On Using B^+-Tree for Efficient Processing for the Boundary Neighborhood Problem

AZZAM SLEIT
Department of Computer Science, King Abdulla II School for Information Technology
University of Jordan,
P.O. Box 13898, Amman 11942,
JORDAN
azzam.sleit@ju.edu.jo

Abstract: - A spatial data set is a collection of spatially referenced objects. The boundary neighborhood problem (i.e. BN problem) in spatial database systems is defined as finding all boundary neighbors for a given object based on special dimensions. This problem arises in several applications mainly in GIS systems such as finding all countries that surround a given lake or sea. Spatial objects can be represented and indexed by their minimum bounding rectangles (i.e. MBR's) which give rough indication for the existence of an object. This paper proposes a solution for the BN problem, based on indexing the space using B^+-tree.

Key-Words: - B^+-tree, Boundary Neighborhood Operator, Spatial Database, Rectangular Object

1 Introduction

Management of spatial data is a requirement for various fields. Space of interest can be two dimensional geographical planes such as a geographic information systems, man-designed space like a layout of a VLSI design or conceptual information space like an electro-magnetic field. Systems of spatial data types or spatial algebras capture the abstractions for points, lines, rectangles and regions that provide the basis for modeling the structure of geometric entities in space together with their relationships, properties and operations. The key function of all spatial systems is the ability to query the database for specific relationships between objects especially topological relations such as the nearest, intersection and adjacency. The processing of such queries depends on the efficiency with which the objects are stored and indexed and how the query utilizes the stored objects.

A point in space may represent the location of an object in space. Cities can be modeled as points in space while lines can be viewed as the basic abstraction for moving through space, or connections such as roads, rivers, or cables. Regions can be considered as the abstraction for objects having extent in 2d-space such as countries, lakes, or parks. Rectangles are often used in approximating regional spatial objects to serve as the minimum rectilinear enclosing objects or more commonly called the minimum bounding rectangles (i.e. MBRs) [4]. In such a case, the approximation gives a rough indication for an object existence within the MBR. A spatial query is then processed in two steps [5]. Firstly, a filter step employs an index to retrieve all MBRs that satisfy the query and possibly some false hits. Secondly, a refinement step uses the exact geometry of the objects to dismiss the false hits. Consequently, proposing that the spatial system represents and indexes regional objects in a two-dimensional space as MBRs, we define the boundary neighborhood problem (BN) as finding all MBRs which surround and touch the MBR of a particular object. Providing an efficient solution for large databases of MBRs has obvious applications in areas such as Geographic Information Systems, VLSI, and wireless computing.

Several indexing structures have been proposed to facilitate accessing regions in multi-dimensional space. The quad-tree [7, 13, 14] which is one of the oldest methods used to index spatial data is a generalization of binary search tree to higher dimensions. It represents recursive subdivision of space into subspaces using iso-oriented hyper planes. In quad-tree variants, each interior node has four descendants, each corresponds to a rectangle. The rectangles are referred to using the compass points: NW, NE, SW, and SE quadrants. The decomposition of the space is performed until the number of points in a rectangle is smaller than a given threshold. Consequently, quad-trees are not necessarily balanced. This decomposition turns out

to perform well when data is uniformly distributed, which is rarely the case with real life data.

R-trees [2, 3] and its variants are MBR-based data structures for indexing spatial databases. It is a hierarchical data structure similar to B^+-tree [1, 8, 12]. Each internal node of R-tree has a large fanout in order to guarantee a minimum height of the tree. The maximum fanout of a node is determined by the size of the disk page, where the tree is stored. Each internal node corresponds to the bounding boxes of the rectangles stored in the descendant nodes. Leaf nodes contain the MBRs of the actual objects in the database. Bounding boxes in internal nodes may overlap, but each input rectangle is represented in only one node at each level of the tree [6]. Like B^+-tree, R-tree is perfectly balanced with $\log_m n$ height, where m is the minimum fanout of an internal node, and n is the number of objects stored. SB^+-tree [9, 10, 11] is a balanced point and region access structure. Each dimension of the space is indexed by an independent SB^+-tree. When a query is processed, only the SB^+-trees corresponding to the dimensions referenced by the query are searched and the output is produced in terms of the outcome of the individual searches. Internal nodes in the SB^+-tree have only keys and pointers to children. Leaf nodes have pointers to data blocks. Each data block contains tuples of the form (*objid*, *relation_type*, *status*, *pr*). *objid* is the object identifier, *relation_type* is the object type, *status* denotes the extent status of the object at the specific data point(i.e. start, end, or continue) and *pr* points to the actual object in the database. Similar to R-tree, SB^+-tree is a secondary-memory resident structure which makes both of them appropriate for database systems.

The following section provides a formal definition for the BN problem. Section 3 proposes a solution for the BN problem using B^+-tree. Section 4 analyzes the performance of the proposed solution and compares it with that using SB^+-tree. Section 5 evaluates experimentally the solution using the US Census Bureau; i.e. the Tiger Collection. The paper is concluded in section 6.

2 The Boundary Neighborhood Operator

The Boundary Neighborhood Operator (BNO) is defined as the set of objects that spatially bound the query object as per the following definition.

Definition

Let s be a query MBR rectangular object, s.startX and s.endX are the start and end coordinates of the object s in the X-axis, while s.startY and s.endY are the start and end coordinates of the object in the Y-axis. R is the object type. The Boundary Neighborhood Operator BNO(R,s) is defined as the set of objects $x_1, x_2, \ldots, x_n$, where s, x_i Є R such that

(

(x_i.startX = s.endX AND ([x_i.startY, x_i.endY] ∩ [s.startY, s.endY] ≠ Φ)) OR

(x_i.endX = s.startX AND ([x_i.startY, x_i.endY] ∩ [s.startY, s.endY] ≠ Φ)) OR

(x_i.startY = s.endY AND ([x_i.startY, x_i.endY] ∩ [s.startY, s.endY] ≠ Φ)) OR

(x_i.endY = s.startY AND ([x_i.startY, x_i.endY] ∩ [s.startY, s.endY] ≠ Φ))

) for i= 1, 2, …, n

This operator can be utilized to find all countries that have borders with a specific country, or all countries that are adjacent to a particular sea. Fig. 1 gives an example of a map of countries labeled as C1, C2, …, C14 and seas labeled as S1, S2, and S3. The countries that have borders with C1 can be obtained using the operator BNO(COUNTRY, 'C1') which outputs {C3, C5, C7, C10, C11, C12}. However, BNO(SEA, 'C12') generates all seas that are adjacent to country C12; namely, {S2, S3}.

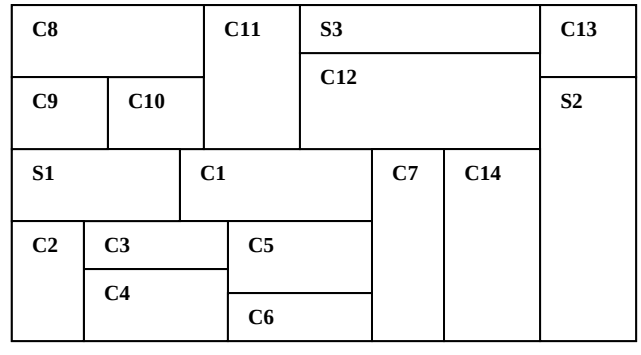

Fig. 1: An example of countries and seas labeled as Ci's and Sj's, respectively

3 Proposed Solution

The B^+-tree index is a balanced page-oriented tree which consists of two types of nodes; namely, index and data nodes. The index nodes of a B^+-tree correspond to the internal nodes while the data nodes

point to actual data objects. Data nodes are linked together, from left to right. A B^+-tree is said to be of order M if every node has from M/2 to M subtrees. The root may have from two to M substrees. The leaves of the tree are at the lowest level of the tree (i.e. level 1) while the root is at the highest level. The number of levels in the tree is the height of the tree. A non-leaf node with j keys contains j+1 pointers to children. A <pointer, key, pointer> is termed an index entry. Key values of a node are kept in sorted order. Thus, a B^+-tree is a multi-level index with the topmost level being the single root page and the lowest level consisting of the set of leaf pages. Leaf nodes have pointers to data blocks which have tuples of the form (*objid, pr*), where *objid* is the object id and *pr* is a pointer that points to the spatial database. A search, insertion or deletion operation starts searching the root to find the page at the next lower level that contains the subtree having the search key in its range. The next lower level page is searched, and so on, until a leaf is reached. The leaf is then searched and the appropriate action is performed. Operations can be unsuccessful; for example, a search may not find the required key. As keys are inserted or deleted, the tree grows or shrinks in size. When an updater tries to insert into a full leaf page or delete from a leaf page with d entries, a page split or page merge may occur. Fig. 2 displays an example of B^+-tree of degree 2.

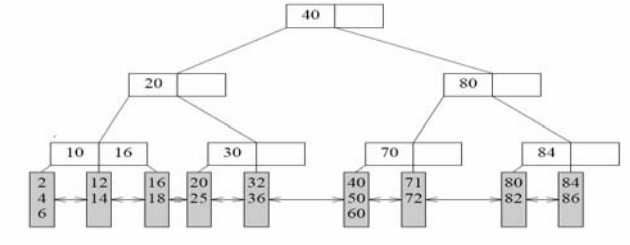

Fig. 2: An example of B^+-tree of degree 2.

We propose a solution for the BN problem based on the divide and conquer paradigm. The BN problem in the two-dimensional space is divided into 4 sub problems; each handles the neighbors related to one of the four margins of the MBRs included in the space. The scope of the BN subproblems is to capture the X-axis/Y-axis starting/ending coordinates for all MBRs. Consequently, we use four B^+-trees; each tree indexes the objects according to one of the four margins, StartX, EndX, StartY and EndY as follows:

- SartX B^+-tree indexes the MBRs in space based on their minimum x-values.
- EndX B^+-tree indexes the MBRs in space based on their maximum x-values.
- SartY B^+-tree indexes the MBRs in space based on their minimum y-values.
- EndY B^+-tree indexes the MBRs in space based on their maximum y-values.

Fig. 4 displays a sample space with the four StartX , EndX , StartY and EndY B^+-trees. To find the neighbors of the right side of the query object, we need to search the StartX B^+-tree for the objects that start at the same X margin that the query object ends. To find the neighbors of the left side of the query object we search the EndX B^+-tree for the objects that end at the same X margin that the query object starts. To find the neighbors to the Lower side of the query object we need to search the EndY B^+-tree for the objects that end at the same Y margin that the query object starts. Likewise, in order to find the neighbors to the Upper side of the query object we need to search the StartY B^+-tree for the objects that start at the same Y margin that the query object ends. The proposed solution is divided into two steps [5]. Firstly, a filter step employs an appropriate index to retrieve all objects that satisfy the query and possibly some false hits [15, 16]. Then, a refinement step uses the exact geometry of the objects to dismiss the false hits [17]. Fig. 3 is the pseudo code for the proposed algorithm.

Algorithm: Boundary Neighborhood of object q and set R(i.e. BNO(R, q))
Input:
1. StartX[N] : Pointer to the Start X B+-Tree of size N .
2. EndX[N] : Pointer to the End X B+-Tree of size N .
3. StartY[N] : Pointer to the Start Y B+-Tree of size N .
4. EndY[N] : Pointer to the End Y B+ Tree of size N .
5. q : query rectangular object.

Output: $\{x_1,x_2,..x_n\}:x_i \in R$, x_i and q are neighbors.

Assumptions: BPlusGetNeighbors is a B+-Tree search function that returns an array having all objects that are indexed at a certain key.

```
{
 rightSideNeighborsArray ← BPlusGetNeighbors(StartX[N] , q.endX)
 leftSideNeighborsArray ← BPlusGetNeighbors (EndX[N] , q.startX )
 upperSideNeighborsArray← BPlusGetNeighbors (StartY[N] , q.endY )
 lowerSideNeighborsArray ← BPlusGetNeighbors (EndY[N] , q.startY)

 for each xi in rightSideNeighborsArray
      if (xi.startY<= q.startY <= xi.endY ) OR (xi.startY<= q.endY <= xi.endY)
         Add(resultArray, xi) .
      end if
 end for

 for each xi in leftSideNeighborsArray
      if (xi.startY<= q.startY <= xi.endY ) OR (xi.startY<= q.endY <= xi.endY)
         Add(resultArray, xi) .
      end if
 end for

 for each xi in upperSideNeighborsArray
       if (xi.startX<= q.startX <= xi.endX ) OR (xi.startX<= q.endX <= xi.endX)
         Add(resultArray, xi) .
       end if
 end for

 for each xi in lowerSideNeighborsArray
       if (xi.startX<= q.startX <= xi.endX ) OR (xi.startX<= q.endX <= xi.endX)
         Add(resultArray, xi) .
      end if
 end for
}
Return resultArray
```

Fig. 3: BNO Algorithm

Fig. 4 displays a sample space with fifteen MBRs representing countries and the corresponding B^+-trees for StartX, EndX, StartY, and EndY. Assuming that we would like to retrieve all MBRs which have borders with f (i.e. MBR(f) = [5, 6, 3, 7], f.startX=5, f.endX=6, f.startY=3 and f.endY=7), BNO(COUNTRIES, f) is executed. The following is a trace for the algorithm.

resultArray = []

StartX B^+-tree is searched for border point f.startX=6, which produces rightSideNeighborsArray= [h, i, j]

During the filtering step: the Y-extents of objects h, i, and j are compared with those of f outputting west border MBRs.

resultArray= [h, i]

EndX B^+-tree is searched for border point f.endX=5, which produces leftSideNeighborsArray= [b, c, e]

During the filtering step: the Y-extents of objects b, c, and e are compared with those of f adding the east border MBRs [c, e] to the previous contents of resultArray.

resultArray= [h, i, c, e]

StartY B^+-tree is searched for border point f.endY=7, which produces upperSideNeighborsArray= [j, k]

During the filtering step: the X-extents of objects j and k are compared with those of f adding the north border MBRs [j, k] to the previous contents of resultArray.

resultArray= [h, i, c, e, j, k]

Finally, EndY B^+-tree is searched for border point f.startY=3, which produces lowerSideNeighborsArray= [j, k]

During the filtering step: the X-extents of objects j and k are compared with those of f adding the south border MBRs [g] to the previous contents of resultArray.

resultArray= [h, i, c, e, j, k, g]

Consequently, BNO(COUNTRY, f) = [h, i, c, e, j, k, g] which represents the countries that have borders with f.

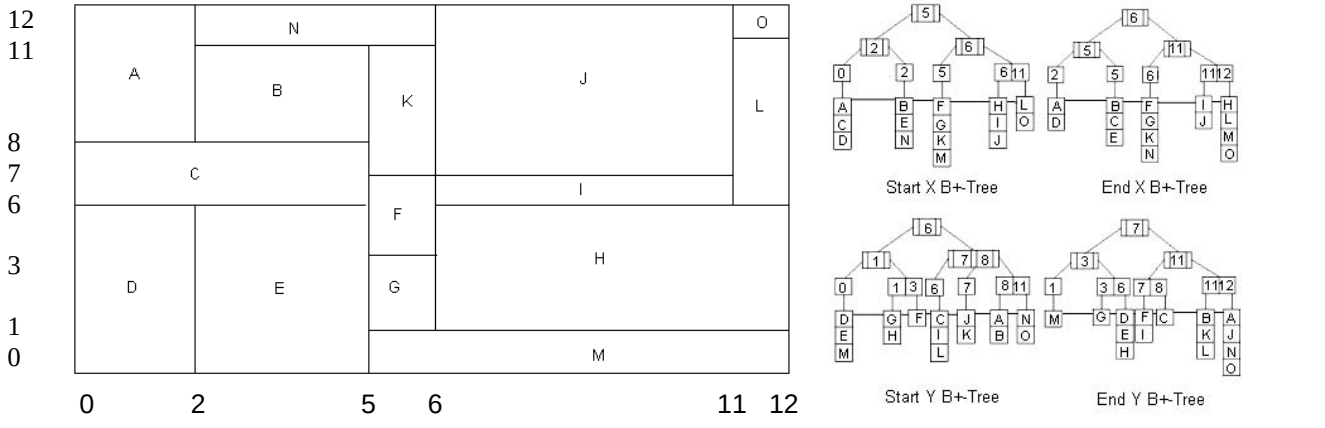

Fig. 4: Sample Space of Fifteen Countries with Corresponding B^+-trees of Degree 2.

4 Cost Analysis and Estimated Performance Comparisons

Operations on trees are dependent on the nature of the trees, which leads to time complexity that is proportional to their height. B^+-tree a secondary-memory resident data structure designed such that each node fits in exactly one page (i.e. I/O block) on disk. Therefore, the performance of operations utilizing B^+-trees is measured in terms of number of disk accesses. A leaf/internal node can hold keys and pointers depending on the size of the I/O block configured for the specific database and operating system. Fig. 5 lists parameters which will be used in our analysis.

Symbol	Interpretation
N	Number of objects in the space.
M	Order of the B^+-tree, which is number of index entries that fit in one page or I/O Block.
F	Fullness factor which is the average number of index entries in the nodes of the tree
P	Pointer size
K	Key Size
G	Page Size
Tb	Size of one leaf node tuple in B^+-tree
U	Number of leaf node tuples that can fit in one page.

Fig. 5: Analysis Parameters

Internal nodes in the B^+-tree are organized in the form <P1, K1, P2, K2,…., Pm, Km, Pm+1>, and a B^+-tree with M degree has a maximum of M separators or Keys;

$$(M * K) + (M+1) * P \leq G$$
$$M \leq (G - P) / (K + P) \quad (1)$$

The maximum height for N objects indexed by a B^+-tree happens when all N objects are indexed at different points. If we assume that all internal nodes have their full capacity which is M, this leads to a tree of height $O(\log_M(N / M))$. Therefore, operations will be proportional to $O(\log_M(N / M))$. However, internal nodes will be partially full and the fullness factor will play a role in defining the height of the tree. With a fullness factor F, operations have number of IOs proportional to

$$\log_{M*F}(N / (M * F)) \quad (2)$$

The proposed algorithm to find all neighbors uses four B^+-trees to index the objects. It searches the four trees to solve the BN problem. Assuming that we have N objects in the space with an average fullness factor of F for all trees, the number of required disk accesses to find the boundary objects of an MBR is given by:

$$4 * \log_{M*F}(N / (M * F)) \quad (3)$$

There is another part in the complexity of BN problem which is the complexity to access the data block indexed at a certain leaf node. A data block holds tuples in the form (*Objid, Pointer*) in addition to two pointers; one points to the data block associated with the next indexing point and the other points to the data block associated with the previous indexing point. We have

$$(U * Tb) + (2 * P) \leq G$$
$$U \leq (G – (2 * P)) / Tb \quad (4)$$

Consequently, the maximum number of tuples in a data block that can fit in one page is (G – (2 * P)) / Tb. In the following, we discuss best, worst and average case time complexities to access the data blocks.

Best Case Performance:
The best case happens when all rectangles are stacked either on the X axis or Y axis. Assuming that the N objects are stacked with respect to the X axis, the maximum number of neighbors for any object is two. In this case, the StartX and EndX B^+-trees have one data block with N tuples four each. However, the StartY and EndY B^+-tree have N data blocks each with only one tuple. Therefore, the maximum number of data-block I/Os required to solve the NB problem in this case is two.

Worst Case Performance:
The worst case happens when all (N – 1) objects are neighbors to the query one. The StartX and EndX B^+-trees have two data blocks for each. The first data block has one tuple while the second has N- 1 tuples. The StartY and EndY B^+-trees have N-1 data blocks for each of which N-2 blocks have one tuple while one data block has two tuples. To access the data block we need

$$[(((N-1) * T_b) + (2* P)) / G] \quad (5)$$

Average Case Performance:
The average case happens when all objects have the same number of boundary neighbors except for the objects that are at the edges of the space. Assuming that each object is a square with equal sides O and the data space is normalized to [0 , O×SQRT(N)] where N is the number of objects in the space. This space is approximated by a grid of size SQRT(N) × SQRT(N) and has the following four properties:

If Xs is a point in the X axis, Xs Є [0,O × SQRT(N)) , Xs mod O equals zero -> we have SQRT(N) objects stacked over the X axis start at Xs.

If Xe is a point in the X axis, Xe Є (0,O × SQRT(N)] , Xe mod O equals zero -> we have SQRT(N) objects stacked over the X axis and end at Xe.

If Ys is a point in the Y axis , Ys Є [0, O × SQRT(N)) , Ys mod O equals zero -> we have SQRT(N) objects stacked over the Y axis and start at Ys.

If Ye is a point in the Y axis , Ye Є (0, O × SQRT(N)] , Ye mod O equals zero -> we have SQRT(N) objects stacked over the Y axis and end at Ye.

Let's take the indexing point ip=i in the X-axis direction. We need to estimate the size of the data block associated with ip=i. All objects that pass through the line X=i will appear in i.datablock. The worst case happens if X ≠ 0 and X ≠ O × SQRT(N). In this case, we have SQRT(N) objects starting at X=i and SQRT(N) objects ending at X=i. The data block pointed to by ip in the StartX B^+-tree holds SQRT(N) tuples at the ip while the EndX B^+-tree holds SQRT(N) tuples at ip. To access the data block in one of the trees we need [((SQRT(N) * Tb) + (2* P)) / G] IOs. To access the four data blocks in the four B^+-trees, we need:

$$4 * [((SQRT(N) * Tb) + (2* P)) / G] \quad (6)$$

Analytical Performance Comparison:
This section compares the performance of B^+-tree versus SB^+-tree with respect to the BN problem. Both types are paged secondary-memory structures. The SB^+-tree is used as a solution for many spatial operations such as join, region overlap, distance and direction. Our proposed solution uses four B^+-trees whereas two SB^+-tree are required to solve the BN problem (one SB^+-tree for the X axis and another for the Y axis). The comparison assumes a spatial grid of SQRT(N) × SQRT(N). Using SB^+-tree, we have SQRT(N) indexing points for the X axis and SQRT(N) indexing points for the Y axis. The height of the SB^+-tree will be $\log_{M*F}((SQRT(N) + 1) / (M * F))$. Responding to a BNO query requires traversing the SB^+-trees four times. Therefore, the required number of I/Os is:

$$4 * \log_{M*F}((SQRT(N) + 1) / (M * F)) \quad (7)$$

The second step is to estimate the size of the data bock that is indexed at an indexing point ip=i in the X-axis direction. Data blocks in SB^+-trees have three pointers; one points to the data block associated with the next indexing point, the second points to the data block associated with the previous indexing point and the third points to the overflow data block associated with the same indexing point. We have SQRT(N) objects starting at X=i and SQRT(N) objects ending at X=i. All 2*SQRT(N) objects are in the data block pointed to by ip. Accordingly, the number of I/Os required to access the data block is $[(2* SQRT(N) * T_{sb} + (3 * P)) / G]$, where T_{sb} is the size of a tuple in the SB^+-tree. To solve the BN problem, the number of I/Os required to access the data blocks is given by:

$$4 * [(2* SQRT(N) * T_{sb} + (3 * P)) / G] \quad (8)$$

Consequently, the total number of I/Os required to solve the BN Problem using SB^+-tree is:

$$4 * (\log_{M*F} (SQRT(N) + 1) / (M * F) + [(2*SQRT(N) * T_{sb}+(3 *P)) /G]) \quad (9)$$

However, the total number of I/Os required to solve the BN problem using B^+-tree is:

$$4 * (\log_{M*F}((SQRT(N))/(M * F)) + [(SQRT(N)*T_b+(2*P)) /G]) \quad (10)$$

Consequently, the estimated performance of B^+-tree is at least 50% better than that of the SB^+-tree when solving the BN problem.

5 Experimental Results

In this section, we experimentally study the performance of B^+-tree with respect to the BN Problem. The performance of B^+-tree is compared with that of SB^+-tree since both of them are secondary-memory structures which are appropriate for databases. We have implemented the algorithms for the BN problem in Java, and the experiments were conducted using the US Census Bureau; i.e. the Tiger Collection. The response time were obtained in terms of the number of IO pages needed to perform the query. Each experiment was repeated several times for various objects in the space and the average number of I/O's was calculated. Fig. 6 demonstrates the performance of B^+-tree versus SB^+-tree in solving the BN problem. For these particular experiments, datasets were picked randomly from the Tiger database in order to compare both structures for various numbers of objects in the

space. B^+-tree consistently demonstrates performance superiority over SB^+-tree for various disk page sizes and large input space.

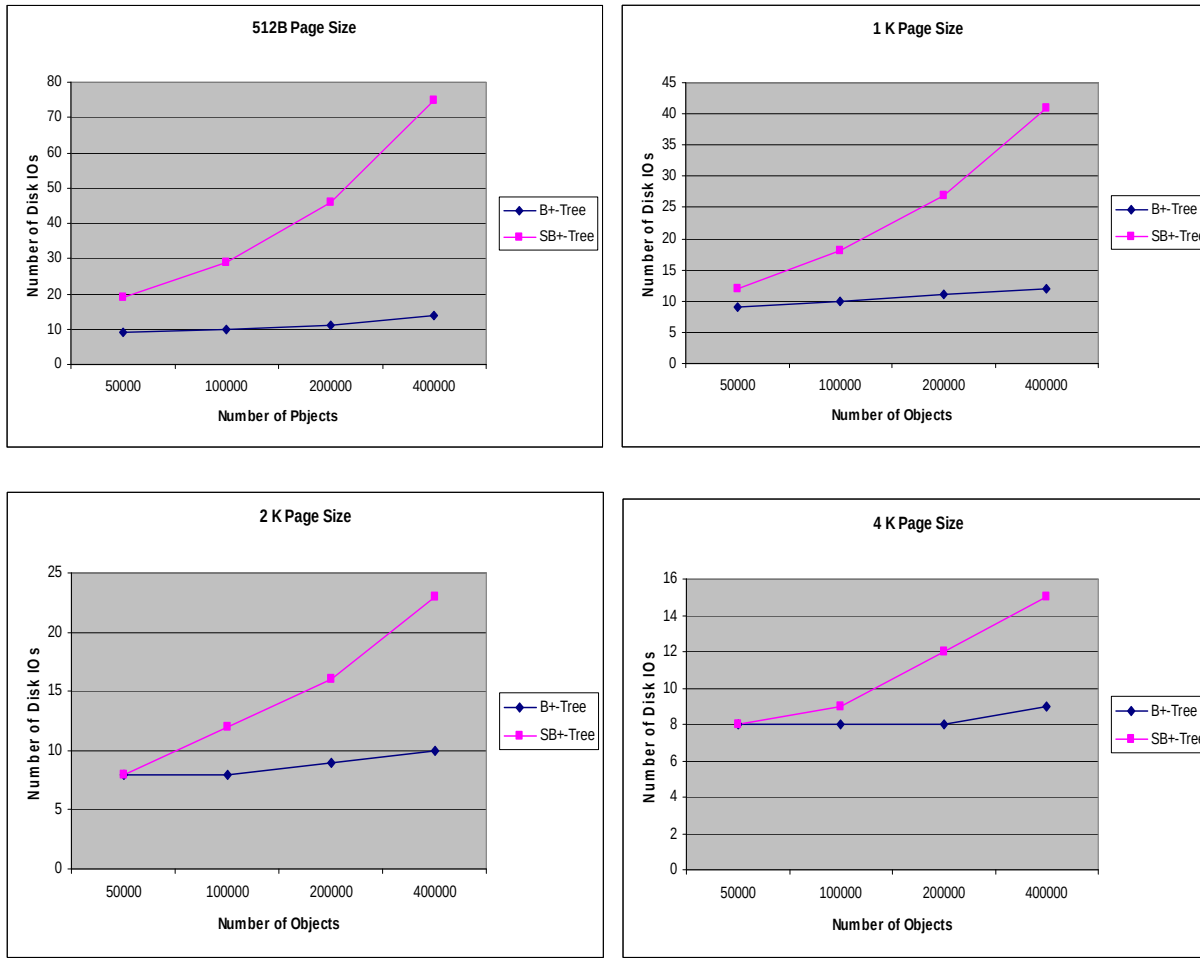

Fig. 6: Performance of B^+-tree versus SB^+-tree in terms of number of objects for various page sizes (experimental).

Fig. 7 illustrated the ratio (number of IO's using B^+-tree/ number of IO's using SB^+-tree) in terms of number of objects in the space for various database page sizes. Our results show that both structures have similar performance for smaller number of objects and smaller page size. However, B^+-tree displays superiority over SB^+-tree when the number of objects increases in space. This is attributed to the higher efficiency of B^+-tree at the data blocks level when the density of objects in space increases. Fig. 8 compares experimental results versus the previously suggested average-case analysis of B^+-tree presented in the previous section. The fullness factor was taken to be 0.67. The proposed solution has a minimum threshold for number of IOs which equals to 8 IOs; four IOs to search through the four B^+-trees and four IOs to access the four data blocks in the trees.

The minimum threshold occurs when:

$$[\log_{M*F}((SQRT(N))/(M * F) <= 1] \text{ AND } [(((SQRT(N))\times T_b)+(2*P))/G <=1] \quad (11)$$

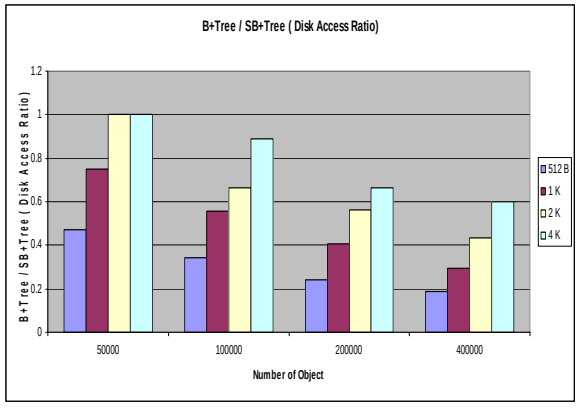

Fig. 7: I/O Ratio of B^+-tree versus SB^+-tree in terms of number of objects for various page sizes (experimental).

The situation in (11) experimentally occurs for small number of objects relative to page size. Analytical and experimental results tend to coincide for page sizes 2 and 4.

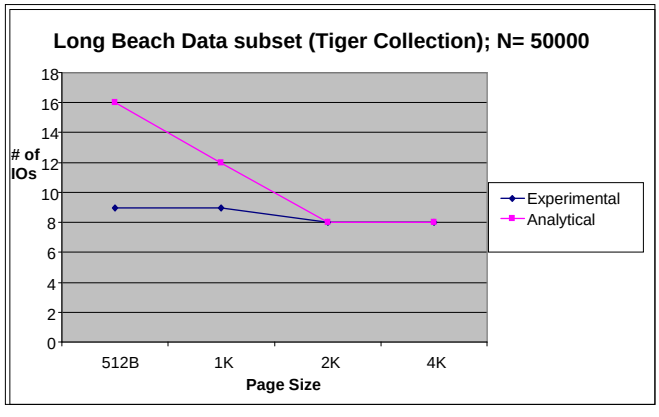

Fig. 8: Estimated versus Analytical Performance of B^+-tree

6 Conclusion

The boundary neighborhood (BN) problem has not received much attention in the literature. Most spatial indexing structures tend to perform well when solving the window operation. This paper presentes a solution for the BN problem using B^+-tree, which is widely acceptable as the indexing structure for conventional and non-conventional databases. The proposed solution is based on two steps. Firstly, a filter step which employs the index to retrieve all MBRs that satisfy the query and possibly some false hits. Secondly, a refinement step uses the exact geometry of the objects to dismiss the false hits. Experimental results show the superiority of B^+-tree over SB^+-tree in solving the BN problem. B^+-tree out performs SB^+-tree by at least 50% in terms of number of pages to be accessed. Although, the proposed solution was discussed for 2-dimensional space, it can be utilized for higher dimensions. Finally, the BN problem can be further investigated as a spatial join operation.

References:

[1] D. Comer, The ubiquitous B-tree, *ACM Computing Surveys*, Vol. 11, 1979, pp. 121-137.

[2] A. Guttman, R-trees: a dynamic index structure for spatial searching. *In Proc. ACM SIGACT-SIGMOD Conf. Principles Database Systems*, 1984, pp. 569-592.

[3] T. Sellis, N. Roussopoulos, N., and C. Faloutsos, The R+-tree: a dynamic index for multi-dimensional objects, *In Proceedings of the Thirteenth International Conference on Very Large Databases (VLDB)*, 1987, pp. 507-518.

[4] H. Samet, Hierarchical Representations of Collections of Small Rectangles, *ACM Computing Surveys*, Vol. 20, No. 4, 1988.

[5] J. A. Orenstein, Redundancy in Spatial Databases, *In Proceedings of the ACM SIGMOD Int'l Conf. Management of Data*, 1989, pp. 294-305.

[6] R. H. Guting, An Introduction to Spatial Database Systems. *VLDB Journal*, Vol. 3, 1994, pp. 357-399.

[7] H. Samet, Spatial data structures, *Modern Database Systems (ACM Press and Addison-Wesley)*, 1995, pp. 361-385.

[8] W. G. Aref and H. Samet, A Window Retrieval Algorithm for Spatial Databases Using Quadtrees, *In Proceedings of the 3rd ACM Workshop on Geographic Information Systems*, 1995, pp. 69-76.

[9] A. Ibrahim and F. Fotouhi, Indexing and retrieving point and region objects, *SPIE*, 2670, 1996, pp. 321-336.

[10] A. Ibrahim, F. Fotouhi, and S. Hasan, The SB^+-tree: an efficient index structure for join spatial relations, *International Journal of Geographical Information Science*, Vol. 11, No. 2, 1997, pp. 163-182.

[11] A. Ibrahim, F. Fotouhi, and A. AL-Badarneh, Efficient Processing of Spatial Selection and Join Operations using SB^+-tree, *International Database Engineering & Applications Symposium*, 1997, pp. 279-288.

[12] B. C. Ooi and K. L. Tan, B-trees: bearing fruits of all kinds, *In Proceedings of the 13th Australasian Database Conference, IEEE CS Press*, 2002.

[13] H. Samet, Depth-first k-nearest neighbor finding using the MaxNearestDist estimator, *In Proceedings of the 12th International Conference on Image Analysis and Processing*, Mantova, 2003, pp. 486- 491.
[14] H. Samet, Foundations of Multidimensional and Metric Data Structures. (Morgan-Kaufmann), 2006.
[15] M. Wu, C. Lin, and C. Chang, A Color Re-Indexing Scheme using Genetic Algorithms, *WSEAS Transactions on Systems*, Vol. 5, No. 6, 2006, pp. 1309- 1314.
[16] P. W. C. Prasad, A. Assi, B. Mills, Binary Decision Diagrams: A Mathematical Tool for the Path-Related Objective Functions, *WSEAS Transactions on Systems*, Vol. 5, No. 12, 2006, pp. 2868-2875.
[17] J. Ferreira, M. Crisostomo, and A. P. Coimbra, Walk Control of a Biped Robot using Neuro-Fuzzy, *WSEAS Transactions on Systems*, Vol. 5, No. 12, 2006, pp. 2892-2898.