Geo-Skip List Data Structure – Implementation and Solving Spatial Queries

Mansi A. Radke
Visvesvaraya National Institute of Technology,
Nagpur, India
mansiaradke@gmail.com

Vaibhav Varshney
3D PLM Software
Pune, India
vaibhavballi@gmail.com

Umesh A. Deshpande
Visvesvaraya National Institute of Technology,
Nagpur, India
uadeshpande@gmail.com

Abstract—A major portion of the queries fired on the internet have spatial keywords in them, the storage and retrieval of spatial data has become an important task in today's era. Given a geographic query that is composed of query keywords and a location, a geographic search engine retrieves documents that are the most textually and spatially relevant to the query keywords and the location, respectively, and ranks the retrieved documents according to their joint textual and spatial relevance to the query. The lack of an efficient index that can simultaneously handle both the textual and spatial aspects of the documents makes existing geographic search engines inefficient in answering geographic queries. There are data structures which facilitate storage and retrieval of geographical data like R-trees, R* trees, KD trees etc. We propose Geo-Skip list data structure which is also one such data structure which is inspired from the skip list data structure. It is simple, dynamic, partly deterministic and partly randomized data structure. This structure brings out the hierarchy of administrative divisions of a region very well. Also it shows an improvement in the search efficiency as compared with R-trees. In this paper, we propose algorithms for the implementation of basic spatial queries with the help of Geo-Skip List data structure – namely, point query, range query, finding the nearest neighbour query and kth nearest neighbour query.

Keywords*-Spatial data, complexity, skip list, hierarchy, geo-skip list, efficiency, R-tree, kth nearest neighbor*

*****

I. Introduction

A spatial data structure is used for storage and efficient access of geographical data. R-trees [2] and its variants like R+ and R* trees are most commonly used to store spatial data. In a k-way R-tree with k entries in every node, the leaf node entry consists of two fields, the first field is a minimum bounding rectangle of a geographical region and the second field points to the data or attributes of the region. An entry of an internal node of the tree also has two fields, the first field is the minimum bounding rectangle such that it encloses the rectangles of all the entries of its children nodes and the second field is a pointer to a child node. A major disadvantage which we found in using these structures for accessing spatial/geographical data is about finding geographical hierarchy of a geoname. For example, the hierarchy of New York City is Continent - North America, Country - United States of America, and State - New York. However, the internal nodes of an R-tree represent a fictitious bounding rectangle rather than a true entity in the geographical hierarchy. So, when a query - "Find the hierarchy of New York City" is given, the R-tree will find the minimum bounding rectangles which encompass NewYork City rather than the hierarchy as stated above.

Typically a geographical location is represented as a tuple <latitude, longitude>. The types of queries most commonly observed over the web are:

1. Find the hierarchy of a place
2. Find all places (e.g. restaurants, malls etc.) within say "x" km radius of a place
3. Find a place whose coordinates are given as say (x, y)
4. List all the subdivisions falling under list of all states of country India.

We need a structure that nicely represents hierarchy and at the same time also has an efficient search complexity. In this work, we propose a data structure, called Geo-skip list, for storing, indexing and searching the geographical places in the world. In our structure, each node corresponds to a real entity and the concept of bounding rectangles is removed. The data structure is inspired from skip lists proposed by Pugh [3]. One important feature of this data structure is that it is partly deterministic and partly randomized and gives efficient search time complexity. In Geo-skip list, the search is fast and keeps on focusing progressively. It overcomes the complex procedures like node splitting, balancing and restructuring which is necessary in R-trees. Also this data structure is overly simple with the salient feature that it brings out the hierarchy naturally. Other salient features of Geo-Skip List are that it is easy to add, delete new entry from Geo-skip list without extra overhead of splitting and merging. Not that for answering queries like - "Find all cities in India" Geo-skip list is better structure to use. Also we prove that the geo-skip list gives a search time that is comparable to R-tree.

The paper is structured as follows. The next section reviews the currently existing data structures for spatial data. This is followed by section 3 in which we explain the proposed approach in detail. The experimental results are presented in section 4. Section 5 concludes the paper and points out some directions for future work.

II. Literature Reeview

While searching for documents over the web, quite often a user is interested in getting the information about some entity specific to a location. For example, a user may fire a query say "Schools in Nagpur city". To address this, we need to create an index for both textual and spatial information using the following two approaches.

a. Use separate indices for both textual and spatial information.
b. Use a hybrid index[10] for both textual and spatial index i.e. inverted file on top of R tree, R tree on top of inverted file, KR* tree [14] , IR tree[8] etc.

In the first approach we used separate indices for the textual and the spatial information. In this, the result for a query containing geonames is obtained by the following steps:

i. Retrieve textually relevant documents by searching on the textual index using the textual keywords.
ii. Filter the obtained result obtain from step one through a spatial index (R-tree, kd-tree, quad-tree etc.) by matching the spatial keywords present in the query.
iii. Rank the documents obtained.

Drawback of above approach is that the results obtained from step one might have many irrelevant results since it uses only text and does not consider the location.

In the second approach, we combine textual keywords and spatial location of query term together to create an index. For the above example, we combine the textual and spatial keyword to form a new word like school_nagpur where the prefix indicates a textual keyword followed by the location or geoname. Now inverted index will be created on school_nagpur to get relevant results. On the other hand we can create hybrid index by combining textual and spatial index, like inverted file on R tree called Hybrid$_I$ and R tree on top of inverted file called Hybrid$_R$. For searching results in Hybrid$_I$, search is first made on textual keyword on inverted index and then search will be made on location keyword based on the R tree. In case of Hybrid$_R$, it is vice a versa.

Various approaches used to store spatial data can be classified as - Tree base methods[11], and Space filling methods [12] [13].

Tree base methods can further subdivided into two types - space partitioning and data partitioning. In the space partitioning approach, the space is divided into tiles and in data partitioning the data object is divided into subsets. Examples of the space partitioning method are KD-tree and quad tree. A drawback of these algorithms is that a data object might span across the border between tiles and hence it needs to be stored twice. This problem occurs when the data objects are very large. The most common data-partitioning approach is R-tree family of indexing. It stores each object only once but has problem of node merging and splitting.

Space filling methods plot the regions on a continuous curve. The idea behind this approach is that set of points which are close to each other in space will also be close to each other on the curve. The most widely used space filling algorithm is Z-curve filling and Hilbert curve filling algorithm.

R-tree has good average logarithmic time complexity for searching any location. R tree is B tree based structure and it defines minimum and maximum number of entries in the node. If the number of entries goes beyond the maximum limit specified, then node splitting needs to be done. The splitting needs to be done in such a way that it should form two minimum sized rectangles that contain all the entries which were present in the overflow node. If we want optimum solution for splitting then there is Exhaustive algorithm where we try all the possible combination of rectangles and select the optimum sized rectangles. The number of possible feasible solutions for this approach will be $2^{M-1}$where M is the maximum number of entries in the page.There are other approaches for splitting with less cost like Quadratic cost algorithm where total cost is quadratic in M and linear in the number of dimensions. Linear cost algorithm which is linear in M.But these algorithms do not guarantee optimal size of rectangles.

There are few variants of R tree which try to solve these problems by minimizing both coverage and overlapping. Examples of such trees are R* Tree, R+ Tree.

The structure which is based on binary tree based indexing technique is KD-tree. KD-tree is an extension of binary tree where discriminating attribute is used to divide entries. Discriminating attribute will be different at different levels of tree. Suppose the data is two dimensional, then at the root node data will be divided into two parts depending on the value of the X co-ordinate and in next subsequent level, entries will be divided into its left and right child depending on the values of Y co-ordinates. KD-tree has a problem when internal node of tree gets deleted. Say node ‘P’ is an internal node in the tree and ‘P’ is deleted then ‘P’ should be replaced by one of sub tree whose root is ‘P’. Let ‘j’ be the discriminating attribute of node ‘P’. Then replacement would be either a node which is in the right sub tree with smallest value of jth attribute or a node which is in left sub tree with largest value of jth attribute. This replacement may also cause many successive replacements .

As mentioned earlier, R-trees do not represent the hierarchy very well. The rectangles encompassing a particular region/regions is a fictitious rectangle, and does not represent an entity in the overall hierarchy. In order to overcome the disadvantages of the existing approaches, we propose a new data structure called Geo-skip list for storage, indexing of spatial data and efficient search of geonames. The proposed data structure is explained in the following section.

III. PROPOSED WORK

A. Geo-Skip List Data structure

The proposed data structure, Geo-skip list, consists of multiple levels of linked structures. It is a combination of a linear list and a skip list. Every level represents a hierarchy of a geographical region. The world is divided into administrative groups such as continents, countries, states etc. Hence, we have defined five levels - continents, countries, states, cities, and localities. Every level can potentially be represented as a collection of skip lists. However, since the number of entries at the first three levels is less, these can be represented as simple linked lists. Minimum number of entries required to represent a level using a skip list can be treated as a design parameter. For the Getty database [10], which we have used for experimentation, we have represented the continents (less than 10), the countries (around 200), and the states (around 1000), as simple linked lists. The other two levels of cities and localities are represented using a collection of skip lists. This makes the data structure partly deterministic and party randomized, since skip lists are randomized structures.

The data structure is shown in the figure 1. Here the maroon colored nodes represent continents, green ones represent countries, yellow nodesrepresent states, dark blue ones represent cities in the worldand light blue nodes are the localities in various cities. Every node in the linked lists (i.e. at the first three levels) contains the following fields –

i) The minimum latitude, the maximum latitude, the minimum longitude and the maximum longitude of the region.
ii) Right and left pointer point tothe next and previous node in the linked list at the same level.
iii) Down pointer to point to a node in the list which is one level below in the hierarchy.

For example, the right pointer in the ‘America Continent’ node points to ‘Europe Continent’ node. The down pointer in the ‘America Continent’ node points to the first node that represents a country in ‘America Continent’.

The last two levels are represented by a collection of skip lists as explained below. Every node in a skip list consists of the following fields.

i) The minimum latitude, the maximum latitude, the minimum longitude and the maximum longitude of the region.
ii) Right and left pointer which points to a node at the same sub-level of the skip list. A skip list itself is made up of multiple sub-levels. These sub-levels should not be confused with the levels of the Geo-skip list data structure itself.
iii) Down pointerwhich points to a node at a lower sublevel of the skip list. Thelast node at the lowest sublevel of a skip list points back to anode at the above level of the Geo-skip list.

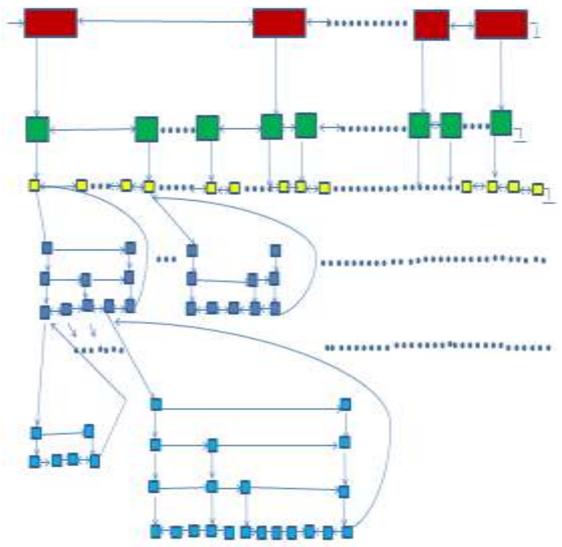

Figure1: Geo skip list data structure

B. *Inserting a new location into the Geo-Skip List*

The Geo-skip list data structure will be updated very rarely as the places in the world and their information do not change frequently. Also, when we need to insert a new location, we assume that we know its hierarchy. For e.g. if we want to add a state say ‘Illinois’, we would already have the information with us that this belongs to country USA of continent North America. The following algorithm is used for insertion.

1) Suppose we need to add new location ‘P’ into Geo-skip list. ‘P’ will get added to appropriate level depending on information about that location.

2) If ‘P’ is continent then start searching for position for adding ‘P’ from start (Starting node in continent). The continents are stored in a linear link list. To add ‘P’ in linear list, traverse the list from start, using the right pointers in the node till,

    i. Right pointer is not equal to null and
    ii. least latitude value of node is less than least latitude value of ‘P’ or
    iii. the node having same least latitude value but its least longitude value is less than ‘P’.

   Once we get such node, add ‘P’ to the right of that node if its right pointer is equal to null and its latitude value is less than ‘P’ or having same latitude value but its longitude value less than ‘P’ else add ‘P’ to the left of that node.

3) If ‘P’ is country (level two location) then find out to which continent it belongs. The countries are stored in a linear link list. Linearly search for that continent in level one and follow down pointer of that continent. Then in level two search for the position of ‘P’ in the same way as we did in level 1. Once we get the required position, add ‘P’ as in step 2.

4) If ‘P’ is a state (level three location) then the steps similar to step 3 are executed.

5) If ‘P’ is a district/city (level four location), then find out from which state, country and continent it belongs. Suppose we need to add district/city ‘Chicago’ in Geo-skip list. City ‘Chicago’ belongs to continent ‘North America’, country ‘United states of America (USA)’ and state ‘Illinois’. To add ‘Chicago’ in Geo-skip list we follow following steps.

    i. Traverse the first level list till we find the continent North America.
    ii. In the node for North America, follow the down pointer and start traversing the list. Keep following the right pointer till we find the country USA.
    iii. Now, follow the down pointer and start traversing the list till we get State Illinois.
    iv. In Illinois follow the down pointer and start traversing the skip list. First check for sublevel information of node, if sublevel of node is not equal to 1 then keep following the right pointer, to locate the appropriate position of the node. Once the appropriate location is found, follow the down pointer of

node. Repeat above procedure till sublevel of node is not equal to 1.

v. If sublevel is '1', we follow the right pointer till right pointer is not equal to 'USA' and find the appropriate place for the node to be inserted. When this is located, we insert Chicago at appropriate position. If Chicago is the last node of sublevel one then we set its right pointer to Maharashtra (third level entry).

vi. Now we toss a coin, and decide whether to promote Chicago to higher sub-levels in the skip list. If the value is yes, we create a copy of Chicago and start with Illinois and follow down pointer of node till we reach sublevel 2 i.e. (previous sublevel + 1) and insert the node. We again toss a coin, and if we need to promote Chicago to level 3, we repeat the same procedure and this goes on. We might even have to create a new sublevel if the coin toss keeps promoting Chicago to higher and higher levels.

6) If 'P' is any locality/town/village of any district/city then it needs to be added in level five as per the steps similar to those of 5).

C. *Searching a location in the Geo-Skip List*

Suppose Geo-skip list is ready to use and we want to search a particular location 'P' in Geo-skip list then we need to follow following steps.

1) Start Searching from leftmost node of the first level of the Geo-skip list. Keep comparing node value with the value of 'P' in each level of Geo-skip list, if it is same then return that node information. Follow the right pointer and locate the appropriate position using the key of comparison. When we find the appropriate node where we can search 'P', follow its down pointer.

2) In second and third level repeat the same steps that we followed in 1.

3) Compare the key values of the node with those of all the entries in level four. If it is same, then return else follow skip list search technique at this level. Check for sublevel information of node if it is not equal to one then follow right pointer of node till right pointer of node is not equal to null and <least latitude, least longitude> pair of node is less than <least latitude, least longitude> pair of 'P'. If we find 'P' then return else follow down pointer of appropriate node. Repeat this procedure till sublevel of node is not equal to one. If sublevel of node is equal to one then follow right pointer of node till right pointer of node is not equal to state (level three entry) of that node and remaining comparisons are same. If we find that entry then return else follow down pointer.

4) If location was not found in level four then search in level five. Repeat the same search step as of searching in level four.

5) If we did not find the location yet then go back to district/city where we have searched last and check if we can search in the previous node of that district/city node. If yes then search for all the localities of that district/city. If we find 'P' then return else go back to district/city node and repeat same procedure for node previous to current district/city. Jump to upper level if we did get that entry in present level of Geo-skip list.

6) Repeat back-searching of node till threshold number of times which is 4. Every time we search in level five increments the number of attempts made for searching entry 'P' in Geo-skip list by one. If search condition fails while following back pointer then jump to the upper level of Geo-skip list.

If the number of attempts equals to threshold value then return that location was not found.

D. Deletion of location from geo skip list

1) Let 'P' be the entry which we want to delete from Geo-skip list.
2) Search for 'P' by using searching algorithm for any location in Geo-skip list.
3) If the location was not found then return.
4) If location was found then check the level to which it belongs. If it is in level one, two or three then delete that entry from Geo-skip list. If there is an entry previous to 'P' then set its right pointers to the right pointer of 'P'.

If 'P' is from level four or five then then delete 'P' from all the sublevels in that level. If there is an entry previous to 'P' then set its right pointer to the right pointer of 'P' in all the sublevels.

We explain the search procedure in a later section. However, to briefly explain the working of our structure let us consider the following example. Suppose we want to find the hierarchy of a locality named 'Times Square'. We first find the latitude and longitude from a thesaurus and use it for search. The search always begins from the top leftmost corner. It traverses the list sequentially using the right pointers, until the latitude of the place lies in between the least and most latitude of the current node being traversed. If it does, then the down pointer is followed. The next level is searched in the same manner and when the latitude value lies in between the least and most latitude of the current node, the down pointer is followed. The process is repeated until the node is found. The node at each level at which the down pointer is followed is remembered and that gives the hierarchy of the place. When we travel right (with the right pointer) in this data structure, we are essentially walking over the world map from left to right. When we travel down (with a down pointer) we are going one level down in the geographical hierarchy. The search complexity is linear at the first three levels (since they are simple linked lists). At the lower two levels, the search complexity is logarithmic with respect to the number of nodes present in a particular linked list (since a skip list ensures logarithmic search complexity with high probability [3]). The search will require the complexity of the order of number of continents + number of countries + number of states + log(number of cities in the particular state) + log(number of localities in the particular city). Since, the number of entries in level four and level five are much more than total number of entries in first three levels, we can say that total time complexity for any operation on Geo-skip list is logarithmic. In the worst case, when there is overlap between two regions, then more than one node needs to be searched at a particular level and the complexity will increase by a constant factor. The search using Geo-skip list is fast as it keeps on focusing progressively. Apart from this, it overcomes the complexprocedures like node splitting, balancing and restructuring which are necessary in R-trees. Moreover, a Geo-skip list brings out the hierarchy naturally.

For example, the right pointer in the 'America Continent' node points to 'Europe Continent' node. The down pointer in the 'America Continent' node points to the first node that represents a country in 'America Continent'.

The last two levels are represented by a collection of skip lists as explained below. Every node in a skip list consists of the following fields.

i) The minimum latitude, the maximum latitude, the minimum longitude and the maximum longitude of the region.

ii) Right and left pointer which points to a node at the same sub-level of the skip list. A skip list itself is made up of multiple sub-levels. These sub-levels should not be confused with the levels of the Geo-skip list data structure itself.

iii) Down pointer which points to a node at a lower sublevel of the skip list. The last node at the lowest sublevel of a skip list points back to a node at the above level of the Geo-skip list.

E. Geo-Skip List on Disk

Structure like R tree uses one complete page size to store entries in one node. R tree is not a completely main memory structure it uses disk space to store its information. There are various approaches to store objects on disk and retrieve them. We use serialization to store our Geo-skip list on the disk. We calculate the size of single node in Geo-skip list and depending on that we find out the maximum number of nodes that we can safely store in given free main memory.

Let maximum number of nodes that we can store in given free main memory is max_nodes. We start building our list from level 1, i.e. the continents and go on adding data for each level. Once we reach level 3 i.e. level at which states are stored, we serialize the data. After adding state entry, we create index for all districts/city in that particular state and then all the localites/towns of that district/city. We calculate total number of nodes required for creation of this index and if we can store this information in available space i.e. number of nodes is less than the max_nodes, then we serialize this object and store it on disk. If we cannot store the complete state information in one file then we store all objects in level five (localities) of particular state in different file after checking their size requirement. While adding entries in Geo-skip list, we take a record of total number of nodes which we have added. If we add one complete structure (complete skip list) in Geo-skip list and count is less than max_nodes then we store that structure on disk. For retrieving Geo-skip list structure, we need to use object file for a state entry from a disk if that state information stored in single file. If the state information stored in different files on disk then we need to use particular file from a disk depend on key of comparison of that entry.

Suppose we are not storing objects on disk file. In that case we calculate total amount of main memory used by Geo-skip list. Total number of continents in the word is 7. Continents having maximum number of country is Africa having 53 countries. Different countries are sub-divided differently depending on their administrative subdivisions. We will be using respective subdivision of a country at corresponding level in Geo-skip list. Countries like India have state and territories in its first level, districts in second level and cities in third level. Russia has federal subject in level one, district in level two and rural in level three. Romania has counties in level one, communes in level two and villages in level three. We treat division of all countries at respective level in the same way in a Geo-skip list i.e. State, Federal Subject, Counties of India, Russia and Romania respectively will be come in third level of Geo-skip list. Depending on their latitude and longitude information entries will get sorted in respective levels.

The country having maximum number of states or the country having maximum number of subdivision at first level is Russia. Russia is divided into 83 Federal Subjects. In level four of Geo-skip list maximum number of entries is present for Country Romania. Romania is divided into 41 Counties in level one. County (First level division of Romania) Suceava of Romania has maximum number of districts 113. In level five of Geo-skip list we are storing the information about city/districts of different states. Spain is administratively divided into Autonomous communities at first level then it gets divided into Provenance at level second then into Municipalities at level third. These municipalities will be at level five of Geo-skip list. Municipalities of Spain have the maximum number of entries than any other entries in level five of Geo-Skip list. Burgos Provenance of Spain having total 371 numbers of Municipalities. Division of each locality i.e. level four division of any country is very less. Each locality is generally divided into few (10 to 20) villages and tehsil/town will have some important locations like Hospital, Bus-stop, Library, Railway Station, Temple, Theatre, Market Area, Damp, River, and Lake etc. For a place like New York City, the number of such important landmarks is likely to be very high.

Suppose we have 'M Bytes' free RAM in our system. When we store data of 'N' nodes in a linked list, then space utilized to store these 'N' data node will be C1*'N' bytes where C1 is the number of bytes required to store every node. When we store 'N' data items in skip list then total 'N' data item will be present in level one and higher level will also contain few elements. We are using non-deterministic approach for storing data item in higher level with probability of 0.5 that the element will be promoted to next higher level. We can say that total 'N/2' elements will present in level two. 'N/4' elements will be present in level three and so on. If we add all this values then we will get total 2*N elements we are adding into skip list.

Total number of entry stored in skip list for storing 'N' data node will be

$$= \sum (N + (N/2) + (N/4) + (N/8) + (N/16) \ \ldots)$$
$$= N \sum (1 + ½ + ¼ + 1/8 + 1/16 \ldots\ldots)$$
$$= N * 2$$

Now we can calculate the maximum number of nodes that we can store in Geo-skip list after level three such that memory required by all these objects will not exceed the total free main memory size. Once we reach to the maximum number we will store that (state or city) object on disk. Following equation will calculate maximum number of nodes that we can store on disk after level three of Geo-skip list without exceeding total free main memory size.

$$C1*C2*(113+113*(371*C3+371*C3)) <= M$$

Where,
C1: Memory in bytes required by a node in Geo-skip list.
C2: Maximum number of nodes that we in Geo-skip list without exceeding total free main memory.
C3: Maximum number of important location in any localities of Geo-skip list.
M: Total free main memory size in Bytes.

F. *Answering point query with Geo-Skip List*

For answering point query, suppose we need to search for a location P in Geo-Skip list we always start searching from starting node i.e. leftmost node of level one (continent). As the data is stored in secondary memory, for searching node 'P' in geo-skip list we need to follow following steps:

1. Coordinates of the point P say <Latp,Longp> is fetched from the query.

2. Relevant portion of the index from the is fetched from secondary memory and each index entry is directly inserted in doubly linked list (modified skip list structure below level three) in main memory.

3. Compare present level nodes which are fetched from the secondary memory and check if the Maximum Bounding Rectangle range encompasses the point P.

   a. If node range encompasses point P (we call it as a probable node), then fetch the next level index for the present parent node and repeat step 3 till level 4 is reached.

   b. Else traverse to the next node on the present level and repeat step 3 till all the nodes fetched on present level are checked once and then follow the backward pointer to the parent node recursively till all the probable nodes are traversed.

4. From level 4 skip list pattern is used for storing the data, so from this level we need to follow skip list searching technique. Check if the present node or next node is the required point we are looking for,

   a. If required point P is found then return all the information about this point and skip to the end.

   b. Else if the latitude of point P lies in between latitudes of these two nodes then follow the down pointer of first node and repeat step 4 on the sublevel of skip list, else follow the right pointer and repeat step 4 till all the nodes are checked once and then follow the backward pointer to the parent node recursively till all the probable nodes are traversed.

5. If point P is not yet found in the fetched data stored in skip list pattern then traverse back using the backward pointer to the parent node on the previous level and repeat step 3.

6. If no such point P is found after traversing all the probable nodes, then return result that location is not found.

G. *Answering range query with Geo-skip List*

For answering range query, suppose we need to search for locations falling in a particular range x around a location point P in Geo-Skip list, we always start searching from starting node in level one (continent). All probable parent/ancestor nodes to P will be first searched in level one (continent) then level two (country) then level three (state) and so on and node P will be searched among child nodes of the probable ancestor nodes. As the data is stored in secondary memory, for searching node 'P' in geo-skip list we need to follow following steps:

1. Coordinates of given point say <Latp, Longp> and range 'x'is fetched from the query.

2. Relevant portion of the index from the is fetched from secondary memory and each index entry is directly inserted in doubly linked list (modified skip list structure below level three) in main memory.

3. Compare present level nodes which are fetched from the secondary memory and check if even one of the following conditions stands true, a. Point P lies in MBR of the present node b. Perpendicular distance calculated using haversine formula between latitude of P and latitude of either minimum or maximum coordinate of present node is less than or equal to given range x. Then the present node is a probable node, then we fetch the next level index for the present parent node and repeat step 3 till level 4 is reached.

4. If no condition mentioned in step 3 stands true then follow the right pointer to the next node and repeat step three. If all probable nodes in present level are traversed then follow the back pointer to the parent level repeat step 3 with the right pointer node.

5. From level 4 skip list pattern is used for storing the data, so from this level we need to follow skip list searching technique. On this level we only have coordinates of the locations and no MBR coordinates. Compare present skip list sub level node and check perpendicular distance calculated using haversine formula between latitude of P and latitude of coordinate of present node.

   a. If the calculated distance falls in range x then follow the down pointer of the node till the last sublevel of skip list is reached.

   b. Else follow the right pointer to the present node in present sublevel of skip list. If all probable nodes in skip list are traversed then follow the back pointer to the parent level and then repeat step 3 with the right pointer node.

6. On last sublevel of skip list find distance between point P and all fetched nodes

a. If distance falls in range x, then output the location pointed by present node, follow the right pointer of the present node and repeat step 6.

b. Else follow the right pointer of the present node and repeat step 6. If all nodes in this sublevel of skip list are traversed then follow the back pointer to the parent level repeat step 3 with the right pointer node.

7. If no known location in given range to point P is found after traversing all the probable nodes, then return result that no locations exist in the given range.

H. Answering nearest neighbour query with Geo-Skip List

For answering nearest neighbor query, we need to search for a location which is nearest in terms of distance to point P in Geo-Skip list. Firstly we need to find the point P in Geo-Skip list. We have used algorithm for answering point query to find the point P. The node P and its nearest neighbor will be searched among child nodes of the probable ancestor nodes. As the data is stored in secondary memory, for searching node ‘P’ and its nearest neighbor in geo-skip list we need to follow following steps:

1. Coordinates of given point say <Latp, Longp> fetched from the query.

2. Relevant portion of the index from the is fetched from secondary memory and each index entry is directly inserted in doubly linked list (modified skip list structure below level three) in main memory.

3. Compare present level nodes which are fetched from the secondary memory and check if the Maximum Bounding Rectangle range encompasses the point P.

   a. If node range encompasses point P (we call it as a probable node), then fetch the next level index for the present parent node and repeat step 3 till level 4 is reached.

   b. Else traverse to the next node on the present level and repeat step 3 till all the nodes fetched on present level are checked. Then follow the backward pointer to the parent node recursively till all the probable nodes are traversed.

4. From level 4 skip list pattern is used for storing the data, so from this level we need to follow skip list searching technique. Check if the present node or next node is the required point we are looking for,

   a. If required point P is found skip to step 6.

   b. Else,

      i. If the latitude of point P lies in between latitudes of present and its next right node then follow the down pointer of first node and repeat step 4 on the sublevel of skip list.

      ii. Else follow the right pointer and repeat step 4 till all the nodes are checked once and then follow the backward pointer to the parent node recursively till all the probable nodes are traversed.

5. If point P is not yet found in the fetched data stored in skip list pattern then traverse back using the backward pointer to the parent node on the previous level and repeat step 3. If no such point P is found after traversing all the probable nodes, then return result that location is not found.

6. Follow the left and right pointer of the point P and calculate distance between P and the left and right nodes of P. Store the minimum between distance calculated and previous Min. If P does not have a left or right pointer, then skip to step 10.

7. Calculate the perpendicular distance between latitudes of P and its left and right nodes and store minimum value lat_Min.

8. If Min is the distance between P and its left node, then repeat step 6 and 7 with point P, left of the current_left node and current_right node. Else with point P, current_left and right of the current_right node, while lat_Min < Min on both sides of node p.

9. Traverse back to the parent level and calculate the perpendicular distance between latitude of P and consecutive left and right node to the previously visited left and right nodes of the parent node (If there are no previously visited left or right node then parent node will be considered as left and right node) and store minimum values lat_Min_left and lat_Min_right. a. If, lat_Min_left < Min, then connect the skip lists of left node and parent node. Repeat step 6 and 7 with point P, left of the current_left and current_right node. Similar steps will be followed if lat_Min_right < Min. While lat_Min_left < Min && lat_Min_right < Min. b. Else, skip to step 10.

10. Return that node in the output which is at distance Min from point P.

11. If point P does not have a node to its left, then follow the back pointer to the previous level and check for node to the left of the parent node.

    a. If a node present to the left of the parent node, then find out the extreme right node in the consecutive children levels and connect the rightmost node in lowest sublevel of skip list to the left of node P. Repeat step 6.

    b. Else follow the back pointer to the previous level and repeat step 7.

Similar steps are followed if point P does not have a node to its right.

I. Answering kth nearest neighbour query with Geo-Skip List

For answering Kth nearest neighbor query, we need to search for location which is Kth nearest in terms of distance to point P in Geo-Skip list. Firstly we need to find the point P in Geo-Skip list and then we will find 1st nearest neighbor of point P, then 2nd nearest neighbor and so on. We have used algorithm for answering nearest neighbor query to search for consecutive K nearest neighbors of point P. As the data is stored in secondary memory, for searching node 'P' and its nearest neighbor in geo-skip list we need to follow following steps:

1. Coordinates of given point say <Latp, Longp> fetched from the query.

2. Relevant portion of the index from the is fetched from secondary memory and each index entry is directly inserted in doubly linked list (modified skip list structure below level three) in main memory.

3. Compare present level nodes which are fetched from the secondary memory and check if the Maximum Bounding Rectangle range encompasses the point P.

   a. If node range encompasses point P (we call it as a probable node), then fetch the next level index for the present parent node and repeat step 3 till level 4 is reached.

   b. Else traverse to the next node on the present level and repeat step 3 till all the nodes fetched on present level are checked. Then follow the backward pointer to the parent node recursively till all the probable nodes are traversed.

4. From level 4 skip list pattern is used for storing the data, so from this level we need to follow skip list searching technique. Check if the present node or next node is the required point we are looking for,

   a. If required point P is found skip to step 6.

   b. Else,

      i. If the latitude of point P lies in between latitudes of present and its next right node then follow the down pointer of first node and repeat step 4 on the sublevel of skip list.

      ii. Else follow the right pointer and repeat step 4 till all the nodes are checked once and then follow the backward pointer to the parent node recursively till all the probable nodes are traversed.

5. If point P is not yet found in the fetched data stored in skip list pattern then traverse back using the backward pointer to the parent node on the previous level and repeat step 3. If no such point P is found after traversing all the probable nodes, then return result that location is not found.

6. Start a counter K. Follow the left and right pointer to the point P and calculate distance between the P and left and right nodes to P. Store the minimum between distance calculated and previous Min. If either of the left or right nodes are not present to point P then skip to step 11.

7. Calculate the perpendicular distance between latitudes of P and its left and right nodes and store minimum value lat_Min.

8. If Min is the distance between P and its left node, then repeat step 6 and 7 with point P, left of the current_left node and current_right node. Else with point P, current_left and right of the current_right node, while lat_Min < Min on both sides of node p.

9. Traverse back to the parent level and calculate the perpendicular distance between latitude of P and consecutive left and right node to the previously visited left and right nodes of the parent node (If there are no previously visited left or right node then parent node will be considered as left and right node) and store minimum values lat_Min_left and lat_Min_right.

   a. If, lat_Min_left < Min, then connect the skip lists of left node and parent node. Repeat step 6 and 7 with point P, left of the current_left and current_right node. Similar steps will be followed if lat_Min_right < Min. While lat_Min_left < Min && lat_Min_right < Min.

   b. Else, skip to step 10.

10. Decrement the counter K by one. Exclude the node which is at a distance Min from P. Reset the Min to minimum distance of the left and right marked nodes of the point P. Go back to step 6. Repeat this step till the counter reaches zero.

11. When counter reaches zero, return the node in output which is at distance Min from point P.

12. If point P does not have a node to its left, then follow the back pointer to the previous level and check for node to the left of the parent node.

    a. If a node present to the left of the parent node, then find out the extreme right node in the consecutive children levels and connect the rightmost node in lowest sublevel of skip list to the left of node P. Repeat step 6.

    b. Else follow the back pointer to the previous level and repeat step 7.

Similar steps are followed if point P does not have a node to its right.

IV. EXPERIMENTATION

We have compared Geo-skip list data structure with R tree for disk and main memory implementation.Data which we used for comparing Geo-skip list with R tree is random data generated by us and Getty TGN database. The data of minimum and maximum latitude and longitude is missing in TGN database for many entries as the population of this information is underway. For experimentation, we have currently used self-generated approximate data for minimum and maximum latitude and longitude of node.

Table1: JSI Library (Main memory implementation of R tree) VS Main memory implementation of Geo-skip list.

Sr. No.	Operation	Time taken by Geo-skip list (microsecond)	Time taken by JSI Library (microsecond)
1.	Searching for an entry present at the last level. (location inside District)	239.325	823.065
2.	Searching for an entry Present at top level. (Continent)	118.999	896.927
3.	Searching For location which is not present.	159.783	843.445
4.	Time for creating index.	1284000	137000

Table2: Disk Implementation of R tree VS Disk Implementation of Geo-skip List.

Sr. No.	Operation	Time taken by Geo-skip list (millisecond)	Time taken by R tree (millisecond)
1.	Searching for an entry present at the last level. (location inside District)	87.477253	64.010905
2.	Searching for an entry present at the 2nd last level. (District)	84.668321	46.762833
2.	Searching for an entry Present at top level. (Continent)	0.125871	25.029153
3.	Searching for location present at level two(country)	0.168839	36.476122
4.	Searching For location which is not present.	88.079405	11.953245
5.	Time for creating index.	1323	548

We have compared Geo-skip list with JSI Library. JSI is java library which is simple main memory implementation of R-tree. While comparing with JSI Library we have not used disk space to store the objects of Geo-skip list. When we have compared Geo-skip list with R-tree (disk implementation) we have used disk space for storing the objects using serialization.

As, we can see in tables 1 and 2, the time taken by Geo skip list is less than that of an R-tree in some cases. By the use of cache we can decrease time taken by Geo-skip list to search particular location. Index creation in Geo-skip list takes more time than in R tree. But the index is created only once and that same index will be used for future work. Deletion and insertion operations are very rare in spatial location so we can consider that same index structure will be used for long time for performing search operation.

V. CONCLUSION AND FUTURE WORK

In this paper, we have presented an efficient data structure, called Geo-skip list, for representing geographic / spatial data. It represents geographic hierarchy very well since it does not uses fictitious bounding boxes like- in R trees. It represents any geographical region by its least and most co-ordinates rather than a single point. Search complexity of our structure on disk is better than R-trees in some cases while the one time index creation activity takes longer time for our structure.

We have successfully implemented the Geo-skip List data structure. Now, we have also proposed how various spatial queries can be solved using the data structure Geo-skip List.

We have successfully implemented Geo-Skip list data structure and proposed and proposed algorithms for insertion, deletion, searching in the proposed data structure. Also we have proposed and successfully implemented algorithms for answering four basic spatial queries i.e. point, range, nearest neighbor and kth nearest neighbor.

In future we can use geo-skip list to create hybrid index for answering queries based on geographical location. Combining geo-skip list with textual keyword based index will yield good results as compared to current hybrid structures in use. Geo-skip list can also be used in web applications like ‘Google Maps’ where we need to search for particular location on earth. With the help of Geo-skip list we can get hierarchy of any particular location in efficient manner.

We have stored Geo-skip list data in secondary memory along with index. Many optimization algorithms can be applied on index to further enhance the performance of this data structure. Also different storage methods and tools can be used to reduce the latency for fetching data from secondary memory to primary memory.

A spatial database is typically too large to even fit on an average-sized secondary storage device and data spills over to tertiary storage. So there is need for ways to use such distributed data efficiently. Algorithms exploiting parallel computation for answering spatial queries can boost up the overall performance of the Geo-Skip list to great extent. We suggested the means to divide the objects and strategy to store them on the disk. We have mentioned that while dividing those objects, it is important that objects should have complete meaningful structure i.e. we can’t store half of the structure of skip list in one file and other half in other file.

Suppose we are storing information about city New York. New York is highly populated and developed city. There will be many locations to store in New York. If the total number of nodes required to store geographical location of New York City exceeds free main memory, in that case we will not be able to store information about New York City in a single disk. To overcome this drawback we need to provide solution to divide skip list into different subparts such that it will regain its original structure after storing in different files on hard disk and later fetching them back into main memory.

VI. ACKNOWLEDGEMENT

We sincerely thank Getty [15] team for providing us their database TGN of geographical place names.

REFERENCES

[1] Barewar, A., Radke M.A., Deshpande, U.A. “Geo Skip List Data Structure - storing spatial data and efficient search of geographical locations” in Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference)

[2] A. Guttman, “R-Trees: A Dynamic Index Structure for Spatial Searching,” Proc. ACM SIGMOD ’1984,pp. 47-57.

[3] William Pugh. Skip list a probabilistic lternative to balanced tree. Publish in: MagazineCommunications of the ACM CACM Homepagearchive. Volume 33 Issue 6, June 1990, Pages 668-676.

[4] ChanopSilpa-Ana, Richard Hartley: Optimized KD-tree for fast image description matching. CVPR 2008.

[5] G. Brent Hall, Michael G. Leahy, Open Source Approaches in Spatial Data Handling, pp.105-129, 2008

[6] Rosie Jones,Ahmed Hassan and Fernando Diaz. Geographic Features in Web Search Retrieval. GIR’ 08 Proceedings of the 2nd international workshop on Geo-graphical information retrieval page 57-58.workshop on Geo-graphical information retrieval page 5758.

[7] Ramaswamy Hariharan, Bijit Hore, Chen Li, Sharad Mehrotra, Processing SpatialKeyword (SK) Queries in Geographic Information Retrieval (GIR) System SSDBM 2007, Banff, Canada.

[8] Xin Cao, Gao Cong, Christian S. Jensen, Beng Chin Ooi.Collective Spatial Keyword querying. Published in: SIGMOD '11 Proceedings of the 2011 ACM SIGMOD International Conference on Management of data Pages 373-384.

[9] http://www.getty.edu/research/tools/vocabularies/tgn/

[10] Zhou, Y., Xie, X., Wang, C., Gong, Y., Ma, W.Y.: Hybrid index structures for locationbased web search. In: CIKM ’05: Proceedings of the 14th ACM international conference onInformation and knowledge management, New York, NY, USA, ACM Press (2005) 155–162

[11] M. de Berg, M. van Krefeld, M. Overmars, andO. Schwarzkopf.Computational Geometry: Algorithmsand Applications. Springer, 2000.

[12] V. Gaede and O. G unther. Multidimensional access methods. ACM Comput. Surv, 1998, 30(2):170-231.

[13] C. B. Ohm, G. Klump, and H.-P. Kriegel. XZ-Ordering: A Space-Filling Curve for Objects with Spatial Extension,1999. In SSD, pages 75-90.

[14] Ramaswamy Hariharan, Bijit Hore, Chen Li, Sharad Mehrotra, Processing Spatial-Keyword (SK) Queries in Geographic Information Retrieval (GIR) System.SSDBM 2007, Banff, Canada.