A Fast and Compact Indexing Technique for Moving Objects

Yonghun Park
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
yhpark1119@gmail.com

Ling Liu
College of Computing
Georgia Institute of Technology
Atlanta, GA, USA
ling.liu@cc.gatech.edu

Jaesoo Yoo
Dept. Information and
Communication Engineering
Chungbuk National University, Korea
yjs@chungbuk.ac.kr

Abstract

Advances in ubiquitous connectivity and location sensing technology have fuelled a rich collection of location based services (LBSs). Efficient spatial indexing techniques are one of the most effective optimization methods to improve the quality of services. Although a variety of spatial index structures like R-tree family and grid variant index structures have been proposed and deployed in real time location based service provisioning systems, they are known to perform poorly when there is high degree of the skewedness in both density distribution and spatial resolution of mobile objects. First, it is hard to decide the optimal resolution of the grid structure. Second, it is equally hard to build a balanced R-Tree like index structure that is effective in handling highly skewed distribution of mobile objects. With these issues in mind, we introduce the concept of spatial order sequences and propose a fast and compact index structure for moving objects by utilizing spatial order sequences through a number of density-conscious optimizations. First, we propose the concept of Ordered-Cell Group (OCG) and design a OCG based grid index structure. Second, we speed up the search efficiency of OCGs by effective compaction of identifiers of OCG cells to maximize the fan-out of index node and decrease the depth of the index. Finally, we develop an efficient query processing algorithm that can effectively utilize OCG cells to speed up the processing of spatial queries. Our experimental results demonstrate the effectiveness of our approach compared to existing index techniques.

1. Introduction

The ubiquitous connectivity and wide deployment of mobile devices have made location aware computing and location based services (LBSs) a popular information technology in the recent years. LBSs provide mobile users convenient services for retrieval of location-dependent information, such as nearby gas stations, restaurants, transportation, and points of interest, and delivery of content to the mobile users on the road in real time. For example, with LBS, one can find the current location of a moving vehicle, the estimated time to reach a place, or where to meet their friends through location sharing services, such as Foursquare. As the number of mobile devices and the number of location based services grow rapidly, one of the most effective optimization techniques in scaling location based services is to employ efficient spatial indexing structures and algorithms for managing location information of moving objects and querying moving objects in real time.

A spatial index is considered effective if it can handle both retrieval and update of the location information of moving objects in real time within desired response time. However, the index structure that is efficient for querying moving objects may not be effective for frequent location updates and vice versa. Thus, the first requirement in designing an effective spatial index is the capability of handling both frequent location updates and fast processing of location queries in real time. The second requirement in designing an effective spatial index is the capability of providing high performance of LBSs no matter whether the moving objects follow the uniform spatial distribution or skewed spatial distribution. In real world, moving objects move continually and follow skewed spatial distribution. Thus, the result of a query tends to change continously according to the movement of the object such that the query is re-evaluated frequently to keep the accuracy of the result. Furthermore, when the density of mobile objects is changed unpredictably, the index structure should be self-tunable in response to the change of density.

Various spatial index structures have been proposed for managing and querying moving objects. We can categorize them into three classes: the Grid file index structure, the R-tree like indexing structure[1], and the B^+-tree variant Index structures based on the Grid file. Unfortunately, most of existing indexing structures and algorithms to date fail to meet both of the above-mentioned requirements.

Concretely, the Gird structure is one of the most popular main memory resident index structures for processing moving object queries and managing moving objects. However, Grid index suffers from two limitations. First, it is memory-resident and cannot handle very large datasets and thus it fails to perform when the size of the moving objects exceeds the memory boundary. Second, the performance of grid index is also affected by the relation between the resolution (cell size) and data distribution. If the resolution is too high, the storage cost and query processing cost become worse because most of cells include very small number of objects, with cell-based disk blocks, the storage utilization becomes low. In addition, when the size of cells is too small, the number of cells relevant to a query will be increased, which increases the number of disk I/O. If the grid resolution is too large, then the query processing cost becomes worse. This is because when the size of cell is too large, each cell includes too many objects. When a query is overlapped with a large cell, all objects in the cell will be retrieved and evaluated against the query. Therefore, it is a known challenge for grid structure to decide the optimal resolution.

The second and third categories of index structures are disk-resident. Many R-tree variant index structures have been proposed to date, such as R*-tree[2], TPR-tree[3] and TPR*-tree[4]. They are efficient for real time query processing but suffer from high update cost due to frequent splitting and merging of index nodes in the presence of high rates of location updates, which is common for moving objects. To overcome the update performance problems, several research proposals put forward the efforts of employing Grid file index structure, a memory resident indexing technique or employing Grid file in conjunction with B$^+$-tree. Representative examples include B^x-tree[5], B^{dual}-tree[6] and ST^{2}B-tree[7]. They first overlay the index space with a grid which divides the index space into the same-sized cells. Then, instead of directly using cell based inverted index structure like the Grid file, it will sort the cells using an ordering method such as Hilbert curve or Z-order curve [9]. The cells are used as data nodes in the index structures and are managed by B$^+$-tree index. Only ST^{2}B-tree considers various density distributions, but it still suffers from the limitation of deciding the appropriate resolution of the grid.

In this paper, we propose a fast and compact index structure for moving objects, called Ordered-Cell Group index (OCG-index for short), which is efficient even when the density of moving objects is skewed. The OCG-index has a number of unique features. First, we introduce the concept of spatial order sequences and the concept of Ordered-Cell Group. By utilizing the Ordered-Cell Group (OCG), we make better utilization of the storage space by assigning each OCG to a disk block instead of mapping a cell to a disk block. This guarantees the number of objects in a leaf node and enables us to reduce the depth of the index. It leads to save both the storage cost and query processing cost. Second, we design a OCG based grid index structure by utilizing spatial order sequences through a number of density-conscious optimizations. Third, we speed up the search efficiency of OCGs by effective compaction of identifiers of OCG cells to maximize the fan-out of index node and decrease the depth of the index. We observe that there is an opportunity to compress the identifiers of cells using techniques like prefix-tree[8]. By compression, we can increase the fan-out of a node and reduces both the storage cost and query processing cost. Finally, we develop an efficient query processing algorithm that can effectively utilize OCG cells to speed up the processing of spatial queries. Our experimental results demonstrate the effectiveness of our approach compared to existing index techniques.

The rest of paper is organized as follows. Section 2 describes the related work and defines the problems. Section 3 introduces the proposed index structures and section 4 shows the performance evaluation between the proposed index and other index structures. Finally, section 5 concludes the paper and mentions the future work.

2. Background and Related Work

Ordering methods, such as Z-order curve and Hilbert curve, are two of the most popular space filling curves, which frequently used for linearization of multidimensional data. A key property of these ordering functions is that it can map multidimensional data to one dimension while preserving the locality of the data points. These ordering methods were originally introduced in [10]. Once the data is sorted according to these ordering, any one-dimensional data structure can be used for indexing the data, such as binary search trees and B-trees. Hilbert curve is proposed to increase the locality of the sequence in comparison with Z-order.

Grid structure divides the data space into cells of the same size according to the resolution. It works like a hash function, which can quickly find the cell in which a moving object resides. Recently, several research efforts use grid structure and cell ordering sequences to store the data of moving objects inside a B$^+$-tree, which shows to improve the update problem of R-tree variant index structures while maintaining good query processing performance for moving objects [5][6][7]. B^x-tree[5] is the first index structure using grid structure managed by B$^+$-tree with ordering sequences such as Z-order curve and Hilbert curve. Each cell has identifier on the ordering sequences and is put into B$^+$-tree. B^x-tree manage at least two of the B$^+$-tree according to the time sequence to process the future trajectory queries efficiently. It helps to reduce the search boundary of queries. And also, the link of the leaf node to next node in B$^+$-tree is to save the cost traveling index when processing queries. B^{dual}-tree[6] also has similar approach to B^x-tree. However, B^{dual}-tree improves the query processing time by reducing the search boundary, compared to B^x-tree.

Both B^x-tree and B^{dual}-tree are based on grid structures. However, they do not consider the various distributions of objects. To solve the problem, ST^{2}B-tree[7] is proposed. ST^{2}B-tree divides the index space according to the object distributions. They manage a global grid structure and the number of objects contained in each cell, and distinguish the regions that have similar density by image processing algorithm. Then, they assign B^x-tree for each region with appropriate resolution for its density. However, it cannot handle the environment with highly diverse density distributions. Also ST^{2}B-tree suffers from the same limitation of resolution as the Grid file index approach because of the cost of managing and adapting the global grid structure to the density of moving objects.

We define the problems of previous work. The first problem is on deciding an appropriate resolution for a grid structure. ST^{2}B-tree tried to solve the problem by clustering the areas having similar density but it has the limitation in the environment with highly skewed objects as we mentioned above. The second problem is on the size for the cell identifiers in a grid structure. By using the identifiers of fixed size, the size of the identifiers depends on the resolution of grid structure regardless of how many empty cells are in the grid. Thus with a grid with high resolution (smaller cell size), the cell identifiers can be quite large and non-negligible in terms of storage and query processing cost. This observation motivates us to group the cells according to the number of objects contained in the cells, such that we map order cell groups to data nodes of the index instead of cells. By reducing the number of nodes in the index structure, it leads to significant saving in both the storage cost and query processing cost.

3. The OCG-Index Structure and Algorithms

In this section we first introduce the basic concepts involved in defining our OCG-index structure and then describe the query processing algorithm for the OCG-index.

3.1. Ordered Cell Group (OCG)

Given a road network and its grid overlay, we first utilize an ordering method, either Z-order curve or Hilbert curve, to group cells into groups such that each group contains equal or similar number of objects. We call such groups the Ordered Cell Groups (OCGs). We assign each OCG a unique identifier using the list of cell identifiers. The equation 1 computes the set of cells that form an ordered cell group g and the identification of g, where G is a set of all OCGs, $g.id$ is the identifier of an instance g of G, C is a set of all cells in grid, $c.id$ is the identifier of a cell c, and N_{fanout} is the number of fan-out of a data node.

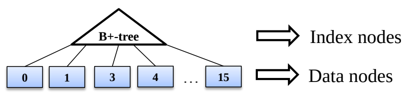
$$g_i.id = \{c_n.id | N_{fanout} \geq \sum_{j=n}^{m} |c_j.O|, c_j \in C, \\ c_j.id < c_{j+1}.id, 0 \leq n \leq m < |C|\}, g_i \in G \tag{1}$$

Figure 1 provides an illustrative example to show how to decide OCGs using Z-order for a grid with the resolution of 4 by 4. The number of cells is 16, i.e., $|C|$ = 16. Figure 1(a) presents the naive approach to indexing moving objects using a grid structure. Figure 1b, Figure 1c, and Figure 1d present a scenario where moving objects are changing their locations at the time t_0, t_1 and t_2 and how to decide OCGs in each of these three time instances: t_0, t_1 and t_2. Suppose that the fan-out of data node is 6. In Figure 1b, there are 6 objects in whole data space at t^0. In this cast, there is just one OCG g_0 in the data structures and the identifier of g_0 is 0. At t^1, an object is added into the cell 2 and g_0 is split because of overflow. To split a g_0, we first find the middle object in the ordering sequence and then find the cell overlapped with the middle object. The identifier of the found cell becomes a split key. In this example, the split key is 7. We create a new OCG g_1, put all objects lying in the cells whose the identifiers are bigger than 7 into g_1 and remove these objects from g_0. Finally, we insert g_1 into the index. At time t_2, when a few objects are inserted to g_0 and overflow occurs at t_2. Thus, 4 is selected as a split key, and a new OCG g_2 is created by splitting g_0 again. The identifiers of g_0, g_1, g_2 are 1, 7, 4 respectively. Our OCG-index uses OCGs as the index entries, instead of indexing every cell as the entries in the index as done in the naive approach, the number of OCGs inserted into the index is much smaller as shown in Figure 1. By utilizing OCG as the unit of indexing, we no longer need to keep every cell identifier in the index, which significantly reduces the size of index and enhances the search efficiency of index.

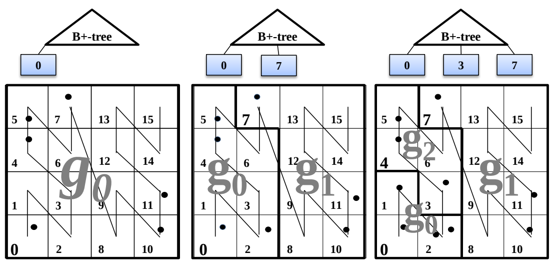
3.2. Computing the cell identifier on OCG

In this section we describe how to get an identifier of a grid cell given an Z-order group. In this paper, we use two typical ordering methods, Z-order and Hilbert curve. Hilbert curve follows 'U' shape while Z-order follows 'Z' shape. Given that Hilbert curve is proposed to increase the locality of the sequence in comparison with z-order, we first describe how to obtain Z-order curves of a multidimensional space and the properties of Z-order curves.

Z-order curve by design splits data space into 2^n (n is the binary bits of Z-value) number of equalized and disjoint subspace and the Z-value of a point in multidimensional space is simply calculated by interleaving the binary representations of its coordinate values. The Z-value is used as the name for each subspace. Z-order curve has two important properties: (i) If subspace A encloses subspace B, the



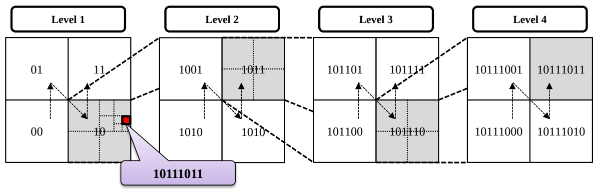
(a) Naive approach



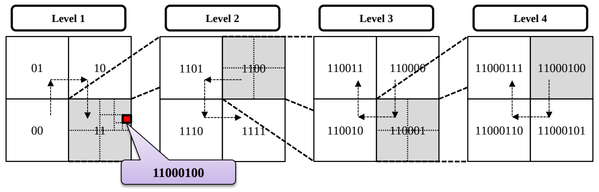
(b) OCGs at t_0 (c) OCGs at t_1 (d) OCGs at t_2

Figure 1: An example of deciding OCGs on Z-order sequence

name of subspace A is a prefix of that of subspace B. (ii) If subspace A does not have the same Z-order prefix with subspace B, the subspace A does not intersect with subspace B.



(a) Identifiers of cells on Z-order curve



(b) Identifiers of cells on Hilbert-curve

Figure 2: Bit patterns as identifiers of cells on the ordering sequences

We use bit patterns as cell identifiers. An identifier presents the sequence value of a cell on the ordering algorithms. Figure 2 illustrates how the identifier is generated on both Z-order curve and Hilbert curve through the ordering algorithms. It also shows that the same coordinates have different identifier in different algorithms. Both algorithms use 2 additional bits to represent the hierarchical levels of cells in the Grid. In Z-order curve, the front bit and rare bit of the additional bits present x and y coordinates of cells in the data space with 2×2 resolution respectively, as shown in Figure 2a. While Z-order curve uses always the same shape, in Hilbert curve, the shape is rotated to keep the locality of the sequence according the position of cells as shown in Figure 2b. The length of an identifier is determined by the size of the smallest cells in the Grid. We show step by step how a cell identifier is created on Z-order curve and Hilber curve in Figure 2a and Figure 2b respectively. For instance, in Figure 2a we show the cell with 2 bits identifier `10` in level 1 is first split into 4 cells in level 2, each with 4 bits identifier with `10` as Z-order prefix. The cell with identifier `1011` in level 2 continues to be split into 4 cells in level 3, each with `1011` as the Z-order prefix. In level 4, we show that when the cell with identifier `101110` in level 3 is split into 4 cells, their cell identifiers are created by using the same Z-order prefix `101110`. Similar prefixing scheme is used for creating cell identifiers on Hilbert-curve.

3.3. Compression of identifiers

It is known that one way to decrease the height of the index tree is to increase the fan-out of the index nodes. An advantage of using Z-order prefix on Z-order curve or Hilbert prefix on Hilbert curve to create cell identifiers is the opportunity for compression of identifiers. Given that each OCG is an entry in the OCG-index and the identifier of an OCG is composed by the set of cell identifiers of the cells contained in this OCG, thus, we can increase the fan-out of the index nodes by compressing the identifiers of the cells.

By looking at the cell identifiers closely, we can observe that a fair number of cells have many zeros in their tails when these cells do not need to be further split while some of their sibling cells continue to be split. Thus, an intuitive approach to compression of cell identifiers is simply to omit the number of `0` on the tail of the identifiers. For example, if an identifier is `011000000` and its size is 64bit (8byte), we present the identifier as 4 bit value `0110`. If we have the value `0110` and the maximum length of the identifier, the decompression is processed by putting the number of `0` on the tail of the values according to the maximum size of identifier. To maximize the fan-out of index nodes, it is important to find a cell that has as many as `0` on the tail of the identifier as a split key when splitting an OCG because of the overflows when more spatial objects enter the spatial region indexed by a given OCG. We call the identifier with the maximum number of zeros in its tail the shortest identifier. Formally, we define the shortest identifier as follows.

Definition 1 (Shortest Identifier).
Let $[id_s,\ id_e]$ denote the identifier range in a given Grid structure, where id_s and id_e are start and end identifiers of the range respectively, and $L_{sufix}(id)$ denote the length of `0` as the suffix of a given identifier id. The shortest identifier, denoted by $ID_{shortest}$, is an identifier with the maximum length of `0` as its suffix in $[id_s,\ id_e]$, formally defined by

Equation 2:

$$ID_{shortest}(id_s, id_e) = \{id_i | L_{sufix}(id_i) \geq L_{sufix}(id_j), id_s \leq \exists id_i \leq id_e, id_s \leq \forall id_j \leq id_e\} \quad (2)$$

For example, given the identifier range [`11010011`, `11001000`], the shortest identifier is `11010000`. Figure 3 shows the algorithm to find the shortest split identifier for a given identifier range.

```
Find_shortest_split_identifier($id_1$, $id_2$)

// Input = $id_1$ : identifier 1, $id_2$ : identifier 2
// Output = split identifier
split_id = 0 ;
FOR (i = max_length – 1; i >= 0; i -= 1) {
    $id_{temp}$ = 1 << i ;
    split_ id |= $id_2$ & $id_{temp}$;
    IF (($id_2$ & $id_{temp}$) != ($id_1$ & $id_{temp}$)) BREAK ;
}
RETURN split_id ;
```

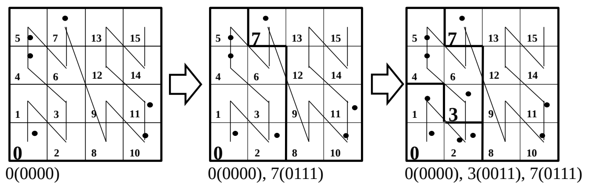
Figure 3: Algorithms to find the shortest split identifier

We use additional information about the length of the identifiers when putting the compressed identifiers into the index nodes. The length information is also used when the identifiers are read from the index nodes. The identifiers are compressed only when they are stored on disk. When processing in memory, we always use the full length of identifiers. To compress identifiers, the length information is predefined as well as the size of the identifiers. Suppose that the size of identifier is 64 bits and the length of an identifier is 4 bits. Each value of 1 in the length information reflects 4bit length of compressed identifier. This means that the identifier is compressed with 4 bits as a unit. A length value `0000` presents that the length of the identifier is 4 and the length value `0001` presents that the length of the identifier is 8. Both size and length of identifiers are adjustable parameters to be initialized during the system initialization. Given that there is a tradeoff between the length value and the storage cost, it is important to discuss the guidelines for determining the optimal length value of an identifier.

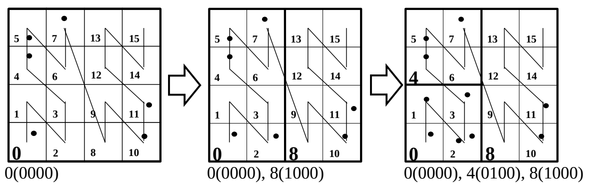
3.4. Maintaining the index structure

When an OCG is split due to an overflow over the specified cap of group size in terms of the number of spatial objects, we need to decide a split key in order to split this OCG into two OCGs with smaller spatial regions. To keep the optimal performance of the index structure, we need to consider the compression efficiency and the storage utilization and make a better tradeoff. To achieve better compression efficiency, we want this split key to be as short as possible since it will serve as the prefix. However, to obtain better storage utilization, an OCG should be split into two OCGs with equal or similar number of spatial objects. Thus the key is decided by the position of the median object on the ordering sequence. One way to incorporate good compression efficiency into account when choosing the optimal split key is to balance storage utilization with compression efficiency. For example, the split key is decided as short as possible within 1/3 median identifiers of the entries in an OCG. If id_s is the identifier of entry at n/3 lowest position on the ordering sequence and id_e is the identifier of entry at 2n/3 lowest position on the ordering sequence, then the split key is decided by $ID_{shortest}(id_s, id_e)$ in Equation 2.

In the minimum storage utilization, the first method (50%) is better than the second methods (33%). However, as shown in Section 3.3, the fan-out of index node in the second method is twice more than that in the first method. In the other words, although the storage utilization in the second method is less than in the first method, the number of entries in the second method is in general bigger than the first method. Figure 4 visually shows the difference between the cases. Figure 4a presents the case only considering storage utilization. In the case, according to the increase of objects, the OCGs are split and the split key is decided by the median objects in the ordering sequence. so two groups are made and the difference number of objects between the two groups are equle to or less then 1. Figure 4b presents the case considering both storage utilization and compression efficiency.



(a) Consider only storage utilization



(b) Consider storage utilization and compression efficiency

Figure 4: Index maintenance by tradeoff between storage utilization and compression efficiency

3.5. Query Processing with OCG-index

In this section we describe the algorithm for processing queries utilizing the proposed index. Due to space constraint, we provide the pseudo code for range query processing algorithm in Figure 5. kNN queries can be seen as applying a ranking method that sorts the range query results by the ascending order of their distances to the query focal object. In the range query processing algorithm, we first calculate the smallest identifier min_id and the largest identifier max_id overlapped with the given range in the manner similar to the quad tree search style (line 01∼02). The algorithm starts with finding a node including min_id and retrieving objects that match the query q in the node (line 04∼05). Since the index structure is B$^+$-tree, we just use next link of the node to find the next node. If the next node is overlapped with the range given by query q, we continue to search objects for q in the next node. This process continues until the next node is no longer related to q (line 06∼09). If the identifier of the next node is larger than max_id, the algorithm terminates. Otherwise, it is continued by finding the next smallest identifier which is an identifier of the cell overlapped with q and the identifier is bigger than the identifier of node searched previously. Equation 3 presents the definition of the next smallest identifier ID_{next_search}, given a query q and the smallest identifier at least id. c denotes a cell in the grid matrix.

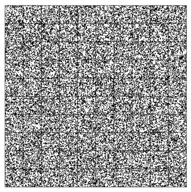
$$ID_{next_search}(q, id) = \{c_i.id | c_i.id \le c_j.id, \\ c_i \ge id, \exists c_i \in q.region, \forall c_j \in q.region\} \quad (3)$$

```
Range_query_processing( index, q)
// Input = index : an index structure, q : a query
// Output = result set
01 min_id = the smallest identifier of cells overlapped with q;
02 max_id = the largest identifier of cells overlapped with q;
03 WHILE ( true) {
04     n = find a node including min_id in index;
05     result_set <- retrieve objects overlapped with q in n;
06     WHILE ( n.next_node is overlapped with q) {
07         n = n.next_node;
08         result_set <- retrieve objects overlapped with q in n;
09     }
10     IF ( n.id > max_id ) BREAK;
11     min_id = find the next smallest identifier of cells
              overlapped with q;
12 }
13 RETURN result_set;
```

Figure 5: Algorithm for processing range queries

4. Performance Evaluation

In this section, we evaluate the performance to show the characteristics of OCG-index in various environments. As the ordering method, Z-order and Hilbert curve motheds are used. We also compare the performance according to the compression. The evaluation shows the performance difference in comparison with R*-tree, TPR*-tree and Grid structures. Z_OCG_UNCOM and Z_OCG_COM denote the OCG-index with Z-order, and OCG-index with Z-order with compression, respectively. H_OCG_UNCOM and H_OCG_COM denote the OCG-index with Hilbert curve, and OCG-index with Hilbert curve with compression, respectively. For Grid structures, we measured the performance with various resolution among 512×512($GRID512$), 256×256($GRID256$), 128×128($GRID128$), 64×64($GRID64$). We evaluated the storage cost, query processing cost and update cost according to the number of objects, size of queries and velocity of objects. To see the performance according to the density of objects, two different environments are used. Figure 6 shows the environment according to the distribution of objects. Figure 6a and Figure 6b shows random and skewed objects distribution, respectively. As default settings, the size of a disk block assigned to a node is set 2048 bytes and the number of objects inserted into index strucutres is set 1,000,000(1M). The data space is 1000×1000.


(a) Random distribution

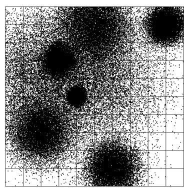
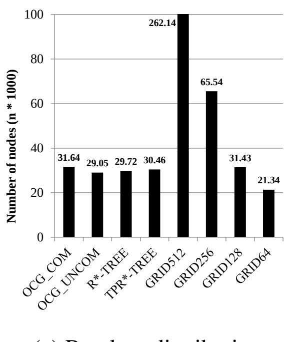

(b) Skewed distribution

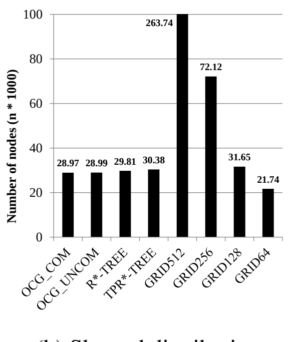
Figure 6: Object distribution

We first evaluated the storage cost OCG-indexes, R*-tree, TPR*-tree and Grid structures. To measure the storage cost, we count the total number of nodes for each index structures required. Figure 7 presents the comparison. Figure 7a shows the storage cost in random object distribution and Figure 7b shows the storage cost in skewed object distribution. It shows that tree-based indexes require simular storage cost while the storage cost of Grid is exponentialy increased according to the increase of resolution. We omitted the storage cost of Z_OCG_COM and Z_COG_UNCOM because they have almost the same number of node with regardless of ordering methods.

To show the benefits of OCG-indexes, we compared the number of index nodes for each index structures required. Figure 8 shows the total number of index nodes for each index structure requires in random object distribution according to the number of objects. H_OCG_UNCOM reduces about 65% of index nodes than R*-tree. H_OCG_COM reduces about 80% of index nodes than R*-tree and 30% of index nodes than H_OCG_UNCOM. We also measured the average number of entries in an index node to show the storage utilization in terms of index nodes. Figure 8b shows the average number of entities. The number of entries depends on the fan-out of node that the index structures allow.

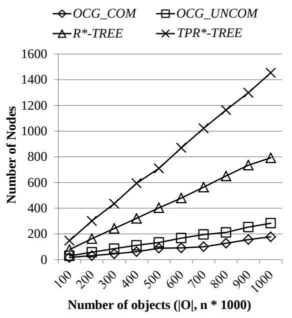


(a) Random distribution

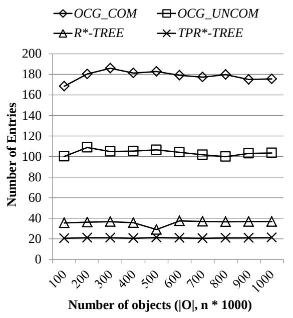


(b) Skewed distribution

Figure 7: Total storage cost

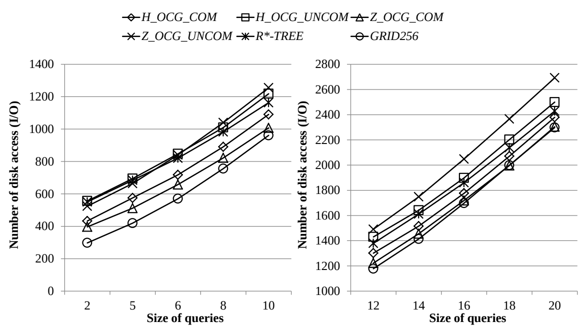


(a) Number of index nodes

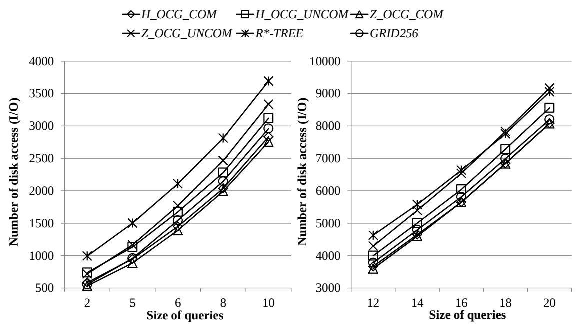


(b) Storage utilization

Figure 8: Storage cost in terms of index nodes



(a) Query processing in random distribution



(b) Query processing in skewed distribution

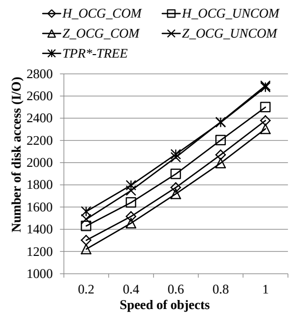
Figure 9: Query processing cost according the size of queries

The performance difference is from the size of each entry. Each entry in OCG-index includes only the identifier and address of a child node and the length value of the identifier while each entry in R*-tree includes 2 coordinators and the address of chlid node address, and each entry in TPR*-tree additionally includes the two vectors for each dimension. H_OCG_UNCOM and H_OCG_COM increase the fan-out of index nodes about 4 times and 8 times and than R*-tree, respectively. Since Grid structures does not have index nodes, we did not compare the number of index nodes with Grid structures. The numbers of Z_OCG_UNCOM and Z_OCG_COM are almost the same as H_OCG_UNCOM and H_OCG_COM. So that we omitted OCG-indexes using Z-order in these evaluations.

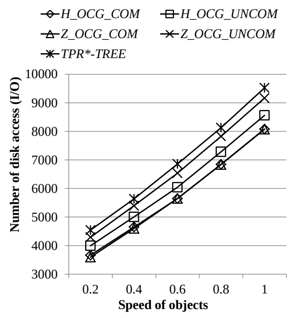
We compared the range query processing cost according to the size of queries as shown in Figure 9. We performed 100 queries in both random and skewed object environment. $GRID256$ is chosen to be compared because 256×256 was the optimal resolution in this evaluation environment, heuristically. As the cost of query processing, we measure the number of disk I/O while processing the queries. In the random distribution environment, R*-tree has the worst performance and $GRID256$ is the best performance result. all the OCG-indexes are in the middle as shown in Figure 9a. However, in the skewed distribution environment, $GRID256$ becomes worse as shown in Figure 9b. H_OCG_COM and Z_OCG_COM become more efficient than $GRID256$. In overall, OCG-indexes improve the performance of processing queries. OCG-indexes without compression save 5∼10% of cost and OCG-indexes with compression save 15∼30% of cost from that of R*-tree. As shown in Figure 9 grid structure is very good in random distribution environment, but it is not efficient in skewed distribution environment.

We compared the query processing cost with TPR*-tree on moving range queries according to the speed of object movements in both random and skewed distribution environments. We performs 100 range queries for 10sec future from the queries issue and set the size of queris 10×10. Figure 10 shows the perfomance comparison. In overall, OCG-indexes show the better performance than TPR*-tree. OCG-indexes without compression save about 5∼10% of cost and OCG-indexes with compression save about 15∼25% of cost from that of TPR*-tree.

For the comparison on update cost, we measured the number disk I/O while update the location of objects according to the speed of object movement in both random and skewed distribution environments. Figure 11 shows the evaluaton of the cost. Since R*-tree is desigined for static

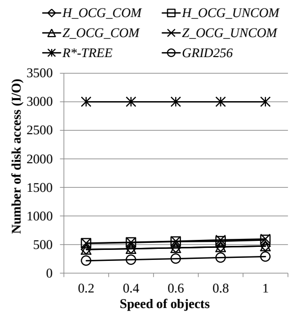


(a) Query processing in random distribution

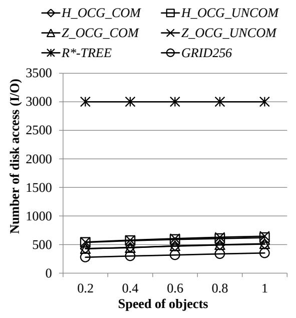


(b) Query processing in skewed distribution

Figure 10: Moving query processing cost according to the size of queries with TPR*-tree



(a) Object update in random distribution



(b) Object update in skewed distribution

Figure 11: Update cost according to the speed of objects

objects and sqatial queries, it suffers from the update of objects. Grid structuer shows the best performance on the update of objects' locations. It is because grid structure access the data node directly in hash manner. If the gird structure is optimized the most of update are done with accessing just one data node. The performance of optimized grid can be idle. In overall, OCG-indexes are in the middle of R*-tree and grid but they are very close to the idle performance, relatively. In this environment, OCG-indexes with compression save about 10% of update cost than OCG-indexes without compression.

5. Conclusion

Advances in ubiquitous connectivity and location sensing technology have fuelled a rich collection of location based services (LBSs). Efficient spatial indexing techniques are one of the most effective optimization methods to improve the quality of mobile services. In this paper, we argue that the index structure for LBSs should be efficient for spatial query processing in the presence of frequent location updates. We have introduced the concept of spatial order sequences and the concept of Ordered-Cell Group (OCG) and design a OCG based grid index structure (OCG-index) through a number of density-conscious optimizations. We speed up the search efficiency of OCGs by effective compaction of identifiers of OCG cells to maximize the fan-out of index node and decrease the depth of the index. In addition, we develop an efficient query processing algorithm that can effectively utilize OCG cells to speed up the processing of spatial queries. Our experimental results are compared with R*-tree, TPR*-tree, and grid structures. We show that OCG-index improves R-tree variant indexes in every aspect we measured and its performance is close to the optimized grid structures.

6. Acknowledgment

This work is partially sponsored by grants from NSF CISE NetSE program and SaTC program.

References

[1] A.Gutmann, "R-trees: A dynamic index structure for spatial searching", In Proc. ACM SIGMOD intl. conf. Management of Data, 1984, pp. 599- 609.

[2] N. Beckmann, H. P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an efficient and robust access method for points and rectangles", In Proc. ACM SIGMOD intl. conf. Management of Data, 1990, pp. 322-331.

[3] Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the positions of continuously moving objects. In: Proceedings of the ACM SIGMOD, pp. 331342 (2000)

[4] Tao, Y., Papadias, D., Sun, J.: The TPR-tree: an optimized spatio-temporal access method for predictive queries. In: Proceedings of the international conference on very large data bases pp. 790801 (2003)

[5] Jensen, C.S., Lin, D., Ooi, B.C.: Query and update efficient B$^+$-tree based indexing of moving objects. VLDB 768779 (2004)

[6] M. L. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-tree: Indexing Moving Objects by Space Filling Curves in the Dual Space. VLDB J. 2008.

[7] S. Chen, B. C. Ooi, K. L. Tan and M. A. Nascimento, "The ST^{2}B-tree: A Self-Tunable Spatio-Temporal B$^+$-tree Index for Moving Objects", In Proc. ACM SIGMOD intl. conf. Management of Data, 2008, pp. 29-42.

[8] Rudolf Bayer , Karl Unterauer, Prefix B-trees, ACM Transactions on Database Systems (TODS), v.2 n.1, p.11-26, March 1977

[9] Butz, A.R.: Alternative algorithm for Hilberts space-filling curve. IEEE Trans. Comput. C-20(4), 424426 (1971)

[10] G. M. Morton, A computer Oriented Geodetic Data Base; and a New Technique in File Sequencing, Technical Report, Ottawa, Canada: IBM Ltd, 1996.