RESEARCH ARTICLE

Stronger Compact Representations of Object Trajectories

Adrián Gómez-Brandón[a], Gonzalo Navarro[b], José R. Paramá[a], Nieves R. Brisaboa[a], Travis Gagie[c]

[a] Universidade da Coruña, CITIC, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain; [b] Millennium Institute for Foundational Research on Data, Department of Computer Science, University of Chile, Beauchef, Santiago 8370448, Chile; [c] Faculty of Computer Science, Dalhousie University, University Avenue, Halifax B3H 1W9, Canada

ARTICLE HISTORY
Compiled August 14, 2023

ABSTRACT

GraCT and ContaCT were the first compressed data structures to represent object trajectories, demonstrating that it was possible to use orders of magnitude less space than classical indexes while staying competitive in query times. In this paper we considerably enhance their space, query capabilities, and time performance with three contributions. (1) We design and evaluate algorithms for more sophisticated nearest neighbour queries, finding the trajectories closest to a given trajectory or to a given point during a time interval. (2) We modify the data structure used to sample the spatial positions of the objects along time. This improves the performance on the classic spatio-temporal and the nearest neighbour queries, by orders of magnitude in some cases. (3) We introduce RelaCT, a tradeoff between the faster and larger ContaCT and the smaller and slower GraCT, offering a new relevant space-time tradeoff for large repetitive datasets of trajectories.

KEYWORDS
Moving objects, Mobile computing, Trajectories representation, Compact data structures, Nearest neighbour algorithms.

1. Introduction

During the last decade, the number of GPS devices has sharply increased due to their popularization on different objects: cars, ships, smartphones, smartwatches, etc. Consequently, a large amount of data about the route followed by objects along time (*trajectory*) are collected. That information is very useful in applications like traffic management, analysis of human movement, tracking animal behaviour, security and surveillance, military logistics and combat, and emergency-response planning (Gudmundsson, Laube, and Wolle 2008). However, storing and processing that enormous amount of data is a challenge that requires the development of new time- and space-efficient data structures and indexes (Zheng and Zhou 2011).

Various proposals to represent trajectories exist, but all of them can be roughly classified into two groups depending on the type of movements that the objects can perform. In the first group, the movements of the objects are constrained by a network.

. CONTACT Adrián Gómez-Brandón. Email: adrian.gbrandon@udc.es

The second group, instead, allows objects move freely in a space with no restrictions. This paper belongs to the second group.

The most basic query supported by applications dealing with moving objects is to retrieve the *trajectory* of an object during a period of time or at a specific time instant. However, most of those applications need more sophisticated queries, like *spatio-temporal* queries, which identify the objects that are within a spatial region during a period of time, *nearest neighbour* queries, which return the objects that are closest to a given point or trajectory during a time interval, and even more sophisticated queries related to mining and clustering trajectories (Gudmundsson, Kreveld, and Speckmann 2004; Alamri, Taniar, and Safar 2013; Lee, Han, and Whang 2007; Cao, Mamoulis, and Cheung 2005).

Various disk-based data structures were proposed to store and index trajectories since the 1990s. In recent years, the sizes of the main memories have increased, and the gaps in time performance along the memory hierarchy have widened. As a consequence, in-memory indexes have become more popular in several areas, both for centralized and distributed deployments. In particular, different in-memory indexes for representing trajectories were proposed (Cudre-Mauroux, Wu, and Madden 2010; Zheng et al. 2018). In parallel, the field of *compact data structures* (Navarro 2016) has emerged as a technique to operate larger datasets in main memory, or to use fewer nodes in distributed in-memory deployments. Compact data structures compress the data in such a way that queries can be run directly on the compressed data. This type of compression not only saves space, but it also expands the scenarios where a fast in-memory solution is affordable.

GraCT and ContaCT (Brisaboa et al. 2019; Brisaboa et al. 2021) are two recent in-memory indexes for moving object trajectories that build on compact data structures. They were shown to require orders of magnitude less space than classic solutions while offering competitive time performance. On large datasets they were still able to run in main memory, whereas other indexes needed to run on disk, where they were orders of magnitude slower.

Both GraCT and ContaCT have two components: spatial indexes called *snapshots* locating the objects at regular time intervals, and *logs* encoding the movements of the objects between snapshots. The structures use different techniques to compress the logs. GraCT uses *grammar compression*, which represents the trajectories with a context-free grammar that exploits their similarities. Therefore, GraCT obtains better compression when there are many similar trajectories. ContaCT, instead, is based on *delta compression*, which encodes shorter movements with fewer bits. The specific encoding used can compute in constant time the position of an object at any desired time instant. Therefore, ContaCT tends to obtain better performance than GraCT in several queries, at the price of worse compression.

GraCT and ContaCT support extracting trajectories, spatio-temporal queries, and a restricted form of nearest neighbour queries: they report the objects that are closest to a spatial point at a given instant of time.

1.1. *Contributions*

In this paper we present contributions along three lines.

More sophisticated queries. We design and evaluate new algorithms for GraCT and ContaCT to solve more complex queries related to data mining. Those queries

require obtaining the objects that are closest to a spatial point during a period of time (*KNN during an interval*) (Gao et al. 2007; Frentzos et al. 2007) or searching for the trajectories that are closest to a given trajectory (*KNN of trajectories*) (Tang et al. 2011). KNN queries have attracted considerable attention from the research community. They can be classified into three main types (Güting, Behr, and Xu 2010): i) the query and the data objects are static points, ii) the query is a trajectory and the data objects are static points or the query is a static point and the data objects are trajectories, and iii) the query and the data objects are trajectories. The original GraCT and ContaCT only handled a KNN query of the first type, restricting the query to a single time instant.

The second and third types open the possibility of new queries like (type ii) *find the two closest trajectories of animals to a given static point (e.g., a food source) in the time interval* $[t_b, t_e]$ (Gao et al. 2007), or *observe the closest ambulances to the site of an accident* (Güting, Behr, and Xu 2010) and (type iii) *find the two animal trajectories nearest to a predefined one during the time period* $[t_b, t_e]$ (Gao et al. 2007), or *which vehicles accompanied president Obama on his trip through Berlin* (Güting, Behr, and Xu 2010). These new KNN queries are also the basis to solve data-mining queries like *moving patterns together* (Gudmundsson, Kreveld, and Speckmann 2004; Alamri, Taniar, and Safar 2013) and *trajectory clustering* (Lee, Han, and Whang 2007).

In this paper we extend GraCT and ContaCT to handle KNN queries of type ii and iii. Though both queries are an order of magnitude slower than the basic nearest neighbour query, which is not surprising, they still run in a few milliseconds. The ability of ContaCT to compute minimum bounding rectangles of trajectories in constant time makes it 2–4 times faster than GraCT on these complex queries.

Better data representations. The original GraCT and ContaCT variants combined an existing geometric representation for the snapshots with a representation for the logs. The snapshot representation, based on quadtrees, was not the ideal one to support the types of queries we needed to handle. In this paper we design a snapshot representation that is not only more space-efficient, but also better suited to the queries we need to run on GraCT and ContaCT.

Concretely, we introduce a new snapshot representation based on R-trees, which speeds up those queries by orders of magnitude in some cases. That improvement is most noticeable in spatio-temporal queries, which become up to 200 times faster, and in nearest neighbour queries, which improve by a factor of 2–10. As a consequence, the new snapshots make GraCT and ContaCT faster than the MVR-tree (Tao and Papadias 2001), a classical spatio-temporal index, both on spatio-temporal and nearest neighbour queries.

New space/time tradeoffs. We propose a new data structure, called RelaCT, that combines the strong points of ContaCT and GraCT on highly repetitive sets of trajectories.

GraCT compresses more because it is based on Re-Pair, a grammar compressor (Kieffer and Yang 2000) that exploits both repetitions in the sequence and high-order frequency bias in the symbols. ContaCT, instead, is based on simple delta-compression that takes advantage only of the differences between one object position and the next one. The other side of the coin is that GraCT usually has to run a sequential decompression involving several symbols to obtain the position of an object at a given time instant, whereas ContaCT is able to obtain that position in constant

time.

RelaCT is based on Relative Lempel-Ziv (Kuruppu, Puglisi, and Zobel 2010), a technique designed to compress highly repetitive collections, such as the genome of several individuals of the same species, while retaining nearly constant-time access. As a result, RelaCT brings a new tradeoff, exploiting repetitiveness in the sequences while staying close to ContaCT in speed.

More in detail, the new structure chooses some *reference* trajectories and encodes those using ContaCT, while the others are encoded *relatively* to the references, that is, indicating what to change in a reference to obtain each other trajectory. The structure, called *RelaCT* for *Rela*tive *C*ompression of *T*rajectories, exploits the similarity of trajectories to obtain compression, while keeping the ability of ContaCT to efficiently access the trajectories, faster than GraCT. RelaCT obtains relevant space-time tradeoffs. For example, in one of our datasets, *KNN of trajectories* using RelaCT is twice as fast as GraCT, wasting just 5% more space, and it is 1.5 times slower than ContaCT, which uses twice the space.

1.2. *Outline of the paper*

Section 2 presents the state of the art in representing, compressing, and indexing trajectories. Section 3 introduces several background knowledge that will be used later, including several data structures, as well as GraCT and ContaCT. Section 4 shows the queries supported by GraCT and ContaCT before this work. Section 5 presents the new KNN queries for GraCT and ContaCT, including an experimental study of their performance. Section 6 introduces the second contribution of this work, the snapshots based on the R-tree, also including an experimental study. Section 7 presents RelaCT, the third contribution of this work, again with its experimental study. Finally, Section 8 shows our conclusions and future work.

2. State of the art

Different structures for representing moving objects and their trajectories were designed in the last decades. In this section, we present the most relevant strategies for modelling, compressing, and indexing free trajectories.

2.1. *Modelling trajectories*

Trajectories can be modelled as continuous space-time functions. Since objects emit their location at discrete time instants, trajectories are digitized as a list of timestamped positions. As the frequency of the timestamps increases, the accuracy of the trajectory improves, but more space is required to represent it, which impacts on the costs of transmission, storage, and processing. Various *trajectory simplification* techniques (Douglas and Peuker 1973; Meratnia and By 2004; Trajcevski et al. 2006; Potamias, Patroumpas, and Sellis 2006; Muckell et al. 2011; Lin et al. 2017; Liu et al. 2015) aim to discard less relevant timestamped positions in order to reduce those costs. In this paper we stick to the simplest method (Potamias, Patroumpas, and Sellis 2006), which collects the positions at regular time intervals.

2.2. *Compressing trajectories*

The best-known method to reduce the amount of space needed to store trajectories, simplified or not, is *delta compression*. This method stores the first position of the trajectory and then stores the difference between each new position and the previous one. That is, it stores the first position and a sequence of *movements*. Delta compression exploits the fact that (i) consecutive positions are generally close to each other, and (ii) smaller numbers can be stored using fewer bits. A complete trajectory is efficiently extracted by adding each new difference to the previous (already computed) position. Instead, obtaining the position of an object at a specific time instant t requires computing all the positions preceding t. Some methods sample the positions at regular timestamps, introducing a space-time trade-off to compute the position at any time t.

Several systems use delta compression, including TrajStore (Cudre-Mauroux, Wu, and Madden 2010) and SharkDB (Zheng et al. 2018). Trajic (Nibali and He 2015) uses delta compression but encodes each point as the difference between a predicted point and the real one. A different technique is used in REST (Zhao et al. 2018).

2.3. *Indexing trajectories*

The traditional spatio-temporal indexes for trajectories are based on the R-tree (Guttman 1984). The 3DR-tree (Vazirgiannis, Theodoridis, and Sellis 1998) replaces the MBRs (Minimum Bounding Rectangles) of the R-tree with MBBs (Minimum Bounding Boxes), where the third dimension represents the time. Since the MBB can cover a long period of time, the MBB can be too large and this may spoil the search performance. An attempt to avoid this problem (Pfoser, Jensen, and Theodoridis 2000) introduces two new indexes: STR-tree, which modifies the procedure that builds the MBBs, and TB-tree, which splits the trajectories into portions to produce smaller MBBs. Other indexes, like the HR-tree (Nascimento and Silva 1998) and MVR-tree (Tao and Papadias 2001), conceptually store an R-tree for each timestamp. Those R-trees are called *versions* and, to save space, several versions can share nodes.

Grid-based indexes split the space into cells and build a temporal index for each cell. SETI (Chakka, Everspaugh, and Patel 2003), for example, indexes the trajectories of each cell by time with an R*-tree.

A different approach is followed by the SEST-Index (Gutiérrez et al. 2005; Worboys 2005), which uses two components: *snapshots* and *logs*. The snapshots are spatial indexes that record the positions of the objects at regular timestamps. The log stores "changes" (e.g., objects that appear or disappear at some position) between consecutive pairs of snapshots.

2.4. *Combining compression and indexing*

A few methods combine compression and indexing in a single structure. TrajStore (Cudre-Mauroux, Wu, and Madden 2010) divides each trajectory into subtrajectories, each of which is stored in a cell whose size depends on the data distribution. Each cell contains a temporal and a spatial index (a quadtree) with all the subtrajectories falling in the cell. TrajStore is a lossy method, however, because each cell clusters its subtrajectories by similarity and only stores a representative of each cluster.

SharkDB (Zheng et al. 2018) combines delta compression and indexing. The time dimension is split into fixed-length intervals. SharkDB stores one point for each trajectory and interval of time. Those points that belong to the same interval are stored as

a column of a column-oriented database. The columns of SharkDB are encoded with delta compression.

GraCT (Brisaboa et al. 2019) and ContaCT (Brisaboa et al. 2021) use the same architecture of the SEST-Index (Gutiérrez et al. 2005; Worboys 2005), *logs* and *snapshots*, but they compress the trajectories and support a larger variety of queries. GraCT and ContaCT represent the space as a tessellation of equal-sized squares (cells), and assume that every object emits its position at regular time instants. The snapshots are compact spatial indexes (k^2-trees (Brisaboa, Ladra, and Navarro 2014), a quadtree variant) that store the location of all objects at regular timestamps. The logs store the movements of objects between snapshots. The main difference between GraCT and ContaCT is the method to compress those logs. GraCT compresses them with *grammar compression* (Kieffer and Yang 2000), whereas *delta compression* is applied in ContaCT. Instead of storing the consecutive movements, however, ContaCT represents those differences using bitmaps and other compact data structures. This enables ContaCT to compute several kinds of queries in constant time, outperforming GraCT. The grammar compression of GraCT, on the other hand, exploits the repetitiveness of movements of objects (e.g., ships tend to follow similar paths), whereas the delta compression of ContaCT only exploits spatial locality. On repetitive trajectories, then, GraCT obtains better compression than ContaCT.

3. Background

This section presents different general concepts that are needed to understand our contributions, and how different compact data structures are combined for compressing and indexing spatial information.

3.1. *Operations over bitmaps*

A bitmap or bitvector is an array whose elements are valued 0 or 1. There are two widely used operation over bitmaps: $rank_b(B, p)$ computes the number of times bit b appears in bitmap B until position p, and $select_b(B, i)$ returns the position of the i-th bit b in bitmap B. Those operations can be computed in $O(1)$ time by adding an additional structure of $o(n)$ bits to the n bits used by the bitmap $B[1..n]$ (Munro 1996). A related operation, $select_next_b(B, p)$, returns the position of the next bit b after position p in B. Although it can be solved in $O(1)$ time using $select_next_b(B, p) = select_b(B, rank_b(B, p) + 1)$, a direct implementation of *select_next* is as fast as *rank* in practice (Navarro 2016).

When B is sparse, that is, when the number of 1-bits m is much smaller than the total number of bits of the bitmap, an alternative representation based on Elias-Fano encoding (Okanohara and Sadakane 2007) uses only $m \log(n/m) + 2m$ bits in total, and answers *rank* queries in time $O(\log(n/m))$ and *select* in $O(1)$ time.

3.2. *Relative Lempel-Ziv*

Relative Lempel-Ziv (Kuruppu, Puglisi, and Zobel 2010) (RLZ) is a dictionary-based technique from the Lempel-Ziv family (Ziv and Lempel 1977, 1978), which compresses one sequence with respect to another sequence called the *reference*. Let R be the reference and S be an input sequence. RLZ compresses S by using a Lempel-Ziv parse,

where R plays the role of the dictionary. That is, S is represented as a sequence of z phrases $S = w_1 w_2 \dots w_z$, where every w_i is the longest substring of R that is a prefix of $w_i \dots w_z$. Each phrase w_i is encoded with a pair of values: a position in R where it occurs, and its length $|w_i|$. For example, with $S = abracadabra$ and $R = dabrac$, S is represented with three phrases, $S = w_1 w_2 w_3$, where $w_1 = abrac$ (occuring at $R[2..6]$), $w_2 = a$ (at, say, $R[2..2]$), and $w_3 = dabra$ (at $R[1..5]$). The RLZ representation of S with reference R is then $(2, 5), (2, 1), (1, 5)$.

An issue for RLZ is how to choose a reference from a set of potentially similar sequences to compress. One choice is to choose one such sequence as the reference, in which case it is called a *real* reference. Instead, *artificial* references can be built by combining sequences from the set or even generating new ones. One of the most powerful methods for building artificial references concatenates uniform samples of the subsequences (Liao et al. 2016).

To succeed in generating the phrases, R must contain every distinct symbol in S. Alternatively, it might be possible to specify a phrase formed by an explicit symbol, or a short substring, without referencing R.

3.3. *Range minimum/maximum queries*

Given an array of integers $A[1, n]$, the range minimum query $rmq(A, i, j)$ returns the position of the leftmost minimum in $A[i..j]$. Analogously, the range maximum query $rMq(A, i, j)$ computes the position of the leftmost maximum in $A[i..j]$. Interestingly, each of these queries can be answered in $O(1)$ time with a structure that uses only $2n + o(n)$ bits and does not access A (Fischer and Heun 2011; Ferrada and Navarro 2017).

ContaCT (Brisaboa et al. 2021) includes a structure that solves both queries, rmq and rMq, within at most $3n + o(n)$ bits. The structure uses rmq and rMq structures over the local minima and maxima, respectively, whose positions are marked in a bitmap. After obtaining the extreme local minimum and maximum, these are compared with the values at the extremes of the queried interval, $A[i]$ and $A[j]$. Therefore, to solve the query in $O(1)$ time, we need to store the array A or a structure that retrieves its cells in $O(1)$ time.

3.4. *The k^2-tree*

A k^2-tree (Brisaboa, Ladra, and Navarro 2014) represents a binary matrix M of size $s \times s$ with a k^2-ary tree, built by recursively splitting M into k^2 submatrices of the same size. Thus, in each level i the size of the submatrices is s^2/k^{2i} cells. The algorithm starts by splitting the matrix into k^2 submatrices of size s^2/k^2, each corresponding to a child of the root node. When the submatrix is full of 0-bits, the node stores a 0-bit; otherwise, it stores a 1-bit. The children of a node are placed in Z-order. For example, in Figure 1, the four children of the root node correspond to the first four 8×8 submatrices, and as seen, the first and the third submatrices in Z-order are represented with 0-bits because they only contain 0-bits. For each node with a 1-bit, we continue recursively splitting its submatrix into k^2 smaller submatrices. This procedure is repeated until reaching a submatrix full of 0s or until the submatrices are individual cells, whose contents are also stored as bits. Therefore, every empty submatrix is encoded with only one node.

The tree is represented without using pointers, using just two bitmaps, T and L.

Bitmap T is the levelwise concatenation of the bits of all the internal nodes, whereas L stores the nodes in the last level (the cell descriptions). The navigation of the tree is supported by *rank* and *select* operations over T. Given a 1-bit at position p in T, its k^2 children are sequentially located from position $children(p) = rank_1(T, p) \times k^2$ of $T : L$, which denotes the concatenation of T and L. The parent of a node at position p of $T : L$ is computed as $parent(p) = select_1(T, \lfloor p/k^2 \rfloor)$. For example, in Figure 1, the position of the first child of the node at $T[3]$ is $rank_1(T, 3) \times 2^2 = 8$. Therefore, its children are stored at $T[8..11]$. The parent of one of those children, for example $T[10]$, is computed as $select_1(T, \lfloor 10/k^2 \rfloor) = 3$.

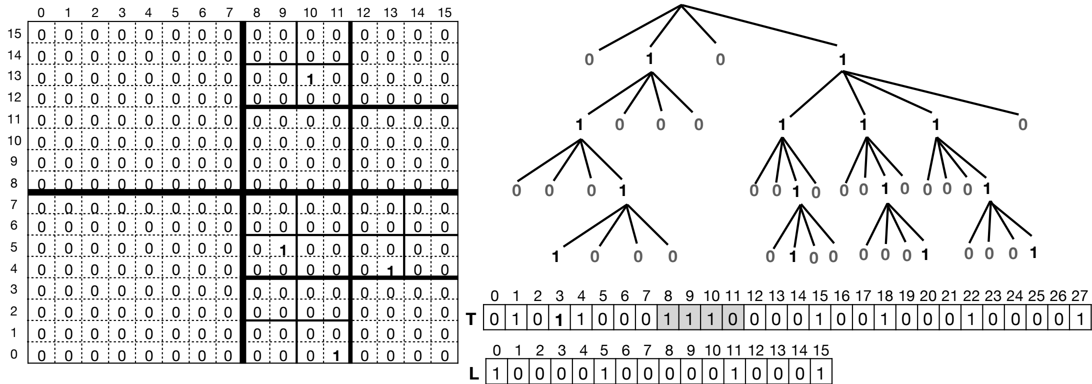


Figure 1.: Example of a k^2-tree with $k = 2$.

By traversing the tree, we can obtain different information about the 1s in the matrix: in a top-down traversal we can discern which 1s are within a region, and from a leaf, we can obtain its position in the whole matrix with a bottom-up traversal.

3.5. *R-trees*

The R-tree (Guttman 1984) is a classical spatial index analogous to a B-tree. The variant that stores points is a balanced multiary tree where the leaves store point sets. Each subtree is summarized with its Minimum Bounding Rectangle (MBR), that is, the smallest rectangle containing all the points in its leaves. Each internal node points to several subtrees and stores their MBRs. To find all the points within a region, we start from the root and recursively enter into every subtree whose MBR intersects the region.

Although R-trees are dynamic and do not consider compression, there is a static version (Brisaboa et al. 2013) where the nodes are compressed. Notice that each MBR can be represented by two coordinates: the bottom-left and the top-right corners. The compressed version represents the bottom-left corner as the difference with respect to the bottom-left of its parent node. The top-right corner is encoded as the difference with the bottom-left corner of the same node.

Figure 2 shows an example of R-tree. The left part shows the points and the MBRs of the nodes; the right part shows the resulting R-tree. To find all objects contained in the query window (or region) Q, the algorithm only traverses the nodes in grey, whose MBRs contain or intersect with Q. In the leaves, the points are checked one by one and added to the solution if they qualify. On the bottom of the right part we show how the corners of node R_6 are encoded.

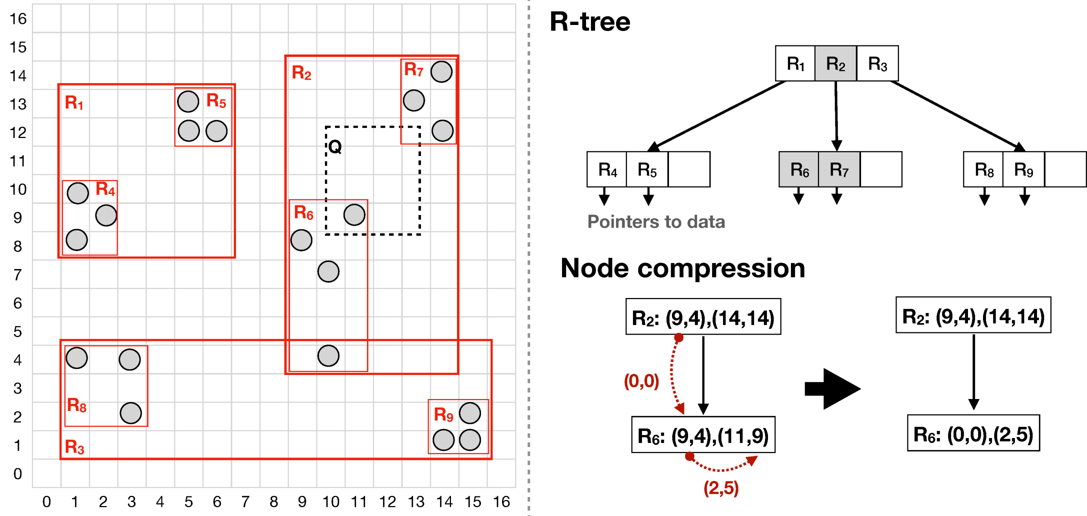


Figure 2.: Example of an R-tree and how its nodes are compressed.

3.6. *Snapshots*

As we presented in Section 2.4, GraCT and ContaCT structures share a spatial index called a *snapshot*. The snapshots are represented essentially as k^2-trees.

We consider the space as a raster, that is, a tessellation of equal-sized squares (cells). Therefore, the locations where there are objects can be represented as a binary matrix having one bit per square of the raster, that is, each 1-bit represents a cell with at least one object. By using the k^2-tree to store that matrix, we also obtain an index over the positions containing objects. In fact, the k^2-tree can be seen as a modern sophisticated version of a region quadtree (Samet 1984). Observe in the left part of Figure 3 that the space is represented as a raster with some cells with objects. On that grid, we depict the quadrants of the k^2-tree (with $k = 2$) shown on the right part. Recall that the k^2-tree is represented with just the bitmaps T and L.

However, we also need to know which objects are within each cell. That is, we need to label those 1-bits of the matrix with the identifiers of the objects lying within the corresponding cell. For this purpose, the snapshot includes an array *perm* and a bitmap Q. Those are filled by traversing the bitmap L from left to right, and for each 1-bit in L, appending to *perm* the list *ids* of object identifiers that lie on the corresponding cell, also adding to Q as many 1-bits as objects are in *ids* minus 1, followed by a 0-bit. Therefore, the objects that are within a leaf $L[i] = 1$ can be located in $perm[l..r]$, where $l = select_0(Q, rank_1(L, i)-1)+1$ and $r = select_next_0(Q, l)$. With this method, after traversing the k^2-tree, the objects within a region can be efficiently identified. In our example, the cell (9,5) is represented with the 1-bit at position 33 of L. This is the second 1-bit of L, therefore, to check how many objects are located in that cell, we search for the position of the first 0-bit (obtained by subtracting 1 to 2, where the 2 comes from the second 1-bit of L) in Q, which in our case is at position 2. Then, we search from the next position (3) until reaching a 0-bit. In our case, there is one 0-bit at position 4, which indicates that there are two objects (corresponding to the positions 3 and 4 of Q) in the cell (9,5). In the same positions 3 and 4 of *perm*, we can obtain the identifiers of the objects within the cell (9,5).

Given an object identifier, computing its location requires detecting its leaf on the

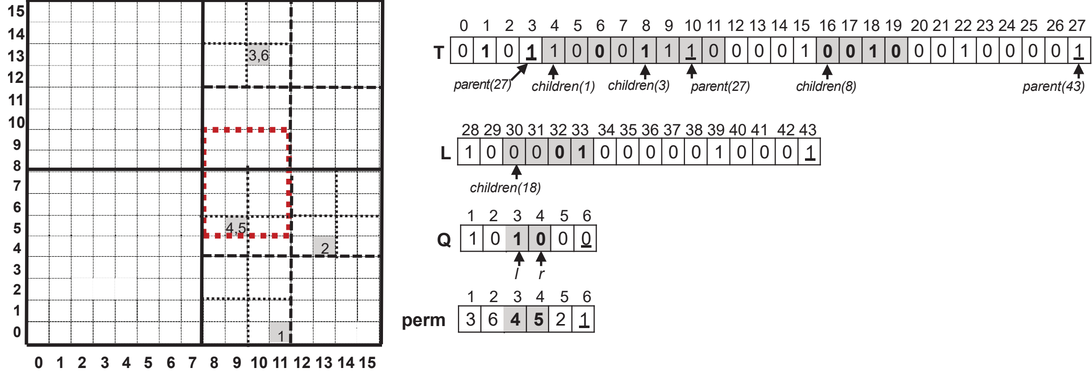


Figure 3.: Example of a snapshot, the steps followed to retrieve the objects within a region, and the location of a specific object.

k^2-tree, and traversing the tree bottom-up. This requires identifying the position of the object in *perm*. To avoid a linear search, the snapshot includes a structure over *perm* (Munro et al. 2012) that uses $n(1 + \epsilon \log_2 n)$ additional bits and computes the location of the object in *perm* in time $O(1/\epsilon)$, where $n = |perm|$ and $0 < \epsilon \leq 1$. Then, the corresponding leaf of that object on the k^2-tree is computed with *rank* and *select* operations. Finally, by traversing the k^2-tree upwards, we can compute the position of that leaf in the space. Therefore, computing the location of an object takes $O(1/\epsilon + \log_k s)$ time. In Figure 3, to obtain the location of object 1, the index of this object in *perm* is computed: 6. It is the fourth $(rank_1(Q, 6 - 1) + 1 = 4)$ leaf with objects, and its position in T:L corresponds to $select_1(L, 4) = 43$. From that position on, the algorithm traverses the tree up to the root (underlined 1-bits) by running *parent* operations. Since each node determines a specific submatrix, the path of the traversal determines the position of the object.

To obtain the objects within a region, the algorithm starts looking for the leaves of the k^2-tree whose labels are 1-bits and that are within the queried region. Those leaves can be computed by traversing the k^2-tree from the root following the nodes whose regions overlap the query area. For each leaf obtained, the algorithm computes its range $perm[l..r]$ of and adds those objects to the solution. For example in Figure 3, to obtain the objects within the region $\langle(8, 5) \times (11, 9)\rangle$, starting at the root of the tree, we traverse the tree with *children* operations through those 1-bits (the shadowed positions of T and L), which represent regions with objects that intersect or are contained within the queried region. Finally, we detect that the sixth element of L (at position 33) is the only leaf with objects within the region, and its corresponding range is $perm[3..4] = 4{,}5$. Hence, 4 and 5 are the identifiers of the objects within the region.

Since the range and values of *perm* is computed in constant time, the total time for a query of area $p \times q$ retrieving *occ* objects is $O(p + q + (occ + 1)k \log_k s)$ (Navarro 2016, Sec. 10.2.1).

3.7. *Compact data structures for trajectories*

The same snapshots are used by both GraCT and ContaCT. The difference between them is in the way the log is compressed. Recall that the log stores the movements between snapshots.

3.7.1. GraCT

To compress the logs GraCT exploits the repetitiveness of the movements by compressing the log with RePair (Larsson and Moffat 2000), a *grammar compressor*.

In the upper part of Figure 4, we can see the original trajectory. The first step for compressing the log is to translate the original trajectory into a sequence of differences. That is, the first position of the object is stored in absolute coordinates, and the rest as differences with respect to the previous position. Thus, the object's position at t_0 is represented as (0,1), and its position at t_1, which in absolute coordinates is (1,0), is represented as $(+1, -1)$. These relative coordinates are the symbols that RePair will compress.

Now, observe in Figure 4, in the middle part, that GraCT adds the snapshots at regular intervals of time. The first snapshot stores the first position in absolute coordinates, the rest of relative coordinates are processed following RePair algorithm.

From the sequence of relative coordinates, RePair takes the most frequent pair of consecutive symbols (relative coordinates in this step). Those occurrences are replaced by a new symbol, and a rule is added to the grammar to keep record of that substitution. In our example, there are five occurrences of the pair of symbols $\langle(+2, +1), (+1, +1)\rangle$, RePair replaces the five occurrences by a new symbol A, and adds a rule $A \rightarrow (+2, +1), (+1, +1)$ to the grammar. This process continues as long there are two or more appearances of a pair of symbols, considering original and new symbols. For example, in our case, we assume that the pair $\langle A, B\rangle$ appears more than once (the figure only displays part of the trajectory) and thus, all appearances of $\langle A, B\rangle$ are replaced by C, and the corresponding rule $C \rightarrow AB$ is added to the grammar.

At the end, the compression produces a sequence of symbols composed of two types of symbols: *terminals* and *nonterminals*. Terminals are the original symbols that were not replaced, whereas nonterminals are the new symbols defined by the grammar. For example, the log of Figure 4 contains three nonterminals: A, B, and C. Observe that nonterminals represent two or more consecutive movements, for example, in Figure 4, B replaces two movements: moving one position to the right and one position up, and then, one position to the right.

GraCT enriches the basic rules of RePair with additional information. For each nonterminal, it stores: $\#t$, the number of movements; (x, y), the total relative displacement in coordinates of those movements; and *mbr*, the relative *MBR* that encloses all the movements of the nonterminal. The nonterminal C in Figure 4 includes the nonterminals A and B, and corresponds to four movements. Applying the movements of C is equivalent to moving five positions right and three up. Finally the relative MBR enclosing its movements is $mbr = (0, 0, +5, +3)$.

With the extra information, GraCT avoids the decompression of some nonterminals. For example, in Figure 4, to retrieve the position at time instant t_6, the algorithm starts by retrieving the position of the object from the snapshot S_0. Then, it traverses the log, until the symbol that contains the information at t_6. During that traversal, the algorithm adds the movements to the previous computed position. In the example, after obtaining the position $(0, 1)$ at t_0 from S_0, the algorithm adds the first movement $(+1, -1)$ to $(0, 1)$. The result $(1, 0)$ is the position at t_1. Then, the positions at t_2 and t_3 are computed as $(1, 0)+(+2, +1) = (3, 1)$ and $(3, 1)+(+1, -1) = (4, 0)$, respectively. The next entry from the log is a nonterminal representing $\#t = 2$ movements, thus that entry has information about the time interval $[t_4, t_5]$. Since it does not reach t_6, we can directly compute the position of the object at t_5 by adding the information of the rule $(x, y) = (+3, +2)$ to $(4, 0)$, whose result is $(7, 2)$. The next entry of the log

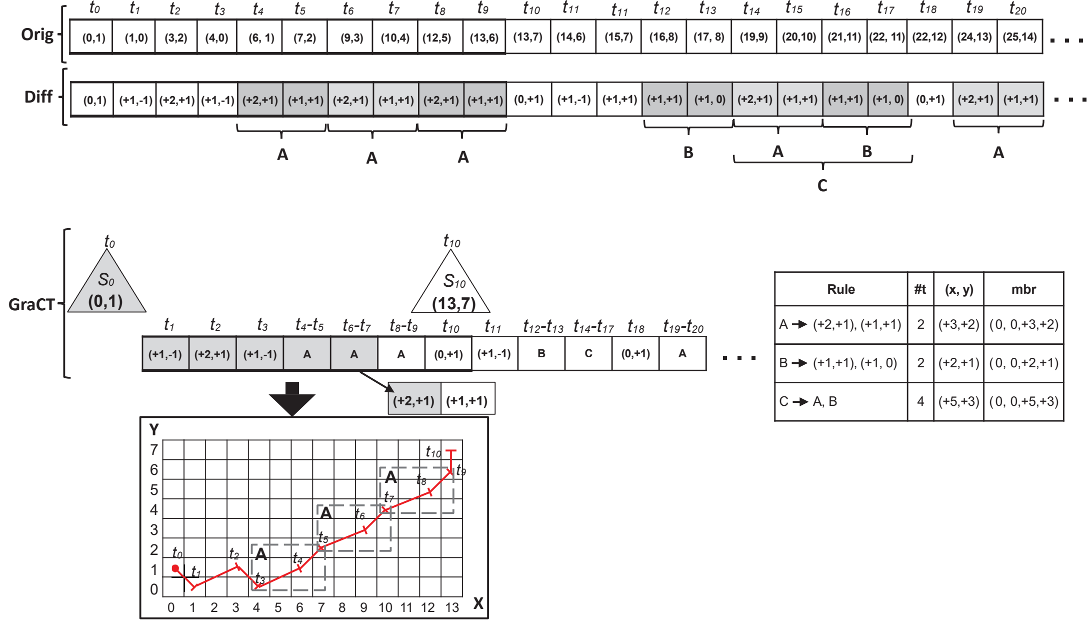


Figure 4.: Example of GraCT for an object trajectory. The snapshots and logs are represented with triangles and arrays, respectively.

covers $[t_6, t_7]$. Since that interval contains t_6, the symbol A has to be decompressed using its rule in the grammar, in this case, $A \to (+2, +1), (+1, +1)$. The first element corresponds to t_6 and thus it gives us the movement of the object for t_6, that is, the position at t_6 is $(7, 2) + (+2, +1) = (9, 3)$.

Similarly, GraCT can compute the rectangular area where an object moves during a time interval represented by a nonterminal. For example, we know that the position at t_3 is $(4, 0)$ and the next symbol is the nonterminal A, whose time interval is $[t_4, t_5]$. By adding $(4, 0)$ to the *mbr* of the rule $(0, 0, +3, +2)$, we know that the *MBR* that covers the movements of that object in $[t_4, t_5]$ has its bottom-left corner at $(4, 0) + (0, 0) = (4, 0)$ and its top-right corner at $(4, 0) + (+3, +2) = (7, 2)$.

Those tricks avoid sometimes decompressing nonterminals, thus speeding up queries. This enables GraCT to achieve time performance comparable to classic spatio-temporal indexes. The main feature of this structure, however, are its good compression ratios on highly repetitive datasets.

3.7.2. ContaCT

ContaCT stores the log movements by using *delta compression*, but its approach is completely different from the classic one. Instead of storing the displacements between pairs of timestamped positions as a pair of integers, it represents those differences by using two bitmaps for each dimension. Let us define a dimension as $D \in \{X, Y\}$, and two bitmaps D_p and D_n. The bitmap D_p stores the positive displacements in dimension D, and D_n the negative displacements. For each positive displacement of c cells, it appends c 0-bits and one 1-bit to D_p, and a 1-bit to D_n. A negative displacement of c cells appends c 0-bits followed by a 1-bit to D_n, and appends one 1-bit to D_p. A zero displacement is represented with a 1-bit in D_n and a 1-bit in D_p.

Observe, in the upper part of Figure 5, the array of differences of an object tra-

	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	t_{14}	t_{15}	t_{16}	t_{17}	
	(0,1)	(+1,-1)	(+2,+1)	(+1,-1)	(+2,+1)	(+1,+1)	(+2,+1)	(+1,+1)	(+2,+1)	(+1,+1)	(0,+1)	(+1,-1)	(+1,+1)	(+1,0)	(+2,+1)	(+1,+1)	(+1,+1)	(+1,0)	. . .
Y		-1	+1	-1	+1	+1	+1	+1	+1	+1	+1	-1	+1	0	+1	+1	+1	0	
Y_p		0	1	0	1	1	1	1	1	1	1	0	1	0	1	1	1	0	
		1	01	1	01	01	01	01	01	01	01	1	01	1	01	01	01	1	
Y_n		1	0	1	0	0	0	0	0	0	0	1	1	0	0	0	0	0	
		01	1	01	1	1	1	1	1	1	1	01	01	1	1	1	1	1	

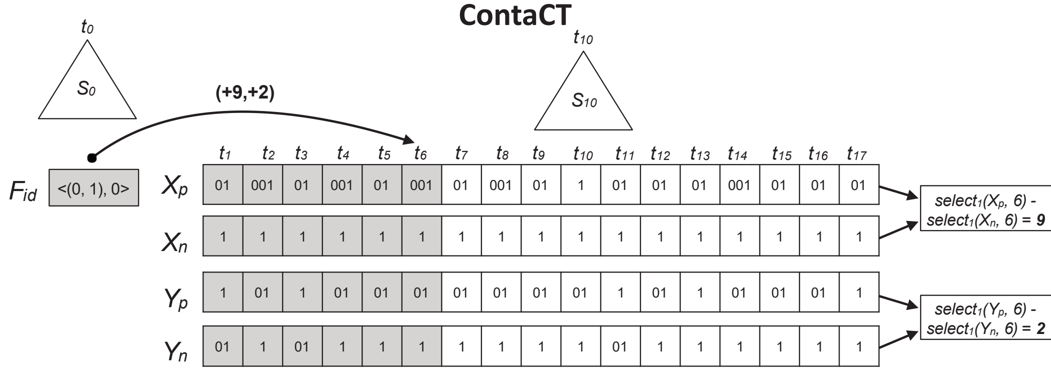


Figure 5.: Example of ContaCT for an object trajectory. The snapshots and logs are represented with triangles and arrays, respectively.

jectory, with the first position in absolute coordinates. The process of obtaining the bitmaps of ContaCT is depicted below for the Y coordinate. First, the values of the Y are extracted in the Y array. Then two arrays are created, Y_p form the positive differences and Y_n for the negative differences. For example, in t_1, the Y array contains a -1, thus the corresponding position of Y_p is 0, and in Y_n, there is a 1. Finally, in the bitmap version of Y_p and Y_n, in each entry, there are as many 0-bits as the value stored in the integer version of the array, plus a 1-bit. For example, in t_1, Y_n stores a 1, thus in the bitmap version there is a 01.

This technique makes it possible to compute the position of an object in constant time. In order to compute the cumulative movement from t_0 to t_i, the algorithm only needs to compute the number of 0-bits in the positive bitmap until the i–th 1-bit, and subtract the number of 0-bits in the negative bitmap until the i–th 1-bit, for each dimension. Therefore, the cumulative movement until t_i in dimension D is computed as $select_1(D_p, i) - select_1(D_n, i)$. ContaCT keeps in F_{id} the initial position of the object id, so by adding the cumulative displacement to that value, we obtain the position at t_i.

At the bottom of Figure 5, we illustrate the complete ContaCT structure for the trajectory and how we can obtain the cumulative movement until t_6. Basically, this is the number of 0-bits in X_p until t_6 (9), minus the number of 0-bits in X_n until t_6 (0), this can be computed in constant time as $select_1(X_p, 6) - select_1(X_n, 6) = 14 - 5 = 9$. For the Y coordinate, the process is analogous, $select_1(Y_p, 6) - select_1(Y_n, 6) = 9 - 7 = 2$. Therefore, the cumulative movement is $(9, 2)$ and by adding it to F_{id}, we obtain the

position of the object at t_6 in $O(1)$ time.

Notice that the position of the object is not obtained from a snapshot, but instead from F_{id}, thus avoiding the traversal of the k^2-tree. The structure that obtains the position of an object in *perm* is then not required.

In addition, ContaCT uses one *rmq* and *rMq* structure for the local minima and maxima, respectively, and the bitmap that marks the positions where the local minima and maxima occur. This structure is replicated for each dimension. Since ContaCT can compute those positions in constant time, the minimum and maximum values can be computed in constant time for each dimension, thus making it possible to obtain the *MBR* of an object between two time instants also in $O(1)$ time.

4. Queries

We now present the queries supported by GraCT and ContaCT in the literature (Brisaboa et al. 2019; Brisaboa et al. 2021); we will expand this set with new queries in subsequent sections. We define a trajectory of n movements of an object *id* as $\mathcal{T}_{id} = \{\langle t_0, p_0\rangle, \langle t_1, p_1\rangle, \ldots, \langle t_n, p_n\rangle\}$, where each pair $\langle t_i, p_i\rangle$ stores the position p_i of the object *id* at time instant t_i. We classify the queries in three groups: *trajectory*, *spatio-temporal*, and *nearest neighbour* queries.

4.1. *Trajectory queries*

This group includes three kinds of queries, all of them are related to obtaining some information from the original set of trajectories.

The first query of this group is *Object position*, which computes the position of a specific object at a given time instant t_q.

Definition 4.1. *Given an object identifier id and a specific time instant t_q, the object position query computes the location p_q such that $\langle t_q, p_q\rangle \in \mathcal{T}_{id}$.*

An extension of *Object position* is *Object trajectory*, which instead of computing the position at a specific time instant, computes all the positions of the object during an interval of time $[t_b, t_e]$.

Definition 4.2. *Given an object identifier id and a time interval $[t_b, t_e]$, the object trajectory query computes the sequence of locations $\langle t_i, p_i\rangle \in \mathcal{T}_{id}$ such that $t_b \leq t_i \leq t_e$, in increasing order of t_i.*

In GraCT, the object position query requires $O(\log_k s + \delta + \log n)$ time in the worst case, where δ is the distance between snapshots. Instead, ContaCT solves it in $O(1)$ time. Both require $O(t_e - t_b)$ additional time for an object trajectory query.

A non-classical query for structures that compress trajectories is computing the MBR of the trajectory of an object during an interval of time $[t_b, t_e]$. This query is quite useful for obtaining summary information about the path followed by an object between t_b and t_e, without computing its whole trajectory.

Definition 4.3. *Given an object identifier id and a time interval $[t_b, t_e]$, the MBR query returns the smallest rectangular area R such that, for every element $\langle t_i, p_i\rangle \in \mathcal{T}_{id}$ where $t_b \leq t_i \leq t_e$, it holds that $p_i \in R$.*

In ContaCT, this query can be solved in $O(1)$ time by using *rmq* and *rMq* structures. Instead, GraCT requires $O(\log_k s + (t_e - t_b + \delta) + \log n)$ time in the worst case, because it may need to extract the whole trajectory from t_b to t_e (though in practice it can skip most nonterminals using the *mbr* fields).

4.2. *Spatio-temporal queries*

Spatio-temporal queries identify those objects that satisfy a spatio-temporal constraint, like being within a region during an interval of time. This group includes the typical queries supported by methods that focus on indexing trajectories (Section 2.3) and methods like TrajStore and SharkDB.

The simplest query, *Time Slice*, retrieves the objects within a region r_q at a time instant t_q.

Definition 4.4. *Given a region r_q and a time instant t_q, the time slice query returns the set of object identifiers O, such that, for each $id \in O$, there exists a pair $\langle t_q, p_q \rangle \in \mathcal{T}_{id}$ where $p_q \in r_q$.*

The query *Time Interval* extends *Time Slice* so that the queried time instant t_q becomes an interval $[t_b, t_e]$ of time.

Definition 4.5. *For a given region r_q and a time interval $[t_b, t_e]$, the time interval query returns the set of object identifiers O, such that, for each $id \in O$, there exists at least one pair $\langle t_i, p_i \rangle \in \mathcal{T}_{id}$ where $t_b \leq t_i \leq t_e$ and $p_i \in r_q$.*

4.3. *Nearest neighbour queries*

Nearest neighbour queries compute the objects closest to a spatial geometry (such as a point or a line). Systems that index trajectories with R-trees (MVR-tree, SETI, Trajstore, SharkDB) are efficient at computing the K objects closest to a given position at a given time instant. This is the only nearest neighbour query supported by GraCT and ContaCT.

Definition 4.6. *The K-nearest neighbour query for a point p_q at time instant t_q returns a set O of objects such that $|O| = K$ and $d(p_q, id_1) \leq d(p_q, id_2)$ for any objects $id_1 \in O$ and $id_2 \notin O$, where $d(p_q, id)$ is the Euclidean distance from point p_q to the position of object id at time instant t_q (i.e., such that $\langle p_q, t_q \rangle \in \mathcal{T}_{id}$).*

GraCT and ContaCT can efficiently handle the three types of queries: *trajectory*, *spatio-temporal*, and *nearest neighbour*. The other indexes are only efficient in running one or two types of queries. This is a consequence of the architecture of GraCT and ContaCT: they explicitly represent trajectories of objects with the log, which allows solving *trajectory* queries, and they are equipped with spatial indexes (snapshots) at regular time instants, which help in solving the *spatio-temporal* and *nearest neighbour* queries. The MVR-tree, for example, stores an R-tree per time instant, which solves *spatio-temporal* and *nearest neighbour* queries fast, but recovering the trajectory of a given object is costly.

5. Supporting complex nearest neighbour queries

In this section we introduce two new more sophisticated nearest neighbour queries not supported in the original GraCT and ContaCT articles: *KNN of trajectories* and *KNN during an interval*. We show how the data structures already present in both GraCT and ContaCT can be used to solve these sophisticated queries: First, the spatial index (snapshot) is able to prioritize the objects according to a distance bound. Second, both structures can implement an operation *refine* that shrinks the previous bound to a tighter one.

5.1. *KNN of trajectories*

Given a query trajectory $\mathcal{T}_q$, our goal is to obtain the K trajectories that are closest to it during an interval of time $[t_b, t_e]$. We then obtain a list of objects whose trajectories during $[t_b, t_e]$ are closest to $\mathcal{T}_q$. There are various choices in the literature to define a closeness (or similarity) measure between trajectories. The simplest measures assume that the points are already aligned and are variants of averaging the Euclidean distances between the corresponding points (Su et al. 2020). More sophisticated variants enable varying the alignment between the points of both trajectories. For example, the *edit distance* (EDR) (Chen, Özsu, and Oria 2005) computes the number of 'edits' needed to transform one trajectory to the other, where the cost of substituting one point by another may be their Euclidean distance in space. *Dynamic time warping* (DTW) (Berndt and Clifford 1994) is a variant of the latter that allows a single point in one trajectory to align with many of the other. The *Discrete Fréchet distance* (DFD) (Eiter and Mannila 1994), instead, measures the maximum Euclidean distance between the aligned points. It can be regarded as an adaptation of the Hausdorff distance (Hausdorff 2005) to trajectory points, and has become a standard measure of distance between two parametric curves.

We choose a simplification of the DFD that suits our scenario where the points are already aligned. This becomes simply the farthest distance between all the (already aligned) points along the time interval. Such a definition is also connected with variants of the Euclidean distance that take the L_p-norm over the distances between the aligned points (Su et al. 2020); while the classic Euclidean distance uses L_1, our definition corresponds to using L_∞. Besides its support in the literature, an algorithmic advantage of this definition is that we can easily compute lower and upper bounds of the maximum distance by using the MBR of the trajectory, and these bounds can be progressively refined by successively splitting the trajectory into subtrajectories and using their MBRs. Note that the DFD measure does not depend on some of those MBRs (e.g., an MBR whose maximum distance to the trajectory is smaller than the current lower bound). We can then focus on the parts of the trajectory that can contain the maximum distance and avoid further splitting the subtrajectories that cannot. Instead, using a similarity measure that sums or averages the pointwise distances forces us to consider every point. Indeed, various techniques that build on sums of distances allow returning approximate answers in order to perform efficiently, for example PDTW (Keogh and Pazzani 2000), STLCSS (Vlachos, Kollios, and Gunopulos 2002), and STLC (Shang et al. 2017). Instead, we always find the correct KNN answers under our distance.

Definition 5.1. *The K-Nearest neighbour of trajectories for a given trajectory $\mathcal{T}_q$ at time interval $[t_b, t_e]$ returns a set O of objects such that $|O| = K$ and $d_{max}(\mathcal{T}_q, \mathcal{T}_1) \leq$*

$d_{max}(\mathcal{T}_q, \mathcal{T}_2)$ *for the trajectory* $\mathcal{T}_1$ *of any* $id_1 \in O$ *and* $\mathcal{T}_2$ *of any* $id_2 \notin O$*, during* $[t_b, t_e]$*. We denote with* $d_{max}(\mathcal{T}_q, \mathcal{T}_i)$ *the maximum Euclidean distance between two trajectories,* $d_{max}(\mathcal{T}_q, \mathcal{T}_i) = \max\{d(p_k, p_j),\ \langle p_k, t_l\rangle \in \mathcal{T}_q,\ \langle p_j, t_l\rangle \in \mathcal{T}_i,\ t_l \in [t_b, t_e]\}$*.*

The idea to solve this query without comparing the trajectory of every object with $\mathcal{T}_q$ is to prioritize both the traversal of snapshots towards promising objects, and the precise computation of the trajectories of the objects, so that we only compute as much as necessary to identify the K closest trajectories. To compute priorities efficiently, the algorithm uses just the MBRs of trajectories to compute a range $[l, h]$ where the maximum distance between both trajectories must lie.

GraCT and ContaCT implement the same algorithm, though they differ in the way the MBRs are computed and refined. In general terms, we divide our exposition in two parts: *prioritizing the objects using the snapshots* and *refining the object distance bounds*. Algorithm 1 gives the pseudocode.

5.1.1. Prioritizing the objects using the snapshots

We take advantage of the hierarchical structure of the k^2-tree representing the snapshots, to sort the objects according to their chances of being close to $\mathcal{T}_q$. The objects to be prioritized are obtained from the snapshots in the interval $[t_b, t_e]$, that is, the snapshots from time instant $\lfloor t_b/\delta \rfloor \times \delta$ to $\lfloor t_e/\delta \rfloor \times \delta$, where δ is the number of time instants between snapshots. The idea is to process first the k^2-tree nodes that are closer to $\mathcal{T}_q$, independently of their depth in the k^2-tree.

The algorithm builds a priority queue Q_{global} where each element, called a *header*, is a triple $\langle n, l, h\rangle$. The term n is a k^2-tree node and $[l, h]$ is a range bounding the maximum distance between $\mathcal{T}_q$ and any object in the region of n. Q_{global} is a min-heap sorted by l and the ties are broken by h. We know the maximum speed M at which an object moves in the dataset, and use it to compute $[l, h]$ by expanding the area of n in all directions at the maximum speed, and then comparing the expanded region with the MBR of $\mathcal{T}_q$.

Let R be the region defined by n. The snapshot gives the position of the objects in a given time instant, say τ. Since every object moves at most $c = (t_e - \tau) \cdot M$ cells on every direction from time instant τ to t_e, an object within $R = [x_1, y_1] \times [x_2, y_2]$ at τ can only move within the expanded region $R' = [x_1 - c, y_1 - c] \times [x_2 + c, y_2 + c]$. Hence, we define l and h as the minimum and maximum distances, respectively, between R' and the MBR of $\mathcal{T}_q$. When the query spans several snapshots, the range $[t_b, t_e]$ is intersected with the area covered by the log of each snapshot when performing this computation.

The process then starts by adding the roots of the k^2-trees of all the snapshots involved in the query; see lines 1–5 of Algorithm 1, where *distances* computes the described distance estimation between $\mathcal{T}_q$ and k^2-tree nodes. The nodes whose objects are estimated to have more chances to be closer to $\mathcal{T}_q$ are at the top of Q_{global}. We thus traverse the internal nodes of all the involved snapshots by popping the elements from Q_{global}, and reinserting the children of the extracted k^2-tree nodes (lines 7–12). It is then more likely that we reach sooner the k^2-tree leaves that contain the objects whose trajectories are closer to $\mathcal{T}_q$.

5.1.2. Refining the object distance bounds

Once the children of a node extracted from Q_{global} are leaves, we do not reinsert those leaves into Q_{global}, but rather extract the objects associated with each leaf (lines 14–16;

Algorithm 1: KNNTrajectory$(K, \mathcal{T}_q, t_b, t_e)$

```
Q_global ← ∅; result ← ∅;
MBR_q ← MBR(T_q, t_b, t_e);
for S_i ∈ [S_{⌊t_b/δ⌋×δ}, ..., S_{⌊t_e/δ⌋×δ}] do
    ⟨l, h⟩ ← distances(S_i.root.R, MBR_q)
    Q_global.add(⟨S_i.root, l, h⟩);
while Q_global ≠ ∅ and |result| < K do
    ⟨e, l, h⟩ ← Q_global.pop();
    if e is a node then
        for nonempty node ∈ e.children do
            if node is internal then
                ⟨l_new, h_new⟩ ← distances(node.R, MBR_q);
                Q_global.add(⟨node, l_new, h_new⟩);
            else
                for id ∈ e.objects and id.traj not initialized do
                    ⟨l_new, h_new⟩ ← init(id.traj, T_q);
                    Q_global.add(⟨id, l_new, h_new⟩);
    else
        ⟨e', l', h'⟩ ← Q_global.top();
        if h ≤ l' then
            result ← result ∪ {e};
        else
            ⟨l_new, h_new⟩ ← refine(e.traj, T_q);
            Q_global.add(⟨e, l_new, h_new⟩);
return result
```

the objects are $id \in e.objects$) and insert those. This means that Q_{global} has not only headers associated with k^2-tree nodes, but also with objects. Those objects are also associated with an MBR, which is not anymore bounded using the maximum speed, but with data from their actual trajectory, which is stored in *id.traj*. In principle, function *init* initially computes the MBR of the trajectory during $[t_b, t_e]$ using the *mbr* query provided by GraCT or ContaCT, and uses it to provide a range $[l_{new}, h_{new}]$ of maximum distances to the MBR of $\mathcal{T}_q$.

As the algorithm progresses, the trajectory of the object will be successively split along time intervals to provide a better estimation, and *id.traj* will become a max-heap of pieces of this trajectory. Each element of *id.traj* contains the minimum (d_{min}) and maximum (d_{max}) distance between the *MBR* of the object during some interval $[t_i, t_j] \subseteq [t_b, t_e]$ and the MBR of $\mathcal{T}_q$ at the same interval. The exact partitioning into intervals in the beginning (*init*) and after successive refinements (*refine*) depends on the log used (GraCT or ContaCT); we will describe them later.

The queue *id.traj* is sorted by d_{max}, and the ties are broken with d_{min}. That is, we locate on top of the queue the time interval most likely to contain the point of the trajectory of *id* that is farthest from $\mathcal{T}_q$. The top of *id.traj* is used to compute the range $[l, h]$ with which the header is prioritized in Q_{global}.

Lines 18–23 show how we process object headers. We first examine the next header in Q_{global}. If the maximum bound h of the current object does not exceed the minimum bound l' of the next header, we can be sure that the current object is the next result, and include it in the result. Otherwise, we refine the trajectory estimation of the current object e. This is done by partitioning the top trajectory interval in *e.traj* into smaller subintervals, whose upper-bound distances to $\mathcal{T}_q$ will be tighter, and reinserting them into *e.traj*. This is done by function *refine*, which provides a new estimation

$[l_{new}, h_{new}]$ that is used to reinsert the object in Q_{global}.

5.1.3. *Computing the MBR of the input trajectory*

Along the algorithm, we need to compute MBRs of the input trajectory $\mathcal{T}_q$ between arbitrary time instants $[t_i, t_j]$. We then start the query by preprocessing $\mathcal{T}_q$ so as to build range minima and maxima (rmq and rMq) query structures (Fischer and Heun 2011; Ferrada and Navarro 2017) on the values of $\mathcal{T}_q$ along each axis (X and Y), in time $O(|\mathcal{T}_q|)$. We can then compute any $MBR(\mathcal{T}_q, t_i, t_j)$ in constant time as

$$[X[rmq(X, t_i, t_j)], Y[rmq(Y, t_i, t_j)] \times [X[rMq(X, t_i, t_j), Y[rMq(Y, t_i, t_j)].$$

5.1.4. *GraCT*

Let $S = \{s_1, s_2, \ldots, s_t\}$ be the symbols of the log covering $[t_b, t_e]$. In GraCT, we can compute in constant time the MBR of each $s_r \in S$, either from a single movement (if s_r is a terminal) or else from the mbr data we store for s_r. Let s_r span times $[t_i, t_j]$. We compute the minimum and maximum distances, d_{min} and d_{max}, between the MBR of s_r and $MBR(\mathcal{T}_q, t_i, t_j)$. This is stored in a tuple $\langle t_i, t_j, d_{min}, d_{max}, p_{i-1}, s_r \rangle$, where p_{i-1} is the object position at time t_{i-1}. The *init* operation adds all those tuples to the priority queue $id.traj$, and returns the values h_{new} and l_{new} as the maximum d_{max} and d_{min} values, respectively, of all those tuples.

Since $id.traj$ is a max-heap sorted by d_{max}, h is always the d_{max} value on top of the queue, so it can be computed in $O(1)$ time. Instead, the value of l can be located in another position of the queue. To avoid traversing the queue looking for the maximum d_{min}, we use another max-priority queue sorted the d_{min} values, and synchronized with $id.traj$. That arrangement allows us to compute l in $O(1)$ time as well.

Once $id.traj$ is initialized, every time we call *refine*, it takes the tuple on top of the queue. If the tuple has $t_j = t_i$, we have obtained the exact maximum distance of the object to $\mathcal{T}_q$. Otherwise, the tuple refers to a nonterminal s_r and we apply the corresponding rule to expand it. The tuple is then split into two that cover time intervals $[t_i, t_m]$ and $[t_{m+1}, t_j]$. After obtaining the MBRs associated with those intervals, their d_{min} and d_{max} values are computed with respect to $MBR(\mathcal{T}_q, t_i, t_m)$ and $MBR(\mathcal{T}_q, t_{m+1}, t_j)$, respectively, and reinserted in $id.traj$.

For example, in Figure 6, the first step takes the tuple $\langle t_9, t_{12}, 10, 6, (10, 6), C \rangle$. We observe that $s_r = C$ and $C \rightarrow A, B$, and A has $\#t = 2$, $(x, y) = (+3, +2)$ and $mbr = (0, 0, +3, +2)$. Since A lasts two movements, the new tuples cover the intervals $[t_9, t_{10}]$ and $[t_{11}, t_{12}]$. For the first one, the previous position is still $(10, 6)$, and for the second one, the previous position is $(10, 6) + (+3, +2) = (13, 8)$. Then, the MBR of each new tuple can be computed by adding those previous positions to the stored mbr for A and B, respectively. With those MBRs, the distances of the tuples are computed. Notice that our $\mathcal{T}_q$ is a horizontal line in $y = 0$, thus its MBR covers all the cells at $y = 0$. For the first new tuple the distances are $d_{min} = 6$ and $d_{max} = 8$, and for the second they are $d_{min} = 8$ and $d_{max} = 9$. The new values for h_{new} and l_{new} are 9 and 8, respectively.

5.1.5. *ContaCT*

ContaCT can compute arbitrary MBRs in constant time, which simplifies the implementation of *init* and *refine*. In principle, *init* could initialize $id.traj$ with a single entry relating $MBR(id, t_b, t_e)$ and $MBR(\mathcal{T}_j, t_b, t_e)$, computing d_{min} and d_{max} from

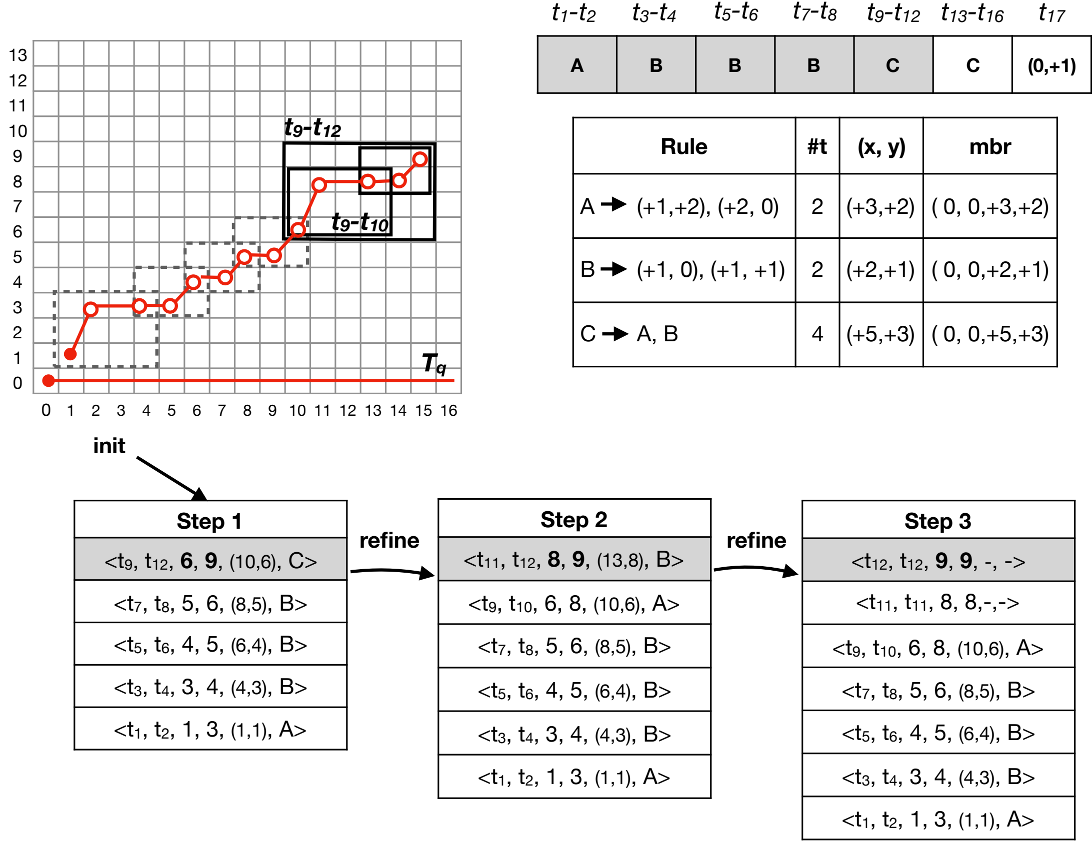


Figure 6.: Example of GraCT for *KNN of trajectories*. The queues below the structure represent the different steps and states of *id.traj*.

those. However, the time interval $[t_b, t_e]$ can cover several snapshots, so *init* must insert one tuple for each. For each involved snapshot starting at time τ, it computes the distances d_{min} and d_{max} between the MBRs of *id* and $\mathcal{T}_q$ within times $[\max(\tau, t_b), \min(\tau + \delta - 1, t_e)]$.

The mechanics are then exactly as for GraCT, except that the tuples stored have the form $\langle t_i, t_j, d_{max}, d_{min}\rangle$. To apply *refine* on such a tuple, the algorithm divides it by half, $t_m = \lfloor (t_i + t_j)/2 \rfloor$. For each half the algorithm computes its MBR and the values d_{max} and d_{min} with respect to $\mathcal{T}_q$, all in constant time.

Figure 7 shows an example. After the *init* operation we compute the *MBR* during $[t_0, t_{12}]$ with $d_{min} = 1$ and $d_{max} = 9$. With *refine*, that tuple is split into the intervals $[t_0, t_6]$ and $[t_7, t_{12}]$. According their MBRs we obtain the distances $d_{max} = 5$ and $d_{min} = 1$, and $d_{max} = 9$ and $d_{min} = 5$, respectively. Therefore the new tuples are $\langle t_0, t_6, 5, 1\rangle$ and $\langle t_7, t_{12}, 9, 5\rangle$, and the new boundaries for the maximum distance are $l = 5$ and $h = 9$.

5.2. *KNN during an interval*

In this kind of query, we compute the K objects whose trajectories during an interval of time $[t_b, t_e]$ are the closest to a given point p_q. The distance between a trajectory and p_q is defined as the minimum distance to p_q at any time instant $t \in [t_b, t_e]$. This distance

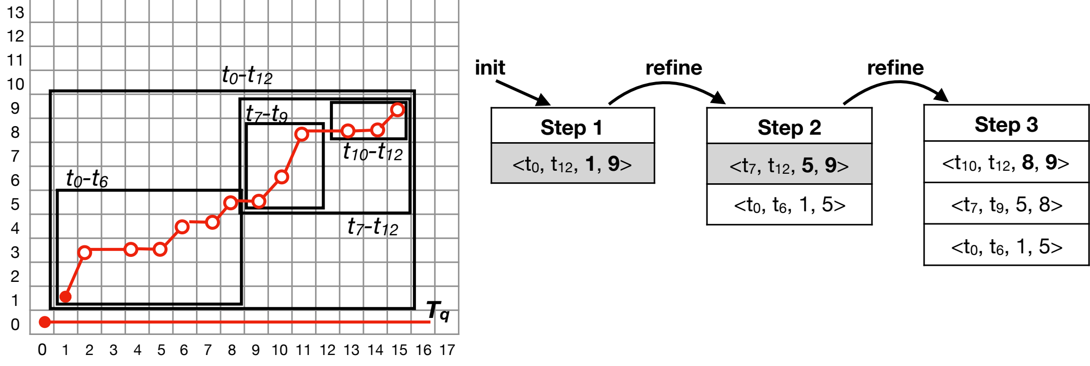


Figure 7.: Example of ContaCT for *KNN of a trajectory*. The right part represents the different steps and states of $id.traj$.

measure was applied in previous works and applications. For example, in applications that study animal habits (Gao et al. 2007) and in social networks where users want to plan a trip based on routes of friends that visit their points of interest (Tang et al. 2011).

Definition 5.2. *The K-Nearest neighbour of an interval* $[t_b, t_e]$ *with respect to a point* p_q *returns a set* O *of objects such that* $|O| = K$ *and* $d_{min}(p_q, \mathcal{T}_1) \le d_{min}(p_q, \mathcal{T}_2)$ *for any trajectory* $\mathcal{T}_1$ *of* $id_1 \in O$ *and* $\mathcal{T}_2$ *of* $id_2 \notin O$*, during* $[t_b, t_e]$*. We denote* $d_{min}(p_q, \mathcal{T}_i) = \min\{d(p_q, p_k),\ \langle p_k, t_l\rangle \in \mathcal{T}_i,\ t_l \in [t_b, t_e]\}$.

This query is similar to that of Section 5.1, with two differences:

- The distance is computed with respect to a point p_q, instead of a trajectory $\mathcal{T}_q$.
- This query finds the minimum distance of the objects to p_q during the interval of time $[t_b, t_e]$, instead of the maximum distance to $\mathcal{T}_q$.

The algorithm is then similar to the one of Section 5.1, but instead of computing the maximum distance to the input spatial data (point or trajectory), we compute the minimum distance. The previous algorithm then undergoes some modifications:

- Each $id.traj$ is transformed to a min-heap where the tuples are sorted by the minimum distance d_{min} from the MBR to p_q, and the ties are broken with d_{max}. By splitting the top tuple of $id.traj$, we focus on the interval of time where the object has more chances to be closer to p_q.
- We now have the tuple with the smallest d_{min} on top, thus we can compute l in $O(1)$ time, but now we cannot compute h in $O(1)$ time. Analogously to Section 5.1, we synchronize another min-heap priority queue with $id.traj$, which stores the values of d_{max}. We can then compute h in $O(1)$ time as well.
- Q_{global} is still a min-heap storing tuples $\langle n, l, h\rangle$ sorted by l and breaking ties with h. As we now look for the minimum distance, however, $[l, h]$ now bounds the minimum distance between p_q and the MBR of n. If n is an object, l and h are the minima over the d_{min} and d_{max} values of all the trajectory segments, respectively.
- Since we are comparing the distance to a point p_q instead of a trajectory $\mathcal{T}_q$, we do not need the rmq and rMq structures for computing MBRs on $\mathcal{T}_q$.

5.3. *Analysis*

It is not easy to give meaningful worst-case time guarantees on KNN algorithms, because an adversarial setup where all the objects are at almost the same distance to the query forces the algorithms to inspect nearly every object in the dataset; this is known as the "curse of dimensionality" (Chávez et al. 2001). A useful concept for this kind of algorithms is *range optimality* (Böhm, Berchtold, and Keim 2001; Hjaltason and Samet 2000), which states that an algorithm retrieving the K nearest neighbors performs the same amount of work as the canonical range-search algorithm that finds all the objects at distance d from the query, where d is the distance between the query and its K-th nearest neighbor.

Hjaltason and Samet (Hjaltason and Samet 2000; Hjaltason and Samet 2003) described a generic KNN search algorithm that works on any hierarchical data structure and is range-optimal (with respect to that data structure). The algorithms we have described in this section follow their generic scheme, and are therefore range-optimal.

The hierarchy in our case corresponds to the composition of the k^2-tree and then the recursive partitioning of the trajectory into subtrajectories. In other words, if we had to traverse our data structure in order to find all the trajectories at distance d from our (trajectory or point) query, we would have to traverse exactly the same nodes of the hierarchy. The multiplicative overhead with respect to the range-search algorithm is the $O(\log n)$ time incurred by manipulating the heaps, plus the time to compute the *MBR* queries. This is variable for GraCT and constant for ContaCT, as described.

5.4. *Experimental evaluation*

We now experimentally evaluate the performance of the new KNN queries on GraCT and ContaCT. We modified their original C++ implementations, and used some components of the SDSL library[1] (Gog et al. 2014). There are two possible implementations for ContaCT, where the bitmaps D_p and D_n are represented either in plain form or using a representation for sparse bitmaps called *sdarray* (Okanohara and Sadakane 2007), depending on the magnitude of the differential values. The structure that uses plain bitmaps is labelled as ContaCT, and the one with sparse bitmaps is called ContaCT-SD.

The experiments were run on an Intel® Core™ i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache and 64 GB of RAM, running Debian GNU/Linux 9 with kernel 4.9.0-8 (64 bits), gcc version 6.3.0 with `-O3` optimization.

5.4.1. Datasets

We used the four datasets originally used to evaluate ContaCT (Brisaboa et al. 2021), formed by three real and a pseudo-real one:

- `Ships`: a real dataset that contains the coordinates of 4,461 vessels travelling within the UTM Zone 10 during one month of 2017. The original data can be obtained from MarineCadastre.[2]
- `Planes`: real flight data of 2,263 aircrafts from 30 different airlines between 30 European airports. Altitude is not considered, only latitude and longitude are represented in our dataset. The original data can be obtained from *OpenSky*

1. `https://github.com/simongog/sdsl-lite`
2. `http://marinecadastre.gov/ais`

	Ships	Planes	Taxis	Ciconia
Total objects	4,461	2,263	24	88
Total points	63,093,559	36,741,877	46,677,278	4,390,159
Max x	6,000	229,010	1,074,480	4,073,661
Max y	647,755	46,872	340,142	2,995,928
Max *time*	44,639	172,547	2,102,639	505,573
Size Plain	1,413.47 MB	809.00 MB	1,024.00 MB	107.09 MB
Size Bin	541.54 MB	350.40 MB	426.08 MB	41.87 MB
Size *p7zip*	57.88 MB	85.40 MB	86.91 MB	12.06 MB

Table 1.: Datasets and their dimensions.

Network.[3]

- `Taxis`: a pseudo-real dataset containing trajectories of 24 taxis in New York City during 2013. Since the original dataset only includes the origin and destination of each trip, the trajectory was computed as the shortest path between them by taking into account the road network. The original data are available at *NYC Taxis: A Day in the life*.[4]
- `Ciconia`: a small and non-repetitive real dataset of 88 white storks travelling between Europe and North Africa from 2013 to 2019. The original data can be obtained from *MoveBank Data Repository* (Flack, Fiedler, and Wikelski 2016; Cheng et al. 2019).

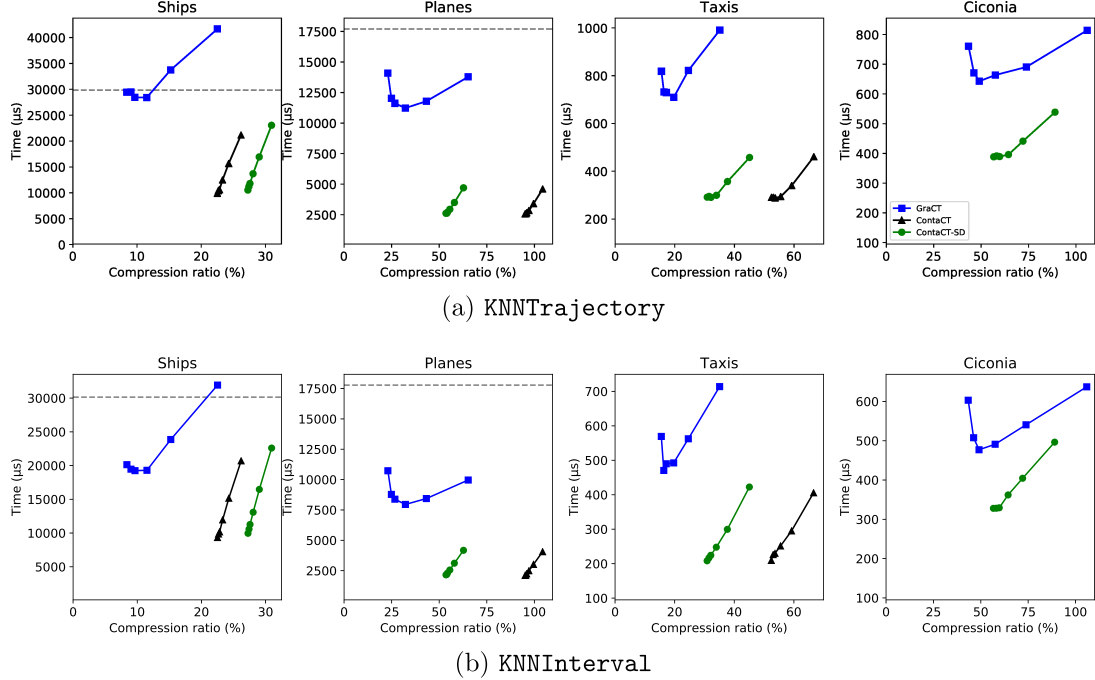
Those datasets are preprocessed as in previous work (Brisaboa et al. 2021). The trajectories are stored in a plain text file composed of four columns: *object identifier*, *time instant*, *x coordinate*, and *y coordinate*. The features of each of our datasets are shown in Table 1. We show the size of the binary representation of each dataset, that is, by using the number of bytes required for each column. The last row is the size after compressing the binary representation with *p7zip* and gives us an idea of how repetitive the data is. We observe that *p7zip* compresses the data to 10%–30% of its binary representation.

5.4.2. Time performance

In our first experiment, we implement the algorithms of KNN of trajectories (`KNNTrajectory`) and KNN during an interval (`KNNInterval`) on GraCT, ContaCT, and ContaCT-SD. All those structures are configured with four different distances δ between snapshots: 30, 60, 120, 240, 360, and 720. For each type of query, we ran 1,000 different queries and compute the average *user time*. In both cases, K is a random value between 1 and 50. The queried interval covers 200 time instants. Additionally, we designed brute force algorithms for solving both KNN queries. They go through all the trajectories computing the distances of each timestamped position with respect to the input trajectory or point and sorting them according to minimum/maximum distance by using a min-heap. The K first trajectories of that heap will be the solution of the query.

Figures 8a and 8b show the average *user time* of the `KNNTrajectory` and `KNNInterval` queries, and the compression ratio of the three structures (compressed

3. `https://opensky-network.org`
4. `http://chriswhong.github.io/nyctaxi/`



(a) `KNNTrajectory`

(b) `KNNInterval`

Figure 8.: Average time and compression ratios for `KNNTrajectory` and `KNNInterval`.

divided by original space, as a percentage). The solid lines represent the three different structures, and their markers the four different configurations. The configuration with the largest $\delta = 720$ obtains the best (smallest) compression ratio, and the one with the shortest $\delta = 30$, the worst compression ratio. In `Ships` and `Planes`, we can observe a horizontal dashed line that represents the running time of the brute force algorithm. In `Taxis` and `Ciconia`, the brute force algorithm requires around 20ms and 2ms, respectively, that is, more than twice the time consumed by the compact data structures. We omit the horizontal dashed lines in those plots. In addition, in `Ciconia` we omit ContaCT because it uses significantly more space than the raw data.

ContaCT and ContaCT-SD are much faster than GraCT at solving `KNNTrajectory` queries. Comparing the least-space configuration of ContaCT or ContaCT-SD with the fastest configuration of GraCT, we see that ContaCT is 2.9 times faster on `Ships`, and ContaCT-SD is 4.3 times faster on `Planes`, 2.3 on `Taxis`, and 1.7 on `Ciconia`. This is because ContaCT avoids the linear traversals of the log performed by GraCT, in order to retrieve the symbols and add them to the priority queue. The number of elements in the priority queue of each object is also smaller in ContaCT than in GraCT. This is more noticeable on `Ships`, the only case where GraCT is slower than the brute force algorithm. Instead, the ContaCT variants outperform the brute force algorithm when solving `KNNTrajectory`, being 3 times faster on `Ships` and 8.4 on `Planes`.

The results for the `KNNInterval` queries are similar, but we observe an improvement in time performance. This is because the query is simpler and does not compute MBRs on an input trajectory. Since GraCT initializes the queue with more elements, the improvement with respect to `KNNTrajectory` is more evident, and now the three structures outperform the brute force algorithm. In the dataset where the brute force algorithm is closer to the performance of the three structures (`Ships`), we observe that GraCT is 1.5, ContaCT is 3.2, and ContaCT-SD is 3 times faster.

Although ContaCT is significantly faster than GraCT, it generally uses much more

space. Except on `Ciconia`, in the tested datasets with $\delta = 720$, GraCT uses 40%–80% of the space of ContaCT. This difference is due to the ability of GraCT to exploit the repetitiveness of movements between trajectories.

5.5. *Discussion*

The experimental evaluation shows the performance of our proposed nearest-neighbor algorithms for `KNNTrajectory` and `KNNInterval` queries on GraCT and ContaCT. Both structures offer a good space-time trade-off, but each is better depending on the application. When compression is a primary requirement, GraCT is the best option because it exploits the repetitiveness between trajectories, and thus it uses half the space of ContaCT. However, GraCT is slower than ContaCT to solve `KNNTrajectory` and `KNNInterval` queries. Note that, in the first step to solve these queries, GraCT needs to traverse the log, which is not necessary in ContaCT. During that traversal, GraCT inserts in the priority queue each log entry that belongs to the queried interval, whereas ContaCT initially adds only one entry per object. The subsequent process of each entry is also more costly in GraCT than in ContaCT. Hence ContaCT is more suitable for scenarios where time performance is a primary goal and space is secondary.

Therefore, we observe two weak points of these structures when solving `KNNTrajectory` and `KNNInterval` queries: (1) the performance of GraCT is not too far from the brute force algorithm; (2) ContaCT obtains a good time performance, but there is an important difference in compression compared to GraCT. In the following sections, we propose solutions to both points, which have positive effects on other types of queries.

6. Snapshots based on R-trees

One of the weakest points in the algorithms for `KNNTrajectory` is that the snapshot based on k^2-trees does not give any information on where the objects can be during the interval up to the next snapshot. That is, they only store the position of the objects at the time instant represented by the snapshot. Therefore, to upper bound the movements of the object during the time interval until the next snapshot, the query algorithms expand the k^2-tree node areas assuming that the objects move at the maximum possible speed in all possible directions. This leads to poor filtration performance, sometimes worse than the brute-force algorithm. In this section we present an alternative snapshot data structure to alleviate this problem, which instead of storing the current position of the objects, stores their MBR up to the next snapshot, that is, for each object id in the snapshot at time τ we store $MBR(id, \tau, \tau + \delta - 1)$. For storing those MBRs, we use a static compressed R-tree (Brisaboa et al. 2013) instead of the k^2-tree.

The first reason for this choice is that the k^2-tree is conceptually a region quadtree, which stores points, not MBRs, while the R-tree is designed to store MBRs. The R-tree and its variants are the most well-known and commonly used storage techniques (Azri et al. 2013) and are the basis of several real systems Rigaux, Scholl, and Voisard 2002, Section 6.1.3. A second advantage is that the R-tree is a *data-driven structure* Section 6.1.3. These structures partition the space into rectangular areas by following the distribution of the objects, which makes the partition a better bound on the positions the objects will have up to the next snapshot. In contrast, *space driven structures* Section 6.2 like the region quadtree partition the space independently of the indexed

objects, which results in poorer bounds.

6.1. *Structure*

The new snapshot is then an R-tree storing, at its leaves, the object identifiers and their MBRs, as described. The internal nodes store the MBRs of the descendant MBRs.

Note that the snapshot does not contain the precise positions of the objects at its time instant τ. Since the log of GraCT does not store those positions, we add the necessary structures to the snapshots in GraCT. For each snapshot, we store a bitmap B whose size is the number of objects. We then have $B[id] = 1$ if object id appears in the snapshot. The cell coordinates are stored in the same order in two arrays, X and Y, so that the position of id is $(X[rank_1(B, id)], Y[rank_1(B, id)])$.

Figure 9 shows an example of a snapshot based on an R-tree. The left part represents the positions of the objects at τ and their trajectories during $[\tau, \tau + \delta - 1]$. The right part illustrates the structure of the snapshot: the R-tree, the bitmap B, and the arrays X and Y. To obtain the objects that can be within the region delimited by the dashed line during $[\tau, \tau + \delta - 1]$, the algorithm traverses the tree following the nodes whose MBRs intersect the queried region. In the first level, it only checks R_3, and then its children. From those children, R_8 intersects the region and includes the object with id 3. To obtain the location of the object at τ, we just have to compute $(X[rank_1(B, 3)], Y[rank_1(B, 3)]) = (10, 13)$.

Since every R-tree node stores the MBR that wraps the trajectories of the descendant objects, we can now prioritize the nodes with respect to a KNN query (trajectory or point) using the distances from the node MBR to the queried object (trajectory MBR or point). In the leaves of the R-tree, each object stores the MBR of its trajectory during $[\tau, \tau+\delta-1]$. Only when the individual MBR of an object is extracted from Q_{global} we run *init* to compute a more precise trajectory in $[t_b, t_e]$. All these explicitly stored MBRs allow the algorithm compute a more accurate minimum and maximum distance between the area of a node or object and a trajectory or point.

This new type of snapshot not only improves *nearest neighbour queries*. The k^2-tree based snapshots also need to expand, using the maximum speed, the queried regions R of classical *spatio-temporal* queries, in order to determine if a k^2-tree node must be inspected or not. This produces more candidate objects than with R-trees, which know the precise MBR of the objects descending from each node. Precisely, if any object under an R-tree node are within R at any $t \in [\tau, \tau+\delta-1]$, the node MBR must intersect the query region R. Therefore, the algorithm simply runs a classical R-tree traversal following the nodes that intersect R.

In summary, with the snapshots based on R-trees, we can compute the location of an object at the snapshot time instant τ in constant time, and obtain a stricter region to better prioritize or filter the nodes and objects in both *nearest neighbour* and *spatio-temporal* queries. The worst-case time complexities are then unchanged, and the KNN algorithms stay range-optimal (over this new structure), as described in Section 5.3. The next section shows that these advantages turn into orders-of-magnitude improvements in query times in practice.

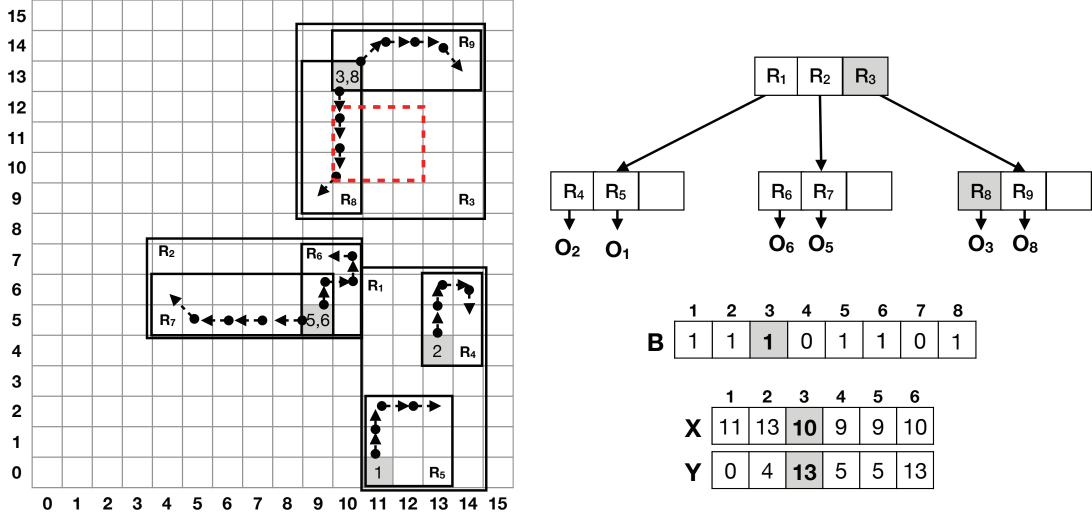


Figure 9.: Example of a snapshot based on an R-tree.

6.2. *Experimental evaluation*

We implemented GraCT and ContaCT with snapshots based on R-trees, in C++, using an existing R-tree implementation (Brisaboa et al. 2013).[5] We represent B as a plain bitmap, its *rank* being supported in $O(1)$ time with a structure that adds 6.25% extra space to the bitmap. Both of them are included in the SDSL library. In figures, GraCT, ContaCT, and ContaCT-SD with this new type of snapshots are labelled with the '-R' suffix.

We built the three structures with different values of δ (30, 60, 120, 240, 360, 720) and used the same datasets presented in Section 5.4. We ran the following queries:

- `ObjectPosition`: We averaged 20,000 different queries where the objects and time instants are chosen randomly.
- `ObjectTrajectory`: We computed a set of 10,000 queries for randomly chosen objects and intervals. The span of the interval was around 2,000 time instants.
- `MBR`: We averaged the time of 1,000 queries for randomly chosen objects and time intervals of 200 time instants.
- `TimeSlice S` and `TimeSlice L`: Both cases included 1,000 queries of a region at a random time instant. In `TimeSlice S` the regions were small (40×40 cells), and in `TimeSlice L`, they were large (320×320 cells).
- `TimeInterval S` and `TimeInterval L`: In the first kind of query, we performed 1,000 queries for small regions (40×40 cells) and short intervals of time (100 time instants). The second type runs the same number of queries with large regions (320×320 cells) and long time intervals (200 time instants).
- `KNN`: We averaged 1,000 queries for random positions at random time instants. The value of K was randomly chosen between 1 and 50.
- `KNNTrajectory` and `KNNInterval`: We averaged each kind of query over 1,000 queries where K is randomly chosen between 1 and 50. The span of the trajectory and the queried time interval is 200 time instants.

5. https://lbd.udc.es/research/serangequerying/

6.2.1. Trajectory queries and space usage

Recall that to compute the position of an object at a given time instant t, GraCT requires $O(\log_k s + \delta + \log n)$ time, where the first term is the cost of traversing the snapshot at the latest time $\tau \leq t$ to obtain the position of the object at time τ. Since the snapshots based on R-trees can compute object positions in constant time, obtaining the position at time t requires time $O(\delta + \log n)$. Instead, in ContaCT the computation of the position does not depend on the snapshot, and it is always constant time.

Figure 10a shows that effect in `ObjectPosition` queries. GraCT-R can solve the query in around 20%–85% of the time required by GraCT. Figures 10b and 10c show that the difference is smaller for `ObjectTrajectory` and `MBR` queries. This is because a larger fraction of the time in those queries is spent in traversing and/or decompressing portions of the log, which takes the same time in both GraCT variants. The difference shrinks when δ increases, because fewer snapshots need to be accessed to cover the same trajectory. While GraCT has a cost of $O(\log_k s)$ for each such snapshot, and thus improves for these queries as it uses less space, GraCT-R and the ContaCT variants only incur a constant overhead per snapshot, so their time is mostly insensitive to δ (GraCT-R improves with shorter logs because it must sequentially scan $\delta/2$ unnecesary movements on average). Since the time interval of `MBR` queries lasts 200 time instants, and `ObjectTrajectory` queries cover 2,000 time instants, the difference of performance between GraCT variants is smaller in `MBR` queries. On the other hand, ContaCT and ContaCT-SD are constant and much faster, because they do not need to traverse any snapshot nor logs, just to compute MBRs in constant time. For example, ContaCT takes 0.3–3.2 microseconds for the `ObjectPosition` and `MBR` queries on all the datasets.

In addition, there is hardly any difference between the compression ratio achieved by the structures with the original snapshot and the new one. The new space usage is around 90%–100% of the structures based on k^2-trees.

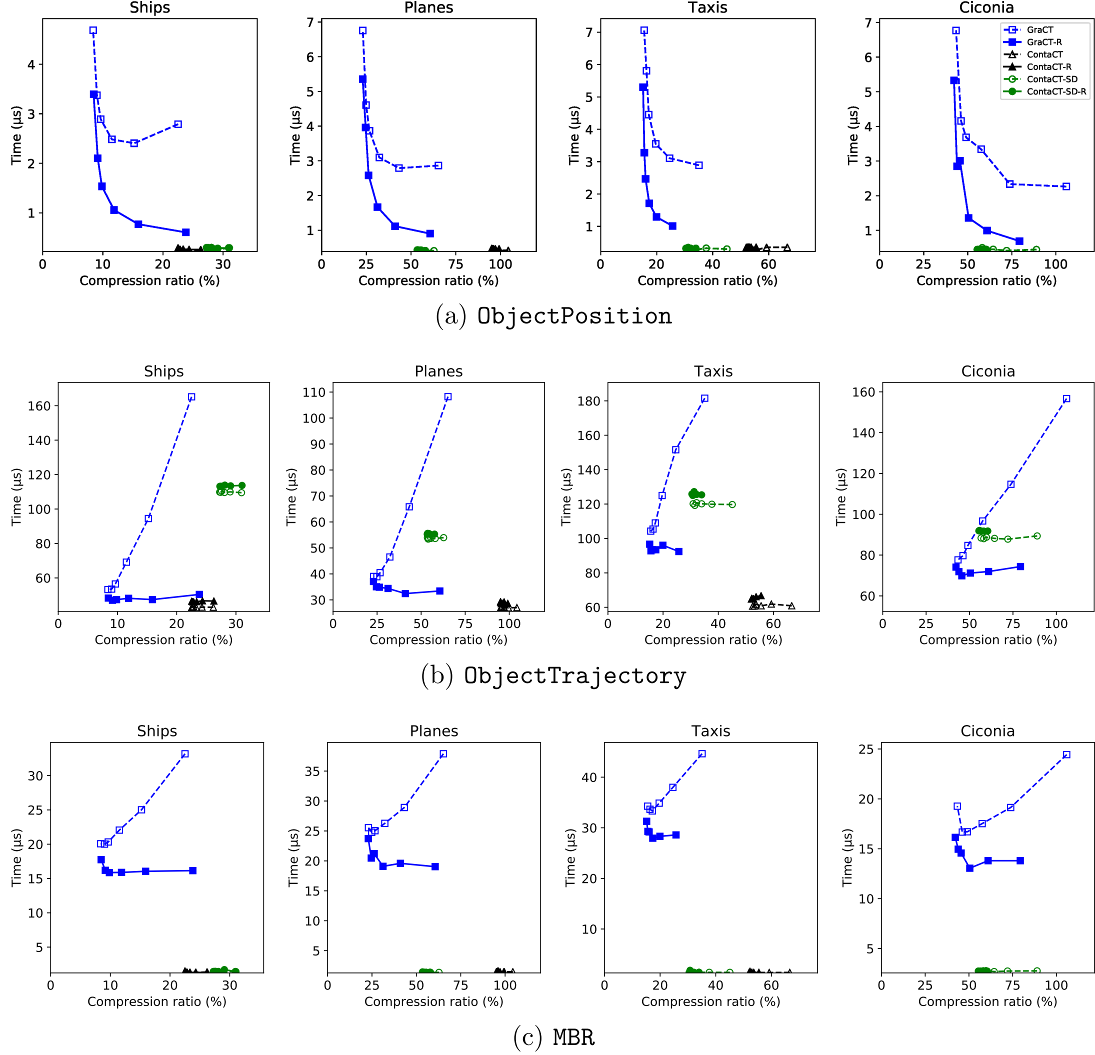
6.2.2. Spatio-temporal queries

Figure 11 shows the average times for the spatio-temporal queries. Figures 11a and 11b show an improvement around 2–30 and 2-140 times in `TimeSlice` with GraCT-R and ContaCT-R compared to GraCT and ContaCT, respectively. Since snapshots based on R-trees obtain a tighter area about where the object is moving, the set of candidate objects is smaller, which reduces the number of positions to check and improves the performance. For the same reason, similar speedups are seen on `TimeInterval` queries, where GraCT-R and ContaCT-R are 2–275 and 2–200 times faster than GraCT and ContaCT, respectively. The fastest configurations reach 12–80 microseconds in GraCT-R, 13–150 in ContaCT-R, and 12–160 in ContaCT-SD-R.

6.2.3. Nearest neighbour queries

Figure 12 shows that the R-tree based snapshots also run faster on `KNN` queries. The effect is more noticeable on the larger datasets: the R-trees are 7–16 times faster on `Ships`, and 1.3–3.3 times faster on the others.

The improvements are larger for the more complex queries, `KNNTrajectory` and `KNNInterval`. For example, in `Ships` and `Planes`, both queries are 10–60 and 2.3–40 times faster, respectively, using R-trees. In the rest of the datasets, R-trees perform 1.1–3.0 times faster. The datasets with more objects display better improvements, showing the ability of the snapshots based on R-trees to better prioritize the objects.



(a) ObjectPosition

(b) ObjectTrajectory

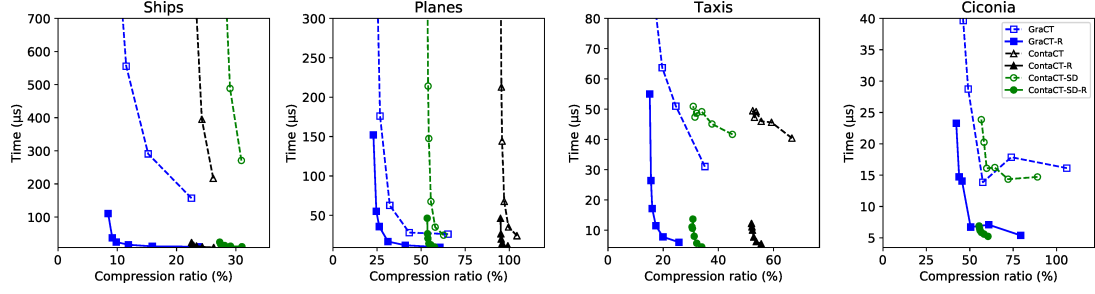
(c) MBR

Figure 10.: Space and time for trajectory queries.

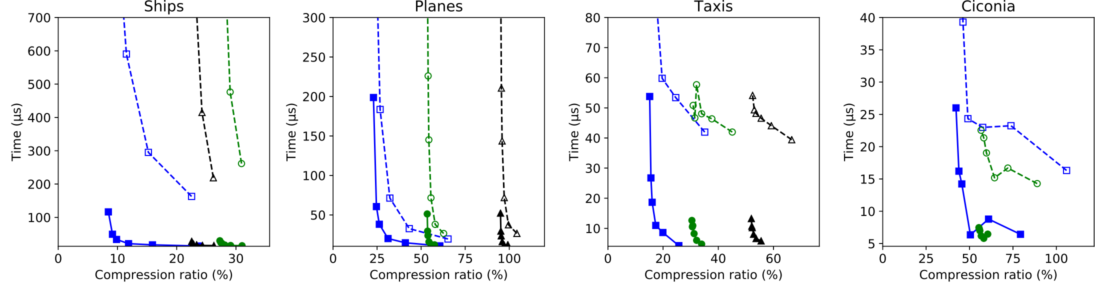
The fastest configurations reach 230–2,100 microseconds in GraCT-R, 190–900 in ContaCT-R, and 190–930 in ContaCT-SD-R.

6.3. *Comparison with a spatio-temporal index*

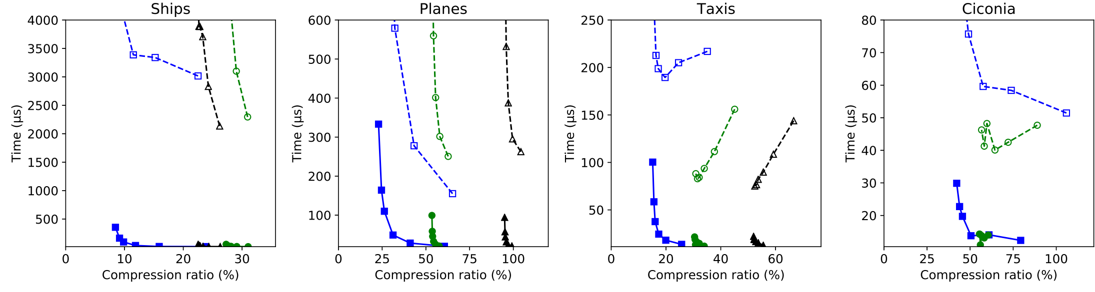
In this section, we compare ContaCT-R and GraCT-R with the MVR-Tree (Tao and Papadias 2001), a classical spatio-temporal index that uses a set of R-trees along time. Each R-tree is called a *version* and stores the MBR of the objects during an interval of time. Note that consecutive versions can be quite similar. To save space usage, when two consecutive R-trees share a subtree, that subtree in the second R-tree points to the first subtree. Since its basis is the R-tree, the MVR-Tree is designed for solving `TimeSlice`, `TimeInterval`, `KNN`, and `KNNInterval` queries. For supporting `ObjectPosition` and `ObjectTrajectory`, it would require traversing all the nodes of the versions that intersect with the queried interval of time. Further, for `KNNTrajectory` it has no efficient mechanism to compute the distance between the information of a node and the trajectory.



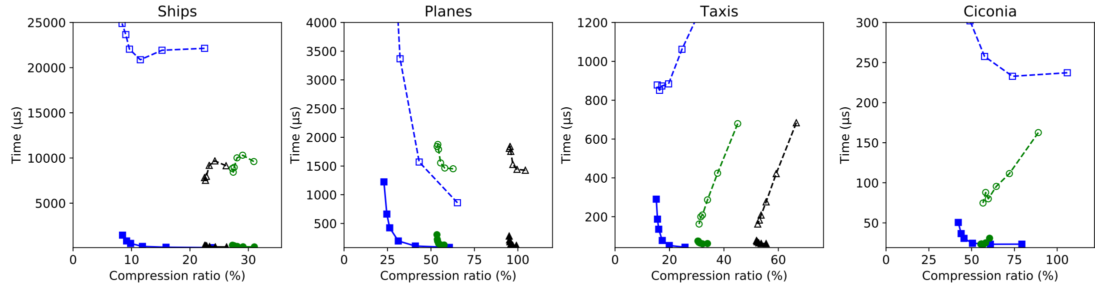
(a) `TimeSlice S`



(b) `TimeSlice L`



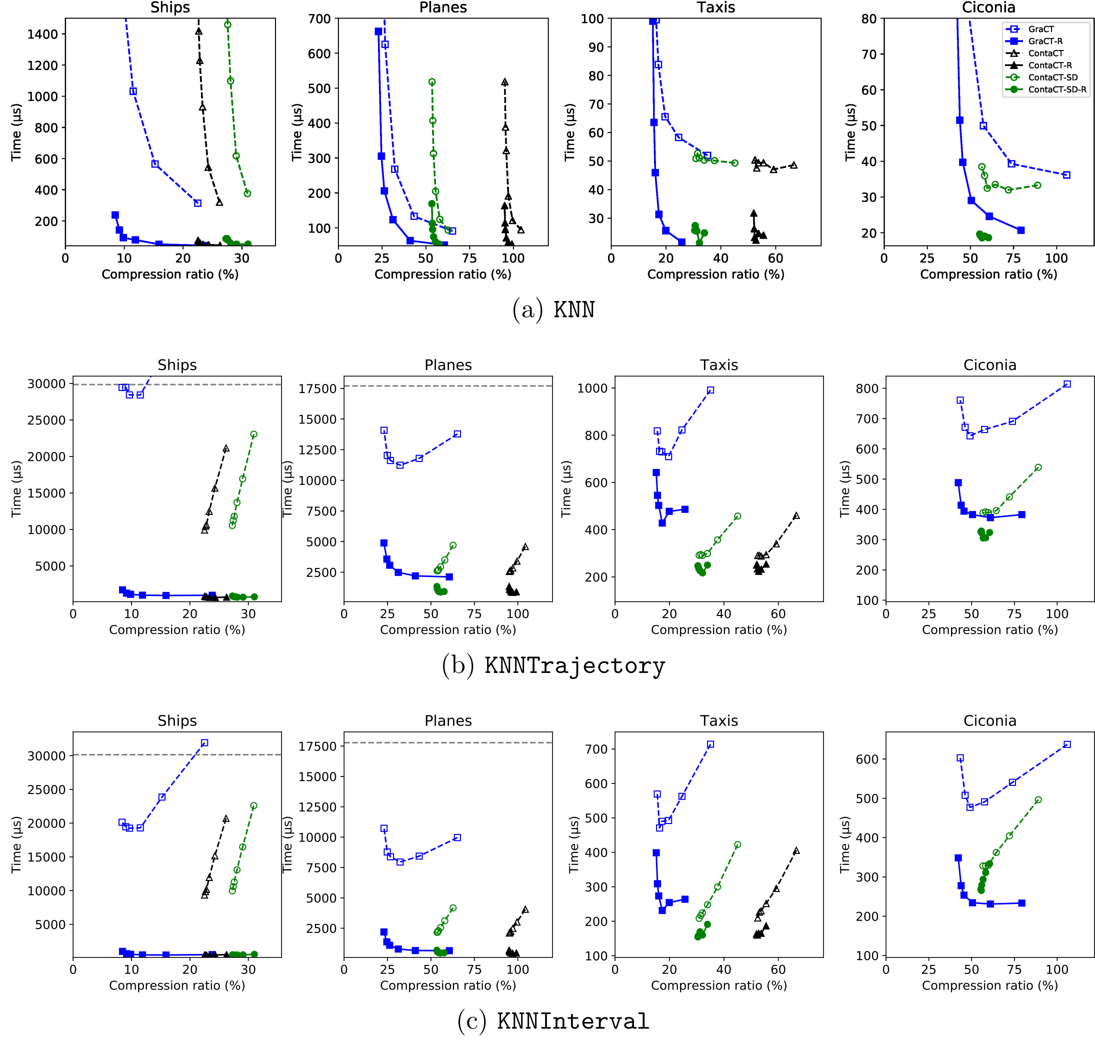
(c) `TimeInterval S`



(d) `TimeInterval L`

Figure 11.: Space and time for spatio-temporal queries.

Therefore, we compared ContaCT-R and GraCT-R with the MVR-tree in `TimeSlice`, `TimeInterval`, `KNN`, and `KNNInterval` queries on the two datasets with the most objects: `Ships` and `Planes`. The configuration of the queries are identical to those presented before. We set $\delta = 120$ on ContaCT-R and GraCT-R, with the variant of plain bitmaps in `Ships`, and sparse bitmaps in `Planes`. We used the MVR-tree implemented in the C++ `spatialindex` library with default parameters (the capacity



(a) KNN

(b) KNNTrajectory

(c) KNNInterval

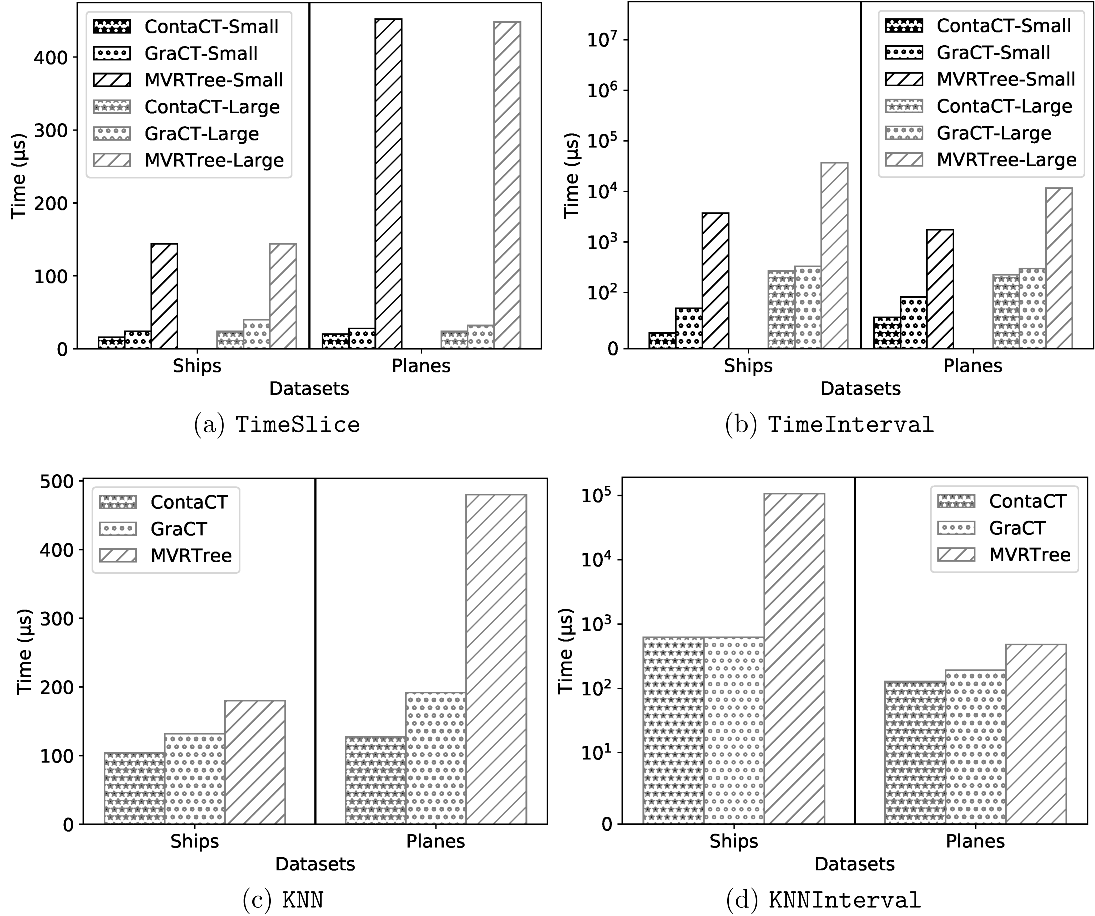
Figure 12.: Space and time for nearest neighbour queries.

of each node set to 10 records and the fill factor set to 70%).[6] For a fair comparison, we load the MVR-tree into main memory, thus avoiding any disk access at query time.

The average time of each query is shown in Figure 13. The MVR-tree obtains its best results in `TimeSlice` and `KNN` queries, because it needs to traverse only one *version*, even so, GraCT-R and ContaCT-R are faster. On `Ships`, ContaCT-R and GraCT-R are 3–9 and 1.3–1.7 times faster for `TimeSlice` and `KNN` queries, respectively. The differences are more remarkable on `Planes`: 14–22 times faster for `TimeSlice`, and 2.5–3.8 for `KNN`.

Queries `TimeInterval` and `KNNInterval` cover a larger interval of time and the MVR-tree has to check more *versions*, thus the differences between the compact data structures and MVR-tree grow more sharply (we use logscales). On `Planes`, for example, GraCT-R and ContaCT-R can solve `TimeInterval` queries in 50–300 microseconds, whereas the MVR-tree needs 1.7–11.6 milliseconds. The smallest difference occurs for `KNNInterval` on `Planes`, where GraCT-R and ContaCT-R are still 2.5 and 3.7 times faster than the MVR-tree, respectively.

6. `http://libspatialindex.github.io`



(a) TimeSlice (b) TimeInterval

(c) KNN (d) KNNInterval

Figure 13.: Comparison with the MVR-tree. Note the logscale on the right plots.

With respect to space usage, the MVR-trees on Ships and on Planes require 12.16 GB and 11.72 GB, respectively. The sizes of GraCT-R and ContaCT-R indexes, even using a large-space configuration ($\delta = 120$), are around 100 times smaller.

6.4. *Discussion*

In our evaluation, we observe that both kinds of snapshots require similar space. The most remarkable differences are in time performance on spatio-temporal and nearest neighbour queries. To solve those queries, the algorithm uses the snapshots to obtain a rough idea of the area where the object moves during the queried interval. According to that area, the objects are selected or prioritized, depending on the query. Snapshots based on R-trees can retrieve that area as the MBR where the object is moving between two snapshots. Snapshots based on k^2-trees, instead, compute that area assuming that every object moves at the maximum speed of the fastest object. That assumption enlarges the area where an object is moving, which is more noticeable when the number of objects is larger. The tighter area obtained from the snapshots based on R-trees allows us to get a better and smaller set of candidates, thereby improving the time performance by orders of magnitude.

Concerning trajectory queries, snapshots only affect GraCT. In that structure, the algorithms of trajectory queries need to obtain the positions of the objects from the

snapshots. In snapshots based on R-trees, those positions are directly stored in an array. Hence, they are retrieved in constant time, avoiding the top-down traversal of the snapshots based on k^2-trees. For this reason, GraCT-R outperforms the time performance of GraCT in this particular kind of queries.

In summary, with R-tree based snapshots, we use about the same space of k^2-tree based snapshots, but the time performance is greatly improved on all queries. In fact, this new type of snapshot makes GraCT and ContaCT faster than the MVR tree, while using 100 times less space.

7. Relative compression of trajectories

The preceding evaluation shows that, while ContaCT is considerably faster than GraCT in several queries, the latter index is generally much smaller: GraCT uses 40%–80% of the space of ContaCT.

GraCT exploits the repetitiveness of the datasets, whereas ContaCT only takes advantage of the fact that most movements are small. In larger datasets, the repetitiveness of the movements can play an important role to reduce the space. GraCT exploits repetitiveness using grammar compression (Kieffer and Yang 2000) of the trajectories, as explained in Section 3.7.1, and this induces a certain overhead when accessing trajectories at random positions. In this section we explore instead Relative Lempel-Ziv (RLZ) (Kuruppu, Puglisi, and Zobel 2010), another compressor for highly repetitive sequences that enables fast random access to them.

Our new index, *Relative Compression of Trajectories (RelaCT)*, adapts ContaCT to highly repetitive datasets by compressing the trajectories using RLZ. With this structure, we achieve a space usage closer to that of GraCT, and a time performance similar to that of ContaCT, which leads to a good space-time tradeoff.

7.1. *Structure*

As said, the RelaCT index builds on Relative Lempel-Ziv (RLZ) (Kuruppu, Puglisi, and Zobel 2010). A special trajectory R called the *reference* is created and saved in plain form (this can be one of the trajectories in the dataset, a concatenation of parts of those, a synthetic sequence, etc.). Each trajectory in the dataset is then represented as the concatenation of z *phrases*: $w_1 w_2 \dots w_z$. Each phrase w_i is represented as a pair of integers (p_i, l_i), where p_i is a position in the reference and l_i is the length of the phrase, that is, $w_i = R[p_i..p_i + l_i - 1]$. The RLZ algorithm generates the phrases in greedy form, maximizing l_i at each step; recall Section 3.2.

7.1.1. The reference

In RelaCT, the reference R is an artificial trajectory built by concatenating some of the real trajectories. Note that we regard each trajectory as a sequence of movements, that is, of relative displacements. To build R, we copy the first trajectory to it and then, for each new trajectory $S[1..n]$:

(1) We compute the number of phrases z obtained from applying the RLZ algorithm to S, with respect to the current reference R.
(2) If the fraction z/n is below a parameter $0 < \alpha < 1$, the new trajectory S is well represented by the reference R. Otherwise, we append S to R.

We note that the trajectories that are included in R can be represented by a single phrase, pointing to their position in R. On the other hand, we represent the reference R using the ContaCT structure. This allows us compute any cumulative displacement, as well as relative MBRs, on R in constant time.

7.1.2. Log representation

The obtained reference R is global for the whole RelaCT index. Once R is defined with the process above, the log $S_{id}[1..n]$ of relative movements of every object id between two snapshots is compressed by applying RLZ on S_{id} with respect to R. This results in a sequence of z pairs (p_i, l_i) representing the substrings w_i of R that make up S_{id}.

The pointers p_i are concatenated into an array $P_{id}[1..z]$, whereas the lengths l_i are represented by marking with 1s in a bitmap $L_{id}[1..n]$ the starting position of each phrase w_i in S_{id}, so that $l_i = select_1(L_{id}, i) - select_1(L_{id}, i-1)$. In addition, the log stores the initial position of the trajectory and its time instant as $F_{id} = \langle (x_{id}, y_{id}), t_{id} \rangle$. Finally, two arrays $X_{id}[1..z]$ and $Y_{id}[1..z]$ store the cumulative movement from the beginning of the trajectory until the end of each phrase. Figure 14 shows an example.

To compute the position of object id at time instant t_q, we find the phrase that contains t_q with $j = rank_1(L, t_q - t_{id})$. The cumulative movement until the beginning of that phrase is $(X_{id}[j-1], Y_{id}[j-1])$. Since the jth phrase starts at time instant $t = t_{id} + select_1(L_{id}, j)$, we have computed the cumulative movement until $t-1$.

We now have to add the cumulative movement from t to t_q. This is obtained from $R[P_{id}[j]..P_{id}[j]+t-t_q]$, since that substring of the reference is equal to the one we are querying. That sum of movements can be computed in constant time in the reference as $\Delta(P_{id}[j] + t_q - t) - \Delta(P_{id}[j] - 1)$, where

$$\Delta(i) = (select_1(X_p, i) - select_1(X_n, i), select_1(Y_p, i) - select_1(Y_n, i)),$$

is computed with the structures defined by ContaCT on R; recall Section 3.7.2.

By both results we obtain the total displacement of the object from the beginning to t_q. The location of the object at t_q is computed by adding that displacement to (x_{id}, y_{id}):

$$(x_{id}, y_{id}) + (X_{id}[j-1], Y_{id}[j-1]) + \Delta(P_{id}[j] + t_q - t) - \Delta(P_{id}[j] - 1).$$

Figure 14 shows how the cumulative movement is computed for $t_q = t_{10}$. This is the 8th movement of the trajectory because $t_q - t_{id} = 8$. That movement lies on the second phrase because $rank_1(L, 8) = 2$. This phrase starts at the time instant $t = t_2 + select_1(L, 8) = t_8$. Therefore the cumulative movement from the beginning of the trajectory until t_7 is $(5, 2)$. The remaining displacement from t_8 to t_{10} is computed as $\Delta(10) - \Delta(7) = (3, 2)$. Finally, the cumulative movement position until t_{10} is obtained as $(5, 2) + (3, 2) = (8, 4)$. This value added to $(x_{id}, y_{id}) = (1, 3)$ results in the position of the object at t_{10}, $(1, 3) + (8, 4) = (9, 7)$.

7.1.3. Absence of information

In many cases, we lack information about the location of an object at some time instants, for different reasons (e.g., precision errors, low GPS signal, GPS device not working). We need a mechanism to represent that absence of information. Just as for GraCT and ContaCT, in RelaCT we use a bitmap $M_{id}[1..n]$ per log, setting $M_{id}[i] = 1$

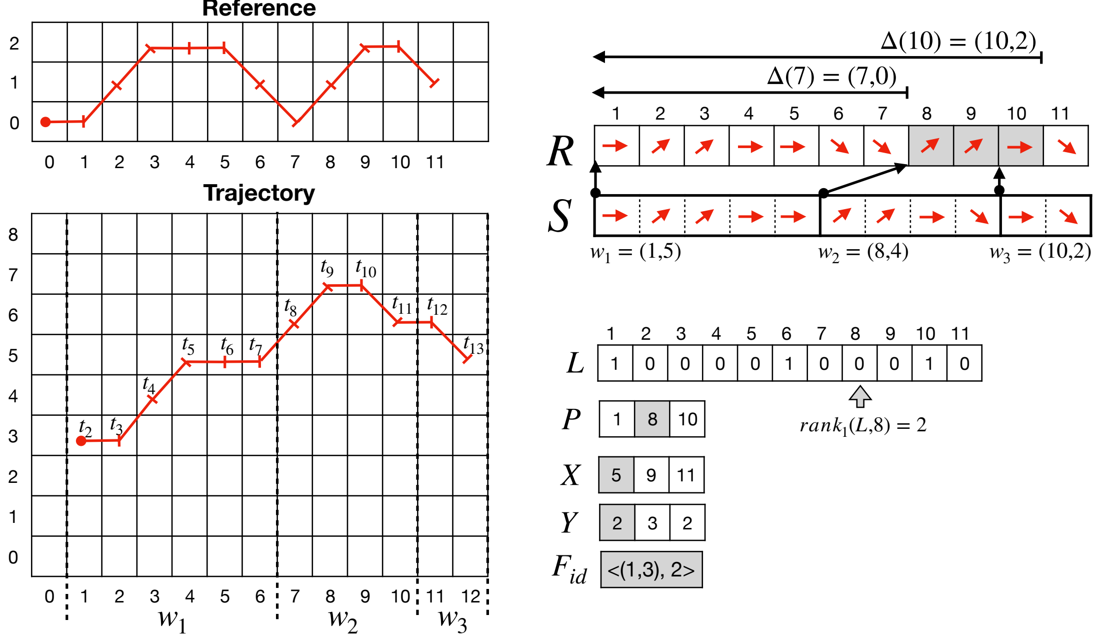


Figure 14.: Log structure for RelaCT with a reference and its corresponding trajectory.

Algorithm 2: ObjectPosition$(id,\ t_q)$

---

if $t_q = t_{id}$ **then return** (x_{id}, y_{id});
$j \leftarrow rank_1(L_{id}, t_q)$;
$p \leftarrow P_{id}[j]$;
$p' \leftarrow p + (t_q - select_1(L_{id}, j))$
$dx \leftarrow \Delta(p').x - \Delta(p-1).x$;
$dy \leftarrow \Delta(p').y - \Delta(p-1).y$;
return $(x_{id} + X_{id}[j-1] + dx, y_{id} + Y_{id}[j-1] + dy)$;

---

when there is data about the location of the object at $t_{id} + i$, or else $M_{id}[i] = 0$.

The mechanism for obtaining the location of an object works similarly with this bitmap, but instead of working with time instants, it uses movements. We compute the movement corresponding to time instant t_q as $m_q = rank_1(M_{id}, t_q - t_{id})$. A given movement m is mapped back to its time with $t = t_{id} + select_1(M_{id}, m)$. We will ignore this mapping for simplicity in the sequel, still using t_q instead of m_q in the descriptions.

7.2. *Queries*

We now describe how the queries are handled with this data structure. The first three simple queries take constant time. As a consequence, the complexities of the following, more complex, queries is identical to those of ContaCT.

7.2.1. Object position

Algorithm 2 summarizes the constant-time procedure we have explained to retrieve the position of an object at a given time instant, considering absence of information.

Algorithm 3: ObjectTrajectory(id, t_b, t_e)

```
result ← ∅;
if t_id ≥ t_b then  result ← result ∪ ⟨t_id, x_id, y_id⟩ ;
t ← max(t_b − t_id, 1); j ← rank_1(L_id, t);
t_b ← select_1(L_id, j);
t_e ← select_1(L_id, j + 1) − 1;
p ← P_id[j];
Δ_pre ← Δ(p − 1);
while t_id + t ≤ t_e do
    dx ← Δ(p + t − t_b).x − Δ_pre.x;
    dy ← Δ(p + t − t_b).y − Δ_pre.y;
    result ← result ∪ ⟨t_id + t, x_id + X_id[j − 1] + dx, y_id + Y_id[j − 1] + dy⟩;
    t ← t + 1;
    if t = t_e + 1 then
        j ← j + 1; p ← P_id[j];
        Δ_pre ← Δ(p − 1);
        t_b ← t_e + 1;
        t_e ← select_1(L_id, j + 1) − 1;
return result;
```

Algorithm 4: MBR(id, t_b, t_e)

```
mx ← rmq_X(t_b, t_e); my ← rmq_Y(t_b, t_e);
Mx ← rMq_X(t_b, t_e); My ← rMq_Y(t_b, t_e);
min_x ← Best(id, {mx, t_b, t_e}, X, x, min); min_y ← Best(id, {my, t_b, t_e}, Y, y, min);
max_x ← Best(id, {Mx, t_b, t_e}, X, x, max); max_y ← Best(id, {My, t_b, t_e}, Y, y, max);
return [min_x, min_y] × [max_x, max_y];
```

7.2.2. Object trajectory

Algorithm 3 solves this query in optimal time, that is, constant per retrieved position. It is, in practice, more efficient than querying object positions one by one. The algorithm proceeds by phrases. Lines 3–7 compute the phrase number i, the time interval $[t_b, t_e]$ it spans, and the starting object position in the reference, p, and in space, Δ_{pre}. Then it adds the points of this phrase to the trajectory in lines 9–12. Line 13 checks if we have completed the phrase, in which case the data of the next phrase, $i + 1$, is computed in lines 14–17 before continuing.

7.2.3. Minimum bounding rectangle

To compute the MBR that covers the trajectory of an object between two time instants t_b and t_e, we use the structure of ContaCT that computes the position of the local minimum and maximum in constant time using at most $3n + o(n)$ bits per coordinate, where n is the length of the log. We call those operations rmq_D and rMq_D, for $D \in \{X, Y\}$. To compute the final results, we must compare those local extremes with the values at the endpoints of the queried time interval. To obtain those values to compare, we use the same procedure of the *ObjectPosition* query.

Algorithm 4 shows the pseudocode for this constant-time query. The minima and maxima of the local extremes, for both coordinates, are computed in mx, my, Mx, and My. The comparison with the endpoints is done by *Best*, which is depicted in Algorithm 5.

More than an algorithm, *Best* should be regarded as a macro to avoid writing similar code 4 times. It receives in *pos* a list of positions to compare, in D/d the coordinate (X/x or Y/y), and in Op what to take from the values (minimum or maximum).

Algorithm 5: Best(id, pos, D, d, Op)

```
values ← ∅;
for m ∈ pos do
    i ← rank_1(L_id, m);
    p ← P_id[i];
    p' ← p + (m − select_1(L_id));
    values ← values ∪ {d_id + D_id[i − 1] + Δ(p').d − Δ(p − 1).d};
return Op(values);
```

Algorithm 6: TimeSlice(R, t_q)

```
τ ← ⌊t_q/δ⌋ × δ;
candidates ← S_τ.intersect(R);
result ← ∅;
for id ∈ candidates do
    p ← ObjectPosition(id, t_q);
    if p ∈ R then result ← result ∪ {id} ;
return result;
```

Algorithm 7: TimeInterval(R, t_b, t_e)

```
result ← ∅; checked ← ∅;
τ_b ← ⌊t_b/δ⌋; τ_e ← ⌈t_e/δ⌉;
for i ∈ [τ_b, τ_e] do
    τ ← i × δ;
    candidates ← S_τ.intersect(R);
    for id ∈ candidates do
        if id ∉ checked then
            if Contained(id, R, t_b, t_e) then result ← result ∪ {id} ;
            checked ← checked ∪ {id};
return result
```

Algorithm 8: Contained(id, R, t_b, t_e)

```
if t_e − t_b < λ then
    T ← ObjectTrajectory(id, t_b, t_e)
    for ⟨t, p⟩ ∈ T do
        if p ∈ R then return true;
    return false
else
    mbr ← MBR(id, t_b, t_e);
    if mbr ⊆ R then return true ;
    if mbr ∩ R = ∅ then return false ;
    t_m ← t_b + ⌊(t_e − t_b)/2⌋
    return (Contained(id, R, t_b, t_m) or Contained(id, R, t_m + 1, t_e))
```

7.2.4. Time Slice

We retrieve the objects that are within a region R at a time instant t_q by computing their position at t_q with *ObjectPosition*, and checking if they are within R. We use the preceding snapshot, at time $\tau \leq t_q < \tau + \delta$, to find the candidates that have chances of being within R at time t_q, namely those whose MBR in the snapshot (which covers their positions in $[\tau, \tau + \delta - 1]$) intersects R. Algorithm 6 shows the pseudocode.

Algorithm 9: Knn(K, p_q, t_q)

```
Q_c ← ∅; Q_r ← ∅, capped to size K;
τ ← ⌊t_q/δ⌋ × δ;
Q_c.add(⟨S_τ.root, 0, +∞⟩);
while Q_c ≠ ∅ and (|Q_r| < K or Q_c.min < Q_r.max) do
    ⟨e, l, h⟩ ← Q_c.pop();
    if e is an internal node then
        for node ∈ e.children do
            ⟨l_new, h_new⟩ ← distances(node.R, p_q);
            Q_c.add(⟨node, l_new, h_new⟩);
    else
        for id ∈ node.objects do
            p_id ← ObjectPosition(p_id, t_q);
            Q_r.add(⟨id, d(p_id, p_q)⟩);
return Q_r;
```

7.2.5. *Time Interval*

Retrieving the objects that are within a region R at any time instant of the interval $[t_b, t_e]$ can be solved similarly to *Time Slice*. As seen in Algorithm 7, we take the snapshots covering any time instant in $[t_b, t_e]$ and search them looking for the candidates (lines 2–5). For each candidate, the algorithm checks if it is contained within R at any time instant of $[t_b, t_e]$. The set *checked* is used to avoid checking several times the objects that appear in more than one snapshot.

An object is checked to be contained in R during $[t_b, t_e]$ in Algorithm 8. Lines 7–9 compute the object's MBR and checks for two immediate inclusion-exclusion conditions (MBR contained in or disjoint with R). If those are not met, the time interval is partitioned in two halves and those are recursively checked in line 11. Lines 1–5 handle short enough intervals, defined by a parameter λ, by directly obtaining the object's trajectory and checking its positions one by one.

7.2.6. *Nearest neighbour queries*

To obtain the K objects closest to a point p_q at a time instant t_q, we proceed as in Algorithm 9. The algorithm takes the snapshot $\mathcal{S}_\tau$ that covers the time interval $[\tau, \tau + \delta - 1]$ containing t_q, and traverses its R-tree nodes according to their proximity to the point p_q. The leaf nodes with the objects that have more chances to be closer to p_q are then reached earlier. We use a min-heap priority queue Q_c to prioritize those nodes, according to the minimum distance l to R, and breaking ties with the maximum distance h. In every iteration the algorithm takes the element on top of Q_c. If it is an internal R-tree node, its children are reinserted to Q_c (lines 6–9). If, instead, it is a leaf, lines 11–13 insert its objects prioritized by their precise distance to p_q at time t_q, into a min-heap Q_r capped to size K.

The algorithm repeats those steps until there are K elements in Q_r and there is no element in Q_c that can improve the distance of the K-th element of Q_r with respect to p_q. That is, the object on top of Q_c has an l value larger than the distance of the last element on Q_r (line 4). The result is the set of K elements of Q_r.

7.2.7. Complex nearest neighbour queries

For *KNN of trajectories* and *KNN during an interval*, the algorithms of RelaCT are completely identical to those presented in Section 5.1. Recall that there are two stages: *prioritizing the object by using snapshots* and *refining the object distance bounds*. For the first one, the algorithm traverses the nodes of the R-tree as in GraCT and ContaCT, prioritizing the nodes according to their distances to the input data (trajectory or point) in constant time. In the second stage, since RelaCT can compute the MBR between two time instants just as in ContaCT, we refine the bounds of the object distance by applying the same algorithm of ContaCT.

7.3. *Speeding up queries*

We can improve the time performance of the queries by adding to each log an additional structure that stores the minimum and maximum value within each phrase for each axis $D \in \{X, Y\}$: $D_m[1..z]$ and $D_M[1..z]$, respectively. A range minimum query structure rmq_D, using $2z + o(z)$ bits (Fischer and Heun 2011; Ferrada and Navarro 2017), is added on D_m, and an analogous range maximum query rMq_D is added on D_M. In total, the extra structures use $4z \log s + 8z + o(z)$ bits, where s is the size of the represented two-dimensional space. Those structures replace the original bitvectors rmq_D and rMq_D of the basic RelaCT.

With these additional structures, we can compute the MBR of the phrases that are completely contained in a time interval $[t_b, t_e]$ without accessing the reference. More precisely, the MBR of the phrases $w_i \ldots w_j$ is

$$[X_m[rmq(X_m, i, j)], Y_m[rmq(Y_m, i, j)]] \times$$
$$[X_M[rMq(X_M, i, j)], Y_M[rMq(Y_M, i, j)]].$$

While these structures do not affect the worst-case complexities of the queries, they can be used to improve the practical performance of *MBR* and *Time Interval* queries.

7.3.1. Minimum Bounding Rectangle

To compute *MBR* on the interval $[t_b, t_e]$, let us assume that $w_i \ldots w_j$ are the phrases completely contained in the queried time interval. The MBR of those whole phrases, MBR_C, can be computed as shown above. The result of the query will be MBR_C, except when the phrases that are not completely contained but intersect $[t_b, t_e]$ can change it. We compute the MBRs of those (whole) phrases, w_{i-1} and w_{j+1}, as

$$\begin{aligned} MBR_{i-1} &= [X_m[i-1], Y_m[i-1]] \times [X_M[i-1], Y_M[i-1]] \\ MBR_{j+1} &= [X_m[j+1], Y_m[j+1]] \times [X_M[j+1], Y_M[j+1]]. \end{aligned}$$

If MBR_{i-1} and MBR_{j+1} are completely contained within MBR_C, then there is nothing else to do. Otherwise, the part of the trajectory not covered by $w_i \ldots w_j$ might enlarge MBR_C. Thus, if $MBR_{i-1} \not\subseteq MBR_C$, we compute *MBR* on the interval $[t_b, t'_e] \subseteq [t_b, t_e]$ that overlaps the phrase w_{i-1}, using the reference as we explained in Section 7.2.3, and enlarge MBR_C so as to contain that MBR. We handle MBR_{j+1} analogously. After those adjustments, MBR_C is the answer to the query.

Figure 15 illustrates an example, where we compute *MBR* from time instant t_8 to t_{41}. Here, MBR_C covers the phrases $w_3 w_4 w_5 w_6$ and the time interval $[t_{14}, t_{36}]$. It is

computed using the arrays X_m, Y_m, X_M, and Y_M, and their *rmq* and *rMq* structures. The values pointed by the range minimum and maximum structures are marked in bold. We obtain $MBR_C = [2,0] \times [7,6]$. The MBR of the phrases at the extremes (w_2 and w_7) are computed by directly accessing the arrays of minima and maxima. Hence, we have $MBR_2 = [2,1] \times [4,2]$ and $MBR_7 = [1,2] \times [5,5]$. Since MBR_7 is completely within MBR_C but MBR_2 is not, we have to compute on the reference the MBR between time instants t_8 and t_{13}. That MBR is $[2,1] \times [2,2]$, which is completely contained in MBR_C. Therefore, our answer is MBR_C.

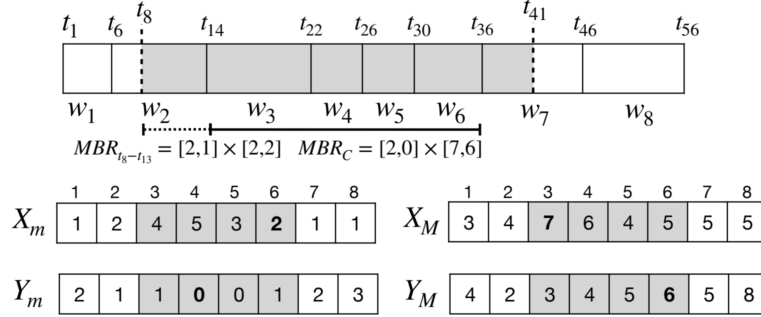


Figure 15.: Example of computing the minimum bounding rectangle between two time instants.

This approach avoids as much as possible to use the reference, which speeds up the *MBR* query. This also improves the performance of several other queries that directly use the *MBR* query, such as the complex nearest neighbour queries.

7.3.2. Time Interval

We now exploit the fact that we can obtain the MBR of a sequence of phrases very quickly to speed up *Time Interval* queries, which make intensive use of *MBR* queries. Concretely, for each candidate, this query performs a binary search with such queries.

We go further than merely exploiting the faster *MBR* algorithm obtained in the previous section. We maintain the binary search, but work on whole phrases as much as possible, because computing their MBR is faster. We exploit the fact that, if the MBR of a sequence of phrases overlapping $[t_b, t_e]$ is contained in R, then the object qualifies.

Let $w_{i-1} \dots w_{j+1}$ the phrases that minimally contain $[t_b, t_e]$. Our algorithm computes the MBR of the current interval of phrases without resorting to the reference. If it is contained in R, the object is added to the output and we finish. Otherwise, the interval of phrases $[i-1..j+1]$ is halved (into whole references) and we continue recursively by each subinterval. When the interval is formed by a single phrase whose MBR is not contained in R, we resort to the previous procedure with the time interval of the phrase. Note also that, in the binary search of phrase sequences, we can also abort the branches where the MBR is disjoint from R.

Figure 16 shows how we check if an object is contained within $R = [5,5] \times [5,5]$ during the interval $[t_8, t_{41}]$. Those time instants are contained in the phrases $w_2w_3w_4w_5w_6w_7$. The algorithm starts computing the MBR covered by all those phrases, MBR_{2-7}. As it intersects the queried region, we split it into MBR_{2-4} and MBR_{5-7}, which cover $w_2w_3w_4$ and $w_5w_6w_7$, respectively. Since MBR_{2-4} does not intersect R, we stop splitting it, and continue recursively with MBR_{5-7}. We continue

in this way until reaching the interval of the phrase w_7, which covers the time interval $[t_{36}, t_{41}]$. Since it intersects R, it is partitioned in two halves: $MBR_{t_{36}-t_{38}}$ and $MBR_{t_{39}-t_{41}}$. Since $MBR_{t_{36}-t_{38}}$ is completely contained in R, the algorithm stops and the object is added to the solution.

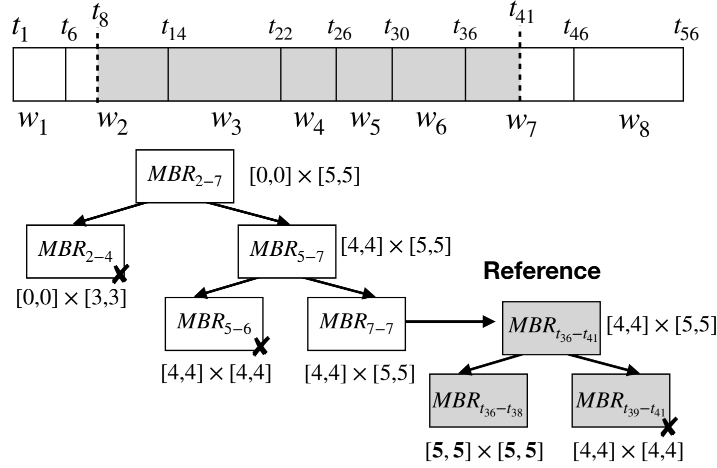


Figure 16.: Simulation of the procedure to detect if an object is within a region during an interval of time.

7.4. *Experimental evaluation*

We experimentally compare RelaCT with GraCT and ContaCT. The evaluation focuses on larger datasets, which are more repetitive, since RelaCT shows little advantage on the smaller ones. Since the snapshots based on R-trees have shown much better performance than the original ones, we only test those snapshots.

7.4.1. Large datasets

We build larger variants of the original datasets of `Ships` and `Taxis`, with the same format:

- `ShipsLarge`: a real dataset from *MarineCadastre* that contains 19,764 vessels that move around the coast of the USA during the first four months of 2020. As in the first experiment, the raster model of this dataset uses a cell size of $10^{-3} \times 10^{-3}$ degrees, and the frequency with which an object emits its location is normalized to regular intervals of 1 minute.
- `TaxisLarge`: a pseudo-real dataset containing trajectories of 433 taxis in New York City during 2013. As in the first experiment, the dataset only contains the origin and destination of each trip, thus each trajectory was computed as the shortest path between them by taking into account the road network. The original data are available at *NYC Taxis: A Day in the life*. The cell size of the raster model is $10^{-5} \times 10^{-5}$ degrees, and the signals are taken at regular intervals of 15 seconds.

As before, Table 2 shows the dimensions of the datasets, their size in plain form, binary form, and *p7zip*-compressed. The compression ratios of *p7zip* are around 8% and 20% in `ShipsLarge` and `TaxisLarge`, respectively.

	ShipsLarge	TaxisLarge
Total objects	19,765	433
Total points	671,088,660	851,369,612
Max x	1,199,974	1,379,380
Max y	891,731	664,900
Max *time*	262,079	2,102,691
Size Plain	16,838.92 MB	20,480.00 MB
Size Bin	7,040.00 MB	8,931.22 MB
Size *p7zip*	574.69 MB	1,738.86 MB

Table 2.: Large datasets and their dimensions.

7.4.2. Compression

We built GraCT, ContaCT, and RelaCT on both datasets, with δ values 30, 60, 120, 240, 360, and 720. We have variants ContaCT and ContaCT-SD, as in Section 5.4. Since the reference of RelaCT is represented with ContaCT, we have the corresponding configurations RelaCT and RelaCT-SD. In addition, the RelaCT configurations that include the structure for speeding up the queries (Section 7.3) are suffixed with '+'.

Figure 17 shows the size of those structures and the compression ratios. In both datasets, GraCT obtains the best compression ratio. In ContaCT and RelaCT, the variants using sparse bitmaps are the smallest ones.

GraCT is still the only structure that outperforms the compression ratios of *p7zip*, whereas ContaCT-SD uses around 3.7 and 1.6 times more space than *p7zip* in ShipsLarge and TaxisLarge, respectively. RelaCT-SD and RelaCT-SD+ are between both. For example, the configuration of GraCT with the space-time trade-off $\delta = 120$ uses 81% and 77% of the space required by RelaCT-SD in ShipsLarge and TaxisLarge, respectively. On the other hand, RelaCT-SD is much smaller than ContaCT-SD.

The relative compression of trajectories is a good approach to use space close to grammar compression, while computing object positions and MBRs in constant time. To solve *MBR* and *Time Interval* queries faster, RelaCT-SD+ increases the space of RelaCT by around 8%–60%. The next experiments measure the time performance achieved.

7.4.3. Query performance

We test the query performance of the different structures on both large datasets, using the same settings defined in Section 6.2.

Figure 18a shows that all the RelaCT variants have similar time performance for ObjectPosition queries. It was expected that the '+' variants perform similarly as the basic ones, since they do not affect this query. ContaCT and ContaCT-SD are 1.5–2.5 times faster than RelaCT, but use nearly twice the space. GraCT is twice as slow as RelaCT on ShipsLarge, but uses about half its space, with $\delta = 240$. The situation is similar on TaxisLarge, except that GraCT is now many times slower than RelaCT. Overall, RelaCT-SD offers a very relevant tradeoff for this query.

ObjectTrajectory queries behave in a similar way. However, as shown in Figure 18b, there is a noticeable difference between the RelaCT structures that use plain (RelaCT and RelaCT+) and sparse bitmaps (RelaCT-SD and RelaCT-SD+). The

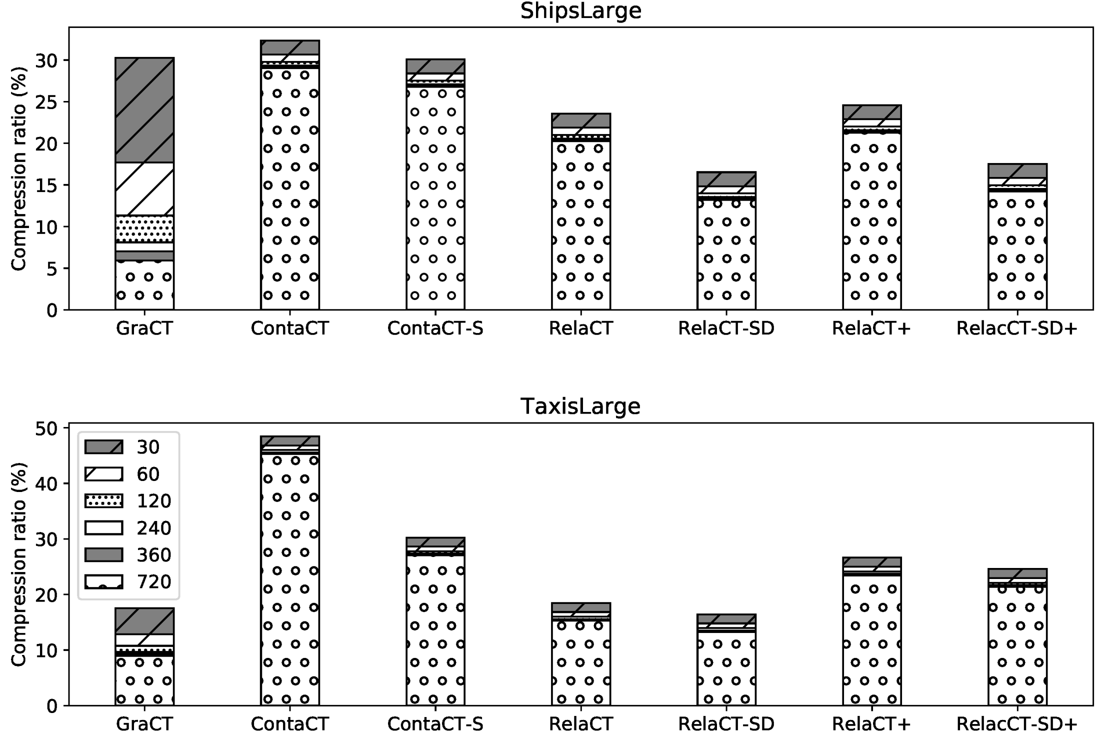


Figure 17.: Compression ratios of the structures with different values of δ.

cause of this difference is that computing the displacements on the reference with ContaCT-SD is slower than with ContaCT; observe that the original ContaCT is 1.7–1.9 times faster than ContaCT-SD on the same kind of queries. RelaCT is also slower than ContaCT, because it incurs the cost of synchronizing every phrase with the reference. In addition, no RelaCT variant is competitive with GraCT in space-time. Thus, RelaCT does not stand out for this query.

The time performance on `MBR` queries is similar to that of `ObjectPosition`, but on `ShipsLarge`, RelaCT+ and RelaCT-SD+ improve by up to 1.4 times the performance of RelaCT and RelaCT-SD, respectively. Instead, there is no significant difference on `TaxisLarge`. The RelaCT variants are 3.0–5.5 times faster than GraCT, and 40%-100% slower than ContaCT. Relative compression is 1.25 times worse than GraCT in space consumption, but it obtains the best space-time trade-off in `MBR` queries: they are only slightly slower than ContaCT while using about half the space.

Figure 19 shows the performance for spatio-temporal queries. In `TimeSlice` with both large and small regions, the variants of RelaCT obtain competitive times compared to the remaining structures. On `ShipsLarge`, the RelaCT-SD variants reach very similar time and space to GraCT, with $\delta = 120$. In `TaxisLarge`, instead, RelaCT-SD becomes 2%–15% faster than GraCT using slightly more space. It is also slightly slower than ContaCT, using about half the space. Since this query only involves one time instant, the structures labelled with '+' do not change the performance.

Figures 19c and 19d show that the time performance for `TimeInterval` worsens in all the structures with respect to `TimeSlice`. The `TimeInterval` algorithm of RelaCT and ContaCT is different from that of `TimeSlice`, running a binary search through the MBRs. This intensive use of *MBR* queries highlights the difference between RelaCT and the simpler ContaCT, though much of the gap is recovered with the '+'-suffixed

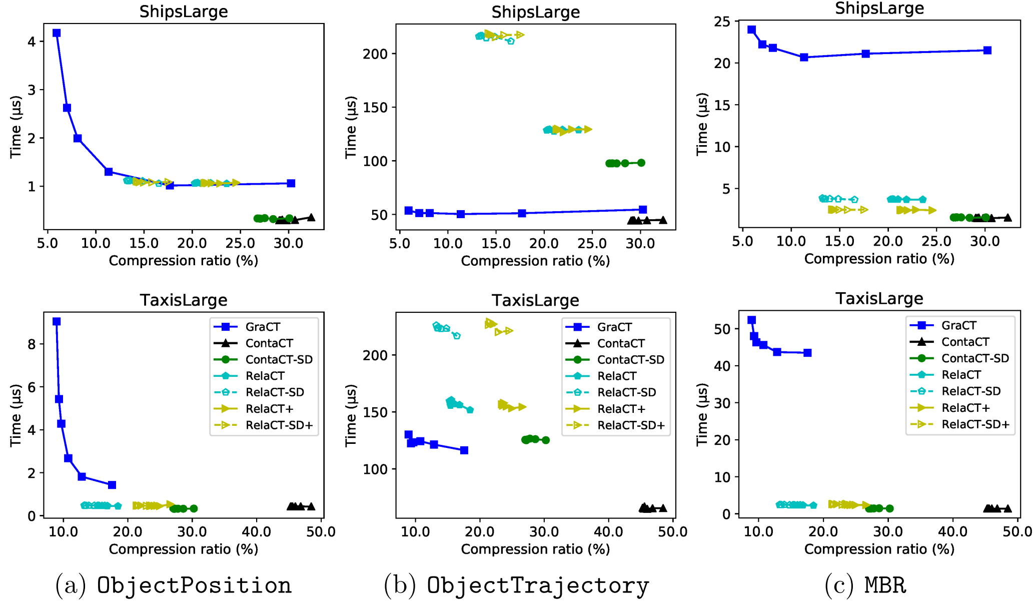


Figure 18.: Space and time for trajectory queries.

variants of RelaCT, which are more clearly faster than the basic ones (taking 60%–90% of their time), at least on `ShipsLarge`. For example, for `TimeInterval L`, comparing the fastest configurations in `ShipsLarge`, we observe that RelaCT+ and RelaCT-SD+ are 1.1 times slower than GraCT, and 1.3 times slower with respect to the two structures of ContaCT.

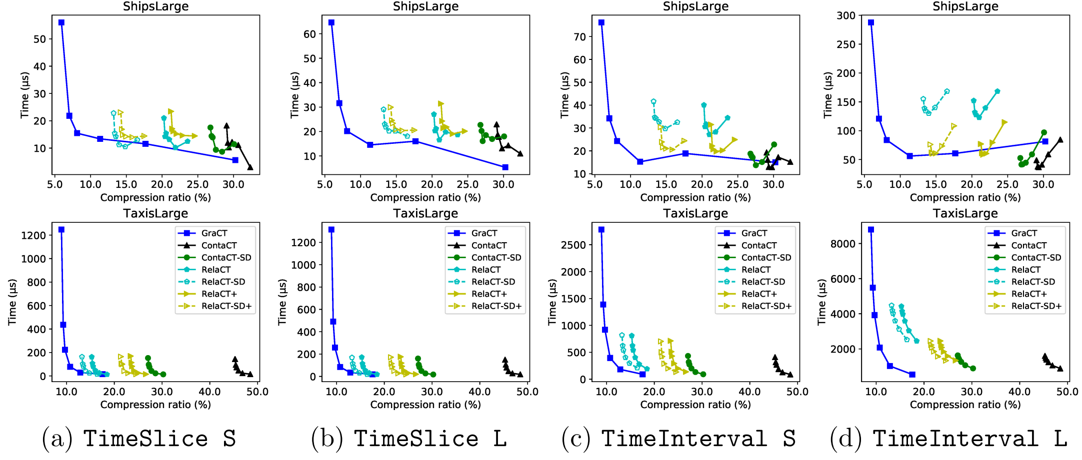
In general, all the structures obtain comparable performance on spatio-temporal queries, with GraCT, closely followed by RelaCT, using much less space than ContaCT.

The space-time trade-offs for nearest neighbour queries can be observed in Figure 20. There is no significant time difference between the four RelaCT variants, so RelaCT-SD is always the best choice for its smaller space usage.

Figure 20a shows that RelaCT-SD is the most interesting variant on for `KNN` queries on `TaxisLarge`: it is just 9% slower than ContaCT-SD and 1.4 times faster than GraCT with $\delta = 120$, and its space is much closer to that of GraCT than to ContaCT. On `ShipsLarge`, instead, GraCT is smaller and 10% faster than RelaCT-SD.

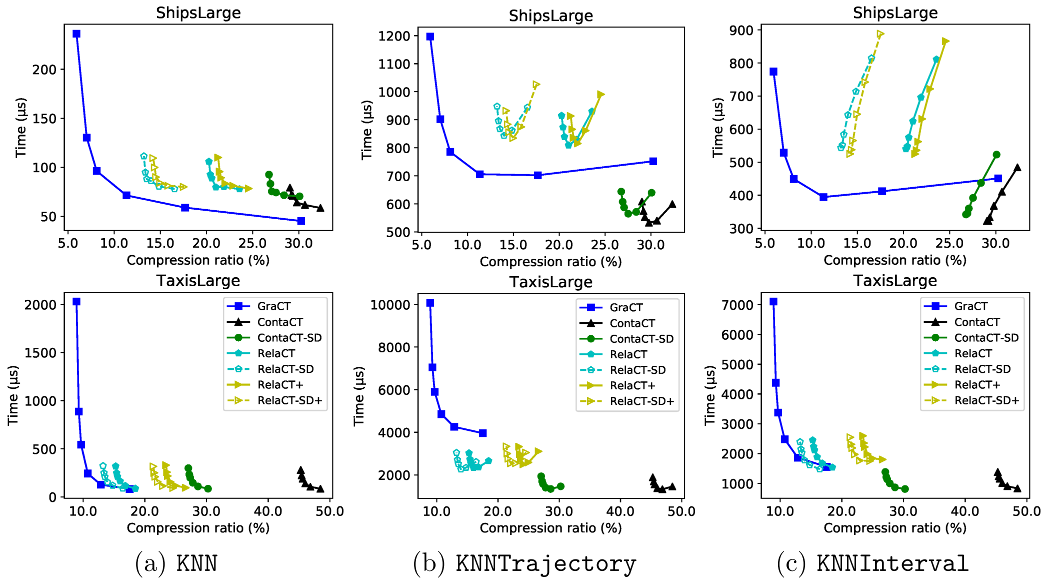
With respect to `KNNTrajectory` (Figure 20b), GraCT dominates RelaCT-SD with $\delta = 120$, by around 10% in time. Instead, RelaCT-SD is much faster than GraCT and 1.5 times slower than ContaCT (which needs twice the space) on `TaxisLarge`.

The last complex nearest neighbour query, `KNNInterval`, is shown in Figure 20c. On `TaxisLarge` the performance is similar to `KNNTrajectory`, though this time RelaCT-SD is only 1.4 times faster than GraCT with $\delta = 120$. On `ShipsLarge`, the RelaCT variants are faster when δ increases. This can be explained by the larger number of elements maintained in the priority queues from the beginning of the algorithm, as more snapshots are covered by the query. For example, before adding each element to the queue, we have to compute its MBR. This effect is more noticeable in RelaCT than in ContaCT, whose MBRs are computer faster. On `ShipsLarge`, GraCT is the dominant solution, whereas RelaCT-SD offers a very good tradeoff on `TaxisLarge`.



(a) `TimeSlice S` (b) `TimeSlice L` (c) `TimeInterval S` (d) `TimeInterval L`

Figure 19.: Space and time for spatio-temporal queries.



(a) `KNN` (b) `KNNTrajectory` (c) `KNNInterval`

Figure 20.: Space and time for nearest neighbour queries.

7.5. *Discussion*

Overall, RelaCT-SD is the most remarkable variant. There are small differences with the performance of RelaCT, but the compression is better. This technique offers times slightly over those of ContaCT, but it uses half the space. It uses slightly more space than GraCT, and has competitive query times. In particular, RelaCT-SD offers an outstanding space-time tradeoff for `ObjectPosition` and `MBR` queries, and it is close (sometimes better and sometimes worse) to the dominant performance of GraCT for spatio-temporal and nearest neighbour queries. The worst performance of RelaCT-SD is obtained in `ObjectTrajectory` queries, where it is sharply dominated by GraCT. Note that each phrase that covers the interval of time needs to be synchronized with the reference, which adds a cost to each processed phrase.

The configurations of RelaCT suffixed with '+' provide a new space-time tradeoff, being clearly faster in `MBR` and `TimeInterval`. Indeed, they reduce the gap with GraCT in time performance of `TimeInterval`. However, the difference in space consumption

between GraCT and those variants increases.

8. Conclusions

Previous work has demonstrated that compressed indexes for large collections of object trajectories in free space can compete with classical indexes in query performance while using orders of magnitude less space. In this work we introduce new algorithms and data representations that yield stronger compressed indexes, in terms both of functionality and of space-time performance.

(1) We introduce new algorithms for more sophisticated nearest-neighbour queries. Previous compressed indexes could only find the objects that were closest to a spatial point at a certain time instant. We now consider the queries *KNN on an interval* and *KNN of trajectories*, which have been studied in the literature (Gao et al. 2007; Tang et al. 2011). The former extends the basic query to a time interval, considering the least distance between the object and the query point during the interval. The second compares object trajectories with a given trajectory, looking for the maximum distance reached during a time interval. Our new algorithms solve those more complex queries on the existing compressed indexes, GraCT (Brisaboa et al. 2019) and ContaCT(Brisaboa et al. 2021), in about an order of magnitude more time than the basic nearest neighbour query, but still within a few milliseconds.

(2) Motivated by the fact that estimating the MBR of an object during a period of time is key for an efficient nearest neighbour algorithm, we introduce a new data structure for storing the positions of the objects at sampled times during the trajectories, based on R-trees instead of the quadtree-like data structure used in previous compressed structures. The R-tree maintains the MBR of the object during the sampled time period and, without increasing the space of the data structures, improves the performance of all nearest neighbour queries by a factor of 2–10. The new representation in fact improves the times for all the other queries, reaching speedups of two orders of magnitude on spatio-temporal queries.

(3) Motivated by the fact that ContaCT uses twice the space of GraCT, but it is much faster at computing object positions and MBRs, two of the most basic queries, we define RelaCT, a new compressed index that exploits redundancies in the trajectories using Relative Lempel-Ziv, a compression method that provides fast random access to the data. This is in contrast to the grammar-compression used by GraCT, which is slower for access. On large repetitive datasets, RelaCT uses about half the space of ContaCT and is only slightly slower. Instead, it uses slightly more space than GraCT and offers competitive query times, particularly outperforming it on the mentioned queries. RelaCT then provides a new relevant tradeoff between both previous compressed indexes.

A relevant future work direction is to introduce dynamism in these compressed indexes. Right now, all of them are static, so they must be rebuilt in order to add or remove new objects, and extend or modify trajectories. The easiest of those challenges is to extend already existing trajectories, as this involves appending movements to the logs and possibly create new snapshots. In the case of GraCT, this implies adapting the context-free grammar to accommodate longer strings. This could be achieved by replacing RePair (Larsson and Moffat 2000), which is an offline grammar compressor,

by an online version like FOLCA (Maruyama et al. 2013). ContaCT and RelaCT are even easier to adapt, as their construction is already online. Other kinds of updates, like modifying past trajectories or adding/removing objects, are more complex and require not only rebuilding logs, but also updating snapshots. While k^2-trees offer dynamic versions (Brisaboa, Bernardo, and Navarro 2012), we are unaware of any compressed dynamic R-tree data structure, only the classic pointer-based one (Guttman 1984).

Another research direction of interest is to further expand the functionality with new queries that have been shown to be useful in the literature. For example, we could extend the functionality of the structures to detect *moving-together patterns* (Gudmundsson, Kreveld, and Speckmann 2004; Alamri, Taniar, and Safar 2013), that is, objects that move together during a period of time. This query could be solved by obtaining the closest objects from a snapshot, and refining their proximity with a similarity function applied on the objects' MBRs. With a similar approach, those structures could also detect common patterns between trajectories (*trajectory clustering*) (Lee, Han, and Whang 2007) and mine sequential patterns from trajectories (Cao, Mamoulis, and Cheung 2005), two important tasks in applications related to travel recommendation or life pattern understanding.

Acknowledgements

For the A Coruña team: This work was supported by CITIC, as Research Center accredited by Galician University System, is funded by "Consellería de Cultura, Educación e Universidade from Xunta de Galicia", supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by "Secretaría Xeral de Universidades" (Grant ED431G 2019/01), Xunta de Galicia/FEDER-UE under Grants [ED431C 2021/53; IG240.2020.1.185]; Ministerio de Ciencia e Innovación under Grants [PID2020-114635RB-I00; PDC2021-120917-C21]. Gonzalo Navarro was funded by ANID – Millennium Science Initiative Program – Code ICN17_002 and by Fondecyt grant 1-200038. Travis Gagie was funded by Fondecyt grant 1171058 and NSERC Discovery Grant RGPIN-07185-2020.

Conflict of Interest

The authors have no conflicts of interest to declare that are relevant to the content of this article.

Data availability statement

The data that support the findings of this study are openly available in figshare at https://doi.org/10.6084/m9.figshare.c.5740388.v2, reference number 10.6084/m9.figshare.c.5740388.v2.

References

Alamri, Sultan, David Taniar, and Maytham Safar. 2013. "Indexing moving objects for directions and velocities queries." *Information Systems Frontiers* 15 (2): 235–248.

Azri, S., U. Ujang, F. Anton, D. Mioc, and A. A. Rahman. 2013. "Review of Spatial Indexing Techniques for Large Urban Data Management." In *Proceedings of International Symposium & Exhibition on Geoinformation ISG 2013.* Kuala Lumpur, Malaysia.

Berndt, Donald J, and James Clifford. 1994. "Using dynamic time warping to find patterns in time series." In *KDD workshop,* 10:359–370. 16. Seattle, WA, USA:

Böhm, C., S. Berchtold, and D. Keim. 2001. "Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases." *ACM Computing Surveys* 33 (3): 322–373.

Brisaboa, N. R., G. de Bernardo, and G. Navarro. 2012. "Compressed Dynamic Binary Relations." In *Proc. 22nd Data Compression Conference (DCC),* 52–61.

Brisaboa, N. R., S. Ladra, and G. Navarro. 2014. "Compact Representation of Web Graphs with Extended Functionality." *Information Systems* 39 (1): 152–174.

Brisaboa, Nieves R., Travis Gagie, Adrián Gómez-Brandón, Gonzalo Navarro, and José R. Paramá. 2021. "An index for moving objects with constant-time access to their compressed trajectories." *International Journal of Geographical Information Science* 35 (7): 1392–1424. eprint: https://doi.org/10.1080/13658816.2020.1833015.

Brisaboa, Nieves R., Adrián Gómez-Brandón, Gonzalo Navarro, and José R. Paramá. 2019. "GraCT: A Grammar-based Compressed Index for Trajectory Data." *Information Sciences* 483:106–135. https://doi.org/https://doi.org/10.1016/j.ins.2019.01.035.

Brisaboa, Nieves R, Miguel R Luaces, Gonzalo Navarro, and Diego Seco. 2013. "Space-efficient representations of rectangle datasets supporting orthogonal range querying." *Information Systems* 38 (5): 635–655.

Cao, Huiping, Nikos Mamoulis, and David W Cheung. 2005. "Mining frequent spatio-temporal sequential patterns." In *Proc. 5th IEEE International Conference on Data Mining (ICDM),* 82–89.

Chakka, V. Prasad, Adam Everspaugh, and Jignesh M. Patel. 2003. "Indexing Large Trajectory Data Sets With SETI." In *Proc. Conference on Innovative Data Systems Research (CIDR).*

Chávez, E., G. Navarro, R. Baeza-Yates, and J.L. Marroquin. 2001. "Searching in Metric Spaces." *ACM Computing Surveys* 33 (3): 273–321.

Chen, Lei, M Tamer Özsu, and Vincent Oria. 2005. "Robust and fast similarity search for moving object trajectories." In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data,* 491–502.

Cheng, Yachang, Wolfgang Fiedler, Martin Wikelski, and Andrea Flack. 2019. "“Closer-to-home” strategy benefits juvenile survival in a long-distance migratory bird." *Ecology and evolution* 9 (16): 8945–8952.

Cudre-Mauroux, P., E. Wu, and S. Madden. 2010. "TrajStore: An adaptive storage system for very large trajectory data sets." In *Proc. 26th IEEE International Conference on Data Engineering (ICDE),* 109–120.

Douglas, D. H., and T. K. Peuker. 1973. "Algorithms for the Reduction of the Number of Points Required to Represent a Line or its Caricature." *The Canadian Cartographer* 10 (2): 112–122.

Eiter, Thomas, and Heikki Mannila. 1994. "Computing discrete Fréchet distance."

Ferrada, H., and G. Navarro. 2017. "Improved Range Minimum Queries." *Journal of Discrete Algorithms* 43:72–80.

Fischer, Johannes, and Volker Heun. 2011. "Space-efficient preprocessing schemes for range minimum queries on static arrays." *SIAM Journal on Computing* 40 (2): 465–492.

Flack, A, W Fiedler, and M Wikelski. 2016. *Data from: Wind estimation based on thermal soaring of birds.* https://doi.org/doi:10.5441/001/1.bj96m274. http://dx.doi.org/10.5441/001/1.bj96m274.

Frentzos, Elias, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. 2007. "Algorithms for nearest neighbor search on moving object trajectories." *Geoinformatica* 11:159–193.

Gao, Yun-Jun, Chun Li, Gen-Cai Chen, Ling Chen, Xian-Ta Jiang, and Chun Chen. 2007. "Efficient k-nearest-neighbor search algorithms for historical moving object trajectories." *Journal of Computer Science and Technology* 22 (2): 232–244.

Gog, S., T. Beller, A. Moffat, and M. Petri. 2014. "From Theory to Practice: Plug and Play with Succinct Data Structures." In *Proc. 13th International Symposium on Experimental Algorithms (SEA),* 326–337.

Gudmundsson, Joachim, Marc van Kreveld, and Bettina Speckmann. 2004. "Efficient detection of motion patterns in spatio-temporal data sets." In *Proc. 12th Annual ACM International Workshop on Geographic Information Systems,* 250–257.

Gudmundsson, Joachim, Patrick Laube, and Thomas Wolle. 2008. "Movement Patterns in Spatio-Temporal Data." *Encyclopedia of GIS* 726:732.

Gutiérrez, G., G. Navarro, A. Rodríguez, A. González, and J. Orellana. 2005. "A Spatio-Temporal Access Method based on Snapshots and Events." In *Proc. 13th ACM International Symposium on Advances in Geographic Information Systems (GIS),* 115–124.

Güting, Ralf Hartmut, Thomas Behr, and Jianqiu Xu. 2010. "Efficient k-nearest neighbor search on moving object trajectories." *The VLDB Journal* 19:687–714.

Guttman, Antonin. 1984. "R-trees: A Dynamic Index Structure for Spatial Searching." In *Proc. ACM International Conference on Management of Data (SIGMOD),* 47–57.

Hausdorff, Felix. 2005. *Set theory.* Vol. 119. American Mathematical Soc.

Hjaltason, G., and H. Samet. 2000. *Incremental similarity search in multimedia databases.* Technical report 4199. Department of Computer Sience, University of Maryland.

Hjaltason, G. R., and H. Samet. 2003. "Index-driven similarity search in metric spaces." *ACM Transactions on Database Systems* 28 (4): 517–580.

Keogh, Eamonn J, and Michael J Pazzani. 2000. "Scaling up dynamic time warping for datamining applications." In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining,* 285–289.

Kieffer, J. C., and E.-H. Yang. 2000. "Grammar-based codes: A new class of universal lossless source codes." *IEEE Transactions on Information Theory* 46 (3): 737–754.

Kuruppu, S., S. J. Puglisi, and J. Zobel. 2010. "Relative Lempel-Ziv Compression of Genomes for Large-Scale Storage and Retrieval." In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE),* 201–206. LNCS 6393.

Larsson, N Jesper, and Alistair Moffat. 2000. "Off-line dictionary-based compression." *Proceedings of the IEEE* 88 (11): 1722–1732.

Lee, Jae-Gil, Jiawei Han, and Kyu-Young Whang. 2007. "Trajectory clustering: a partition-and-group framework." In *Proc. ACM International Conference on Management of Data (SIGMOD),* 593–604.

Liao, Kewen, Matthias Petri, Alistair Moffat, and Anthony Wirth. 2016. "Effective construction of relative Lempel-Ziv dictionaries." In *Proceedings of the 25th International Conference on World Wide Web (WWW),* 807–816.

Lin, Xuelian, Shuai Ma, Han Zhang, Tianyu Wo, and Jinpeng Huai. 2017. "One-pass Error Bounded Trajectory Simplification." *Proceedings of the VLDB Endowment* 10, no. 7 (March): 841–852.

Liu, Jiajun, Kun Zhao, Philipp Sommer, Shuo Shang, Brano Kusy, and Raja Jurdak. 2015. "Bounded quadrant system: Error-bounded trajectory compression on the go." In *Proc. 31st IEEE International Conference on Data Engineering (ICDE),* 987–998.

Maruyama, S., Y. Tabei, H. Sakamoto, and K. Sadakane. 2013. "Fully-Online Grammar Compression." In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE),* 218–229.

Meratnia, Nirvana, and Rolf A. de By. 2004. "Spatiotemporal Compression Techniques for Moving Point Objects." In *Proc. 9th International Conference on Extending Database Technology (EDBT),* 765–782.

Muckell, Jonathan, Jeong-Hyon Hwang, Vikram Patil, Catherine T Lawson, Fan Ping, and SS Ravi. 2011. "SQUISH: an online approach for GPS trajectory compression." In *Proc. 2nd International Conference on Computing for Geospatial Research & Applications,* 1–8.

Munro, J. Ian. 1996. "Tables." In *Proc. 16th Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS),* 37–42.

Munro, J Ian, Rajeev Raman, Venkatesh Raman, and Srinivasa Rao. 2012. "Succinct representations of permutations and functions." *Theoretical Computer Science* 438:74–88.

Nascimento, Mario A., and Jefferson R. O. Silva. 1998. "Towards historical R-trees." In *Proc. ACM Symposium on Applied Computing (SAC),* 235–240.

Navarro, Gonzalo. 2016. *Compact Data Structures – A practical approach.* New York, NY: Cambridge University Press.

Nibali, A., and Z. He. 2015. "Trajic: An Effective Compression System for Trajectory Data." *IEEE Transactions on Knowledge and Data Engineering* 27 (11): 3138–3151.

Okanohara, D., and K. Sadakane. 2007. "Practical Entropy-Compressed Rank/Select Dictionary." In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX),* 60–70.

Pfoser, Dieter, Christian S. Jensen, and Yannis Theodoridis. 2000. "Novel Approaches to the Indexing of Moving Object Trajectories." In *Proc. 26th International Conference on Very Large Data Bases (VLDB),* 395–406.

Potamias, M., K. Patroumpas, and T. Sellis. 2006. "Sampling Trajectory Streams with Spatiotemporal Criteria." In *Proc. 18th International Conference on Scientific and Statistical Database Management (SSDBM),* 275–284.

Rigaux, Ph, Michel Scholl, and Agnes Voisard. 2002. *Spatial databases: with application to GIS.* San Francisco, CA: Morgan Kaufmann.

Samet, Hanan. 1984. "The Quadtree and Related Hierarchical Data Structures." *ACM Computing Surveys* 16:187–260. https://doi.org/10.1145/356924.356930.

Shang, Shuo, Lisi Chen, Zhewei Wei, Christian Søndergaard Jensen, Kai Zheng, and Panos Kalnis. 2017. "Trajectory similarity join in spatial networks." *Proceedings of the VLDB Endowment* 10 (11).

Su, Han, Shuncheng Liu, Bolong Zheng, Xiaofang Zhou, and Kai Zheng. 2020. "A survey of trajectory distance measures and performance evaluation." *The VLDB Journal* 29 (1): 3–32.

Tang, Lu-An, Yu Zheng, Xing Xie, Jing Yuan, Xiao Yu, and Jiawei Han. 2011. "Retrieving k-nearest neighboring trajectories by a set of point locations." In *Proc. International Symposium on Spatial and Temporal Databases,* 223–241.

Tao, Yufei, and Dimitris Papadias. 2001. "MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries." In *Proc. 27th International Conference on Very Large Data Bases (VLDB),* 431–440.

Trajcevski, Goce, Hu Cao, Peter Scheuermann, Ouri Wolfson, and Dennis Vaccaro. 2006. "On-line data reduction and the quality of history in moving objects databases." In *Proc. 5th ACM International Workshop on Data Engineering for Wireless and Mobile Access,* 19–26.

Vazirgiannis, Michalis, Yannis Theodoridis, and Timos K. Sellis. 1998. "Spatio-Temporal Composition and Indexing for Large Multimedia Applications." *ACM Multimedia Systems Journal* 6 (4): 284–298.

Vlachos, Michail, George Kollios, and Dimitrios Gunopulos. 2002. "Discovering similar multidimensional trajectories." In *Proceedings 18th international conference on data engineering,* 673–684. IEEE.

Worboys, Michael F. 2005. "Event-oriented approaches to geographic phenomena." *International Journal of Geographical Information Science* 19 (1): 1–28.

Zhao, Yan, Shuo Shang, Yu Wang, Bolong Zheng, Quoc Viet Hung Nguyen, and Kai Zheng. 2018. "REST: A reference-based framework for spatio-temporal trajectory compression." In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining,* 2797–2806.

Zheng, Bolong, Haozhou Wang, Kai Zheng, Han Su, Kuien Liu, and Shuo Shang. 2018. "SharkDB: An in-memory column-oriented storage for trajectory analysis." *World Wide Web* 21 (2): 455–485.

Zheng, Yu, and Xiaofang Zhou, eds. 2011. *Computing with Spatial Trajectories.* New York, NY: Springer.

Ziv, Jacob, and Abraham Lempel. 1977. "A universal algorithm for sequential data compression." *IEEE Transactions on information theory* 23 (3): 337–343.

Ziv, Jacob, and Abraham Lempel. 1978. "Compression of individual sequences via variable-rate coding." *IEEE transactions on Information Theory* 24 (5): 530–536.