chilli:99:8:(567, 1511, 0), (807, 1636, 0),carrot:147:6:(567, 1511, 0), (807, 1636, 0),tomatoe:156:5:(567, 1511, 0), (807, 1636, 0),cucumber:60:5:(567, 1511, 0), (807, 1636, 0),chilli:132:7:(567, 1511, 0), (807, 1636, 0),cucumber:179:2:(567, 1511, 0), (807, 1636, 0),chilli:67:8:(567, 1511, 0), (807, 1636, 0),cucumber:133:9:(567, 1511, 0), (807, 1636, 0),lemon:66:4:(567, 1511, 0), (807, 1636, 0),pumpkin:85:9:(567, 1511, 0), (807, 1636, 0),cabbage:72:7:(567, 1511, 0), (807, 1636, 0),pumpkin:188:6:(567, 1511, 0), (807, 1636, 0),lemon:77:3:(567, 1511, 0), (807, 1636, 0),pumpkin:53:2:(567, 1511, 0), (807, 1636, 0),cabbage:88:4:(567, 1511, 0), (807, 1636, 0),carrot:183:6:(567, 1511, 0), (807, 1636, 0),lemon:102:7:(567, 1511, 0), (807, 1636, 0),cucumber:98:2:(567, 1511, 0), (807, 1636, 0),cabbage:65:5:(567, 1511, 0), (807, 1636, 0),cucumber:140:4:(567, 1511, 0), (807, 1636, 0),lemon:96:5:(567, 1511, 0), (807, 1636, 0),cucumber:110:5:(567, 1511, 0), (807, 1636, 0),tomatoe:164:9:(567, 1511, 0), (807, 1636, 0),cucumber:90:9:(567, 1511, 0), (807, 1636, 0),chilli:130:8:(567, 1511, 0), (807, 1636, 0),pumpkin:90:4:(567, 1511, 0), (807, 1636, 0),lemon:116:2:(567, 1511, 0), (807, 1636, 0),pumpkin:109:4:(567, 1511, 0), (807, 1636, 0),tomatoe:126:3:(567, 1511, 0), (807, 1636, 0),carrot:65:4:(567, 1511, 0), (807, 1636, 0),lemon:181:5:(567, 1511, 0), (807, 1636, 0),cucumber:63:7:(567, 1511, 0), (807, 1636, 0),cabbage:56:2:(567, 1511, 0), (807, 1636, 0),cucumber:81:8:(567, 1511, 0), (807, 1636, 0),chilli:94:5:(567, 1511, 0), (807, 1636, 0),carrot:177:3:(567, 1511, 0), (807, 1636, 0),cabbage:94:5:(567, 1511, 0), (807, 1636, 0),carrot:99:5:(567, 1511, 0), (807, 1636, 0),tomatoe:94:5:(567, 1511, 0), (807, 1636, 0),pumpkin:69:4:(567, 1511, 0), (807, 1636, 0),lemon:191:4:(469, 705, 0), (528, 1079, 0),pumpkin:67:5:(469, 705, 0), (528, 1079, 0),cabbage:191:5:(469, 705, 0), (528, 1079, 0),cucumber:98:9:(469, 705, 0), (528, 1079, 0),lemon:156:4:(469, 705, 0), (528, 1079, 0),pumpkin:81:3:(469, 705, 0), (528, 1079, 0),chilli:75:3:(469,

705, 0), (528, 1079, 0),carrot:55:3:(469, 705, 0), (528, 1079, 0),lemon:78:3:(469,
705, 0), (528, 1079, 0),carrot:158:7:(469, 705, 0), (528, 1079, 0),chilli:124:5:(469,
705, 0), (528, 1079, 0),pumpkin:123:4:(469, 705, 0), (528, 1079, 0),lemon:181:5:(469,
705, 0), (528, 1079, 0),carrot:199:2:(469, 705, 0), (528, 1079, 0),lemon:153:4:(469,
705, 0), (528, 1079, 0),carrot:71:5:(469, 705, 0), (528, 1079, 0),tomatoe:195:7:(469,
705, 0), (528, 1079, 0),carrot:121:3:(469, 705, 0), (528, 1079, 0),chilli:170:6:(469,
705, 0), (528, 1079, 0),cucumber:183:5:(469, 705, 0), (528, 1079, 0),cabbage:84:5:(469,
705, 0), (528, 1079, 0),pumpkin:117:5:(469, 705, 0), (528, 1079, 0),lemon:104:8:(469,
705, 0), (528, 1079, 0),carrot:188:4:(469, 705, 0), (528, 1079, 0),cabbage:181:5:(469,
705, 0), (528, 1079, 0),cucumber:126:7:(469, 705, 0), (528, 1079, 0),cabbage:148:6:(469,
705, 0), (528, 1079, 0),pumpkin:68:2:(469, 705, 0), (528, 1079, 0),chilli:80:8:(469,
705, 0), (528, 1079, 0),cucumber:79:6:(469, 705, 0), (528, 1079, 0),tomatoe:194:3:(469,
705, 0), (528, 1079, 0),cucumber:110:9:(469, 705, 0), (528, 1079, 0),tomatoe:116:9:(469,
705, 0), (528, 1079, 0),pumpkin:83:8:(469, 705, 0), (528, 1079, 0),tomatoe:115:2:(469,
705, 0), (528, 1079, 0),carrot:194:3:(469, 705, 0), (528, 1079, 0),chilli:147:3:(469,
705, 0), (528, 1079, 0),carrot:53:7:(469, 705, 0), (528, 1079, 0),tomatoe:61:3:(469,
705, 0), (528, 1079, 0),pumpkin:64:9:(469, 705, 0), (528, 1079, 0),lemon:76:6:(469,
705, 0), (528, 1079, 0),pumpkin:149:7:(469, 705, 0), (528, 1079, 0),lemon:65:6:(469,
705, 0), (528, 1079, 0),pumpkin:94:7:(469, 705, 0), (528, 1079, 0),cabbage:143:4:(469,
705, 0), (528, 1079, 0),cucumber:180:6:(469, 705, 0), (528, 1079, 0),cabbage:153:4:(469,
705, 0), (528, 1079, 0),pumpkin:52:3:(469, 705, 0), (528, 1079, 0),cabbage:100:4:(469,
705, 0), (528, 1079, 0),carrot:143:4:(469, 705, 0), (528, 1079, 0),cabbage:121:8:(469,
705, 0), (528, 1079, 0),cucumber:117:4:(469, 705, 0), (528, 1079, 0),tomatoe:132:7:(469,
705, 0), (528, 1079, 0),carrot:104:3:(469, 705, 0), (528, 1079, 0),lemon:54:9:(469,
705, 0), (528, 1079, 0),cucumber:58:4:(469, 705, 0), (528, 1079, 0),tomatoe:52:3:(469,
705, 0), (528, 1079, 0),pumpkin:164:5:(469, 705, 0), (528, 1079, 0),tomatoe:86:7:(469,
705, 0), (528, 1079, 0),pumpkin:59:5:(469, 705, 0), (528, 1079, 0),cabbage:135:3:(641,
458, 0), (661, 1168, 0),carrot:50:7:(641, 458, 0), (661, 1168, 0),tomatoe:187:3:(641,
458, 0), (661, 1168, 0),pumpkin:87:3:(641, 458, 0), (661, 1168, 0),lemon:103:8:(641,
458, 0), (661, 1168, 0),pumpkin:109:7:(641, 458, 0), (661, 1168, 0),cabbage:175:7:(641,
458, 0), (661, 1168, 0),carrot:167:9:(641, 458, 0), (661, 1168, 0),chilli:130:5:(641,
458, 0), (661, 1168, 0),carrot:97:2:(641, 458, 0), (661, 1168, 0),tomatoe:134:5:(641,
458, 0), (661, 1168, 0),cucumber:70:8:(641, 458, 0), (661, 1168, 0),chilli:101:7:(641,
458, 0), (661, 1168, 0),pumpkin:146:7:(641, 458, 0), (661, 1168, 0),tomatoe:141:5:(641,

458, 0), (661, 1168, 0),carrot:116:3:(641, 458, 0), (661, 1168, 0),tomatoe:141:3:(641, 458, 0), (661, 1168, 0),carrot:86:8:(641, 458, 0), (661, 1168, 0),tomatoe:83:9:(641, 458, 0), (661, 1168, 0),carrot:57:5:(641, 458, 0), (661, 1168, 0),tomatoe:58:3:(641, 458, 0), (661, 1168, 0),cucumber:137:9:(641, 458, 0), (661, 1168, 0),cabbage:159:9:(641, 458, 0), (661, 1168, 0),pumpkin:121:7:(641, 458, 0), (661, 1168, 0),tomatoe:182:2:(641, 458, 0), (661, 1168, 0),cucumber:198:3:(641, 458, 0), (661, 1168, 0),cabbage:109:9:(641, 458, 0), (661, 1168, 0),cucumber:197:6:(641, 458, 0), (661, 1168, 0),tomatoe:198:5:(641, 458, 0), (661, 1168, 0),cucumber:174:9:(641, 458, 0), (661, 1168, 0),cabbage:194:2:(641, 458, 0), (661, 1168, 0),pumpkin:74:2:(641, 458, 0), (661, 1168, 0),lemon:141:6:(641, 458, 0), (661, 1168, 0),cucumber:65:9:(641, 458, 0), (661, 1168, 0),lemon:180:2:(641, 458, 0), (661, 1168, 0),pumpkin:88:9:(641, 458, 0), (661, 1168, 0),cabbage:193:3:(641, 458, 0), (661, 1168, 0),cucumber:181:4:(641, 458, 0), (661, 1168, 0),cabbage:124:4:(641, 458, 0), (661, 1168, 0),pumpkin:165:8:(641, 458, 0), (661, 1168, 0),tomatoe:169:8:(693, 1658, 0), (796, 2000, 0),cucumber:142:3:(693, 1658, 0), (796, 2000, 0),chilli:71:9:(693, 1658, 0), (796, 2000, 0),cucumber:193:5:(693, 1658, 0), (796, 2000, 0),lemon:109:8:(693, 1658, 0), (796, 2000, 0),carrot:105:5:(693, 1658, 0), (796, 2000, 0),cabbage:111:8:(693, 1658, 0), (796, 2000, 0),carrot:79:2:(693, 1658, 0), (796, 2000, 0),chilli:96:5:(693, 1658, 0), (796, 2000, 0),carrot:136:9:(693, 1658, 0), (796, 2000, 0),cabbage:71:4:(693, 1658, 0), (796, 2000, 0),carrot:180:9:(693, 1658, 0), (796, 2000, 0),lemon:141:7:(693, 1658, 0), (796, 2000, 0),carrot:134:2:(693, 1658, 0), (796, 2000, 0),chilli:83:5:(693, 1658, 0), (796, 2000, 0),carrot:118:3:(693, 1658, 0), (796, 2000, 0),tomatoe:164:7:(693, 1658, 0), (796, 2000, 0),pumpkin:83:9:(693, 1658, 0), (796, 2000, 0),tomatoe:195:3:(693, 1658, 0), (796, 2000, 0),carrot:83:6:(693, 1658, 0), (796, 2000, 0),tomatoe:153:4:(693, 1658, 0), (796, 2000, 0),cabbage:99:9:(693, 1658, 0), (796, 2000, 0),chilli:156:2:(693, 1658, 0), (796, 2000, 0),carrot:134:6:(693, 1658, 0), (796, 2000, 0),chilli:192:3:(693, 1658, 0), (796, 2000, 0),pumpkin:180:3:(693, 1658, 0), (796, 2000, 0),tomatoe:125:3:(693, 1658, 0), (796, 2000, 0),carrot:165:2:(693, 1658, 0), (796, 2000, 0),cabbage:82:4:(693, 1658, 0), (796, 2000, 0),cucumber:50:6:(693, 1658, 0), (796, 2000, 0),chilli:86:8:(693, 1658, 0), (796, 2000, 0),carrot:126:5:(693, 1658, 0), (796, 2000, 0),tomatoe:122:9:(693, 1658, 0), (796, 2000, 0),pumpkin:60:6:(693, 1658, 0), (796, 2000, 0),cabbage:185:2:(693, 1658, 0), (796, 2000, 0),pumpkin:167:4:(693, 1658, 0), (796, 2000, 0),tomatoe:149:2:(693, 1658, 0), (796, 2000, 0),carrot:194:6:(693, 1658, 0), (796, 2000, 0),cabbage:74:5:(693, 1658, 0), (796, 2000, 0),cucumber:81:6:(693, 1658, 0), (796, 2000, 0),tomatoe:161:6:(693, 1658,

0), (796, 2000, 0),cucumber:123:4:(693, 1658, 0), (796, 2000, 0),lemon:112:9:(693, 1658, 0), (796, 2000, 0),cucumber:63:9:(693, 1658, 0), (796, 2000, 0),chilli:81:9:(693, 1658, 0), (796, 2000, 0),cucumber:160:4:(693, 1658, 0), (796, 2000, 0),tomatoe:68:3:(531, 857, 0), (610, 1177, 0),pumpkin:130:3:(531, 857, 0), (610, 1177, 0),lemon:131:9:(531, 857, 0), (610, 1177, 0),pumpkin:189:2:(531, 857, 0), (610, 1177, 0),lemon:53:3:(531, 857, 0), (610, 1177, 0),cucumber:152:6:(531, 857, 0), (610, 1177, 0),cabbage:54:3:(531, 857, 0), (610, 1177, 0),cucumber:54:2:(531, 857, 0), (610, 1177, 0),cabbage:87:4:(531, 857, 0), (610, 1177, 0),cucumber:173:5:(531, 857, 0), (610, 1177, 0),tomatoe:113:8:(531, 857, 0), (610, 1177, 0),cucumber:148:8:(531, 857, 0), (610, 1177, 0),lemon:143:3:(531, 857, 0), (610, 1177, 0),carrot:198:2:(531, 857, 0), (610, 1177, 0),tomatoe:199:8:(531, 857, 0), (610, 1177, 0),cucumber:139:5:(531, 857, 0), (610, 1177, 0),chilli:182:9:(531, 857, 0), (610, 1177, 0),carrot:175:8:(531, 857, 0), (610, 1177, 0),lemon:187:5:(531, 857, 0), (610, 1177, 0),cucumber:117:7:(531, 857, 0), (610, 1177, 0),chilli:102:3:(531, 857, 0), (610, 1177, 0),cucumber:113:3:(531, 857, 0), (610, 1177, 0),tomatoe:140:5:(531, 857, 0), (610, 1177, 0),cucumber:137:9:(531, 857, 0), (610, 1177, 0),tomatoe:196:4:(531, 857, 0), (610, 1177, 0),cucumber:177:7:(531, 857, 0), (610, 1177, 0),tomatoe:117:5:(531, 857, 0), (610, 1177, 0),cucumber:143:7:(531, 857, 0), (610, 1177, 0),chilli:168:2:(531, 857, 0), (610, 1177, 0),carrot:137:9:(531, 857, 0), (610, 1177, 0),chilli:139:5:(486, 1204, 0), (562, 1554, 0),cucumber:144:2:(486, 1204, 0), (562, 1554, 0),tomatoe:177:5:(486, 1204, 0), (562, 1554,

... ...

Level:4

Node:(1, 2, 0), (9999, 10000, 0) Level:4 TimeTag:Mnth TimeKeeper:1,3,6,7,10, EntryCount:4 Entries:[(2001, 2003, 0), (4271, 4845, 0)];[(1, 2, 0), (2328, 3164, 0)];[(4002, 4001, 0), (8000, 6230, 0)];[(6000, 6220, 0), (9999, 10000, 0)]; Inverted List:turmeric:162:3:(2001, 2003, 0), (4271, 4845, 0),pepper:190:2:(2001, 2003, 0), (4271, 4845, 0),ginger:139:4:(2001, 2003, 0), (4271, 4845, 0),chilli:191:5:(2001, 2003, 0), (4271, 4845, 0),garlic:168:3:(2001, 2003, 0), (4271, 4845, 0),poppyseed:163:9:(2001, 2003, 0), (4271, 4845, 0),onion:90:5:(2001, 2003, 0), (4271, 4845,

... ...

Appendix E

Performance Measurement Data set

Following results show as an example of performance measurement data set which is used for plotting the graph of IR structure and STIR structure

K vs TIME,I/O

IR
SearchData:1000,I/O:15937,Timetaken:594.511587400001,K=1,
SearchData:1000,I/O:19804,Timetaken:749.8406787,K=2,
SearchData:1000,I/O:21186,Timetaken:941.4657049,K=4,
SearchData:1000,I/O:23974,Timetaken:1630.9242325,K=8,
SearchData:1000,I/O:29667,Timetaken:4550.5868404,K=16,
SearchData:1000,I/O:34591,Timetaken:5545.6849,K=20,

STIR

SearchData:1000,I/O:4457,Timetaken:337.1293412,K=1,
SearchData:1000,I/O:4517,Timetaken:340.4078773,K=2,
SearchData:1000,I/O:4577,Timetaken:343.537919800001,
SearchData:1000,I/O:5550,Timetaken:357.3227812,K=8,
SearchData:1000,I/O:6618,Timetaken:373.218177,K=16,
SearchData:1000,I/O:7130,Timetaken:4550.5868404,K=20,

SQ vs Time,I/O

IR

SearchData:500,I/O:10401,Timetaken:465.4484085,K=4,

SearchData:1000,I/O:21186,Timetaken:941.4657049,K=4,

SearchData:1500,I/O:30935,Timetaken:1316.6032043,K=4,

SearchData:2000,I/O:41905,Timetaken:1793.9051933,K=4,

STIR

SearchData:500,I/O:2401,Timetaken:174.7503981,K=4,

SearchData:1000,I/O:4577,Timetaken:343.537919800001,K=4,

SearchData:1500,I/O:6667,Timetaken:487.5905545,K=4,

SearchData:2000,I/O:8643,Timetaken:677.233157899999,K=4,

DS vs time,I/O

STIR

SearchData.:5000,I/O:4577,Timetaken:343.537919800001,K=4,

SearchData.:10000,I/O:6407,Timetaken:851.4727888,K=4,

SearchData.:15000,I/O:5645,Timetaken:1641.696207,K=4,

SearchData.:20000,I/O:7133,Timetaken:2455.6977427,K=4,

Appendix F

STK-kNN set

Retrieved Top-7 spatio-temporal data set for the search data "*B28,61,17,61,17,0,0, Cabbage,192,5,01/7/2012,1000,30/7/2012,1000*" is shown below:

Entity:S1428,Loc:(50, 91, 0), (50, 91, 0)
Distance:74.813102722168
chilli:108:6,pumpkin:92:4,cabbage:105:8,carrot:53:6,
From:27-07-12 11:00
To:27-07-12 15:00

Entity:S1212,Loc:(69, 168, 0), (69, 168, 0)
Distance:151.211776733398
cabbage:87:6,carrot:189:9,lemon:52:3,pumpkin:170:4,
From:15-07-12 11:00
To:15-07-12 20:00

Entity:S1045,Loc:(200, 128, 0), (200, 128, 0)
Distance:177.881988525391
chilli:111:6,carrot:149:7,cabbage:118:6,carrot:87:2,
From:19-07-12 10:00
To:19-07-12 20:00

Entity:S955,Loc:(254, 173, 0), (254, 173, 0)
Distance:248.16325378418

lemon:108:2,carrot:178:4,cabbage:137:8,pumpkin:77:9,
From:21-07-12 11:00
To:21-07-12 15:00

Entity:S1543,Loc:(316, 53, 0), (316, 53, 0)
Distance:257.528625488281
cabbage:98:2,pumpkin:64:4,cabbage:73:7,pumpkin:143:7,
From:09-07-12 09:00
To:09-07-12 15:00

Entity:S261,Loc:(105, 380, 0), (105, 380, 0)
Distance:365.656951904297
cabbage:108:9,carrot:65:7,lemon:146:5,carrot:53:6,
From:13-07-12 09:00
To:13-07-12 15:00

Entity:S1005,Loc:(483, 163, 0), (483, 163, 0)
Distance:446.542266845703
tomatoe:191:2,cucumber:179:7,cabbage:119:8,cucumber:196:5,
From:05-07-12 09:00
To:05-07-12 15:00

Entity:S1996,Loc:(498, 128, 0), (498, 128, 0)
Distance:450.876922607422
lemon:197:5,pumpkin:107:3,cabbage:51:3,carrot:196:6,
From:05-07-12 10:00
To:05-07-12 20:00

Entity:S921,Loc:(332, 378, 0), (332, 378, 0)
Distance:451.400054931641
cabbage:105:7,cucumber:58:6,
From:21-07-12 10:00
To:21-07-12 15:00

Appendix G

STK-kNN with Time Uncertainty set

Retrieved 10 expected spatio-temporal data set with probability score for the search data "*B28,17,19,17,19,0,0, Cabbage,192,5,1/5/2012,1300,1/5/2012,1300*" is shown below:

S891,(7, 24, 0), (7, 24, 0)
Distance:11.18034
cabbage:87:6,carrot:189:9,lemon:52:3,
Score:0.125
01-05-12 12:00 01-05-12 16:00

S330,(55, 44, 0), (55, 44, 0)
Distance:45.48626
tomatoe:191:2,cucumber:179:7,cabbage:119:8,
Score:0.2296875
01-05-12 17:00 01-05-12 18:00

S603,(66, 38, 0), (66, 38, 0)
Distance:52.55473
chilli:108:6,pumpkin:92:4,cabbage:105:8,
Score:0.0328125
01-05-12 09:00 01-05-12 17:00

S15,(95, 23, 0), (95, 23, 0)
Distance:78.10249
lemon:197:5,pumpkin:107:3,cabbage:51:3,carrot:196:6,
Score:0.0328125
01-05-12 10:00 01-05-12 18:00

S993,(72, 79, 0), (72, 79, 0)
Distance:81.3941
cabbage:105:7,cucumber:58:6,
Score:0.2296875
01-05-12 17:00 01-05-12 18:00

S42,(56, 103, 0), (56, 103, 0)
Distance:92.61209
pumpkin:92:4,cabbage:105:8,
Score:0.375
01-05-12 10:00 01-05-12 12:00

S60,(113, 34, 0), (113, 34, 0)
Distance:97.16481
pumpkin:107:3,cabbage:51:3,carrot:196:6,
Score:0.0189824380165289
01-05-12 09:00 01-05-12 20:00

S28,(51, 119, 0), (51, 119, 0)
Distance:105.622
Score:0.075
01-05-12 10:00 01-05-12 15:00

S629,(117, 64, 0), (117, 64, 0)
Distance:109.6586
cabbage:50:4,
Score:0.0189824380165289

01-05-12 09:00 01-05-12 20:00

S982,(146, 50, 0), (146, 50, 0)
Distance:132.6725
tomatoe:191:2,cucumber:179:7,cabbage:119:8,
Score:0.375
01-05-12 10:00 01-05-12 11:00

Query point date:01-05-12 12:00 01-05-12 12:00
End of file
Print on:13-Jul-13 10:25:59 PM

Appendix H

STK with Time based Nearest Neighbor set

Retrieved 6 expected spatio-temporal data set based on nearest neighbor time
for the search data "*B28,17,19,17,19,0,0, Cabbage,192,5,1/5/2012,0900,1/5/2012,0900*"
is shown below:

S1669,(1941, 1950, 0), (1941, 1950, 0)
Distance:2725.90112304688
pumpkin:92:4,cabbage:105:8,
01-05-12 10:00 01-05-12 17:00

S1781,(1915, 1847, 0), (1915, 1847, 0)
Distance:2635.14477539063
tomatoe:191:2,cucumber:179:7,cabbage:119:8,
01-05-12 11:00 01-05-12 15:00

S1332,(1895, 1941, 0), (1895, 1941, 0)
Distance:2687.18579101563
lemon:197:5,pumpkin:107:3,cabbage:51:3,carrot:196:6,
01-05-12 11:00 01-05-12 15:00

S1437,(1908, 1960, 0), (1908, 1960, 0)
Distance:2709.86376953125

cabbage:87:6,carrot:189:9,lemon:52:3,
01-05-12 15:00 01-05-12 16:00

S1877,(1949, 1966, 0), (1949, 1966, 0)
Distance:2742.8876953125
chilli:108:6,pumpkin:92:4,cabbage:105:8,
01-05-12 16:00 01-05-12 20:00

S1590,(1957, 1770, 0), (1957, 1770, 0)
Distance:2613.3505859375
cabbage:87:6,carrot:189:9,lemon:52:3,
01-05-12 18:00 01-05-12 20:00

Query point date:01-05-12 10:00 01-05-12 10:00
End of file
Print on:17-Sep-13 11:06:13 PM
Total data:6

Index