ARTICLE TEMPLATE

An index for moving objects with constant-time access to their compressed trajectories

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

[a]Universidade da Coruña, Centro de investigación CITIC, Facultade de Informática, A Coruña, Spain; [b]Faculty of Computer Science, Dalhousie University, Halifax, Canada; [c] Millennium Institute for Foundational Research on Data, Department of Computer Science, University of Chile, Beauchef 851, Santiago, Chile

ARTICLE HISTORY
Compiled October 1, 2020

ABSTRACT
As the number of vehicles and devices equipped with GPS technology has grown explosively, an urgent need has arisen for time- and space-efficient data structures to represent their trajectories. The most commonly desired queries are the following: queries about an object's trajectory, range queries and nearest neighbor queries. In this paper we consider that the objects can move freely and we present a new compressed data structure for storing their trajectories, based on a combination of logs and snapshots, with the logs storing sequences of the objects' relative movements and the snapshots storing their absolute positions sampled at regular time intervals. We call our data structure *ContaCT* because it provides *Con*stant-*t*ime *a*ccess to *C*ompressed *T*rajectories. Its logs are based on a compact partial-sums data structure that returns cumulative displacement in constant time, and allows us to compute in constant time any object's position at any instant, enabling a speedup when processing several other queries. We have compared ContaCT experimentally with another compact data structure for trajectories, called GraCT, and with a classic spatio-temporal index, the MVR-tree. Our results show that ContaCT outperforms the MVR-tree by orders of magnitude in space and also outperforms the compressed representation in time performance.

KEYWORDS
Moving objects; Trajectories representation; Spatio-temporal query.

1. Introduction

Managing data on object trajectories is by no means a new topic, but it has attracted a lot of attention recently (Mahmood *et al.* 2019) as the number of vehicles and devices — cars, ships, planes, drones, smartphones, smartwatches — equipped with GPS technology has grown explosively. The data generated by these multiple sources have a wealth of applications, including traffic management, analysis of human movement, tracking animal behavior, security and surveillance, military logistics and combat, and emergency-response planning (Gudmundsson *et al.* 2008). The vast quantities of data now involved, however, can make storing and processing them a challenge, especially

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

on mobile devices, so an urgent need has arisen for data structures that are more time- and space-efficient (Zheng and Zhou 2011).

By a *trajectory* we mean the route followed by a moving object during an interval of time. The approaches to storing trajectories depend on the kind of vehicle or device involved, the application, the geographical scale and resolution required, and other criteria. They can be roughly divided into two groups: those designed for objects moving over networks, such as roads or public transportation networks, and those designed for objects moving freely in space, either two- or three-dimensional. In this paper we focus on the second case.

The most basic queries that a data structure storing trajectories should support are *object position* and *object trajectory*: given an object's identifier and either a time instant or a time interval, respectively, they return that object's position at that instant or its trajectory during that interval. For the applications we have mentioned, however, more sophisticated queries are necessary. The most common are *range queries*: given a spatial range and a time instant or a time interval, *time slice* and *time interval* return the identifiers of the objects in the spatial range in that instant or during that interval, respectively. We also consider the simplest variant of the popular *nearest neighbor* queries: given a position, an integer k, and a time instant, return the identifiers of the k closest objects to that point in that instant. The last query we consider in this paper is *minimum bounding rectangle (MBR)*: given an object's identifier and a time interval, return the smallest axis-aligned rectangular spatial range containing that object for that whole time interval.

Solving those queries efficiently requires storing and indexing the trajectory data in different forms. Since the 1990s, several different disk-based data structures and indexes for mobile objects have been proposed. As we explain in Section 2, some solutions are oriented to efficiently solving *object position* and *object trajectory*, while others index the space to efficiently answer *range* and *nearest neighbor*. Many of those that index the space enrich the classical R-tree index (Guttman 1984), by adding a new dimension to represent time.

As internal memories have grown, however, in-memory indexes have become more popular in many areas of computer science, since internal memory is several orders of magnitude faster than secondary memory. In particular, in-memory trajectory indexes have been proposed (Cudre-Mauroux *et al.* 2010, Zheng *et al.* 2018). To keep large datasets in memory, it is often necessary to compress them. The common compression technique is *delta compression*, which represents each position of an object relative to its position in the previous instant, thus exploiting encodings that use fewer bits to represent smaller numbers. In order to retrieve the position of the object at a specified instant, the differences must be summed up. This operation, called *partial sum*, is sped up by storing sampled absolute positions.

An alternative to delta compression is grammar-based compression, which compresses datasets by representing them as context-free grammars that expand to yield those datasets. Grammar-based compression exploits (and depends on) the repetitiveness of the datasets, which in this case means the similarities between the trajectories of many objects. In a previous paper (Brisaboa *et al.* 2019) the authors proposed a system of *Gra*mmar *C*ompressed *T*rajectories (GraCT), which also stores summary data associated with the grammar to speed up queries.

In this paper, we propose a new in-memory data structure that indexes compressed trajectories, called *Con*stant-*t*ime *a*ccess to *C*ompressed *T*rajectories (*ContaCT*). We return to the idea of differentially encoding the trajectories, with each object storing a compressed *log* of its movements. Like other differential encoders, we achieve com-

pression whenever consecutive movements in trajectories tend to be comparatively small. The important novelty is that ContaCT encodes logs with modern and compact *bitmap* data structures supporting *constant-time* queries (Navarro 2016). For example, constant-time calculation of partial sums enables us to efficiently answer *object position* and *object trajectory* queries, which are fundamental. To answer *range* and *nearest neighbor* queries efficiently, ContaCT combines the bitmaps with *snapshots*, which represent the absolute positions of all the objects at regular time periods using a compact data structure called a k^2-tree (Brisaboa *et al.* 2013).

We further augment the bitmaps with compact data structures that support, in constant time, another query called *range minimum/maximum* (Fischer and Heun 2011). This query takes an interval in a sequence and returns the positions of the minimum and maximum values in that interval. Since the bitmaps store the objects' coordinates, we can use *range minimum/maximum* queries to compute an object's *MBR* in constant time. With constant-time *MBR* queries we can speed up *time interval* queries, by filtering out candidates if their MBRs for the query interval do not intersect the query range.

Constant-time *MBR* queries are important in their own right. For example, some visualization tools (Leontiadis *et al.* 2011, Becker *et al.* 2015, Li *et al.* 2011) display MBRs of all objects during time intervals, and data mining problems can be sped up by using MBRs, such as detecting objects that move together (Gudmundsson and van Kreveld 2006, Gudmundsson *et al.* 2004, Ma *et al.* 2013), extracting sequential patterns from trajectories (Cao *et al.* 2005, Ye *et al.* 2009, Xiao *et al.* 2010), and trajectory clustering (Li *et al.* 2010a,b).

We have compared ContaCT experimentally with GraCT and with a classic spatio-temporal index, called the MVR-tree (Tao and Papadias 2001b), and found ContaCT to be generally somewhat larger than GraCT but much faster at finding objects' positions and minimum bounding rectangles (and competitive for the rest of the queries) and much smaller and somewhat faster than the MVR-tree.

In our experiments we used datasets from four different types of moving objects: ships, planes, city taxis, and birds. ContaCT achieves reasonably good compression ratios, representing both trajectory data and the auxiliary data structures in 20%–60% of the sizes of the original uncompressed datasets. Although these space usages are up to 3 times higher than those achieved with GraCT, ContaCT is 4–14 times faster at answering *object position* queries and 4–9 times faster at answering *MBR* queries, which are the queries that best exploit its constant-time computations. For the other queries, ContaCT is up to 3.5 times faster than GraCT, unless we enlarge the latter to use half the space of ContaCT, at which point their performance becomes similar. We note that the birds' trajectories are not repetitive and for this dataset GraCT used space much closer to ContaCT while remaining significantly slower. ContaCT is 60–90 times smaller than the MVR-tree but as fast, and even faster on *time interval* queries.

The rest of the paper is laid out as follows: in Section 2 we review the state of the art; in Section 3 we summarize the basic concepts of compact data structures; Section 4 covers the structure of ContaCT; Section 5 explains how it supports queries; in Section 6 we present our experimental results; finally, in Section 7 we present our conclusions and outline directions for future work.

2. State of the art

The research on moving objects can be roughly classified into two groups: one oriented to objects that move over networks (Popa *et al.* 2015) (roads, streets, or public transportation networks), and another oriented to objects that move freely in the space (typically in two or three dimensions). This work belongs to the second group. In this section we describe how free trajectories are modeled, and then cover the most relevant previous work on compressing and indexing trajectory data.

2.1. *Modeling trajectories*

Trajectories can be understood as continuous space-time functions. Due to storage constraints and the frequency of emission of GPS devices, however, they are actually represented as discrete sequences of pairs $(position, time)$. That is, trajectories are usually represented as lists of timestamped points in a two- or three-dimensional space.

The displacement of an object between two consecutive timestamps is called a *movement*. A movement in a d-dimensional space involves displacements of different magnitude (which could be 0) along each of the dimensions of the space.

By increasing the sampling rate of the timestamps one obtains better accuracy, but also requires handling larger amounts of data, which in turn increases storage, transmission, and processing costs. Reducing or simplifying the trajectory representations is useful for decreasing those costs as data volumes grow.

Trajectory simplification aims to reduce the size of a trajectory by discarding some of its less relevant points. The simplified trajectories can then be compressed and indexed with other techniques. We can distinguish between two kinds of trajectory simplification methods: batch mode and online mode.

The batch mode methods compress the trajectory as a whole. The Douglas-Peucker algorithm (Douglas and Peuker 1973) discards the most redundant points and keeps those with relevant information. A similar method is Top-Down Time Ratio (Meratnia and de By 2004), which is identical to Douglas-Peucker, but exploits the time dimension. Other methods approximate trajectories by deciding on the relevance of points considering the network (Schmid *et al.* 2009, Ta *et al.* 2016).

The online mode methods compress the trajectory as they scan it by maintaining a buffer of the recent points. The simplest method (Potamias *et al.* 2006) is to collect the positions at regular time intervals. Other algorithms, like Dead Reckoning (Trajcevski *et al.* 2006) and STTrace (Potamias *et al.* 2006), exploit speed and direction of objects to discern which points produce an important change in the trajectory path. SQUISH (Muckell *et al.* 2011) is another online method, whose approximation error cannot be bounded. SQUISH-E (Muckell *et al.* 2014) is an evolution of SQUISH that offers a compression bound and an error bound. Other online approximation methods are OPERB (Lin *et al.* 2017), and BQS (Liu *et al.* 2015).

This paper does not focus on trajectory simplification. We assume that the timestamped points of a trajectory are already recorded at regular intervals. This is the case for many GPS devices in real life[1]. We also assume that the recorded timestamped points correspond to the same time instants for all the objects in the collection. In addition, we discretize the space, dividing it into square cells of a fixed size and identifying positions with their containing cell. This corresponds to the well-established

[1]There can be time instants without values, however, because objects could stop recording their position. For example, ships turn off the GPS when they are at a port. We do address that case.

raster model.

2.2. *Compressing trajectories*

Once the trajectory is simplified, we can apply some lossless compression method to further reduce its size. The most common method is *delta compression*: after storing the first position of the trajectory, each subsequent position is encoded as its difference with respect to the previous one. That is, the trajectory is encoded as a sequence of consecutive movements. Those movements tend to be small, and so the differences can be stored with fewer bits than the original coordinates. While the whole trajectory can be efficiently recovered by accumulating differences, computing just the position of an object at a given time t is not so efficient, because it requires decompressing the whole trajectory until the queried time t. Some methods introduce a space/time trade-off by sampling some absolute positions at regular intervals of time.

Delta compression is used in systems like TrajStore (Cudre-Mauroux *et al.* 2010) and SharkDB (Zheng *et al.* 2018). Trajic (Nibali and He 2015), instead, predicts the next point of a trajectory and stores the difference between the predicted point and the real one. The deviations are then encoded with a (usually) small number of bits. A different technique is used in REST (Zhao *et al.* 2018), where every trajectory is represented by the concatenation of parts of other trajectories, which are known as references. A spatio-temporal threshold adjusts the error between the original and the represented trajectories.

As expected, systems like Trajic (Nibali and He 2015), STTrace (Potamias *et al.* 2006) or SQUISH (Muckell *et al.* 2011), which just store the trajectories, can retrieve trajectories very efficiently. However, they are inefficient at solving *range* or *nearest neighbor* queries. For solving *range* queries, they need to retrieve the trajectory of every object during the queried interval of time and check if some point of the trajectory is within the queried region. For *nearest neighbor* queries, they compute the position of every object at the queried time instant.

A different approach is taken by GraCT (Brisaboa *et al.* 2019), which uses instead *grammar compression* on the set of the differential trajectories of all the objects, in the hope that many objects will follow similar trajectories. Grammar compression produces a context-free grammar that generates (only) the set of the trajectories. Grammar non-terminals are enriched with summary data that allows skipping without expanding them when a trajectory is traversed. Therefore, the trajectories are not only encoded in less space, but they are also faster to traverse.

Although our proposal is based on GraCT, the compression of the trajectories is totally different. ContaCT uses a differential compression of the trajectories, by using bitmaps and other compact data structures. Those structures allow us to compute the position of an object in constant time, thereby outperforming GraCT in many queries. Although the trajectory compression of ContaCT is generally worse than that of GraCT, it is also more robust because it only relies on the assumption that consecutive movements along a trajectory are relatively small. This is a more basic assumption than the fact that many objects share similar trajectories, which is needed for the grammar-compression of GraCT to outperform that of ContaCT.

2.3. *Indexing trajectories*

Spatio-temporal indexes for object trajectories are mostly based on the R-tree (Guttman 1984), a classic spatial data structure that encloses objects with Minimum Bounding Rectangles (MBRs). Each node of an R-tree includes an MBR that wraps the MBRs of their children or objects in leaves. A search for the objects within an area can be efficiently solved by descending through the nodes whose MBRs intersect the queried area. Therefore, the queries run more efficiently when the MBRs are smaller.

The 3DR-tree (Vazirgiannis *et al.* 1998) is a spatio-temporal index that replaces MBRs by MBBs (Minimum Bounding Boxes), where the third dimension represents the time. Since this time dimension can cover a very long interval, the MBBs may become large and damage the search performance. Some indexes (Pfoser *et al.* 2000) try to solve this problem by modifying the strategy that creates MBBs (STR-Tree), or by inserting partial trajectories as MBBs in an R-tree (TB-tree).

Indexes like MR-tree (Xu *et al.* 1990), HR-tree (Nascimento and Silva 1998), HR+-tree (Tao and Papadias 2001a), and MV3R-tree (Tao and Papadias 2001b) adopt a different strategy: they conceptually store an R-tree for each timestamp. Storing a full R-tree per timestamp requires a large amount of space, thus they only store the parts of the R-tree that differ from the preceding one.

Grid-based indexes split the space into several partitions and build a temporal index for each partition. SETI (Chakka *et al.* 2003), for example, divides the space into cells and, for each cell, it indexes the trajectories by time with an R*-tree. Other grid-based indexes are Multi Time Split B-Tree (Zhou *et al.* 2005), Compressed Start-End tree (Wang *et al.* 2008), PIST (Botea *et al.* 2008), and GCOTraj (Yang *et al.* 2018).

All these indexes based on R-trees can efficiently solve *range* and *nearest neighbor* queries. However, they are very slow for *object trajectory* queries because retrieving each trajectory position requires a top-down traversal of an R-tree.

A completely different approach is presented in the PA-tree (Ni and Ravishankar 2007), which approximates each trajectory by a single continuous polynomial, avoiding MBBs and R-trees. Since the original and the approximate trajectories are not identical, they keep the maximum deviation between both so as to detect false negatives.

A recent approach is to store trajectories in a distributed computing framework and augment it with spatio-temporal indexes. PRADASE (Ma *et al.* 2009) and CloST (Tan *et al.* 2012) use Hadoop and a spatio-temporal index. TrajSpark (Zhang *et al.* 2017) is based on Spark (Zaharia *et al.* 2016) and adds a two-level spatio-temporal index. Other indexes like MD-HBase (Nishimura *et al.* 2013), R-HBase (Huang *et al.* 2014), and GeoMesa (Hughes *et al.* 2015) are based on distributed key/value storages.

The SEST-Index (Gutiérrez *et al.* 2005, Worboys 2005) uses a different approach, based on *snapshots* and *logs*. A time sampling interval is defined and a spatial index is stored for each sampled timestamp, recording the positions of all the objects at that timestamp. A log of "events" signals the objects that appear at some position, or disappear, between each consecutive pair of snapshots.

Our index follows the model of snapshots and logs, but it has important differences with SEST-Index. First, ContaCT uses compact data structures to store the information in an efficient way, whereas SEST-Index does not consider compression. Second, the information stored in the ContaCT log contains the relative movements of each individual object. The "events" stored by SEST-Index, instead, are suitable for detecting when an object is within a region, but not for obtaining its trajectory.

2.4. *Combining compression and indexing*

There are a few indexes that combine spatial indexing with trajectory compression. TrajStore (Cudre-Mauroux *et al.* 2010) is an example. Like SETI (Chakka *et al.* 2003), it splits every trajectory into subtrajectories, each one confined to a cell. Instead of storing those subtrajectories in plain form, they are compressed in each cell. TrajStore can be considered a lossy method because it groups similar trajectories and stores only one of them. These representative trajectories are encoded using delta compression. A quadtree indexes the cells of the space and, inside each cell, it stores a temporal index. Like the indexes based on the R-tree, TrajStore can efficiently solve *range* queries.

Another system combining compression and indexing is SharkDB (Zheng *et al.* 2018). It splits the time dimension into intervals of a given length, and stores one point for each trajectory and interval of time. The points contained in each interval are stored as a column of a column-oriented database and encoded with delta compression. SharkDB focuses on solving *nearest neighbor* queries.

GraCT (Brisaboa *et al.* 2019) uses the same snapshot-and-log architecture of the SEST-Index (Gutiérrez *et al.* 2005) combined with grammar compression of the log, as explained. The snapshots are useful for speeding up *range* and *nearest neighbor* queries, because from the snapshot closest to the query we can use spatial range queries to filter the objects that have a chance of belonging to the spatial query window during the queried time interval. In addition to speeding up the scanning of compressed trajectories, the MBRs stored for nonterminals are useful for discarding portions of the trajectories that cannot intersect the query window.

Our index can be placed in this same category. ContaCT uses the same architecture of GraCT, but it uses delta compression instead of grammar compression. Further, it uses a clever format for delta compression that allows it to find the position of any object at any time instant in *constant time*, without the need for traversals. It can also compute the MBR of any object along any time interval on the fly, in constant time as well, which is useful for speeding up *range* and *nearest neighbor* queries (GraCT has precomputed MBRs only for its grammar nonterminals). This ability is relevant by itself, as a summary of the movements of an object along a time period.

Note that systems that just store trajectories (Trajic, STTrace, SQUISH) are efficient for retrieving the whole trajectory of each object; STTrace can also answer *object trajectory* efficiently because it samples absolute positions. However, they cannot efficiently compute *range* and *nearest neighbor* queries, where they must check the position of every object. Conversely, systems storing the trajectory data in R-trees (SETI, Trajstore, SharkDB) are efficient at *range* and *nearest neighbor* queries, because they can quickly filter the candidates. However, they need to traverse several paths in one or more R-trees in order to recover a given trajectory. The indexes based on snapshots and logs (SEST-Index, GraCT, ContaCT) have sufficient information to efficiently handle both types of queries, because they store the trajectories explicitly while providing snapshots to speed up the filtration of candidate objects.

3. Compact data structures

An emerging research area termed *compact data structures* (Navarro 2016) focuses on the design of data structures that compress the data and can solve queries without the need to decompress the whole structure. Their space consumption is usually close to that of the standard compressed form of the data, and their query times are competi-

tive with those of classical data structures. Our structure is composed of several basic compact data structures and compression methods, which we cover in this section.

3.1. *Rank and select on bitmaps*

Rank and *select* operations over bitmaps are widely used in compact data structures: $rank_b(B, p)$ is the number of times bit b occurs in the bitmap B up to position p, and $select_b(B, i)$ is the position of the i-th occurrence of bit b in bitmap B.

These operations can be solved in $O(1)$ time with just $o(n)$ bits of space on top of the n bits used by $B[1..n]$ (Munro 1996). In practice, the implementations of *rank* are faster than those of *select*. 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* makes it as fast as *rank* in practice (Navarro 2016).

When B has $m \ll n$ 1s, 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. *Partial sums*

Given a sequence of positive values $d_1, d_2, \ldots, d_m$, a *partial sum* computes $x_i = \sum_{j=1}^{i} d_j$ for any given i. An obvious way to solve these queries in constant time is to store all the answers x_i explicitly, using $m \log n$ bits, with $n = x_m$.

By using bitmaps with *rank* and *select* functionality, it is possible to build a compact partial sums data structure that still answers in constant time. Since $0 < x_1 < x_2 < \ldots < x_m$, we can set a bitmap $B[1..n]$, having 1s at the positions x_i. It then holds that $x_i = select_1(B, i)$. By using the Elias-Fano representation, the space of the data structure is $m \log(n/m) + 2m$ bits.

An alternative setup is that we have increasing values x_i and store them in compact form by marking them in B, in a way that allows us to retrieve any value in constant time. The bitmap B can then be regarded as the concatenation of the differential values $d_i = x_i - x_{i-1}$ written in unary. Each difference d_i is then encoded in $\log(n/m) + 2$ bits on average, where n/m is the average difference value.

It is instructive to compare this result with a classical differential encoding. For example, δ codes (Bell *et al.* 1990) use $\log d_i + o(\log d_i)$ bits to encode d_i, adding up to $\sum_{i=1}^{m} \log d_i + o(\log d_i) \leq m \log(n/m) + o(m \log(n/m))$ bits. The inequality approaches equality when the values d_i are close to each other. To retrieve the values x_i with partial sums in time $O(t)$, however, we must sample the absolute values every t positions, for an extra space of $(m/t) \log n$ bits. For example, with the extra space of $2m$ bits of the bitmap encoding we described, we can only achieve $O(\log n)$ time, not $O(1)$.

Recall that we use differential encoding for the compression of the trajectories. Those differential movements, in a two-dimensional space, are composed of horizontal and vertical displacements, which can be represented by using partial sum structures. Indeed, in ContaCT, we use a small variation of the constant-time partial sums structure, which allows us to represent differences that are 0 as well. In our case, the value of a difference $d_i = v$ is represented as $v + 1$, that is, v consecutive zeros followed by a 1 to mark the end of the number. Therefore, the value x_i can be obtained as $select_1(B, i) - i$.

3.3. *Range minimum/maximum queries*

Let $A[1..n]$ be an array of integers. The *range minimum query* $rmq(A, i, j)$ returns the position k, where $A[k]$ is the minimum value in $A[i..j]$. The *range maximum query* $rMq(A, i, j)$ computes the position of the maximum instead of the minimum. Notice that those types of queries are useful for obtaining the MBR of an object, where the corners correspond to the minimum and maximum value of each dimension from the trajectory.

Fischer et al. (Fischer and Heun 2011) proposed an rmq structure that answers in $O(1)$ time and uses only $2n+o(n)$ bits, without the need to access A. An obvious variant solves rMq in $O(1)$ time using other $2n + o(n)$ bits. A simplified variant (Ferrada and Navarro 2017) is also $O(1)$ time, but it obtains the best time in practice.

3.3.1. Queries on arrays with runs

Observe that rMq (rmq) queries return only the position in A of the maximum (minimum) value. In our application, we will need to store A for other reasons, but our array A will have long runs of nonincreasing or nondecreasing values. We can exploit this regularity to design smaller range query data structures (Gagie *et al.* 2017). Let us consider rMq; the case of rmq is similar. We use a bitmap $B[1..n]$ that has 1s only at the positions where there is a local maximum. A local maximum occurs at position i of A if $A[i'] < A[i] > A[i + 1]$, where $i' < i$ is the largest position where $A[i'] \neq A[i]$ (assume $A[0] = A[n + 1] = -\infty$). The rMq structure will be built on an array A' that stores the values $A[i]$ at local maxima (recall that A' will not be stored). For example:

$$\begin{aligned} A &: 1,3,2,2,2,5,8,9,7,6,7,3,6,7,8,7,7,5,3,2 \\ B &: 0,1,0,0,0,0,0,1,0,0,1,0,0,0,1,0,0,0,0,0 \\ A' &: 3,9,7,8 \end{aligned}$$

Now, for a given range $A[i..j]$, the maximum value is either a local maximum within that interval or one of the values at the extremes, $A[i]$ or $A[j]$. We therefore map $A[i..j]$ to the corresponding area $A'[i'..j']$ where the local maxima in $A[i..j]$ lie, find the maximum in $A'[i'..j']$, and compare it with $A[i]$ and $A[j]$. The interval in A' is computed as $i' = rank_1(B, i - 1) + 1$ and $j' = rank_1(B, j)$ and the value of $A'[k']$ is obtained with $A[select_1(B, k')]$.

In our example, to find $rMq(A, 5, 15)$, we first compute $i' = rank_1(B, 5-1)+1 = 2$ and $j' = rank_1(B, 15) = 4$. The query $rMq(A', 2, 4) = 2$ tells that the largest local maximum in $A[5..15]$ is $A[select_1(B, 2)] = A[8] = 9$. We compare it with the extremes, $A[5] = 2$ and $A[15] = 8$ to conclude that $rMq(A, 5, 15) = 8$.

Instead of the $2n + o(n)$ bits of the classical rMq structure, we use $n + 2n' + o(n)$ bits, where $n' < n$ is the number of local maxima in A.

3.4. *k^2-trees*

The k^2-tree (Brisaboa *et al.* 2014) is a compact data structure for binary matrices. It can be regarded as a space- and time-efficient version of a region quadtree (Samet 1984), that is, a spatial index. Indeed, the k^2-tree is the basis of the spatial index used in ContaCT to store the object locations at selected time instants.

The k^2-tree is a k^2-ary tree built by recursively splitting the binary matrix into k^2 submatrices of equal size. In level i, the size of these submatrices is n^2/k^{2i}. In the first

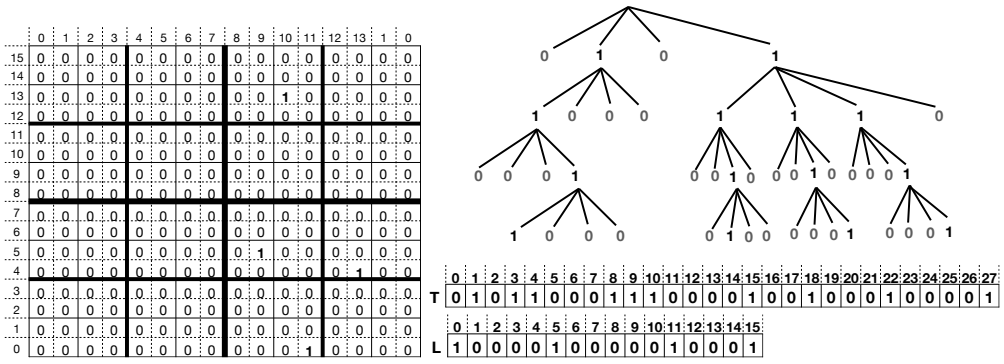


Figure 1. A k^2-tree.

step, the matrix is split and sorted from left to right and from top to bottom. Each submatrix corresponds to a child of the root node, whose value is 0 if the submatrix is full of 0s, and 1 otherwise. The procedure continues recursively for each child with value 1 until reaching a submatrix full of 0s or the individual cells (submatrices of size 1×1).

Therefore, each internal node stores k^2 bits, and this is sufficient to describe the binary matrix. The structure is stored in two bitmaps, T and L. The bits of internal levels are concatenated in T following a level-wise traversal, and L stores the last level of the tree. The bitmap T is provided with *rank* and *select* functionality. See Figure 1.

We can efficiently collect the 1s of a region of the matrix by traversing the appropriate subtrees using *rank* operations on T. For example, given a 1 (i.e., a node) at position p in T, its k^2 children start at position $rank_1(T, p) \times k^2$ of $T : L$. Similarly a bottom-up traversal can be simulated with *select* operations on T. For example, given a position p of $T : L$, its parent is located at position $select_1(T, \lfloor p/k^2 \rfloor)$ of T.

In ρ dimensions, the k^2-tree is generalized to a k^ρ-tree, where each internal node stores k^ρ bits according to the subgrids into which its grid is partitioned. In this paper we use $k = 2$ and, in most cases, $\rho = 2$.

4. Constant time access trajectories

Our proposal *Constant time access trajectories* (ContaCT) also uses a raster representation of the space and a discretization of the time, storing discretized object positions at regular time intervals. The cell size and time interval span can be configured depending on the domain. There is a trade-off between accuracy and data size: if the cell size or the time interval span decrease, the trajectory accuracy increases, but the size of the dataset grows.

The use of a raster model introduces some imprecision and errors, when we compare the original trajectory with the one obtained from the raster representation. Those errors can be bounded depending on the type of query (Cao *et al.* 2006). Assume the raster model uses cells of size $c \times c$. Every time our structure obtains a cell, if it reports its center as the corresponding point, the maximum distance between the original point and the retrieved one is half of the diagonal of a cell, $c/\sqrt{2}$.

That error bound directly affects queries related to the trajectory of an object (i.e., obtaining the position of an object, its trajectory, or computing its MBR), where every returned point can vary by at most $c/\sqrt{2}$ units. On the other hand, for region queries,

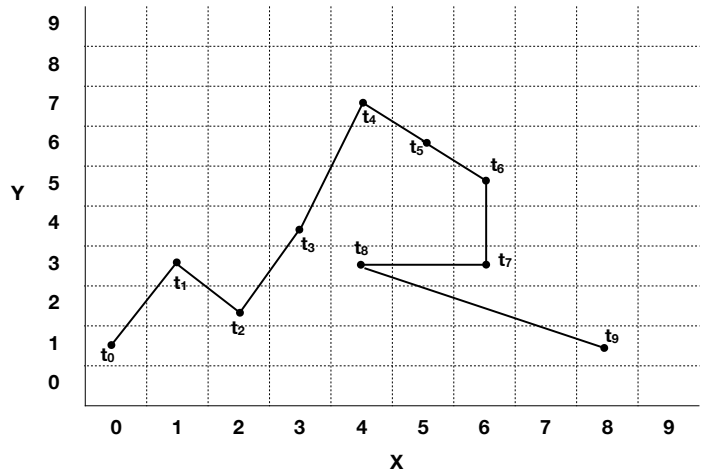


Figure 2. Example trajectory of an object over a two-dimensional space.

the input region $[x_1, x_2] \times [y_1, y_2]$ can partially intersect some cells of the raster. By including those cells, our index can retrieve some spurious objects from the expanded region $[x_1 - c, x_2 + c] \times [y_1 - c, y_2 + c]$. Finally, in nearest neighbor queries we can return objects that are off being the nearest ones by at most $c\sqrt{2}$. This worst case occurs when two objects are located in a diagonal at the nearest and farthest corners of the same cell, so they are indistinguishable after discretizing their positions.

In ContaCT, the input data are already in raster form, with the cell size defined depending on the kind of objects to handle. Therefore, those error bounds are assumed by the domain. In our experimental evaluation, we show how the size and time performance of ContaCT evolves with different cell sizes.

ContaCT is structured into *snapshots* of the space at regular time intervals and *logs* to represent the movements of each object trajectory. Unlike in GraCT, the log of ContaCT on its own is sufficient at answering object and trajectory queries (as well as the new MBR queries); the snapshots are used only to speed up range and nearest neighbor queries.

Each snapshot is represented using the k^ρ-tree data structure described in Section 3.4. The log data for an object is, in essence, the description of its consecutive movements along each coordinate using the constant-time partial sums data structure described in Section 3.2. This allows reconstructing the position of any object at any time instant in constant time. Further, the logs are enriched with the rmq/rMq structures described in Section 3.3. These allow computing the MBR containing the positions of any object during any period of time, which is useful for speeding up range queries and, as explained, it is interesting in itself as an independent query.

We now describe the log and snapshot structures in detail.

4.1. *The log representation*

The movements of each object along each dimension D is a sequence of positive and negative values (i.e., increasing or decreasing the object position along that dimension). We represent them in unary using two bitmaps, D_p and D_n. For each positive movement of c cells, we append c 0s and a 1 to D_p, and a 1 to D_n. For each negative movement of c cells, we append a 1 to D_p and c 0s followed by a 1 to D_n. A zero

movement implies then appending a 1 to both D_p and D_n.

This allows us to easily compute the position of the object along dimension D in time instant t in constant time, as shown in Section 3.2:

$$d_0 + (select_1(D_p, t) - t) - (select_1(D_n, t) - t) \;=\; d_0 + select_1(D_p, t) - select_1(D_n, t),$$

where d_0 is the position at time $t = 0$.

Figure 2 shows an example trajectory in dimensions X and Y, along time instants t_0 to t_9. The absolute positions of the object along those time instants are:

$$\begin{aligned} A_X[0..9] &= 0, 1, 2, 3, 4, 5, 6, 6, 4, 8 \\ A_Y[0..9] &= 1, 3, 2, 4, 7, 6, 5, 3, 3, 1 \end{aligned}$$

and these correspond to the following differential movements ($A_D[i] - A_D[i-1]$), starting from position $(x_0, y_0) = (0, 1)$ at time instant t_0:

$$\begin{aligned} \Delta_X[1..9] &= 1, 1, 1, 1, 1, 1, 0, -2, 4 \\ \Delta_Y[1..9] &= 2, -1, 2, 3, -1, -1, -2, 0, -2 \end{aligned}$$

We then represent the trajectory with the following four bitmaps: two for dimension X and two for dimension Y:

$$\begin{aligned} X_p &= 0101010101011100001 \\ X_n &= 11111110011 \\ Y_p &= 0011001000111111 \\ Y_n &= 1011101010011001 \end{aligned}$$

For instance, the position $(x_6, y_6) = (6, 5)$ of the object at t_6 is found with

$$\begin{aligned} x_6 \;=\; A_X[6] \;&=\; x_0 + select_1(X_p, 6) - select_1(X_n, 6) \;=\; 0 + 12 - 6 \;=\; 6, \\ y_6 \;=\; A_Y[6] \;&=\; y_0 + select_1(Y_p, 6) - select_1(Y_n, 6) \;=\; 1 + 13 - 9 \;=\; 5. \end{aligned}$$

Since we can compute the position of an object in $O(1)$ time, the snapshots are not used in ContaCT to compute the position of objects.

4.2. *The minimum bounding rectangle support*

For each object and dimension D, we include the support for range maximum/minimum queries described in Section 3.3.1. We choose this arrangement because objects tend to move in a specific direction for long time periods, therefore there are few local minima and maxima and the indexes tend to be small.

Let the array A_D be the array A of Section 3.3.1, which stores the absolute object positions in dimension D. ContaCT then represents (in principle, see later) the bitmaps $Bmax_D$ and $Bmin_D$ marking the local maxima and minima, respectively, and will simulate access to their associated arrays $Amax'_D$ and $Amin'_D$ by exploiting the fact

that it has access to A_D through bitmaps D_p and D_n:

$$\begin{aligned}
Amax'_D[i] &= A_D[select_1(Bmax_D, i) - 1] \\
&= d_0 + select_1(D_p, select_1(Bmax_D, i) - 1) - select_1(D_n, select_1(Bmax_D, i) - 1), \\
Amin'_D[i] &= A_D[select_1(Bmin_D, i) - 1] \\
&= d_0 + select_1(D_p, select_1(Bmin_D, i) - 1) - select_1(D_n, select_1(Bmin_D, i) - 1),
\end{aligned}$$

with the assumption that $select_1(D, 0) = 0$.

Continuing our example above, the corresponding structures for the arrays of positions A_X and A_Y are:

$$\begin{array}{lcllcl}
Bmax_X & = & 0000000101 & Amax'_X & = & 6{,}8 \\
Bmin_X & = & 1000000010 & Amin'_X & = & 0{,}4 \\
Bmax_Y & = & 0100100000 & Amax'_X & = & 3{,}7 \\
Bmin_Y & = & 1010000001 & Amin'_Y & = & 1{,}2{,}1
\end{array}$$

We can then, for example, compute $Amax'_X[2] = A_X[select_1(Bmax, 2) - 1] = A_X[9] = x_0 + select_1(X_p, 9) - select_1(X_n, 9) = 0 + 19 - 11 = 8$.

The coordinate range of the MBR of a trajectory between time instants t and t' is then found as

$$[A_D[rmq(A, t, t')], A_D[rMq(A, t, t')]].$$

The MBR of our example trajectory between time instants t_3 and t_8 is then found as

$$\begin{aligned}
& [A_X[rmq(A_X, 3, 8)], A_X[rMq(A_X, 3, 8)]] \times [A_Y[rmq(A_Y, 3, 8)], A_Y[rMq(A_Y, 3, 8)]] \\
= \; & [A_X[3], A_X[7]] \times [A_Y[3], A_Y[5]] \\
= \; & [2, 6] \times [2, 7].
\end{aligned}$$

Figure 3 illustrates the technique on the dimension Y of the trajectory of Figure 2, where a query on the interval $[t, t'] = [1, 6]$ is mapped onto $Amax'_Y[1..2]$ for the maxima and $Amin'_Y[1..1]$ for the minima.

Note that for each coordinate, D, the rMq and rmq operations on A_D are carried out with the bitmaps $Bmax_D$ and $Bmin_D$, using together $2|A_D|$ bits, and with the structures of $2|Amax'_D|$ bits (for rMq) plus $2|Amin'_D|$ bits (for rmq) built on the arrays $Amax'_D$ and $Amin'_D$. Access to the arrays A'_D and A_D themselves is provided with the bitmaps D_p and D_n, as explained.

To further save $|A_D|$ bits of space, we exploit the fact that local maxima and minima must alternate: we replace $Bmax_D$ and $Bmin_D$ by a single bitmap S_D, with $S_D[i] = 1$ iff $Bmax_D[i] = 1$ or $Bmin_D[i] = 1$ (see Figure 3). If the first bit of S_D came from $Bmax_D$, then $Amax'_D[i] = A_D[select_1(S_D, 2i - 1) - 1]$ and $Amin'_D[i] = A_D[select_1(S_D, 2i) - 1]$, and the other way if the first bit of S_D came from $Bmin_D$.

For example, in Figure 3, to obtain the MBR on the interval $[t, t'] = [1, 6]$, with S_y we observe that the first and last local minima/maxima are at time instants $rank_1(S_y, 1 - 1) + 1 = 2$ and $rank_1(S_y, 6) = 4$, respectively. Since the first 1-bit in S_y comes from $Bmin_y$, the second and fourth 1-bit correspond to the first and second local maxima. Therefore the local maximum is the maximum value in $Amax'_y[1..2]$

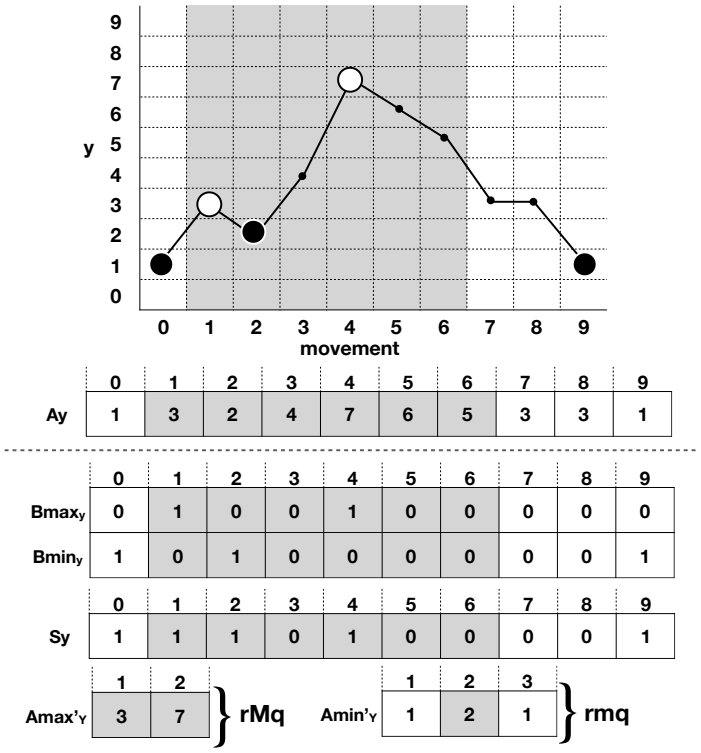


Figure 3. Illustration of the MBR computation on the Y coordinates.

(i.e., 7). Instead, the third 1-bit represents the second local minimum, thus the local minimum is $Amin'_y[2] = 2$. By comparing those local minima/maxima with the values at the extremes, as explained in Section 4.2, we obtain the global maximum and minimum: 7 and 2.

4.3. *Missing information*

Each object id stores a tuple $F(id) = \langle t_0, (x_0, y_0)\rangle$ with the initial time instant and position of its trajectory. Given a time instant t, and knowing the time sampling span d, an algebraic calculation yields the corresponding sample t_i, with $i = \lfloor (t - t_0)/d \rfloor$. The bitmaps $D_p(id)$ and $D_n(id)$ along each dimension D, plus the information to support rmq/rMq queries, then suffices to represent the log of each object id.

A complication is that, in some domains, GPS coordinates sent by an object can be erroneous or incomplete, because of network problems or disconnection of GPS devices. Although ContaCT does not address the problem of erroneous information (Zheng 2015), it can deal with missing information. In some contexts, the missing coordinates can be deduced by interpolation, but in others, it may be necessary to represent that lack of information.

To handle this situation, ContaCT stores an additional bitmap $T(id)$ that stores, for each time instant, whether there is (1) or not (0) information on the position of object id at that time instant. Every time interval we query is then first mapped with $rank_1$ operations on $T(id)$ before accessing the information stored. The bitvectors $D_p(id)$ and $D_n(id)$ and the information on maxima and minima are stored only with respect to the time instants where there is information on the position of object id.

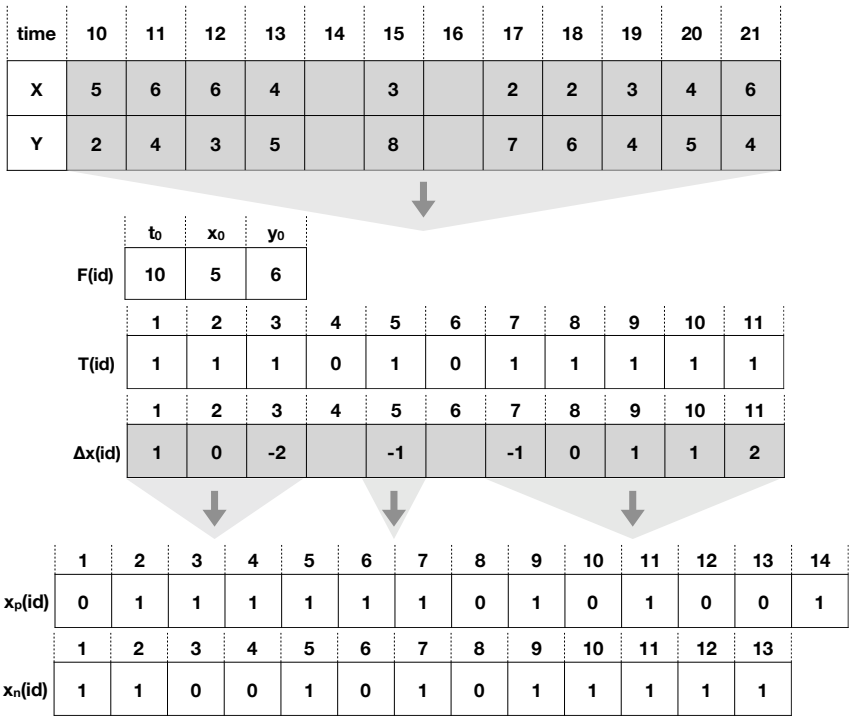


Figure 4. Example of log representation with missing information.

Figure 4 shows an example. On top we have a trajectory represented by arrays X and Y, which contain its coordinates. The bitmap $T(id)$ is then used to mark which positions have data. The array $\Delta_X(id)$ is built only on the mapped positions, and the bitmaps $X_p(id)$ and $X_n(id)$ are used to represent it.

Note that the differences are computed with respect to not exactly consecutive time instants, for example, $\Delta_X(id)[4]$ is computed as the difference between time instants 15 and 13 because the information at 14 is missing.

Nevertheless, we can still use our formulas on X_p and Y_p to compute the position of an object. For example, to know where the object is at time instant $t_8 = 18$, we first map it to $rank_1(T, 8) = 6$. We then recover the position $x_8 = 3$ with $A_X[8] = x_0 + select_1(X_p, 6) - select_1(X_n, 6) = 5 + 8 - 10 = 3$.

4.4. *The snapshots*

ContaCT stores a snapshot every d time instants. A snapshot at time t_s consists of:

- A k^ρ-tree marking the cells in the space that contain at least one object at time t_s.
- An array *perm* of integers storing the identifiers of the objects in the cells.
- A bitmap Q that maps the objects in *perm* to the marked cells.

Recall from Section 3.4 that the nonempty cells of the k^ρ-tree correspond to 1s in its bitvector L. We traverse L from left to right and, for each 1 found, which represents a cell where the objects $id_1, \ldots, id_k$ lie, we append those identifiers to the array *perm* and append $k-1$ 1s followed by a 0 to bitmap Q. It is then easy to see that the objects

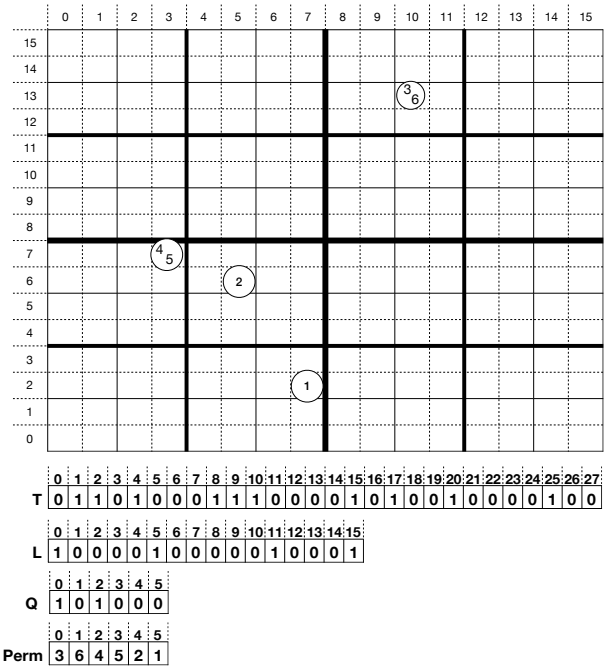


Figure 5. Structure of an example snapshot.

corresponding to some $L[i] = 1$ are $perm[l..r]$, with $l = select_0(Q, rank_1(L, i) - 1) + 1$ and $r = select\_next_0(Q, l)$.

The objects lying within a rectangular region R can be reported by simulating a top-down traversal of the k^ρ-tree, starting at the root and entering all the nonempty branches that overlap R. The areas covered by each k^ρ-tree node are algebraically computed as we descend. For each nonempty cell we find inside R, we list the corresponding objects as described.

Figure 5 shows the components of a snapshot whose k^2-tree is that of Figure 1. The circles in the grid indicate which objects lie inside each of the 4 nonempty cells. For example, according to the k^2-tree layout, the cell $(3, 7)$ corresponds to $L[5]$. This is the second leaf with objects because $rank_1(L, 5) = 2$. Its corresponding values are then in $perm$ between positions $select_0(Q, 2 - 1) + 1 = 2$ and $select_0(Q, 2) = select\_next_0(Q, 2) = 3$. Indeed, the objects are $perm[2..3] = \langle 4, 5 \rangle$.

4.5. *Total space*

Let a dataset consist of m objects on a ρ-dimensional grid of size s^ρ, moving along n time instants at a maximum speed $s_{\max}$, and including a events where an object disappears (and possibly reappears later). Assume we store a snapshot every d time instants. The breakdown of the space usage of ContaCT is as follows:

- The bitmaps $D_p(id)$ and $D_n(id)$ for each object id add up to $n(1 + \log s_{\max})$ each, for each dimension D. Each movement is written either in D_p or in D_n, however, so this adds up to $n(2 + \log s_{\max})$ for both bitmaps.
- The bitmap $S_D(id)$ uses n bits for each object id and dimension D.
- The rmq and rMq data structures use $2n'$ bits when there are n' minima/maxima. Since it always holds that $n' \leq n/2$, they add $2n$ bits per dimension and per

d	Distance between snapshots in number of time instants
$s_{\max}$	Speed of the fastest object in the dataset (maximum speed)
$\mathcal{S}_t$	Snapshot that stores the locations of the objects at time instant t
$D_p(id)$	Bitmap that stores in unary the positive movements of the object id along dimension D
$D_n(id)$	Bitmap that stores in unary the negative movements of the object id along dimension D
$F(id)$	Tuple that stores the initial time instant and position of object id
$T(id)$	Bitmap that marks the time instants with data of object id
$S_D(id)$	Bitmap that marks the positions of object id that are a local minima or maxima along dimension D
$extend(R, \delta)$	Maximum expansion of region R during δ time instants considering the maximum speed
$\overline{p_a\ p_b}$	Euclidean distance between points p_a and p_b
$\overline{R\ p_b}$	Minimum Euclidean distance between p_b and any possible point within the region R
$\overline{\overline{R\ p_b}}$	Maximum Euclidean distance between p_b and any possible point within the region R

Table 1. Summary of notations.

object.

- The bitmap $T(id)$ adds n bits per object.
- Each of the n/d snapshots is a k^ρ-tree with $k = 2$, where each node uses 2^ρ bits to signal its children. It then uses in total at most $2^\rho m \log s$ bits, when each object induces a separate path (of length $\log_{2^\rho} s^\rho = \log s$) in the tree. In addition, we store $perm$ and Q, adding $m(1 + \log m)$ bits per snapshot.
- The a times when objects disappear, and possibly reappear later, use $2a \log m$ bits.

The maximum space is then

$$\begin{aligned} & m(\rho(n(2 + \log s_{\max}) + n + 2n) + n + (n/d)(2^\rho \log s + 1 + \log m)) + 2a \log m \\ = \quad & mn\rho \log s_{\max} + mn(5\rho + 1) + (mn/d)(2^\rho \log s + 1 + \log m) + 2a \log m \end{aligned}$$

bits (plus an ignored sublinear term to support *rank*, *select*, and other queries). This space can be interpreted as follows:

- The component $mn\rho(1 + \log s_{\max})$ corresponds to a differentially encoded representation of the trajectories of all the objects, that is, a clever encoding of the raw data without support for queries (each movement is in $[-s_{\max}, +s_{\max}]$ per coordinate).
- The $4\rho + 1$ bits per movement (e.g., 9 bits in two dimensions) is the extra space needed to support queries on the trajectories.
- The $(2^\rho \log s + 1 + \log m)/d$ bits per movement is the extra space needed to support the spatial queries that speed up some operations. This can be controlled with the parameter d and it becomes small with very reasonable values for d.
- The $2a \log m$ bits can typically be ignored because a tends to be small. In addition, for every time instant an object is disappeared, we save $\rho(5 + \log s_{\max})$ bits from the trajectory data (D_p, D_n, S_D, rmq and rMq structures).

5. Queries

We now describe how the queries are solved in ContaCT. For simplicity, we will assume a two-dimensional space. For the formalizations, let us define the 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.

Algorithm 1: ObjectPosition$(id,\ t_q)$

```
1 if t_q = F(id).t_0 then return (F(id).x_0, F(id).y_0);
2 if T(id)[t_q − F(id).t_0] = 0 then return null;
3 j ← rank_1(T(id), t_q − F(id).t_0);
4 dx ← select_1(X_p(id), j) − select_1(X_n(id), j);
5 dy ← select_1(Y_p(id), j) − select_1(Y_n(id), j);
6 return (F(id).x_0 + dx, F(id).y_0 + dy);
```

5.1. *Object position*

This query returns the position of the object at a given time instant t_q, formally:

Definition 5.1. *The Object position query, for an object identifier id and a time instant t_q, returns the location p_q such that $\langle t_q, p_q\rangle \in \mathcal{T}_{id}$.*

The algorithm follows the procedure shown in the previous section. Algorithm 1 shows the pseudocode; we assume t_q is within bounds for simplicity. For example, in Figure 4, t_q can go from $t_0 = 10$ to $t_{11} = 21$.

In line 1, the algorithm checks if t_q is the first time instant represented by the log, in which case it returns the position (x_0, y_0) stored in F. Otherwise, we access position $t_q - F(id).t_0$ in $T(id)$ in line 2. If there is a 0, we have no information, and the algorithm returns null.

Otherwise, in line 3 the algorithm obtains the number of movements, j, until t_q. Lines 4–5 compute the cumulative movements in both coordinates over the first j time instants. Finally, in line 6, the position at t_q is obtained by adding the cumulative movements to the initial position.

5.2. *Object trajectory*

This query returns the positions of a given object *id* during a time interval $[t_b, t_e]$.

Definition 5.2. *The Object trajectory query, for an object identifier id and a time interval $[t_b, t_e]$, returns the sequence of locations $\langle t_i, p_i\rangle \in \mathcal{T}_{id}$ such that $t_b \le t_i \le t_e$, in increasing order of t_i.*

It first uses the basic method of the object position query to find the position of the object in the first valid time instant, not before t_b. It then traverses the bitmaps T, X_p, X_n, Y_p, and Y_n using the fast *select_next* operation (recall Section 3.1) to obtain each consecutive point until surpassing the time t_e.

Algorithm 2 shows the pseudocode; again we assume that $[t_b, t_e]$ is within bounds for simplicity. It returns in *result* all the tuples $\langle t, (x, y)\rangle$ of trajectory points at time t and spatial coordinates (x, y). Lines 3–10 obtain the first valid time position, j, not before t_b, and the corresponding positions xp, xn, yp, yn in the bitmaps X_p, X_n, Y_p, and Y_n. Those positions suffice to return any desired point. Line 11 also obtains, from j, the first valid time $t \in [t_b, t_e]$. The algorithm then computes the remaining positions until the time instant t_e (lines 12–16). At each iteration, it inserts a new tuple in *results* and finds the next valid time t and the next corresponding positions xp, xn, yp, and yn.

Algorithm 2: ObjectTrajectory(id, t_b, t_e)

```
1  t_0 ← F(id).t_0; x_0 ← F(id).x_0; y_0 ← F(id).y_0;
2  result ← ∅;
3  if t_b = t_0 then
4  |   j ← 1;
5  |   xp ← 0; xn ← 0;
6  |   yp ← 0; yn ← 0;
7  else
8  |   j ← 1 + rank_1(T(id), t_b − t_0 − 1);
9  |   xp ← select_1(X_p(id), j); xn ← select_1(X_n(id), j);
10 |   yp ← select_1(Y_p(id), j); yn ← select_1(Y_n(id), j);
11 t ← select_1(T(id), j);
12 while t_0 + t ≤ t_e do
13 |   result ← result ∪ ⟨t_0 + t, (x_0 + xp − xn, y_0 + yp − yn)⟩;
14 |   t ← select_next(T(id), t);
15 |   xp ← select_next(X_p(id), xp); xn ← select_next(X_n(id), xn);
16 |   yp ← select_next(Y_p(id), yp); yn ← select_next(Y_n(id), yn);
17 return result;
```

5.3. *Minimum bounding rectangle*

The Minimum bounding rectangle (MBR) is the minimum rectangular area that contains the trajectory of an object during an interval of time $[t_b, t_e]$; formally:

Definition 5.3. *For 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 $\langle t_i, p_i \rangle \in \mathcal{T}_{id}$ with $t_b \leq t_i \leq t_e$, it holds that $p_i \in R$.*

ContaCT can efficiently compute the MBR of a trajectory thanks to the rmq and rMq data structures, as shown in Section 4.2. This is an interesting query because it provides summary information about the path followed by an object without computing the whole trajectory. In addition, it is used as a tool to efficiently compute other queries.

Algorithm 3 shows how MBR(id, t_b, t_e) returns the smallest axis-aligned rectangle that contains every point visited by the object *id* during the time interval $[t_b, t_e]$, which we assume for simplicity to be within bounds and to satisfy $t_b < t_e$ (if $t_b = t_e$, the query boils down to an object position query).

Lines 1–11 compute, using the same technique as for object trajectories, the first and last positions of the interval, j_b and j_e, after filtering out the invalid positions in $T(id)$, as well as the object positions, (x_b, y_b) and (x_e, y_e), at those extremes of the time interval of interest.

Once the positions at the extremes are computed, the algorithm looks for the minimum and maximum, in both coordinates, among the local minima and maxima, respectively, that occur in the range $[j_b, j_e]$. The desired values are obtained in lines 12–15 using procedure *best*, which calculates the smallest (resp. greatest) value from the local minima (resp. maxima). Line 16 then returns the global maximum and minimum in each dimension, by comparing the first value, the best of local minima/maxima, and the last value of the interval.

To compute *best*, the procedure receives the corresponding bitvector B, the operation to carry out (Op, which is either rmq or rMq over the corresponding conceptual array A'), and the structures to compute values of the conceptual array A (d_0, D_p, and D_n). In lines 19–21, it uses B to compute the range $[pos_b, pos_e]$ of the array A' where the local minima/maxima lie (it returns *null* if no local values exist in the range;

Algorithm 3: MBR$(id,\ t_b,\ t_e)$

```
1  t_0 ← F(id).t_0; x_0 ← F(id).x_0; y_0 ← F(id).y_0;
2  if t_b = t_0 then
3  |   j_b ← 1; x_b ← x_0; y_b ← y_0;
4  else
5  |   j_b ← 1 + rank_1(T(id), t_b − t_0 − 1);
6  |   x_b ← x_0 + select_1(X_p(id), j_b) − select_1(X_n(id), j_b);
7  |   y_b ← y_0 + select_1(Y_p(id), j_b) − select_1(Y_n(id), j_b);
8  j_e ← rank_1(T(id), t_e − t_0);
9  x_e ← x_0 + select_1(X_p(id), j_e) − select_1(X_n(id), j_e);
10 y_e ← y_0 + select_1(Y_p(id), j_e) − select_1(Y_n(id), j_e);
11 if j_b > j_e then return null;
12 x_max ← best(Bmax_X, j_b, j_e, rMq(Amax'_X), x_0, X_p, X_n);
13 x_min ← best(Bmin_X, j_b, j_e, rmq(Amin'_X), x_0, X_p, X_n);
14 y_max ← best(Bmax_Y, j_b, j_e, rMq(Amax'_Y), y_0, Y_p, Y_n);
15 y_min ← best(Bmin_Y, j_b, j_e, rmq(Amin'_Y), y_0, Y_p, Y_n);
16 return [min(x_b, x_min, x_e), max(x_b, x_max, x_e)] × [min(y_b, y_min, y_e), max(y_b, y_max, y_e)];
17
18 best(B, j_b, j_e, Op, d_0, D_p, D_n)
19   pos_b ← 1 + rank_1(B, j_b − 1);
20   pos_e ← rank_1(B, j_e);
21   if pos_b > pos_e then return null;
22   j_best ← Op(pos_b, pos_e);
23   pos_best ← select_1(B, j_best);
24   return d_0 + select_1(D_p, pos_best) − select_1(D_n, pos_best);
```

we assume *null* does not participate in the minima/maxima of line 16). In line 22, it obtains the largest/smallest maximum/minimum position j_{best} in $A'[pos_b..pos_e]$, and in line 23 it maps j_{best} to its corresponding position pos_{best} in A. Finally, in line 24 it returns the corresponding value $A[pos_{best}]$; recall that we assume $select_1(D, 0) = 0$.

We are using the bitmaps $Bmax_D$ and $Bmin_D$ for simplicity in the pseudocode but, as explained, we actually store a combined bitmap S_D. Procedure *best* should then receive S instead of B, and compute pos_b and pos_e in lines 19–20 over S. They should then be corrected as follows: if the first bit of S is a minimum/maximum and we are looking for a minimum/maximum, respectively, then $pos_b \leftarrow 1 + \lfloor pos_b/2 \rfloor$ and $pos_e \leftarrow \lceil pos_e/2 \rceil$. Otherwise, $pos_b \leftarrow \lceil pos_b/2 \rceil$ and $pos_e \leftarrow \lfloor pos_e/2 \rfloor$.

5.4. *Time slice*

Given a rectangular region of the space R and a time instant t_q, a time slice query returns all the objects that are within R at time t_q, more formally:

Definition 5.4. *For a rectangular region R and a time instant t_q, the slice query returns the set O of object identifiers such that, for each $id \in O$, there exists a pair $\langle t_q, p_q \rangle \in \mathcal{T}_{id}$ where $p_q \in R$.*

ContaCT uses snapshots to efficiently solve this query. Recall that snapshots are placed at regular time intervals. Since there is likely no snapshot at t_q, we use the closest snapshot to filter the objects that have chances of reaching R at t_q; these objects are called *candidates*. To check if a candidate is within R at time t_q, we simply compute its position at time t_q using the query *ObjectPosition*.

Finding the smallest possible set of candidates is the key factor for the efficiency of this query. To do this, we take into account the maximum speed of an object in our dataset, $s_{\max}$. For example, if an object is at $(3, 8)$ and $s_{\max} = 1$, then after

Algorithm 4: TimeSlice(R, t_q)

```
1  l ← ⌊t_q/d⌋ · d;
2  r ← l + d;
3  if t_q = l then return S_l.region(R) ;
4  candidates ← ∅;
5  result ← ∅;
6  if t_q − l < r − t_q then
7  |  candidates ← S_l.region(extend(R, t_q − l)) ∪ S_l.app;
8  else
9  |  candidates ← S_r.region(extend(R, r − t_q)) ∪ S_r.dis;
10 for c ∈ candidates do
11 |  p ← ObjectPosition(c, t_q);
12 |  if p ∈ R then result ← result ∪ {c} ;
13 return result;
```

2 time instants the object cannot reach the region $[6, 10] \times [7, 12]$. Since we know the position of all the objects at the time instant t_s of a snapshot, we can obtain the candidates as those objects that are within the region R' at time t_s, where R' is the result of expanding R by $s_{\max} \times |t_q - t_s|$ positions in all directions. That is, $R' = extend(R, |t_q - t_s|)$, where

$$extend([b_x, e_x] \times [b_y, e_y], \delta) \quad = \\ [b_x - \delta \cdot s_{\max}, e_x + \delta \cdot s_{\max}] \times [b_y - \delta \cdot s_{\max}, e_y + \delta \cdot s_{\max}].$$

Notice that the smaller the time difference δ, the smaller R' will be, and thus the smaller the set of candidates will tend to be. Let us define $\mathcal{S}_t$ as the snapshot at time instant t. In order to minimize the number of candidates, we query the snapshot closest to t_q.

Algorithm 4 implements this query. It first finds the two snapshot times, l and r, that surround the queried time instant, t_q. If $t_q = l$, we simply return the set of objects in R using the query $region(R)$ on the data structure of snapshot $\mathcal{S}_l$. For the typical case, lines 6–9 choose the closest snapshot and query it for the extension of R that captures all the possible candidates. Lines 10–13 then check the candidates one by one to report those that are inside R at time t_q.

Because we do not have position information for all the objects at all time instants, there may be objects that are within R at time t_q, but they have no information at the chosen snapshot. To handle this problem, we store two additional arrays of object identifiers, *app* and *dis*, for each snapshot $\mathcal{S}_t$: *app* contains the identifiers of the objects that are not in $\mathcal{S}_t$, but appear in the log after time instant t and before $t + d$, whereas *dis* stores the objects that are in the log after time $t - d$ and before t, but are not represented in $\mathcal{S}_t$. For this reason, in lines 7 and 9, our candidates are extended with the objects in $\mathcal{S}_l.app$ or $\mathcal{S}_r.dis$, depending on the case.

Note that arrays *app* and *dis* are closely related with the bitmaps $T(id)$. An object identifier id is added to $\mathcal{S}_t.app$ when the entry corresponding to t in $T(id)$ is set to 0 and there is a 1 before the next snapshot, that is, within $T(id)[t + 1, t + d - 1]$. Similarly, to be in $\mathcal{S}_t.dis$, the 1 has to be within $T(id)[t - d + 1, t - 1]$.

5.5. *Time interval*

Time interval queries obtain the objects that were within a rectangular region R at some time instant of an interval $[t_b, t_e]$, that is:

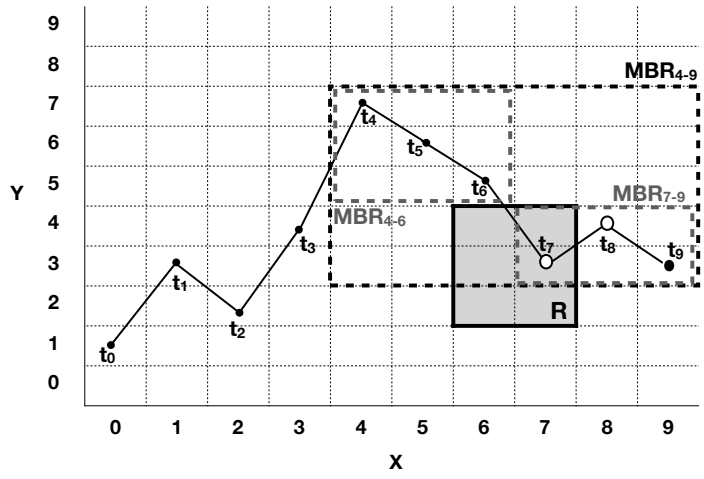


Figure 6. Example of a time interval query over a trajectory.

Definition 5.5. *For a rectangular region R and a time interval $[t_b, t_e]$, the time interval query returns the set O of object identifiers 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 $p_i \in R$.*

A naive approach to solving this query adapts the *TimeSlice* algorithm so that (i) the region is extended considering time instant t_e (if we use $\mathcal{S}_l$) or t_b (if we use $\mathcal{S}_r$), and (ii) for each candidate, we compute the positions of the trajectory from t_b until the object is inside R or we reach t_e. This approach, however, is affordable only if t_e is close to t_b.

A more efficient approach for large timespans makes use of the *MBR* query. If we compute the MBR of the object along $[t_b, t_e]$ and it is totally within R, we do know that the object appears inside R during the interval. Otherwise, if the MBR is disjoint from R, we do know that the object does not appear inside R during the interval. Otherwise, the object may or may not appear in R. We then divide the interval in two halves, $[t_b, t_m]$ and $[t_m + 1, t_e]$, and proceed recursively in both.

Since computing MBRs requires a number of calculations, we switch to the naive approach as soon as the length of the time interval falls below a parameter λ.

Figure 6 illustrates an example for $[t_b, t_e] = [t_4, t_9]$, $R = [6, 7] \times [2, 4]$, and $\lambda = 3$. We first compute the MBR enclosing the movements between time instants t_4 and t_9, $MBR(t_4, t_9) = [4, 9] \times [3, 7]$. Since this MBR intersects R, but is not contained in it, we must split the time interval into $[t_b, t_m] = [t_4, t_6]$ and $[t_m + 1, t_e] = [t_7, t_9]$, and consider both halves. Since $MBR(t_4, t_6) = [4, 6] \times [5, 7]$ is disjoint with R, we can abandon this time interval. On the other hand, $MBR(t_7, t_9) = [7, 9] \times [3, 4]$ intersects R but is not contained in it, so we must continue considering this time interval. Now, when this interval is divided into $[t_7, t_8]$ and $[t_9, t_9]$, both are shorter than λ, so we proceed on them point by point. We finally report the object because it is within R at time t_7.

A final note is that the interval $[t_b, t_e]$ can span several consecutive snapshots, thus we can split $[t_b, t_e]$ into several subintervals, according to which is the closest snapshot for each time instant. In each such snapshot, the region R must then be extended by at most $(d/2) \cdot s_{\max}$ units. All the objects that are in the expanded range R' of some intervening snapshot are verified. This yields far fewer candidates than if we take only one snapshot and extend it according to the whole length of the time interval.

Once we have to verify an object coming from the expanded range R' of some

Algorithm 5: TimeInterval(R, t_b, t_e)

```
1  checked ← ∅;
2  l ← ⌊t_b/d⌋; r ← ⌈t_e/d⌉;
3  for i ∈ [l, r] do
4  |   t ← i · d;
5  |   bwd ← max(t_b, t − ⌊d/2⌋);
6  |   fwd ← min(t_e, t + ⌈d/2⌉);
7  |   if bwd ≤ fwd then
8  |   |   candidates ← S_t.region(extend(R, max(fwd − t, t − bwd)));
9  |   |   if t > l · d then candidates ← candidates ∪ S_t.dis;
10 |   |   if t < r · d then candidates ← candidates ∪ S_t.app;
11 |   |   for c ∈ candidates do
12 |   |   |   if c ∉ checked then
13 |   |   |   |   if Contained(c, R, t_b, t_e) then result ← result ∪ {c} ;
14 |   |   |   |   checked ← checked ∪ {c};
15 return result
```

Algorithm 6: Contained(c, R, t_b, t_e)

```
1  if t_e − t_b < λ then
2  |   T ← ObjectTrajectory(c, t_b, t_e)
3  |   for ⟨t, p⟩ ∈ T do
4  |   |   if p ∈ R then return true;
5  |   return false
6  else
7  |   mbr ← MBR(c, t_b, t_e)
8  |   if mbr ⊆ R then return true ;
9  |   if mbr ∩ R = ∅ then return false ;
10 |   t_m ← t_b + ⌊(t_e − t_b)/2⌋
11 |   if Contained(c, R, t_b, t_m) or Contained(c, R, t_m + 1, t_e) then return true ;
12 |   return false
```

snapshot, however, we directly verify it for the whole time interval $[t_b, t_e]$. We are careful to not work again on an object that has already been considered in a previous snapshot, by maintaining a set *checked* of the already verified objects.

Algorithm 5 gives the pseudocode. Line 2 computes the range of snapshots that surround the interval $[t_b, t_r]$, which will be queried for the timestamps closest to each. For each of those snapshots the algorithm obtains the object candidates in the interval of time $[bwd, fwd]$ (lines 7–10). The procedure continues checking each candidate object that has not been checked already, reporting those that fall inside R at some time in $[t_b, t_e]$ (lines 11–14).

The check for candidates is done by Algorithm 6, which is of independent interest. If the time interval is shorter than λ, it obtains the trajectory of the object and checks every position one by one, in lines 2–5 (in practice we check if $p \in R$ for every consecutive point as we obtain it from the trajectory, so we may preempt the extraction of the points). Otherwise, we carry out the described recursive procedure using MBRs, in lines 7–12.

5.6. *Nearest neighbor queries*

This query returns the K objects that are nearest to a given point p_q at a given time instant t_q, formally:

Definition 5.6. *For a point p_q at time instant t_q, the K-Nearest neighbor query re-*

turns 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., p such that $\langle p, t_q \rangle \in \mathcal{T}_{id}$).

The query can be naively solved by computing $ObjectPosition(c, t_q)$ for all objects c and retaining those at minimum distance to p_q. While a nearby snapshot should be useful to focus on some candidate objects only, no simple criterion like the one we used for *TimeSlice* can be used here.

From the various methods in the literature, we choose one that enjoys several good properties, like minimizing the number of distances computed (Bustos and Navarro 2009). It assumes that one has a hiearchical partition of the space at t_q (which we do not have in general). It uses a priority queue of *candidate regions*, Q_c, and a priority queue of *best known results*, Q_r.

The queue Q_r is a max-priority queue that stores objects sorted by their distances to p_q. It retains only the K objects closest to p_q seen so far: Once Q_r reaches size K, we remove from Q_r the farthest object after every new insertion. That is, we insert the new object and then remove the first (i.e., largest distance) object from the queue. The queue also gives, in constant time, the distance to the K-th closest object seen so far, which decreases as the search progresses.

The queue Q_c, instead, is a min-priority queue. It contains nodes of the hierarchy, where internal nodes stand for regions of the space and leaves stand for individual objects. The queue sorts the nodes by their minimum possible distance to p_q (which is zero if p_q is inside the node region). From Q_c we can extract the region closest to p_q at any step.

The algorithm starts with Q_c containing the root region and Q_r being empty. At every iteration, it extracts the closest region from Q_c. If it is a single object (i.e., a leaf of the hierarchy), the object is inserted into Q_r. Otherwise, the children nodes of the region are reinserted in Q_c. The algorithm stops as soon as the minimum possible distance from the closest region of Q_c is not smaller than the Kth distance to the results already known in Q_r. At this point, Q_r is the answer to the query.

The algorithm performs best if one manages to find close results fast. From the various heuristics to do this (Bustos and Navarro 2009), we choose to break ties between equally close regions by the maximum possible distance between the regions and p_q.

While we do not have a spatial index of the space at time t_q, we can obtain an approximate version from the k^ρ-tree of the nearest snapshot $\mathcal{S}_t$. This structure does induce a hierarchical partition of the space, where regions of the same level do not overlap. By assuming that the regions are extended by $|t - t_q| \cdot s_{\max}$ units in all directions, we obtain a hierarchy of regions that can be used to implement the algorithm correctly.

To formalize the algorithm we introduce some additional notation:

- $\overline{p_a\ p_b}$ denotes the Euclidean distance between points p_a and p_b.
- $\underline{R\ p_b}$ is the minimum Euclidean distance between p_b and any possible point within the region R, being zero if $p_b \in R$. It can be easily computed in constant time.
- $\overline{R\ p_b}$ is the maximum Euclidean distance between p_b and any possible point within the region R. It can also be computed in constant time.

Given a region R, a time interval $[t, t_q]$, and a position p_q, the minimum Euclidean distance to p_q that any object within R at time t can reach during $[t, t_q]$ can be computed as $\underline{extend(R, |t - t_q|)\ p_q}$. The maximum possible distance can be computed

Algorithm 7: Knn$(K,\ p_q,\ t_q)$

```
1  Q_r ← ∅, capped to size K; Q_c ← ∅;
2  l ← ⌊t_q/d⌋ · d; r ← l + d;
3  if t_q − l < r − t_q then
4  |   t ← l;
5  |   for a ∈ S_l.app do
6  |   |   p_a ← ObjectPosition(a, t_q);
7  |   |   Q_r.add(⟨a, p_a p_q⟩);
8  else
9  |   t ← r;
10 |   for d ∈ S_r.dis do
11 |   |   p_d ← ObjectPosition(d, t_q);
12 |   |   Q_r.add(⟨d, p_d p_q⟩);
13 Q_c.add(⟨S_t.root, 0, +∞⟩);
14 while Q_c ≠ ∅ and (|Q_r| < K or Q_c.min < Q_r.max) do
15 |   ⟨node, d_min, d_max⟩ ← Q_c.extractMin;
16 |   if node is a k^ρ-tree leaf then
17 |   |   for c ∈ node.objects do
18 |   |   |   p_c ← ObjectPosition(c, t_q);
19 |   |   |   Q_r.add(⟨c, p_c p_q⟩);
20 |   else
21 |   |   for nonempty node′ ∈ node.children do
22 |   |   |   R ← extend(node′.region, |t − t_q|);
23 |   |   |   Q_c.add(⟨node′, R p_q, R p_q⟩);
24 return Q_r;
25
```

similarly.

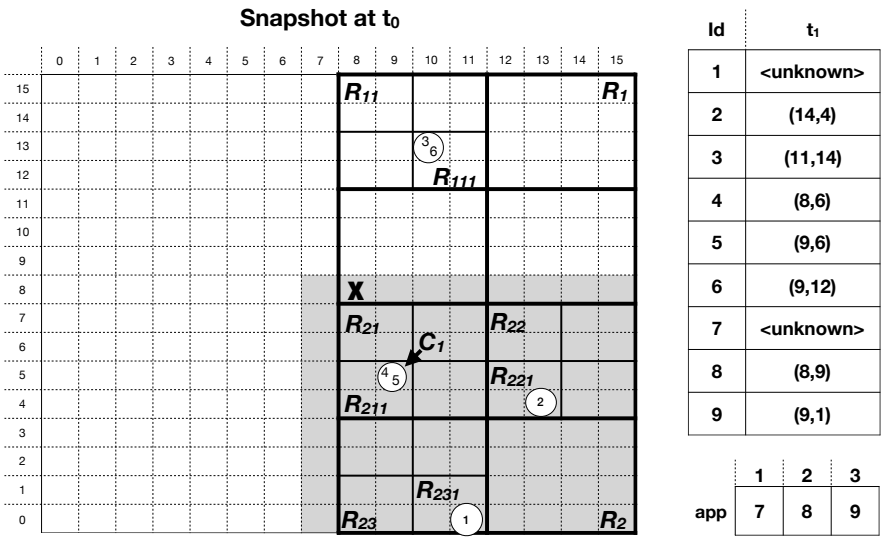
Algorithm 7 gives the pseudocode. We insert pairs $\langle c, d\rangle$ in Q_r, where c is the object and d its distance to p_q at time t_q. In Q_c we insert triples $\langle e, d_{\min}, d_{\max}\rangle$, where e is an object or a region and $d_{\min}$ and $d_{\max}$ are the minimum and maximum possible distances, respectively, from the object/region to p_q at time t_q. The queue Q_c sorts by $d_{\min}$ and breaks ties with $d_{\max}$.

Lines 1–3 initialize and choose the closest snapshot to use, $\mathcal{S}_t$. Lines 4–12 collect the spare objects that are not collected in the snapshot, but could exist in time t_q, and add them directly as candidates in Q_r. Line 13 initializes Q_c with the root of $\mathcal{S}_t$ and then we start the iterations, lines 14–23, until we process all the nodes in Q_c or the best candidate in Q_c has no better lower bound than the currently known Kth answer in Q_r: Line 15 extracts from Q_c the node with the closest region. If it is a leaf, then lines 17–19 insert its associated objects (*node.objects*, recall Section 4.4) in Q_r. Otherwise, it is an internal node, and lines 21–23 add its children back to Q_c, extending its regions (*node′.region*).

Figure 7 shows an example on our running snapshot, at t_0. The query asks for the $K = 3$ nearest neighbors of point $p_q = (8, 8)$ at time $t_q = t_1$. On the top left, the figure shows the hierarchical space partitioning induced by the snapshot k^2-tree. On the top right we show the actual positions of all the objects at t_1 (which the algorithm does not know). We also show the list *app* of the objects that appear between t_0 and the next snapshot. Assuming $s_{\max} = 1$, the table on the bottom shows the trace of the query.

First, Q_c is initialized with *root*, which covers the complete space, and Q_r includes those objects in *app* with information at t_1, that is, O_8 and O_9. The object O_7 is not added to Q_r because its position at t_1 is unknown.

In the first iteration, the *root* is extracted and its nonempty children, R_1 and R_2,



	Q_c	Q_r	$Q_r.max$
Step 0	$\langle root, 0\rangle$	$\langle O_8, 1\rangle, \langle O_9, 7.07\rangle$	∞
Step 1	$\langle R_1, 0\rangle, \langle R_2, 0\rangle$	$\langle O_8, 1\rangle, \langle O_9, 7.07\rangle$	∞
Step 2	$\langle R_2, 0\rangle, \langle R_{11}, 3\rangle$	$\langle O_8, 1\rangle, \langle O_9, 7.07\rangle$	∞
Step 3	$\langle R_{21}, 0\rangle, \langle R_{11}, 3\rangle, \langle R_{22}, 3\rangle, \langle R_{23}, 4\rangle$	$\langle O_8, 1\rangle, \langle O_9, 7.07\rangle$	∞
Step 4	$\langle R_{211}, 2\rangle, \langle R_{11}, 3\rangle, \langle R_{22}, 3\rangle, \langle R_{23}, 4\rangle$	$\langle O_8, 1\rangle, \langle O_9, 7.07\rangle$	∞
Step 5	$\langle C_1, 2\rangle\langle R_{11}, 3\rangle, \langle R_{22}, 3\rangle, \langle R_{23}, 4\rangle$	$\langle O_8, 1\rangle, \langle O_9, 7.07\rangle$	∞
Step 6	$\langle R_{11}, 3\rangle, \langle R_{22}, 3\rangle, \langle R_{23}, 4\rangle$	$\langle O_8, 1\rangle, \langle O_4, 2\rangle, \langle O_5, 2.24\rangle$	2.24

Figure 7. Example of a nearest neighbor query.

are added back to Q_c. Their 8×8 regions are extended by 1 unit in each direction to compute their best distance to p_q. In Figure 7 the expanded region of R_2 is shaded. In both cases, the best distance is zero because p_q belongs to the expanded areas. The queue will then break the tie with their maximum distance, thus R_1 precedes R_2.

In Step 2, the algorithm extracts R_1, which has only one nonempty child, R_{11}. This is added back to Q_c with best distance 3. Step 3 extracts R_2 and reinserts its nonempty regions of size 4×4: R_{21}, R_{22}, and R_{23}. They are added to Q_c with their best distances 0, 3 and 4, respectively. R_{21} is extracted at Step 4, and it only contains a nonempty subregion of size 2×2, R_{211}. This is added to Q_c and becomes its head because of its best distance 2. The next step then splits R_{211}, reinserting only the cell C_1, since it is the only one with objects. Since the best distance of C_1 is 2, it is processed in Step 6. At this point, the objects included in C_1, O_4 and O_5, are added to Q_r with their correct distances at t_1 computed.

Since Q_r is capped to size 3, object O_9 is removed and we have for the first time $K = 3$ results. From now on, we are not interested in any object farther than $\overline{p_q O_5} \approx 2.24$ from p_q. Because the best possible distance in Q_c is 3, we stop and return Q_r.

The presented algorithm is different from the one used in GraCT (Brisaboa *et al.* 2019), where Q_c only contains objects and the procedure is divided into two phases: (i) traversal of the snapshot and (ii) traversal of the log from the time instant of the snapshot until t_q. In the first phase, Q_c is filled with those objects in *app/dis* and the candidates obtained by traversing the snapshot, in increasing distance of the nodes towards p_q. Objects in Q_c are stored along with their position at their first time instant (that of the snapshot, except for the objects in *app/dis*). In the second phase, when an object is extracted from Q_c, the net movement of the next nonterminal of the log

is added, obtaining a new position, and then it is inserted back in Q_c with a new priority. When a candidate reaches time t_q, it is inserted in Q_r as in our algorithm. The algorithm stops when no candidate, even moving at maximum speed, can get closer to p_q, at time t_q, than the current Kth candidate.

Notice that the main difference between both algorithms is in the priority queue of candidates Q_c. In the experimental evaluation we compare GraCT and both algorithms on ContaCT.

6. Experimental evaluation

We compare ContaCT with GraCT and MVR-tree. GraCT (Brisaboa *et al.* 2019) is a representative of the modern compact data structures, which combine compression and fast access to the data, whereas MVR-tree (Tao and Papadias 2001b) is one of the best exponents of the classical approaches, where the speed is the main goal, without worrying too much about space. MVR-tree is a spatio-temporal index designed to solve *range* and *nearest neighbor* queries. It can be combined with other structures, as done with MV3R-tree (Tao and Papadias 2001b), which adds auxiliary 3DR-trees for speeding up basic queries like retrieving the trajectory of an object.

For our experimental evaluation, we implemented ContaCT in C++, using components from the SDSL library[2] (Gog *et al.* 2014). We used the author's implementation of GraCT, also in C++ using SDSL library. This implementation uses a balanced version of Re-Pair by G. Navarro[3] to build the grammar, and represents the extra information on nonterminals using DACs (Brisaboa *et al.* 2013) with an unlimited number of levels and without a predefined chunk size. We used the MVR-tree of the spatialindex library with default parameters (the capacity of each node set to 10 records and the fill factor set to 70%).[4] The experiments were conducted 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 `-O9` optimization.

For supporting *rank* operations on bitmaps represented in plain form, we use an implementation that requires 6.25% of additional space. 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. Bitmaps $T(id)$ and S_c tend to be sparse, thus they are always implemented with *sdarray*. The rmqs and rMqs are implemented following the latest results (Ferrada and Navarro 2017).

6.1. *Datasets*

During the experimental evaluation, we used four sources of data, three containing real-world data and the other with pseudo-real data:

- `Ships`: a real dataset obtained from MarineCadastre.[5] The data contain the location of 4,461 vessels traveling inside the UTM Zone 10 during one month of 2017.

[2] `https://github.com/simongog/sdsl-lite`
[3] `http://www.dcc.uchile.cl/gnavarro/software/repair.tgz`
[4] `http://libspatialindex.github.io`
[5] `http://marinecadstre.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 2. Datasets and their dimensions.

- `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 Network*.[6]
- `Taxis`: a pseudo-real dataset containing trajectories of 433 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*.[7]
- `Ciconia`: a small and non-repetitive real dataset of 88 white storks traveling between Europe and North Africa from 2013 to 2019. The original data can be obtained from *MoveBank Data Repository* (Flack *et al.* 2016, Cheng *et al.* 2019).

ContaCT deals with raster data, that is, every position is stored into a discrete grid. Every location sent by an object is then represented by the corresponding cell of that grid. The size of the cell is chosen depending on the application, with smaller cells when more detail is required. In our datasets, following recommendations that depend on the sizes of objects we index,[8] the sizes were set to 10×10 meters in `Ships`, 100×100 meters in `Planes`, and 1×1 meters in `Taxis` and in `Ciconia`. The impact of the precision of the cells in the space and time performance is discussed in Section 6.4. The time also requires a normalization because each object emits its location with different frequency, and our structure needs to synchronize those positions at regular time instants. We used regular intervals of 1 minute for `Ships`, 15 seconds for `Planes` and `Taxis`, and 6 minutes for `Ciconia`, which matches the (approximate) frequency in the actual data (and we used the smallest of the sampling intervals for `Taxis`).

After the discretization, we detected some errors in several locations of our real datasets (`Ships`, `Planes` and `Ciconia`), because some movements would require extremely high speed. To filter these errors, a maximum speed parameter was set for each dataset: 800 km/h in `Planes`, 234 km/h for `Ships` and 54 km/h in `Ciconia`. As a consequence, those signals that imply exceeding the maximum speed were removed: 0.01%, 0.83% and 0.42% of the signals were deleted from `Ships`, `Planes` and `Ciconia`, respectively. In addition, an object can emit its signals with different frequency; for example, in `Ships`, the frequency is lower when they are in a port. The asynchronism between the frequency of emission of GPS devices and the regular time instants considered in ContaCT makes it possible that there are some time instants without

[6] `https://opensky-network.org`

[7] `http://chriswhong.github.io/nyctaxi/`

[8] `https://en.wikipedia.org/wiki/Decimal_degrees`

	Index	ContaCT						GraCT					
	d	30	60	120	240	360	720	30	60	120	240	360	720
Ships	Size	141.82	131.44	126.23	123.62	122.74	121.87	121.92	82.43	62.33	52.25	48.85	45.40
	Snap	20.86	10.48	5.27	2.66	1.79	0.91	23.96	12.03	6.05	3.05	2.04	1.04
	Log	120.96	120.96	120.96	120.96	120.96	120.96	97.95	70.40	56.28	49.20	46.81	44.36
	R-plain	10.0%	9.30%	8.93%	8.75%	8.68%	8.62%	8.63%	5.83%	4.41%	3.70%	3.46%	3.21%
	R-bin	26.19%	24.27%	23.31%	22.83%	22.67%	22.50%	22.51%	15.22%	11.51%	9.65%	9.02%	8.38%
Planes	Size	197.91	182.78	175.22	171.41	170.15	168.88	205.74	136.39	101.66	84.21	78.41	72.57
	Snap	30.40	15.28	7.71	3.91	2.65	1.37	35.59	17.87	9.00	4.56	3.08	1.58
	Log	167.51	167.51	167.51	167.51	167.51	167.51	170.15	118.51	92.65	79.66	75.33	70.98
	R-plain	24.46%	22.59%	21.66%	21.19%	21.03%	20.87%	25.43%	16.86%	12.57%	10.41%	9.69%	8.97%
	R-bin	56.48%	52.16%	50.00%	48.92%	48.56%	48.20%	58.72%	38.92%	29.01%	24.03%	22.38%	20.71%
Taxis	Size	192.05	160.54	144.77	136.88	134.25	131.61	149.52	104.76	83.78	73.21	69.67	66.13
	Snap	63.07	31.56	15.79	7.90	5.27	2.64	66.88	33.46	16.75	8.38	5.59	2.79
	Log	128.97	128.97	128.97	128.97	128.97	128.97	82.64	71.30	67.03	64.83	64.09	63.33
	R-plain	18.75%	15.68%	14.14%	13.37%	13.11%	12.85%	14.60%	10.23%	8.18%	7.15%	6.80%	6.46%
	R-bin	45.07%	37.68%	33.98%	32.13%	31.51%	30.89%	35.10%	24.59%	19.66%	17.18%	16.35%	15.52%
Ciconia	Size	37.25	30.17	26.97	25.02	24.37	23.72	44.31	30.91	24.10	20.54	19.35	18.16
	Snap	14.19	7.11	3.90	1.95	1.30	0.65	15.27	7.65	4.20	2.10	1.40	0.70
	Log	23.07	23.07	23.07	23.07	23.07	23.07	29.04	23.26	19.91	18.44	17.95	17.46
	R-plain	41.38%	28.18%	25.18%	23.36%	22.76%	22.15%	34.79%	28.86%	22.51%	19.18%	18.07%	16.95%
	R-bin	88.98%	72.07%	64.41%	59.75%	58.20%	56.65%	105.8%	73.83%	57.57%	49.06%	46.22%	43.36%

Table 3. Structure sizes (in MB) and compression ratios.

information about the position of the object. In the cases where the difference between two consecutive signals is less than 15 time instants (of those actually considered by ContaCT), we interpolate the locations of those time instants.

Trajectories are usually stored in a plain text file composed of four columns: *object identifier*, *time instant*, *x coordinate*, and *y coordinate*. To obtain a fair comparison, we stored all this information in binary form by using the minimum number of bytes required for each column. For example, in `Ships`, two bytes are used to represent the first column (max value 4,461), two for the time instant column (max value 44,639), two for the x-axis (max value 6,000), and three for the y-axis (max value 647,755).

Table 2 shows a description of the datasets, their binary and plain text size, and their size after compressing them with *p7zip*. The last row gives us an idea of how compressible the data is; we observe that *p7zip* compresses the data to 10%–30%[9] of its binary representation.

6.2. *Compression*

We first analyze the compression ratios of ContaCT and compare them with those of GraCT. We applied ContaCT and GraCT on each dataset, using distances between snapshots $d = 30$, 60, 120, 240, 360, and 720 time instants. The value d can be adjusted depending on the type of application, as it provides a space/time tradeoff. As we show below, as d decreases the size of the structure increases and its time performance improves in most of the queries. Thus, an application that focuses on performance may choose the smallest d that allows keeping the whole structure within the available memory, whereas if the goal is compression one can use the maximum d that yields acceptable time performance.

In Table 3, the first row of each collection shows the size of each configuration depending on their d value, whereas the second and third rows break down the total space into that required by the snapshots and the compressed log. ContaCT represents D_p and D_n using plain bitmaps on `Ships` and sparse bitmaps on `Planes`, `Taxis` and

[9]The values are the size of the compressed file as a percentage of the size of the original file.

`Ciconia`. We observe that most of the space is occupied by the compressed log.

In the case of GraCT, the log reduces its size when d increases but, as expected, this does not occur in the case of ContaCT. In GraCT, the quotient of the log space with $d = 720$ versus $d = 30$ is 0.4 on `Planes`, whereas with ContaCT it is 1.0. As d increases, the length of the sections of the log (between snapshots) also increases, thus GraCT's grammar compressor finds more repetitiveness. ContaCT, instead, does not exploit this redundancy. As we see soon, this higher space usage is well used by ContaCT to provide much faster evaluation of some queries.

The last two rows of each dataset show the compression ratios computed with respect to the plain and binary representations, respectively. As explained, GraCT exploits the redundancy of trajectory data to obtain better compression. With $d = 720$, it uses 1.3–2.7 times less space than ContaCT and 75%–85% of the space needed by *p7zip*. The exception is `Ciconia`, which is not repetitive and makes GraCT use 50% more space than *p7zip*, and over 75% of the space of ContaCT.

Still, ContaCT obtains competitive compression ratios: the version with $d = 720$ uses 10%–25% of the space of a plain representation of the data, 20%–60% of the space of a binary representation, and about twice the space used by *p7zip* (which just compresses the data; it cannot solve any query without decompressing the whole dataset). To compare with another system that uses differential compression (and also does not support queries), we built Trajic (Nibali and He 2015), which used 177.91 MB on `Ships` (46% more than ContaCT), 242.61 MB on `Planes` (44% more than ContaCT), and 22.43 MB on `Ciconia` (5% less than ContaCT).[10]

6.3. *Query performance*

We now compare the response times of ContaCT and GraCT on the queries described in Section 5. As in the first experiment, we use different distances between snapshots, $d = 30$, 60, 120, 240, 360, and 720. The response times reported are the average of the user times spent by the algorithms to solve a set of queries of the same type. We consider eight queries:

- *ObjectPosition*: We compute a set of 20,000 queries for randomly chosen objects and time instants.
- *ObjectTrajectory*: We average 10,000 queries for randomly chosen objects and intervals. The difference between t_b and t_e is set to 2,000 time instants.
- *TimeSlice S*: We perform 1,000 queries for small random regions, of 40×40 cells, and a random time instant t_q.
- *TimeSlice L*: The same, but on large regions, of 320×320 cells.
- *TimeInterval S*: We perform 1,000 queries for a small random region of 40×40 cells during a random interval of 100 time instants.
- *TimeInterval L*: The same, but with a random region of 320×320 cells and a time interval of length 800.
- *MBR*: We perform 1,000 queries for randomly chosen objects along random time intervals of 200 instants.
- *Knn*: We average over 1,000 queries for random positions at random time instants. The value of K is chosen at random between 1 and 50. We also show experiments for fixed $K = 1$, $K = 10$, and $K = 100$.

Figures 8 to 11 show the average query time versus compression ratio (with respect

[10] It crashed when building on `Taxis`.

to the binary representation) for these queries. For each structure we show a line with six points, corresponding to the different values of parameter d (30, 60, 120, 240, 360, and 720; the higher d, the lower the space usage). ContaCT refers to the variant that uses plain bitmaps for D_p and D_n, and ContaCT-S to the one that uses sparse bitmaps. ContaCT with plain bitmaps is excluded from `Ciconia` because it uses more than 400% space.

6.3.1. Object position

Figure 8(a) shows the space-time tradeoffs for *ObjectPosition*. ContaCT answers this query in around 200–600 nanoseconds, whereas GraCT needs a few microseconds, being an order of magnitude slower.

Such a difference is expected. To solve this query, GraCT must obtain the absolute position of the object at the closest snapshot, and then traverse the grammar-compressed log while accumulating differences. When it reaches the nonterminal containing t_q, it must enter the grammar tree until reaching the desired leaf. On a log of d time instants, this takes at least $O(\log d)$ time. ContaCT, instead, directly computes the position of the object in $O(1)$ time, without even having to consult a snapshot or traversing a trajectory.

The dependence of GraCT on the value of d is also clear in the figure: as d increases the response times decline because the portion of log to traverse is larger, up to $\frac{d}{2}$. The time of ContaCT, instead, remains constant as d changes.

6.3.2. Object trajectory

Figure 8(b) shows that ContaCT obtains about the same performance as GraCT on this query. The reason is that, once reaching the first position of the object (where ContaCT is much faster), both GraCT and ContaCT take $O(1)$ amortized time per extracted position.

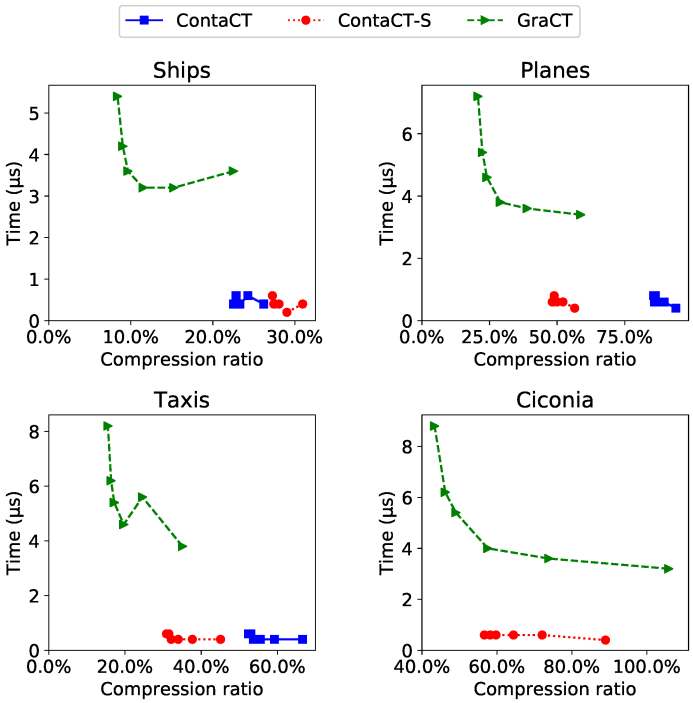
The reason why the times of GraCT improve as d increases is that every new snapshot reached along the trajectory must be accessed to find the new absolute position of the object. ContaCT, instead, is completely independent of the snapshots.

6.3.3. Time slice

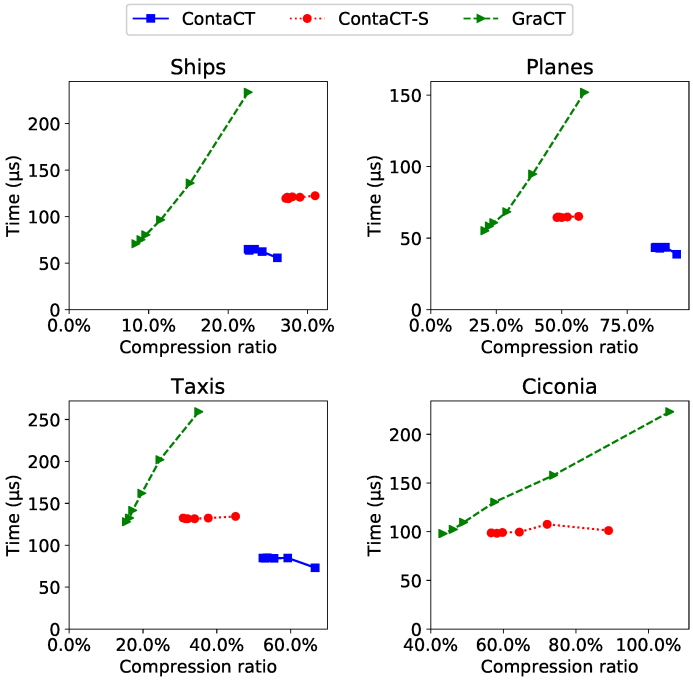
ContaCT does not obtain significant improvements on time slice queries. As shown in Figure 9, ContaCT is much faster when using the least-space configuration, but GraCT obtains better performance when using the same space of ContaCT. The exception is `Ciconia`, where ContaCT stays significantly faster even when GraCT gets to use the same space.

Since this query consists of traversing the closest snapshot and verifying all the candidates found with *ObjectPosition*, one would expect a large difference in performance as we obtained for that query. This is not the case, however, because GraCT can do better, without necessarily computing the position of each object at time t_q. After processing each nonterminal of the log in the way to t_q, GraCT computes the extended region with respect to the current object position, in order to determine if the object still has chances of being within the queried region at t_q. If not, the object is discarded immediately. The difference with respect to the constant-time computation of the object position at t_q made by ContaCT is then not as large as for the *ObjectPosition* query.

Figure 9 shows that the time performance worsens on both structures as d increases.

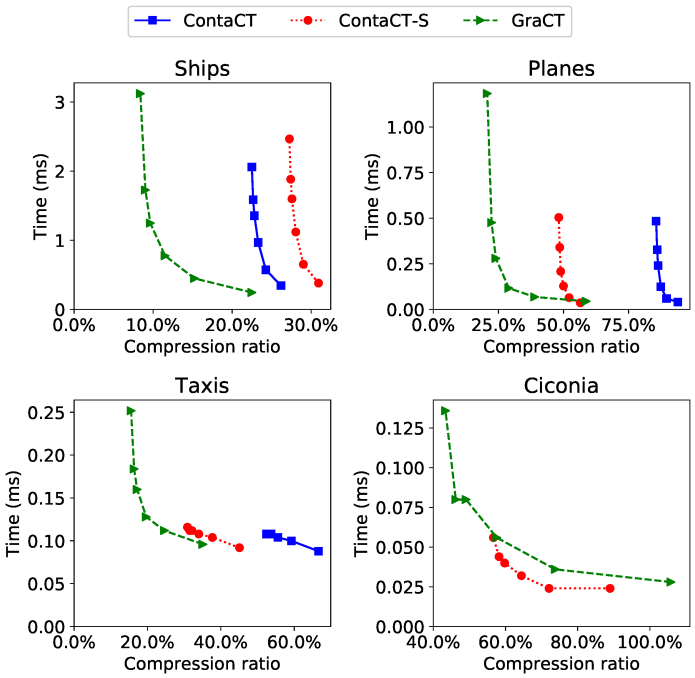


(a) *ObjectPosition*

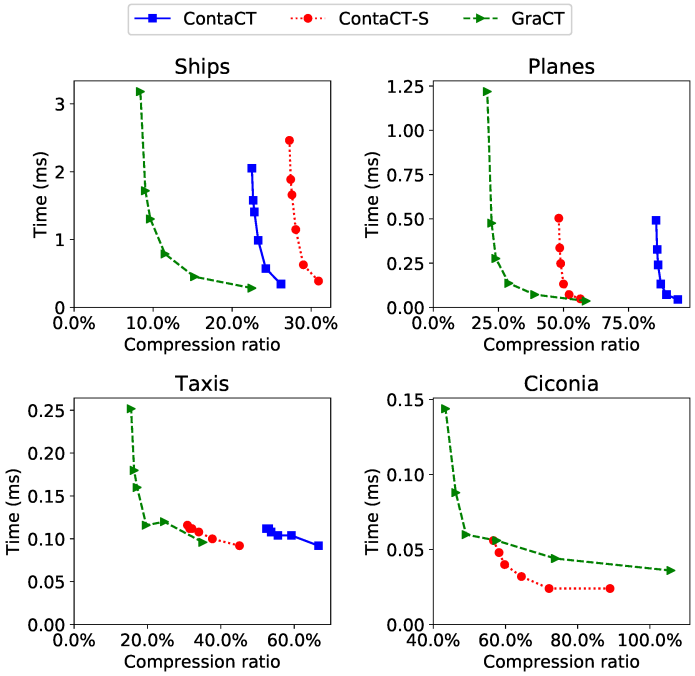


(b) *ObjectTrajectory*

Figure 8. Space and time for *ObjectPosition* and *ObjectTrajectory* queries, in microseconds per query.



(a) *TimeSlice S*



(b) *TimeSlice L*

Figure 9. Space and time for *TimeSlice* queries, in milliseconds per query.

The main reason of that slope of the times in ContaCT is the number of candidates. As d increases, the expanded region that contains the candidates increases proportionally along each direction, more precisely depending on the distance between the snapshot and the queried time instant. Besides, in GraCT, computing *ObjectPosition* requires reading up to $\frac{d}{2}$ entries from the log.

6.3.4. Time interval

Figure 10 shows the average times of time interval queries, using $\lambda = 20$ for ContaCT. We observe that ContaCT is faster than GraCT in all cases, except on `Planes` and `Ships`. In the first dataset, when the space is similar for both structures, GraCT is 1.3–1.6 times faster than ContaCT. Although in `Ships`, when querying large regions and intervals, ContaCT clearly outperforms GraCT, in `TimeInterval S` GraCT is as fast as ContaCT and still uses 80% of its space. In the other cases, comparing the fastest configuration of both structures, ContaCT is 1.7–3.4 times faster in `Taxis`, and 1.2–2.3 in `Ciconia`.

The reason why ContaCT outperforms GraCT on this query more sharply than for *TimeSlice*, especially on large intervals, is twofold. On the one hand, it is less likely that GraCT can preempt the traversal of the log by determining that the object is not in the spatial region at any time in $[t_b, t_e]$. On the other hand, once GraCT arrives at the region, it is also harder to verify the whole interval, and ContaCT is better at this.

A surprising effect in ContaCT is that, in some cases, time worsens when the space increases (i.e., d decreases). The reason is that we minimize the size of the spatial windows where we collect candidates, by using all the intermediate snapshots. This aims to minimize the number of candidate objects to verify, at the cost of querying more snapshots. When there are few candidate objects, it is better to just test them instead of working too much on the snapshots trying to discard them. In those cases, we can ignore some intermediate snapshots and increase the window size, so that the curves are nonincreasing with the space. This is an optimization parameter that a sophisticated deployment can introduce.

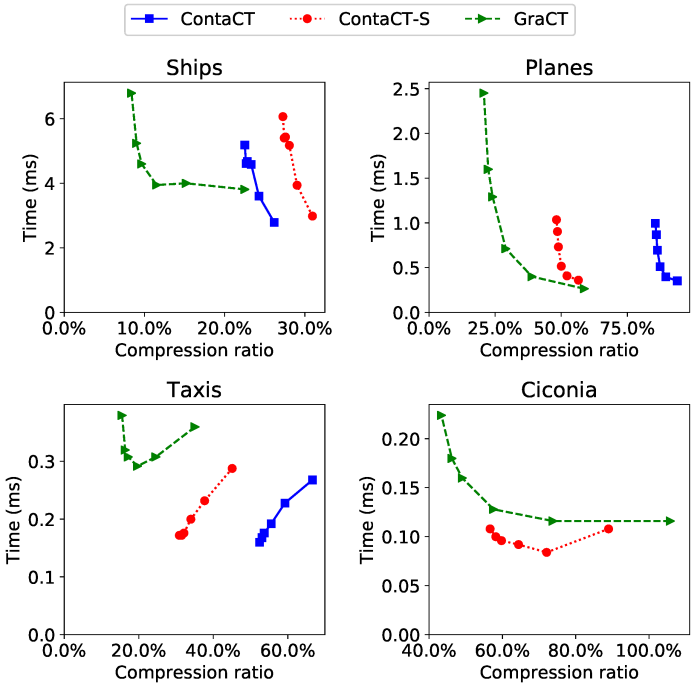
Both indexes use a hierarchical verification based on the MBRs. However, since ContaCT can compute any MBR on the fly, it can apply a perfect binary search on the whole interval $[t_b, t_e]$. GraCT, instead, must follow the partitioning given by the grammar, where each nonterminal stores its MBR. It may require traversing several nonterminals to cover the queried interval, and even several snapshots on large intervals.

The reason why the time of GraCT is not always decreasing as it uses more snaphots is that, although the extended regions where we find candidates decrease in size, it has to run the spatial query on more snapshots.

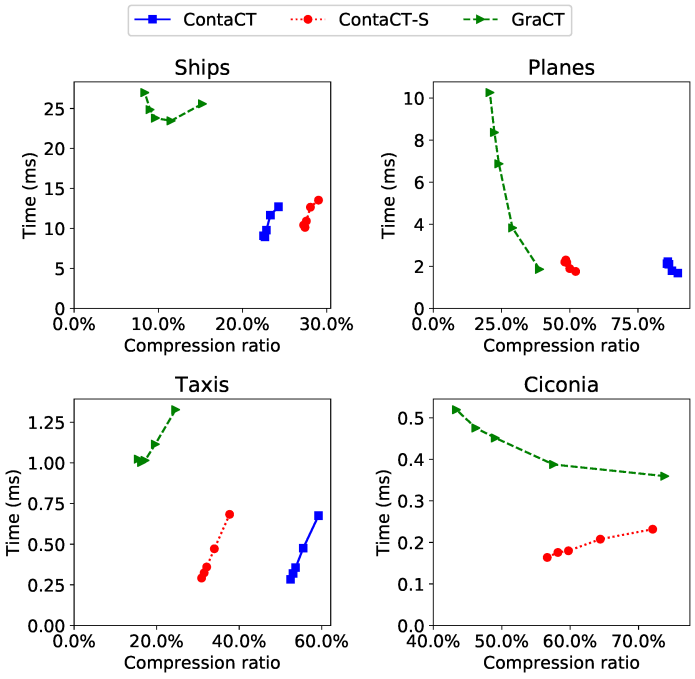
6.3.5. Minimum bounding rectangle

This is a new query enabled by ContaCT in $O(1)$ time, which has independent interest. In GraCT, computing the MBR requires traversing the portion of the log involved in the queried time interval and taking the union of the MBRs of the maximal nonintervals that cover $[t_b, t_e]$, which costs at least $O(\log d)$.

Figure 11(a) confirms that the difference in time is very significant, as expected: ContaCT solves the query in a few microseconds, outperforming GraCT by a factor of 4.5–8.8 in all datasets. As in *ObjectTrajectory*, the performance of GraCT worsens

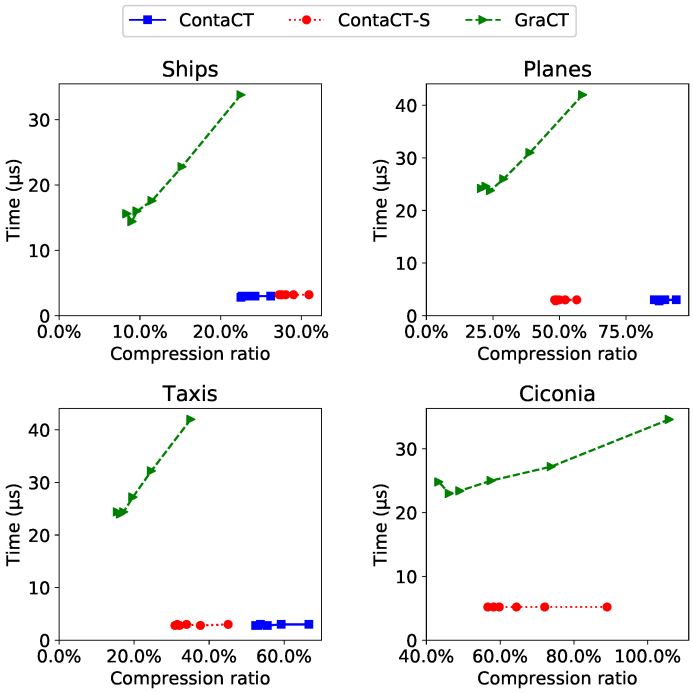


(a) *TimeInterval S*

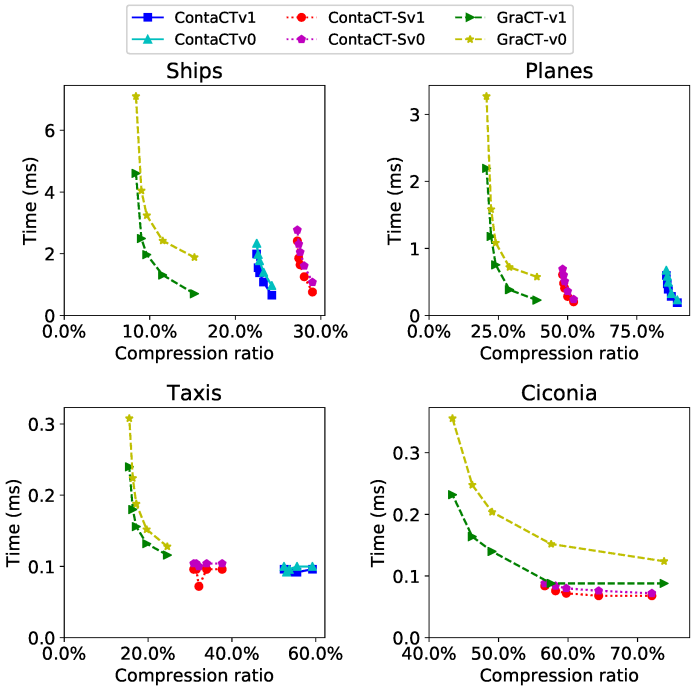


(b) *TimeInterval L*

Figure 10. Space and time for *TimeInterval* queries, in milliseconds per query.

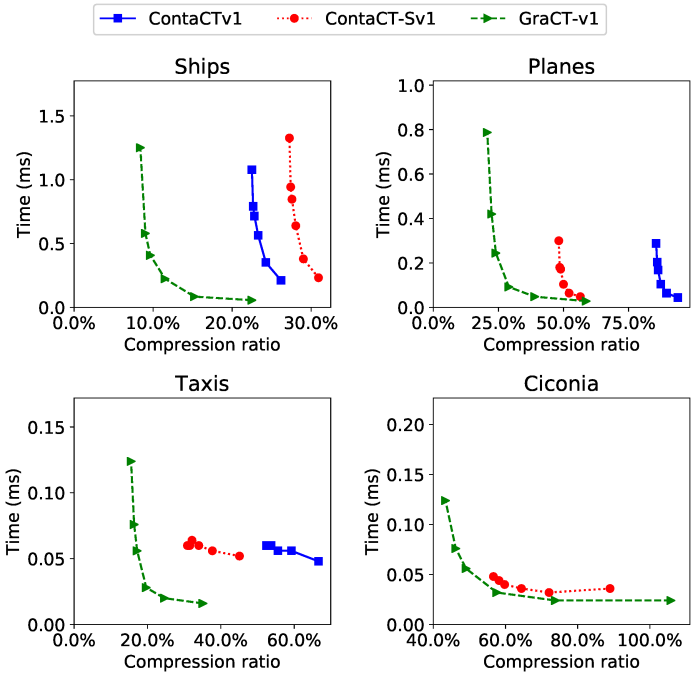


(a) *MBR*

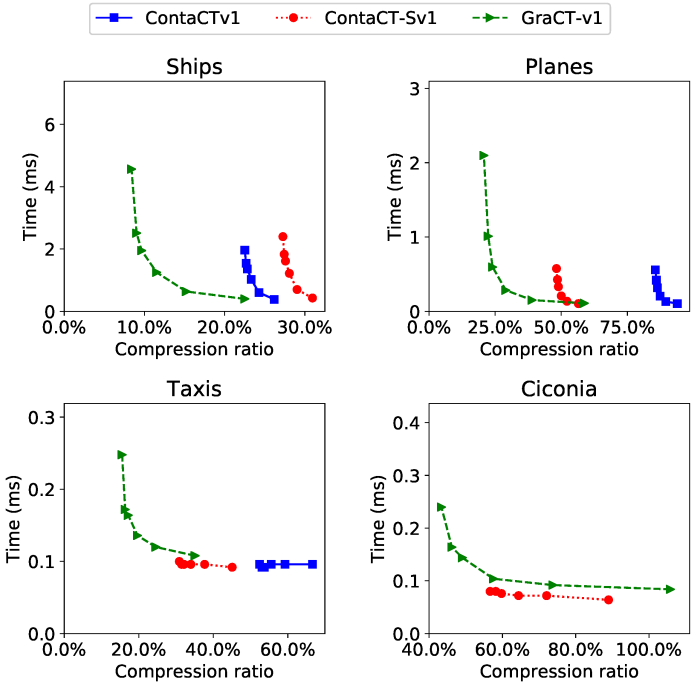


(b) *Knn*

Figure 11. Response time for *MBR* and *Knn* queries.



(a) *Knn with K=1*



(b) *Knn with K=10*

Figure 12. Response time for Knn with different values of K, in milliseconds per query.

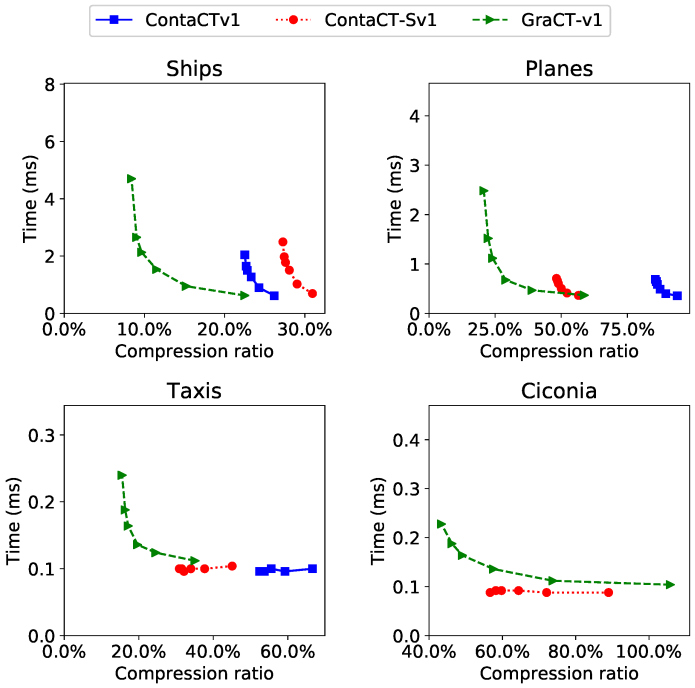


Figure 13. Response time for Knn with K=100.

when d is smaller, because of the costs of obtaining the absolute position of the object in each snapshot covering $[t_b, t_e]$.

6.3.6. Nearest neighbor

In order to study this query, we implemented two variants of the query: (v_0) the original algorithm of GraCT (Brisaboa *et al.* 2019) and (v_1), the one presented in Section 5.

In our experiments, we run GraCT with its original setup (v_0) and with the algorithm we present in this paper (v_1). We also present ContaCT with our own algorithm (v_1). In addition, we present a variant of v_0 adapted to ContaCT: We traverse the snapshot and insert all the objects found in Q_c (which now stores objects), plus those in *app/dis*, with their current distance to t_q. We then extract consecutive objects from Q_c and directly evaluate their position at t_q (in ContaCT, it makes no sense to advance progressively in the log as we do in v_0 of GraCT). The queue Q_r maintains the best candidates found at time t_q. We stop extracting candidates from Q_c when their known position, even if they move towards the query at maximum speed, cannot outperform the Kth closest candidate we have in Q_r.

As shown in Figure 11(b), version v_1, which we introduce in this paper, always outperforms v_0, more noticeably in GraCT. In general, GraCT obtains the same time performance of ContaCT with its maximum-space configuration, where GraCT uses 70%-80% of the space of ContaCT. In `Ciconia`, however, ContaCT is always slightly faster. Both versions sharply worsen their performance when d increases. During the process of adding the candidates to Q_c, the algorithm extends the regions represented by each traversed node by an extension that increases linearly with d along each direction. Thus, the larger d, the higher the uncertainty about where the objects of each node can reach, and thus more candidates have to be considered. GraCT depends even more sharply on d because the position of each candidate has to be computed,

which requires scanning, at most, $\frac{d}{2}$ entries from the log.

Figures 12 and 13 show the times for *Knn* queries with different values of K, using only the variant v_1. As it can be seen, ContaCT becomes better than GraCT as we look for more objects. This is expected, because ContaCT is better than GraCT at computing the position of a candidate object, and those become more numerous as K grows.

6.4. *Precision considerations*

In the preceding experiments, we chose the size of the grid to match that of the objects to track, as explained. This choice may vary depending on the application needs, however. In this section we study how the precision of the raster model affects the size and query performance of ContaCT and ContaCT-S. We built both structures fixing the distance between snapshots to the minimum-space configuration, $d = 720$, and using different cell sizes: 1×1, 10×10, 100×100, $1{,}000 \times 1{,}000$, and $10{,}000 \times 10{,}000$ meters (recall that our defaults are 10×10 for `Ships`, 100×100 for `Planes`, and 1×1 for `Taxis` and `Ciconia`).

Figure 14 shows how the space requirements of ContaCT and ContaCT-S evolve as the sizes of the cells grow. In all datasets, ContaCT is highly dependent on the cell size. Indeed, the quotient between the size of the maximum and minimum space configuration on `Ships` is 13, on `Planes` is 250, on `Taxis` is 8, and on `Ciconia` is 38. That quotient is much smaller on ContaCT-S, being around 1.8–3 in all datasets. The main cause of that difference between both structures is the implementation of the log. When the cells are smaller, the distances in cells between consecutive positions are larger, and thus the bitmaps of the log are longer and sparser. The space of ContaCT grows proportionally to the bitmap lengths. Instead, since ContaCT-S exploits the sparsity of bitmaps, it handles small cells better. As the cell size grows, the bitmaps become shorter and denser, and trying to exploit sparsity is even counterproductive.

The figure also includes the space obtained by Trajic (which does not support queries), except on `Taxis`, where it could not be built, as explained. The space of Trajic is similar to that of Contact-S, except on `Ciconia` with large cells. At this granularity, the objects barely move, which Trajic compresses very well. ContaCT-S, instead, is not optimized to handle many consecutive time instants without movement. We verified that the peak of Trajic for the largest cells in `Ships` is correct.

The performance of constant-time queries, like *ObjectPosition* and *MBR*, varies very little as we change the cell size, but the other queries generally speed up with larger cells. Figure 15 shows the response times of *TimeInterval* and *Knn* queries when varying the cell sizes. In *TimeInterval* queries, we compute the objects within a region of size 320×320 meters during an interval of 800 time instants. In *Knn* queries we set a random value of K between 1 and 50. Other queries yield analogous results.

Both ContaCT and ContaCT-S improve their performance as the size of cells grows, except for the two smallest cell sizes on `Planes`. The reason for the general improvement is that, with larger cells, the grids of the snapshots are smaller. The number of k^2-tree nodes traversed to collect all the candidates that fall in a region is proportional to the number of candidates recovered and to the perimeter of the region measured in number of cells (Navarro 2016, Sec. 10.2.1). The latter decreases proportionally to the cell size, which explains the roughly straight diagonals in most plots. The contrary effect, namely having more candidates to verify due to coarser cells in the snapshot, turns out to be not significant: between the structures with cells

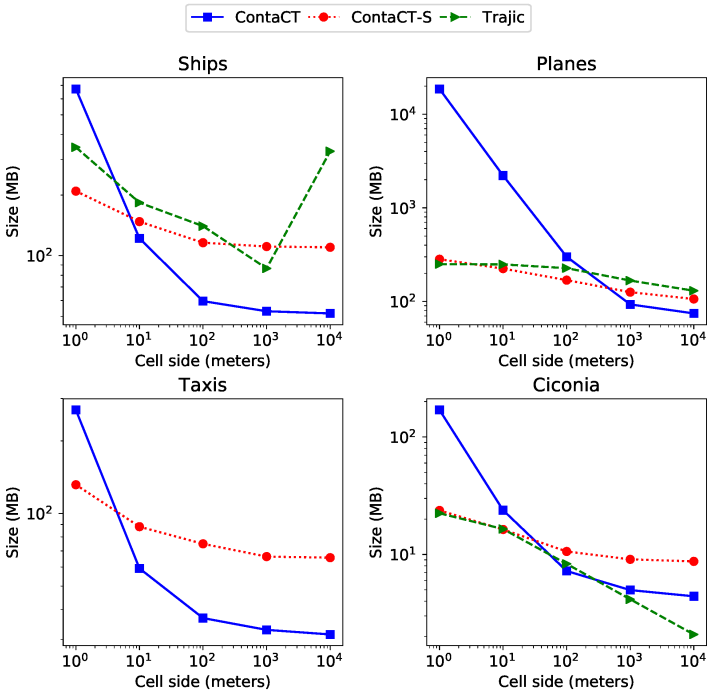


Figure 14. Evolution of the space of ContaCT, ContaCT-S, and Trajic with different cell sizes.

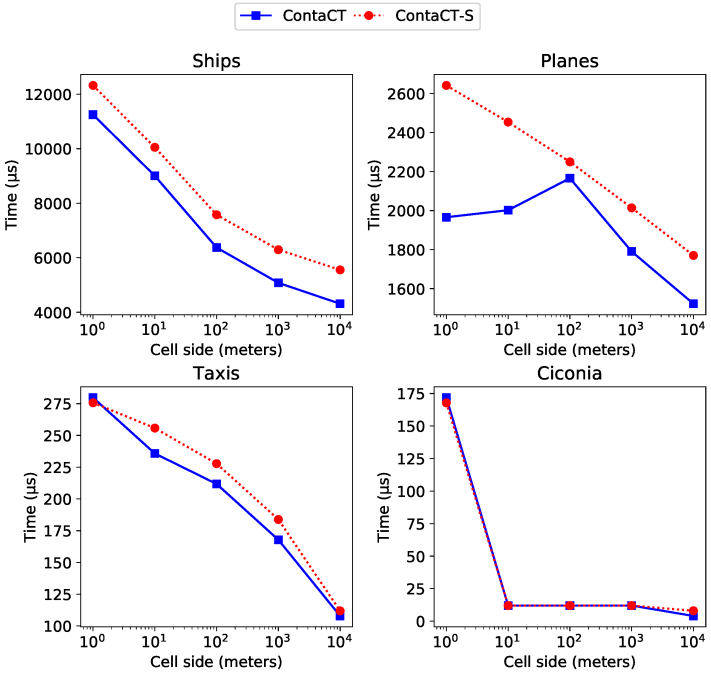
of 1×1 and $10,000 \times 10,000$ meters, the candidates grow by around 0.01% in all the datasets, except on `Ciconia`, where they grow by 2.5%.

Regarding the first two points of `Planes`, the reason lies in the implementation of operation *select* on the plain bitmaps used by ContaCT (`select_support_mcl` in the SDSL library). When the bitmaps are very sparse, ContaCT uses a lot of space (recall Figure 14), but in exchange it precomputes and stores all the *select* answers in the sparse regions. The probability of having the desired *select* answer directly stored then increases, and so decreases the average time to obtain the position of any object. These very sparse areas appear especially on `Planes` with cell sizes 1×1 and 10×10, due to the high speed of its objects. The performance of *select* on the remaining configurations varies by around $\pm 5\%$ only.

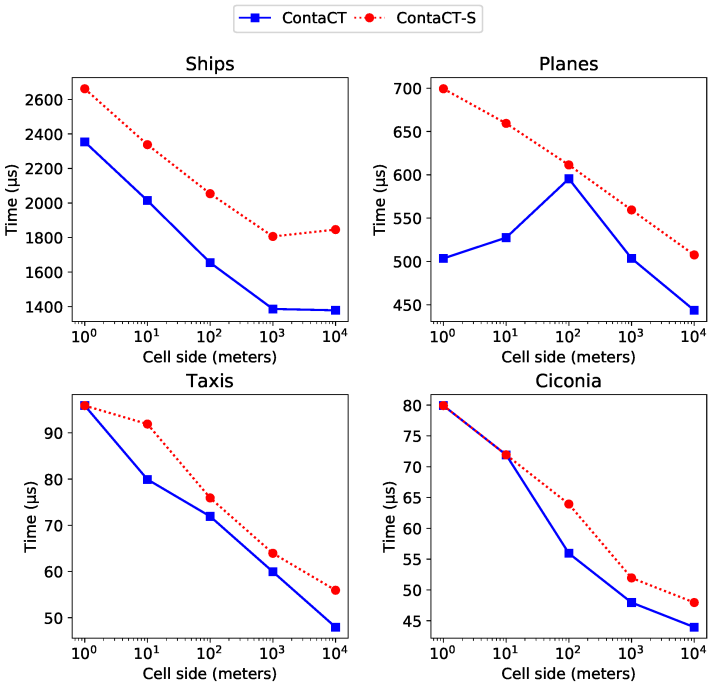
6.5. *Scalability*

We study the scalability of our structure in terms of compression ratios and query times by generating larger `Taxis` datasets from the same source, with sizes 5,120 MB, 10,240 MB, and 20,480 MB. The resulting four datasets then have approximately 1, 5, 10, and 20 GB of data (in plain form). While we expect ContaCT to retain its constant time in *ObjectPosition*, *ObjectTrajectory* (per item returned), and *MBR* queries, the time of *range* and *nearest neighbor* queries should grow linearly with the dataset size, because the number of candidates to verify grows linearly.

Figure 16(a) shows the evolution of the compression ratio, building the indexes with distance $d = 720$ between snapshots. We can observe that ContaCT (with sparse bitmaps) essentially maintains the same compression ratio as the dataset grows, only decreasing from 31% to 27%. Since GraCT exploits repetitiveness due to its grammar compression, and repetitiveness increases in our dataset as it grows, the compression

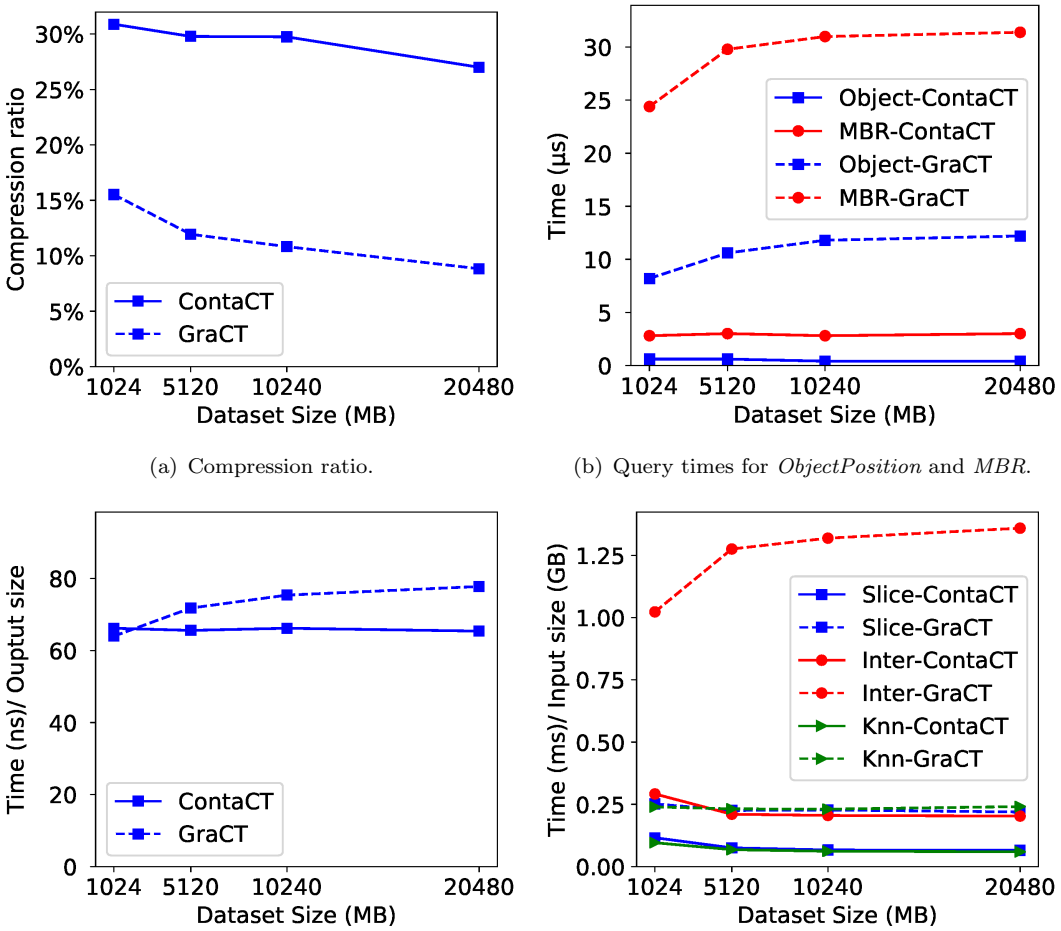


(a) *TimeInterval*



(b) *Knn*

Figure 15. Response time for *TimeInterval* and *Knn* queries with different cell sizes, in microseconds per query.



(a) Compression ratio.

(b) Query times for *ObjectPosition* and *MBR*.

(c) Query times of *ObjectTrajectory*.

(d) Other query times, over input size.

Figure 16. Evolution of compression ratio and query times as the dataset grows. Solid and dashed lines correspond to ContaCT and GraCT, respectively

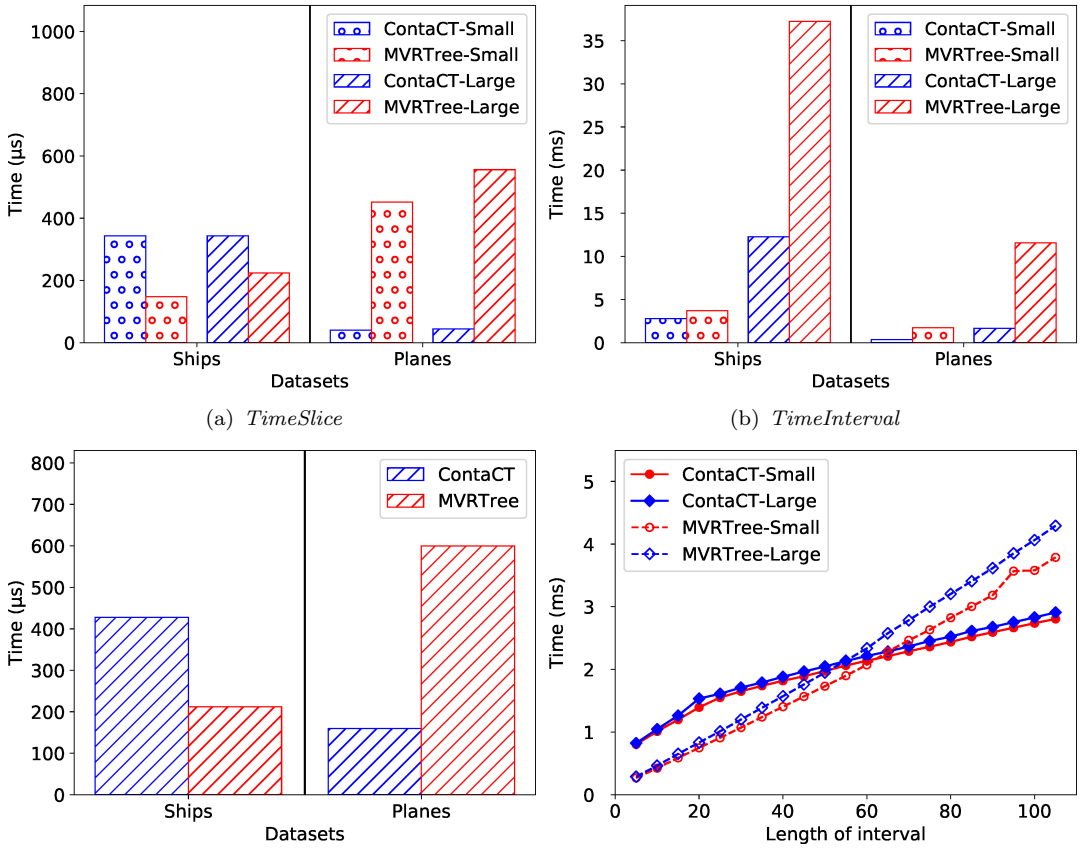
ratio of GraCT decreases significantly, from 15% to 9%.

Figures 16(b) and 16(c) show that, as expected, ContaCT retains constant time for the queries where it has $O(1)$ time complexity (per returned point, in the case of trajectories). Its times range from a few nanoseconds to a few microseconds, depending on the query. GraCT, instead, shows a logarithmic increase in query times, which is proportional to the height of the parse tree of its grammar. Indeed, the average heights of the nonterminals that form the logs, from the smallest to the largest dataset, are 2.21, 2.47, 2.61, and 2.76.

The rest of the queries are shown in Figure 16(d), with the query time divided by the dataset size in GB. As expected, the query times grow linearly with the data size in both indexes (GraCT grows superlinearly for large *TimeInterval* queries). In the case of ContaCT, all the times are below 0.2 milliseconds per GB.

6.6. *Comparison with a spatio-temporal index*

The MVR-tree (Tao and Papadias 2001b) is composed of several R-trees, each one called a version. Each version is associated with a different interval of time and



(a) *TimeSlice* (b) *TimeInterval*

(c) *Knn* (d) Growing *TimeInterval* queries on `Ships`

Figure 17. Query time comparison of ContaCT with the MVR-tree, running in main memory.

represents the positions of all the objects during that interval. Often, the differences between two versions are small, in fact, they can contain identical nodes. The MVR-tree exploits this feature by sharing common nodes between consecutive R-trees. This structure can efficiently solve *TimeSlice*, *TimeInterval*, and *Knn* queries: the algorithm traverses the versions involved in the queried time interval, following those nodes that intersect the spatial point or region of the query. On the other hand, solving *ObjectPosition* and *ObjectTrajectory* queries is costly.

In our experimental setup, we built MVR-trees on `Ships` and on `Planes`, with sizes 12.16 GB and 11.72 GB, respectively. In our experiments, both ContaCT and MVR-tree run entirely in main memory. ContaCT was configured with the same settings used in the previous experiments, fixing $d = 30$. Note that even this maximum-space configuration of ContaCT uses 88 times less space on `Ships`, and 61 times less space on `Planes`, than MVR-tree.

Figure 17 shows the times for the queries supported by the MVR-tree, comparing it with the maximum-space configuration of ContaCT. We observe that ContaCT is as fast as, and up to 6 times faster than, MVR-tree on most queries. The exception are the *TimeSlice* and *Knn* queries on `Ships`, where MVR-tree is up to 3 times faster. Those are the most efficient queries for the MVR-tree, because it needs to traverse one R-tree only.

TimeInterval queries, instead, involve an interval of time, and thus MVR-tree needs to traverse multiple versions. These are the queries where ContaCT outperforms MVR-

tree more sharply, see Figure 17(b). In order to study the turning point of *TimeInterval* queries on `Ships`, we run queries varying the span of their time interval. As the span increases, both structures slow down, but the times of MVR-tree increase much faster. Indeed, ContaCT outperforms MVR-tree when the length of the time interval surpasses 65 time instants. Therefore, ContaCT is not as dependent on the length of the time interval as MVR-tree, outperforming it on long intervals.

7. Conclusions

We have introduced a new compact data structure, called ContaCT, that indexes compressed trajectories of freely-moving objects. Although it can be regarded as a differential compressor and each trajectory is encoded separately as a sequence of consecutive positions, it is novel in that the differences between those positions are stored such that in *constant time* we can find the position of a given object at a given time, and the minimum bounding rectangle of a given object's trajectory for a given time interval. In combination with some auxiliary compact data structures, we use these constant-time queries to speed up other queries, such as finding the objects in a given spatial range at a given time instant or interval, or the closest objects to a given point at a given time instant.

Our experiments show that ContaCT compresses datasets to 20%–60% of the space of their uncompressed representations, which is about twice the space obtained by compressors like `p7zip`. While previous systems like GraCT (Brisaboa *et al.* 2019) use 1.3–2.7 times less space by exploiting similarities in the trajectories, ContaCT is significantly faster: it is 4–14 times faster than GraCT at finding objects' positions (taking a few nanoseconds), and 4–9 times faster for computing minimum bounding rectangles (taking a few microseconds). An effect of solving the minimum bounding rectangles so quickly is that ContaCT is up to 3.5 times faster than GraCT in finding the objects that are inside a spatial window within a time interval (taking a few tenths of milliseconds per GB of data). In the other queries, ContaCT also outperforms GraCT, but the latter can obtain similar time by adjusting its space/time trade-off while staying smaller than ContaCT. We note, however, that the compression results of ContaCT are more robust because they rely on the only fact that consecutive movements in a trajectory are relatively small, whereas GraCT requires that trajectories of different objects are similar. This difference shows up in a database of bird trajectories, where the space usage of both indexes are closer and ContaCT offers much better space/time trade-offs.

Compared to a classical spatio-temporal index (MVR-tree (Tao and Papadias 2001b)), ContaCT uses two orders of magnitude less space and it is competitive in query times, being even up to 6 times faster in some cases. For instance, in queries that find the objects within a region during a time interval, ContaCT is 3 times slower in some datasets when the queried time is just one time instant, but it becomes faster as soon as the time span surpasses 65 time instants.

For simplicity, we have assumed that all the trajectories are sampled at the same regular time instants. In practice, some objects may naturally emit more samples than others, and one may like to sample them differently. Since ContaCT stores each trajectory independently, it can easily use different time sampling rates for each. Solving most queries requires some assumptions, however, like a linear interpolation of the positions at query times t_q, and the time instants t at which the snapshots $\mathcal{S}_t$ are built. Note that GraCT would not perform well in this scenario, because its

compression relies on global similitude between different trajectories.

Future work involves using ContaCT to handle more complex queries, particularly more sophisticated variants of nearest neighbor queries. Two such queries directly extend our basic nearest neighbor query. The first one transforms the queried time instant into an interval of time, and considers the closest distance of an object to a point along the time interval (Gao *et al.* 2007). The second one considers the nearest neighbors of a trajectory (Tang *et al.* 2011), that is, computing the trajectories most similar to a given one according to a trajectory distance measure (Su *et al.* 2020). Besides, there are other queries focused on data mining, like moving-together patterns (Gudmundsson *et al.* 2004), which detects objects that travel together; or trajectory clustering (Lee *et al.* 2007), which looks for the most common patterns of movement. In all these queries, the ability of ContaCT for efficiently computing the trajectory-summarization primitive MBR will be key for detecting similarity and proximity between trajectories.

Data and Codes Availability Statement

Data and codes that support the findings of this study are available with the identifiers at the private link `https://figshare.com/s/6780c0aea935c8e9b10e`.

Acknowledgements

We thank the reviewers for their valuable comments, which helped improve our presentation considerably.

Funding

This work was supported by Xunta de Galicia/FEDER-UE under Grants [IN848D-2017-2350417; IN852A 2018/14; ED431C 2017/58]; Xunta de Galicia and European Union (European Regional Development Fund- Galicia 2014-2020 Program) with the support of CITIC research center under Grant [ED431G 2019/01]; Ministerio de Ciencia, Innovación y Universidades under Grants [TIN2016-78011-C4-1-R; RTC-2017-5908-7]; G.N. was supported by ANID - Millennium Science Initiative Program under Grant [ICN17_002]; and Fondecyt under Grant [1-200038]. T.G. was supported by NSERC under grant [RGPIN-2020-07185].

References

Becker, M., *et al.*, 2015. Viztrails: An information visualization tool for exploring geographic movement trajectories. *In*: *Proc. 26th ACM Conference on Hypertext & Social Media*. 319–320.

Bell, T.C., Cleary, J., and Witten, I.H., 1990. *Text compression*. Prentice Hall.

Botea, V., *et al.*, 2008. Pist: An efficient and practical indexing technique for historical spatio-temporal point data. *GeoInformatica*, 12 (2), 143–168.

Brisaboa, N., Ladra, S., and Navarro, G., 2013. DACs: Bringing direct access to variable-length codes. *Information Processing and Management*, 49 (1), 392–404.

Brisaboa, N.R., Ladra, S., and Navarro, G., 2014. Compact representation of web graphs with extended functionality. *Information Systems*, 39 (1), 152–174.

Brisaboa, N.R., *et al.*, 2019. GraCT: A grammar-based compressed index for trajectory data. *Information Sciences*, 483, 106 – 135.

Bustos, B. and Navarro, G., 2009. Improving the space cost of k-nn search in metric spaces by using distance estimators. *Multimedia Tools and Applications*, 41 (2), 215–233.

Cao, H., Wolfson, O., and Trajcevski, G., 2006. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal*, 15 (3), 211–228.

Cao, H., Mamoulis, N., and Cheung, D.W., 2005. Mining frequent spatio-temporal sequential patterns. *In*: *Proc. 5th IEEE International Conference on Data Mining (ICDM)*. 82–89.

Chakka, V.P., Everspaugh, A., and Patel, J.M., 2003. Indexing large trajectory data sets with SETI. *In*: *Proc. Conference on Innovative Data Systems Research (CIDR)*.

Cheng, Y., *et al.*, 2019. "Closer-to-home" strategy benefits juvenile survival in a long-distance migratory bird. *Ecology and evolution*, 9 (16), 8945–8952.

Cudre-Mauroux, P., Wu, E., and Madden, S., 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 Peuker, T.K., 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.

Ferrada, H. and Navarro, G., 2017. Improved range minimum queries. *Journal of Discrete Algorithms*, 43, 72–80.

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

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

Gagie, T., *et al.*, 2017. Document retrieval on repetitive collections. *Information Retrieval*, 20, 253–291.

Gao, Y.J., *et al.*, 2007. Efficient k-nearest-neighbor search algorithms for historical moving object trajectories. *Journal of Computer Science and Technology*, 22 (2), 232–244.

Gog, S., *et al.*, 2014. From theory to practice: Plug and play with succinct data structures. *In*: *Proc. 13th International Symposium on Experimental Algorithms (SEA)*. 326–337.

Gudmundsson, J., Laube, P., and Wolle, T., 2008. Movement patterns in spatio-temporal data. *Encyclopedia of GIS*, 726, 732.

Gudmundsson, J. and van Kreveld, M., 2006. Computing longest duration flocks in trajectory data. *In*: *Proc. 14th Annual ACM International Symposium on Advances in Geographic Information Systems (GIS)*. 35–42.

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

Gutiérrez, G., *et al.*, 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.

Guttman, A., 1984. R-trees: A dynamic index structure for spatial searching. *In*: *Proc. ACM International Conference on Management of Data (SIGMOD)*. 47–57.

Huang, S., *et al.*, 2014. R-HBase: A multi-dimensional indexing framework for cloud computing environment. *In*: *Proc. Workshops of the IEEE International Conference on Data Mining (ICDM)*. 569–574.

Hughes, J.N., *et al.*, 2015. Geomesa: A distributed architecture for spatio-temporal fusion. *In*: *Proc. SPIE*. vol. 9473.

Lee, J.G., Han, J., and Whang, K.Y., 2007. Trajectory clustering: a partition-and-group framework. *In*: *Proc. ACM International Conference on Management of Data (SIGMOD)*. 593–604.

Leontiadis, I., *et al.*, 2011. On the effectiveness of an opportunistic traffic management system for vehicular networks. *IEEE Transactions on Intelligent Transportation Systems*, 12 (4), 1537–1548.

Li, Z., *et al.*, 2010a. Swarm: Mining relaxed temporal moving object clusters. *Proceedings of the VLDB Endowment*, 3 (1-2), 723–734.

Li, Z., *et al.*, 2011. Movemine: Mining moving object data for discovery of animal movement patterns. *ACM Transactions on Intelligent Systems and Technology*, 2 (4), 1–32.

Li, Z., *et al.*, 2010b. Incremental clustering for trajectories. *In*: *Proc. International Conference on Database Systems for Advanced Applications*. 32–46.

Lin, X., *et al.*, 2017. One-pass error bounded trajectory simplification. *Proceedings of the VLDB Endowment*, 10 (7), 841–852.

Liu, J., *et al.*, 2015. Bounded quadrant system: Error-bounded trajectory compression on the go. *In*: *Proc. 31st IEEE International Conference on Data Engineering (ICDE)*. 987–998.

Ma, Q., *et al.*, 2009. Query processing of massive trajectory data based on MapReduce. *In*: *Proc. 1st International Workshop on Cloud Data Management (CloudDB)*. 9–16.

Ma, S., Zheng, Y., and Wolfson, O., 2013. T-share: A large-scale dynamic taxi ridesharing service. *In*: *Proc. 29th IEEE International Conference on Data Engineering (ICDE)*. 410–421.

Mahmood, A.R., Punni, S., and Aref, W.G., 2019. Spatio-temporal access methods: a survey (2010-2017). *GeoInformatica*, 23 (1), 1–36.

Meratnia, N. and de By, R.A., 2004. Spatiotemporal compression techniques for moving point objects. *In*: *Proc. 9th International Conference on Extending Database Technology (EDBT)*. 765–782.

Muckell, J., *et al.*, 2011. SQUISH: an online approach for gps trajectory compression. *In*: *Proc. 2nd International Conference on Computing for Geospatial Research & Applications*. 1–8.

Muckell, J., *et al.*, 2014. Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica*, 18 (3), 435–460.

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

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

Navarro, G., 2016. *Compact Data Structures – A practical approach*. Cambridge University Press.

Ni, J. and Ravishankar, C.V., 2007. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Transactions on Knowledge and Data Engineering*, 19 (5), 663–678.

Nibali, A. and He, Z., 2015. Trajic: An effective compression system for trajectory data. *IEEE Transactions on Knowledge and Data Engineering*, 27 (11), 3138–3151.

Nishimura, S., *et al.*, 2013. MD-HBase: design and implementation of an elastic data

infrastructure for cloud-scale location services. *Distributed and Parallel Databases*, 31 (2), 289–319.

Okanohara, D. and Sadakane, K., 2007. Practical entropy-compressed rank/select dictionary. *In*: *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*. 60–70.

Pfoser, D., Jensen, C.S., and Theodoridis, Y., 2000. Novel approaches to the indexing of moving object trajectories. *In*: *Proc. 26th International Conference on Very Large Data Bases (VLDB)*. 395–406.

Popa, I.S., *et al.*, 2015. Spatio-temporal compression of trajectories in road networks. *GeoInformatica*, 19 (1), 117–145.

Potamias, M., Patroumpas, K., and Sellis, T., 2006. Sampling trajectory streams with spatiotemporal criteria. *In*: *Proc. 18th International Conference on Scientific and Statistical Database Management (SSDBM)*. 275–284.

Samet, H., 1984. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16, 187–260.

Schmid, F., Richter, K.F., and Laube, P., 2009. Semantic trajectory compression. *In*: *Proc. 11th International Symposium on Spatial and Temporal Databases (SSTD)*. 411–416.

Su, H., *et al.*, 2020. A survey of trajectory distance measures and performance evaluation. *The VLDB Journal*, 29 (1), 3–32.

Ta, N., *et al.*, 2016. Semantic-aware trajectory compression with urban road network. *In*: *Proc. International Conference on Web-Age Information Management*. Springer, 124–136.

Tan, H., Luo, W., and Ni, L.M., 2012. CloST: A Hadoop-based storage system for big spatio-temporal data analytics. *In*: *Proc. 21st ACM International Conference on Information and Knowledge Management (CIKM)*. 2139–2143.

Tang, L.A., *et al.*, 2011. Retrieving k-nearest neighboring trajectories by a set of point locations. *In*: *Proc. International Symposium on Spatial and Temporal Databases*. 223–241.

Tao, Y. and Papadias, D., 2001a. Efficient historical R-trees. *In*: *Proc. International Conference on Scientific and Statistical Database Management (SSDBM)*. 223–232.

Tao, Y. and Papadias, D., 2001b. 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, G., *et al.*, 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, M., Theodoridis, Y., and Sellis, T.K., 1998. Spatio-temporal composition and indexing for large multimedia applications. *ACM Multimedia Systems Journal*, 6 (4), 284–298.

Wang, L., *et al.*, 2008. A flexible spatio-temporal indexing scheme for large-scale GPS track retrieval. *In*: *Proc. International Conference on Mobile Data Management (MDM)*. 1–8.

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

Xiao, X., *et al.*, 2010. Finding similar users using category-based location history. *In*: *Proc. 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems*. 442–445.

Xu, X., Han, J., and Lu, W., 1990. RT-tree: An improved R-tree index structure for spatiotemporal databases. *In*: *Proc. 4th International Symposium on Spatial Data*

Handling. vol. 2, 1040–1049.

Yang, S., He, Z., and Chen, Y.P.P., 2018. GCOTraj: A storage approach for historical trajectory data sets using grid cells ordering. *Information Sciences*, 459, 1–19.

Ye, Y., *et al.*, 2009. Mining individual life pattern based on location history. *In*: *Proc. 10th International Conference on Mobile Data Management: Systems, Services and Middleware*. 1–10.

Zaharia, M., *et al.*, 2016. Apache Spark: A unified engine for big data processing. *Communications of the ACM*, 59 (11), 56–65.

Zhang, Z., *et al.*, 2017. TrajSpark: A scalable and efficient in-memory management system for big trajectory data. *In*: *Proc. 1st International Joint Conference APWeb-WAIM, Part I*. 11–26.

Zhao, Y., *et al.*, 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, B., *et al.*, 2018. Sharkdb: an in-memory column-oriented storage for trajectory analysis. *World Wide Web*, 21 (2), 455–485.

Zheng, Y., 2015. Trajectory data mining: an overview. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 6 (3), 1–41.

Zheng, Y. and Zhou, X., eds., 2011. *Computing with Spatial Trajectories*. Springer.

Zhou, P., *et al.*, 2005. Close pair queries in moving object databases. *In*: *Proc. 13th Annual ACM International Workshop on Geographic Information Systems (GIS)*. 2–11.