www.ietdl.org

Published in IET Software
Received on 28th November 2012
Revised on 7th August 2013
Accepted on 15th August 2013
doi: 10.1049/iet-sen.2012.0188

ISSN 1751-8806

Dual-expansion indexing for moving objects

Jun-Hong Shen[1], *Ye-In Chang*[2], *Fang-Ming Chang*[2]

[1]*Department of Information Communication, Asia University, Taichung 41354, Taiwan*
[2]*Department of Computer Science and Engineering, National Sun Yat-Sen University, Kaohsiung 80424, Taiwan*
E-mail: shenjh@asia.edu.tw

Abstract: With the development of wireless communications and mobile computing technologies, the applications of moving objects have been developed in many topics, for example, traffic monitoring. Such applications need to track the current and near-future locations of the moving objects. This motivates the development of spatial-temporal indices to support efficient querying about such locations of the moving objects. Therefore, in this study, the authors propose a dual-expansion indexing (DEI) to support the current and near-future prediction of the moving objects. To filter out more number of the data blocks that do not contain the final result, the query region can be expanded in each of eight directions individually. To further reduce the number of the data blocks that should be examined, the qualified data blocks can be expanded according to the direction towards the query region. Moreover, only the objects moving to the query region will be checked in the query process of DEI. Therefore our method can reduce more number of retrieved data blocks and that of input/output operations than the existing method. Experimental results show that the query process of DEI is more efficient than that of the existing method.

1 Introduction

With the development of wireless communication and mobile computing, the interest in the application of moving objects increases gradually, including traffic monitoring, navigation systems and geographic information systems. For example, a driver may want to know the traffic condition ahead of his current position during the next 30 min. Such applications need to track the current and near-future locations of the moving objects. The feature of the moving objects is that objects change their locations continuously [1]. Conventional spatial databases cannot support to store the moving objects efficiently, because the databases will be updated frequently. Spatial–temporal databases deal with objects that change their locations continuously. By location detection devices (e.g. global postioning system devices), the moving objects can get their current locations [2]. Then, this information is reported to the server via wireless network or other communication network. The server stores the update from the moving objects. This scenario enables location-based services, such as querying the current or near-future locations of the mobile devices [3, 4]. Each moving object has spatial and temporal attributes [5]. Therefore it is important to index moving objects for efficiently answering queries about the moving objects, that is, spatial–temporal indices.

In this paper, we focus on time slice queries for the current and near-future prediction of the moving objects. The time slice query is one kind of the predictive query that finds all objects that cross a certain area at time t_q. For example, in the traffic management system, the time slice queries can find out the areas with predicted traffic jam in the next 10 min. For location-based advertising, the time slice query can find out customers with mobile phones around the department store in the next 5 min and then advertisements can be sent to them via the short message service. To predict the current and future positions of the moving objects, extra information (e.g. the velocity and the destination) should be provided. The motion of a moving object in the d-dimensional space is modelled by a reference location $\boldsymbol{x}_{\text{ref}} = (x_1, x_2, \ldots, x_d)$ at a reference time t_{ref} and a velocity vector $\boldsymbol{v} = (v_1, v_2, \ldots, v_d)$ [5]. The predicted location $\boldsymbol{x}_q$ of the moving object at any instance time t_q can be computed by $\boldsymbol{x}_q = \boldsymbol{x}_{\text{ref}} + \boldsymbol{v} \times (t_q - t_{\text{ref}})$.

Many spatial–temporal access methods are proposed to support spatial–temporal queries for the current and predicted future positions of the moving objects. Among the existing methods, the B^x_r−tree [6] improves the CPU performance of the TPR-tree [7] by expanding the query region first, and improves the input/output (I/O) performance of the B^x-tree [8] by expanding data blocks additionally. For the problem of the current and future prediction of the moving objects, the B^x_r−tree can reduce more number of I/O operations in the query process than those methods by expanding the query region and data blocks. In the B^x_r−tree method, it partitions the data space into uniform blocks. Each object is inserted into a block. Then, the same index structure of the B^x-tree, which is an extension of the B^+-tree, is used to index these blocks. The query process of the B^x_r−tree contains three steps: the query expansion, the filtration and object checking. The query expansion step expands the query region such that it covers all possible data blocks. Next, the filtration step filters out the impossible blocks. Then, for those passed data blocks,

objects are checked for deciding whether they really should be in the result. The step of object checking is correlated with the I/O performance. However, there are some disadvantages in the B_r^x−tree. First, data blocks whose objects jump over the query region will be checked in the query expansion step and the filtration step. Second, the objects moving far away from the query region are considered in the answer in the step of object checking. Third, unrelated directions are considered in the expansion.

Therefore, in this paper, we propose a dual-expansion indexing (DEI) containing an auxiliary data structure, 'velocity histogram' and a new query process strategy to solve those problems mentioned above. The information in the velocity histogram is used to find out the candidate blocks containing the result objects. Thus, unnecessary accessing the unqualified objects in the disk can be avoided. To filter out unqualified objects for the query result as more as possible, in our proposed method, each block records the maximum and minimum velocities of each of eight directions inside, instead of recording only those of each of four directions in the B_r^x−tree method. Based on the proposed data structure, the query region can be expanded in each of eight directions individually, instead of being expanded in four directions once in the B_r^x−tree method. Moreover, in our DEI method, the data blocks can be expanded according to the direction towards the query region, instead of being expanded in four directions in the B_r^x−tree method. In this way, the query process of DEI checks less number of data blocks because it considers the minimum velocity of each of eight directions. That is, by fully exploiting the information provided in the velocity histogram, our proposed method can avoid accessing unqualified objects in the pruned blocks from the disk. Furthermore, in the qualified blocks, only the objects moving to the query region will be checked in the query process of DEI, instead of all objects in the B_r^x−tree method. Therefore our method can reduce more number of retrieved data blocks and that of I/O operations than the B_r^x−tree. From our simulation study, we have shown that the query process of the DEI method is more efficient than that of the B_r^x−tree in terms of the average number of retrieved data blocks and that of I/O operations.

The rest of this paper is organised as follows. In Section 2, we gives the related work of spatial–temporal access methods that answer queries about the current and future prediction of the moving objects. In Section 3, we present our proposed method, DEI, to answer queries about current and future. In Section 4, we compare the performance of the proposed method with that of the B_r^x−tree by simulation. Finally, in Section 5, we give the conclusion.

2 Related work

The spatial–temporal access methods for current and future data are categorised into three categories [5, 9].

- *The original space-time space:* The main idea of the original space-time space is to transform moving objects in the original space to lines in the time-space domain. Assuming that the objects move in d-dimensional space ($d = 1, 2, 3$), the future trajectories of those methods in this category may be indexed as lines in the $(d+1)$-dimensional space. The methods in this category are PMR-quadtree [10] and MOVIES [11]. Using the methods in this category to index moving objects has a main drawback: it needs large amount of space to preserve line segments [5].
- *Transformation methods:* The main idea of transformation methods is to transform moving objects in the original space to the points in the dual space (the velocity-space domain). To overcome the drawbacks of spatial–temporal indexing in the time-space domain, the time-space domain is transformed into another space. The main idea is that it is easier to represent and query the data in this new space representation [5]. The methods in this category include duality transformation [12], SV-Model [13] and PSI [14]. The transformation techniques suffer from two drawbacks: (i) there is no guarantee that objects that are near to each other in the primal space will still be near to each other in the dual space; (ii) rectangular range queries in the primal space are always transformed into polygonal range queries in the dual space, which calls for complicated algorithms for evaluation [5, 9].
- *Parametric spatial access methods:* The main idea is to make the bounding rectangles functions of time so that the enclosed moving objects will be in the same rectangle [5]. Assuming that the objects move in the d-dimensional space, this category indices data in its native d-dimensional space, which is possible by parameterising the index structure using velocity vectors and thus enables the index to be viewed at any future time. The methods in this category include the PR-tree [15], the STAR-tree [16], the TPR-tree [7], the TPR*-tree [17], the B^x-tree [8] and the B_r^x−tree [6].

Among the existing methods, the parametric spatial access methods have been applied largely, since they need little memory space to preserve parametric rectangles, and still provide a good performance. Therefore, in this paper, our research belongs to this category. The parametric spatial access methods index the objects at some reference time points. To reduce the computing time, the parametric rectangles are expanded first, instead of the motion of all moving objects. These kinds of methods are classified into two classes: (i) the expansion of data blocks and (ii) the expansion of the query region. The methods for the expansion of data blocks include the PR-tree, the STAR-tree, the TPR-tree and the TPR*-tree, and the methods for the expansion of the query region include the B^x-tree and the B_r^x−tree. According to the query time, the forward or the backward expansion is used in data blocks and the query region. If the query time t_q is earlier than the reference time t_{ref} (Case 1, the 'past' query), the query region is expanded using the actual velocities, termed as a forward expansion, and data blocks are expanded using the opposite directions of velocities, termed as a backward expansion. If the reference time t_{ref} is earlier than the query time t_q (Case 2, the 'future' query), the query region is expanded using the backward expansion, and data blocks are expanded using the forward expansion.

The TPR-tree employs the idea of parametric bounding rectangles in the R*-tree with velocities to index linear functions of time [7]. The TPR*-tree uses the same data structure as the TPR-tree, and integrates a new set of insertion and deletion algorithms that aim at minimising a certain cost function [17]. Since the bounding rectangles in the R*-tree tend to overlap, the R*-tree-based methods are prone to low update efficiency. To improve update efficiency of the moving objects, the first B^+-tree-based index, the B^x-tree [8], has been proposed. In the B^x-tree, each object position is mapped to a point in the one-dimensional space by a space-filling curve. These points are then indexed in a B^+-tree. In the query process, the B^x-tree uses the query-window enlargement to guarantee

a correct answer. The B^x_r−tree extends the B^x-tree and proposes a new query processing algorithm for the B^x-tree, which fully exploits the available data statistics to reduce the query enlargement. Zhang *et al.* [3] extends the B^x-tree [8] to support uncertain moving objects. Considering skewed velocity distributions, the velocity partitioning technique [4] is applied in the TPR*-tree and the B^x-tree to speed up query processing.

3 Dual-expansion indexing

In this section, we present our proposed DEI, which answers queries about the current and future prediction of moving objects. First, we introduce the data structure used in DEI. Second, we present the insertion algorithm for inserting a new moving object into the index. Third, the query process is described.

3.1 Data structure

In the time axis, we set n future reference time points t_{ref} to store the positions of the moving objects at those time points. The update interval between the adjacent reference time points is set to UI. When the index at the reference time becomes out-of-date, that index will be deleted and a new index will be constructed at the next reference time.

At each reference time point, a data space is partitioned into uniform data blocks, and moving objects are inserted into the corresponding blocks. That means, DEI has a record of the data space at each reference time. DEI stores the blocks as the index that is a set of two-dimensional arrays, the velocity histogram. That means, DEI can store the whole velocity histogram into the main memory. Similar to the B^x-tree [8] and the B^x_r−tree [6], at each reference time, the data blocks in the two-dimensional space are linearised by the space-filling curve. The space-filling curve is a continuous path, which passes through every point in the two-dimensional space once to map it to a one-dimensional sequence number on the curve. A sub-tree of the B^+-tree is reserved for indexing moving objects at each reference time. For an object, the value indexed by the B^+-tree is the concatenation of its partition number for the reference time and the sequence number of the space-filling curve of the data block containing it.

In the blocks and the global region, to preserve the maximum and minimum velocities of moving objects in each direction, DEI records the absolute values of the maximum and minimum velocities of each of eight directions, *L* (left), *R* (right), *U* (up), *D* (down), *LU* (left-up), *RU* (right-up), *LD* (left-down) and *RD* (right-down) directions, as depicted in Fig. 1*a*. We define the absolute value of the velocity (*AbsV*), such that we can easily explain the process of the insertion without considering that the value of the velocity is positive or negative. The absolute value of the maximum velocity (*Max_AbsV*) and that of the minimum velocity (*Min_AbsV*) in the direction mean the maximum value and the minimum value of *AbsV* in each direction, respectively. For example, there are two moving objects with velocity vectors (− 5, 1) and (3, − 4) in the two-dimensional space. (Note that in form (*a*, *b*), *a* and *b* represent the velocity in the *x*-axis and that in the *y*-axis, respectively.) That means that *AbsV*s are (5, 1) and (3, 4), respectively. Therefore *Max_AbsV* is (5, 4) and *Min_AbsV* is (3, 1). Fig. 1*a* shows the absolute maximum and minimum velocities in each direction in a block. For example, *LU.Max_AbsV* and *LU.Min_AbsV* store the absolute maximum and minimum velocity vectors of the *LU* direction in the block, respectively.

The objects in the *LU*, *LD*, *RU* and *RD* directions must be considered in the two corresponding directions: *L*, *R*, *U* and *D*. For example, the objects in the *LU* direction will be considered for *Max_AbsV*s and *Min_AbsV*s of the *L* and *U* directions. The reason is that the objects in the *LU* direction have the components of *L* and *U* directions. That is, these objects will impact the expansion of the blocks in *L* and *U* directions. In addition, the absolute maximum and minimum velocities among the moving objects in the global region are recorded in *gMax_AbsV* and *gMin_AbsV*, respectively. To preserve the positive or negative value of the original velocity, we define the unit vector (*UV*) as shown in Fig. 1*b*. Moreover, we let $P_w(x, y)$ be the endpoint of the query region or the data block, $w \in LU, LD, RU, RD$. Furthermore, we let $P_L = P_D = P_{LD}$ and $P_R = P_U = P_{RU}$. In DEI, each block has four lists: $List_{LU}$, $List_{LD}$, $List_{RU}$ and $List_{RD}$. Each object indexed in the block is linked by one of the lists according to the direction of its velocity. The objects moving towards the *L* and R directions are linked by $List_{LU}$ and $List_{RU}$, respectively. In this way, we can check only the objects in the lists moving towards the query region. The objects in the lists can be retrieved from the disk with the help of the index structure of the B^+-tree.

3.2 Insertion algorithm

In DEI, the information about a new moving object is processed by **procedure** *InsertObject* shown in Algorithm 1 (see Fig. 2), where the corresponding variables used are listed in Table 1. In **procedure** *InsertObject*, the object is

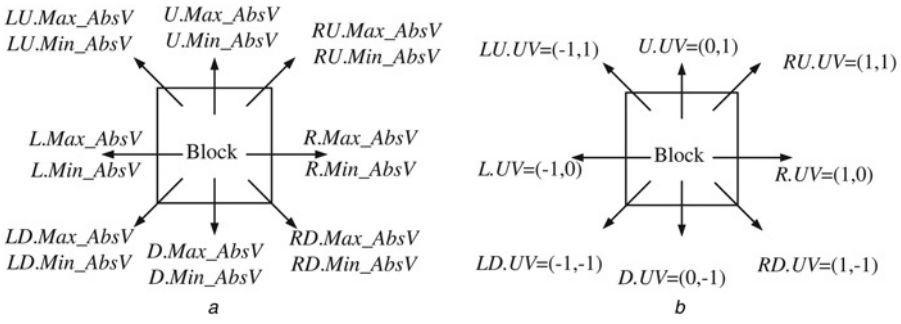

Fig. 1 *Structure of a block*

a Absolute maximum and minimum velocities in each direction in a block
b Definition of a unit vector

doi: 10.1049/iet-sen.2012.0188

Algorithm 1

```
1: procedure InsertObject(obj, t_now)
2:     for all (i ∈ RefList) do
3:         NewLoc ← FindNewLocation(obj, RefList[i], t_now)
4:         BlockId ← FindBlock(NewLoc, RefList[i])
5:         InsertBlock(BlockId, obj, newLoc)
6:     end for
7: end procedure
```

Fig. 2 *Procedure InsertObject(obj, t_now)*

Table 1 Variables used in the insertion algorithm

Notation	Description
Obj	the identifier of the object
obj.vel	the velocity of the object
obj.loc	the location of the object
RefList	the array which stores all reference times
t_now	the time of the insertion

inserted into all records of the data space at all reference time points. At each reference time point, the block which the object should be inserted into will be found out. The location *NewLoc* of the object at the reference time point is computed by **function** *FindNewLocation*. Then, **function** *FindBlock* finds out the data block *BlockId* which the new location is located in. Finally, **procedure** *InsertBlock* shown in Algorithm 2 (see Fig. 3) is provoked to insert the object into that block.

In **procedure** *InsertBlock*, the velocity of the object is first transformed into the absolute value *AbsV* by **function** *AbsoluteV*. According to the direction of the object, **function** *FindDirection* obtains the information (i.e. *MaxAbsV* and *MinAbsV*) about the direction which is the same as that of the object (*DirId*). Then, the information about the corresponding directions of the object (*corDirsId*) (i.e. those directions which have the component of the velocity of the object) is found out by **function** *FindCorrespondingDirection*. For each corresponding direction, **function** *FindMax* and **function** *FindMin* find out the new *Max_AbsV* and *Min_AbsV* by comparing the original *Max_AbsV* and *Min_AbsV* with *AbsV* of the object, respectively. For example, we have the original *Max_AbsV* = (5, 4) and *Min_AbsV* = (3, 1), and the new inserted object with *AbsV* = (2, 6). The new *Max_AbsV* and *Min_AbsV* will be (5, 6) and (2, 1), respectively. Finally, **function** *FindList* finds out which list the object is linked to according to the direction of the object, and **function** *LinkObject* adds this object to that list.

For example, Fig. 4*a* shows a block with the information about three directions. When a new object with velocity vector (−1, −7) is inserted into this block, **function** *AbsoluteV* transforms (−1, −7) into *AbsV* (1, 7). According to the velocities of the *x*- and *y*-axes, we know that *DirId* is *LD* (the left-down direction) and *corDirIds* contains *LD*, *L* and *D*. Then, *LD* is compared with (1, 7), and *L* and *D* are compared with the *x* and *y* components of (1, 7), respectively. Since *LD.Max_AbsV.y* = 4 is smaller than 7, *LD.Max_AbsV.y* is changed from 4 to 7. Under the same circumstance, *LD.Min_AbsV.x*, *L.Min_AbsV.x* and *LD.Max_AbsV.y* are also changed. Fig. 4*b* shows the result

Algorithm 2

```
1: procedure InsertBlock(BlockId, obj, newLoc)
2:     AbsV ← AbsoluteV(obj.vel)
3:     DirId ← FindDirection(BlockId, obj.vel)
4:     corDirIds ← FindCorrespodingDirections(BlockId, DirId)
5:     for all i ∈ corDirIds do
6:         corDirIds[i].Max_AbsV ← FindMax(corDirIds[i].Max_AbsV, AbsV)
7:         corDirIds[i].Min_AbsV ← FindMin(corDirIds[i].Min_AbsV, AbsV)
8:     end for
9:     ListId ← FindList(BlockId, DirId)
10:    LinkObject(ListId, obj)
11: end procedure
```

Fig. 3 *Procedure InsertBlock(BlockId, obj, newLoc)*

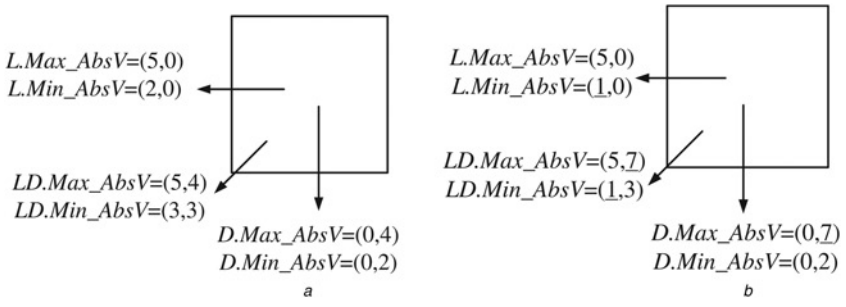

Fig. 4 *Index information in a block*

a Block recording the information of three directions
b Result after inserting an object

after the insertion of the object. Finally, the object is added to list $List_{LD}$ in the block according to its direction.

3.3 Query process

Similar to that in the B_r^x−tree, the query process in DEI contains three steps: (i) the query expansion, (ii) the filtration and (iii) object checking. Step 1 expands the query region such that it covers all possible data blocks. Step 2 expands those possible data blocks to check if they cover with the query region to filter out the impossible data blocks. Step 3 checks the qualified data blocks processed by Steps 1 and 2 to find out the final result.

In Step 1, the index with the reference time point t_{ref} close to the query time point t_q is selected. Then, the query region, *QR*, is expanded in each of eight directions with the global maximum and minimum speeds, and the difference of the query time point and the reference time point by **procedure** *ExpandQueryRegion* shown in Algorithm 3 (see Fig. 5). There are two cases of the expansion of the query region: (i) $t_q \le t_{ref}$ (Case 1, the past case); (ii) $t_q > t_{ref}$ (Case 2: the future case). If $t_q \le t_{ref}$, the query region is expanded with the global maximum and minimum *AbsV*s of the current expanding direction *ED* (lines 4–6). The reason is that the reference time goes back to the query time. Therefore the expanding direction of the query region is forward. If $t_q > t_{ref}$, the query region is expanded with the global maximum and minimum *AbsV*s of the opposite direction *OD* of direction *ED* (lines 7–9). The reason is that the reference time goes forward to the query time. Therefore the expanding direction of the query region is backward. No matter in Case 1 (the past case) or Case 2 (the future case), we expand the point of the expanding direction with *Max_AbsV* and the point of the opposite direction with *Min_AbsV*. After expanding the query region in each of eight directions, those expanded query regions are recorded in set *EQR* (line 11).

Take Fig. 6*a* as an example of Case 2 (the future case), where $t_{ref} = 10$, $t_q = 11$, $R.gMax\_AbsV = 3$ and $R.gMax\_AbsV = 2$. Consider that the query region is expanded in direction *L* in Fig. 6*a*, that is, *ED* = *L* and *OD* = *R*. Since this is the future case, the query region is expanded in direction *L* with the global maximum and minimum *AbsV*s of opposite direction *R*, that is. *R.gMax_AbsV* and *R.gMin_AbsV*, respectively. Therefore the left edge of the query region is expanded with speed $R.gMax\_AbsV = 3$ and the right edge is expanded with speed $R.gMin\_AbsV = 2$. Fig. 6*b* shows the corresponding expanded query region (EQR). For block A, it means that all objects in block *A* cannot move to the query region at the query time $t_q = 11$. The reason is that the left edge of the query region moving

Algorithm 3

1: **procedure** $ExpandQueryRegion(t_q, t_{ref}, QR)$
2: **for** each direction *ED* recorded in query region *QR* **do**
3: $OD \leftarrow$ the opposite direction of direction *ED*
4: **if** $(t_q \le t_{ref})$ **then** /* the *past* case */
5: $QR.P_{ED} \leftarrow QR.P_{ED} - ED.UV \times ED.gMax\_AbsV \times (t_q - t_{ref})$
6: $QR.P_{OD} \leftarrow QR.P_{OD} - ED.UV \times ED.gMin\_AbsV \times (t_q - t_{ref})$
7: **else** /* the *future* case */
8: $QR.P_{ED} \leftarrow QR.P_{ED} - OD.UV \times OD.gMax\_AbsV \times (t_q - t_{ref})$
9: $QR.P_{OD} \leftarrow QR.P_{OD} - OD.UV \times OD.gMin\_AbsV \times (t_q - t_{ref})$
10: **end if**
11: $EQR \leftarrow EQR \bigcup \{QR\}$
12: **end for**
13: **end procedure**

Fig. 5 *Procedure ExpandQueryRegion*

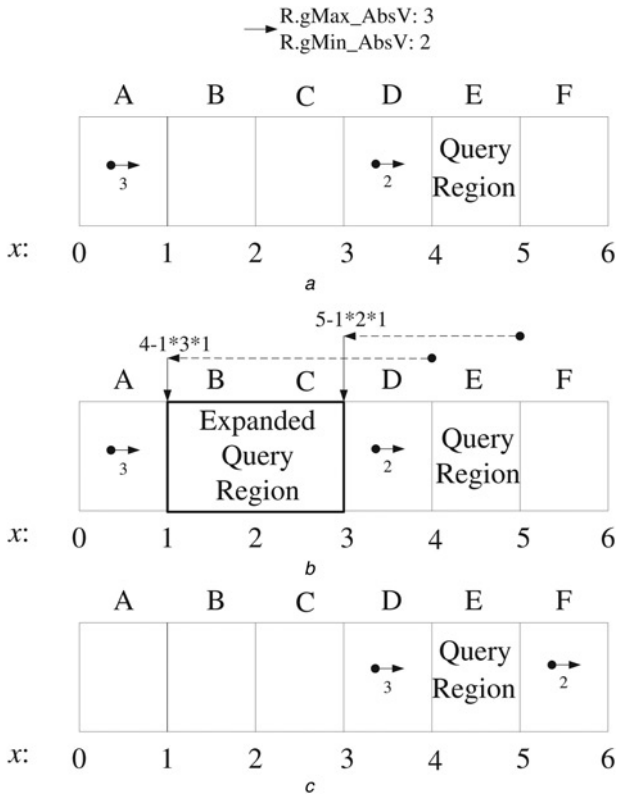

Fig. 6 *Example of the expansion of the query region*

a Case 2 (the future case)
b Expanded query region
c Locations of the objects at the query time

with speed *gMax_AbsV* cannot reach block A. For block D, it means that all objects in block D will jump over the query region at the query time. The reason is that the right edge of the query region moving with speed *gMin_AbsV* still jump over block D. Fig. 6*c* shows the locations of the objects at the query time t_q, where the objects are really not in the query region.

In Step 2, each data block intersected with the expanded query regions (set *EQR*) is expanded with the maximum and minimum speeds of a certain direction by **procedure** *ExpandBlock* shown in Algorithm 4 (see Fig. 7). Then, those expanded blocks that do not overlap with query region *QR* are filtered out. Similar to Step 1, there are two cases of the expansion of the data block: (i) $t_q \le t_{ref}$ (Case 1, the past case); (ii) $t_q > t_{ref}$ (Case 2: the future case). If $t_q \le t_{ref}$, data blocks are expanded in the directions against the query region (lines 3–4). The data blocks are expanded with the maximum and minimum *AbsV*s of the direction that is opposite to the expanding direction (lines 9–11). The reason is that the reference time goes backward to the query time. Therefore the expansion of data blocks is backward. If $t_q > t_{ref}$, data blocks are expanded in the directions towards the query region (lines 5–6) The data blocks are expanded with the maximum and minimum *AbsV*s of the direction that is the same as the expanding direction (lines 12–14). The reason is that the reference time goes forward to the query time. Therefore the expansion of data blocks is forward. As Step 1, the point of the expanding direction and that of the opposite direction are expanded with speed *Max_AbsV* and *Min_AbsV*, respectively. The reason is the same as it in Step 1. The direction considered in Step 2 is opposite to that in Step 1. This is because they have the opposite view to the query. After the expansion of the data

doi: 10.1049/iet-sen.2012.0188

Algorithm 4

1: **procedure** *ExpandBlock*(t_q, t_{ref}, EQR)
2: **for** each block DB intersected with expanded query region EQR **do**
3: **if** ($t_q \leq t_{ref}$) **then**
4: $RD \leftarrow$ the direction from query region QR to DB
5: **else**
6: $RD \leftarrow$ the direction from DB to QR
7: **end if**
8: $OD \leftarrow$ the opposite direction of direction RD
9: **if** ($t_q \leq t_{ref}$) **then** /* the *past* case */
/* EDB is the expanded data block */
10: $EDB.P_{RD} \leftarrow DB.P_{RD} + OD.UV \times DB.OD.Max\_AbsV \times (t_q - t_{ref})$
11: $EDB.P_{OD} \leftarrow DB.P_{OD} + OD.UV \times DB.OD.Min\_AbsV \times (t_q - t_{ref})$
12: **else** /* the *future* case */
13: $EDB.P_{RD} \leftarrow DB.P_{RD} + RD.UV \times DB.RD.Max\_AbsV \times (t_q - t_{ref})$
14: $EDB.P_{OD} \leftarrow DB.P_{OD} + RD.UV \times DB.RD.Min\_AbsV \times (t_q - t_{ref})$
15: **end if**
16: **if** ($EDB \bigcap QR \neq \emptyset$) **then**
17: $CDB \leftarrow CDB \bigcup \{DB\}$
18: **end if**
19: **end for**
20: **end procedure**

Fig. 7 *Procedure ExpandBlock*

blocks, if the expanded data block is intersected with the query region, this block is recorded in set *CDB* that will be examined in Step 3 (lines 16–18).

In Step 3, only those objects linked into the lists whose directions towards the query region from t_{ref} to t_q are checked to find out the final result. If $t_q \leq t_{ref}$ (the past case), the lists in the examined data block whose direction is moving away to the query region are checked. Since the lists in the block are constructed at t_{ref} that is after the query time point t_q, the objects in the lists whose direction is moving away the query region may be in the query region at t_q. If $t_q > t_{ref}$ (the future case), the lists in the examined data block whose direction is moving towards the query region are checked. Since the lists in the block are constructed at t_{ref} that is before the query time point t_q, the objects in the lists whose direction is moving towards the query region may move in the query region at t_q.

3.4 Query example

Fig. 8*a* shows an example of a one-dimensional space, where seven moving objects *o*1–*o*7 are inserted into three blocks A, B and C with the reference time $t_{ref} = 10$. Fig. 8*b* shows the corresponding index transformed by DEI. For example, in block A, the maximum and minimum absolute velocities of the *R* direction, *R.Max_AbsV* and *R.Min_AbsV*, are 3 and 2, which are from objects *o*3 and *o*2, respectively. Since only object *o*1 is moving in the *L* direction, the maximum and minimum absolute velocities of the *L* direction, *L.Max_AbsV* and *L.Min_AbsV*, are 1 and 1, which are from object *o*1, respectively. The maximum absolute velocity of the global region in the *R* direction, *R.gMax_AbsV*, is 3, which is the maximum value among *R.Max_AbsV*'s in blocks A, B and C. The minimum absolute velocity of the global region in the *R* direction, *R.gMin_AbsV*, is 2, which is the minimum value among *R.gMin_AbsV*'s in blocks A, B and C. *L.gMax_AbsV* and *L.gMin_AbsV* are processed in the similar way.

Consider that someone asks 'find all objects that cross the query region at $t_q = 11$'. First, the query region is expanded. Since the query time $t_q = 11$ is after the reference time $t_{ref} = 10$ ($t_q > t_{ref}$, the future case), the query region is expanded with *gMax_AbsV* and *gMin_AbsV* of the opposite direction of the expanding direction. Therefore, the query region is expanded in direction *L* with speed *gMax_AbsV* and *gMin_AbsV*, and in direction *R* with speed *L.gMax_AbsV* and *L.gMin_AbsV*, respectively. Fig. 8*c* shows the process and the result of Step 1. According to line 8 of **procedure** *ExpandQueryRegion* shown in Algorithm 3 (see Fig. 5), in the expansion of direction *L*, the expanded left endpoint of the query region *QR* is equal to $QR.PL - R.UV \times R.gMax\_AbsV \times (t_q - t_{ref}) = 3 - 1 \times 3 \times 1 = 0$. According to line 9 of **procedure** *ExpandQueryRegion*, in the expansion of direction *L*, the corresponding expanded right endpoint of *QR* is equal to $QR.P_R - R.UV \times R.gMin\_AbsV \times (t_q - t_{ref}) = 4 - 1 \times 2 \times 1 = 2$. The expansion of the query region in direction *R* is processed in the similar way. In Fig. 8*c*, it is obvious that blocks A, B and E are covered with the EQR. Therefore blocks A, B and E will be checked in Step 2.

Since $t_q = 11 > t_{ref} = 10$ (the future case), the filtered data blocks are expanded in the direction towards the query region with the maximum and minimum absolute velocities of the direction that is the same as the expanding direction. In Step 2, blocks A and B are expanded in direction *R* with their corresponding *R.Max_AbsV* and *R.Min_AbsV*, and block E is expanded in direction *L* with its *L.Max_AbsV* and *L.Min_AbsV*. Fig. 8*d* shows the expanded region of block A. According to line 13 of **procedure** *ExpandBlock* shown in Algorithm 4 (see Fig. 7), the right endpoint of the expanded region of block A is equal to $A.P_R + R.UV \times A.R.Max\_AbsV \times (t_q - t_{ref}) = 1 + 1 \times 3 \times 1 = 4$. According to line 14 of **procedure** *ExpandBlock*, its corresponding left endpoint is equal to $A.P_L + R.UV \times A.R.Min\_AbsV \times (t_q - t_{ref}) = 0 + 1 \times 2 \times 1 = 2$. Blocks B and E are expanded in the similar way. Their corresponding expanded regions are shown in Figs. 8*e* and *f*, respectively. It is obvious that only the expanded region of block A covers with the query region. In Step 3, since this example is of the future case, the lists in the examined data block whose direction is moving towards the query region are checked. In this

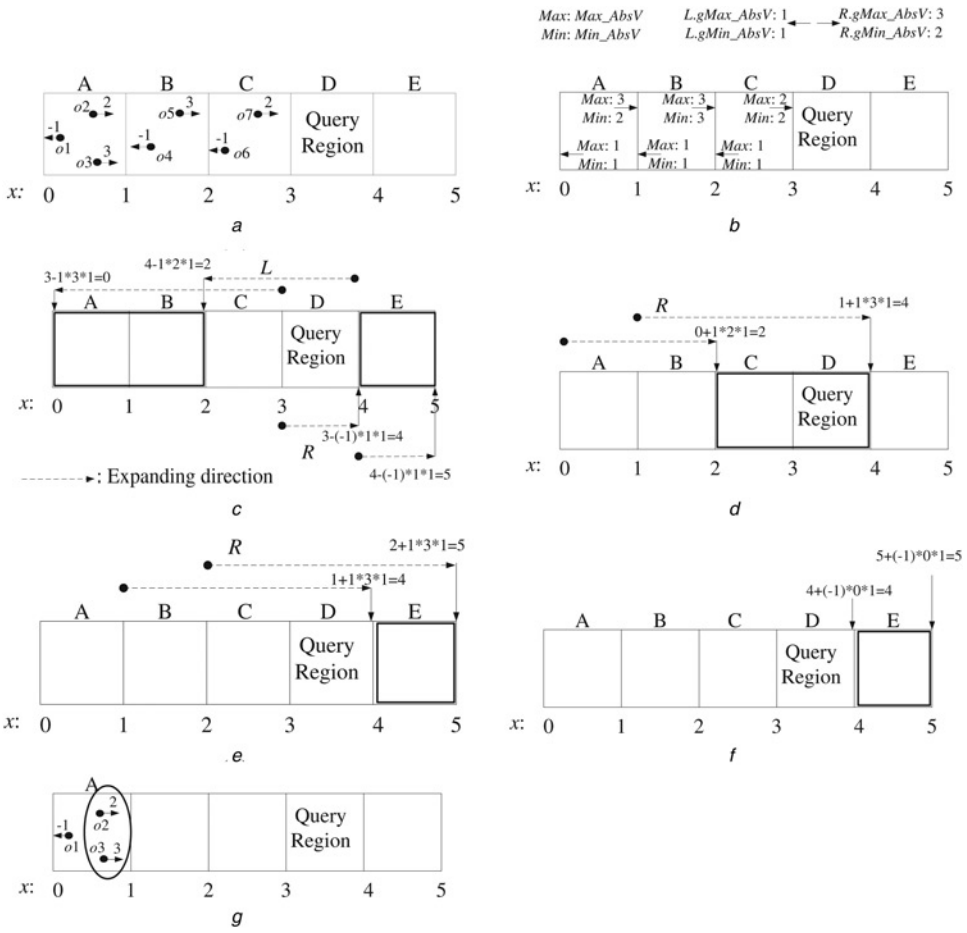

Fig. 8 *Query example*

a Moving objects at $t_{ref} = 10$
b Index transformed by DEI at $t_{ref} = 10$
c Query expansion
d Expanded region of block A
e Expanded region of block B
f Expanded region of block E
g Checked objects

example, only the objects in lists $List_{RU}$ and $List_{RD}$ in block A, which move towards the query region, are checked, that is, objects $o2$ and $o3$ in Fig. 8*g*.

4 Performance evaluations

In this section, we study the performance of our proposed DEI for the current and future prediction of moving objects. First, the performance model is presented briefly. Second, we compare the performance of DEI with that of the B_r^x−tree [6].

4.1 Performance model

Our simulation applies the same generator used in the B_r^x−tree [6]. The parameters used in our simulation are listed in Table 2, and their settings are listed in Table 3. The total data space is 512×512 km^2. The index contains two partitions at two future time points. The data space is partitioned into 512×512 equal-sized data blocks, that is, the size of a data block is 1×1 km^2. Objects are assigned either speed 25 or 200 m/s. The fractions of objects with speed 200 m/s are: 2, 10, 20, 30, 40, 50, 60, 70, 80, 90 and 98%. Parameter SF is used to control the fraction of objects with speed 200 m/s. Parameter UI is used to control the update interval between two reference time points, whose value is from 2 to 20 min in increments of 2 min. Parameter OI represents the number of objects inserted into the index. The number of objects, OI, is from 100×10^3 to 1000×10^3

Table 2 Parameter settings used in the generation of the simulation

Parameter	Description
SF	The fraction of objects with speed 200 m/s
UI	The update interval between two reference time points
OI	The number of objects inserted in the index
SE	The average spatial extent of the queries

Table 3 Parameters and their settings used in the simulation

Parameter	Setting
SF	2%, 10%, 20%, 30%, 40%, 50%, 60%, 70%, 80%, 90%, 98%
UI	2, 4, 6, 8, 10, 12, 14, 16, 18, 20
OI	100, 200, 300, 400, 500, 600, 700, 800, 900, 1000 ($\times 10^3$)
SE	1×1, 2×2, 3×3, 4×4, 5×5, 6×6, 7×7, 8×8, 9×9, 10×10 (km^2)

doi: 10.1049/iet-sen.2012.0188

in increments of 100×10^3. Queries have an average spatial extent, SE, from 1×1 to $10 \times 10\,\text{km}^2$. We consider the average number of retrieved data blocks and that of I/O operations per query as our performance measures.

4.2 Experimental results

In our simulation, we will change one of the four parameters listed in Table 3 at one time in the first four experimental results. In each experimental result, 1000 queries are executed. We define a default setting for the base case as follows. The fraction of objects with speed 200 m/s, SF, is 50%. The update interval, UI, between two reference time points is 2 min. The number of objects inserted into the index, OI, is 500×10^3, and the data distribution is uniform. The average spatial extent, SE, of the queries is $5 \times 5\,\text{km}^2$. In the fifth experimental result, we evaluate the effect of data distributions. In the last experimental result, we evaluate the effect of the update cost.

In the first experimental result, we vary parameter SF, the fraction of objects with speed 200 m/s, and the other parameters are set to the base case. Figs. 9*a* and *b* show the average number of retrieved data blocks and that of I/O operations, respectively, with the increase of the value of SF. In Fig. 9*a*, we can observe that the average number of retrieved blocks based on DEI increases slightly as the fraction of high-speed objects increases. On the other hand, the average number of retrieved blocks based on the B_r^x−tree increases dramatically as the fraction of high-speed objects increases. The reason is that the minimum velocities of eight directions in the blocks are considered in DEI. Therefore DEI can reduce the impact of the maximum velocities of eight directions in the blocks. This means that DEI can filter out more number of data blocks that do not contain the final answers than the B_r^x−tree. In Fig. 9*b*, the difference between DEI and the B_r^x−tree is obvious. In Fig. 9*a*, the results are almost the same when the fractions of objects with speed 200 m/s are 2 and 10%. However, they can be distinguished clearly in Fig. 9*b*. The reason is that the objects are partitioned into four lists in DEI. Therefore only the objects in the lists moving to the query regions will be checked. That is, the I/O performance of DEI will be improved substantially. In this experimental result, DEI has average improvements of 28 and 56% on the average number of retrieved data blocks and that of I/O operations over the B_r^x−tree, respectively. Note that the percentage of the improvement from our proposed DEI to the B_r^x−tree is computed as

$$\frac{B_r^x\text{−tree} - \text{DEI}}{B_r^x\text{−tree}} \times 100$$

In the second experimental result, we vary the parameter UI, the update interval between two reference time points. Figs. 10*a* and *b* show the average number of retrieved data blocks and that of I/O operations, respectively, with the increase of the value of UI. The average number of

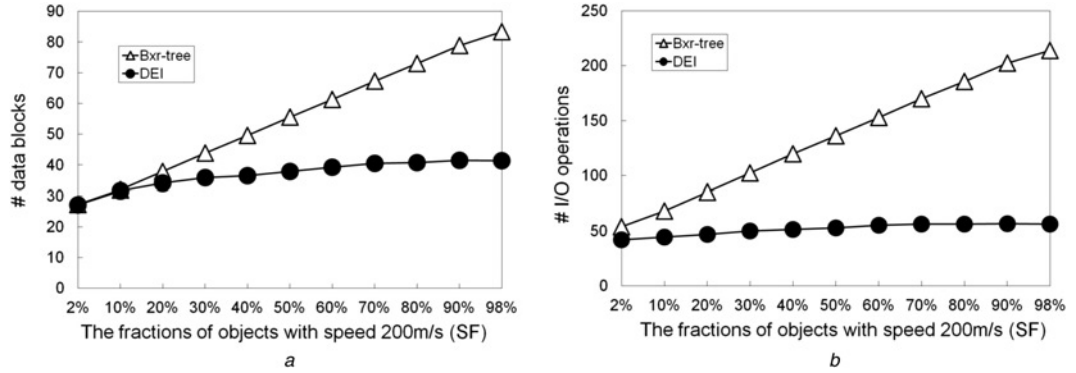

Fig. 9 *Comparison of changing the value of SF*

a Average number of retrieved data blocks
b Average number of I/O operations

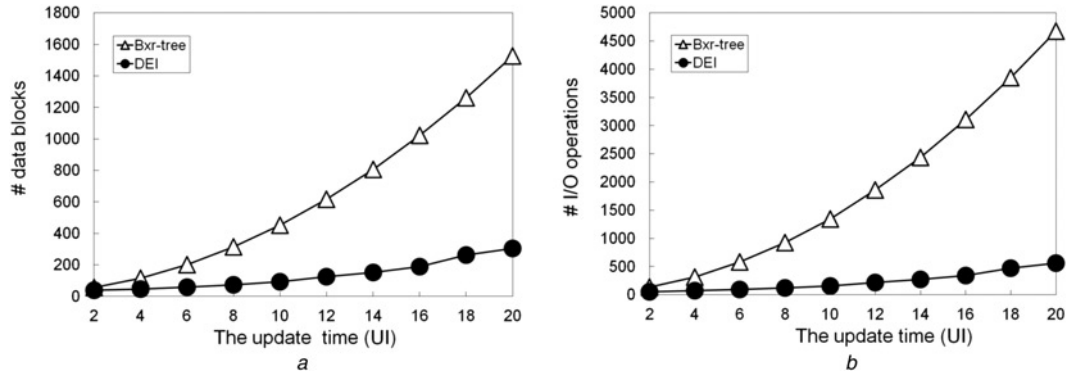

Fig. 10 *Comparison of under changing the value of UI*

a Average number of retrieved data blocks
b Average number of I/O operations

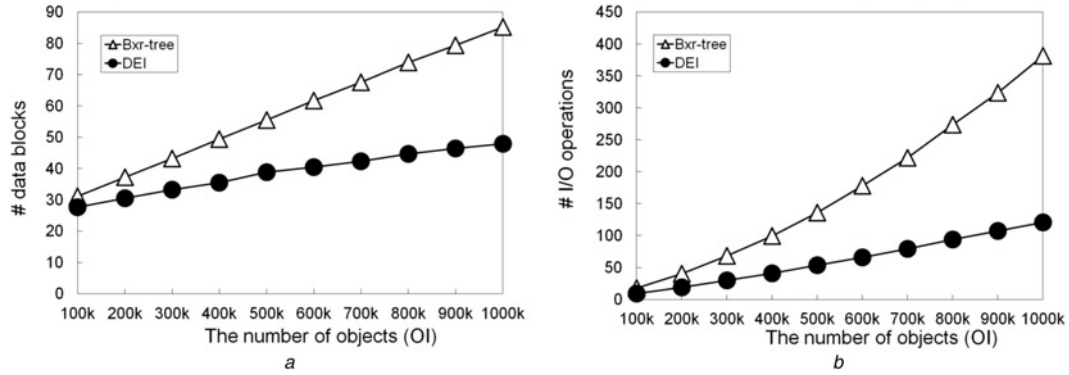

Fig. 11 *Comparison of under changing the value of OI*

a Average number of retrieved data blocks
b Average number of I/O operations

retrieved blocks and that of I/O operations based on DEI increase slightly as the update interval increases. On the other hand, the average number of retrieved blocks and that of I/O operations based on the B_r^x−tree increase a lot as the update interval increases. That is, the I/O performance of DEI is better than that of the B_r^x−tree. In this experimental result, DEI has average improvements of 72 and 84% on the average number of retrieved data blocks and that of I/O operations over the B_r^x−tree, respectively.

In the third experimental result, we vary the parameter OI, the number of objects inserted in the index. Figs. 11*a* and *b* show the average number of retrieved data blocks and that of I/O operations, respectively, with the increase of the value of OI. Although the number of the objects increases, the average number of retrieved data blocks and that of I/O operations based on DEI do not increase obviously. On the contrary, the average number of retrieved data blocks and that of I/O operations based on the B_r^x−tree increase substantially as the number of the objects increases. DEI has average improvements of 31 and 61% on the average number of retrieved data blocks and that of I/O operations over the B_r^x−tree, respectively.

In the fourth experimental result, we vary the parameter SE, the spatial extent of the query. Figs. 12*a* and *b* show the average number of retrieved data blocks and that of I/O operations, respectively, with the increase of the value of SE. In Fig. 12*a*, the results of DEI are always better than those of the B_r^x−tree. As the size of the spatial extent increases, the frequency of overlaps between the query region and data blocks increases, resulting in the increase of the average number of retrieved data blocks. In Fig. 12*b*, we can observe that the performance of I/O operations based on DEI is better than that of the B_r^x−tree. The reason is that only the objects moving to the query region will be checked in DEI. Therefore, DEI can filter out more number of useless objects than the B_r^x−tree, resulting in the decrease of the number of I/O operations to retrieve those useless objects. In this experimental result, DEI has average improvements of 33 and 62% on the average number of retrieved data blocks and that of I/O operations over the B_r^x−tree, respectively.

In the fifth experimental result, to evaluate the effect of data distributions, we generate datasets to simulate scenarios where moving objects cluster around certain locations of interest (hotspots). To generate such datasets, a set of hotspots are randomly picked. Around each hotspot, the moving objects cluster within a certain distance [18]. In this experimental result, we vary the number of hotspots from 10 to 10 000. When the number of hotspots is 10, the data distribution is heavily skewed. When the number of hotspots is 10 000, the data distribution is near uniform. The number of objects inserted into the index, OI, is set to 1000×10^3, and the other parameters are set to the base case. Figs. 13*a* and *b* shows the average number of retrieved data blocks and that of I/O operations, respectively, with the increase of the number of hotspots. From both figures, we can observe that the performance of DEI is better than that of the B_r^x−tree. When the number of

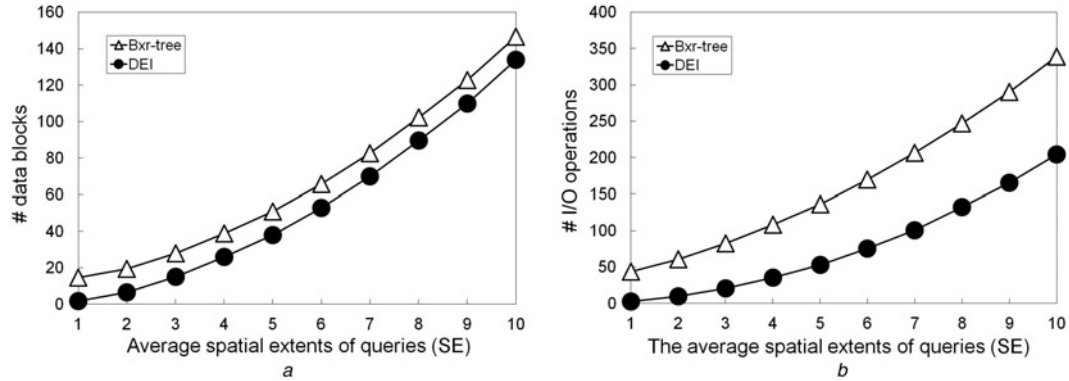

Fig. 12 *Comparison of under changing the value of SE*

a Average number of retrieved data blocks
b Average number of I/O operations

doi: 10.1049/iet-sen.2012.0188

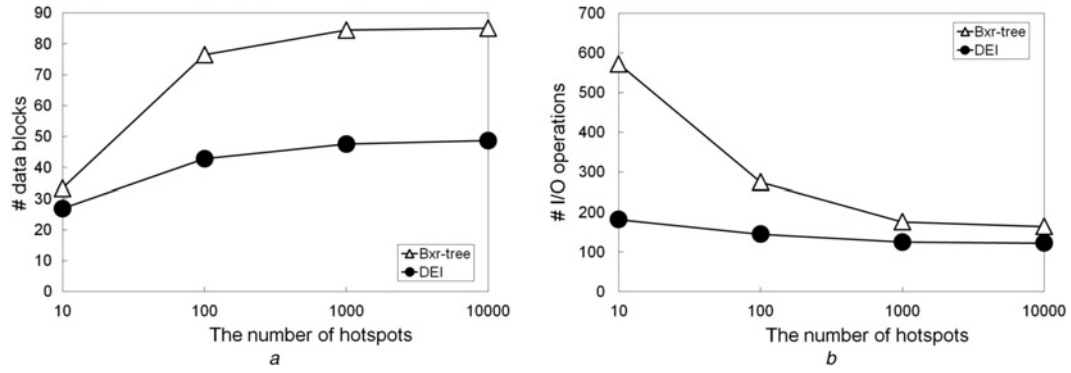

Fig. 13 *Comparison of under changing the number of hotspots*

a Average number of retrieved data blocks
b Average number of I/O operations

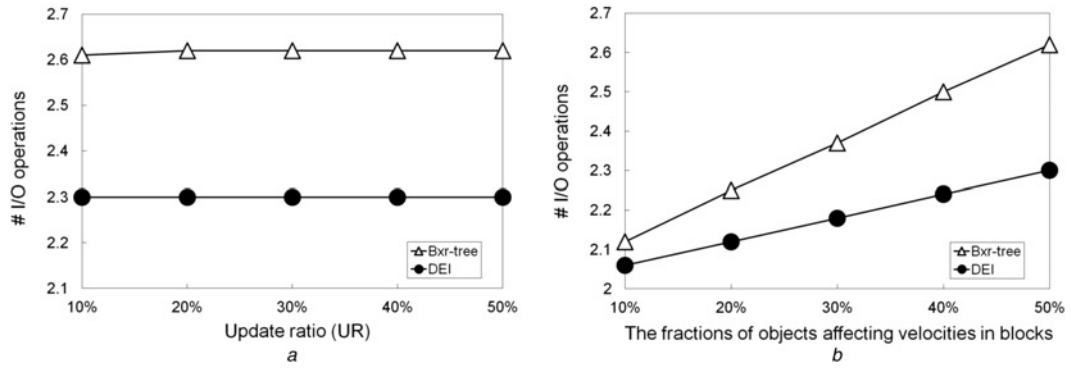

Fig. 14 *Comparison of update cost*

a Effect of changing the update ratio
b Effect of changing the fractions of objects affecting velocities in blocks

hotspots is small, the moving objects cluster in few blocks. Therefore the average number of retrieved data blocks is small. However, in this case, each block may index a large number of the moving objects, resulting in the increase of the average number of I/O operations. On the other hand, for the same value of OI, when the number of hotspots is large, the moving objects cluster in many blocks. Therefore the average number of retrieved data blocks is large. However, each block indices a few number of the moving objects, resulting in the decrease of the average number of I/O operations. In this experimental result, DEI has average improvements of 38 and 42% on the average number of retrieved data blocks and that of I/O operations over the B_r^x-tree, respectively.

In the last experimental result, we evaluate the effect of the update cost. The update cost amortised over insertion and deletion is measured by the number of I/O operations. First, we evaluate the impact on the update ratio of the updated moving objects to the whole moving objects, UR. Second, we evaluate the impact on the fraction of updated objects that affect the maximum or minimum absolute velocity in blocks, AF. If the updated object contributes to the maximum or minimum absolute velocity in a block, the corresponding moving objects in this block should be retrieved to recompute the maximum or minimum absolute velocity. Otherwise, only I/O operations for that updated object are needed. The number of objects inserted into the index, OI, is set to 1000×10^3. Fig. 14*a* shows the average number of I/O operations with the increase of the value of UR from 10 to 50% and AF = 50%. Both DEI and the B_r^x-tree have stable performance with the increase of the value of UR. The average number of I/O operations based on DEI is less than that on the B_r^x-tree. The reason is as follows. In DEI, if the updated object affects the maximum or minimum absolute velocity in the block, only the moving objects stored in the same list in the block are retrieved to recompute that velocity. On the contrary, in the B_r^x-tree, all the moving objects in the block are retrieved to recompute that velocity. In Fig. 14*a*, DEI has an average improvement of 12% on the average number of I/O operations over the B_r^x-tree. Fig. 14*b* shows the average number of I/O operations with the increase of the value of AF from 10 to 50% and UR = 50%. As the value of AF increases, the average number of I/O operations based on DEI and that on the B_r^x-tree increase. The reason is as follows. As value of AF increases, the number of the updated objects affecting the maximum or minimum absolute velocity in blocks increases. This increases the number of I/O operations to retrieve the corresponding moving objects indexed in blocks to recompute the maximum or minimum absolute velocity. In Fig. 14*b*, DEI has an average improvement of 8% on the average number of I/O operations over the B_r^x-tree.

5 Conclusions

In this paper, we focus on the problem of the current and future prediction for moving objects in spatial–temporal

databases. We have proposed a DEI to answer the predictive query from current to future. In our method, the data space is divided into blocks, and each block preserves the maximum and minimum velocities of eight directions. Based on the proposed data structure, the three steps of the query process are improved substantially. In the query expansion step, the query region can be expanded in eight directions individually, instead of being expanded in four directions once in the B_r^x−tree method. In the filtration step, the data blocks can be expanded according to the direction towards the query region, instead of being expanded in four directions in the B_r^x−tree method. In the step of object checking, only the objects moving to the query region will be checked in the query process of DEI. That is, the query process of DEI can filter out the useless data efficiently. Our experimental results have shown that the query process of DEI is more efficient than that of the B_r^x−tree in terms of the average number of retrieved data blocks and that of I/O operations.

6 Acknowledgment

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC 99-2221-E-468-019.

7 References

1 Chen, S., Nascimento, M.A., Ooi, B.C., Tan, K.L.: 'Continuous online index tuning in moving object databases', *ACM Trans. Database Syst.*, 2010, **35**, (3), pp. 17:1–52
2 Chen, H.L., Chang, Y.I.: 'Nine-areas-tree-bit-patterns-based method for continuous range queries over moving objects', *IET Softw.*, 2011, **5**, (1), pp. 54–69
3 Zhang, M., Chen, S., Jensen, C.S., Ooi, B.C., Zhang, Z.: 'Effectively indexing uncertain moving objects for predictive queries'. Proc. VLDB Endowment, 2009, vol. 2, no 1, pp. 1198–1209
4 Nguyen, T., He, Z., Zhang, R., Ward, P.: 'Boosting moving object indexing through velocity partitioning'. Proc. VLDB Endowment, 2012, vol. 5, no 9, pp. 860–871
5 Mokbel, M.F., Ghanem, T.M., Aref, W.G.: 'Spatio-temporal access methods', *IEEE Data Eng. Bull.*, 2003, **26**, (2), pp. 40–49
6 Jensen, C.S., Tiesyte, D., Tradisauskas, N.: 'Robust B^+-tree-based indexing of moving objects'. The Seventh Int. Conf. Mobile Data Management, 2006, pp. 12–21
7 Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: 'Indexing the positions of continuously moving objects'. Proc. ACM SIGMOD Int. Conf. Management of Data, 2000, pp. 331–342
8 Jensen, C.S., Lin, D., Ooi, B.C.: 'Query and update efficient B^+-tree based indexing of moving objects'. Proc. 30th VLDB Conf., 2004, pp. 768–779
9 Nguyen-Dinh, L.V., Aref, W.G., Mokbel, M.F.: 'Spatio-temporal access methods: part 2 (2003–2010)', *IEEE Data Eng. Bull.*, 2010, **33**, (2), pp. 46–55
10 Tayeb, J., Ulusoy, O., Wolfson, O.: 'A quadtree-based dynamic attribute indexing method', *Comput. J.*, 1998, **41**, (3), pp. 185–200
11 Dittrich, J., Blunschi, L., Salles, M.A.V.: 'Indexing moving objects using short-lived throwaway indexes'. Proc. 11th Int. Symp. Advances in Spatial and Temporal Databases, 2009, pp. 189–207
12 Kollios, G., Gunopulos, D., Tsotras, V.J.: 'On indexing mobile objects'. Proc. 18th ACM SIGACT-SIGMOD-SIGART Symp. Principles of Database Systems, 1999, pp. 261–272
13 Chon, H.D., Agrawal, D., Abbadi, A.E.: 'Storage and retrieval of moving objects'. Proc. Second Int. Conf. Mobile Data Management, 2001, pp. 173–184
14 Porkaew, K., Lazaridis, I., Mehrotra, S.: 'Querying mobile objects in spatio-temporal databases'. Proc. Int. Symp. Advances in Spatial and Temporal Databases, 2001, pp. 59–78
15 Cai, M., Revesz, P.: 'Parametric R-tree: an index structure for moving objects'. Proc. Int. Conf. Management of Data, 2000, pp. 57–64
16 Procopiuc, C.M., Agarwal, P.K., Peled, S.H.: 'STAR-tree: an efficient self-adjusting index for moving objects'. Proc. Workshop on Algorithm Engineering and Experimentation, 2002, pp. 178–193
17 Tao, Y., Papadias, D., Sun, J.: 'The TPR*-tree: an optimized spatio-temporal access method for predictive queries'. Proc. 29th VLDB Conf., 2003, pp. 790–801
18 Chen, S., Jensen, C.S., Lin, D.: 'A benchmark for evaluating moving object indexes'. Proc. VLDB Endowment, 2008, vol. 1, no 2, pp. 1574–1585

doi: 10.1049/iet-sen.2012.0188