The VLDB Journal manuscript No.
(will be inserted by the editor)

Man Lung Yiu · Yufei Tao · Nikos Mamoulis

The B^{dual}-Tree: Indexing Moving Objects by Space Filling Curves in the Dual Space

Received: date / Accepted: date

Abstract Existing spatiotemporal indexes suffer from either large update cost or poor query performance, except for the B^x-tree (the state-of-the-art), which consists of multiple B^+-trees indexing the 1D values transformed from the (multi-dimensional) moving objects based on a space filling curve (Hilbert, in particular). This curve, however, does not consider object velocities, and as a result, query processing with a B^x-tree retrieves a large number of false hits, which seriously compromises its efficiency. It is natural to wonder "can we obtain better performance by capturing also the velocity information, using a Hilbert curve of a higher dimensionality?". This paper provides a positive answer by developing the B^{dual}-tree, a novel spatiotemporal access method leveraging pure relational methodology. We show, with theoretical evidence, that the B^{dual}-tree indeed outperforms the B^x-tree in most circumstances. Furthermore, our technique can effectively answer progressive spatiotemporal queries, which are poorly supported by B^x-trees.

Keywords Access Method · Spatiotemporal · Space Filling Curve

1 Introduction

A spatiotemporal database supports efficient query processing on a large number of moving objects, and has numerous applications (e.g., traffic monitoring, flight control, etc.) in practice. The existing studies can be classified into two categories, depending on whether they focus on *historical retrieval*, or *predictive search*. In this paper, we consider predictive search, where the goal is to report the objects expected to qualify a predicate in the future (e.g., find the aircrafts that will appear over Hong Kong in the next 10 minutes).

Man Lung Yiu
Department of Computer Science, University of Hong Kong
E-mail: mlyiu2@cs.hku.hk

Yufei Tao
Department of Computer Science, City University of Hong Kong
E-mail: taoyf@cs.cityu.edu.hk

Nikos Mamoulis
Department of Computer Science, University of Hong Kong
E-mail: nikos@cs.hku.hk

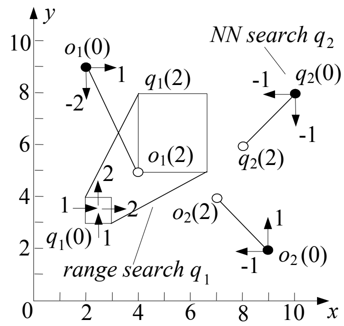


Fig. 1 Examples of spatiotemporal data and queries

An object is a multi-dimensional point moving with a constant velocity, and issues an update to the server whenever its velocity changes. In Figure 1, object o_1 is at coordinates (2, 9) at time 0, and its velocities (represented with arrows) on the x- and y- dimensions equal 1 and -2, respectively. A negative value means that the object is moving towards the negative direction of an axis. Similarly, object o_2 positions at (9, 2) at time 0, and is moving at velocity -1 (1) on the x- (y-) axis.

A *range query* returns the objects that will appear (based on their existing motion parameters) in a moving rectangle q during a (future) time interval qt. Figure 1 shows a query q_1 with $qt = [0, 2]$, whose extents at time 0 correspond to box $q_1(0)$. The left (right) edge of q_1 moves towards right at a velocity 1 (2), and the velocity of its upper (lower) boundary is 2 (1) on the y-dimension. Box $q_1(2)$ demonstrates the extents of q_1 at time 2. Notice that $q_1(2)$ has a larger size than $q_1(0)$ since the right (upper) edge of q_1 moves faster than the left (lower) one. The query result contains a single object o_1, whose location (4, 5) at time 2 falls in $q_1(2)$.

Given a moving point q and a time interval qt, a *nearest neighbor* (NN) query finds the object with the smallest "minimum distance" to q during qt. In Figure 1, for instance, query q_2 has a location (10, 8) at time 0, and moves with ve-

Table 1 Comparison of predictive spatiotemporal indexes

	B^{dual}	**B**^{x}	**TPR***	**STRIPES**
Query cost	low	high	low	high
Update cost	low	low	high	low
Storage size	low	low	low	high
DBMS integration	easy	easy	hard	hard

locity -1 on both axes. Assuming qt = [0, 2], the NN of q_2 is o_2, whose minimum distance $\sqrt{5}$ to q_2 (obtained at time 2) is smaller than that $\sqrt{17}$ of o_1. In general, a kNN query returns the k objects with the smallest minimum distances.

Motivation and Contributions. Existing indexes [16, 13, 6] for moving points suffer from certain disadvantages. Although the TPR-tree [15, 16] has good query performance, it incurs expensive update cost, and thus, is not appropriate for real-time applications with frequent updates. STRIPES [13] is efficient to update, but (as explained in Section 2.1) has high space consumption and low query performance. Neither structure can be easily integrated in an existing DBMS because they are based on techniques that are not supported by a relational database. The B^x-tree [6], the state-of-the-art, involves several B^+-trees indexing the order of objects on a space filling curve (e.g., Hilbert [3]). Hence, it can be incorporated into an off-the-shelf DBMS (e.g., Oracle, DB2, etc.). As explained in Section 2, however, B^x-trees do not achieve satisfactory query efficiency due to the large number of "false hits" (i.e., non-qualifying objects that need to be inspected).

In this paper, we present the B^{dual}-tree, which combines the advantages of the existing solutions without sharing their disadvantages. Table 1 summarizes the properties of the new structure compared to the previous indexes. In particular, the B^{dual}-tree (i) *handles both updates and queries effectively*, (ii) *is space-efficient*, and (iii) *is readily implementable using pure relational technologies, as long as B^+-trees are supported*. The B^{dual}-tree is motivated by a simple observation that the Hilbert curve deployed in a B^x-tree considers only objects' locations (i.e., not velocities), which leads to the retrieval of numerous false hits. The B^{dual}-tree is also based on B^+-trees, but avoids false hits by indexing a Hilbert curve of a higher dimensionality, which captures both locations and velocities.

It turns out that the above observation generates numerous non-trivial issues. From the algorithmic perspective, specialized methods must be designed for utilizing a high dimensional Hilbert curve to perform range and kNN search (the original B^x solutions are no longer applicable, as explained in Section 4.3). From a theoretical viewpoint, it is well known that, as the dimensionality increases, the efficiency of Hilbert curves drops, making it important to justify why the penalty can be compensated by retrieving fewer false hits. We derive analytical formulae that elaborate why and when a curve capturing velocities provides better query performance. Our equations quantify the cost of B^{dual}- and B^x-trees, and prove that our method outperforms B^x-trees in most circumstances.

We further demonstrate the superiority of the B^{dual}-tree by showing that it has higher applicability, and can effectively answer progressive queries that are poorly supported by B^x-trees. For example, a progressive (aggregate) range query aims at continuously refining its estimate about the *number* of qualifying objects, and (when allowed to execute till termination) returns the actual result. A B^{dual}-tree can produce a highly accurate estimate *well before* the query terminates, while a B^x-tree incurs significant error until nearly the end of processing.

Specifically, our contributions include:

– establishing the importance of capturing velocities in indexing moving objects based on space filling curves;
– developing the algorithms for using a velocity-conscious Hilbert curve to solve various types of queries;
– careful analysis of the performance of B^{dual}- and B^x-trees;
– solutions for progressive spatiotemporal search;
– extensive experimental evaluation of the proposed and existing spatiotemporal indexes.

The rest of the paper is organized as follows. Section 2 reviews the previous work directly related to ours. Section 3 formally defines the problem. Section 4 explains the structure of the B^{dual}-tree and its range/kNN algorithms. Section 5 provides theoretical justification about the superiority of our technique over B^x-trees. Section 6 discusses progressive processing, while Section 7 experimentally compares B^{dual}-trees against the previous structures. Finally, Section 8 concludes the paper with directions for future work.

2 Related Work

The existing predictive spatiotemporal structures can be classified into 3 categories, depending on whether they focus on the dual space, adopt the TPR-tree representation, or resort to a space filling curve. Next, we discuss each category in turn.

2.1 Dual Space Indexing

Kollios et al. [7] present a transformation that converts a 1D moving object to a static point in a 2D "dual space". Agarwal et al. [1] extended the transformation to arbitrary dimensionality, and propose theoretical indexes that achieve good asymptotic performance. These solutions, however, are not efficient in practice due to the large hidden constants in their complexities.

Kollios et al. [8] developed a practical access method based on similar transformations. Let o be a 2D point whose movement on the i-th dimension ($1 \leq i \leq 2$) is given by $o[i](t) = o[i] + o.v[i] \cdot (t - t_{ref})$, where $o.v[i]$ is its velocity along this dimension, and $o[i](t)$, $o[i]$ are its i-th coordinate at a future timestamp t and the (past) reference time t_{ref}, respectively. The *Hough-X representation* of o is a vector ($o.v[1]$, $o[1]$, $o.v[2]$, $o[2]$), and its *Hough-Y representation*

$(\frac{-o[1]}{o.v[1]}, \frac{1}{o.v[1]}, \frac{-o[2]}{o.v[2]}, \frac{1}{o.v[2]})$. Accordingly, four 2D R-trees are created to manage the following 2D spaces, respectively:

- Hough-X of Dimension 1 containing points of the form $(o.v[1], o[1])$;
- Hough-X of Dimension 2 for points of $(o.v[2], o[2])$;
- Hough-Y of Dimension 1 for $(\frac{-o[1]}{o.v[1]}, \frac{1}{o.v[1]})$;
- Hough-Y of Dimension 2 for $(\frac{-o[2]}{o.v[2]}, \frac{1}{o.v[2]})$.

An object o may be inserted in various ways depending on the velocities of o. If $o.v[1]$ is small[1] (or large), a point $(o.v[1], o[1])$ (or $(\frac{-o[1]}{o.v[1]}, \frac{1}{o.v[1]})$) is inserted in the R-tree managing the Hough-X (or Hough-Y) of Dimension 1. Similarly, if $o.v[2]$ is small (or large), a point $(o.v[2], o[2])$ (or $(\frac{-o[2]}{o.v[2]}, \frac{1}{o.v[2]})$) is incorporated. Hence, two R-tree insertions are needed for o, whose entry in each R-tree, however, contains the object's complete motion parameters (i.e., the parameters are duplicated).

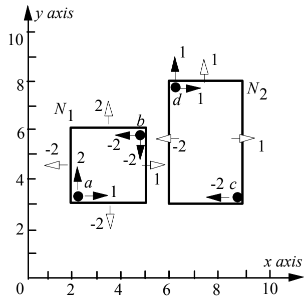
To evaluate a range query, the algorithm of [8] relies on a heuristic that decides an appropriate dimension to search (i.e., only one dimension is considered). Assume that the first dimension is chosen; then, the query is converted to two "simplex queries" in the Hough-X and Hough-Y spaces of the dimension, which are answered using the R-trees. The problem with this approach is that *all* objects qualifying the query along *only one* dimension must also be retrieved. Consider uniform data distribution and a query with selectivity 1/10 along each dimension. Around 1/100 of the objects satisfy the query, whereas the above algorithm may access 1/10 of the dataset, fetching an excessive number of false hits.

Patel et al. [13] propose STRIPES, where 2D moving objects are mapped to 4D points (by the Hough-X transformation) that are indexed by a PR bucket quadtree. Since the tree includes data on both dimensions, STRIPES does not retrieve false hits. This, however, does not imply lower query cost, because a node in STRIPES may contain an arbitrarily small number of entries, and hence, more pages need to be accessed to obtain the same number of results. Furthermore, low page utilization also leads to large space consumption. To alleviate the problem, the authors of [13] suggest a "half-page" storage scheme. Specifically, a leaf node with occupancy at most 50% is stored in half of a page, whereas a full page is used for leaf nodes with over 50% occupancy (see [13] for details).

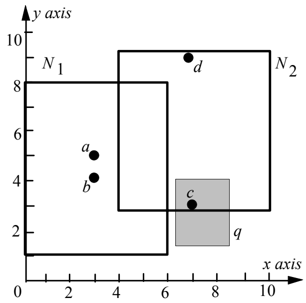
2.2 The TPR-Tree

Saltenis et al. [15] propose the TPR-tree (later improved in [16]) that augments R-trees [4] with velocities to index moving objects. Figure 2a shows an example. The black dots represent the positions of 4 objects a, b, c, d at time 0, and the arrows indicate their movements. Figure 2b illustrates the object locations at timestamp 1.

A node in the TPR-tree is represented as a *moving rectangle* (MOR), which includes an SBox and a VBox. The



(a) SBox/VBox at time 0



(b) Node extents at time 1

Fig. 2 A TPR-tree example

SBox is a rectangle that tightly encloses the locations of the underlying objects at time 0, while the VBox is a vector bounding their velocities. For example, the SBox of node N_1 is a rectangle with a projection [2, 5] ([3, 6]) on the x- (y-) dimension. Its VBox equals (-2, 1, -2, 2), where the first/second number captures the smallest/largest object velocity on the x-dimension (decided by b, a respectively), and similarly, the third and fourth values concern the y-axis. Figure 2 demonstrates a VBox with 4 white arrows attached to the edges of the corresponding SBox.

The extents of an MOR grow with time (at the speeds indicated by its VBox) so that at any future timestamp it contains the locations of the underlying objects, although it is not necessarily tight. For example, in Figure 2b, at time 1 the MOR of N_1 (or N_2) is considerably larger than the minimum bounding rectangle for its objects. Consider a range query at time 1 whose search region q is the shaded rectangle in Figure 2b. Since N_1 at time 1 does not intersect q, it does not contain any result, and can be pruned from further consideration. On the other hand, the query examines N_2, which contains the only qualifying object c.

The TPR-tree has been deployed to solve a large number of spatiotemporal problems (e.g., kNN retrieval [2], location-based queries [19], etc.). However, it has a major defect: each insertion/deletion requires numerous page accesses. Therefore, TPR-trees are not feasible for real-life applications where objects issue updates frequently.

2.3 The B^x-Tree

The solutions in the previous sections cannot be easily integrated into an existing relational database, since considerable changes are required in the "kernel" of a system (e.g., query optimization, concurrency control, introducing "half-pages", etc.). Motivated by this, Jensen et al. [6] propose the B^x-tree, which consists of B^+-trees indexing the transformed 1D values of moving objects based on a space filling curve (e.g. Hilbert curve). In [9], B^x-trees are extended to manage historical data.

Figure 3 shows an exemplary B^x-tree on 4 moving points. The location of an object at the reference time 0 is mapped to a Hilbert value, which is indexed by a B^+-tree. Object

[1] We refer the interested readers to [8] for the criteria of "small".

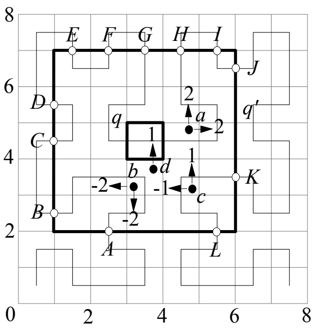


Fig. 3 A B^x-tree example

updates are highly efficient by resorting to the B^+ insertion/deletion procedures. To process a range query, the query region is enlarged to cover the locations of the qualifying objects at time 0. Consider, for example, the small rectangle in Figure 3 as a range query q at timestamp 1. To avoid false misses, q is expanded to rectangle q', according to the maximum object velocities on the two dimensions. For example, since 2 is the largest velocity along the positive direction of the x-axis, the distance between the right edges of q' and q equals 2 (i.e., the length traveled with speed 2 in one timestamp). The enlargement guarantees that if an object appears in q at time 1, its location at time 0 must fall in q'.

Region q' intersects the Hilbert curve into 6 one-dimensional intervals: AB, CD, EF, GH, IJ, KL shown in Figure 3. As a result, 6 one-dimensional range queries are executed on the B^+-tree for retrieving the points in q'. For each resultant object, its actual location and velocity are verified against the original query. In this example, only c and d satisfy the query, although all the objects are examined. In the sequel, we refer to the processing strategy as the *1D-range reduction*, because it reduces spatiotemporal search to several 1D range queries on the B^+-tree.

Since expanding the query based on the maximum velocities of the entire dataset may lead to an excessively large search region, the B^x-tree uses histograms to maintain the largest velocities of objects in various parts of the data space (so that smaller query enlargement is sufficient based on these velocities). However, in case there are slow and fast objects across the whole data space, the benefits of the histogram are limited, in which case the query performance of B^x-trees can be much worse than that of TPR-trees. Another problem of the B^x-tree is that it does not support kNN queries efficiently, because kNN search is performed using iterative range queries (by expanding the search area incrementally), as opposed to a single-traversal algorithm [2] for the TPR-tree.

3 Problem Definition and Notations

We represent a d-dimensional (in practice, d = 2 or 3) moving point o with

– a *reference timestamp* $o.t_{ref}$,
– its coordinates $o[1]$, $o[2]$, ..., $o[d]$ at time $o.t_{ref}$, and
– its current velocities $o.v[1]$, $o.v[2]$, ..., $o.v[d]$.

For example, object o_1 in Figure 1 has reference time $o_1.t_{ref} = 0$, coordinates $o_1[1] = 2$, $o_1[2] = 9$, and velocities $o_1.v[1]$ = 1, $o_1.v[2] = -2$. We use vector $o(t)$ = ($o[1](t)$, $o[2](t)$, ..., $o[d](t)$) to denote the location of o at a timestamp $t \geq o.t_{ref}$, where, for $1 \leq i \leq d$:

$$o[i](t) = o[i] + o.v[i] \cdot (t - o.t_{ref}) \tag{1}$$

A database consists of N moving points o, each of which issues an update whenever its velocity changes. The reference time $o.t_{ref}$ equals the time of its last update. In accordance to the existing techniques [1,7,6,13], we consider that an object issues at least one update every T timestamps.

A d-dimensional moving rectangle (MOR) r is captured by

– a *reference timestamp* $r.t_{ref}$,
– a *spatial box* (SBox), a $2d$-dimensional vector ($r_{\vdash}[1]$, $r_{\dashv}[1]$, ..., $r_{\vdash}[d]$, $r_{\dashv}[d]$), where [$r_{\vdash}[i]$, $r_{\dashv}[i]$] is the i-th ($1 \leq i \leq d$) projection of r at time $r.t_{ref}$, and
– a *velocity box* (VBox), a $2d$-dimensional vector ($r.V_{\vdash}[1]$, $r.V_{\dashv}[1]$, ..., $r.V_{\vdash}[d]$, $r.V_{\dashv}[d]$), where $r.V_{\vdash}[i]$ (or $r.V_{\dashv}[i]$) indicates the velocity of the left (or right) edge on the i-th dimension.

Denoting the spatial extents of r at a timestamp $t \geq r.t_{ref}$ as $r(t)$ = ($r_{\vdash}[1](t)$, $r_{\dashv}[1](t)$, ..., $r_{\vdash}[d](t)$, $r_{\dashv}[d](t)$), we have:

$$r_{\vdash}[i](t) = r_{\vdash}[i] + r.V_{\vdash}[i] \cdot (t - r.t_{ref})$$
$$r_{\dashv}[i](t) = r_{\dashv}[i] + r.V_{\dashv}[i] \cdot (t - r.t_{ref})$$

A range query specifies a time interval qt = [$qt_{\vdash}$, $qt_{\dashv}$], and an MOR q whose reference time is $qt_{\vdash}$. For instance, for the range search in Figure 1, qt = [0, 2], and the query q_1 is an MOR with reference time 0, SBox (2, 3, 3, 4), and VBox (1, 2, 1, 2). An object o satisfies q if $o(t)$ falls in $q(t)$ for some $t \in qt$.

A k nearest neighbor query has a time interval qt = [$qt_{\vdash}$, $qt_{\dashv}$], and a moving point q with reference time $qt_{\vdash}$. The minimum distance $d_{min}(o, q)$ between an object o and q equals the shortest Euclidean distance between $o(t)$ and $q(t)$ for all $t \in qt$. The query result consists of the k objects with the smallest d_{min}.

For both query types, we say that q is a *timestamp query* if qt involves a single timestamp (i.e., $qt_{\vdash}$ = $qt_{\dashv}$); otherwise, q is an *interval query*. Our objective is to minimize the CPU and I/O cost in answering a query.

4 The B^{dual}-Tree

Section 4.1 discusses the structure of the proposed index and its update algorithms. Section 4.2 clarifies the decision of a crucial parameter of the B^{dual}-tree. Section 4.3 explains why the query algorithms of B^x-trees are inefficient for B^{dual}-trees. Section 4.4 illustrates the concept of "perfect MOR", based on which Section 4.5 elaborates the range and kNN algorithms.

4.1 The Structure and Update Algorithms

A B^{dual}-tree has two parameters: a *horizon* H, and a *reference time* T_{ref}. H decides the farthest future time that can be efficiently queried. Similar to TPR-trees [15], a B^{dual}-tree constructed at time t optimizes queries about the period $[t, t + H]$. Queries that concern timestamps later than $t + H$ are also correctly answered, but they are not optimized due to their lower importance (predicting about a distant timestamp is not useful since many objects may have issued updates by then).

The second parameter T_{ref} is needed to convert data to their duals. T_{ref} is not necessarily equal to the construction time of the tree (the computation of T_{ref} will be discussed in the next section). Let o be a moving point with a reference timestamp $o.t_{ref}$, coordinates $o[1], ..., o[d]$, and velocities $o.v[1]$, ..., $o.v[d]$; its dual is a $2d$-dimensional vector:

$$o^{dual} = (o[1](T_{ref}), ..., o[d](T_{ref}), o.v[1], ..., o.v[d])$$

where $o[i](T_{ref})$ is the i-th coordinate of o at time T_{ref}, and is given by

$$o[i](T_{ref}) = o[i] + o.v[i] \cdot (T_{ref} - o.t_{ref})$$

Equivalently, o^{dual} is a point in a $2d$-dimensional *dual space*, which contains d *location dimensions* (for the first d components of o^{dual}) and d *velocity dimensions*. The dual space can be mapped to a 1D domain using any space filling curve. We choose the Hilbert curve because it preserves the spatial locality better than other curves [6], leading to lower query cost. The Hilbert value of o^{dual} can be computed using a standard algorithm [3], based on a *partitioning grid* that divides the data space into $2^{\lambda \cdot 2d}$ cells. In particular, the grid has 2^{λ} cells on each dimension, and λ is an integer called the *resolution*.

Objects whose duals fall in the same cell have identical Hilbert values, which are indexed by a B^+-tree. Each leaf entry stores the detailed information of an object (i.e., its reference time, locations, and velocities). An insertion/deletion is performed in the same way as a B^+-tree, by accessing $O(\log N)$ pages where N is the dataset cardinality.

As with the B^x-tree, a B^{dual}-tree is composed of two B^+-trees BT_1 and BT_2. Each tree has two states: (i) a *growing state* when objects can be inserted/deleted, and (ii) a *shrinking state* when only deletions are allowed. At any time, one tree is in the growing state, and the other in the shrinking state. They swap states every T timestamps, where T is the largest interval between two consecutive updates from the same object.

Initially, BT_1 (BT_2) is in the growing (shrinking) state for time interval $[0, T)$, when all the updates are directed to BT_1, and BT_2 remains empty. During $[T, 2T)$, the states

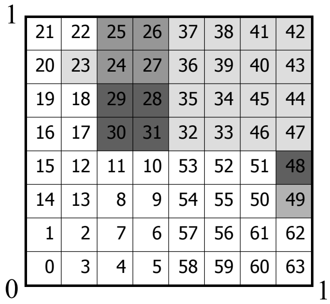


Fig. 4 Hilbert range decomposition ($d = 1$, $\lambda = 3$)

of BT_1 and BT_2 are reversed. In this period, every insertion is performed in BT_2. A deletion, however, may remove an object from BT_1 or BT_2, depending on whether it was inserted during $[0, T)$ or $[T, 2T)$, respectively. At time $2T$, BT_1 becomes empty (all the objects inserted during $[0, T)$ have issued updates), and the two trees switch states again.

Given a query, both BT_1 and BT_2 are searched, and the results are combined to produce the final answer. Since BT_1 and BT_2 are symmetric, in the rest of Section 4, we focus on a single tree, and refer to it simply as a B^{dual}-tree.

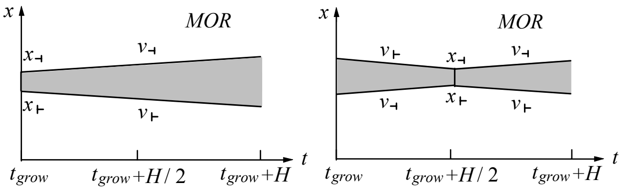
4.2 Deciding the Reference Time

The selection of T_{ref} has a significant impact on query performance. Without loss of generality, consider a B^{dual}-tree that enters the growing state at time t_{grow}. Although previous methods [1,13,8] set T_{ref} to t_{grow}, we will show that a better choice of T_{ref} is $t_{grow} + H/2$, where H is the horizon parameter.

Any cell c in the partitioning grid can be regarded as a d-dimensional MOR (moving rectangle) whose SBox (VBox) captures the projection of the cell on the location (velocity) dimensions of the dual space. Figure 4 shows an example where $d = 1$, and the dual space has $2d = 2$ dimensions. The partitioning grid contains $2^{3 \cdot 2} = 64$ cells (i.e., the resolution $\lambda = 3$), and the number in each cell is the Hilbert value (of any point inside). The cell 53, for example, has a 1D SBox [0.5, 0.625] (its projection on the horizontal axis) and a VBox [0.375, 0.5], assuming that all the dimensions have a domain [0, 1].

Given a range query q (an MOR), objects in a cell c need to be inspected if and only if the MOR c intersects q during the query interval qt. For example, assume T_{ref} = 0 and let c be the cell in Figure 4 with value 53. According to the SBox and VBox of c, the spatial extent of c at time 1 is $c(1)$ = [0.5 + 0.375, 0.625 + 0.5] = [0.875, 1.125]. For a query with q = [0.7, 0.8], $q.V$ = [0.1, 0.1], and qt = [0, 1], all the objects with Hilbert value 53 must be examined because $q(1)$ = [0.8, 0.9] intersects $c(1)$; otherwise, we may risk having false misses.

Hence, to maximize query efficiency, we should reduce the probability that the MOR of c intersects a query q. The analysis in [15] shows that the probability is decided by the "integrated area" of c during $[t_{grow}, t_{grow} + H]$, calculated



(a) $T_{ref} = t_{grow}$ (b) $T_{ref} = t_{grow} + H/2$

Fig. 5 Integrated areas with different t_{ref}

as:

$$\int_{t_{grow}}^{t_{grow}+H} AREA(c(t))\ dt \tag{2}$$

where $c(t)$ is the spatial extents of c at time t. Figures 5a and 5b illustrate the integrated areas (the grey regions) of a 1D MOR, assuming reference time $T_{ref} = t_{grow}$ and $T_{ref} = t_{grow} + H/2$, respectively. Let l (l_v) be the side length of a cell on a location (velocity) dimension. For general dimensionality d, the integrated area of c equals:

$$\int_0^{T_{ref}-t_{grow}} (l+t\cdot l_v)^d\ dt + \int_0^{t_{grow}+H-T_{ref}} (l+t\cdot l_v)^d\ dt \tag{3}$$

Lemma 1 *Equation 3 is minimized when T_{ref} equals $t_{grow}+H/2$.*

Proof Let us consider the difference between the integrated area achieved with $T_{ref} = t_{grow} + t'$ for any $t' \in [0, H/2)$ and that with $T_{ref} = t_{grow} + H/2$. The difference equals:

$$\begin{aligned}
&\int_0^{t'} (l + t \cdot l_v)^d\ dt + \int_0^{H-t'} (l + t \cdot l_v)^d\ dt \\
&-2\int_0^{H/2} (l + t \cdot l_v)^d dt \\
&= \int_{H/2}^{H-t'} (l + t \cdot l_v)^d\ dt - \int_{t'}^{H/2} (l + t \cdot l_v)^d\ dt \\
&\geq (\frac{H}{2} - t')(l + l_v \cdot \frac{H}{2})^d - (\frac{H}{2} - t')(l + l_v \cdot \frac{H}{2})^d \\
&= 0
\end{aligned}$$

which indicates that $T_{ref} = t_{grow} + H/2$ produces a smaller area than $T_{ref} = t_{grow} + t'$. By symmetry, we can show that the same is true for $T_{ref} = t_{grow} + H - t'$ (given any $t' \in [0, H/2)$), thus completing the proof. □

Rigorously, Formula 2 captures the access probability of a cell for a timestamp range query q with $qt_{\vdash}$ uniformly distributed in $[t_{grow}, t_{grow} + H]$. Hence, $T_{ref} = t_{grow} + H/2$ minimizes the query cost only in this scenario. The analysis for general queries is more complex. Nevertheless, the above analysis shows that t_{grow} is most likely not an appropriate value for T_{ref}. As a heuristic, we set T_{ref} to $t_{grow} + H/2$ in any case, which leads to lower query cost than $T_{ref} = t_{grow}$ in all of our experiments.

4.3 Pitfall of the 1D-Range Reduction

As reviewed in Section 2.3, the B^x-tree adopts the 1D-range reduction for solving range search. Specifically, the search is converted to several 1D range queries on the B^+-tree (e.g., in Figure 3, six queries are required for segments AB, CD, EF, GH, IJ, KL, respectively). In this section, we will show that this strategy is inefficient for B^{dual}-trees, and outline an alternative processing framework.

To apply the 1D-range reduction, we need to transform range search into a *simplex query* in the dual space. As mentioned in Section 2.1, a simplex query specifies a set of linear constraints (each corresponding to a $2d$-dimensional half-space in the dual space), and aims at finding the (dual) points that fall in the intersection of all the half-spaces. Although these constraints can be formulated using the derivation of [8], the intersection can be a very complex polyhedron, rendering it difficult to compute the smallest set of segments on the Hilbert curve constituting the polyhedron.

Furthermore, even if an algorithm was available to discover these segments, applying the 1D-range reduction to B^{dual}-trees would suffer from another problem: the number of required 1D range queries increases exponentially with the dimensionality d. This is a well-known drawback of the Hilbert curve, regardless of the shape of a search region. For simplicity, we explain the phenomenon using regular regions, which are hyper-squares covering b cells (of the partitioning grid) on each dimension (i.e., the square contains totally b^{2d} cells), where b is an integer smaller than 2^{λ}. Consider each cell c such that (i) it is at the border of the square, and (ii) the cell whose Hilbert value precedes that of c lies outside the square. The number of necessary 1D range queries (for finding all the points in the square) equals exactly the number of cells satisfying the conditions (i) and (ii). Unfortunately, there are on average b^{2d-1} such cells in the square [11]. Note that the value of b depends on the resolution λ. A typical value of λ is 10 (i.e., each dimension consists of 1024 cells), and thus, b is at the order of 100 for a search region covering 10% of each axis. In this case, issuing b^{2d-1} 1D range queries becomes prohibitively expensive.

Motivated by this, we devise an alternative query evaluation strategy for B^{dual}-trees. We avoid generating 1D range queries, and instead focus on developing algorithms for checking whether the subtree of an intermediate entry (in a B^{dual}-tree) may contain any object satisfying a spatiotemporal predicate. As we will see, this strategy allows us to re-use the existing algorithms of TPR-trees for query processing with B^{dual}-trees (applying simple modifications). Furthermore, as shown in Section 4.5, our approach supports kNN search much more efficiently than B^x-trees. In particular, we retrieve the nearest neighbors in a single traversal of the tree, as opposed to the iterative solution (with multiple range queries) for B^x-trees.

4.4 The MOR Representation of an Intermediate Entry

Next, we show that each intermediate entry e can be associated with a set of MORs $\{r_1, r_2, ... r_m\}$ with the following property: at any future time t, the location of any object o in the subtree of e is enclosed in the extents of some MOR. Formally, there exists at least one MOR r_i (for some $i \in [1, m]$) such that $r_i(t)$ covers $o(t)$. These MORs are crucial for the query algorithms presented in the next section.

In fact, as a property of the B^+-tree, an intermediate entry e is implicitly accompanied by an interval $[e.h_\vdash, e.h_\dashv)$, which contains the Hilbert values of all the objects in the subtree. We refer to $[e.h_\vdash,\ e.h_\dashv)$ as the *Hilbert interval* of e. Each integer in the interval corresponds to a cell in the partitioning grid. As mentioned in Section 4.2, each cell can be regarded as an MOR, and thus, e can be trivially associated with $e.h_\dashv - e.h_\vdash$ MORs. However, the number of these MORs can be $2^{\lambda \cdot 2d}$ in the worst case (i.e., all the cells in the grid), such that the resulting query algorithms incur expensive CPU cost. In the sequel, we present an approach that associates e with at most $(4^d - 1) \cdot (2\lambda - 1)$ MORs.

Our goal is to break $[e.h_\vdash, e.h_\dashv)$ into several disjoint intervals, such that the union of the cells in each interval is a hyper-square in the dual space. For example, in Figure 4, [23, 49] can be broken into 6 intervals [23, 23], [24, 27], [28, 31], [32, 47], [48, 48], [49, 49] satisfying the above condition. In particular, the cells in [23, 23], [24, 27], [32, 47] constitute 1×1, 2×2, and 4×4 squares, respectively. Each resulting square can be regarded as an MOR whose projection on a location/velocity dimension is identical to that of the square (e.g., [23, 49] can be associated with 6 MORs).

We say that an MOR is *perfect* if

- it is created by a $2d$-dimensional square of cells in the partitioning grid, and
- the cells have continuous Hilbert values.

In fact, a set of cells can produce a perfect MOR if and only if their Hilbert values constitute an interval of the form

$$[a \cdot 2^{i \cdot 2d},\ (a+1) \cdot 2^{i \cdot 2d} - 1] \quad (4)$$

where i is an integer in [0, λ], and a another integer in [0, $2^{2d \cdot (\lambda - i)} - 1$]. For instance, [32, 47] can be represented in the above form with a = 2 and i = 2, and hence, it leads to a perfect MOR.

Lemma 2 *An intermediate entry e can be associated with at most $(4^d - 1) \cdot (2\lambda - 1)$ perfect MORs.*

Proof Since all the MORs in this proof are perfect MORs, we omit the word "perfect" for simplicity. We say that an MOR is of *level* i if it is generated by an interval of Formula 4. Let $[x, y]$ be the Hilbert interval of e, and S the set of MORs that will be associated with e. Consider, among the MORs whose Hilbert intervals contain $[x, y]$, the one r of the minimum level l. The lemma is trivially correct if $[x, y]$ is identical to r (i.e., S has a single MOR r). Next, we focus on the case where $[x, y]$ is a proper subset of r, i.e., the length of $[x, y]$ is shorter than $2^{l \cdot 2d}$.

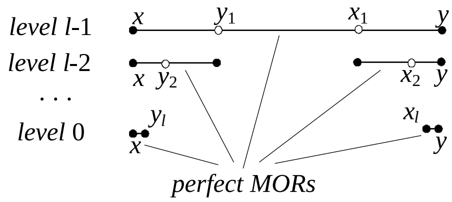


Fig. 6 Illustration of the proof for Lemma 2

We add to S all the MORs of level $l - 1$ whose Hilbert intervals fall in $[x, y]$. Since each interval has a length of $2^{(l-1) \cdot 2d}$, there can be at most $\lfloor (2^{l \cdot 2d} - 1)/2^{(l-1) \cdot 2d} \rfloor = 4^d - 1$ such MORs. If we remove these intervals from $[x, y]$, the remaining part of $[x, y]$ consists of at most two disjoint intervals $[x, y_1]$ and $[x_1, y]$, appearing at both ends of $[x, y]$ respectively, as shown in Figure 6. Each of the two remaining intervals is shorter than $2^{(l-1) \cdot 2d}$.

Let us recursively decompose $[x, y_1]$ into MORs of lower levels. If $[x, y_1]$ is already an MOR of level $l - 2$, no recursion is necessary. Otherwise, we add to S all the MORs of level $l - 2$ (whose Hilbert intervals are) fully enclosed in $[x, y_1]$. By the reason mentioned earlier, at most $4^d - 1$ such MORs are added. We remove their intervals from $[x, y_1]$, which has only one remaining part $[x, y_2]$ with length shorter than $2^{(l-2) \cdot 2d}$ (there is not any remaining interval of $[x, y_1]$ on the right end, because y_1 must be "aligned" with a perfect interval of level $l - 2$). The decomposing process can be repeated at most l times, such that eventually we obtain an interval $[x, y_l]$ of length at most $4^d - 1$ (see Figure 6), corresponding to at most $4^d - 1$ level-0 MORs (i.e., an MOR for each integer in $[x, y_l]$).

Since the same situation also applies to interval $[x_1, y]$, it is clear that, at each level i $(0 \le i \le l - 2)$, at most $2 \cdot (4^d - 1)$ MORs are created in S. At level $l-1$, as mentioned earlier, at most $4^d - 1$ MORs are obtained. Hence, the largest size of S equals $(4^d - 1) \cdot (2l - 1)$. Given that l is at most λ (the resolution of the grid), the size of S is bounded by $(4^d - 1) \cdot (2\lambda - 1)$. □

Figure 7 presents the algorithm for finding the perfect MORs of a non-leaf entry e in arbitrary dimensionality. Note that the actual number of MORs produced is usually much smaller than the upper bound stated in Lemma 2 (e.g., the number 6 for the interval [23, 49] in Figure 4 is significantly lower than the upper bound $(4^1 - 1) \cdot (2 \cdot 3 - 1) = 15$). The algorithm terminates in $O(4^d \cdot \lambda)$ time. Since d = 2 or 3 in most real applications, the computational cost is essentially linear to the resolution λ.

An important implication of Lemma 2 is that a B^{dual}-tree is as powerful as a TPR-tree in terms of the queries that can be supported. Intuitively, since the intermediate entries of both structures can be represented as MORs, an algorithm that applies to a TPR-tree can be adapted for the B^{dual}-tree. Adaptation is needed only because an entry of a TPR-tree has a single MOR, while that of a B^{dual}-tree corresponds to multiple ones. In the next section, we demonstrate this by developing the range and kNN algorithms.

```
Algorithm Decompose(a Hilbert Interval HI)
S:=∅; // S will contain the decomposed perfect MORs eventually
r0:= the MOR covering the entire dual space;
ω0:= the interval of the Hilbert domain;
L:={(r0, ω0)}; // L is a FIFO queue
while (L is not empty)
  remove the first element (r, ω) of L; //r is a perfect MOR
  if (ω intersects HI) then
    if (ω is covered by HI) then
      add r to S;
    else if (the length of ω > 1) then
      divide r into 4^d identical perfect MORs;
      for each resulting MOR r' and its Hilbert interval ω'
        add (r', ω') to L;
return S;
```

Fig. 7 Decomposing a Hilbert interval

4.5 Query Algorithms

Let e be an intermediate entry, which is associated with m MORs $r_1, r_2, ..., r_m$ ($m \leq (4^d-1)\cdot(2\lambda-1)$), returned by the algorithm of Figure 7. Given a range query q, the subtree of e is pruned if no r_i ($1 \leq i \leq m$) intersects q during the query interval qt. The processing algorithm starts by checking, for each root entry e, whether any of its associated MORs intersects q during qt (in the same way as in TPR-trees [15]). If yes, the algorithm accesses the child node of e, carrying out the process recursively until a leaf node is reached. Then, detailed information of each object encountered is simply examined against the query predicate.

We proceed to discuss nearest neighbor retrieval. For an MOR r, let $d_{min}(r, q)$ be the minimum distance between rectangle $r(t)$ and point $q(t)$ for all $t \in qt$. For an intermediate entry e associated with m MORs r_1, ..., r_m, we define the *minimum distance* $d_{min}(e, q)$ between e and q as:

$$d_{min}(e, q) = \min_{i=1}^{m} (d_{min}(r_i, q)) \tag{5}$$

Given an intermediate entry e and an object o in its subtree, we have $d_{min}(e, q) \leq d_{min}(o, q)$, where $d_{min}(o, q)$ is the smallest distance between o and q during qt. Similarly, if e' is an intermediate entry in the subtree of e, it holds that $d_{min}(e, q) \leq d_{min}(e', q)$. These properties permit us to deploy the "best-first" algorithm [5] for kNN search with a B^{dual}-tree. Specifically, the algorithm uses a heap H to organize all the (leaf/non-leaf) entries encountered in ascending order of their minimum distances to q. Initially, all the root entries are inserted to H. Then, the algorithm repeatedly processes the entry e that has the smallest minimum distance among the elements in H. Specifically, if e is an intermediate entry, we en-heap the entries in its child node; otherwise (e is an object), it is returned as the next NN. The algorithm terminates as soon as k objects have been reported.

It remains to clarify the computation of $d_{min}(e, q)$. By Equation 5, this is equivalent to calculating the minimum distance $d_{min}(r, q)$ between an MOR r and a moving point q, for which we are not aware of any existing solution[2]. In the sequel, we provide a method that obtains $d_{min}(r, q)$ in $O(d)$ time.

Let $d_{min}(t)$ be the minimum distance between $r(t)$ and $q(t)$ at a particular timestamp t. Denote $[r_{\vdash}[i](t), r_{\dashv}[i](t)]$ and $q[i](t)$ as the projection of $r(t)$ and $q(t)$ on the i-th dimension, respectively ($1 \leq i \leq d$). We use $d_{min}[i](t)$ to represent the minimum distance between the 1D interval $[r_{\vdash}[i](t), r_{\dashv}[i](t)]$ and value $q[i](t)$. Hence:

$$d^2_{min}(t) = \sum_{i=1}^{d} (d_{min}[i](t))^2 \tag{6}$$

To derive $d_{min}[i](t)$, we need to solve t from the following equations (each a linear equation of t):

$$r_{\vdash}[i](t) = q[i](t) \tag{7}$$

$$r_{\dashv}[i](t) = q[i](t) \tag{8}$$

Let $t_{\vdash}[i]$ be the solution of Equation 7, and $t_{\dashv}[i]$ that of Equation 8. We have:

$$d_{min}[i](t) = \begin{cases} r_{\vdash}[i](t) - q[i](t) & \text{if } (t_{\vdash}[i] < t_{\dashv}[i] \text{ and } t < t_{\vdash}[i]) \\ & \text{or } (t_{\vdash}[i] > t_{\dashv}[i] \text{ and } t > t_{\dashv}[i]) \\ 0 & \text{if } (t_{\vdash}[i] < t_{\dashv}[i] \text{ and } t \in [t_{\vdash}[i], t_{\dashv}[i]]) \\ & \text{or } (t_{\vdash}[i] > t_{\dashv}[i] \text{ and } t \in [t_{\dashv}[i], t_{\vdash}[i]]) \\ q[i](t) - r_{\dashv}[i](t) & \text{if } (t_{\vdash}[i] < t_{\dashv}[i] \text{ and } t > t_{\dashv}[i]) \\ & \text{or } (t_{\vdash}[i] > t_{\dashv}[i] \text{ and } t < t_{\vdash}[i]) \end{cases} \tag{9}$$

Therefore, $t_{\vdash}[i]$ and $t_{\dashv}[i]$ partition the time dimension into 3 disjoint pieces such that, when t falls in each piece, $d_{min}[i](t)$ can be represented as a piecewise linear function. Combining Equations 6-9, we can see that the time axis is divided into $6d$ segments by the $2d$ values $t_{\vdash}[1]$, $t_{\dashv}[1]$, ..., $t_{\vdash}[d]$, $t_{\dashv}[d]$ such that, when t is in each segment, $d^2_{min}(t)$ (in Equation 6) is a quadratic function of t (i.e., totally $6d$ quadratic functions).

Recall that our goal is to obtain $d_{min}(r, q)$, which equals the minimum of $d_{min}(t)$ for $t \in qt$. To find $d_{min}(r, q)$, it suffices to compute the minimum of each of the $6d$ quadratic functions, and $d^2_{min}(r, q)$ is the smallest of these $6d$ minimums. Solving the minimum of a quadratic function involves only trivial mathematical manipulation, and can be achieved in $O(1)$ time.

5 Theoretical Evidence about the Necessity of Capturing Velocities

In this section, we analyze theoretically why and when it is important to capture velocities in indexing moving objects with a space filling curve. For this purpose, we derive analytical formulae that mathematically reveal the behavior of the B^{dual}- and B^x-trees, subject to the following simplification and assumptions:

[2] The closest method was presented by Benetis et al. [2] for the problem of continuous NN search. Applying their derivation in our case, however, results in formulae that are much more complex than our results.

– **Simplification 1.** Although each (B^{dual}- or B^x-) tree consists of two B^+-trees, due to symmetry we discuss only one of them. In particular, we consider that the B^{dual}- and B^x-trees both contain a B^+-tree with the same reference time T_{ref}, and aim at comparing the query performance of the two B^+-trees. Without ambiguity, we still use the name B^{dual} and B^x to distinguish the two trees, respectively.
– **Simplification 2.** The Hilbert values for both indexes are computed using a partitioning grid with resolution λ. This is reasonable because λ = 10 is enough for ensuring that very few objects have the same Hilbert value.
– **Simplification 3.** Since the entries of B^{dual}- and B^x-trees have identical formats, the two trees on the same dataset have the same space consumption. We consider that the number N_L of leaf nodes in each B^+-tree equals $2^{i \cdot 2d}$, where i is an integer at most λ.
In practice, N_L is proportional to the dataset cardinality, and therefore, this simplification implies that the cardinality equals $N_L \cdot f$, where f is the node fanout (the average number of entries in a node). In practice, f is independent of N_L, and equals 69% of the node capacity (i.e., the largest number of objects in a leaf node).
– **Simplification 4.** We measure the query cost as the number of leaf nodes accessed (in practice the intermediate levels of a B^+-tree are usually memory-resident).
– **Simplification 5.** All the queries are timestamp queries. We use $qt_\vdash$ to denote the query timestamp (recall that $qt_\vdash$ is the starting time of a general query interval qt).
– **Assumption 1.** The duals of the objects are uniformly distributed in the dual space. Furthermore, the query distribution is also uniform; specifically, for range (kNN), the search region (query point) is randomly distributed in the data space.
– **Assumption 2.** As discussed in Section 4.4, each intermediate entry is accompanied by an interval of Hilbert values. Then, the interval for the parent entry of each leaf node covers $2^{(\lambda-i)\cdot 2d}$ values, i.e., $\frac{1}{N_L}$ of the Hilbert domain (because of the previous assumption).

The above assumptions are needed for obtaining rigorous equations that are not excessively complex, and but can capture the behavior of alternative structures. As we will see, our findings are highly intuitive, and are valid also in general scenarios (as demonstrated in the experiments). Section 5.1 first develops a cost model that quantifies the overhead of range search for the B^{dual}-tree. Then, Section 5.2 presents a similar model for the B^x-tree, and compares it with that of the B^{dual}-tree. In Section 5.3, we extend the analysis to kNN search.

5.1 The Range Search Cost of B^{dual}-Trees

Our derivation is based on the following lemma.

Lemma 3 *The parent entry of each leaf node is associated with a single perfect MOR (returned by the algorithm of Figure 4), whose projection on each (location or velocity) dimension covers $1/2^i$ of that dimension.*

Proof By Assumption 2, the parent entry of the first (leftmost) leaf node has a Hilbert interval [0, $2^{(\lambda-i)\cdot 2d}$), the entry of the second leaf has an interval [$2^{(\lambda-i)\cdot 2d}$, $2 \cdot 2^{(\lambda-i)\cdot 2d}$), and so on. In general, the entry of the j-th ($1 \le j \le N_L$) leaf has an interval

$$[(j-1)\cdot 2^{(\lambda-i)\cdot 2d}, j \cdot 2^{(\lambda-i)\cdot 2d} - 1]$$

Note that the above formula is consistent with Formula 4, by setting a to $j-1$, and replacing the i in Formula 4 with $\lambda - i$. Therefore, the intermediate entry is associated with a single perfect MOR, which contains $2^{(\lambda-i)\cdot 2d}$ cells in the partitioning grid. Since the MOR is a hyper-square, its edge on each dimension contains $(2^{(\lambda-i)\cdot 2d})^{\frac{1}{2d}} = 2^{\lambda-i}$ cells. Given that there are totally 2^λ cells on a dimension, the length of the edge accounts for $2^{\lambda-i}/2^\lambda = 1/2^i$ of a dimension. □

We illustrate the lemma using Figure 4 (where d = 1 and λ = 3). Suppose that N_L equals $2^{2\cdot 2} = 16$ (i.e., i = 2 in Lemma 3). As a result, the parent entry of the first leaf node has a Hilbert interval [0, 3], where the value 3 is obtained as $2^{(\lambda-i)\cdot 2d} - 1$. As in Figure 4, this interval covers 4 cells that form a square, whose side length is 1/4 of the corresponding dimension (as stated in Lemma 3). It is easy to verify that the same is true for all the leaf nodes.

Without loss of generality, assume that each location dimension has a unit length, and each velocity dimension has a length V. Let e be the parent entry of any leaf node, and r its associated perfect MOR. Denote L (L_V) as the projection length of r on a location (velocity) dimension. Lemma 3 states that:

$$L = 1/2^i$$
$$L_V = V/2^i$$

Recall that L describes the size of r at the reference time T_{ref} of the B^{dual}-tree. Hence, if $L(qt_\vdash)$ is the extent of $r(qt_\vdash)$ at the query timestamp $qt_\vdash$, we have:

$$L(qt_\vdash) = L + L_V \cdot (qt_\vdash - T_{ref}) \quad (10)$$

As discussed in Section 4.5, the child node of e needs to be accessed if and only if $r(qt_\vdash)$ intersects the query region q. We concentrate on the case that the region is a square with side length L_Q. By Assumption 1, q uniformly distributes in the data space, in which case the probability P_{acs} that $r(qt_\vdash)$ and q intersects equals (this is based on the well-known result [18] on the intersection probability of two random rectangles):

$$P_{acs} = (L(qt_\vdash) + L_Q)^d \quad (11)$$

where $L(qt_\vdash)$ is given in Equation 10. The subscript of P_{acs} indicates that P_{acs} is also the probability that the child node of e is visited in answering q. Thus, the expected query overhead IO^{dual}_{range} is computed as:

$$IO^{dual}_{range} = N_L \cdot P_{acs} \quad (12)$$

where N_L is the number of leaf nodes. Recall that N_L has two equivalent representations, i.e., it equals $2^{i\cdot 2d}$, or can be written as N/f, where N is the dataset cardinality, and f the node fanout. Solving i from the equation $2^{i\cdot 2d} = N/f$ leads to $2^i = (N/f)^{\frac{1}{2d}}$. Combining the above analysis, Equation 12 can be resolved into a closed formula:

$$IO^{dual}_{range} = \frac{N}{f}\cdot\left(\left(\frac{f}{N}\right)^{1/2d} + V\cdot\left(\frac{f}{N}\right)^{1/2d}\cdot(qt_{\vdash} - T_{ref}) + L_Q\right)^d \quad (13)$$

5.2 Comparison between B^{dual}- and B^x-trees

Before analyzing the performance of the B^x-tree, we first present an alternative strategy for processing range search, which is *never slower* than the original algorithm [6]. Recall that the B^x-tree adopts a Hilbert curve in the d-dimensional spatial space (excluding the velocity dimensions). The curve is also defined over a partitioning grid, where each cell c can still be regarded as an MOR. In particular, the SBox of the MOR is simply c, and its VBox covers each of d velocity dimensions *entirely*.

For any algorithm, objects whose Hilbert values are equal to those of c must be accessed, if and only if the MOR c intersects the search region q sometime in the query interval qt (in order to prevent false misses), based on the reasoning elaborated in Section 4.2. This observation implies that the range query algorithm of B^{dual}-trees can be applied to B^x-trees as well. Specifically, we associate each intermediate entry e of a B^x-tree with a set of MORs, and visit its child node only if any of the MORs intersects q during qt.

A result similar to Lemma 3 also holds for B^x-trees:

Lemma 4 *The parent entry of each leaf node in the B^x-tree is associated with a single MOR, whose projection on each location (velocity) dimension covers $1/2^{2i}$ of (completely) that dimension.*

Proof The reason why the MOR covers each velocity dimension completely has been mentioned earlier. To prove the lemma regarding the location projections, we need Assumption 2, i.e., the Hilbert interval of e includes $2^{(\lambda-i)\cdot 2d}$ values. Similar to the proof of Lemma 3, it is easy to show that the $2^{(\lambda-i)\cdot 2d}$ cells in the interval constitute a hypersquare, whose edge, therefore, contains $(2^{(\lambda-i)\cdot 2d})^{\frac{1}{d}}$ (note that the outmost exponent is not $\frac{1}{2d}$ because the Hilbert curve concerns only location dimensions), that is, $2^{2(\lambda-i)}$ cells. Given that there are $(2^{2\lambda d})^{1/d} = 2^{2\lambda}$ cells per dimension, the length of the edge accounts for $1/2^{2i}$ of the dimension. □

Following the notations in the previous section, we use L' (L'_V) for the projection length of the MOR of e, and according to the previous lemma:

$$L' = 1/2^{2i}$$
$$L'_V = V$$

where V is the length of a velocity dimension. Note that Equations 10-12 are still valid for B^x-trees (replacing L and L_V with L' and L'_V, respectively). Based on these results, we obtain the formula for the cost of B^x-trees:

$$IO^x_{range} = \frac{N}{f}\cdot\left(\left(\frac{f}{N}\right)^{1/d} + V\cdot(qt_{\vdash} - T_{ref}) + L_Q\right)^d \quad (14)$$

where the semantics of the variables are identical to those of Equation 13. Comparing the cost models of B^{dual}- and B^x-trees, we observe the following characteristics of the two structures:

- The query cost increases monotonically with $qt_{\vdash} - T_{ref}$, i.e., predicting farther into the future is more expensive.
- The resolution λ of the partitioning grid does not affect the query performance, as long as λ is sufficiently large. If λ is too small, numerous objects have the same Hilbert value, and they must be searched altogether even if only one of them may qualify the query.
- If $qt = T_{ref}$ (the query timestamp coincides with the reference time of the tree), a B^x-tree actually has better performance (the term $(f/N)^{1/d}$ in Equation 14 is smaller than $(f/N)^{1/2d}$ in Equation 13). In general, a B^x-tree better preserves objects' spatial locality, since the Hilbert curve of an B^{dual}-tree attempts to capture also the locality along the velocity dimensions. Processing a query with $qt = T_{ref}$ requires only objects' locations, in which case a B^x-tree incurs lower cost than a B^{dual}-tree.
- As $qt_{\vdash}$ increases, the efficiency of the B^x-tree deteriorates considerably faster than that of the B^{dual}-tree. When $qt_{\vdash}$ reaches a certain threshold t_Θ, the B^{dual}-tree starts outperforming its competitor, and the difference becomes larger as $qt_{\vdash}$ grows further. We can quantify t_Θ as the value of $qt_{\vdash}$ that makes IO^{dual}_{range} equivalent to IO^x_{range}, or specifically:

$$t_\Theta = T_{ref} + \frac{1}{V}\cdot\left(\frac{f}{N}\right)^{1/2d}$$

- If V (i.e., the length of a velocity dimension) is large, t_Θ is small, meaning that a B^{dual}-tree is better than a B^x-tree even if the query timestamp $qt_{\vdash}$ is very close to T_{ref}. This confirms the intuition that ignoring velocities is feasible in practice only if objects have similar motion parameters. In an application where objects can have drastically different speeds, the B^{dual}-tree is the more effective solution.

5.3 Discussion on kNN Search

Next, we will show that the previous observations for range search also hold for kNN retrieval, due to an inherent connection between the two query types. Given a kNN query q, let $dist$ be the distance between the k-th NN and point q at the query time $qt_{\vdash}$. Then, the cost of a B^{dual}-tree (in solving q) is identical to that of a range query with the same

$qt_{\vdash}$, whose search region is a circle centering at q with radius $dist$. This is a well-known property of the best-first algorithm [5]. In fact, the algorithm is optimal, meaning that any other method deploying the B^{dual}-tree to process a kNN query will incur at least the same overhead.

Using the technique (illustrated at the beginning of Section 5.2) of associating an intermediate entry in a B^x-tree with a set of MORs, the best-first algorithm can also be applied to this index for answering a kNN query optimally (in the sense as mentioned earlier). In fact, this algorithm (traversing the tree only once) is expected to significantly outperform that of [6] that requires numerous range queries. With this improvement, the kNN cost of a B^x-tree is also identical to the overhead of a range query, formulated in the same fashion as explained earlier for B^{dual}-trees. Hence, the relative behavior of B^{dual}- and B^x-trees is analogous to that for range search.

Algorithm **Aggregate-Range**(moving region q, query interval qt)
1. $rslt_{est}$:=0; //the estimate of the query result
2. initialize a max-heap H whose elements are of the form (entry, key)
3. **for each** entry e in the root
4. $rslt_{est}$:=$rslt_{est} + n_{qualify}(q, e)$;
5.   add $(e, n_{qualify}(q, e))$ to H;
6. **while** (H is not empty $\wedge$ the user has not terminated the query)
7.   remove the top element (e, est) of H;
8. $rslt_{est}$:=$rslt_{est} - est$;
9.   read the child node nd of e;
10.   **if** (nd is a non-leaf node) **then**
11.   **for each** entry $e' \in nd$
12.   **if** (e' intersects q during qt) **then**
13. $rslt_{est}$:=$rslt_{est} + n_{qualify}(q, e')$;
14.   add $(e', n_{qualify}(q, e'))$ to H;
15.   **else** // nd is a leaf node
16.   **for each** object $o \in nd$
17.   **if** (o satisfies q) **then**
18. $rslt_{est}$:=$rslt_{est} + 1$;

Fig. 8 Progressive aggregate range search algorithm

6 Progressive Query Processing

Conventional algorithms return exact results to a user at the moment they are produced. The query does not terminate until all the results have been generated. On the other hand, progressive algorithms return informative results to a user early, and progressively refine them. The user can terminate the query if the approximate results obtained so far are satisfactory. In this section, we study the progressive versions of range and kNN search. Such queries can be posed by the end users of a database, or by the query optimizer to estimate query selectivity efficiently.

6.1 Aggregate Range Search

An *aggregate range* query retrieves the number of objects that will appear in a moving rectangle q during a time interval qt. For instance, "find the number of aircrafts expected to be within 20 miles from flight UA80 in the next 10 minutes". The query can be processed as a conventional range query, followed by counting the number of qualifying aircrafts. Here, we propose a progressive algorithm that uses the Hilbert intervals of the intermediate entries in a B^{dual}-tree to progressively compute estimates for the result.

Figure 8 shows the pseudocode for the algorithm. Let $n_{qualify}(q, e)$ be the estimated number of qualifying objects in the subtree of entry e. First, the root is loaded, and the estimates for its entries are summed up to $rslt_{est}$, which is the first approximation of the query result (we will discuss how to derive $n_{qualify}(q, e)$ shortly). The root entries are added to a max-heap H. At each step, the entry e in H with the largest $n_{qualify}(q, e)$ is de-heaped, and its contribution in $rslt_{est}$ is replaced by the estimates for the entries in its child node (these entries are also inserted in H). The rationale is that by refining entries with large estimates early, we can reduce the estimation error as soon as possible.

The algorithm continuously improves $rslt_{est}$ to the actual result. At each step (Line 6), the user can discontinue the query processing if the current estimate is satisfactory (e.g., it has converged to a roughly constant value). Convergence can be automatically detected by analyzing the moving average of the last few results. If the moving average stabilizes, we can terminate, with confidence that $rslt_{est}$ will not change significantly afterwards. Such techniques for automatic termination could be particularly useful to a query optimizer for selectivity estimation.

It remains to clarify the computation of $n_{qualify}(q, e)$. Given the node fanout f, we estimate the number of objects in the subtree of e as $f^{level(e)}$, where $level(e)$ is the level of e. Recall that e is accompanied by a Hilbert interval $\mathcal{HI}(e)$, and it is associated with a set S of perfect MORs r (returned by the algorithm of Figure 7), each of which also corresponds to a Hilbert interval (in the form of Formula 4) $\mathcal{HI}(r)$. Consider the set of objects (underlying e) that are covered by an MOR r of S in the dual space. We estimate the cardinality of the set as $f^{level(e)} \cdot |\mathcal{HI}(r)|/|\mathcal{HI}(e)|$, where $|\mathcal{HI}(e)|$ and $|\mathcal{HI}(r)|$ are the lengths of the Hilbert intervals for e and r respectively. Let $\mathbf{Pr}(q, r)$ be the probability that an object in the set satisfies q (it can be calculated using the formula in [17] for predicting the range search selectivity on random moving objects). As a result, $n_{qualify}(q, e)$ can be computed as:

$$n_{qualify}(q, e) = \sum_{\forall r \in S} \left(f^{level(e)} \cdot \mathbf{Pr}(q, r) \cdot \frac{|\mathcal{HI}(r)|}{|\mathcal{HI}(e)|} \right)$$

6.2 kNN Distance Search

Given a moving point q and a time interval qt, a *kNN distance* query retrieves the distance of the k-th nearest object from q (where the distance is defined in Section 3). For instance, "what is the shortest distance between flight UA80 and any other aircraft in the next 10 minutes?". The progressive version of the query provides early estimates of the kNN distance, which are iteratively refined.

Figure 9 presents the details of the algorithm, which maintains an array W containing the estimates for the distances of the k NNs. Specifically, the i-th ($1 \leq i \leq k$) element of W has the form ($W[i].entry$, $W[i].dist$), where $W[i].entry$ is the i-th nearest object currently known or an intermediate entry whose subtree may contain such an object, and $W[i].dist$ equals the predicted i-th nearest distance. Initially, $W[i].entry = \varnothing$ and $W[i].dist = \infty$ for all elements $W[i]$.

The algorithm operates like the best-first NN algorithm [5], maintaining a min-heap H of the visited entries by their minimum distances from q (computed as discussed in Section 4.5). Before an entry e is de-heaped the user can terminate the algorithm, if s/he is satisfied with the current value of $W[k].dist$. Let e be the de-heaped entry. If the child node of e is a leaf, we compute the distance for all the objects o encountered. Otherwise (the child of e is a non-leaf node), for each entry e' in the node, we estimate the largest possible distance $d_{max}(e', q)$ between q and any object in the subtree of e'. Towards this, the algorithm obtains the set S of perfect MORs associated with e'. For each MOR $r \in S$, we compute its maximum distance from q as a function of t (by a set of equations similar those in Section 4.5), and then take the minimum value of this function during $t \in qt$. Thus, $dist$ is bounded by the largest of the minimums for all the MORs, or formally:

$$d_{max}(e', q) = \max_{r \in S} \left(\min_{t \in qt} d_{max}(q(t), r(t)) \right)$$

W is updated whenever we (i) find an object whose distance to q is smaller than $W[k].dist$, or (ii) can assert that such an object exists underneath an intermediate entry e (i.e., $d_{max}(e', q) < W[k].dist$). The subtree of e can be pruned pruned if $d_{min}(e, q)$ (Equation 5) is at least $W[k].dist$.

We close this section by pointing out that the above aggregate range and kNN distance algorithms also apply to the B^x-tree. However, the structure does not provide good result estimates until nearly the end of execution, since its intermediate entries do not incorporate object velocities.

7 Experiments

In this section, we experimentally compare the B^{dual}-tree against the best indexes of the 3 categories in Section 2: STRIPES[3] [13] (representing structures based on dual transformations), the TPR*-tree [16] (an enhanced version of the TPR-tree), and the B^x-tree [6]. All experiments were performed on a machine with a Pentium IV 2.3GHz CPU and 512 Mb of memory. The disk page size is fixed to 1K bytes. We use a relatively small page size to simulate realistic scenarios where the dataset cardinality is much higher. Unless otherwise stated, we do not use memory buffers for consecutive queries or updates. All reported I/O costs correspond to page accesses.

[3] For STRIPES, we store non-leaf nodes as tuples in a relation file and apply the "half-page" storage optimization of [13] for leaf nodes. Sibling half-page nodes are packed into the same physical disk page, in order to minimize I/Os during traversal.

```
Algorithm kNN-Distance(moving point q, query interval qt)
1.  initialize a min-heap H with elements of the form (entry, key);
2.  for each i ∈ [1, k]
3.    W[i].entry:=∅; W[i].dist:=∞;
4.  add (root, 0) to H;
5.  while (H is not empty ∧ the user has not terminated the query)
6.    remove the top (e, dist) element of H;
7.    if W[i].entry = e for any i then
8.      for j:=i to k − 1
9.        W[j].entry:=W[j + 1].entry;
          W[j].dist:=W[j + 1].dist;
10.     W[k].entry:=∅; W[k].dist:=∞;
11.   read the child node nd of e;
12.   if (nd is a non-leaf node) then
13.     for each entry e' ∈ nd
14.       S:=Decompose(the Hilbert interval of e');
15.       d_max(e', q):=max_{r∈S} (min_{t∈qt} d_max(q(t), r(t)));
16.       if (d_max(e', q) < W[k].dist ) then
17.         W[i]:=the element in W with the least W[i].dist larger
            than or equal to dist;
18.         for j:=k downto i + 1
            //update W with pair (e', d_max(e', q))
19.           W[j].entry = W[j − 1].entry;
              W[j].dist = W[j − 1].dist;
20.         W[i].entry:=e'; W[i].dist:=d_max(e', q);
21.       if (d_min(q, e') < W[k].dist) then
22.         add (e', d_min(q, e')) to H;
23.   else // nd is a leaf node
24.     for each object o ∈ nd
25.       if (d_min(q, o) < W[k].dist) then
26.         update W with pair (o, d_min(q, o)) in the same way as in
            Lines 18-20;
```

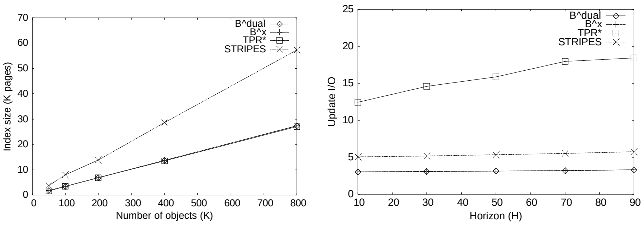
Fig. 9 Progressive kNN distance algorithm

7.1 Data and Query Generation

We generated spatiotemporal data following the methodology of [16,13,6]. The data space is two-dimensional, where each dimension has a domain of $[0, 1000]$. 5000 rectangles are sampled from a real 128K spatial dataset[4]; their centroids model positions of airports. Each object is an aircraft, which moves along the line segment connecting two airports. Initially, each aircraft is positioned at an arbitrary airport, and randomly selects another airport as the destination. At the subsequent timestamps, the aircraft will move from the source airport to the target one, at a speed that is generated in the range [0,5], following a Zipf distribution (skewed towards 0). As soon as the object reaches the destination, it chooses another airport as the next destination, at a new speed obtained in the same way as described earlier. At this moment, the aircraft updates its motion parameters in the underlying index, including a deletion (erasing the previous entry) followed by an insertion. In addition to these updates (caused by switching destinations), an aircraft also issues an update 25 timestamps (= T) after the previous one.

An index with time horizon $H = 2T = 50$ time units is created for each dataset. All objects are created and inserted into the index at time 0. At each update, exactly one object insertion and one deletion is performed. Queries are issued

[4] available at *http://www.rtreeportal.org/datasets/spatial/US/RR.zip*.



(a) Space vs. N (b) Update I/O vs. H

Fig. 10 Index sizes and effect of H on updates

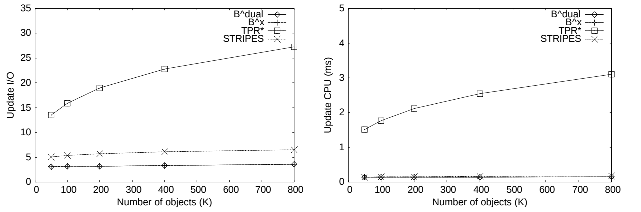
after $H/2$, when two B^{dual}-trees, B^x-trees, and STRIPES-quadtrees are used for indexing moving objects (as opposed to a single TPR*-tree). The resolution level λ for the Hilbert curves of B^{dual}-trees and B^x-trees is 10. We measure the query cost, by averaging it over a workload of 100 queries, issued at different current time t_{now} as the index runs. A moving (square) range query q is generated with the following parameters: the initial spatial extent $q.Slen$ (default 50); the velocity with extent $q.Vlen$ (default 6), centered at a random number in $[-5, 5]$, and the query time interval length $qtlen$ (default 15). $qt_{\vdash}$ is a random instant in $[t_{now}, t_{now} + 40 - qtlen]$. The initial spatial location of q follows the data distribution. For moving kNN queries, parameter $q.Slen$ is replaced by k (default 50).

7.2 Space requirements

Figure 10a shows the sizes of the different indexes, as a function of the number N of moving objects. B^{dual}-, B^x- and TPR*-trees have similar sizes. On the other hand, due to the low storage utilization of PR bucket quadtrees [14], STRIPES occupies much more space. Statistically, pages that store STRIPES leaf nodes are only 33% full on the average, whereas the node occupancy for other indexes is approximately 69%. As verified in subsequent experiments, the size difference affects negatively the query performance of STRIPES.

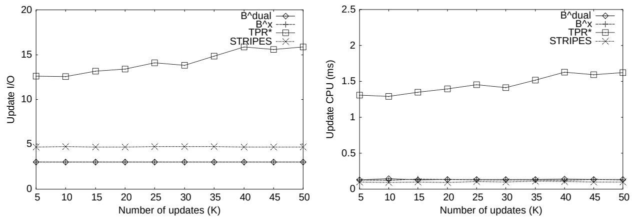
7.3 Update Performance

We compare the update performance of the indexes with respect to various factors. Figure 10b shows the average update cost as a function of time horizon H. As H increases, the overlapping of entries in TPR*-tree increases and more pages are accessed during updates. The other three indexes are not affected by H; only a single path is traversed during updates. Figure 11 shows the average update cost of the indexes as a function of the data size N. The update cost of the TPR*-tree is much higher than that of other indexes and increases with N. As the non-leaf entries of TPR*-tree overlap, multiple paths need be searched during an insertion and deletion. In addition, the tree performs expensive *active tightening* of the nodes during updates. The cost of the other three indexes increase only slightly with N; each update only searches a single path. The B^{dual}- and B^x-trees have the lowest cost as they are both based on the balanced B^+-tree. STRIPES has higher update cost because the PR bucket quadtree is unbalanced; on the average, a longer path, compared to the B^{dual}/B^x-tree, is traversed during updates.



(a) I/O cost (b) CPU cost

Fig. 11 Effect of data size N on the update cost



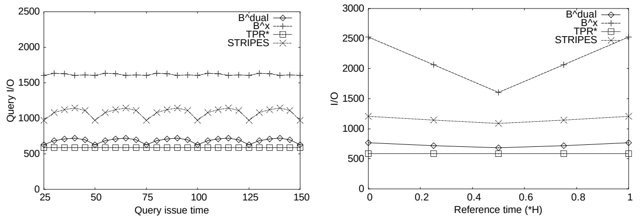
(a) I/O cost (b) CPU cost

Fig. 12 Update cost vs. number of updates

Finally, we study whether the update performance degrades over time. At timestamp 0, 100K objects are inserted to all indexes. We record the update performance of the indexes after every 5K updates. Figure 12 shows the update cost with respect to the number of updates. Observe that the update cost (both I/O and CPU) of the TPR*-tree increases slightly with time. The other indexes are not sensitive to the time of the updates. Summarizing, the high update cost of TPR*-tree makes it impractical for real-time applications. On the other hand, as we will see in the next experiments, TPR*-trees perform consistently better than other structures at queries, thus their query cost should be interpreted as a lower bound when compared with other methods (for update-intensive applications).

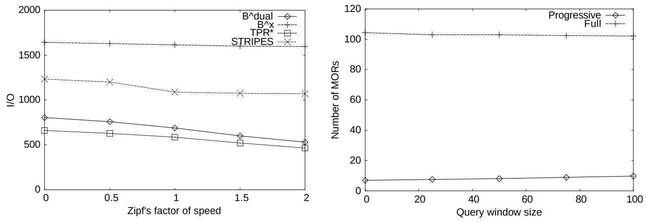
7.4 Query Performance

Figure 13a shows the performance of range queries with the default parameter values, issued at t_{now} as the index runs. The periodic behavior is caused by the use of two indexes. B^{dual} has good performance and it only has small fluctuation of query cost. Figure 13b shows the performance of the indexes on the default range query workload as a function of reference time T_{ref}. The reference time T_{ref} for the B^{dual}-, B^x-trees, and STRIPES can directly be set to values other than the index creation time t_{grow}. This technique is not directly applicable to the TPR*-tree. TPR*-tree performs active tightening of nodes during updates and its effect is equivalent to implicitly update the reference time of affected



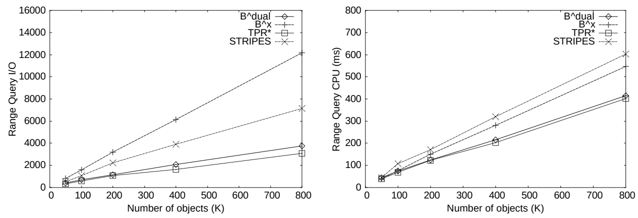
(a) I/O vs. t_{now} (b) I/O vs. T_{ref}

Fig. 13 Index degradation and effect on T_{ref}



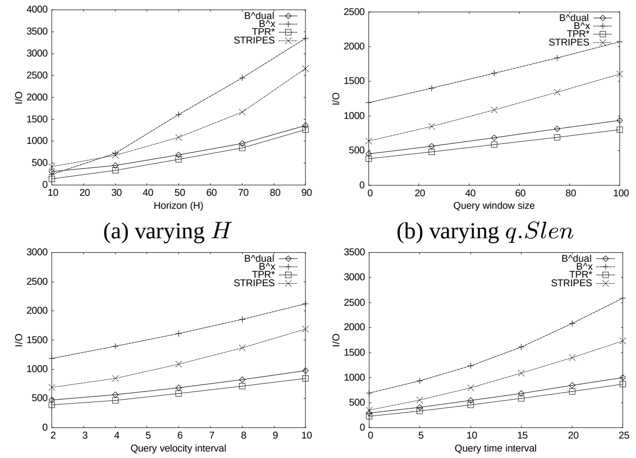
(a) I/O vs. Speed (b) Examined MORs

Fig. 14 Effect of objects' speed distribution / MORs examined per non-Leaf entry during searching the B^{dual}-tree



(a) I/O cost (b) CPU cost

Fig. 15 Effect of data size N



(a) varying H (b) varying $q.Slen$

(c) varying $q.Vlen$ (d) varying $qtlen$

Fig. 16 Range query I/O

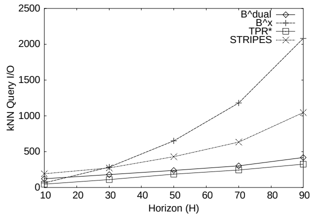
nodes to t_{now} (instead of t_{grow}). Observe that all indexes achieve better performance when T_{ref} is set to $t_{grow}+H/2$. The performance of B^x-tree is too sensitive to T_{ref} because the query region is enlarged by maximum velocities. As T_{ref} approaches $t_{grow}+H/2$, the average query enlargement is reduced. In all subsequent experiments, the value T_{ref} for all indexes (except the TPR*-tree) is set to $t_{grow}+H/2$.

Figure 14a shows the performance of the indexes on the default range query workload, by varying the skewness parameter θ of the Zipf distribution (for generating object velocity values) from 0 (uniform) to 2 (highly skewed). The performance of the B^{dual}-, TPR*- trees, and STRIPES improves because the extents of VBRs (i.e., velocity bounding rectangles) in leaf nodes become smaller when the number of fast moving objects decreases. However, the B^x-tree has little performance gain, since the small number of fast moving objects are still distributed in many different spatial regions, resulting in significant query enlargement. Figure 14b shows the average number of perfect MORs decomposed and examined from an entry, as a function of query window size $q.Slen$. *Full* denotes the standard decomposition method in Section 4.4. *Progressive* represents an optimized version which combines with query predicate checking in a single step, employs branch-and-bound technique for fast computation and returns as soon as a MOR of the entry is found to intersect the query. Consequently, for a B^{dual}-tree entry, only a small number of MORs is computed and examined on the average (6–9 as opposed to 102–106 MORs in the full decomposition). This justifies why searching the B^{dual}-tree is computationally efficient.

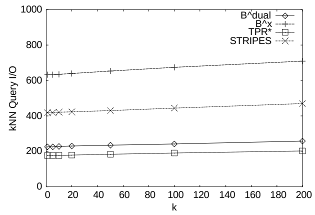
Figure 15 shows the performance of the indexes on the default query workload, on datasets with different number N of moving objects. The I/O cost increases linearly with N. The CPU costs have similar trends. Next, we study the effect of the time horizon H on the indexes. A query workload was generated for each value of H, such that $qtlen$ is proportional to H. Figure 16a shows the range query I/Os as a function of H. The B^{dual}- and TPR*- trees have similar costs. The B^x-tree degrades fast with H, since the average query enlargement increases with H. STRIPES is not cheap, due to its large size.

We also evaluate the performance of the indexes for different values of the query parameters. Figures 16b, 16c, and 16d show the I/O cost when we fix two of the parameters $q.Slen$, $q.Vlen$, and $qtlen$ to their default values and we vary the others. The query cost increases as any parameter value increases. The performance gaps between the indexes are almost insensitive to the parameter values. Note that the case for $qtlen = 0$ corresponds to the special case of timestamp queries.

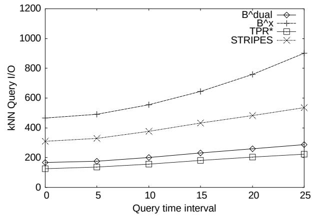
We then study the performance of the indexes for kNN queries with respect to various factors. The timestamp kNN algorithm for B^x-trees proposed in [6] was adapted for moving kNN queries (i.e. for $qtlen \geq 0$). Note that kNN search techniques for STRIPES were not mentioned in [13]. We apply the incremental NN search [5] with equations in Section 4.5 for computing the minimum distance of STRIPES entries from the query. Figure 17a shows the I/O cost of kNN queries as a function of the time horizon H of the indexes, by fixing $k = 50$. The result is consistent with that of range queries: both B^x-tree and STRIPES become much more ex-



(a) varying H

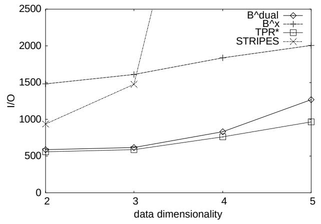


(b) varying k

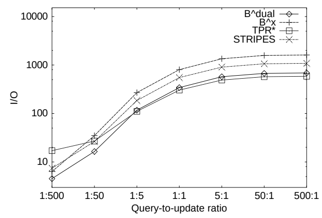


(c) varying $qtlen$

Fig. 17 kNN query I/O

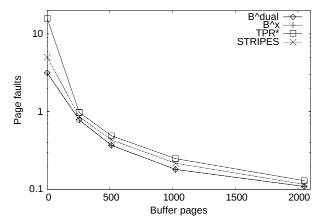


(a) I/O vs. d

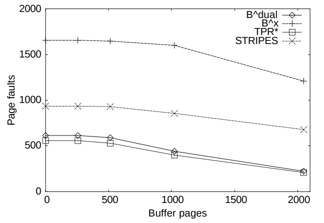


(b) I/O vs. query-to-update ratio

Fig. 18 Effect of d / mixed workloads



(a) Update I/O



(b) Query I/O

Fig. 19 Effect of a memory buffer

pensive than the B^{dual}-tree as H increases. Figure 17b compares the indexes for kNN queries as a function of k, by fixing $qtlen = 15$. The query costs do not change much when k is small compared to the data size N. Figure 17c shows the I/O cost of kNN queries as a function of $qtlen$, for $k = 50$. Note that the case for $qtlen = 0$ corresponds to the special case of timestamp queries. The performance differences are similar to those of range queries; the B^{dual}- and TPR*- trees have almost the same cost, however, the B^x-tree degrades fast as $qtlen$ increases. In general, the indexes show similar behavior in kNN queries as in range queries.

Next, we study the effect of data dimensionality on the query performance of the indexes. Figure 18a shows the query cost on the indexes as a function of the data dimensionality d, with default range queries. B^{dual}-trees and TPR*-trees have similar query costs while B^x-trees have much higher costs. The cost of STRIPES explodes as d increases because a quadtree node split may generate up to 4^d leaf nodes (in the worst case), dramatically reducing the disk utilization of the tree.

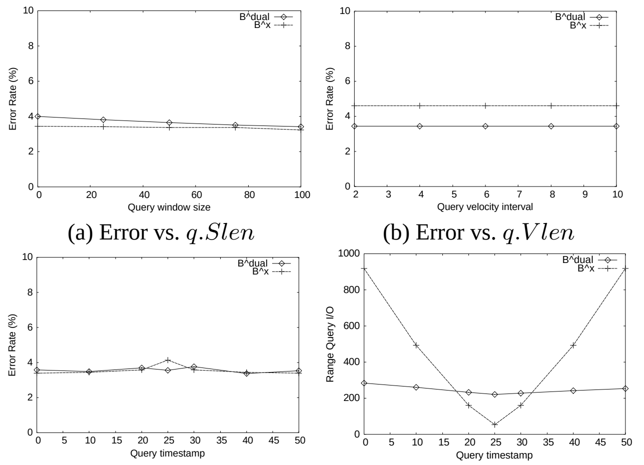
Figure 18b summarizes the performance of the indexes for mixed workloads (with updates and range queries of default parameter values), as a function of the query-to-update ratio. The figure plots the average I/O cost of a *single* operation (either update or query) in the workload. The TPR*-tree is the most expensive index for applications with high update rates. It starts outperforming the B^x-tree and STRIPES at a ratio of 1:50 and the B^{dual}-tree at a ratio of 1:5, since the cost of a typical query is much higher than that of a single update. Notice that the B^{dual}-tree is the best index for update-intensive applications, while it has similar query performance to the TPR*-tree.

So far, we measured update and query I/O without considering the existence of a memory buffer. Practical systems include buffers that reduce the I/O cost, by exploiting the common access patterns of consecutive queries or updates. We compare the performance of the indexes in the presence of an LRU memory buffer (Figure 19), for update and query workloads, using the default parameters and data. The buffer offers significant performance gain for updates (similar for all methods). During updates, more non-leaf pages than leaf pages are accessed, which have high chances to reside in the buffer at subsequent operations. Note that the B^+-tree indexes maintain their performance gain over STRIPES and the TPR*-tree for different buffer sizes. Regarding query performance, the effect of small or moderate buffers is negligible. The reason is that (i) the sequence of queries is random (i.e., queries do not exhibit locality, thus two consecutive queries only share a very small set of common leaf pages) and (ii) 94% of the accessed pages by a query are leaf pages; thus, even when the whole set of directory pages is pinned in the buffer, there is not large performance gain. In short, the buffer size does not affect the relative performance of the indexes.

7.5 Accuracy of the Cost Model

In this section, we test the accuracy of the cost models proposed in Section 5 for timestamp range queries. 100K moving points are inserted into a B^{dual}-tree and a B^x-tree at timestamp 0. We applied a workload of 100 timestamp queries and for each query we measured the actual act_i and estimated est_i cost of the indexes. The *error rate* [17] is then defined as $err = \sum_{i=1}^{100} |est_i - act_i| / \sum_{i=1}^{100} act_i$.

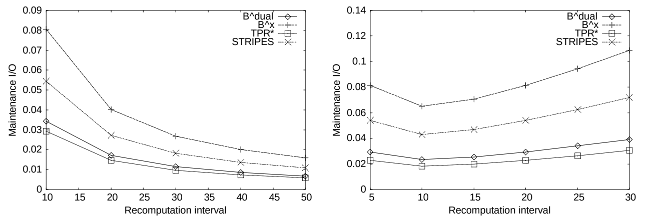
Figure 20a shows the error rate as a function of the query window size. A query refers to a random timestamp $qt_{\vdash}$ in $[t_{grow}, t_{grow} + H]$ ($H = 50$). Observe that the maximum error is only below 4%. The error estimates when varying other parameter values (i.e., query velocity interval $q.Vlen$ and query timestamp $qt_{\vdash}$) are similar and shown in Figures 20b,c. Figure 20d compares the costs of the two trees, as a function of the time instant $qt_{\vdash}$ of timestamp queries. In ac-



(a) Error vs. $q.Slen$ (b) Error vs. $q.Vlen$

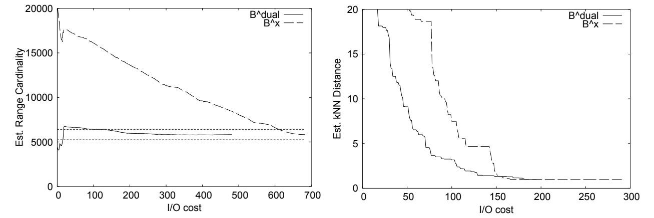
(c) Error vs. $qt_{\vdash}$ (d) Superiority of B^{dual}-trees

Fig. 20 Cost estimation error



(a) Range Query I/O (b) kNN Query I/O

Fig. 21 Continuous query I/O vs. recomputation interval l



(a) Aggregate range query (b) kNN distance query

Fig. 22 Progressive query estimates

cordance to our analysis, there is an instant t_{Θ} ($\approx T_{ref} + 7$ in Figure 20d), after which the B^{dual}-tree begins to outperform the B^x-tree. A symmetric observation holds for values of qt smaller than $T_{ref} - 7$. Figure 20d corresponds to the case which all objects are stored in the same tree, in order to verify our analysis. In practice, when the query timestamp is close to the T_{ref} of one tree, it is far from the T_{ref} of the other tree. The combined effect on two trees is that B^{dual} outperforms B^x for any timestamp.

7.6 Progressive and Continuous Queries

Next, we study the performance of the indexes on continuous queries, by following the evaluation approach in [6]. The objective is to maintain the query result at any time instant. After a query is issued, the query is invoked periodically every l timestamps (the recomputation interval). Each invocation retrieves the set of results for the next l timestamps. When object updates arrive, the set of results is maintained continuously. Deletion of objects from the result set (e.g. for a kNN query) may invalidate the result and the query needs to be recomputed for the next l timestamps. Figure 21a shows the amortized maintenance cost of a range query (with default parameters) per update. As l increases, the query is re-invoked less frequently and the maintenance cost is reduced. Figure 21b shows the amortized maintenance cost of a kNN query (with default parameters) per update, as a function of l. Maintenance costs of all indexes first decrease and then increase. When l becomes too large, a large set of results need to be maintained and the probability of removing an object from the result set increases. This invalidates the result set and forces the query to be recomputed frequently. We note that specialized methods for monitoring continuous (range [10] and NN [12]) queries are preferred over general-purpose spatiotemporal indexes.

In the last set of experiments, we demonstrate the effectiveness of progressive queries on B^{dual}-trees in estimating the result early. We also implemented versions of the progressive algorithms for the B^x-tree. Figure 22a shows the estimated result of an aggregate range query with the standard parameter values $(q.Slen = 50, q.Vlen = 6, qtlen = 15)$ on the default dataset, as a function of the number of nodes read. The horizontal dotted envelope represents a 10% error bound from the actual result (5833). Observe that the B^{dual}-tree converges much faster to a good estimate of the query result, as opposed to the B^x-tree which does not reach the envelope, even at the time needed by the B^{dual}-tree to compute the exact result.

Figure 22b shows the estimated kNN distance as a function of the number of pages accessed for a query with $k = 50, qtlen = 15$ on the default dataset. The B^{dual}-tree progressively refines the estimated distance, however, it reaches an estimate with small relative error slower compared to aggregate range queries. Note that the kNN distance is very small, thus progressive algorithms are prone to relatively larger estimates. Summarizing, aggregate range queries using the B^{dual}-tree can return informative results to the users early.

8 Conclusions

We proposed the B^{dual}-tree, a new spatiotemporal index for predictive search that combines features and advantages of state-of-the-art methods; dual space indexing (STRIPES[13]), fast query processing (TPR*-tree [16]), and fast updates (B^x-tree [6]). We provided an analytical study, which justifies the superiority of B^{dual}-tree compared to the B^x-tree and a thorough experimental evaluation, which shows that our method has the best overall (query and update) performance compared to all three past indexes. Finally, we proposed progressive versions of aggregate (range and kNN) query algorithms that use the B^{dual}**-**tree to predict early an accurate estimate of the result, which is gradually refined.

Acknowledgement

The authors would like to thank Jignesh Patel for his useful feedback on the implementation details of STRIPES. This work was supported by grants HKU 7142/04E and CityU 1163/04E from Hong Kong RGC.

References

1. Agarwal, P.K., Arge, L., Erickson, J.: Indexing Moving Points. In: PODS, pp. 175–186 (2000)
2. Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S.: Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In: IDEAS, pp. 44–53 (2002)
3. Butz, A.R.: Alternative Algorithm for Hilbert's Space-Filling Curve. IEEE Trans. Comput. **C-20**(4), 424–426 (1971)
4. Guttman, A.: R-trees: A Dynamic Index Structure for Spatial Searching. In: SIGMOD, pp. 47–57 (1984)
5. Hjaltason, G.R., Samet, H.: Distance Browsing in Spatial Databases. TODS **24**(2), 265–318 (1999)
6. Jensen, C.S., Lin, D., Ooi, B.C.: Query and Update Efficient B^+-Tree Based Indexing of Moving Objects. In: VLDB, pp. 768–779 (2004)
7. Kollios, G., Gunopulos, D., Tsotras, V.J.: On Indexing Mobile Objects. In: PODS, pp. 261–272 (1999)
8. Kollios, G., Papadopoulos, D., Gunopulos, D., Tsotras, V.J.: Indexing Mobile Objects Using Dual Transformations. VLDB J. **14**(2), 238–256 (2005)
9. Lin, D., Jensen, C.S., Saltenis, S., Ooi, B.C.: Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects. In: MDM (2005)
10. Mokbel, M.F., Xiong, X., Aref, W.G.: SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In: SIGMOD (2004)
11. Moon, B., Jagadish, H.V., Faloutsos, C., Saltz, J.H.: Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. TKDE **13**(1), 124–141 (2001)
12. Mouratidis, K., Papadias, D., Hadjieleftheriou, M.: Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In: SIGMOD (2005)
13. Patel, J.M., Chen, Y., Chakka, V.P.: STRIPES: An Efficient Index for Predicted Trajectories. In: SIGMOD, pp. 637–646 (2004)
14. Pemmaraju, S.V., Shaffer, C.A.: Analysis of the Worst Case Space Complexity of a PR Quadtree. Inf. Process. Lett. **49**(5), 263–267 (1994)
15. Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the Positions of Continuously Moving Objects. In: SIGMOD, pp. 331–342 (2000)
16. Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In: VLDB, pp. 790–801 (2003)
17. Tao, Y., Sun, J., Papadias, D.: Analysis of Predictive Spatio-temporal Queries. TODS **28**(4), 295–336 (2003)
18. Theodoridis, Y., Sellis, T.K.: A Model for the Prediction of R-tree Performance. In: PODS, pp. 161–171 (1996)
19. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based Spatial Queries. In: SIGMOD, pp. 443–454 (2003)