DLMOI: A Moving Object Indexing Method Using Deep Learning

Xiaofeng Liu, Ji Li, Chuanwen Li, and Liangyu Chu

Abstract—**With advancements in positioning technology and the widespread adoption of wireless sensors, numerous wireless handheld and vehicular devices now come equipped with positioning capabilities. This has enabled a variety of new applications and generated large volumes of moving object data. The continuously changing location information of these moving objects requires efficient management in databases. Traditional database systems, which typically assume static attribute values until explicitly updated, face challenges in managing such dynamic, constantly changing location data efficiently. Current moving object indexing structures fall mainly into two categories: grid-based and tree-based indexing. However, each approach has inherent limitations. In this paper, we propose a novel indexing method that combines grid and quadtree structures and utilizes a deep learning model to intelligently determine when leaf nodes should be split or merged. Our method is not just a theoretical concept, but a practical solution that can be applied to a wide range of scenarios. Experimental results demonstrate that our method achieves higher throughput and reduces response times, particularly in skewed moving object distributions, offering significant improvements over existing indexing techniques.**

Index Terms—**Moving Object, Grid-based Indexing, Tree-based Indexing, Quadtree Structure, Deep Learning**

I. Introduction

With the ongoing advancement of mobile computing, location-based services, and GIS applications, interest in moving object databases has grown significantly among researchers [1]. The primary purpose of a moving object database is to provide real-time data updates and query services for moving objects, ensuring users receive query results with specific temporal and spatial accuracy within a defined range. For example, in taxi service applications, both users and taxis are considered moving objects. A typical query operation in this context involves recommending taxis within a specified distance to users [2].

Research on moving object databases mainly focuses on four areas: location modeling, query language, index structure, and uncertainty management [3][4][5][6]. Indexing technology for moving objects, which stores and retrieves their spatio-temporal locations, plays a crucial role in the setup and querying of these databases. Effective indexing for moving objects is essential for efficient data management and enhanced query performance, making it a key area of research in moving object databases.

Moving object indexing can be roughly divided into two types. One type is to partition index nodes based on the object distribution of data, represented by R-tree [7], its variants (R+- tree [8], R * - tree [9]), and index structures constructed by extended trees. The two variants of R-tree, TB tree [10] (Trajectory Bundle tree) and STR-tree, index the movement trajectory, which stores similar trajectory fragments in the same tree node, thereby reducing the number of query traversals. MV3R-tree [11] is a hybrid index structure that uses R-tree and its variants to index historical trajectories. STAR-tree can index moving objects' current and future positions, but it should be suitable for scenarios with infrequent updates. TPR-tree [12] is an index structure that indexes moving objects' current and future positions. Each node defines a spatial rectangle of future time points. However, the indexing performance will significantly reduce if the moving object does not move long. REXP tree [13] indexes moving objects' current and future positions. The index node identifies the expiration time of the motion vector. When the node fails, it is deleted according to a specific strategy to ensure the compactness and effectiveness of the index. TPR*-tree [14] is a variant of the TPR tree, which uses a different index node maintenance algorithm to make the structure more compact.

Another type of indexing organizes nodes based on spatial partitioning, primarily using grids and their variants. The ST2B-tree [15] combines grid and B-tree methods, managing moving objects spatially according to density distribution and grid granularity, adapting to rapidly changing loads. The uniform grid index, P-Grid [16], divides the spatial area into equal-sized grids, assigning moving objects to different grid units based on their coordinates; each unit grid is linked to a data list that stores information about the objects within it. M-Grid [17] improves upon P-Grid by incorporating Hilbert curves to better address multidimensional data query challenges. D-Grid [18] integrates speed information in its query process using a dual spatial grid index structure, accommodating the dynamic behavior of moving objects.

The above two methods can be broadly categorized as tree-based and grid-based index structures. However, grid-based index structures tend to underperform when objects are frequently moved or are unevenly distributed. While tree- based indexing can effectively handle object movement, its query performance often falls short of that provided by grid-based indexing.

In our previous work [19], we introduced GAPI, a GPU-

Manuscript received Jan 26, 2024; revised Nov 21, 2024.

Xiaofeng Liu is an experimenter of State Key Experimental Center of Computer Science,Shenyang, China,110000. (e-mail: centos@vip.qq.com).

Ji Li is a PhD candidate of Northeastern University, Shenyang, China,110000. (e-mail: 408567077@qq.com).

Chuanwen Li is a professor of Northeastern University, Shenyang, China, 110000. (corresponding author to provide. phone: 024-83687385; fax: 024-23890977; e-mail: lichuanwen@mail.neu.edu.cn).

Liangyu Chu is a postgraduate student of Northeastern University, Shenyang, China,110000. (e-mail: 1173087496@qq.com).

accelerated parallel indexing method for spatial moving objects. This method integrates grid and quadtree structures to track leaf node counts as objects enter and exit, allowing dynamic splitting and merging of leaf nodes as needed. Experimental results showed that GAPI significantly outperforms other indexing approaches; however, it still demands substantial computational resources and its performance is heavily dependent on GPU capacity. To overcome these limitations, this paper presents DLMOI (Deep Learning-based Moving Object Indexing), a novel indexing structure that combines grid and quadtree structures with deep learning to efficiently address the challenges of moving object indexing. The main contributions of this paper are as follows:

1) The DLMOI proposed in this paper combines grid and quadtree indexes. It retains the grid index's rapid query capabilities while using the quadtree index to address the issue of spatial and temporal unevenness in the load of moving objects.
2) This paper introduces a deep learning-based split and merge model to address the time-consuming nature of quadtree index operations involving splitting and merging. Objects within the grid are represented as images, and a combination of convolutional neural networks and attention mechanisms is employed to determine whether leaf nodes require splitting or merging.
3) Experimental results demonstrate that our method achieves higher throughput and reduces response times, especially in scenarios with skewed distributions of moving objects. Our process offers significant improvements over the best currently available indexing techniques.

II. Related Concepts and Technologies

A. Spatial Database

The spatial database is a repository of geospatial data and associated attributes stored on a computer in a defined structural format. It serves three fundamental functions: storing and managing spatial data, data querying, and analysis and reasoning based on spatial data. The first function entails the efficient organization and management of geospatial data, facilitating the storage of vast quantities of spatial information in a structured and readily accessible manner. This organization ensures that data can be quickly retrieved and updated as needed. The second function allows users to extract specific information from the spatial database based on criteria such as location, attributes, or spatial relationships. The third function empowers users to conduct analytical tasks like spatial pattern recognition, proximity analysis, and spatial modeling. These analyses are crucial for understanding spatial relationships, making informed decisions, and addressing complex spatial challenges.

Spatial data encompasses information used to describe spatial objects' shape, size, position, and other characteristics. It originates from various sources and comes in multiple forms. These data types can be categorized into the following categories:

1) Attribute data. Attribute data describes an object's various non-spatial attributes, such as text, dates, numbers, and other relevant information.
2) Graphical image data. A large amount of data in the spatial database system needs to describe the spatial attributes with the help of graphics and images.
3) Spatial relationship data. Spatial relationship data describes the topological relationships between geographic elements or objects. It elucidates how different spatial entities are connected or related to each other in the real world.

Spatial data structures are used to describe the structural relationship among spatial data. Based on the data storage methods employed in spatial data models, these structures are primarily categorized into vector and raster data. Vector Data Structures capture geographical entities by recording their coordinates, enabling a precise definition of location, length, and area. This approach is indispensable when the accuracy of spatial details is paramount, facilitating detailed geographic analyses and accurate mapping. Ra Raster Data Structures divide the geographical surface into uniformly sized, precisely adjacent grid cells. It offers a more intuitive representation of spatial data, simplifying implementation and supporting straightforward modifications and expansions. This method is beneficial for depicting continuous spatial phenomena. These spatial data structures allow geographic information to be effectively organized and analyzed, enhancing decision-making processes in diverse fields such as urban planning, environmental management, and resource allocation.

Spatial databases support multiple spatial data types and spatial data models. Compared with traditional databases, spatial databases have the following characteristics:

1) Vast Data Volumes: Spatial databases manage extensive data, encapsulating size, shape, position, and other details of real-world phenomena in specific data structures. The immense volume of spatial data requires specialized management techniques.
2) Complexity and Diversity of Data: Spatial databases store not only spatial data but also accommodate a broad range of data types, including text, numerical values, dates, and symbols. This diversity allows for a comprehensive description of various scientific phenomena, enriched with graphical images and spatial relationship data for a complete representation.
3) Highly accessible. In practical applications, spatial databases are not only repositories for spatial data but are also platforms for efficient retrieval and timely analysis of this stored information, ensuring that users can quickly access and utilize the data as needed.

B. Grid Index

The grid index [20] is a fundamental index structure characterized by its relatively simple construction principle. It divides the spatial range where the target spatial entity set is located into a series of cells of the same size and divides the spatial position into grids. Each cell in the grid index acts as a bucket, keeping track of the number of spatial entities within that cell. As new entities are processed, they are added to the corresponding grid cell, reflecting their current spatial position.

Grid indexes encompass vital operations such as creation, reconstruction, insertion, deletion, and update. During the initial creation of a grid index, an appropriate grid scale is determined based on the statistical characteristics of the data. Each spatial entity is decomposed into its respective grids, and its record is added to all relevant grids. This process continues until all entity records are accounted for. The grids it spans are identified during the insertion of a spatial entity, and the grid code of the spatial features is calculated. The entity's data are then recorded in the grid structure index table. Deleting an entity record involves removing all corresponding index records associated with that entity. When updating the index, the process consists of deleting the existing index record and adding the updated index record to reflect the changes made.

Grid indexing has a meager operation cost, and the most essential query is also relatively simple. The process of using grid index queries can be divided into two steps. First, all the entity grids covered by the query area and included are retrieved to achieve rough queries. Then, based on the rough retrieval result set, records that do not meet the query requirements are eliminated through precise comparison.

The grid index structure has advantages such as ease of maintenance and high scalability. However, its performance is significantly influenced by the grid size. Larger grid partitions lead to more, on average, falling into each `cell and a higher percentage of entities being entirely distributed within a grid. As a result, when performing nearest neighbor queries, it becomes challenging to preliminarily determine the search range and locate the query object, leading to decreased index accuracy. On the other hand, smaller grid partitions result in greater data storage capacity and potential redundancy, with a lower average number of entities falling into each cell. However, this requires multiple range extensions during queries to determine the initial search range, which can lead to reduced query efficiency. In scenarios with uneven spatial data distribution, such as road network environments, the grid index structure may struggle to effectively partition an appropriate grid size. Consequently, it becomes challenging to strike a balance between accuracy and efficiency.

C. Quadtree Index

Quadtree [21] was proposed by Tayeb in 1998. A quadtree is a hierarchical structure that recursively divides a rectangular area. Specifically, it divides a rectangular space into four equal rectangles as its subspaces. In this recursive division, four h-1 power rectangular areas (h is the depth of division) stop until the number of objects in each rectangular area is less than or equal to the given bucket size. As a result of this recursive division, the quadtree has a single root node, and each intermediate node has four child nodes, with each node corresponding to a rectangle. This hierarchical structure allows for efficient spatial querying and indexing of spatial data.

Quadtree finds extensive applications in various fields, including graphic image processing, 2D fast collision detection, and handling sparse data. It serves as a representation tool for spatial objects like point data, curves, surfaces, and volumes and as a spatial indexing technique. One of the commonly used methods in spatial indexing technology, Quadtree is a versatile and flexible data structure that can handle variables of arbitrary dimensions, not limited to two-dimensional data.

Quadtree has proven to be a versatile and widely used spatial indexing technique, particularly suitable for datasets with non-uniform distributions or varying densities. It efficiently organizes spatial data, making it easier to perform operations like range queries and nearest-neighbor searches while reducing the number of comparisons and improving overall query performance. However, some problems exist with using the quadtree structure for indexing. With the deepening of its hierarchy, the query efficiency will drop sharply.

III. DLMOI Index Structure

A. Problem Definition

Given a spatial plane S, the moving objects set $O = \{o_1, o_2, \cdots, o_n\}$.Each of these objects is represented as $o_i = \{o_i^{id}, o_i^x, o_i^y, o_i^t\}$, $o_i^{id}$is the unique identifier of o_i, (o_i^x, o_i^y) is its position, o_i^t is the last update time. Query operations set $Q = \{q_1, q_2, \cdots, q_e\}$.Each query request is expressed as $q_j = \{x_{min}, y_{min}, x_{max}, y_{max}, t_q\}$, $(x_{min}, y_{min}, x_{max}, y_{max})$ defines a rectangular query box, t_q is the time when the query started. The purpose of the moving objects database is to return the moving object located in the query box to the user when the query q_j in Q arrives.

This query method is called a range query, and other types of queries, such as k-nearest neighbor queries, can be transformed into a series of range queries. Therefore, this paper only discusses the support of index structure for this query type.

B. Index Structure

The main work of the moving objects database is to update moving objects' locations and return results according to query requirements. Therefore, the moving object index structure must efficiently meet two primary conditions:

1) Find the object through the object identifier o_i^{id}.
2) Find and update the moving object according to the object position (o_i^x, o_i^y).

The auxiliary index method based on the hash table can support the condition 1).

Definition 1 (auxiliary index) The auxiliary index uses the hash table $\mathcal{H}$ to index all spatial objects according to the o_i^{id} value. $\mathcal{H}$ stores key-value pairs of the form (o_i^{id}, p_bkt, idx), where p_bkt is the memory space location of the bucket , o_i is located in the hybrid index (see definition 2), and idx represents its relative location in the bucket.

Figure 1 shows an example of auxiliary index. According to the nature of the hash table, the storage location of o_i in the memory can be found according to o_i^{id} in constant time by using the auxiliary index.

Grid-based and tree-based indexes have advantages and disadvantages in satisfying condition 2).

The grid-based index method can directly calculate the grid to which o_i belongs according to (o_i^x, o_i^y). However, when the spatial distribution of moving objects is uneven, the number of moving objects in the grid in the hotspot

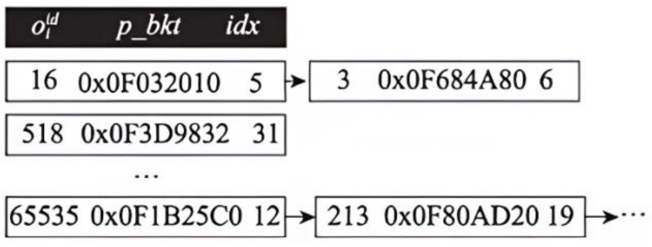

Fig. 1. Example of auxiliary index

area is too large, and updating the object position information will cause hotspots. The grid is frequently locked, reducing the parallel performance of the grid in hotspots.

The tree-based index method reduces object congestion in hotspot areas. Still, each query for o_i by (o_i^x, o_i^y) requires multiple queries from the tree root node to a series of intermediate nodes to leaf nodes, which reduces query efficiency. What has a more significant impact on the overall efficiency is that the tree index needs to constantly adjust the structure to adapt to the distribution of moving objects, and the calculation of whether the leaf nodes need to be adjusted and the adjustment operation itself will consume a lot of computing resources.

To sum up, this paper proposes a hybrid index method combining grids and quadtrees, which avoids the shortcomings of the above two methods.

Definition 2 (hybrid index) The hybrid index $\mathcal{P}$ divides the space plane S into $G_{num} = 2^\rho \times 2^\rho$ grids. Each grid can be converted between grid nodes and quadtrees according to conditions.

The goal of hybrid index is to balance the load of each cell by transforming the grid in the hotspot area into a quadtree. ρ is the grid partitioning parameter, using the selection criteria given in [22]:

$$\rho = \frac{1}{2}\lceil lbN - lbC_L \rceil \tag{1}$$

Where N represents the total number of moving objects in the space, C_L represents the capacity of leaf nodes.

During execution, the number of moving objects in and out of each grid is dynamically judged. The grids that meet the splitting requirements are converted into quadtrees, and the quadtrees that meet the merging requirements are converted back to grids. The leaf nodes are also split and merged in each quadtree according to the conditions.

Figure 2 shows an example of a hybrid index. The upper layer in the right half of figure 2 is a plane S divided into $G_{num} = 2^\rho \times 2^\rho$ grids, where the black grid represents the hotspot grid converted into a quadtree. The lower layers represent the spatial areas corresponding to the quadtree nodes at all levels, and the black nodes represent further subdivided areas. The left half of the figure shows one of the enlarged quadtrees, and the division of the quadtree to its grid is displayed above it. It can be seen that after converting to a quadtree, the region has been more finely divided.

Each grid cell c_i only contains a p_{bucket} pointer, which points to a bucket list $\mathcal{L}_i$, which contains a series of buckets that store a fixed number of all moving objects belonging to the cell. Each quadtree node ξ_i is represented as $\xi_i = \{p_{bucket}, p_{c_1}, p_{c_2}, p_{c_3}, p_{c_4}\}$, where p_{c_1}, p_{c_2}, p_{c_3} and p_{c_4} are respectively pointers to child nodes in the four quadrants with the current node as the parent node.

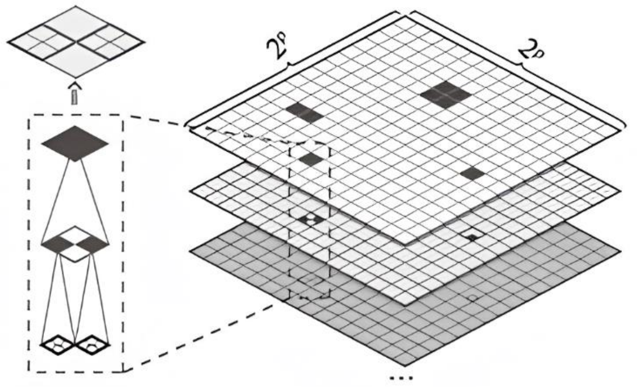

Fig. 2. Example of hybrid index

IV. Node Split and Merge Model

The split and merge operations for grid and quadtree leaf nodes are fundamentally identical. Therefore, grid and leaf nodes will be collectively referred to as nodes in this section.

A. *Moving Object Mapping*

Traditional splitting and merging algorithms typically base decisions on the number of moving objects and the time required for updates, which often involves extensive computations and precise criteria for splitting and merging. This approach can be both time-consuming and prone to inaccurate splits. To address these challenges, this paper proposes a deep learning model to swiftly and accurately determine when nodes should be split or merged.

A primary challenge when using deep learning models for the split and merge task is the variability in the number of objects per node. Within the same quadtree, some nodes may contain only a few objects, while others may contain many, causing substantial fluctuations in the model input size.

To overcome this issue, this paper presents an innovative solution: mapping each node into a square image, with the moving objects within the node represented as points within this image. The mapping formula for the coordinates of the moving objects is as follows:

$$\bar{o}_i^x = \frac{(o_i^x - node.left)}{(node.rigt - node.left)} * l$$
$$\bar{o}_i^x = \frac{(o_i^x - node.left)}{(node.rigt - node.left)} * l \tag{2}$$

Where $\bar{o}_i^x$ and $\bar{o}_i^y$ is the coordinate of the moving object i mapped to the picture. o_i^x, o_i^y is the original coordinate of the moving object i. $node.left$ represents the left boundary of the node. $node.rigt$ represents the right boundary of the node. $node.floor$ represents the lower boundary of the node. represents the upper boundary of the represents the edge length of the picture.

B. *Model structures*

Figure 3 shows the structure of the node split and merge model, shorted as SMM. The model consists of three parts: a Convolutional neural network, an attention mechanism, and a fully connected layer. These three parts will be introduced in detail next.

In a quadtree index, the merging and splitting of nodes is related to the distribution of the objects they contain. After mapping the objects contained in the nodes to images, SMM utilizes a Convolutional Neural Network to extract features from these images. These features can be considered representations of the distribution of the objects.

We believe that a node's merging and splitting often do not depend on itself but are also influenced by the object distribution of neighbor nodes and the entire grid space.

As illustrated in Figure 3, the node merge and split model considers not only the target distribution of the current node but also the target distribution of its three neighbor grids and the upper grid of that node. Input the current grid image, three neighbor grid images, and the upper grid image into five independent convolutional neural networks, respectively. The convolutional neural network of the current grid image and the three neighbor grid images share the same structure but have different parameters. Due to the higher number of objects in the upper grid, the upper grid image needs to be set at a larger size, requiring a more complex convolutional neural network to extract features.

The features extracted from the three neighboring and upper grid images are then fed into an attention mechanism layer. This layer automatically learns and assigns varying weights to the input features, thus allowing the model to focus on more pertinent or significant information. This approach helps ascertain each feature's importance in merging or splitting the current node. We derive the final relevant node features by assigning weights based on their significance and summing them accordingly.

Concatenate the node features with the relevant node features computed using the attention mechanism. Then, input the concatenated vector into a fully connected layer. This fully connected layer aims to analyze the features and determine whether the node should be merged or split. This transformation turns the node merging or splitting problem into a classification task with three categories: static, merge, and split.

The node split and merge model uses cross-entropy as the loss function to train the model. The calculation formula is as follows:

$$L(p,q) = -\sum_{i=1}^{M} p_i \log(q_i) \tag{3}$$

where M is the number of nodes, p_i is the real probability distribution of nodes split and merging, and q_i is the probability of node split and merge predicted by the model.

V. Data Structures and Algorithms

A. Data structures

The grid index in the hybrid index is implemented using a two-dimensional array:

$$array < array < unique\_ptr < Node >, width >, height > \tag{4}$$

Where $width$ and $height$ represent the number of grid columns and rows respectively. The Node class is the parent class of Cell and QuadTree. The cell class represents a grid that contains only one $unique\_ptr < Bucket >$ type pointer. The QuadTree class represents a quadtree:

```
Struct QuadTree{
/* Bucket chain header node */
unique_ptr < Bucket > p_bucket;
/* Child node */
array < unique_ptr < QuadTree > ,4 > children;
/* Parent node */
QuadTree * parent;
/* Coverage range */
int left, right, floor, ceiling;
}                                          (5)
```

The objects in Cell or QuadTree are stored in a bucket linked list structure. Buckets are used to store moving objects on leaf nodes, with a fixed size. The number of buckets under each leaf node is determined by the number of moving objects. If the bucket is full when inserting an object, create a new bucket. If the bucket becomes empty when deleting an object, delete the bucket. The bucket structure is as follows:

```
Struct Bucket{
/* Array for storing objects */
Site[max_size] sites;
/* The number of objects already in the bucket */
int current;
/* Next bucket */
unique_ptr < Bucket > bucket;
}                                          (6)
```

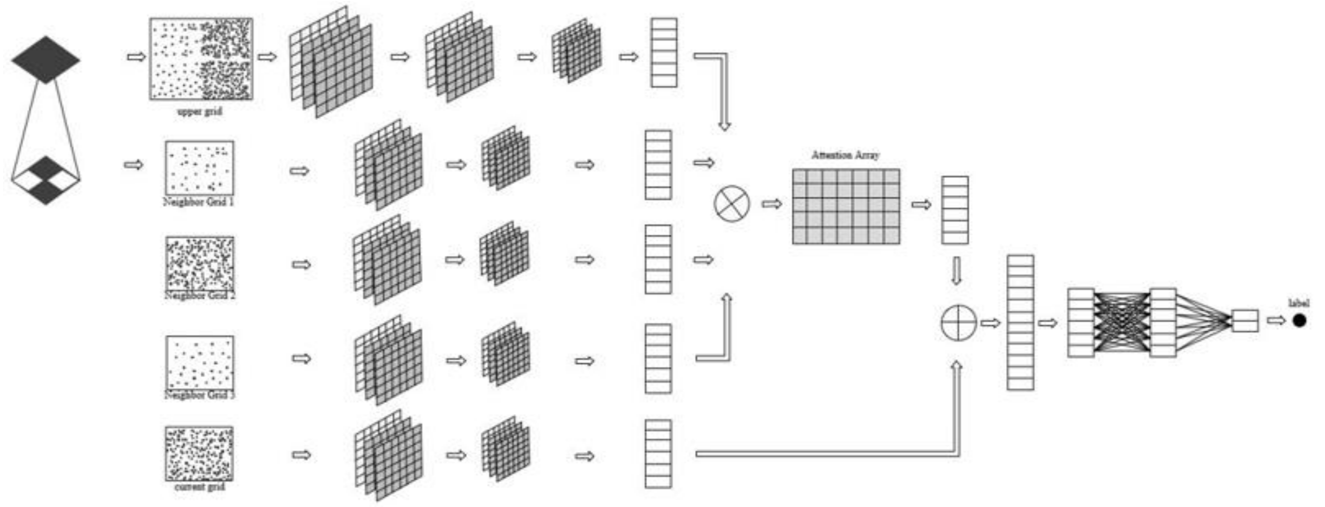

Fig. 3. Structure of node split and merge model

Where, the Site class stores moving objects data, including id, x, y, and update time t_u. It can be inferred from the content contained in the Site class that the data of one of its objects needs to occupy at least 128 bits (4 int values on a 32-bit machine) of memory space. If no protection measures are taken, access conflicts may occur between different threads when reading and writing the Site class in parallel. The traditional method to avoid access conflicts is to lock the Site object when reading or writing it. Since the Site class is the most frequently used class in the DLMOI index, to avoid the impact of locking on performance, the four data in the Site class were merged into one_ m128i type object, using the Intel MMX instruction set _ mm_ Load_ Si128 and_ mm_ Set_ Epi32 operation reads and writes the content, and uses_ mm_ Extract_ Epi32 extracts the corresponding data. This way, DLMOI indexes can correctly read and write Site data in parallel without locking.

B. Algorithms

With the continuous updating of objects, the structure of Quadtree also changes. The insertion algorithm of objects inserts them into appropriate nodes based on coordinates. The deletion algorithm finds the bucket based on the object ID and deletes it. The splitting and merging of cells is the key operation of balancing the Quadtree, only the cells that meet the conditions of splitting and merging can be splited and merged.

1) Spatial Object Insertion Algorithm

The algorithm for inserting object o_i into a leaf node is shown in Algorithm 1:

TABLE I
ADD TO LEAF ALGORITHM

Algorithm 1. ($\boldsymbol{add\_to\_leaf}$) Insert object o_i into leaf node

```
Input: moving object $o_i = \{o_i^{id}, o_i^x, o_i^y, o_i^t\}$
Output: No output. Leaf nodes are updated after the operation completes
1.  $cur\_leaf = get\_leaf(o_i^x, o_i^y)$
    /* Find the inserted leaf node by position $(o_i^x, o_i^y)$ */
2.      $if(is\_full(cur\_leaf.p\_bucket))$
3.          $n\_bucket = new\ Bucket()$;
4.          $insert\_bucket(n\_bucket)$;
5.  $insert\_object(o_i)$
        /* Insert the object $o_i$ into the current node's bucket */
```

During the process of object movement, as the position changes, the object will continuously move between various nodes. The main purpose of Algorithm 1 is to insert an object into a leaf node. The idea is to find the leaf node that the object should be inserted into based on its position (o_i^x, o_i^y), and judge the bucket state of the node. If the bucket is not full, insert object o_i directly into the bucket. If the bucket is full, create a new bucket n_bucket and insert the newly created bucket into the bucket list of the current node. Then insert the object $o_i$into the bucket, and add 1 to the number of objects stored in the bucket.

2) Space Object Deletion Algorithm

The algorithm for deleting object o_i from a leaf node is shown in Algorithm 2:

TABLE II
REMOVE FROM LEAF ALGORITHM

Algorithm 2. ($\boldsymbol{remove\_from\_leaf}$) Remove object o_i from leaf node

```
Input: moving object $o_i = \{o_i^{id}, o_i^x, o_i^y, o_i^t\}$
Output: No output. Leaf nodes are updated after the operation completes
1.  $bucket = get\_bucket(o_i^{id})$
2.  $delete\_from\_bucket(o_i)$
3.  $if(is\_empty(bucket))$
4.      $delete\_bucket(bucket)$;
    /* Delete empty buckets */
```

When moving object positions are updated, objects that no longer belong to the scope of the current leaf node need to be inserted into their new owning leaf nodes and deleted from the current node.

As shown in Algorithm 2, the bucket where the object is located is found according to the unique object identifier o_i^{id}, the object is deleted from the bucket, and the number of objects stored in the bucket is reduced by 1. After object o_i is deleted, if the bucket where object o_i used to be is empty, the bucket will also be deleted.

3) Cell Split Algorithm

The algorithm of Cell Split Algorithm is shown in Algorithm 3.

TABLE III
SPLIT ALGORITHM

Algorithm 3. ($\boldsymbol{split}$) split the cell

```
Input: quadtree leaf node $s\_node = \{p\_bucket, children, parent, left, right, floor, ceiling\}$
Output: No output. Quadtree structure is updated after the operation
        completes
1.  $x\_middle = (left, right)/2$;
2.  $y\_middle = (floor + ceiling)/2$;
3.  $s\_node.children[0] =$
      $new\ Quadtree\ (x\_middle, right, y\_middle, ceiling)$;
/* Initialize $s\_node.children$ [1 - 3] in a similar way*/
4.  $for\ each(child\ in\ s\_node.children)$
5.      $child.parent = s\_node$;
6.      $s\_node.p\_bucket.o_i \rightarrow child.p\_bucket$;
7.  $if(s\_node.parent\ in\ split\_candidates)$
8.      $delete\_from\_split\_candidates(s\_node.parent)$;
9.  $Insert\_split\_candidates(s\_node)$;
```

According to the split and merge algorithm introduced in Section 3, when leaf nodes are considered to be split, the quadtree index structure should divide leaf nodes s_node into four sub grids.

The split of sub grids is based on the principle of evenly dividing top, bottom, left, and right, where $x\_middle = (left, right)/2$, $y\_middle = (floor + ceiling)/2$, as shown in lines 1-2. After the split is completed, the parent nodes of all children are set as the current leaf node, and all objects in the split node s_node are moved to the corresponding child's bucket. If the parent of the split node belongs to the $split\_candidates$ list before the node is not split, split it from the $split\_candidates$ list and then add the current node to the $split\_candidates$ list.

4) Cell Merge Algorithm

According to the split and merge algorithm introduced in Section 3, when leaf nodes are considered to be merged, the quadtree index structure merges four grids into one. It should be noted that since the merge operation involves other nodes, it will only merge when more than half of the surrounding four nodes are judged to be merged.

The algorithm of Cell Merge Algorithm is shown in Algorithm 4:

TABLE IV
MERGE ALGORITHM

```
Algorithm 4. ($merge$) merge the cell
Input: quadtree leaf node $m\_node = \{p\_bucket, children, parent, left, right, floor, ceiling\}$
Output: No output. Quadtree structure is updated after the operation
        completes
1.  $if(for\ each\ m\_node.children\ is\_leaf())$
    /* Find the inserted leaf node by position $(o_i^x, o_i^y)$ */
2.      $m\_node.p\_bucket \leftarrow child.p\_bucket$;
3.      $delete\_null\_bucket(m\_node.p\_bucket)$;
4.      $if(m\_node\ in\ merge\_candidates)$
5.          $delete\_from\_merge\_candidates(m\_node)$;
6.      $if(for\ each\ m\_node.parent.child\ is\_leaf())$
7.          $insert\_merge\_candidates\ (m\_node.parent)$;
```

As shown in Algorithm 4, when a node meets the merge condition, all the buckets of its children are linked and assigned to the bucket of the current node. Adjust the bucket chain of the current node and delete empty buckets. If the current node belongs to a $merge\_candidates$ list, then delete the current node m_node from $merge\_candidates$ list. After merging, if the parent node of the current node belongs to the merge_ Candidates list, then the current node's parent node $m\_node.parent$ joining $merge\_candidates$ list.

VI. EXPERIMENT AND RESULT

This section compares the DLMOI index structure with the state-of-the-art M-Grid [17] and index structure and our previously proposed GAPI [19] index structure.

A. Datasets

The experimental simulation environment is Win10 system. The experimental code is implemented by C++ and python. The spatial object data is generated by MOTO (http://moto.sourceforge.net), an open-source moving object generation tool based on the Brinkhoff [22] algorithm. The experimental parameters are shown in Table Ⅴ.

TABLE V
EXPERIMENT PARAMETERS

parameter	Experimental value	Defaults
Total area/km2	200 000×200 000	—
Query area/km2	0.25, 1, 4, 16, 32	4
number of CPU threads	20	20
update/query ratio	250, 500, 1 000, 2 000	1
number of spatial objects/10^6	4 000, 8 000, 16 000	10
update interval/s	10, 20, 40, 80, 160	10

B. Experimental Analysis

Figure 4 illustrates the experimental results concerning the impact of the update/query ratio on throughput. As this ratio increases, all three indexing structures—DLMOI, GAPI, and M-Grid—demonstrate a rising trend in throughput. In most scenarios, the throughput of DLMOI and GAPI exceeds that of M-Grid. However, when the update/query ratio is low, the throughput for DLMOI and GAPI falls below that of M-Grid. This suggests that DLMOI and GAPI are primarily optimized for update operations.

In scenarios where query operations significantly exceed update operations, DLMOI and GAPI show lower throughput than M-Grid. This is attributed to the Quadtree index employed by DLMOI and GAPI, which requires traversing from the root node to the leaf nodes, thus increasing query time. Additionally, DLMOI consistently outperforms GAPI in throughput across all update/query ratios.

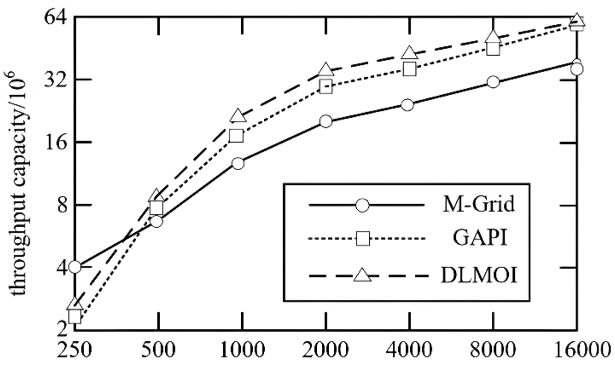

Fig. 4. Effect of throughput on update/query ratio

Figure 5 presents the experimental results on the impact of query area on throughput. As shown, DLMOI consistently outperforms both M-Grid and GAPI across all scenarios. As the query area expands, there is a noticeable decline in overall throughput. This trend can be attributed to the growing computational demands of larger query areas, which increasingly affect the system's transaction processing efficiency.

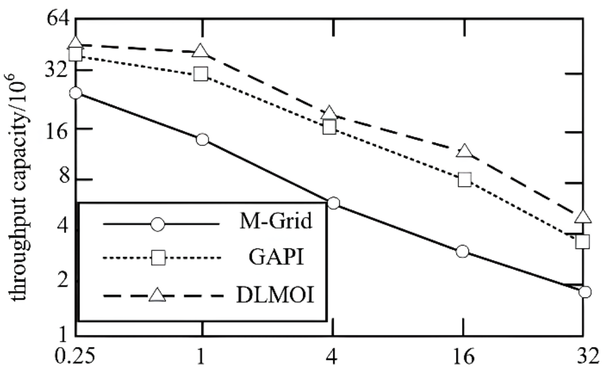

Fig. 5. Effect of throughput on size of queried area

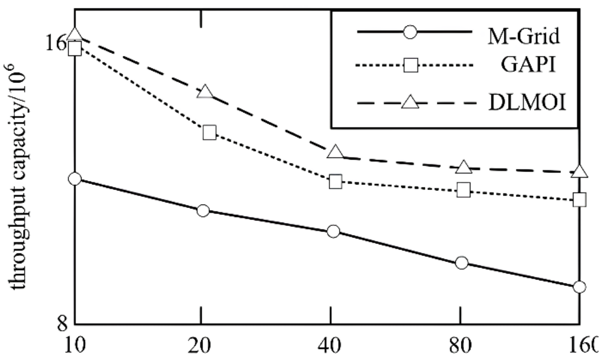

Fig. 6. Effect of throughput on the interval between updating

Figure 6 displays the experimental results on the impact of update interval on throughput. As the update interval increases, objects are more likely to move out of their current nodes, resulting in a significant drop in throughput for M-Grid. In contrast, DLMOI and GAPI maintain stable throughput levels due to their composite indexing strategies that integrate grid- and Quadtree-based structures. These systems also sustain high throughput during node splits and merges, highlighting the effectiveness of the deep learning-based split and merge model. Additionally, Figure 6 shows that DLMOI consistently outperforms both M-Grid and GAPI across all scenarios.

Figure 7 presents the experimental results on how the number of moving objects affects throughput. Throughput gradually declines across all three indexing methods as the number of moving objects increases. Despite this trend, DLMOI maintains superior throughput compared to GAPI and M-Grid, underscoring the effectiveness of its deep learning-based models for splitting and merging, which efficiently manage larger volumes of moving objects.

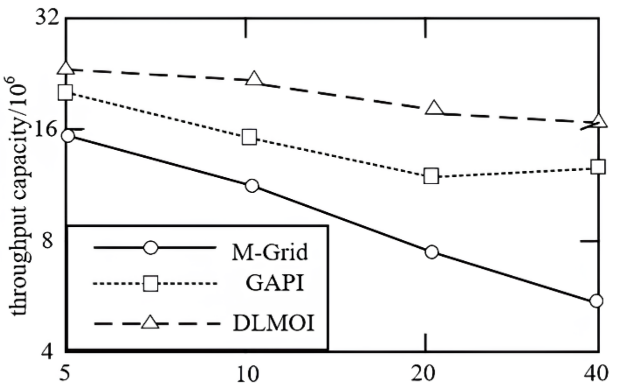

Fig. 7. Effect of throughput on the number of moving objects

VII. Conclusion

Building on the grid index, this paper introduces a hybrid indexing method incorporating quadtree technology, effectively mitigating the shortcomings associated with both tree-based and grid-based indexing systems. The paper introduces a deep learning-based split and merge model to overcome the time-intensive challenges associated with quadtree index splitting and merging. Experimental results show that this novel approach achieves higher throughput and reduced response times, outperforming traditional indexing methods.

References

[1] Huang Y K. Indexing and querying moving objects with uncertain speed and direction in spatiotemporal databases[J]. Journal of Geographical Systems, 2014, 16(2): 139-160.

[2] Nguyen T, He Zhen, Zhang Rui, et al. Boosting moving object indexing through velocity partitioning[J]. Proceedings of the VLDB Endowment, 2012, 5(9): 860-871.

[3] Song M B, Park K J, Ryu J H, et al. Modeling and tracking complexly moving objects in location-based services[J]. J. Inf. Sci. Eng., 2004, 20(3): 517-534

[4] Wang X, Xu J, Wang Y. NLMO: towards a natural language tool for querying moving objects[C].2020 21st IEEE International Conference on Mobile Data Management (MDM). IEEE, 2020: 228-229.

[5] Deng Z, Wang L, Han W, et al. G-ML-Octree: An update-efficient index structure for simulating 3D moving objects across GPUs[J]. IEEE Transactions on Parallel and Distributed Systems, 2017, 29(5): 1075-1088.

[6] Leal E, Gruenwald L, Zhang J. Handling uncertainty in trajectories of moving objects in unconstrained outdoor spaces[C]. 2016 IEEE International Conference on Big Data (Big Data). IEEE, 2016: 492-501.

[7] Guttman A. R-trees: a dynamic index structure for spatial searching[M]. ACM, 1984.

[8] Sellis T, Roussopoulos N, Faloutsos C. The R+-tree: A dynamic index for multi-dimensional objects[J]. 1987.

[9] Beckmann N, Kriegel H P, Schneider R, et al. The R*-tree: an efficient and robust access method for points and rectangles[M]. ACM,

[10] Pfoser D, Jensen C S, Theodoridis Y. Novel Approaches: to the Indexing of Moving Object Trajectories[C]. VLDB'00, Cairo, Egypt, 2000.

[11] Y. Tao, D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Predictive Queries[C]. Pro. of theIntl. Conf. On Very Large Data Bases, VLDB, Sept. 2001: 431-440

[12] Y. Tao, D. Papsdias, J. Sun. The TPR-Tree: An Optimized Spatio-Temporal Access Method for PredictiveQueries[C]. Proc. of the Intl. Conf. On Very Large Data Bases (VLDB) Sept, 2003.

[13] Saltenis S, Jensen C S. Indexing of Moving Objects for Location-Based Services[C]. ICDE'02, SanJose, USA, 2002.

[14] Y. Tao, D. Papsdias, J. Sun. The TPR-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries[C]. Proceedings of the 29th International Conference on Very Large Data Bases. Berlin, Germany: VLDB Endowment, 2003(29) : 790-801.

[15] Chen S, Ooi B C, Tan K L, et al. ST2B-tree: a self-tunable spatio-temporal B+-tree index for moving objects[C]. Proceedings of the 2008 ACM SIGMOD international conference on Management of data. 2008: 29-42

[16] Šidlauskas D, Šaltenis S, Jensen C S. Parallel main-memory indexing for moving-object query and update workloads[C]. Proceedings of the 2012 ACM SIGMOD international conference on management of data. 2012: 37-48.

[17] Kumar S, Madria S, Linderman M. M-Grid: a distributed framework for multidimensional indexing and querying of location based data[J]. Distributed and Parallel Databases, 2017, 35: 55-81.

[18] Xu X, Xiong L, Sunderam V. D-grid: an in-memory dual space grid index for moving object databases[C]. 2016 17th IEEE international conference on mobile data management (MDM). IEEE, 2016, 1: 252-261.

[19] Chen K, Li C, Lu G, et al. An adaptive parallel method for indexing transportation moving objects[J]. Complexity, 2021, 2021: 1-11.

[20] Yu X, Pu K Q, Koudas N. Monitoring k-Nearest Neighbor Queries over Moving Objects[C]. 2013 IEEE 29th International Conference on Data Engineering (ICDE). IEEE Computer Society, 2013:631-642.

[21] Guttman A. R-trees: a dynamic index structure for spatial searching[M]. ACM, 1984.

[22] Chen Su, Ooi B C, Tan K L, et al. ST2B-tree: a self-tunable spatio-temporal B+- tree index for moving objects[C [C]. Proceedings of the 2008 ACM SIGMOD International Confer-ence on Management of Data, Vancouver, Canada, Jun 10-12, 2008. New York: ACM, 2008: 29-42.

[23] Sarma A D, Gollapudi S, Najork M, et al. A sketch- based distance oracle for Web- scale graphs[C]. Proceedings of the3rd International Conference on Web Search and Web Data Mining, New York, Feb 4-6, 2010. New York: ACM, 2010:401-410

Xiaofeng Liu was born in Qianxian, Xianyang, China in 1983. He received the B.S. degree in Electronic Information Engineering from Northeastern University, Shenyang, China, in 2006, and the M.S. degree in Computer Technology from Northeastern University, Shenyang, China, in 2020.
Since 2013, he has been working as an experimenter at the State Key Experimental Center of Computer Science. His research interests include high performance computing, training for computer competitions, and deep learning. He also serves as the coach of the school's programming competition team and owns a number of software copyrights.

Ji Li is currently pursuing Ph.D. degree at Northeastern University in Shenyang, China. He obtained B.S. degree in Information and Computing Science from Shenyang University of Technology in 2018 and a M.S. degree in Computer Software and Theory from Northeast University in Shenyang, China in 2021. His research direction is index and graph research.

Chuanwen Li received the Ph.D. degrees in computer software and theory from Northeastern University, China, in 2011, respectively. He is a professor with Northeastern University. He has also been a Visiting Researcher with Uppsala University and Aalborg University. His current research interests include data management and big data. He is a member of CCF.