kNR-tree: A novel R-tree-based index for querying any k relations among N spatial relations

Anirban Mondal[1] Anthony K. H. Tung[2] Masaru Kitsuregawa[1]

[1] Institute of Industrial Science
University of Tokyo
Japan
{anirban,kitsure}@tkl.iis.u-tokyo.ac.jp

[2] School of Computing
National University of Singapore
Singapore
atung@comp.nus.edu.sg

Abstract

The ever-increasing popularity of spatial data coupled with the prevalence of spatial applications has created the need for efficient processing of spatial queries in spatial database management systems. Incidentally, spatial indexing strategies typically address the indexing of a *single* spatial relation and consequently for indexing N different spatial relations, they maintain N different spatial indexes. In contrast with these existing spatial indexing techniques, we propose a *single* integrated novel R-tree-based structure, which we designate as the kNR-tree, for indexing objects from N spatial relations. While different types of spatial queries (e.g., spatial select queries, spatial join queries and nearest neighbour queries) can be directed to the kNR-tree, this work specifically addresses the processing of spatial select queries (i.e., window queries) on any k relations among N spatial relations. We designate such window queries on any k relations among N spatial relations as kNW queries. The main contributions of our work are two-fold. First, we present an efficient strategy for processing kNW queries using the kNR-tree. Second, we present *incremental* insertion and deletion algorithms for the kNR-tree. Our performance evaluation demonstrates that our proposed technique is indeed effective in reducing the response times of kNW queries as compared to the approach of maintaining separate indexes for each spatial relation.
Keywords: Spatial data, indexing, spatial window queries.

1 Introduction

The ever-increasing popularity of spatial data coupled with the prevalence of spatial applications has created the need for efficient processing of spatial queries in spatial database management systems. Incidentally, spatial indexing strategies typically address the indexing of a *single* spatial relation and consequently for indexing N different spatial relations, they maintain N different spatial indexes. In contrast with these existing spatial indexing techniques, we propose a *single* integrated novel R-tree-based structure, which we designate as the kNR-tree, for indexing objects from N spatial relations.

While different types of spatial queries (e.g., spatial select queries, spatial join queries and nearest neighbour queries) can be directed to the kNR-tree, this work specifically addresses the processing of spatial select queries (i.e., window queries) on any k relations among N spatial relations. We designate such window queries on any k relations among N spatial relations as kNW queries. The reason for addressing kNW queries instead of just considering window queries on a single relation is that in practice, a user may be interested in objects from a number of different relations and different users may be interested in different numbers as well as different kinds of relations.

Now let us understand the importance of kNW queries in the real-world. Given a spatial window S, different users may issue the following queries:

1. Find all bookshops, restaurants and car-parks which are within S.
2. Find all hotels, train stations, shopping centres

and tourist attractions within S.

3. Find all restaurants and shopping centres within S.

In this regard, the main contributions of our work are two-fold:

- We present an efficient strategy for processing kNW queries using the kNR-tree.
- We present *incremental* insertion and deletion algorithms for the kNR-tree.

Our performance evaluation demonstrates that our proposed technique is indeed effective in reducing the response times of kNW queries as compared to the approach of maintaining separate indexes for each spatial relation. The remainder of this paper is organized as follows. Section 2 provides an overview of relevant existing works, while Section 3 presents an overview of the problem. The kNR-tree index structure is proposed in Section 4. Section 5 reports the performance evaluation. Finally, we conclude in Section 6 with directions for future work.

2 Related Work

The R-tree [3] is one of the most popular and widely used multi-dimensional index structures. It is a generalization of the well-known B-tree. Each spatial data object in the R-tree is represented by a Minimum Bounding Rectangle (MBR). Leaf nodes in the R-tree contain entries of the form *(oid, rect)* where *oid* is a pointer to the object in the database and *rect* is the MBR of the object. Non-leaf nodes contain entries of the form *(ptr, rect)* where *ptr* is a pointer to a child node in the R-tree and *rect* is the MBR that covers *all* the MBRs in the child node.

Note that the bounding rectangles at each level of the R-tree can overlap. As a result, an R-tree search accesses multiple leaf nodes. Hence, we can intuitively understand that too much overlap can degrade the search performance significantly. With the objective of reducing such overlaps, some variants of the R-tree, such as the R^+-tree [4] and the R*-tree [1] have been proposed. While the R^+-tree avoids overlapping rectangles in the intermediate nodes of the tree, the R*-tree gives preference to squarish covering boxes with the objective of reducing overlaps. We observe that the problem of overlaps in R-tree-based index structures is an important issue in the context of shared-nothing systems. This is because such overlaps make it difficult to find a good data placement scheme.

3 Problem Overview

This section discusses an overview of the problem.

In our proposed system, every object is represented by its centroid, the implicit assumption being that all objects are points in space. This assumption is consistent with real-life situations primarily because the area encompassed by a given object is usually negligible with respect to that of the area of the universe under consideration. Incidentally, in practice, it is possible for a single object to satisfy more than one descriptor. As a single instance, a hotel may satisfy multiple descriptors such as restaurant, jacuzzi and gymnasium.

The implication is that the number of actual objects retrieved by a particular query need *not* necessarily be proportional to the number of descriptors contained in the query. In other words, if a query contains k descriptors, the number of objects in the query result may *not* be $k \times a$, where a is an integer. In particular, the number of objects in the query result depends upon the descriptors contained in the objects within the spatial window associated with the query. Hence, it is *not* possible to have a priori knowledge concerning the actual number of objects that would be required to satisfy a given query.

Each object has a corresponding descriptor bitmap associated with itself. Each entry position in the descriptor bitmap corresponds to a specific descriptor i.e., position 1 of the descriptor bitmap relates to descriptor 1, position 2 is associated with descriptor 2 and so on. In other words, the descriptor bitmap for each object is *exactly* the same in terms of the entry positions of descriptors. For each descriptor that the object under consideration contains, the corresponding entry in the object's descriptor bitmap is ticked as '1', while all other entries are marked as '0'.

Figure 1 depicts an illustrative example of the descriptor bitmaps associated with the spatial objects. As indicated in Figure 1, the descriptors, $D1$ to $D7$ represent 'hotel room', 'jacuzzi', 'conference room', 'shops', 'gymnasium', 'restaurants' and 'internet cafe' respectively. Since object A (a five-star hotel) satisfies all these descriptors (except $D5$), $D1$, $D2$, $D3$, $D4$, $D6$ and $D7$ are ticked as '1' in object A's bitmap, while $D5$ is marked as '0'. Similarly, object B (a shopping plaza) satisfies only the descriptors $D4$, $D6$ and $D7$, hence $D4$, $D6$ and $D7$ are marked as '1' in object B's bitmap, while the other descriptors are marked as '0'. In the same manner, since object C (a sports facility) only satisfies $D5$, only $D5$ is ticked as '1' in object C's bitmap with all the other descriptors being marked as '0'. User query is also a bitmap whose structure is *exactly* the same in terms of entry positions of descriptors as that of the object descriptor bitmap. The descriptors that the query contains are ticked as '1', while the rest of the descriptors are marked as '0'.

4 kNR-tree: A single integrated index for objects from N different spatial relations

In this section, we present the kNR-tree, which is our proposed R-tree-based indexing structure for process-

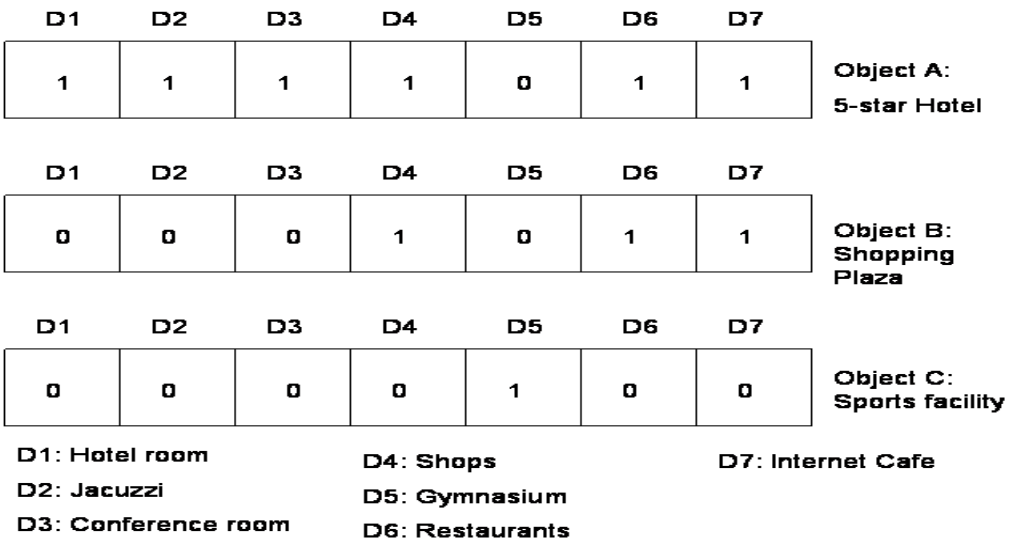


Figure 1: Illustrative example of the descriptor bitmaps

ing kNW queries efficiently. In our study, we have used the R-tree structure [3], although it can be substituted by other base structures or variants of the R-tree [1, 4]. This section also discusses insertion and deletion algorithms for the kNR-tree to indicate that the kNR-tree supports effective incremental insertions and deletions. Additionally, we indicate how the kNR-tree can be used for processing kNW queries efficiently.

The structure of the kNR-tree

Figure 2 depicts an illustrative example of the kNR-tree. In Figure 2, the universe is divided into three rectangular spatial regions A, B and C. As depicted in the figure, H, P, J and S stand for 'hotel', 'presentation room', 'jacuzzi' and 'shopping centre' respectively. The root node's bitmap H,P,J,S = (1,0,0,1) indicates that the universe comprising A,B and C contains a hotel and a shopping centre, but neither a presentation room nor a jacuzzi. For the sake of convenience, we shall use this notation throughout this figure.

A is further divided into three rectangular spatial regions D, E and F respectively. The figure shows that the region covered by D, E and F also contains a hotel and a shopping centre without having any presentation room or jacuzzi. D is further divided into three rectangular spatial regions X, Y and Z. The figure displays that the region encompassed by X, Y and Z contains only a hotel. E is further divided into three rectangular spatial regions T, U and V and the region encompassed by T, U and V contains only a shopping centre. Similarly, B and C are further divided into I, J, K and L, M and N respectively.

Algorithm Insert (R, O)

Inputs:
1) A kNR-tree whose root node is R.
2) A point object O with its corresponding descriptor bitmap B_O which is to be inserted into kNR-tree.
Output: Insertion of object O into the kNR-tree.

```
if R is not a leaf node
  for each MBR entry (ptr, MBR) of R, find out the
  MBR M in which O falls.
  Perform an OR operation between B_O and the
  existing bitmap at R.
  Execute Insert (Childptr, O) /* Childptr is the
  pointer to M's child node */
else
  Insert O into R
  Perform an OR operation between B_O and the
  existing bitmap at R.
end
```

Figure 4: Insertion of an object into the kNR-tree

Insertion

New objects may need to be added to the universe under consideration (e.g., opening of a new restaurant or sports facility in some part of the universe). In some cases, it is possible that some objects, which had been physically existing in the universe, had *not* been considered previously for being indexed by the kNR-tree possibly due to lack of information concerning these objects at that time (or some other reason), but when sufficient information concerning these objects become available, they should be indexed. In either case, efficient handling of insertion of objects into the kNR-tree becomes a necessity to support kNW queries over a

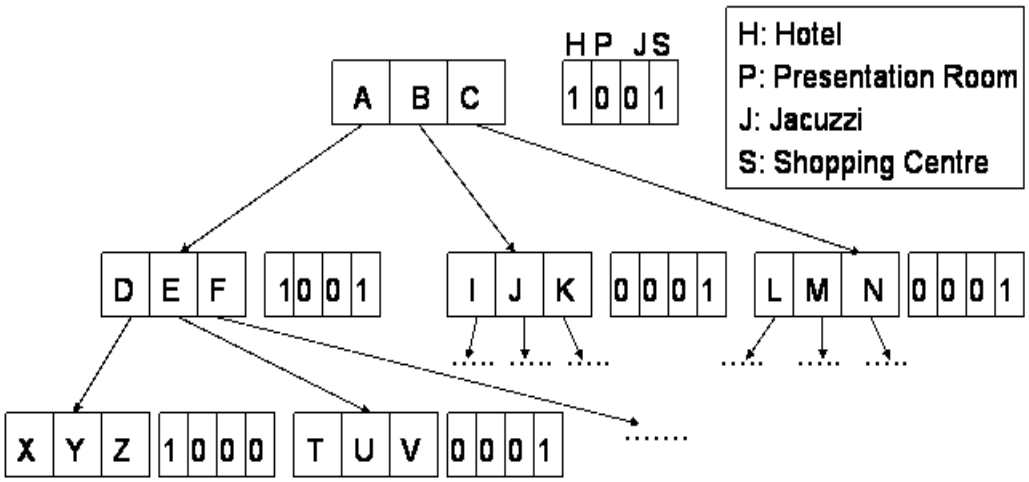


Figure 2: Illustrative example for the kNR-tree

long period of time in which new objects may emerge in the universe.

Our strategy for the insertion of objects into the kNR-tree comprises a top-down traversal strategy (i.e., starting at the root node) which involves both the determination of the leaf node of the kNR-tree where the object should be inserted as well as updates to the bitmaps of those nodes of the tree which fall in the path of the top-down traversal. An update to the bitmap associated with a particular node consists of the execution of an OR operation between the bitmap of the object being inserted and the existing bitmap at that node.

Now let us see how the kNR-tree in Figure 2 is modified in response to the insertion of an object Q, whose bitmap is (0,1,1,0) and which is located within the spatial region covered by X, Y and Z. Figure 3 displays the kNR-tree after Q has been inserted. The insertion operation is initiated from the root node of the kNR-tree. First, the OR operation is performed between Q's bitmap and the existing bitmap at the root node i.e., (1,0,0,1), the new bitmap being (1,1,1,1). Since Q falls within A, the insertion operation traverses to the node comprising D, E and F. Once again, the OR operation is executed between Q's bitmap and the existing bitmap corresponding to the region covered by D, E and F i.e., (1,0,0,1), the new corresponding bitmap being updated to (1,1,1,1) as indicated in Figure 3. Now since Q falls within the region encompassed by X, Y and Z, the insertion operation traverses to the branch associated with X, Y and Z and updates the corresponding bitmap to (1,1,1,0) as shown in Figure 3.

The algorithm for insertion of objects into the kNR-tree is presented in Figure 4. Observe that the insertion algorithm is similar to that of standard R-tree insertion algorithms in that it decomposes the search space and recursively searches the R-tree, the main difference being the updates to the bitmaps associated with the respective nodes of the kNR-tree. Moreover, given that we represent each object as a point, the insertion of an object involves the traversal of only a *single* path from the root node to the leaf node. This is in contrast with the case of representing objects by their respective MBRs (Minimum Bounding Rectangles) which potentially involve the traversal of multiple paths from the root node to the leaf node(s) for the insertion of a single object.

For creating the kNR-tree from scratch, all the bitmaps are initially set to (0,0,0,..0). Then the objects are inserted one by one into the kNR-tree using the insertion algorithm that we have just discussed above. In essence, every time we insert an object O into the kNR-tree, we perform an OR operation on the bitmap corresponding to the region covered by O at each level of the kNR-tree all the way from the root node of the kNR-tree to the leaf node of the kNR-tree where the object is to be inserted.

Deletion

Just as new objects may need to be added to the system, it is also possible for existing objects to disappear from the universe e.g., a restaurant or a gymnasium in the universe may close down. Understandably, if these objects are *not* deleted from the kNR-tree, the results of kNW queries pertaining to these objects would be erroneous because the results would contain non-existent objects. Hence, deletion of objects is necessary for the kNR-tree to accurately reflect and index

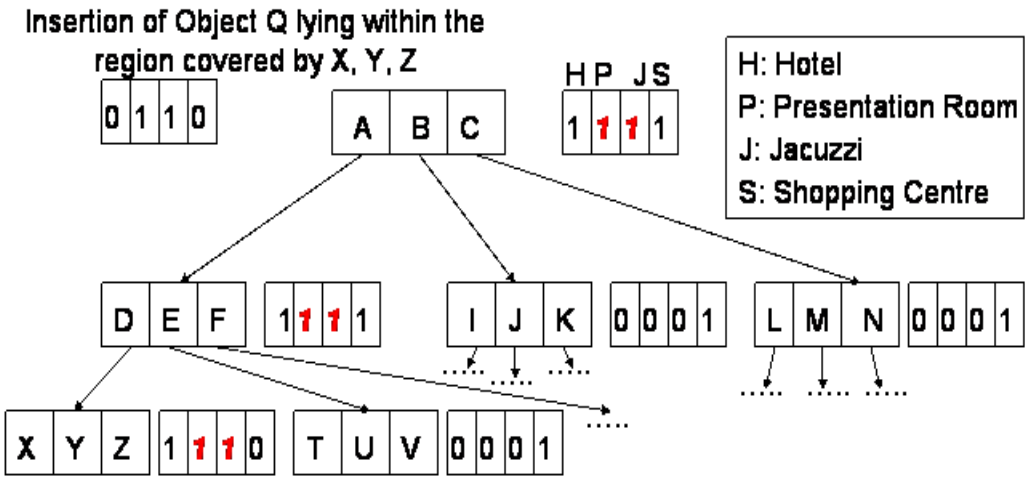


Figure 3: The kNR-tree after an object Q has been inserted

the currently existing objects in the universe over a large time-frame.

Deletion of an object from the kNR-tree comprises a bottom-up traversal approach (i.e., starting at the leaf node associated with the object being deleted) which consists of comparing the bitmap of the object, the respective bitmaps of the other objects associated with the node under consideration and the existing bitmap at that node with the objective of determining whether the existing bitmap at the node needs to be modified due to the object's deletion. In case, the existing bitmap at the node needs any modification(s), the changes should be propogated upwards to the next level of the kNR-tree, otherwise the deletion algorithm terminates. Intuitively we can understand that changes to the bitmaps of the nodes may need to be propagated all the way up to the root node of the kNR-tree.

We shall use Figure 5 to show how the kNR-tree in Figure 3 is modified after the deletion of an object R whose bitmap is (1,1,1,0) and which is located within the spatial region covered by X, Y and Z. The deletion operation is initiated from the leaf node of the kNR-tree associated with X, Y and Z. Assume that other objects lying within X, Y and Z have a '1' in positions 2 and/or 3 of X's bitmap. (We number the positions in the bitmap starting from 1). For example, Q which has just been inserted into X has a '1' in positions 2 and 3 of its bitmap. Also, let us suppose that no object in X, Y and Z has a '1' in position 1 of the bitmap i.e., R was the only object in X, Y and Z which had a '1' in position 1 of its bitmap. Hence, position 1 in the bitmap of the leaf node corresponding to X, Y and Z should now be changed to '0' to reflect R's deletion, thereby modifying the bitmap to

Algorithm Delete (kNR-tree, D)

Inputs:
1) A kNR-tree
2) A point object D with its corresponding descriptor bitmap B_D which is to be deleted from the kNR-tree.

Output: Deletion of D from the kNR-tree.

Note the positions with entry '1' of B_D into a set $Change$
Identify the leaf node $Leaf$ associated with D.
Delete D from $Leaf$
Propagate_Changes ($Leaf$, $Change$)

end

Figure 6: Algorithm for deletion of objects from the kNR-tree

(0,1,1,0) as indicated in Figure 5. Now this change has to be propagated upwards, so the deletion algorithm traverses to the node comprising D, E and F.

For region D, we know that no object has a '1' in position 1 of the corresponding bitmap. Now suppose that there exists no object in E and F for which position 1 of the bitmap contains a '1'. Hence, the new bitmap for the node comprising D, E and F should be (0,1,1,1), the position 4 of this bitmap being marked as '1' because we can see from Figure 3 that at least one object lying within D, E and F has a '1' in position 4 of its bitmap since the bitmap of the node corresponding to T, U and V has a '1' in position 4 of its bitmap. To propagate this change upwards, the deletion algorithm now goes to the root node. As depicted in Figure 5, the bitmaps of B and C do *not* have a '1' in position 1. Since none of the nodes below the root node

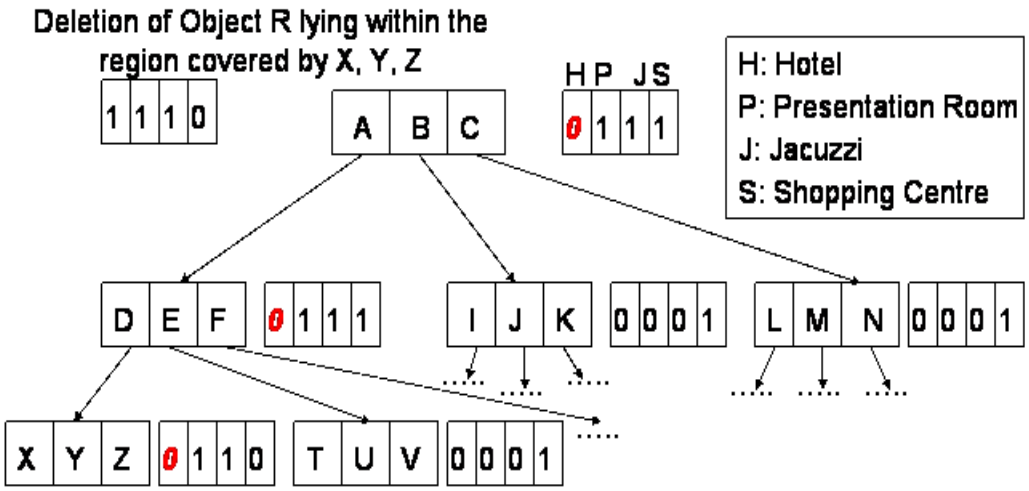


Figure 5: The kNR-tree after an object R has been deleted

contain a '1' in position 1 of their respective bitmaps, position 1 of the root node's bitmap should now be changed to '0', thereby resulting in the new bitmap for the root node being (0,1,1,1) as depicted in Figure 5. The algorithm for deletion of objects from the kNR-tree is presented in Figure 6. Note that the deletion algorithm uses the algorithm *Propagate_Changes* for upward propogation of deletion-related changes. The algorithm *Propagate_Changes* follows standard R-tree algorithms, except that the bitmaps have to be updated when propagating changes upwards.

Updates

The descriptor bitmaps of the spatial objects need to be updated in response to the addition and/or deletion of descriptors concerning existing spatial objects, thereby making it necessary for the kNR-tree to support updates incrementally. For example, a hotel (spatial object), which did not have any jacuzzi previously, may open a new jacuzzi (addition of a new descriptor) to attract customers, while another hotel may decide to close down its in-house gymnasium (deletion of an existing descriptor) as part of its cost-cutting measures. For simplicity, we shall regard the update operation operation as a sequence of deletion followed by insertion.

Notably, for maintaining the correctness and the consistency of the kNR-tree, insertions, deletions and updates should all be *atomic* operation i.e., the object is either inserted, updated or deleted with the corresponding bitmaps fully updated or the system rolls back to the state prior to which the insertion, update or deletion of the object was initiated. Moreover, observe that we have *not* discussed concurrency control issues associated with insertions, deletions and updates, the reason being that this work implicitly assumes low frequency of these operations. Incidentally, this is in consonance with real-world conditions, where new spatial objects such as buildings are *not* typically created or destroyed every day.

However, given that spatial objects may sometimes be created/deleted (albeit with low frequency of occurrence), we believe that the kNR-tree should *not* have to be built from 'scratch' every time any of these operations takes place. Hence, in the interests of efficiency, we have presented the algorithms for incremental insertions and deletions. Furthermore, due to lack of space, we have not discussed issues concerning overflow and underflow in the kNR-tree nodes since they can be essentially handled in the same manner as in standard R-tree algorithms [3, 4, 1], the difference being that descriptor bitmaps also need to be updated to reflect node splits and node merges.

kNW Query Processing using the kNR-tree

Our strategy for processing kNW queries using the kNR-tree comprises a top-down traversal involving only those nodes whose MBRs intersect with the query window such that the bitmap of every node, which falls in the path of the top-down kNR-tree traversal, is checked against the query bitmap to decide whether to go further down the branches emanating from the node. The result set consists of k linked lists, each linked list storing the objects retrieved for one of the k relations. Whenever any object is retrieved, the algorithm first determines which relation(s) it belongs to and then adds the object to the linked list(s) asso-

Algorithm Spatial_ Window (R, W, Q_{bitmap})
Inputs: 1) A kNR-tree whose root node is R.
2) A query window W
3) Q_{bitmap}, the query bitmap
Output: Spatial window query results
if R is not a leaf node
  if R satisfies Q_{bitmap}
    Find each MBR entry M of R intersecting W
    for each M
      execute **Spatial_ Window** (*Childptr*, W, Q_{bitmap})
      / Childptr is the pointer to M's child node */*
else
  if R satisfies Q_{bitmap}
    Find a list L of MBR entries of R that intersect W
    Using MBR_{bitmap}, find each entry M of L which
    satisfies Q_{bitmap}
    for each M
      Check each object within M
      Add each object satisfying Q_{bitmap} to the result set
end

Figure 7: Spatial window query processing algorithm for the kNR-tree

ciated with the object[1]. The kNW query processing algorithm for the kNR-tree is presented in Figure 7. In Figure 7, we define a node (or leaf-node MBR) as *satisfying* a query bitmap if the node (or leaf-node MBR) contains at least one of the k relations associated with the query. Moreover, MBR_{bitmap} refers to the bitmaps corresponding to the MBRs of the nodes of the kNR-tree.

5 Performance Study

This section reports the performance evaluation of our proposed techniques. The machine used for the experiments had processing capacity of 1.7 GHz (Pentium-4), main memory of 768 Mbytes and disk space of 40GB. We ran the experiments under the Redhat Linux (version 7.3) operating system. The interarrival time between queries was fixed at 5 seconds.

We have used a *real-life* dataset *'Greece Roads'*[2] for our experiments. The *'Greece Roads'* dataset contains 23268 rectangles representing the data of roads in Greece. First, we computed the centroid of these rectangles to obtain a dataset of 23268 points before enlarging this dataset by translating and mapping the data. For our experiments, we used more than 200000 points (objects), each point being associated with *at least one* spatial relation from the set of 20 relations used for our experiments. The points were indexed using the kNR-tree. We assumed that one kNR-tree node fits in a disk page (page size = 4096 bytes). Hence, kNR-tree node capacity is the same as page size in our case. We used a fan-out of 64 for the kNR-tree.

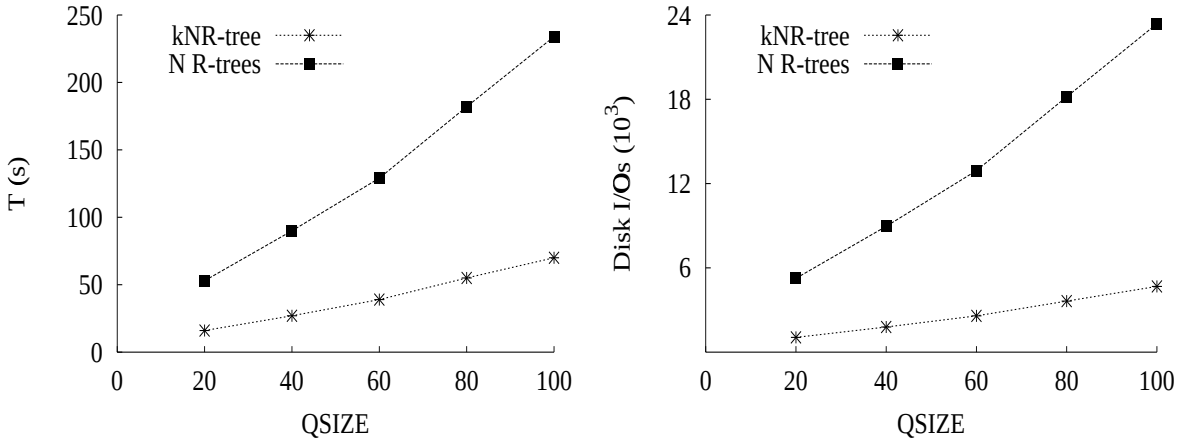
[1]If an object belongs to multiple relations, it will appear in the linked lists of all the relations that it belongs to.

We define the size of a query **QSIZE** as the percentage of the universal space that a query covers. For example, QSIZE = 20 implies that the query covers 20% of the area associated with the universe. We numbered the relations as 1 to N. Each object was associated with at most 3 relations. For deciding the number of relations associated with a particular object, we generated a random number q between 1 and 3 so that the object belongs to q relations. Then we generate q *distinct* random numbers between 1 and N and assign the object to the q relations whose relation numbers match with these generated numbers. For generating queries, we see the value of k in a particular query Q and associate k relations with Q by choosing k *distinct* random numbers between 1 and N. Then we select a point randomly in the domain of the base station under consideration and draw a rectangle of area QSIZE using the point as the centroid of the rectangle. This rectangle is our query window.

Now let us examine the performance of the kNR-tree. As reference, we adopt a traditional approach which uses N different R-trees to index N relations i.e., one R-tree for each relation. Let us designate it as the *'N R-trees'* approach.

Figure 8 shows the effect of variations in QSIZE when k is fixed. Figures 8a and 8b presents the results concerning *query response time* T and total number of disk I/Os incurred for k=5. When QSIZE increases, more branches of the index structures need to be traversed, thus explaining the reason for higher number of disk I/Os and consequently higher query response times for increasing values of QSIZE. While the kNR-tree requires only one traversal from its root node to its leaf nodes, the 'N R-trees' approach needs to make one traversal from the root node to the leaf nodes for *each* of the R-trees corresponding to the queried relations, thereby incurring significantly higher number of disk I/Os (shown in Figure 8b) and hence much higher response times (depicted in Figure 8a) than the kNR-tree. Moreover, if an object satisfies q relations, it would be retrieved only once in case of the kNR-tree, while it would be retrieved q times from q different R-trees in case of the 'N R-trees' approach.

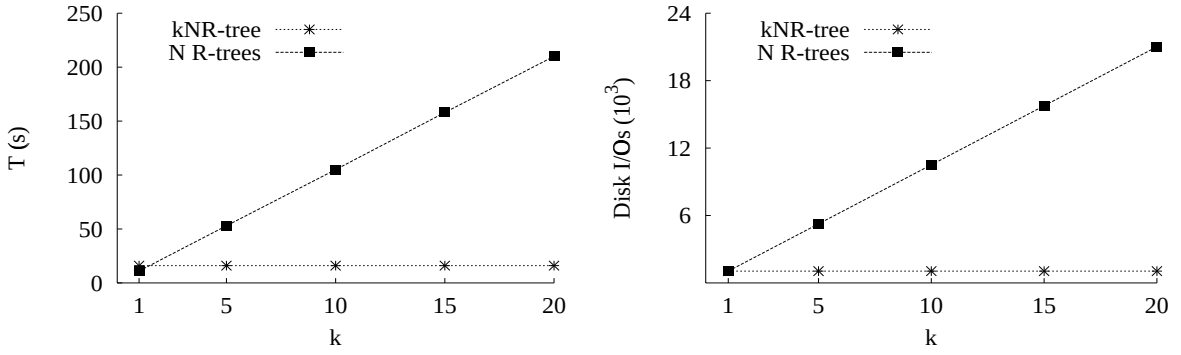
Figure 9 depicts the effect of variations in k when QSIZE is fixed. Figures 9a and 9b show the query response times T and disk I/Os for QSIZE=20. As k increases, kNR-tree's performance gain over the 'N R-trees' approach also increases due to lower number of disk accesses incurred by the kNR-tree as discussed above. Interestingly, the results in Figure 9a indicate that the kNR-tree performs slightly worse than the 'N R-trees' approach when k=1. A detailed examination of the experimental results log revealed that this may be attributed to two reasons. First, the height of the kNR-tree can be expected to be larger than at least some of the individual R-trees in the 'N R-trees' approach. Second, unlike the 'N R-trees' approach, the



(a) T for $k = 5$

(b) Disk I/Os for $k = 5$

Figure 8: Effect of variations in QSIZE



(a) T for QSIZE = 20

(b) Disk I/Os for QSIZE = 20

Figure 9: Effect of variations in k

kNR-tree needs processing time to handle the bitmaps of its nodes during the traversal. In summary, the results in Figures 8 and 9 demonstrate that our proposed kNR-tree significantly outperforms the 'N R-trees' approach.

6 Conclusion

The increasing popularity of spatial data coupled with the prevalence of spatial applications has created the need for efficient processing of spatial queries. In this paper, we have addressed the processing of spatial select (window) queries on any k relations among N spatial relations. Our solution involves the use of our proposed kNR-tree. Our performance evaluation has demonstrated the effectiveness of our proposed kNR-tree-based technique in reducing the response times of kNW queries. In the near future, we intend to make more detailed performance comparisons of our indexing technique with relevant existing techniques.

References

[1] N. Beckmann, H.P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. *Proc. ACM SIGMOD*, 1990.

[2] Datasets. http://dias.cti.gr/~ytheod/research/datasets/spatial.html.

[3] A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, 1984.

[4] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi-dimensional objects. *Proc. VLDB*, pages 507–518, 1987.