SPATIAL R-TREE INDEX BASED ON GRID DIVISION FOR QUERY PROCESSING

Esraa Rslan[1], Hala Abdel Hameed[1] and Ehab Ezzat[2]

[1]Faculty of Computers and Information, Fayoum University, Egypt
[2]Faculty of Computers and Information, Cairo University, Egypt

ABSTRACT

Tracing moving objects have turned out to be essential in our life and have a lot of uses like: GPS guide, traffic monitor based administrations and location-based services. Tracking the changing places of objects has turned into important issues. The moving entities send their positions to the server through a system and large amount of data is generated from these objects with high frequent updates so we need an index structure to retrieve information as fast as possible. The index structure should be adaptive, dynamic to monitor the locations of objects and quick to give responses to the inquiries efficiently. The most well-known kinds of queries strategies in moving objects databases are Rang, Point and K-Nearest Neighbour and inquiries. This study uses R-tree method to get detailed range query results efficiently. But using R-tree only will generate much overlapping and coverage between MBR. So R-tree by combining with Grid-partition index is used because grid-index can reduce the overlap and coverage between MBR. The query performance will be efficient by using these methods. We perform an extensive experimental study to compare the two approaches on modern hardware.

KEYWORDS

MOD, Spatial indexing, R-tree, grid.

1. INTRODUCTION

Moving object applications have become very important in our life and have a lot of usages, such as GPS car navigator and location-based services. Moving object database is a branch of the spatial-temporal database research that has been developed recent years, indexing moving object is one of the key technologies of the moving object database. Tracing the changing positions of objects has become a serious matter and played an essential role in location-based services, such as customers with cell phones and cars. These services need to know the current positions of objects quickly. The index structure should be adaptive and dynamic to keep track of the positions of objects and to provide quick answers to the expected queries [1].

An example for this is the traffic observing service for a population of 10 mega moving objects that are processed with speed 15 m/s. Another example is related to expecting objects positions of an exactness of no less than 100 meters. In this situation, we need up to 1 mega update rate every second. As the update frequency is growing, the workloads are also increasing. High workloads in Sidlauskas's paper can't be managed by traditional disk-based techniques [2]. Rather, we propose a primary memory index that is compatible with the characteristic of parallelism accessible in multicore processors. The main goal of the index structure is to keep up a reliable database and get back the right results.

Indexing techniques are divided into two main groups: tree based and grid based index. Examples of tree based index are R-tree and its variations R+tree, R*tree, TPR-tree and TPR*tree (Time

Parametric Rectangle) [3]. Examples of grid based index are a uniform, parallel, hierarchical, and LU (Luzy Update) grid. There are also many kinds of queries, such as point query and density query.

The R-tree is famous for its efficiency to support information skew and various query types. It can be used to index moving objects in a specific region. The idea of the index structure is to utilize the Minimum Bounding Rectangle (MBR). In any case, due to overlapping between MBRs, the search might be necessary along different paths causing the exceeding of search paths [4]. With the expansion in the amount of index data, it will require more storage space. Subsequently, it will truly influence the search and update efficiency. Though R-tree is known for its power to support various query types, the main problem with the R-tree is its bad update leverage. On the other hand, using a grid index in moving objects is considerably faster than an R-tree in an update and query processing. It is obviously discernible and it can straight forwardly calculate all objects overlapping with the range query with no need to access the index entities.
In this paper, a robust and scalable index technique is proposed. The main goal is to build a hybrid index technique that supports a different number of queries with the minimum cost update. We used the R-tree and grid index together by using R-tree at the root node and grid at the leaf node to achieve two main benefits. The important benefit is to take the advantage of the two indexes as the R-tree is good for searching and querying. Also, grid is efficient in update operations. Experimental studies have shown that the design of the index may help provide answers to range queries with a great number of moving objects. To verify the efficient and effectiveness of the three types of indexes (R-tree, Grid, and hybrid R-tree and Grid), we explore several experiments to make a comparison between them based on the response time with a range queries and the CPU time in updating objects.

The remaining parts of this paper are organized as follows: in section two we introduce the background and related work; the aim herein is to briefly explain the R-tree structure and grid. In section three, the proposed index technique and associated algorithms are explained. In section four, an analysis of the index structure and the experimental results are shown. The conclusion is presented in section five.

2. Background And Related Work

In this section, we introduce two main types of moving objects indexing techniques: tree and grid based indexes.

2.1. Tree Based Index

R-tree is a height balanced tree (all leaf nodes are at the same height). It consists of internal nodes and leaf nodes. The R-tree is known for its robustness to data skew and it supports different types of queries. Yet, it is poor in update performance. Numerous applications including indexing the location of moving objects display workloads described by updates expansion to continuous queries. Indexes are divided into two categories based on data to be stored: (1) indexing the historical location of objects and (2) indexing the current location and predicting the future location of moving objects [3].

An overview of spatial-temporal access techniques can be found in the two papers [5,6]. The two surveys focus on the indexing of past, current, and future objects' positions. Indexes are divided into two parts: the first part is from 1980 to 2003, such as the TPR-tree, the TPR*-tree, the Bx-tree, and the Bdual-tree, STRIPES definition of spatial index is “the primary key to improve the efficiency of queries”; the second part dates back to the period from 2003 to 2010, such as MON tree, RTR-tree TPR-tree, Bx-tree and ST2B-tree.

Guttman[7] first presented an index structure called R-tree for spatial searching. It's known that R-tree is inefficient in update performance because of overlapping and the splitting of MBR, though it is efficient in supporting different types of queries. Six variants of R-tree variants are compared in Manolopouloset al.[8]. The most famous spatial techniques that utilize minimum bounding rectangles are R-trees [6]and their variation R-trees. In TPR-tree that expands the R-tree as objects move in linear motion function. Tao introduced the TPR*-tree that enhances the TPR-tree with advanced insert, update, and delete process. R-tree, TPR-tree, and TPR*-tree have low efficiency in updates because of their node splitting and reinsertion operations. Other than R-trees, B-trees, and Quadtrees[9] ,some indexes can likewise be utilized to moving entities. The B+-tree, proposed by Jensen et al.[10] utilizes space filling curves like Hilbert curves or Z-curves that are used in mapping dimensional locations.

2.1.1 Structure

R-tree consists of internal nodes and leaf nodes [11]. The internal node has pointers to other nodes; leaf node has pointers to objects (entries). As shown in figure 1, R-tree satisfies the following properties:

1. Every leaf node has between *m* and *M* entries unless the node is the root node, where *M* is the maximum set of records in a node, *m* is the minimum number of records that can be held in a node and let *m<=M/2*.

2. For every index entity *(I, tuple-identifier)* in a leaf node, *I* is the smallest rectangle that has the object and *tuple-identifier* is a unique number that identifies the record in the database.

3. Each non-leaf node contains between *m* and *M* children.

4. For each entry *(I, child-pointer)* in a non-leaf node, *I* is the smallest rectangle that overlays all the rectangles in the child node, the *child-pointer* is the address of the lower node.

5. The root node contains at least two child nodes unless it is a leaf.

6. All leaf nodes should be at the same grade.

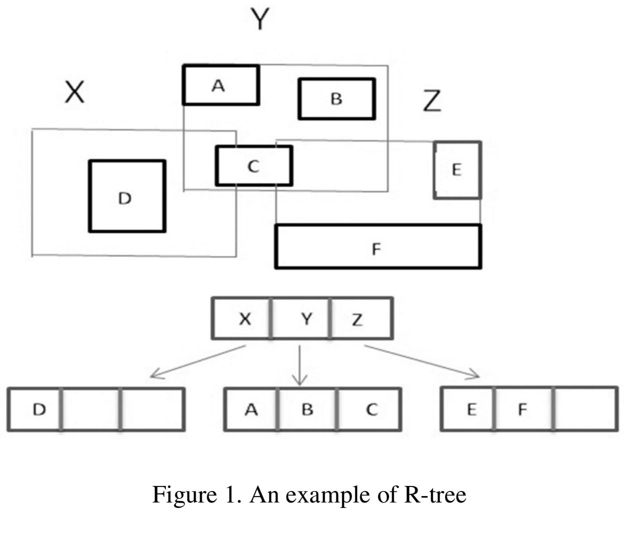

Figure 1. An example of R-tree

Figure 2 shows the structure of R-tree in on every node, there are *Node Size* (*ns*) number of each node and *Minimum Children* (*mc*) a fraction of the full nodes which are the essential attributes in the R-tree [12]. A leaf node contains node metadata like a number of entries, MBR, leaf flag, the id of the parent node, a number of leaf entries and a pointer to the parent node at the end. Likely, internal node consists of the same metadata fields in the leaf node, but the differences between them are the child pointer and MBR, where the MBR is the smallest rectangle that gathers the spatial entities together.

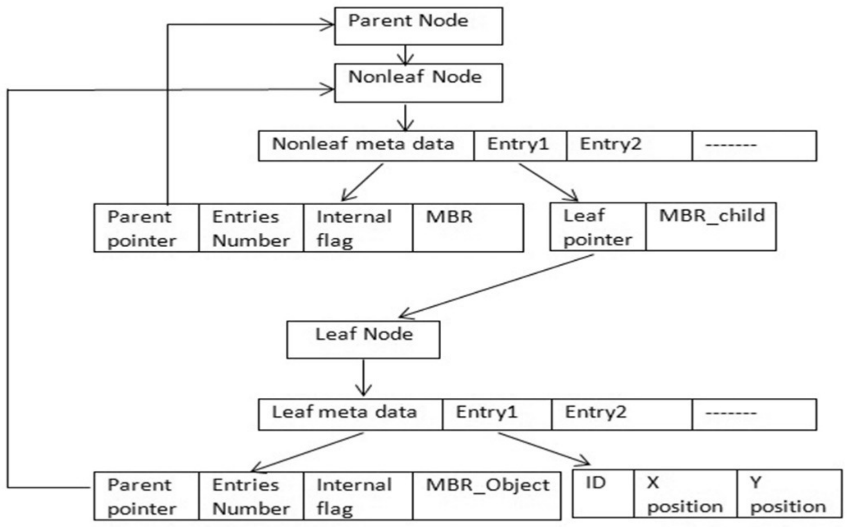

Figure 2. Structure of R-tree

2.1.2 Update, Insertion, and Deletion

Every update operation deletes the object from place and inserts it in another one. Assume the point *P* (*oldX*, *oldY*) is needed to be updated; first we search the tree from root to leaves until we find the node with MBR that have point P (*oldX*, *oldY*), more than one path are visited because of the overlapping between MBRs, then delete the point, but be careful that no underflow occurs (less than the minimum number of entries, often denoted as *m*). After the deletion operation is done, the insertion is applied. The insertion traverses the tree from root to leaves to find the suitable leaf node and insert (*newX*, *newY*). Insertion may cause overflow node (extends the maximum number of objects denoted as *M*). There are many variations of the R-tree in the literature; the uniform R-tree in is only considered.

2.1.3 Query Processing

Two main types of queries are discussed: Range Query and K-nearest neighbour query. A range query is processed by accessing the tree from the root to the leaves if the present node is not a leaf; check every internal node MBR that intersects or overlaps with the MBR of the range query.

2.2 Grid Based Index

Grid is a pre-defined partitioning index in which the region is partitioned into rectangular cells because it indicates the pre-defined spatial region. Grid covers a part of the space that object position is inside the boundaries of a grid cell and this object belongs to this cell. Šidlauskas*et al* [12] proposed the grid index which is one of the principal framework index structures for moving objects database. D-Grid [13] enhances U-Grid which it indexes both the location and velocity of moving objects in the dual space. The dual space is a 2-dimensional Euclidean space, with a first dimension for velocity and the other for location. Another index structure that is based on grid cell and continuous k-nearest neighbor query processing is proposed by Wei Zhang [14].

Existing Concurrency Control (CC) methods for multidimensional indexes are also introduced like Šidlauskas*et al.* [15] who proposed the TwinGrid and the PGrid[16]give high query and update performance. PGrid makes various duplicates of information on a request and at the object granularity (i.e., non-local objects).

2.2.1 Structure

Grid is a 2-d array that every cell in the array matches to a square cell with a length of grid cell size. Every grid cell in the array has a link to a list of buckets which contains the data object [12]. Each bucket has bucket size *bs* and metadata fields where metadata contains *the next bucket, the number of entries and pointer to the next bucket*. Grid is defined by two main parameters: *gcs*, and *bs*.

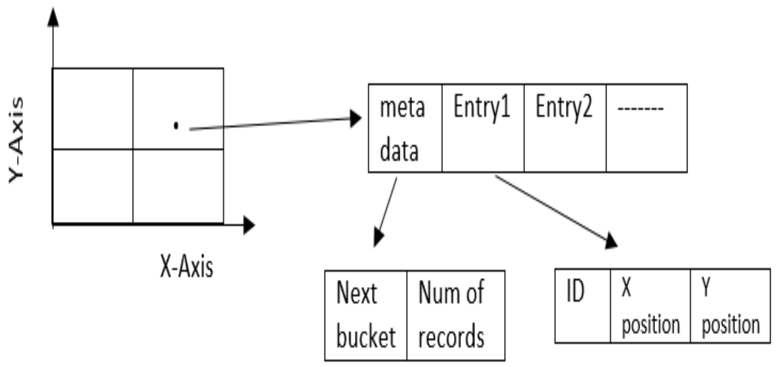

Figure 3. Grid index structure

2.2.2 Update, Insertion, and Deletion

All updates operations are classified into two sets: local and nonlocal. The update is considered local when the new object location falls in the same cell and we need only to change the (*oldX*, *oldY*) with the (*newX*, *newY*). Otherwise, the update is non-local (new location not in the same cell) as the object will be deleted from its current cell, and it will then be inserted inside the new cell. During the insert operation, the new object is always put at the end of the first bucket. In the condition that the first bucket is complete and has no space, a new bucket is created and the pointers are upgraded to make the new bucket the first one. If the first bucket is empty, the next bucket becomes the first one.

2.2.3 Query Processing

The range query divides the cells, which overlap in the range query into two categories: the completely and the partly covered cells. Only the objects that are partly covered are needed to be listed in order to know if they are inside the range or not. KNN algorithm is known by a point *P* and the numbers of nearest neighbour *k*, cells are divided into different sets. Every set has the minimum range to the point P. Similar to the KNN query in the tree index, the queue is used to store these cells as they are sorted from the nearest to the farthest.

3. PROPOSED INDEX TECHNIQUE

R-tree and grid-partition index is proposed, which give execution enhancements in handling different types of queries like range and KNN queries. The proposed index has two main standard data structures: the first is the R-tree and the second one is a grid index. The R-tree is used to divide the space into MBRs then a grid is used in non-leaf nodes to store the data object and to reduce the overlapping between MBRs.In grid implementation, divide space into M x M grid of squares, create linked list for each square, use 2D array to directly access relevant square. It performs less maintenance effort as no grid changes are needed to be done in a grid updates.
It is known that if the number of objects increases a lot of overlapping and converging between MBRs will be generated, approximately; each single update will need about four tree traversals affecting the performance significantly. To reduce node splitting in the R-tree, a grid is used to store data objects at leaf level. The performance of grid in inserting, deleting, and updating algorithms is superior to the R-tree, so it can highly decrease the time complexity of an algorithm. Extensive and broad experiments reveal that our proposed index structure essentially outperforms and beats the existing grid index or R-tree index in single threaded environments.

As shown in figure 4, the index structure has two data structures: the root level is R-tree and each non leaf node is in a grid index structure, taking the advantages of both Tree and Grid index. R-tree is very powerful in supporting different types of queries with minimum time response. In addition, the grid is the best choice in updates because no grid re-adjusting should be done through updates, also grids require less repair effort and index speed is quick. We take the advantage of each one of these indexes in update and query processing. In a query range, grids can number all index objects that cover the range query with no need to access to the index records.

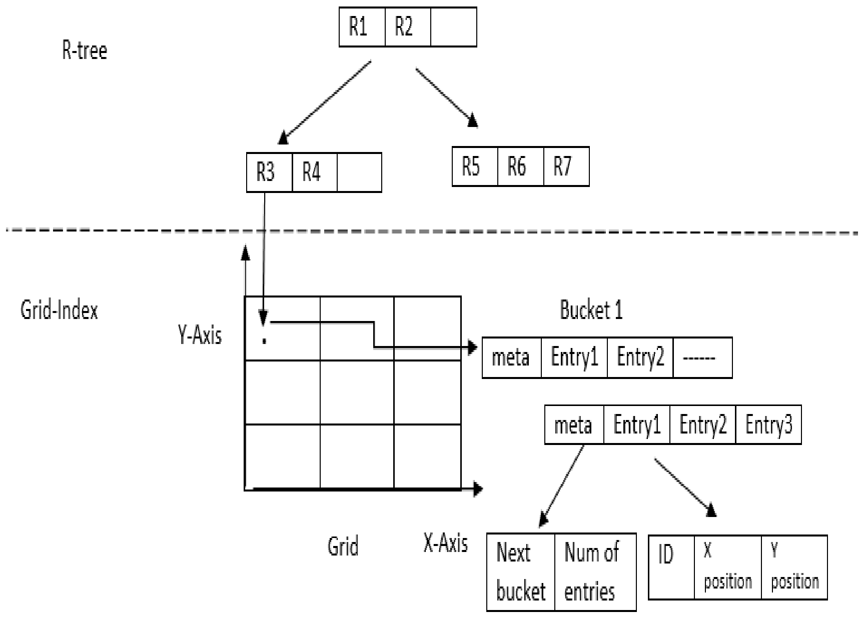

Figure 4. The new index structure

3.1 Index structure

The R-tree is used to implement both root and parent levels, which utilize the MBR to partition the regions, and child node uses a grid to implement MBR with equal sized cells. The grid needs insignificant update costs. Hence, no framework re-adjusting is done when objects change their locations. The index structure is as follows:

1. It has a root node in R-tree.
2. Non-leaf node contains the internal nodes that have the tuple <mbr, ptr> where the mbr is the minimum bounding rectangle of the node and ptr is an indicator to point to the next level node.
3. Leaf node is in grid coordination that divides the MBR into M x M cells, the grid is a 2-d array that every cell in the array matches to a square cell with a length of Grid Cell Size (gcs).
4. At leaf node, grid squares store a pointer to the list of buckets which have the object data. In a huge size of data, getting the relationships from the grid index can highly minimize the complexity of the algorithm in contrast to the conventional R-tree methods. Grid coordinate makes the position of spatial entity objects clear, so reduces the repetition of storage. Also, to avoid the search operation costs.

3.2 Update, Insertion, and Deletion

The R-tree operates every update as a companion of two processes: delete and insert, traversing the tree top- down to locate the suitable leaf and update it. Algorithm 1 is proposed for inserting the new object. The algorithm takes a new object as an input with a unique id, and then inserts it at the index. The first step, the algorithm searches the tree from the root t to leaf to find the suitable child node by searching in the node of min.x≦x≦max.x in x-dimension according to MBR coordinates (min.x, min.y, max.x, max.y) of this object. If it is true, go to the second step, then go to the fourth step. The second step, if the MBR is found, the insertion operation is done by inserting the object into the suitable grid cell based on (x, y) of the new object as the child node is implemented in grid coordination. The third step, the new object is always put at the end of the first bucket in the grid. If the first bucket is complete and has no space, a new bucket is created and the pointers are upgraded to make the new bucket the first one. Otherwise, the first bucket is empty, and then the following bucket is the first one. The first free location at the end of the bucket is located, and the new record is also inserted. The fourth step, if the no MBR is found, create a child node approximate to X-value, Y-value and its leaf node (*x, y*) and insert the new object into the grid expressed by the leaf node.

```
Algorithm1: insert (ObjectTuple new(n), Cell cell)
1. Search object n in tree
2. if tree is non-leaf // then for each entry in n do
3.     if EI overlaps n // true… insert new object in the index
4. firstBucket= *cell ; // the 1st bucket is equal to reference cell
5. if Full(firstBucket) // if firstbucket is full ..create new bucket and make it first ;
6. freePosition= firstBucket.entries[firstBucket.nO] // first free location of the first bucket is
   located
7. insertObject(n, freePosition); // the new object is inserted
8. elseCreate child node (ObjectTuple new(n)) //false…. No MBR found
```

The update process in algorithm 2 takes a new object (*newx, newy*) as an input. Based on its position, the tree is searched to find the MBR node that overlaps with the object. After locating the node that has the object to be updated according to its *oldx, oldy*. The information is retrieved from the primary index (the location of the object, current cell of object), then the algorithm determines the update type whatever is it local or not by contrasting the new and old positions. The out-dated object tuple is over-written with the new x and newly if the update is local as well, the old object tuple is deleted (call delete algorithm) and the new object tuple is inserted into the appropriate child node," update is non-local".

```
Algorithm 2: update(ObjectTuple new (new))
  1. Search in obj Tree t
  2. if EI overlaps obj
  3. SearchSubTree(EP, obj)
  4. newCell= computenewCell(newx, newy);// new cell is located for the object
  5. if newCell = = oldCell // Local update .. in the same cell
  6. overwriteObject(new, obj); // obj is overwritten new
  7. else // Non-local update
     detete (obj);//delete the old object tuple
         insert(new); // insert the new object and update the index
```

In algorithm 3, the delete operation is done as follows: First, search in the tree until finding the child node that contains the object that we want to delete. Second, as child node is implemented in grid coordination. Third, after finding the object the algorithm seeks to shift the last object of the first bucket into the place of the deleted object; all pointers (*pointer1, pointer2, and idx)* of the last object are modified accordingly in the secondary index. Fourth, if the first bucket was empty, it is erased and the following bucket is the first or the grid cell becomes empty, put a null pointer in the cell.

```
Algorithm 3: delete(ObjectTuple obj)
  1.  Searchobj in Tree t
  2.  if t is not a leaf //then for each entry E in t do
  3.  if EI overlaps obj
  4.  SearchSubTree(EP, obj)
  5.  firstBucket = * sindex.ldCell; // cell refers to the frist bucket
  6.  lastObject = firstBckt.entries[firstBckt.nO - 1];
  7.  if firstBucket.nO == 0 then // is empty?
  8.  * sindex.ldCell = firstBucket.next;
  9.  free(firstBucket);
  10. update all pointers of deleted object
```

3.3 Query Processing

The range query is explained and how it works.

3.3.1 Range Query

In range query, the resulting cells that overlap with the range query is divided into the completely and the partly covered cells. The objects that are partly covered are needed to be listed in order to know if they are inside the range or not. The input is the query box q (*x0, y0, x1, y1*) in which (*x0, y0*) and (*x1, y1*) are the lower-left and the upper-right corner coordinates respectively. First, the algorithm searches the node of x1 ≥x ≥x0 in dimension **x** based on the coordinates *(x0, y0, x1, y1)* of the q in the MBR of the child node; if it has the node of **x** value. Then, it searches each node of Max.y1 ≥ y ≥ Min.y0 corresponding to the x-value; if there exists the node of y-value, go to bucket number 1and increment the reader (*R*) of a bucket before coming into it and decrement it when the bucket has been examined. Finally returns the objects which are covered by the query range in the result set.

4. Experimental Results

In order to check the performance of the proposed technique, we compare it with both traditional R-tree and grid by providing an overview of the performance and showing the average CPU time of individual update, range and I/O visits.

4.1 Index Implementations

The index structure is implemented using C++ and compiled with g++ under the Linux. The test environment uses Windows Seven with 64-bit OS, Core (TM) i3-2.10 GHz, 6GB RAM. The indexes are studied under massive datasets; these datasets are generated using an Open-source Moving Object Trace generator (MOTO) that is based on Brinkhoff [17] moving-object generator. Objects move into a network-based roads in five German cities. Table 1 shows the Experimental settings of the generator; the default values are in bold.

Table1. The Experimental settings of the generator

Dataset Parameters	Values
Number of objects ($\times 10^6$)	5, 10, 20, 40, 80
Number of Updates ($\times 10^6$)	300
Monitored region (KM^2)	641 × 864
Velocity(kilometer/hour)	20, 30, 40, 50, 60, 90
Update/queryratio	250:1, …1000:1, …16000:1
Time through updates (s)	10, 20, 40, 80, 160
k nearest neighbor ($\times 10^3$)	0.5,1,2, 4, 8, 16, 32
# network segments	32,750,494
# network nodes	28,933,679

4.2 Comparison of the Indexes

A comparative experiment was done to compare the efficiency of the proposed hybrid index and the conventional R-tree and grid under different conditions and parameters. The different index behaviour of the total CPU update time and different number of moving objects are shown in Figure 5. When the number of objects exceeds, the new index outperforms both R-tree and the grid. With the Grid parameters, it still performs updates a little better and performs similar (range) query performance with the tree in figure 6.

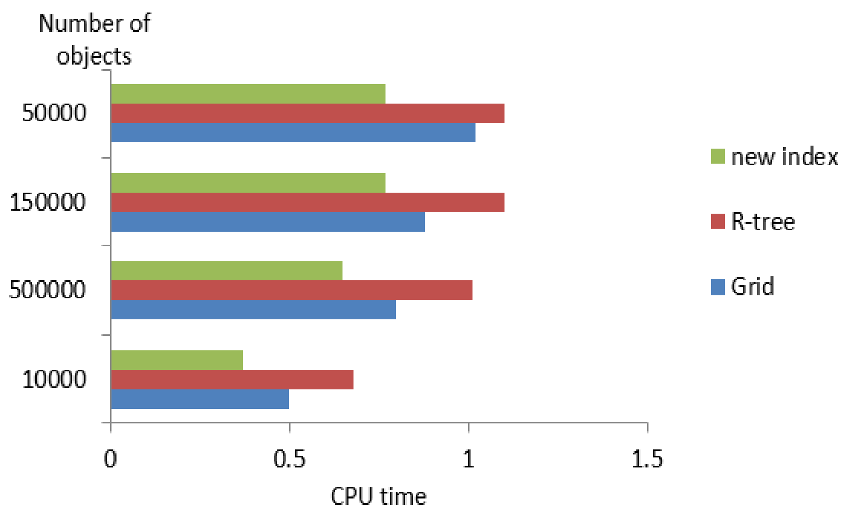

Figure 5. Increasing number of objects in update operation

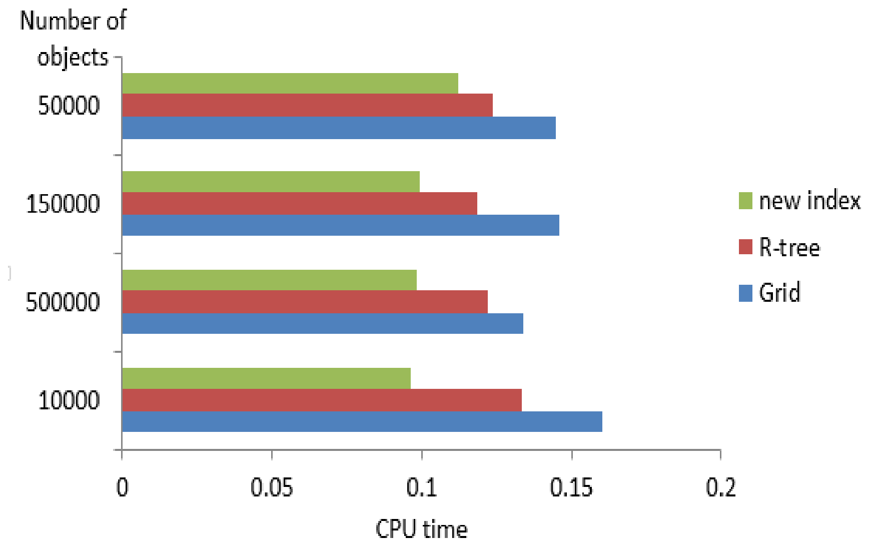

Figure 6. Increasing number of objects in range query

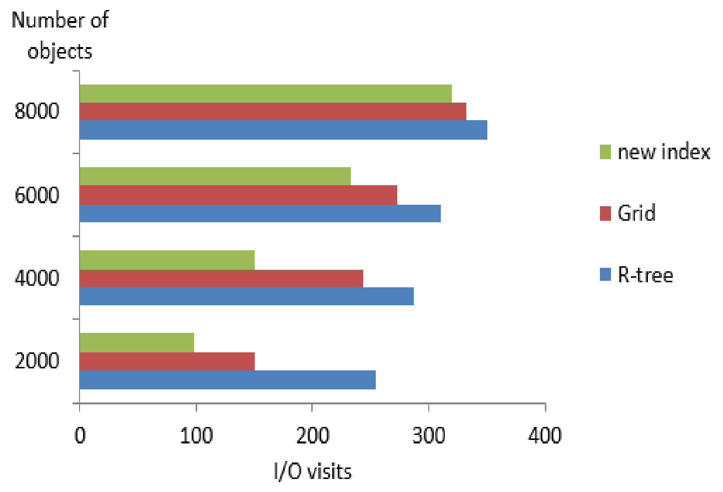

Figure 7. Number of I/O visits

I/O visits are shown in figure 7 when inserting in different number of spatial entity objects in queries.The results exhibit that the new index structure is outperforming than the conventional R-tree because after dividing the region into the first MBRs using R-tree, each MBR is partitioned using the grid. In querying, locate the grid cell connected to the MBR of the objects, because leaf nodes contain the data objects. The tree is built based on the (*x, y*) value of the grid cells, so it can speedily find the grid cell of the related object based on the MBR coordinates of the entity object and insert the object into the related leaf-node; if the MBR of this object not found in the tree, we create a new child node. In a huge size of data, getting the relationships from the grid structure can highly minimize the complexity of the algorithm contrast to the conventional method of looking for MBR containing the spatial entries.

5. Conclusions

A new spatial index technique based on spatial grid coordinates division is introduced, and the description of its data structure and algorithm is also given, we compared the hybridized index with the traditional r-tree index, and grid. The new index can reduce the overlapping and coverage between intermediate nodes. The important point in the new index is to retrieve quickly data and give speed answer effectively based on the current location like knowing the locations of schools, clinics, bazaars, mini-marts that can be used by various levels of people. The index structure is very flexible, so it can improve the retrieval performance of spatial entity objects, and has a better application in practice.

References

[1] W. Choi, B. Moon, and S. Lee, "Adaptive cell-based index for moving objects," Data & Knowledge Engineering, vol. 48, pp. 75-101, 2004.

[2] D. Sidlauskas, S. Saltenis, and C. S. Jensen, "Processing of extreme moving-object update and query workloads in main memory," The VLDB Journal The VLDB Journal : The International Journal on Very Large Data Bases, vol. 23, pp. 817-841, 2014.

[3] Y. Tao, D. Papadias, and J. Sun, "The TPR*-tree: an optimized spatio-temporal access method for predictive queries," in Proceedings of the 29th international conference on Very large data bases-Volume 29, 2003, pp. 790-801.

[4] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles," in Acm Sigmod Record, 1990, pp. 322-331.

[5] M. F. Mokbel, T. M. Ghanem, and W. G. Aref, "Spatio-temporal access methods," IEEE Data Eng. Bull., vol. 26, pp. 40-49, 2003.
[6] L.-V. Nguyen-Dinh, W. G. Aref, and M. Mokbel, "Spatio-temporal access methods: Part 2 (2003-2010)," 2010.
[7] A. Guttman, "R-trees: a dynamic index structure for spatial searching," presented at the Proceedings of the 1984 ACM SIGMOD international conference on Management of data, Boston, Massachusetts, 1984.
[8] Y. Manolopoulos, A. Nanopoulos, A. N. Papadopoulos, and Y. Theodoridis, R-trees: Theory and Applications: Springer Science & Business Media, 2010.
[9] Y. Xia and S. Prabhakar, "Q+ Rtree: Efficient indexing for moving object databases," in Database Systems for Advanced Applications, 2003.(DASFAA 2003). Proceedings. Eighth International Conference on, 2003, pp. 175-182.
[10] C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient B+-tree based indexing of moving objects," in Proceedings of the Thirtieth international conference on Very large data bases-Volume 30, 2004, pp. 768-779.
[11] Y. Gui-juna and Z. Ji-xianb, "A Dynamic Index Structure for Spatial Database Querying Based on R-Trees," in Proceedings of International Symposium on Spatio-temporal Modeling, Spatial Reasoning, Analysis, Data Mining and Data Fusion, 2005, pp. 27-29.
[12] D. Šidlauskas, S. Šaltenis, C. W. Christiansen, J. M. Johansen, and D. Šaulys, "Trees or grids?: indexing moving objects in main memory," in Proceedings of the 17th ACM SIGSPATIAL international conference on Advances in Geographic Information Systems, 2009, pp. 236-245.
[13] X. Xu, L. Xiong, and V. Sunderam, "D-grid: An in-memory dual space grid index for moving object databases," in Mobile Data Management (MDM), 2016 17th IEEE International Conference on, 2016, pp. 252-261.
[14] W. Zhang, J. Li, and H. Pan, "Processing continuous k-nearest neighbor queries in location-dependent application," Int'l Journal of Computer Science and Network Security, vol. 6, pp. 1-9, 2006.
[15] D. Šidlauskas, K. Ross, C. Jensen, and S. Šaltenis, "Thread-level parallel indexing of update intensive moving-object workloads," Advances in Spatial and Temporal Databases, pp. 186-204, 2011.
[16] D. Šidlauskas, S. Šaltenis, and C. S. Jensen, "Parallel main-memory indexing for moving-object query and update workloads," in Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, 2012, pp. 37-48.
[17] T. Brinkhoff, "A framework for generating network-based moving objects," GeoInformatica, vol. 6, pp. 153-180, 2002.

AUTHORS

Esraa Rslan was born in 1991. She received the B.S. degree in Faculty of Computers and Information from Fayoum University, Egypt, in 2012. She is M. S. candidate. Her research interests include Database Managements, Spatial Databases and Database Query Processing. Her E-mail is eaa06@fayoum.edu.eg

Hala Abdel Hameed Mostafa was born in 1977. She received the B.S. degree in Faculty of Computers and Information from Helwan University, Egypt, in 2000. She had a Ph.D. in Computer and Information Systems from Mansoura University, 2012 and Master of Computer Science from Helwan University, 2006. She is now Lecturer at Faculty of Computers and Information Systems, Fayoum University. You can reach her at ham07@fayoum.edu.eg

Ehab Ezzat Hassanein hold a Master degree in Computer Science from Northwestern University in 1989.A Ph.D. from Northwestern University in December 1992. Now he is Chairman of the Information Systems Department of Computers and Information Faculty, Cairo University, Egypt. He may be reached at e.ezat@fci-cu.edu.eg