Information Sciences 566 (2021) 300–325

Contents lists available at ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Space-efficient representations of raster time series ☆

Fernando Silva-Coira [a,*], José R. Paramá [a], Guillermo de Bernardo [a], Diego Seco [b]

[a] Universidade da Coruña, Centro de Investigación CITIC, Facultade de Informática, Campus de Elviña s/n, 15071 A Coruña, Spain
[b] Universidad de Concepción & IMFD, Concepción, Chile

ARTICLE INFO

Article history:
Received 31 July 2020
Received in revised form 19 January 2021
Accepted 12 March 2021
Available online 18 March 2021

Keywords:
Geographic information systems
Raster datasets
Data compression
Indexing
Query processing
Compact data structures

ABSTRACT

Raster time series, a.k.a. temporal rasters, are collections of rasters covering the same region at consecutive timestamps. These data have been used in many different applications ranging from weather forecast systems to monitoring of forest degradation or soil contamination. Many different sensors are generating this type of data, which makes such analyses possible, but also challenges the technological capacity to store and retrieve the data. In this work, we propose a space-efficient representation of raster time series that is based on Compact Data Structures (CDS). Our method uses a strategy of snapshots and logs to represent the data, in which both components are represented using CDS. We study two variants of this strategy, one with regular sampling and another one based on a heuristic that determines at which timestamps should the snapshots be created to reduce the space redundancy. We perform a comprehensive experimental evaluation using real datasets. The results show that the proposed strategy is competitive in space with alternatives based on pure data compression, while providing much more efficient query times for different types of queries.

1. Introduction

The efficient processing and management of spatial data has been a topic of research and development for decades. This has been studied by the Geographic Information Systems (GIS) and spatial databases communities, which propose two main models to represent spatial data, namely vector and raster model. The former represents spatial objects using points and lines connecting such points, and forming more complex representations such as polygons. This model is mainly used to represent human-made features. On the other hand, the raster model represents the space as a tessellation of fixed size tiles, usually squares, each one storing a value. This model is traditionally used to represent real-world elements that were not made by humans. Some examples of spatial data usually represented in rasters are temperatures, precipitations, elevations, and chlorophyll concentrations, to name just a few examples. The Map Algebra [43] is considered a breakthrough in the use of this model[1] in GIS, as it provides a simple and powerful tool to perform geographic analysis. Map algebra was incorporated in many GIS and remote sensing image processing packages, for example, ArcGis of ESRI, QGIS, GDMS-R, or GRASS.

☆ A preliminary partial version was published in [10].

* Corresponding author.
E-mail addresses: fernando.silva@udc.es (F. Silva-Coira), jose.parama@udc.es (J.R. Paramá), guillermo.debernardo@udc.es (G. de Bernardo), dseco@udec.cl (D. Seco).

[1] Although the Map Algebra can be performed on vector data, it is usually performed on raster data.

https://doi.org/10.1016/j.ins.2021.03.035
0020-0255/

In many applications, not just a single raster, but a series of rasters covering the same region at different timestamps has to be stored and analysed. This has been called in the literature a raster time series or a temporal raster, and it has applications, for example, in weather forecast systems [13], prediction of productive fishing areas [4], monitoring forest degradation [18], monitoring soil contamination [15], agriculture [2], data mining on Satellite Image Time Series [34], or in Big Data analytics [3].

Although a raster time series can be conceptualised as an ordered collection of independent rasters, the definition of this new model has several advantages. Some of these advantages are: enabling general visualisation and analysis techniques independent of the application domain, facilitating consistency checking both in the spatial and temporal dimensions, or improving space requirements in data storage (as we show in this work, it is possible to store a raster time series in less space than the original collection of rasters).

Therefore, temporal evolution of raster data has been studied from different perspectives. For example, regarding the modelling of the information, in [27], the classical Map Algebra is extended to deal with temporal rasters. The conceptual solution extends the 2D matrix to a 3D cube, where each slice of the temporal dimension is the raster corresponding to one time instant. Each operation of Map Algebra obtains from one or two input rasters a new one. The operations of 2D Map Algebra are usually divided in three groups. *Local* operations obtain a raster where each output cell is computed as a function (i.e. sum, subtract, etc.) of the input cell/s at the same location in the input raster/s. *Focal* operations calculate the new value of each output cell as a function of the input cells in a neighboring area around its position. Finally, *Zonal* operations compute the value of each output cell as a function of the input cells within a specified zone of the first operand, such zones are determined by zones defined in the second operand. The extension to raster time series simply considers that a cell has, instead of only two Cartesian coordinates X,Y, the Cartesian coordinates plus a third dimension Z, which considers the time. Then, in *Local* operations, the cells used to compute the value of an output cell are those in the same exact position in the 3D space. In *Focal* and *Zonal* operations, the 2D zones used to compute the value of an output cell are now 3D zones. The 3D Map Algebra was later extended to a multidimensional map algebra (MMA) in [28].

Regarding data storage, new technologies to collect and generate geo-referenced data, such as ubiquitous mobile devices, Internet of Things sensors, new remote sensing devices, and so on, are creating huge spatio-temporal data sources that are difficult to store and analyse. For example, remote sensing images are acquired each day at a rate of several terabytes per day [26,47,48], and the archived amount of raster data of this type is slowly approaching the zettabyte scale [37].

Hence, a main challenge has been the volume of information, which is influenced by two characteristics of the temporal raster model: spatial and temporal resolution. Spatial resolution is defined by the tile size. The smaller the tile, the higher the precision of the model, but also the larger the space consumption. In raster time series, the time distance between two consecutive rasters defines the temporal resolution. The higher the temporal resolution, the higher the space consumption. For example, increasing the precision from days to hours requires 24 times more space. Since the new devices allow increasing spatial (e.g. sub-meter) and temporal (hourly) resolutions [5], then the space consumption problem is getting worse.

Data compression has been used to deal with this challenge and reduce storage space and transmission time of raster data [22,45]. Most real systems capable of managing raster data (including raster time series), like Rasdaman, Grass, or even R, as well as raster representation formats such as NetCDF (standard format of the OGC[2]) and GeoTiff, rely on traditional compression methods [38] such as run length encoding, Lempel–Ziv-Welch, or Deflate to reduce storage space. However, the use of these compression methods poses an important drawback to access a given datum or portion of the data, since the whole dataset, or a large portion, must be decompressed.

In contrast, a new family of compression methods, called compact data structures [31], are able to obtain a given datum or portion of the data decompressing only those data. Moreover, most compact data structures are also equipped with indexes within the same compressed space, and thus access times over the compressed data are comparable or better than the classical methods over uncompressed data. This is also due to a better usage of the memory hierarchy, since data can be kept in main memory in compressed from, and thus bigger portions of them can be maintained there. Several compact data structures haven been developed for rasters [6,24,35,23]. Among them, the k^2*raster* [23,24] is usually the most competitive data structure in the space–time trade-off. Although this approach does not reduce the space as much as pure compression methods, it supports several important operations without decompressing the data, such as the retrieval of a specific zone or filtering the tiles in a zone which values are restricted to a range. These operations provide the primitives to more complex analyses such as detecting zones with high flood risk.

In this work, we present a compact data structure for raster time series using a completely novel approach with respect to the classical methods. The usual way to store rasters and raster time series datasets is as simple N-dimensional arrays. In addition, when compression is required, this is achieved by means of classical compression methods, such as Deflate, which, as explained, require decompressing the entire dataset or, at least, in large parts. This is specially unfortunate for the typical operations on rasters, such as obtaining a given window or the *Zonal* operations of Map Algebra, which require to decompress only certain zones to response the query. In contrast, instead of using the classical N-dimensional array setup, our method uses the data arrangement of the k^2*raster*, which actually uses a quadtree-like data arrangement.

The first advantage of using the quadtree data structure is that we have a spatial index over the data. In addition, since our structure is based on the k^2*raster*, we use all its techniques specially adapted to that data arrangement, which are based on

[2] https://www.opengeospatial.org/standards/netcdf

state-of-the-art compact data structures methods. This brings even more benefits; now it is possible to decompress a given datum or a given region without decompressing anything else. Another advantage is that the spatial index is coupled with an index over the values stored at cells, so it is able to filter cells by content as well.

However, by simply using a k^2*raster* per time instant we do not take advantage of the temporal regularities that arise in many real-world raster datasets since, in that case, each time instant would be compressed isolated. Therefore, we use the strategy of snapshots and logs, which has been used in video and spatio-temporal indexes compression [42,7,14]. This allows us to replace repeated portions of the data by pointers to a previous appearance, thus obtaining compression.

We experimentally evaluate our proposal in multiple real raster time series, and compare it with state-of-the-art solutions. Our results confirm that our method is the fastest representation, or very close to it. Although a classical compression method for raster time series improves our compression, it is much slower in all queries, between 10 and 10,000 times slower. Other solutions based on compact data structures are not competitive with our proposal, which yields the best space–time trade-off across all datasets.

The rest of the article is organised as follows. First, we provide some background and basic concepts in Section 2, and revise most relevant related work in Section 3. Then, we formalise the problem and define the basic operations in Section 4. Our first solution, named $T-k^2$*raster*, is described in Section 5 and a variant of such method, named *htkdosr*, is described in Section 6. In Section 7 we present our experimental results. Finally, in Section 8, we present our conclusions and outline directions for future work.

2. Background

2.1. Storage methods for raster time series

A variety of formats and tools have been traditionally used for the storage of raster data. A single raster dataset (i.e. not a time series) can be easily stored as a matrix of values in generic matrix-processing tools, or using a number of raster-based image file formats. File formats such as GeoTIFF actually rely on classical image representations for the raster data, enhanced with metadata to represent geographical attributes of the image.

The extension of traditional storage methods to the representation of raster time series has followed the same two techniques used for single raster datasets: approaches based on generic representations of arrays or multi-dimensional matrices, as well as classical representations of image sequences (i.e. video file formats), can be easily used to store a sequence of raster images.

In this section we will describe NetCDF, a widely-used standard for the representation of multi-dimensional data, and we will also introduce video compression techniques and their potential applications to raster time series.

2.1.1. NetCDF

Network Common Data Form (NetCDF) [25] is a data format coupled with a set of software libraries that is able to represent different types of array-based data. Specifically, as an Open Geospatial Consortium Standard, it is widely used to represent and query raster data, and is supported by most GIS software tools that handle raster data.

Data are arranged in simple N-dimensional arrays. NetCDF can be configured to provide a compressed or uncompressed representation of the underlying data. NetCDF compression is based on Deflate [12], and provides a space–time tradeoff: ten compression levels are available, ranging from level 0 (no compression) to level 9 (maximum compression). When a compressed representation is used, the data can be transparently accessed without performing an explicit decompression, that is, the access procedure is the same whether the data are compressed or not. Additionally, NetCDF is designed to provide efficient access to the data even when compression is applied, using a technique called *chunking*: data is compressed in blocks, so when a specific region of the data has to be accessed only the relevant chunks of data need to be decompressed.

2.1.2. Video compression

If we consider a raster as an image, a raster time series can be assimilated to a sequence of evolving images. Therefore, video formats can be considered related works facing compression of temporal rasters. Video formats store a sequence of images, each of them called a *frame*, and try to exploit regularities in the images and similarity of consecutive images to achieve compression.

Moving Picture Experts Group (MPEG) [14] developed several standards for video representation, all of them compressed. They are based on three types of frames: *intra frames* or *I frames* are frames coded with a classical compression method for still images, *inter frames* or *P frames* are represented only encoding the differences between that frame and its predecessor in the video, and *B frames* are encoded based on past and future frames.

A video is compressed as a sequence of *groups of pictures*, each one starts with an *I* frame and the rest are *B* and *P* frames.

This basic idea of storing only some complete frames and encoding the rest by storing only the differences with others is somehow used by our method. However, MPEG is a lossy method, whereas we are interested in scientific and engineering data, where this type of compression is not feasible. In addition, MPEG standards are only designed for displaying purposes, whereas in our tackled data, we need the ability to query the data, therefore video compression is out of the scope of this work.

2.2. Compact data structures

Compact data structures provide space- and query-efficient representations of data. Unlike compression techniques, that can store data in little space but usually require a decompression of the data to perform queries, compact data structures aim at providing efficient query support over the compressed data, with no decompression being required. We refer to [31] for a survey of the advances in the area in the last decades. In this section we will describe compact data structures to provide *rank* and *select* operations on bitmaps, a well-studied problem that is a key building block for the related work described in Section 3 and our own solutions.

2.2.1. Rank and select on bitmaps

Consider a bitmap $B[0 \dots n-1]$, storing a sequence of n bits. We define the following useful operations:

- $rank_a(B, i)$ counts the occurrences of the bit a in $B[0 \dots i]$.
- $select_a(B, i)$ locates the position for the i^{th} occurrence of a in B.

A number of compact data structures have been proposed to efficiently support both of these operations. Theoretical solutions exist [19] that can solve both rank and select in constant time while requiring only sublinear space (for a total of $n + o(n)$ space, including the original bitmap).

Rank operations are conceptually simple and can be answered efficiently by storing sampled values of cumulative counts at regular intervals, and sequentially counting values between samples. In order to achieve constant time, practical solutions [17] refine this method by using multi-level sampling and precomputed tables to efficiently compute the rank value from the stored samples. By doing this, we can provide very fast rank operations in practice with just 5% extra space in addition to the original bitmap.

Select operations can still be answered in constant time, but solutions are more complex and slower in practice than those for rank. Since most of the data structures used in this work rely solely on rank operations, we refer the reader to the original articles [30,17] for additional details on specific implementations.

3. Related work

3.1. Quadtrees

Quadtress were originally designed to compress images [20,21], although since then, they have been used for different purposes [39,41].

For our work, the most interesting variant is the region quadtree, which is a tree built from a recursive decomposition of the space into four squares of size power of 2. From now on, we denote these squares as *quadboxes*; each quadbox creates a node of the quadtree. In Fig. 1, we show the quadtree corresponding to a binary raster. The root of the tree corresponds to the whole raster, which is divided into 4 quadboxes delimited by thick and solid lines. They are represented in the tree as the four children of the root, whose label is 1-bit, if they contain at least one 1-bit, and 0-bit otherwise. The nodes labelled with a 0-bit are not divided anymore, while the rest are divided recursively following the same procedure, until reaching empty quadboxes (i.e. covered by 0-bits) or reaching the individual cells of the raster.

We denote the four quadboxes children of the root as q_1, q_2, q_3, and $q_4$from left to right and from top to bottom (see the children of the root in Fig. 1). The subsequent levels add one digit per level using the same procedure recursively, as shown in the nodes labelled with a 1-bit[3] in Fig. 1.

Since the original target of the quadtree was to compress images, several pointerless representations of the tree arose. There are two approaches, the first one is to use a *locational code* that for each node of the tree gives its position in the space [40], the second one is to use an implicit ordering [32].

The first example of implicit ordering is the *Treecodes* [32], where the quadtree is represented by a sequence of numbers, each representing a node of the quadtree, following a breadth-first traversal.

3.1.1. k^2tree

The k^2*tree* is a compact data structure initially designed to store web graphs [9], although later it has also been used, among other things, to store raster data [6].

Indeed, a k^2*tree* is a region quadtree represented with an implicit ordering approach. The k^2*tree* of the quadtree of Fig. 1 simply stores the nodes of the tree, following a breadth-first traversal, as a bit array. For practical purposes, the k^2*tree* actually uses two separate bit arrays, L, which is formed by the bits corresponding to the last level of the tree, and T, which contains the rest.

[3] We only show the names of these nodes in order to avoid cluttering the figure.

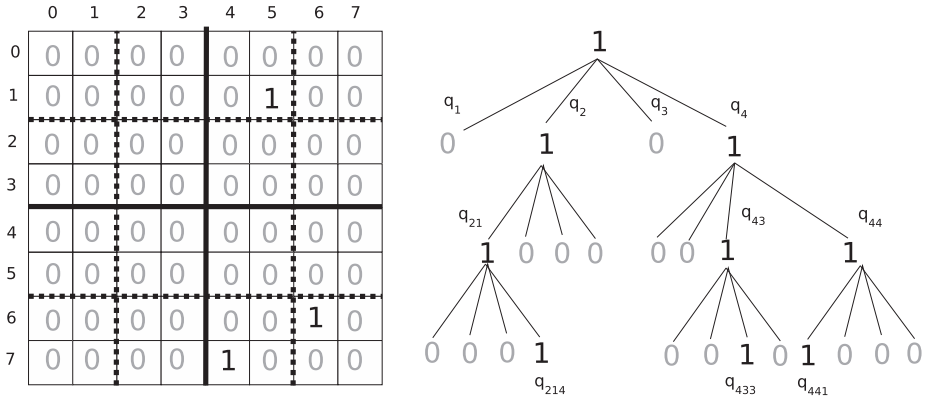

Fig. 1. A binary raster and its corresponding quadtree.

The k^2*tree* is able to answer several types of queries efficiently, including retrieving the value of a single cell, row/column queries, or general range reporting queries (i.e. report all the 1-bits in a range of rows/columns). All these algorithms are similar, and involve emulating a top-down traversal of the conceptual tree. To allow a fast emulation of the top-down traversal, T is equipped with additional data structures to perform *rank* operations on it.

Given a position p in T set to 1, the k^2 children corresponding to that node start at position $p_{children} = rank_1(T, p) \cdot k^2$ of T. If the children are leaves, that is, $p_{children} > |T|$, the values of the cells are retrieved by accessing $L[p_{children} - |T|]$. Thanks to the fast rank operations on T, all the top-down traversals required to answer queries can be answered efficiently using only the bitmaps T and L.

In addition, to obtain a more efficient data structure, instead of dividing each quadbox into four smaller quadboxes, the k^2*tree* can divide each quadbox into $k \times k$ quadboxes, where k is a parameter that can be set in each level.

3.1.2. k^3*tree*

A k^3*tree* can be obtained by simply adding a third dimension to the k^2*tree*. Fig. 2 shows a 3-dimensional binary matrix, its conceptual octree (a 3-dimensional quadtree), and its corresponding k^3*tree* (the bit arrays T and L).

The k^3-tree can be efficiently navigated using the same procedures of k^2*tree*, but extended to three dimensions.

3.2. Compact representations of rasters

3.2.1. k^2*raster*

The k^2*tree* is only capable of storing binary matrices. In [23,24], it was presented the k^2*raster*, a structure that is able to compress and index matrices of integer numbers. The data structure includes not only a spatial index, like in the region quadtree and k^2*tree*, but also an index of the values at cells.

As the k^2*tree*, the k^2*raster* uses a modern compact data structure approach, which supports efficient queries like retrieving the value of a specific cell or finding all cells containing values within a given range of values and/or in a given spatial range.

It divides the space using the same procedure as the k^2*tree*, but now the nodes of the conceptual tree contain the maximum and minimum values of the corresponding quadbox. The subdivision ends when the minimum and maximum values are equal or when the subdivision reaches the cells of the raster. The conceptual tree is compactly represented using binary bit arrays for the topology of the tree and differential encoding for the integers.

More specifically, considering a matrix of size $n \times n$, being n a power of k,[4] the root node of the tree is filled with the minimum and maximum values of the matrix. If these values are different, the matrix is divided into k^2 quadboxes of size n/k. Each of these quadboxes produces a child of the root node, where its minimum and maximum values are also stored. In case that these values are the same, the node/quadbox is not further subdivided, otherwise, the subdivision continues recursively until finding an uniform quadbox, where all the values are the same, or until reaching the cells of the matrix.

Fig. 3 shows an example of the division process (top), the conceptual (centre-top), and the final representation (bottom). The root node contains the minimum and maximum values of the original raster matrix, nodes at level 1 of the tree correspond to quadboxes of size 2×2, and the last level corresponds to the cells of the matrix. Observe that, for example, all values within the bottom-right 2×2 quadbox are the same, therefore it is not further subdivided and thus its corresponding node is a leaf.

[4] Otherwise, the matrix is expanded to size equal to the next power of k, filling it with 0 values. This does not require significant extra space.

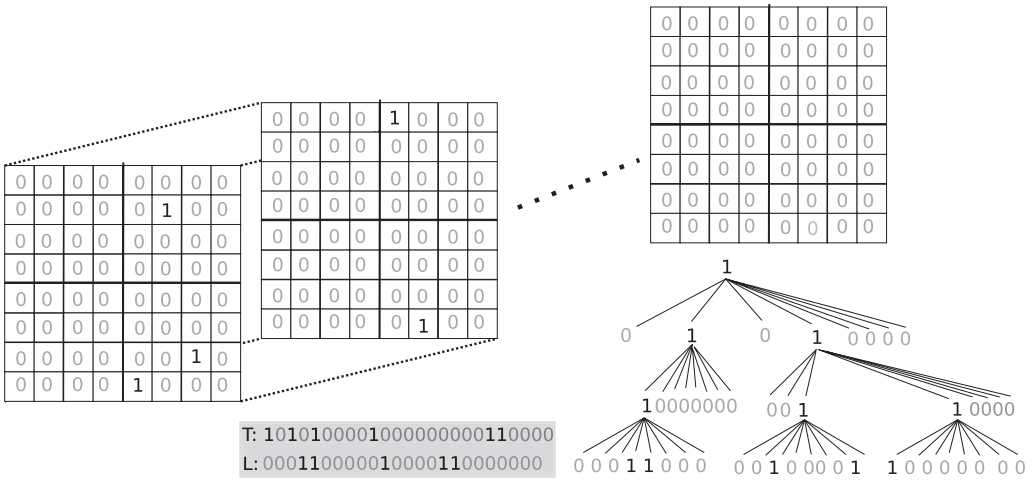

Fig. 2. A sequence of binary rasters and the corresponding k^3*tree*.

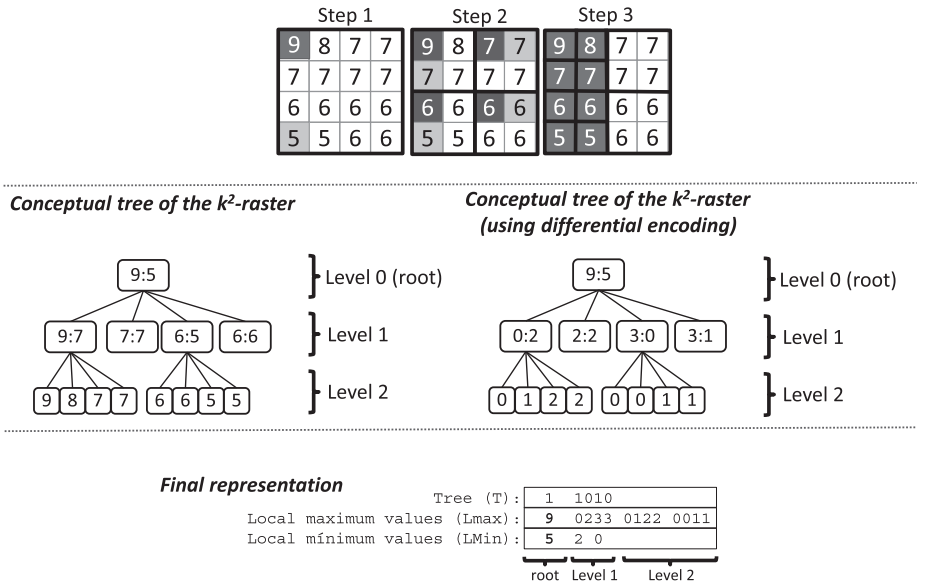

Fig. 3. Example (using $k = 2$) of integer raster matrix (top), conceptual tree of the k^2*raster*, conceptual tree with differential encoding, and final representation of the raster matrix.

The compressed representation contains two main parts. First, the topology of the tree is represented with the same strategy of a k^2*tree*, the only difference is that the last level is not needed, that is, only the bit array T is used. Second, the maximum and minimum values at nodes are encoded as the difference with respect to the maximum/minimum value stored at the parent node. These differences are stored in two sequences *Lmin* and *Lmax*, following again a breadth-first traversal of the conceptual tree. *Directly Addressable Codes* (DACs) [8] are used to store *Lmin* and *Lmax*; this technique obtains compression when storing small integers, and at the same time, provides direct access to any given position. At the last level of the tree, which corresponds to the original cells, only the maximum value is stored. Fig. 3 shows the conceptual tree after the differential encoding (middle-right), and the final representation (bottom).

The k^2*raster* also indexes the data, enabling fast queries on the raster matrix. It has a spatial index, using the tree structure, and an index of the values stored in the quadboxes, thanks to the minimum and maximum values stored in the nodes of the tree. It is possible to simulate a top-down traversal over the conceptual tree by accessing the bitmap T and the *Lmin* and *Lmax* arrays. Previous work [24] showed that some queries, such as finding cells having values within a specific range, are solved faster in a k^2*raster* than in a classical method for storing uncompressed rasters.

There is an enhancement to obtain further compression and speed called *heuristic k^2raster*. This version selects some frequent quadboxes of size bigger than 1×1, which are kept in a separate dictionary. In this way, in the positions of the tree where they would appear, a pointer to their entry in the dictionary is placed instead.

3.2.2. 3D2D-mapping

The 3D-2D mapping [35] transforms a raster matrix into a binary matrix, which is stored in a k^2*tree* to obtain compression and fast queries directly on it.

The transformation from an integer matrix into a binary one is based on the Morton order [29], also known as the Z-order curve. It is one of the most popular space-filling curves, which are functions that provide mapping from multidimensional space to one-dimensional space. These curves have been used as multidimensional indexes, mainly for secondary memory [41]. Morton order stands out as a computational efficient alternative with good preservation of the spatial locality (i.e. near elements in the original space are mapped to close positions in the one-dimensional space). The Morton code of an element can be computed in constant time using a bit interleaving of the binary codes of its indexes in each dimension.

In the left part of Fig. 4, we can see the Z-order curve over a raster matrix. In the right part, we can see the values of the matrix following the Z-order, and over them, the corresponding Morton codes.

Given a matrix M,[5] the mapping to a 2D structure begins by reading M in Z-order. This produces a vector V, which is used to construct a binary matrix BM as follows: cell $BM[x][y]$ is set to 1-bit if $V[x] = y$, otherwise, a 0-bit is stored.

In Fig. 5, we show the resulting matrix BM using the raster matrix of Fig. 3. V contains the values of M in Z-order, just below, BM has one row for each value, and one column for each Morton code. For each row, the 1-bits mark the cells in M having that value. For example, the value 5 is present in the positions 10 and 11 of V, and thus, the first row of BM has these positions set to 1-bit, and the rest store a 0-bit.

The k^2*tree* requires to store matrices of size $n \times n$, where n is a power of two. Therefore, if BM is of size $r \times c$, it is extended to size $n \times n$ with 0-bits in the created cells, being n the smallest power of 2 that is bigger than or equal to r and s. This extended matrix is called BM^n.

Fig. 5 shows the BM matrix corresponding to the raster used in Fig. 3. The matrix on the right is the extended version BM^{16}, which has 16 columns and rows. Finally, under the BM matrix, the k^2*tree* representing BM^{16} is depicted. The k^2*tree* is able to compress big areas full of 0-bits and, as it can be seen in the figures, the bitmaps T and L are much smaller than BM^{16}, and even smaller than the original BM.

Now, using the algorithms to query the k^2*tree*, it is possible to obtain the value of a cell or a window by using algorithms that transform the original spatial regions into the corresponding maximal quadboxes and their associated Morton codes [36,44].

3.3. Compact representation of raster time series

3.3.1. 4D3D-mapping

4D3D-mapping [11] is based on the 3D2D-mapping introduced in Section 3.2.2. The basic idea is to use that mapping to obtain a binary matrix from each original raster representing a time instant. Those binary matrices are extended with 0-bits to obtain a perfect 3D cube of size $m \times m \times m$, being m the smallest number power of 2 bigger than r, s, and τ, where $r \times s$ is the size of the BM matrices resulting from the 3D2D mapping of the original rasters, and τ is the number of time instants. Then, the resulting binary cube is stored using a k^3*tree*, which provides compressed storage and fast queries directly on it.

More formally, given the original rasters $\mathscr{M} = \langle M_1, M_2, \ldots, M_\tau \rangle$, the 3D2D mapping is applied to each M_i, thus obtaining $\langle BM_1, BM_2, \ldots, BM_\tau \rangle$ binary matrices. Recall that all these matrices have the same size $r \times c$. Let BM_i^m be:

- For $1 \leqslant i \leqslant \tau, BM_i$ extended with 0-bits until obtaining a matrix of size $m \times m$.
- For $\tau + 1 \leqslant i \leqslant m$, a matrix of size $m \times m$ full of 0-bits.

Finally, the 4D3D-mapping is obtained by storing $\langle BM_1^m, BM_2^m, \ldots, BM_m^m \rangle$ into a k^3*tree* of size $m \times m \times m$, called BC (from binary cube).

Fig. 6(a) shows a raster time series of three time instants. Fig. 6(b) shows the 3D2D mapping (BM_i) of each time instant. In the part (c), it is shown the resulting cube after extending $\langle BM_1, BM_2, BM_3 \rangle$ to obtain $\langle BM_1^{16}, BM_2^{16}, BM_3^{16} \rangle$ (i.e. matrices of size 16×16), and filling the cube with 13 additional matrices of size 16×16 full of 0-bits, in this way we obtain a cuboid of size $16 \times 16 \times 16$. Finally, the part 6(d) depicts the final representation of the raster time series, as the bitmaps T and L of a k^3*tree*.

[5] We consider it as a 3D structure since the first 2 dimensions are the (x,y) coordinates, and the third one is formed by the values stored at cells.

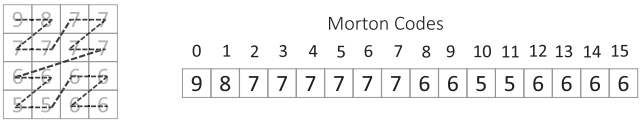

Fig. 4. Z-order curve (left) and the corresponding Morton codes (right).

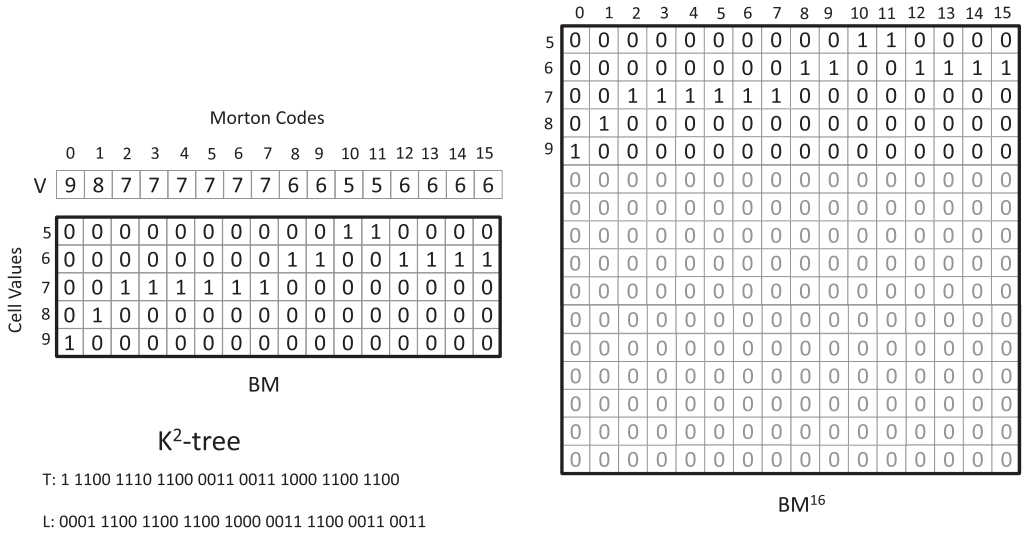

Fig. 5. 3D2D-mapping of the matrix of Fig. 3.

4. Basics

4.1. Problem definition

Let $\mathscr{M}$ be a raster matrix of size $n \times n$ that evolves along time with a timeline of size τ time instants. We can define $\mathscr{M} = \langle M_1, M_2, \ldots, M_\tau \rangle$ as the sequence of raster matrices M_i of size $n \times n$ for each time instant $i \in [1, \tau]$. We assume that each cell of the rasters stores an integer value.

4.2. Query types

In this paper, we deal with the following query types:

- *access*(r, c, t) retrieves the value of the cell (r, c) at time instant t. E.g. temperature in 100 Serrano Street, Madrid[6] at 2019–08-21 11 am.
- *windowQuery*$(r_1, r_2, c_1, c_2, t_1, t_2)$ retrieves all the cell values in a rectangular cuboid defined by its corners (r_1, c_1) and (r_2, c_2) and a time interval $[t_1, t_2]$. E.g. temperatures in Madrid from January to February.
- *rangeQuery*$(r_1, r_2, c_1, c_2, t_1, t_2)$*rMinrMax* retrieves all the cells inside a rectangular cuboid defined by the corners (r_1, c_1) and (r_2, c_2), and a time interval $[t_1, t_2]$, whose values are within the range $[rMin, rMax]$. E.g. zones (cells) in Madrid from January to Febraury with *moderate temperatures*, i.e. between 10 and 25 degrees Celsius. Note that this is the most general query type and some others (including the two above) can be described as particular cases. Another interesting particular case is when $t_1 = t_2$, which is usually referred in the literature of spatio-temporal databases as timestamp queries (in opposition to time-interval queries, when $t_1 \neq t_2$).

4.3. Definitions: q-rows and q-cols

Given a matrix of size n, we divide it into k groups of consecutive rows, each one containing (n/k) rows of the original matrix, and k groups of columns, each one containing (n/k) columns of the original matrix. We call those groups of rows *q-rows* and the groups of columns *q-cols*.

[6] For readability, in examples, we use place names, but query parameters are actually the rows and columns of the raster, either to define a point or a window.

Observe in Fig. 7(a) that, with $k = 2$, a matrix of size 8×8 has two *q-rows* (0,1), each one having 4 rows. In Fig. 7(b), we can see the two *q-cols*.

In addition, given a q-row $q - row_i$ (with n/k rows) and the original row $r \in \{0, \ldots, n-1\}$, we define $relative\_row(q - row_i, r)$ as follows:

- If $r \in q - row_i$, it returns the relative position of row r within $q - row_i$.
- If r is in a q-row prior to $q - row_i$, it returns a 0.
- If r is in a q-row after $q - row_i$, it returns $(n/k) - 1$.

A second function $relative\_col(q - col_i, c)$ works analogously, but with q-cols.

In Fig. 7(c), $relative\_row(1, 6)$ returns 2, whereas $relative\_row(0, 5)$ returns 3 and $relative\_row(1, 1) = 0$.

4.4. Naive solution

A straightforward baseline representation for the temporal raster matrix $\mathscr{M}$ can be obtained by simply representing each raster matrix M_i in a compact way with any of the compact representations in Section 3.2.

From the queries described above, the first one, $access(r, c, t)$, can be easily, and efficiently, solved by performing the equivalent query on the representation of the t-th raster. The same approach holds for the timestamp version of the other types of queries. Their general time-interval version involves querying several compact raster representations.

This kind of solution cannot take advantage of the similarities that are expected to exist between two consecutive rasters in the sequence, namely the same cell in two consecutive timestamps usually contains similar values. Hence, in the following two sections we describe a new approach that is designed precisely to take advantage of the temporal locality.

5. $T - k^2raster$

In a nutshell, our proposal is based on the $k^2raster$ introduced in Section 3.2.1, and a combination of *snapshots* and *logs*, which have been also used by space-efficient indexes in other domains such as trajectories of moving objects [7] or in image compression [14]. The idea is to use sampling at regular intervals of size t_δ. That is, we represent the full raster matrices at some time instants, called the snapshots, and use a differential encoding for the remaining time instants, the logs.

More formally, raster matrices corresponding to sampled time instants, $M_s, s = 1 + i \cdot t_\delta, i \in [0, (\tau - 1)/t_\delta]$, are represented in full with a $k^2raster$ and we will refer to them as *snapshots*. The $t_\delta - 1$ raster matrices $M_t, t \in [s + 1, s + t_\delta - 1]$ that follow a snapshot M_s are encoded using M_s as a reference. To do this, we create a modified $k^2rasterp$ to represent M_t. The $k^2rasterp$ is similar to a $k^2raster$ but, at each step of the construction process, the values in the quadboxes are encoded as differences with respect to the corresponding quadboxes in M_s, rather than as differences with respect to the parent node, as in a regular $k^2raster$.

In the $k^2rasterp$, we also encode the maximum and minimum values at nodes of the conceptual tree of M_t as differences to the corresponding values at the nodes of the snapshot. In addition, when a quadbox in M_t is identical to the same quadbox in M_s, or when all the values in both quadboxes differ only in a unique gap value α, we stop the recursive splitting process and simply keep a reference to the corresponding quadbox of M_s and the gap α (when they are identical, we just set $\alpha = 0$). In practice, keeping that reference is rather cheap as we only have to mark, in the conceptual tree of M_t, that the subtree rooted at a given node p has the same structure as the one of the conceptual tree of M_s. For such purpose, in the final representation of $k^2rasterp$, we include a new bitmap eqB, aligned to the zeroes in T. That is, if we have $T[i] = 0$ (node with no children), we set $eqB[rank_0(T, i)] \leftarrow 1$and set $Lmax[i] \leftarrow \alpha$. Also, if we have $T[i] = 0$, we set $eqB[rank_0(T, i)] \leftarrow 0$and $Lmax[i] \leftarrow \beta$ (where β is the difference between the maximum values of both quadboxes, as the value of the time instant is stored as that difference) to handle the case when the corresponding quadbox of M_t have all cells with the same value (as in a regular $k^2raster$).

The overall construction process of the $k^2rasterp$ for the matrix M_t related to the snapshot M_s can be summarised as follows. Let T_t be a bitmap with the same functionality of the bitmap T of the normal $k^2raster$; equally, we also have the arrays $Lmax_t$ and $Lmin_t$. Starting with the complete matrix M_t, we follow a recursive process. Let us consider q_{t_j} the quadbox of M_t processed in a call of the recursive process, and q_{s_j} the related quadbox in M_s. Let the corresponding maximum and minimum values of q_{t_j} be $maxval_t$ and $minval_t$, those of q_{s_j} be $maxval_s$ and $minval_s$, and let z_{t_j} be the position in the bitmap T_t corresponding to q_{t_j}:

- If q_{t_j} is a 1×1 quadbox, $Lmax_t[z_{t_j}] \leftarrow (maxval_t - maxval_s)$, and the recursion stops. Since in $k^2rasterp$, we have to deal both with positive and negative values, we actually apply the folklore *zig-zag*[7] encoding for the gaps $(maxval_t - maxval_s)$.
- If $maxval_t$ and $minval_t$ are equal, the recursive process stops. we set $T_t[z_{t_j}] \leftarrow 0, eqB[rank_0(T_t, z_{t_j})] \leftarrow 0$, and $Lmax_t[z_{t_j}] \leftarrow (maxval_t - maxval_s)$.

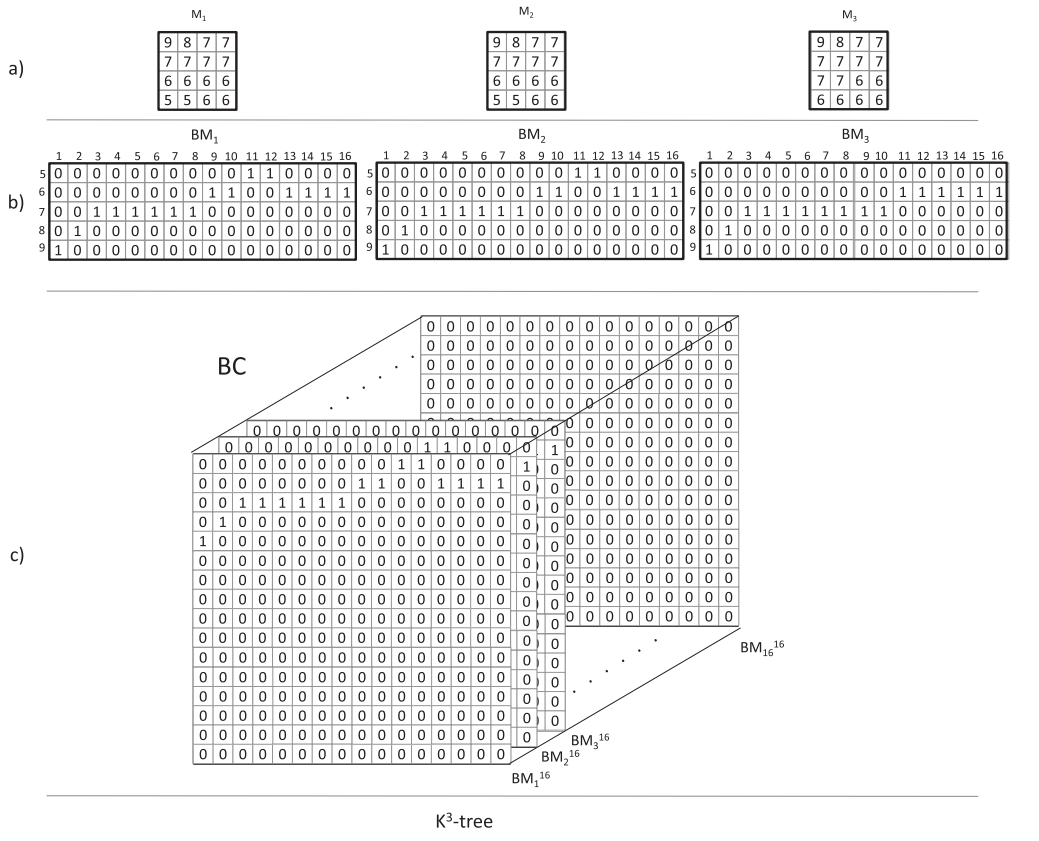

Fig. 6. 4D3D-mapping for the top left submatrices in Fig. 8.

- Let us consider the case when the values of cells at the same position of q_{t_j} and q_{s_j} differ by the same gap α (also, if the values are identical, hence $\alpha = 0$), more precisely, for each cell $c_{j_{lk}}$ of $q_{t_j}, c_{j_{lk}} = c_{s_{lk}} + \alpha$, being $c_{s_{lk}}$ the cell at the same position in q_{s_j}. Then, we set $T_t[z_{t_j}] \leftarrow 0, eqB[rank_0(T_t, z_{t_j})] \leftarrow 1$, and $Lmax_t[z_{t_j}] \leftarrow (maxval_t - maxval_s)$.
- Otherwise, we split q_{t_j} into k^2 quadboxes and continue recursively. We set $T_t[z_{t_j}] \leftarrow 1, Lmax_t[z_{t_j}] \leftarrow (maxval_t - maxval_s)$, and $Lmin_t[rank_1(T_t, z_{t_j})] \leftarrow (minval_t - minval_s)$.

Algorithm 1 shows the pseudocode. The arrays $Lmax[0 \ldots \log_k(n)], Lmin[0 \ldots \log_k(n)], T_t[0 \ldots \log_k(n)]$, and $eqB[0 \ldots \log_k(n)]$ store a stack in each entry. Each stack $T_t[i]$ (resp. $Lmax[i], Lmin[i]$, and $eqB[i]$) keeps the portion of T_j (resp. $Lmax, Lmin$, and eqB) corresponding to level i of the tree. We assume that these arrays, as well as the original matrices M_s and M_t, are global variables.

The Algorithm is initially launched with **build**$(n, 0, 0, 0)$, being n the number of rows/columns of the matrix. When it finishes, to obtain T_t, we simply join $T_t[0], T_t[1], \ldots$ into a single bit array. The same applies to $Lmax, Lmin$, and eqB. Each call to *build* processes one quadbox, initially the entire matrix, and recursion is applied to continue processing quadboxes of submatrices. Each recursive call returns a record containing 5 values: the maximum value of the quadbox in the snapshot, the maximum value of the quadbox in the current time instant, the corresponding minimum values in the snapshot and in the time instant, and a difference value; this difference will be set to ∞ in the general case, but if the snapshot and the time instant have a similar distribution of values (i.e. if all cells have the same value, or all of them differ by the same amount, between the current time and the snapshot), the fifth return value will be this difference.

lines 1–3 cover the simplest base case for recursion, when the processed quadbox is of size 1×1, i.e., an individual cell. In that case, the difference between the value of the processed cell at the time instant and the corresponding value at the snapshot (i.e. $M_t[r, c] - M_s[r, c]$) is added to $Lmax_t[l]$ (the $Lmax$ stack corresponding to the processed level l). The return record contains $M_s[r, c]$ and $M_t[r, c]$ as both maximum and minimum values of the corresponding quadboxes, and the difference of both as last element.

[7] 1 is represented as 1, −1 as 2, 2 as 3, −2 as 4, and so on.

lines 6 and 7 iterate over the q-rows and q-cols of the processed quadbox, in reverse order. Each iteration of the inner loop corresponds to one of the child quadboxes of the processed quadbox. Therefore, line 8 issues one recursive call for each child quadbox of size $(n/k) \times (n/k)$. As explained above, each recursive call returns the maximum value of the child quadbox in the snapshot ($maxval_{sij}$), the maximum value of the child quadbox in the time instant ($maxval_{tij}$), the minimum values in both quadboxes ($minval_{sij}$ and $minval_{tij}$), and the difference $diff_{ij}$ that may be set to ∞ or a value α.

The checks in lines 9–14 are also run for each child quadbox of the processed quadbox. lines 9–10 determine the value of *diff* from the value of the child quadboxes: line 9 sets *diff* to the value returned by the first child, since initially $diff = -\infty$. In subsequent iterations, if another child returns a different value for $diff_{ij}$, we know that the distribution of values, or structure, is not similar in the current quadbox, so *diff* is set to ∞. lines 11–14, similarly, compute the maximum and minimum values of the current quadbox from those of its children.

If the check of line 15 is true, that means that all cells of the processed quadbox have the same value, so we have to remove the entries in $Lmax, T_t, eqB$ corresponding to the child quadboxes. Recall that if a quadbox has the same value in all cells, then the corresponding node of the tree becomes a leaf. Note that we know this *after* processing the children of the processed quadbox, through a set of recursive calls (line 8). Those recursive calls to *build* have filled $Lmax, T_t, eqB$ of level $l+1$ with the corresponding information of the children of the processed quadbox. Therefore, lines 16–20 need to remove the values of $Lmax, T_t, eqB$ corresponding to those children. The *if* of line 18 checks if l is the level before the last one of the tree, since bitmaps T_t and *eqB* are not filled for the last level of the tree. Then, lines 21–22 mark the current quadbox as a leaf, and line 23 stores the difference in maximums between the time instant and the snapshot in *Lmax*. lines 24–32 do the same process, but when the processed node has the same structure in the snapshot and the time instant. This is signalled with a value of *diff* different from ∞. The only difference with respect to the case shown above is that the *epB* gets a 1 in the position of the processed quadbox, to signal the type of leaf.

If the flow reaches line 33, that means that the node corresponding to the processed quadbox is not a leaf, and thus its corresponding position in T_t is set to 1 (line 34), and its positions in *Lmax* and *Lmin* are filled with the differences between the maximum and minimum values, respectively, between the values of the time instant and the snapshot.

Finally, as explained above, the Algorithm must return the maximum and minimum values in the current time and in the snapshot and the difference *diff*, that have been computed in lines 6–14.

Algorithm 1: build(n, l, r, c) builds k^{2}raster′

```
1  if n=1 then
2  |  Lmax_t[l].push(M_t[r,c] − M_s[r,c])
3  |  return (M_s[r,c], M_t[r,c], M_s[r,c], M_t[r,c], M_t[r,c] − M_s[r,c])
4  else
5  |  maxval_t ← −∞; maxval_s ← −∞; minval_t ← ∞; minval_s ← ∞; diff ← −∞;
6  |  for i ← k − 1 ... 0 do /* Iterates over its q-rows */
7  |  |  for j ← k − 1 ... 0 do /* Iterates over its q-cols */
8  |  |  |  (maxval_sij, maxval_tij, minval_sij, minval_tij, diff_ij) ←build(n/k, l + 1, r + i · (n/k), c +
   |  |  |    j · (n/k))
9  |  |  |  if diff = −∞ then diff = diff_ij;
10 |  |  |  else if diff ≠ diff_ij then diff ← ∞;
11 |  |  |  if maxval_sij > maxval_s then maxval_s ← maxval_sij;
12 |  |  |  if minval_sij < minval_s then minval_s ← minval_sij;
13 |  |  |  if maxval_tij > maxval_t then maxval_t ← maxval_tij;
14 |  |  |  if minval_tij < minval_t then minval_t ← minval_tij;
15 |  if maxval_t = minval_t then /* All values are equal */
16 |  |  for i ← 0 ... k^2 − 1 do /* Remove the children, now this node is a leaf */
17 |  |  |  Lmax[l + 1].pop
18 |  |  |  if n > k then
19 |  |  |  |  T_t[l + 1].pop
20 |  |  |  |  eqB[l + 1].pop
21 |  |  T_t[l].push(0)
22 |  |  eqB[l].push(0)
23 |  |  Lmax_t[l].push(maxval_t − maxval_s)
24 |  else if diff ≠ ∞ then /* Quadbox with the same structure as that of the snapshot */
25 |  |  for i ← 0 ... k^2 − 1 do /* Remove the children, now this node is a leaf */
26 |  |  |  Lmax[l + 1].pop
27 |  |  |  if n > k then
28 |  |  |  |  T_t[l + 1].pop
29 |  |  |  |  eqB[l + 1].pop
30 |  |  T_t[l].push(0)
31 |  |  eqB[l].push(1)
32 |  |  Lmax_t[l].push(maxval_t − maxval_s)
33 |  else /* The node is not a leaf */
34 |  |  T_t[l].push(1)
35 |  |  Lmax_t[l].push(maxval_t − maxval_s)
36 |  |  Lmin_t[l].push(minval_t − minval_s)
37 |  return (maxval_s, maxval_t, minval_s, minval_t, diff)
```

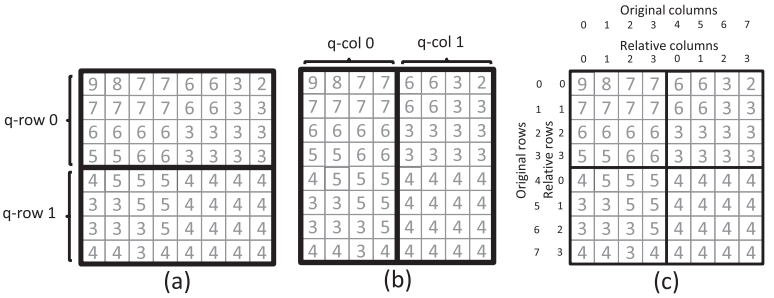

Fig. 7. Definitions.

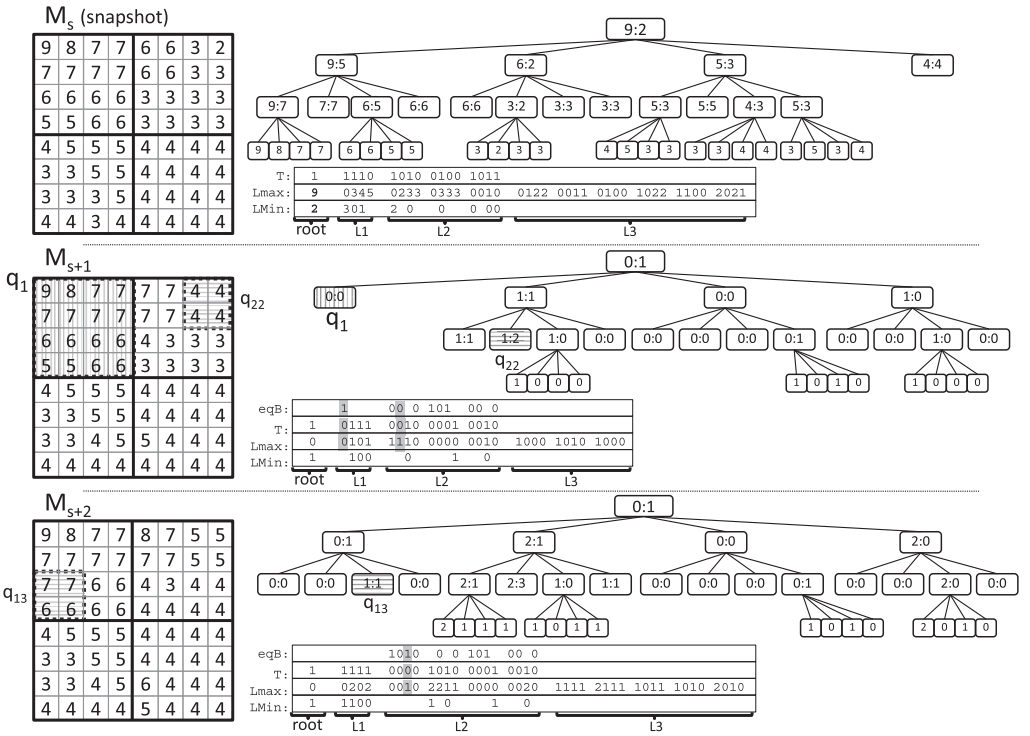

Fig. 8. Structures involved in the creation of a $T-k^2raster$ considering $\tau = 3$.

Fig. 8 includes an example of the structures involved in the construction of a $T-k^2raster$ over a temporal raster of size 8×8, with $\tau = 3$. The raster matrix corresponding to the first time instant becomes a *snapshot* that is represented exactly as the $k^2raster$ in Fig. 3. The remaining raster matrices M_{s+1} and M_{s+2} are represented with two $k^2rasterp$ that are built taking M_s as a reference. We have highlighted some particular nodes in the differential conceptual trees corresponding to M_{s+1} and M_{s+2}.

(i) the node labelled $\langle 0:0\rangle$ highlighted with vertical lines in M_{s+1} indicates that the 4×4 quadboxes q_1 of M_s and M_{s+1} are identical. Therefore, node $\langle 0:0\rangle$ has no children, and we set: $T[1]\leftarrow 0, eqB[0]\leftarrow 1$, and $Lmax[1]\leftarrow 0$. Note that $T, eqB, Lmax$ and $Lmin$ start numbering at position 0, although we introduced blank spaces in eqB and $Lmin$ to align their bits to the corresponding positions in T.

[8] Observe in Fig. 1 that the rows and columns are numbered starting at 0.

(ii) the node labelled $\langle 1:1\rangle$ highlighted with horizontal lines in M_{s+2} illustrates the case in which all the values of q_{13} are increased by $\alpha \leftarrow 1$. In this case, the values $\langle 6,6,5,5\rangle$ of q_{13} in M_s become $\langle 7,7,6,6\rangle$ in q_{13} of M_{s+2}. Again, the recursive traversal stops at that node, and we set: $T[7] \leftarrow 0, eqB[2] \leftarrow 1$, and $Lmax[7] \leftarrow 1$ (values are increased by 1).

(iii) the node labelled $\langle 1:2\rangle$ highlighted with horizontal lines in M_{s+1}. In this case, q_{22} in M_{s+1} has the same maximum and minimum values (set to 4), consequently, the recursive process stops again. In this case, we set $T[6] \leftarrow 0, eqB[2] \leftarrow 0$, and $Lmax[6] \leftarrow 1$.

5.1. Querying

5.1.1. Obtaining a cell value in a time instant

As defined in Section 4.2, $access(r,c,t)$ retrieves the value of the cell at row r and column c [8] of the raster at time instant t. For solving this query, there are two cases: if t is represented by a snapshot, then the Algorithm to obtain a cell in the regular k^2*raster* is used (see Algorithm 2, *getCell*, in [24]), otherwise, a synchronised top-down traversal of the trees representing such time instant (M_t) and the closest previous snapshot (M_s, where $s = t - tmodt_\delta$) is required. Algorithm 2 is a wrapper that delegates in the appropriate procedure according to the case.

Algorithm 2: **access**(r,c,t) returns the value of the cell (r,c) of the raster t

```
if t(modt_δ) = 0 then /* snapshot */
  returnM_t.getCell_Normal_k²-raster(r, c)// Algorithm 2 in [24]
else
  return M_t.getCell(n, r, c, 0, 0, Lmax_s[0], Lmax_t[0])// Algorithm 3
```

Algorithm 3: **getCell**$(n,r,c,z_s,z_t,maxval_s,maxval_t)$ returns the value at cell (r,c)

```
1  if z_s ≠ −1 then
2  |  z_s ← (rank_1(T_s, z_s) − 1) · k² + 1
3  |  z_s ← z_s + ⌊r/(n/k)⌋ · k + ⌊c/(n/k)⌋
4  |  maxval_s ← maxval_s − Lmax_s[z_s]
5  if z_t ≠ −1 then
6  |  z_t ← (rank_1(T_t, z_t) − 1) · k² + 1
7  |  z_t ← z_t + ⌊r/(n/k)⌋ · k + ⌊c/(n/k)⌋
8  |  maxval_t ← Lmax_t[z_t])
9  if (z_s > |T_s| or z_s = −1 or T_s[z_s] = 0) and (z_t > |T_t| or z_t = −1 or T_t[z_t] = 0) then /* Both leaves
     */
10 |  return maxval_s + ZigZag_Decoded(maxval_t)
11 else if z_s > |T_s| or z_s = −1 or T_s[z_s] = 0 then  /* Leaf in Snapshot */
12 |  z_s ← −1
13 |  return getCell(n/k, r mod (n/k), c mod (n/k), z_s, z_t, maxval_s, maxval_t)
14 else if z_t > |T_t| or z_t = −1 or T_t[z_t] = 0 then  /* Leaf in time instant */
15 |  if z_t ≠ −1 and T_t[z_t] = 0 then
16 |  |  eq ← eqB[rank_0(T_t, z_t) − 1]
17 |  |  if eq = 1 then z_t ← −1 ;
18 |  |  else return maxval_s + ZigZag_Decoded(maxval_t) ;
19 |  return getCell(n/k, r mod (n/k), c mod (n/k), z_s, z_t, maxval_s, maxval_t)
20 else  /* Both internal nodes */
21 |  return getCell(n/k, r mod (n/k), c mod (n/k), z_s, z_t, maxval_s, maxval_t)
```

Focusing on the second case, the synchronised traversal inspects the two nodes at each level corresponding to the quadbox that contains the queried cell. The main issue here is that the shape of the trees representing M_t and M_s will be different in most cases. Therefore, it is possible for one of the traversals to reach a leaf earlier than the other. In that case, the traversal that did not reach a leaf continues, but the process must remember the value reached in the leaf of the other tree, since that is the value that will be added to or subtracted from the one found when the other traversal reaches a leaf. Indeed, three cases are possible: (a) the processed quadbox of M_t is uniform, (b) the quadbox of M_s is uniform and, (c) the processed quadbox after applying the differences with the snapshot has the same value in all cells.

Algorithm 3 describes this synchronised traversal. Variable z_s is used to store the current position in the bitmap T of M_s (i.e. T_s) during the downward traversal at any given step of the algorithm; similarly, z_t stores the position in T of M_t (i.e. T_t). When z_s (resp. z_t) has a -1 value, it means that the traversal reached a leaf and, in $maxval_s$ (resp. $maxval_t$) the Algorithm keeps the maximum value stored at that leaf node. Note that, $T_s, T_t, Lmax_s, Lmax_t$, and k are global variables. As shown in

Algorithm 2, this auxiliary procedure is invoked with $z_s = z_t = 0, maxval_s = Lmax_s[0]$ and $maxval_t = Lmax_t[0]$. We assume that the cell at position $(0,0)$ of a raster is the one in the upper-left corner.

In lines 1–8, the Algorithm obtains the child of the processed node that contains the queried cell, provided that in a previous step, the Algorithm did not reach a leaf node (signalled with z_s or z_t set to -1). In $maxval_s$ (resp. $maxval_t$), the Algorithm stores the maximum value stored in that node.

If the condition in line 9 is true, the Algorithm has reached a leaf in both trees, and thus the values stored in $maxval_s$ and $maxval_t$ are added to obtain the final result. If the condition in line 11 is true, the Algorithm has reached a leaf in the snapshot. This is signalled by setting z_s to -1 (it may already contain such value) and then a recursive call continues the process.

The condition in line 14 deals with the case of reaching a leaf in M_t. If the condition of line 15 is true, the Algorithm uses bitmap *eqB* to check whether the quadbox of M_t has the same structure as that of the original M_s quadbox or the quadbox of M_t has the same value in all cells. A 1-bit in *eqB* implies the latter case, and this is solved by setting z_t to -1 and performing a recursive call. A 0-bit implies that we can obtain the value in all the cells by adding the values stored in $maxval_s$ and $maxval_t$, since that value is encoded as a difference between the maximum values of both quadboxes, and thus the traversal ends.

The last case is when the nodes are internal, which simply requires a recursive call.

Fig. 9 illustrates how the Algorithm computes the value of cell (3,2) at $M_t = M_{s+1}$. The call **access(3,2,M_t)** launches **getCell (8,3,2,0,0,9,0)** over M_{s+1}.

In lines 2–4, the Algorithm computes the bit position in T_s corresponding to the child of the root that contains the queried cell. First, it computes $z_s = (rank_1(T_s, 0) - 1) \cdot k^2 + 1 = (1-1) \cdot 2^2 + 1 = 1$ and then $z_s = 1 + \lfloor (3/4) \rfloor \cdot 2 + \lfloor 2/4 \rfloor = 1$, corresponding to the quadbox, q_1. Line 4 computes the maximum value in that quadbox, $maxval_s = maxval_s - Lmax_s[1] = 9 - 0 = 9$.

lines 6–8 do the same with $T_t : z_t = 1$ and $maxval_s = Lmax_t[1] = 0$. In this case, the conditions in lines 9 and 11 are false, but $T_t[1] = 0$, which indicates that the corresponding node is a leaf (see the node labelled with q_1 in the tree corresponding to M_{s+1}). Therefore the condition in line 14 returns true. Since $z_t \neq -1$, then the condition in line 15 is also true, and then the Algorithm obtains the value in the *eqB* bitmap $eq = eqB[rank_0(T_t, z_t) - 1] = eqB[1-1] = 1$, so z_t is set to -1. This means that the subtree corresponding to q_1 of M_{s+1} is equal to that of the snapshot, possibly adding (or subtracting) a value α. Next, we reach line 19, which launches the call **getCell(4,3,2,1,-1,9,0)**. This recursive call processes q_1.

lines 2–4 obtain the position in T_s of the child of q_1 that contains the queried cell: $z_s = (rank_1(T_s, 1) - 1) \cdot k^2 + 1 = (2-1) \cdot 4 + 1 = 5$ and $z_s = 5 + \lfloor (3/2) \rfloor \cdot 2 + \lfloor 2/2 \rfloor = 8$is the position corresponding to the quadbox q_{14}. The maximum value in that quadbox is $maxval_s = maxval_s - Lmax_s[8] = 9 - 3 = 6$. Since, $z_t = -1$, lines 6–7 are not executed.

The condition in line 9 returns true since $T_s[8] = 0$, as we also reached a leaf in the snapshot (see the node labelled q_{14} in the tree corresponding to M_s), and $z_t = -1$. Therefore, the Algorithm returns $maxval_s + ZigZag\_Decoded(maxval_t) = 6 + 0 = 6$.

5.1.2. Obtaining a spatial range

Operation $windowQuery(r_1, r_2, c_1, c_2, t_1, t_2)$ retrieves all the values in a region, and a time interval. For solving it, our Algorithm has to query each raster in the time interval $[t_1, t_2]$.

As in the previous operation, for each $t_i \in [t_1, t_2]$ there are two possible cases: (i) t_i is a snapshot, represented as a $k^2raster$, and thus a *getWindow* operation for normal $k^2raster$ (see [Algorithm 3][24]) is issued; (ii) t_i is a log and therefore a synchronised traversal of the $k^2raster$p representing the time instant t_i and the $k^2raster$ representing the closest previous snapshot is required. A main change in this traversal, with respect to the one used in the previous operation, is that the traversal may require following several branches of both trees, since the queried region can overlap quadboxes corresponding to several nodes of the tree.

Algorithm 4: **windowQuery**$(r_1, r_1, c_2, c_2, t_1, t_2)$ returns the value of the cuboid defined by the spatial corners $[r_1, c_1]$, $[r_2, c_2]$ and time interval $[t_1, t_2]$

for $t_i \in [t_1 \ldots t_2]$ **do**
  if $t_i (mod t_\delta) = 0$ **then** /* snapshot *
    R$[t_i] \leftarrow M_{t_i}$.getWindow_Normal_k^2-raster(r_1, r_2, c_1, c_2)Algorithm 3 in [24]
else
    R$[t_i] \leftarrow M_{t_i}$.getWindow$(n, r_1, r_2, c_1, c_2, 0, 0, Lmax_s[0], Lmax_t[0])$ Algorithm 4
return$R$

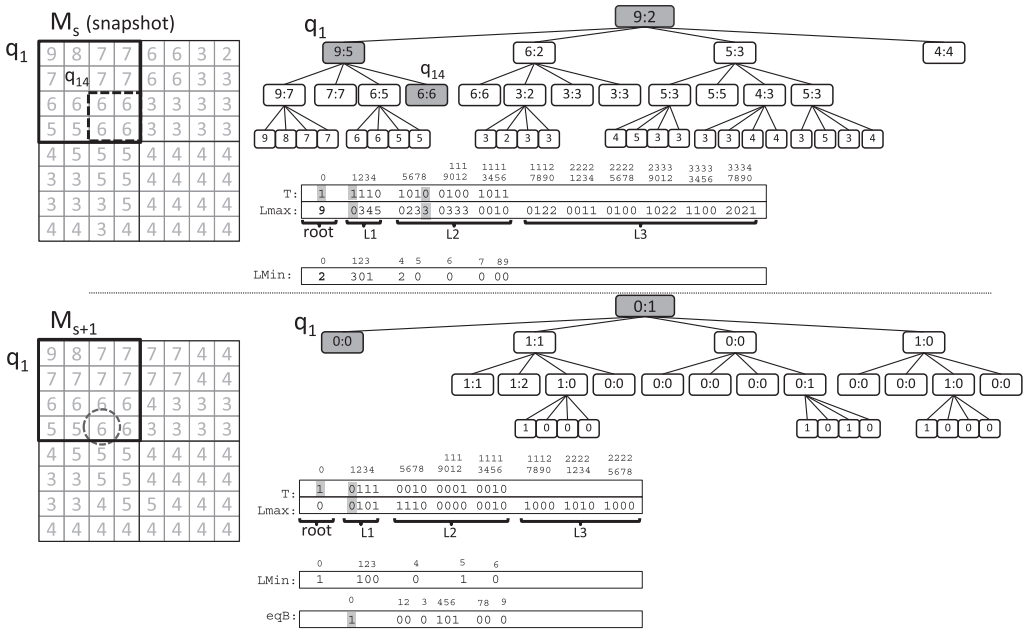

Fig. 9. Obtaining the cell (3,2) of M_{s+1}.

Algorithm 5: **getWindow**$(n, r_1, r_2, c_1, c_2, z_s, z_t, maxval_s, maxval_t)$ returns all cells from region $[r_1, c_1]$ to $[r_2, c_2]$

```
1  if z_s ≠ −1 then z_s ← (rank_1(T_s, z_s) − 1) · k^2 + 1;
2  if z_t ≠ −1 then z_t ← (rank_1(T_t, z_t) − 1) · k^2 + 1;
3  for each q-row i overlapping [r_1, r_2] do
4      r'_1 ← relative_row(i, r_1);   r'_2 ← relative_row(i, r_2);
5      for each q-col j overlapping [c_1, c_2] do
6          c'_1 ← relative_col(j, c_1);   c'_2 ← relative_col(j, c_2);
7          z'_s = z'_t = −1;
8          if z_s ≠ −1 then
9              z'_s ← z_s + k · i + j;
10             maxval_s ← maxval_s − Lmax_s[z_s];
11         if z_t ≠ −1 then
12             z'_t ← z_t + k · i + j;
13             maxval_t ← Lmax_t[z_t];
14         if (z'_s > |T_s| or z'_s = −1 or T_s[z'_s] = 0) and (z'_t > |T_t| or z'_t = −1 or T_t[z'_t] = 0) then
             /* Both leaves */
15             return maxval_s + ZigZag_Decoded(maxval_t) ((r'_2 − r'_1) + 1) · ((c'_2 − c'_1) + 1) times
16         else if z'_s > |T_s| or z'_s = −1 or T_s[z'_s] = 0 then  /* Leaf in Snapshot */
17             z'_s ← −1;
18             return getWindow(n/k, r'_1, r'_2, c'_1, c'_2, z'_s, z'_t, maxval_s, maxval_t);
19         else if z'_t > |T_t| or z'_t = −1 or T_t[z'_t] = 0 then  /* Leaf in time instant */
20             if z_t ≠ −1 and T_t[z'_t] = 0 then
21                 eq ← eqB[rank_0(T_t, z'_t) − 1];
22                 if eq = 1 then z'_t ← −1 ;
23                 else return maxval_s + ZigZag_Decoded(maxval_t) ((r'_2 − r'_1) + 1) · ((c'_2 − c'_1) + 1)
                     times ;
24             return getWindow(n/k, r'_1, r'_2, c'_1, c'_2, z'_s, z'_t, maxval_s, maxval_t);
25         else  /* Both internal nodes */
26             return getWindow(n/k, r'_1, r'_2, c'_1, c'_2, z'_s, z'_t, maxval_s, maxval_t);
```

Algorithm 4 traverses each of the time instants of the queried time interval and calls the appropriate Algorithm for each time instant.

Next, we illustrate the steps of Algorithm 4 for obtaining the window $[0,3] \times [1,4]$ (see the box with very sparse and thick dotted lines in the matrix M_{s+1} of Fig. 10) in the time interval $[M_{s+1}, M_{s+1}]$. Since our query covers only one time instant (the raster M_{s+1}), the query is solved with only the call **getwindow(8,0,1,3,4,0,0,9,0)** on M_{s+1}, which is solved with Algorithm 5.

Each call to getWindow processes one quadbox. The call getwindow(8,0,1,3,4,0,0,9,2) processes the complete matrix M_{s+1}. From line 1 to line 13, the Algorithm cuts the query window into the parts that overlap each of the k^2 quadboxes that comprise the processed quadbox. Then, as it can be seen in Fig. 10, our query region overlaps q_1 and q_2. Therefore, each of these two pieces are processed separately.

lines 1–2 set the pointers of T_s and T_t to the position of the first child of the processed quadbox. In getwindow (8,0,1,3,4,0,0,9,2), z_s and z_t are set to 1, which points to q_1, the first child of the root. The variable i iterates over the q-rows and j iterates over the q-cols that overlap the query window. In our example, i only takes the value of q-row 0 and j iterates between the q-cols 0 and 1.

Each combination of q-row and q-col (i,j) designs a quadbox, for example, the combination $i = 0$ and $j = 0$ corresponds to the quadbox of size 4×4, q_1. Then, the variables $r_1\prime, r_2\prime, c_1\prime$, and $c_2\prime$ are set to the relative rows and columns of the queried window within the processed q-row and q-col. In q_1, the queried window covers from relative row $r_1\prime = 0$ to row $r_2\prime = 1$ of q-row 0 and from relative column $c_1\prime = 3$ to $c_2\prime = 3$ of q-col 0. Observe in Fig. 10 that those are the rows and columns of q_1 that overlap the queried region (see in matrix M_{s+1}, the part of the queried region with vertical lines).

Line 9 moves an auxiliary pointer $z_s\prime$ to point to the position in T_s corresponding to the processed child, in this case, $z_s\prime$ continues pointing to position 1, since q_1 is the first child of the root node. The maximum value of q_1 is computed in line 10, which is $maxval_s = maxval_s - Lmax_s[1] = 9 - 0 = 9$. lines 12–13 set $z_t\prime = 1$ and $maxval_t = 0$.

The conditions in lines 14 and 16 are false since $T_s[1] = 1$, but that of line 19 is true since $T_t[1] = 0$. Given that $eq = 1$, then $z_t\prime$ is set to -1, indicating that we reached a leaf in the tree corresponding to M_{s+1}. Then, in line 24, a new call is issued **getwindow(4,0,1,3,3,1,-1,9,0)**. This call processes q_1.

In the recursive call, line 1 sets $z_s = (rank_1(T_s, 1) - 1) \cdot k^2 + 1 = (2 - 1) \cdot 4 + 1 = 5$. From line 3 to line 13, we find that the only quadbox of size 2×2 that overlaps the queried window is q_{12} (we highlighted it in M_s, both the region in the matrix and the corresponding node in the tree), signalled by q-row $i = 0$ and q-col $j = 1$. Line 9 sets $z_s\prime$ to point to the bit in T_s corresponding to that quadbox: $z_s\prime = z_s + (k \cdot i) + j = 6$. Line 10 obtains $maxval_s = maxval_s - Lmax_s[6] = 9 - 2 = 7$.

The variables $r_1\prime, r_2\prime, c_1\prime$, and $c_2\prime$ are set to relative row $r_1\prime = 0$ to relative row $r_2\prime = 1$ and from relative column $c_1\prime = 1$ to $c_2\prime = 1$, since those are the relative rows and columns within q_{12} that overlap the queried region.

At this point, since $T_s[6] = 0$ (we reached a leaf) and we had already reached a leaf in T_t, the condition in line 14 is true, and the call returns $maxval_s + ZigZag\_Decoded(maxval_t) = 7 + 0 = 7$ $((r_2\prime - r_1\prime) + 1) \cdot ((c_2\prime - c_1\prime) + 1) = ((1 - 0\prime) + 1) \cdot ((1 - 1\prime) + 1) = 2$ times, that is, two 7 values, corresponding to the fragment of the query window that overlaps q_{12}. In Fig. 10, in M_{s+1}, the portion of the queried region is highlighted with vertical lines. This ends the call getwindow(4,0,1,3,3,1,-1,9,0).

Next, the flow returns to the first call getwindow(8,0,1,3,4,0,0,9,2). The *for each* in line 5 sets j to the q-col 1 (maintaining the q-row $i = 0$), this signals the quadbox q_2. lines 9–10 adjust $z_s\prime = z_s + k \cdot i + j = 1 + 2 \cdot 0 + 1 = 2$, to point to q_2 and sets $maxval_s = maxval_s - Lmax_s[2] = 9 - 3 = 6$. lines 12–13 set $z_t\prime = 2$ and $maxval_t = Lmax_t[2] = 1$.

Conditions in lines 14, 16, and 19 are false, thus the flow reaches line 26 and then the call **getwindow(4,0,1,1,1,2,2,6,1)** is issued. This call processes q_2, and thus, the remaining part of the query window. lines 1–2 set $z_s = 9$ and $z_t = 5$. lines 3 and 5 set $i = j = 0$, thus in lines 9–10 we set $z_s\prime = 9$ and $maxval_s = maxval_s - Lmax_s[9] = 6 - 0 = 6$, and lines 12–13 set $z_t\prime = 5$ and $maxval_t = Lmax_t[5] = 1$.

Finally, since $T_s[9] = 0$ and $T_t[5] = 0$, we reached a leaf in both trees, and then the condition in line 14 is true, thus returning $maxval_s + ZigZag\_Decoded(maxval_t) = 6 + 1 = 7, ((r_2\prime - r_1\prime) + 1) \cdot ((c_2\prime - c_1\prime) + 1) = ((1 - 0\prime) + 1) \cdot ((0 - 0\prime) + 1) = 2$ times, that is, two 7 values, corresponding to the slice of the queried region that overlaps q_2, see in Fig. 10 the portion of the queried region with vertical lines. This ends the query.

5.1.3. Obtaining cells with a range of values

Algorithm 6: **rangeQuery**$(r_1, r_2, c_1, c_2, t_1, t_2, rMin, rMax)$ returns the value of the cuboid defined by the spatial corners $[r_1, c_1]$, $[r_2, c_2]$ and time interval $[t_1, t_2]$ whose values are within the range $[rMin, rMax]$

for $t_i \in [t_1 \dots t_2]$ **do**
  if $t_i(modt_\delta) = 0$ **then** /* snapshot */
    R$[t_i] \leftarrow M_{t_i}$.searchValuesInWindow_Normal_k^2-raster(r_1, r_2, c_1, c_2) Algorithm 2 in [24]
  else
    R$[t_i] \leftarrow M_{t_i}$.sValWin$(n, r_1, r_2, c_1, c_2, 0, 0, Lmax_s[0], Lmax_t[0], Lmin_s[0], Lmin_t[0], rMin, rMax)$ Algorithm 3
return M

Algorithm 7: sValWin$(n, r_1, r_2, c_1, c_2, z_s, z_t, maxval_s, maxval_t, minval_s, minval_t, rMin, rMax)$ returns all cells from region $[r_1, r_2]$, $[c_1, c_2]$ whose values are within the range $[rMin, rMax]$

```
1  maxval = maxval_s + ZigZag_Decoded(maxval_t)
2  minval = minval_s + ZigZag_Decoded(minval_t)
3  if minval ≥ rMin and maxval ≤ rMax then /* all cells meet the condition in this branch */
4  |  Output corresponding region of cells
5  |  return
6  else if minval > rMax or maxval < rMin then /* no cells meet the condition in this branch */
7  |  return
8  else if z_s ≠ −1 or z_t ≠ −1 then
9  |  if z_s ≠ −1 then z_s ← (rank_1(T_s, z_s) − 1) · k^2 + 1;
10 |  if z_t ≠ −1 then z_t ← (rank_1(T_t, z_t) − 1) · k^2 + 1;
11 |  for each q-row i overlapping [r_1, r_2] do
12 |  |  r'_1 ← relative_row(i, r_1);  r'_2 ← relative_row(i, r_2);
13 |  |  for each q-col j overlapping [c_1, c_2] do
14 |  |  |  c'_1 ← relative_col(j, c_1);  c'_2 ← relative_col(j, c_2);
15 |  |  |  z'_s = z'_t = −1;
16 |  |  |  if z_s ≠ −1 then
17 |  |  |  |  z'_s ← z_s + k · i + j;
18 |  |  |  |  maxval_s ← maxval_s − Lmax_s[z'_s];
19 |  |  |  |  if z'_s < |T_s| and T_s[z'_s] = 1 then minval_s ← minval_s + Lmin_s[rank_1(T_s, z'_s) − 1];
20 |  |  |  if z_t ≠ −1 then
21 |  |  |  |  z'_t ← z_t + k · i + j;
22 |  |  |  |  maxval_t ← Lmax_t[z'_t];
23 |  |  |  |  if z'_t < |T_t| and T_t[z'_t] = 1 then minval_t = Lmin_t[rank_1(T_t, z'_t) − 1];
24 |  |  |  if z'_s > |T_s| or z'_s = −1 or T_s[z'_s] = 0; then /* Leaf in Snapshot */
25 |  |  |  |  minval_s ← maxval_s
26 |  |  |  |  z'_s ← −1;
27 |  |  |  if z'_t > |T_t| or z'_t = −1 or T_t[z'_t] = 0 then /* Leaf in time instant */
28 |  |  |  |  if z'_t ≠ −1 and T_t[z'_t] = 0 and eqB[rank_0(T_t, z'_t) − 1] = 0 then
29 |  |  |  |  |  minval_t ← (maxval_s + maxval_t) − minval_s;
30 |  |  |  |  else if z'_t ≠ −1 and T_t[z'_t] = 0 and eqB[rank_0(T_t, z'_t) − 1] = 1 then
31 |  |  |  |  |  minval_t ← maxval_t;
32 |  |  |  |  z'_t ← −1;
33 |  |  |  return
   |  |  |  |  sValWin(n/k, r'_1, r'_2, c'_1, c'_2, z'_s, z'_t, maxval_s, maxval_t, minval_s, minval_t, rMin, rMax)
```

The most general query type, *rangeQuery*$(r_1, r_2, c_1, c_2, t_1, t_2)$*rMinrMax*, retrieves all the cells inside a rectangular cuboid defined by $[r_1, c_1]$, $[r_2, c_2]$, and a time interval $[t_1, t_2]$, whose values are within the range $[rMin, rMax]$. The Algorithm is similar to the previous one, being the only difference that, at each node, we must check whether the maximum and minimum values in that quadbox are compatible with the queried range, discarding those that fall outside the range of values sought.

Algorithms 6 and 7 show the pseudocode of this query. As in previous queries, Algorithm 6 checks if the queried time instant is represented with a snapshot or with a log, and then it delegates to the appropriate algorithm.

Algorithm 7 shows the Algorithm for a time instant represented with a log. In lines 1–2, it computes the maximum and minimum values of the processed quadbox, in order to check, in line 3, if those values are completely within the queried range. In that case, the part of the quadbox overlapping the queried region is reported as part of the solution without checking the individual values of the cells. On the contrary, the *if* in line 6 checks whether the maximum and minimum values of the processed quadbox are completely outside of the queried range, and in that case, the Algorithm can safely discard the entire quadbox, again without accessing the individual cells of the quadbox.

If the flow reaches line 8, this means that some of the cells of the processed quadbox meet the queried range, while others do not.

As in previous algorithms, $z_s = -1$ means that the traversal reached a leaf in the snapshot, and a $z_t = -1$ means the same but for a time instant. lines 9 and 10 compute the positions in T_s and T_t of the first child of the processed quadbox in both the snapshot and the log, provided that the Algorithm has not reached a leaf in a previous step.

lines 11–14 cause the *i* and *j* variables iterate over the q-rows (*i*) and q-cols (*j*) that overlap the queried region. Recall that each combination of values of *i* and *j* corresponds to one of the child quadboxes of the processed node, which overlaps the queried region.

Line 17 sets $z_{s\prime}$ to point the position in T_s corresponding to the current processed child. Line 18 computes the maximum value of that quadbox, and line 19 obtains the minimum value, provided that the child is not an individual cell. lines 20–22 do the same for the time instant.

If the check of line 24 is true, that means that the processed child is a leaf in the snapshot, and thus this is signalled setting $z_{s\prime} = -1$, and the minimum value is set to the maximum value, as this always holds in leaves.

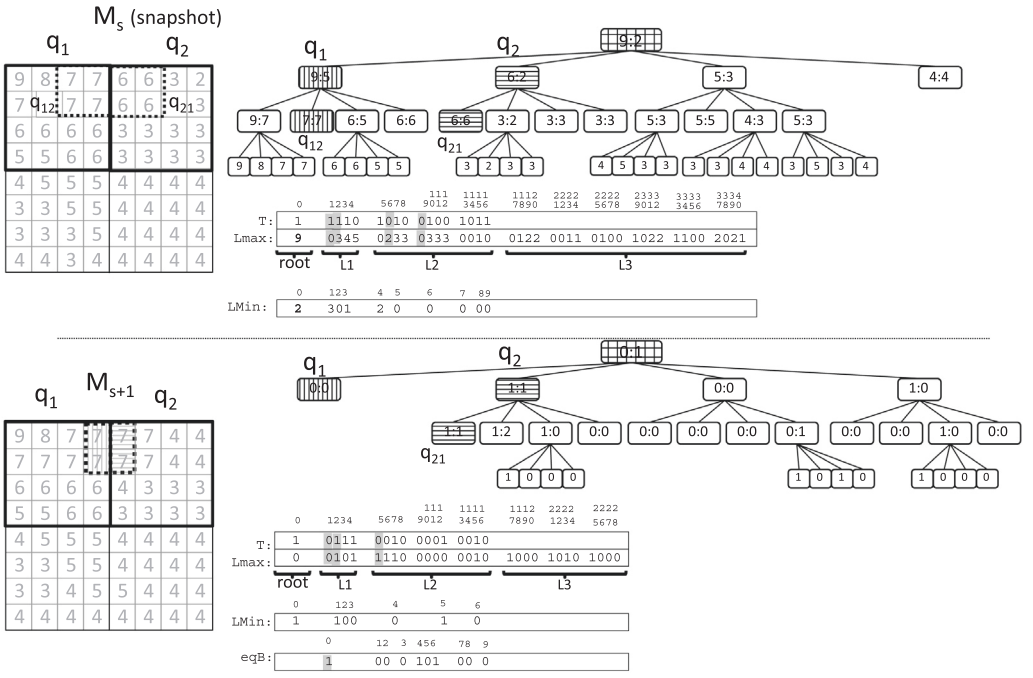

Fig. 10. Obtaining the highlighted region of M_{s+1}.

The check in line 27 is true when the traversal of the time instant reached a leaf. This is again signalled setting $z_{t'} = -1$. lines 28 and 29 compute the minimum value when the *eqB* bitmap signals a quadbox with all cells having the same value, whereas lines 30–31 do the same in case *eqB* bitmap signals a quadbox with the same structure as the snapshot. Finally, if the flow reaches line 33, a recursive call is issued for the processed child quadbox.

6. Heuristic $T - k^2raster$

$T - k^2raster$ is well suited for raster series where rasters representing close time instants do not differ much. Therefore, depending on the dataset, it may be more convenient to use the naive solution of using a normal $k^2raster$ per time instant. Moreover, it could be the case that in a dataset some time instants are better compressed with a normal $k^2raster$ and others are better compressed if we use a $T - k^2raster$ approach.

In this section, we present a heuristic version of $T - k^2raster$ that analyses the dataset and decides when it is appropriate to compress the raster corresponding to a time instant with a normal $k^2raster$ and when as a difference with respect to the last time instant represented with a normal $k^2raster$, or even if it could be more convenient to transform the previous time instant into a snapshot and then compress the current one as a log with respect to that new snapshot. Therefore there are not snapshots at regular time intervals, any time instant represented with a normal $k^2raster$ is considered as a snapshot. Therefore, this method requires a new bitmap *sB* that marks for each time instant which method was used to compress its data. This bitmap is also used to find the reference snapshot that was used to encode a particular log.

Next we present the construction method. Let us consider that, at a given step of the process, we are compressing the matrix M_t corresponding to the time instant t.

- *If sB[t-1]=1*, that is, if the previous time instant is represented with a snapshot. Then, the Algorithm simply compresses M_t using the two approaches: a normal $k^2raster$ and a $k^2raster$p using the compressed matrix of $t-1$ as snapshot. Considering the compressed version corresponding to M_t using $k^2raster$ and that using $k^2raster$p, the Algorithm uses the version that requires less space to represent M_t.
- *If sB[t-1]=0*, that is, if the previous time instant is not represented with a snapshot. Let s be the closest time instant to t represented with a snapshot. The process chooses the smallest configuration among these three:
  1. Use a normal snapshot.
  2. Represent t using a $k^2raster$p with respect to the snapshot at s.
  3. Change the representation of $t-1$ to use a snapshot, and then represent t as $k^2raster$p with respect to the new snapshot at $t-1$. In this case, to take a decision, we consider the space needed for storing the data of $t-1$ and t.

The rest of the operations are the same as in the case of the normal $T-k^2raster$ except that to query a given time instant, the Algorithm has to check whether t is represented with a snapshot or with a log. For this, the process simply accesses $sB[t]$. If it is represented as a log, then it has to find the last snapshot before t, which is simply obtained as $select_1(sB, rank_1(sB, t))$.

7. Experiments

7.1. Experimental framework

In this section, we present experimental results to show the space requirements and processing time of our proposed data structure. For processing time, we evaluate the three types of queries defined in Section 4.2.

We display results of two main variants of our proposal. The first one, $T-k^2raster$, is our basic representation, placing snapshots at regular time intervals. Recall that, for normal rasters, there are two variants of the $k^2raster$. In Section 3.2.1, we described the normal one, but we also mentioned an improvement termed *heuristic* $k^2raster$. The $T-k^2raster$ uses the normal $k^2raster$.

The second variant, *htkdosr*, is the heuristic version described in Section 6 that is able to adaptively select whether to create a snapshot or a log at each time instant. In addition, instead of using the normal $k^2raster$, this variant is built using the more efficient *heuristic* $k^2raster$ implementation.

We compare our proposals with three baseline implementations. The first one, *NetCDF*, is a classical method to store raster data with the possibility of using Deflate compression over the data. The *NetCDF* library[9] (v.4.7.4) was used to implement the query operations. The second baseline is a naive solution based on the *heuristic* $k^2raster$[10], denoted $k^2raster$h. This approach stores a full snapshot for each time instant. The third baseline is the 4D3D-mapping described in Section 3.3.1, that maps 2-dimensional space into a single dimension and uses a k^3tree to store the complete temporal raster.

For *NetCDF*, ten different levels of compression can be configured, where Level 0 means no compression and Level 9 obtains the best compression but with higher cost of access time. In these experiments, we build a three-dimensional raster with a level 2 of compression, which obtains a good trade-off between space and access time. For the 4D3D-mapping, we use the implementation with default configuration parameters as provided by the authors. For all the implementations based on the $k^2raster$, including the baseline and our proposals, we use a hybrid configuration, which uses a value of k ($k_1 = 4$) for the first 4 levels of the tree, and a different k ($k_2 = 2$) for the rest of levels. In $k^2raster$h and *htkdosr* we use $k_{Lst} = 4$, i.e. the heuristic $k^2raster$ replaces the leaves of size 4×4 by pointers to a dictionary. For $T-k^2raster$ we use $t_\delta = 6$, that is, a snapshot is built every 6 instants. These structures were implemented with the SDSL[11] library [16] and C++.

All the experiments were run on a dedicated Intel® Core™ i7-3820 CPU @ 3.60 GHz (4 cores) with cache sizes 32 KB (L1), 256 KB (L2), and 10 MB (L3), and 64 GB of RAM. The operating system was Debian 9.12 with kernel 4.9.0–9-amd64. The code was compiled with gcc version 6.3.0 with `-O3`.

7.2. Datasets

We use real datasets obtained from the following sources:

- NWC collection. This collection contains temperatures from the data provided by the National Weather Service [33]. Among other products, this service provides hourly information of temperatures for Guam (Gurtma) and Hawaii (Hirtma). As these data are just available for 48 h, we downloaded them during four months from September to December 2017. These time series have a space resolution of 2.5 km and their temporal resolution is blocks of 3 h for Gurtma and hourly for Hirtma. The temperatures have been converted to integers by truncating floating point figures.
- NLDAS-2 collection. This collection was generated by the North American Land Data Assimilation System (NLDAS). More concretely, we used data from the NLDAS_FORA0125_H dataset [46] that contains information about the precipitation and fluxes of North America from 1979 to present, e.g., surface temperature, humidity, and radiation. These time series have a space resolution of 1/8 degrees and their temporal resolution is hourly. Data used in these experiments correspond from January to December 2018 for four variables: fraction of total precipitation that is convective (*CONVAPCP*), precipitation hourly total [kg/m^2] (*APCP*), potential evaporation hourly total [kg/m^2] (*PEVAP*), and 2-m above ground Temperature [K] (*TMP*).[12] These variables have been converted to integers considering 2 decimal digits multiplying each value by 100.

[9] https://www.unidata.ucar.edu/software/netcdf/
[10] https://lbd.udc.es/research/k2-raster/
[11] https://github.com/simongog/sdsl-lite

Table 1 shows more details about these datasets. We display the spatial dimensions and the number of time instants included in each dataset, as well as the number of different values stored in the raster dataset. The last column displays the change rate of each dataset, computed as the average proportion of cells that change their value between consecutive time instants. Note that the selected datasets cover a variety of cases, since they have very different number of unique values (from 18 in *agurtma* to almost 7,000 in *TMP*) and change rate (from 0.02 in *CONVAPCP* to 0.77 in *TMP*).

7.3. Space requirements

Table 2 shows the space requirements for datasets in the NWC (top) and NLDAS-2 (bottom) collections, for all the representations. In the NWC collection, *NetCDF* and especially the 4D3D-mapping (4D3D) require significantly more space than all the alternatives based on the k^2*raster*. In the *agurtma* dataset, $T - k^2$*raster* obtains the best compression, since it is able to efficiently exploit temporal regularities; the reduced size of the dataset and the small number of unique values make it difficult for the *htkdosr* to take advantage of its enhanced compression. In the *hirtma* dataset, the baseline k^2*raster*h and our proposal *htkdosr* obtain the best results. This result shows that the best strategy in this dataset is essentially to build a snapshot per time instant, since our adaptive encoding of logs is unable to improve on the baseline. However, the differences among proposals based on the k^2*raster* are small in both datasets.

The datasets in the NWC collection are relatively small, and contain a very small number of unique values, which leads to some unexpected results that do not appear in larger datasets. Particularly, the poor compression of *NetCDF* is an outlier that only occurs in these datasets. However, the comparison between 4D3D and the solutions based on k^2*raster* is fair even in smaller datasets, since all the implementations are based on similar compact data structures. Our results show that solutions based on k^2*raster* variants are more efficient in general, even if 4D3D is reasonably competitive in *agurtma*.

The last four rows in Table 2 display the results obtained for the NLDAS-2 collection. Note that results for 4D3D are omitted for two of the datasets, because we were unable to build the data structure. The reason of these crashes is that the size of the k^3*tree BC* of the 4D3D drastically increases with the size of the matrices and the number of different values. The k^3*tree* construction requires large amounts of memory, and thus this easily produces a crash in the construction process. However, observe that *PEVAP* crashed while *APCP* did not, having both the same size, although, *APCP* has much more different values. The explanation is the distribution of the 1s in the *BC* matrix. The k^3*tree* compresses well the cubes full of 0s as such regions are represented with only one bit. However, as soon as a cube has at least one 1, then it has to be decomposed and then more space is required. Therefore the 1s are more clustered in the case of *APCP* than in the case of *PEVAP*, where the 1s are scattered, and hence, the memory consumption of the construction process of the k^3*tree BC* causes the crash.

In this collection, as expected, *NetCDF* obtains the best compression in all cases. Among compact data structures, we find different results depending on the dataset. In *APCP*, that has a very low change rate, 4D3D obtains the best compression, but our proposal *htkdosr* is relatively close. In the remaining datasets, *htkdosr* is the smallest representation: it is 40% smaller than the baseline k^2*raster*h in *CONVAPCP*, 65% smaller in *PEVAP* and less than 1% smaller in *TMP*. The similar compression in *TMP* is due to the high change rate in this dataset: in rapidly-evolving time series, the *htkdosr* will contain only snapshots and behave as the k^2*raster*h. Therefore, this result shows the effectiveness of the proposed heuristic for the selection of snapshots and logs, that is able to improve compression even in datasets with a high change rate. On the other hand, the regular sampling used in the $T - k^2$*raster* leads to less consistent results: it requires more space than *htkdosr*, and even than the k^2*raster*h baseline, in *TMP* and *PEVAP*, but it is still competitive in *CONVAPCP* and *APCP*.

These experiments demonstrate that our solutions can deal with datasets of different nature, by maintaining good compression ratios. Additionally, our results show the consistency achieved by *htkdosr*, that is either the smallest solution based on compact data structures or very close to it. The number of unique values in the dataset has some effect on compression, with datasets with a higher number of unique values being larger in *htkdosr*, but this effect is limited thanks to the ability to deferentially encode values in the k^2*raster*; for instance, the k^2*raster*h and $T - k^2$*raster* representations are larger for *APCP* than for *PEVAP*, even though the latter has 10 times more unique values. The change rate of the dataset seems to have more impact on space results, since the compression results are significantly worse in *TMP*, that has much higher change rate. However, this is also not consistent in all the datasets, as the representations are able to take advantage of many regularities in the data.

The comparison of our proposals with 4D3D, albeit an incomplete one due to the inability to build 4D3D for all the datasets, suggests that *htkdosr* is more compact, or at least achieves comparable compression, in most real-world datasets. 4D3D improves our results only in *APCP*, and by a relatively small margin. Finally, in *agurtma*, *hirtma* and *TMP*, *htkdosr* is unable to significantly improve the compression of the k^2*raster*h baseline, since it is forced to use snapshots at most time instants. In the following sections we will complement this analysis by focusing on query times, to obtain the space–time trade-off provided by all the implementations.

[12] We present these datasets as a representative subset of this collection. We performed the same experimental evaluation on the remaining datasets of the collection, obtaining similar comparison results to those included in the paper.

Table 1
Details of NWC (top) and NLDAS-2 (bottom) collections.

Name	#rows	#cols	#times	Values			Change
				min	max	#unique	rate
gurtma	193	193	936	19	36	18	0.21
hirtma	225	321	2806	−2	37	40	0.24
CONVAPCP	464	224	2,665	−1	100	101	0.02
APCP	464	224	2,665	−1	10,258	3,267	0.11
PEVAP	464	224	2,665	−79	185	262	0.20
TMP	464	224	2,665	23,421	31,223	6,889	0.77

Table 2
Space requirements (in Megabytes) over NWC (top) and NLDAS-2 (bottom) collections.

Name	NetCDF	4D3D	$k^2raster$h	$T-k^2raster$	htkdosr
gurtma	3,1	4,3	2,4	**2,0**	2,4
hirtma	34	61	**24**	26	**24**
CONVAPCP	**15**	50	50	37	30
APCP	**42**	85	127	112	95
PEVAP	**63**	–	192	233	65
TMP	**246**	–	423	471	421

7.4. Query times

In this section we present the results of the experiments for the queries described in Section 4.2. First, we analyse *access queries, that ask for a cell in a single raster of the time series, and therefore provide a rough estimation of the cost of accessing random elements in the representation. Then, we study windowQuery that involves cells in a spatial window and a time interval, and finally rangeQuery that also queries for a range of valid values. These last two queries are expected to be the most usual real-world queries and provide a more realistic estimation of the actual space–time trade-off achieved by each implementation.*

7.4.1. Times of access

Fig. 11 shows the space/time trade-off for the *access* operation, that obtains the value of a single cell at a specific time instant. In these experiments, we build a set of 100,000 random queries for each dataset. We run the full query set for each dataset 10 times and average the results in microseconds per query. Even though the most compact solution may vary among datasets, query times follow the same pattern in all our tests. *NetCDF* is very inefficient, roughly 1,000 times slower than the other techniques; this is expected, since it is not designed for efficient direct access to individual positions. 4D3D is also very inefficient in the first four datasets, as it is 4–10 times slower than *htkdosr* and still requires more space in most of them. Hence, even if *NetCDF* and 4D3D are not always dominated by other solutions, they are clearly ill-suited for this type of queries.

Focusing on the solutions based on the $k^2raster$, query times are very similar in all the datasets. In the smaller datasets of the NWC collection (Fig. 11a and 11b) query times are well below 1 μs/query, and in the remaining datasets they average around 1 μs/query. Particularly, both $T-k^2raster$ and *htkdosr* obtain query times similar to those of $k^2raster$h in all the datasets. This shows that the overhead of our *getCell* Algorithm is small in general when compared with the equivalent operation in a single heuristic $k^2raster$. In *CONVAPCP*, *htkdosr* is even 20% faster than $k^2raster$h. Several factors contribute to explain this improvement. First, note that the *getCell* algorithm, in some specific cases, is able to avoid the full traversal of the snapshot. Additionally, the traversal of logs should be much faster than the traversal of snapshots, since they are significantly smaller on average. Finally, in *htkdosr* the snapshots will be more frequently accessed, and logs are much smaller; this improves the locality of access, significantly reducing the number of cache misses, at least in the lower levels of the cache.

For datasets *agurtma*, *hirtma* and *TMP*, as explained in the previous section, the *htkdosr* is forced to use snapshots for most time instants, and obtains compression very similar to $k^2raster$h. This also means that the direct comparison between *htkdosr* and $k^2raster$h in these datasets provides a rough estimation of the overhead in query times required by *htkdosr*, where we need to query a bitmap to determine whether the current time instant is a log or a snapshot, and to locate the previous snapshot if necessary. This additional cost is shown to be negligible in practice in our experiments, considering that *access* operations require the simplest traversal of the $k^2raster$ structures. Therefore, in the following sections, we will not further discuss the effect of this additional bitmap, since the cost of the operations will be completely dominated by the traversal of the $k^2raster$ structures.

7.4.2. Times of windowQuery

Next, we analyse the performance of the more complex *windowQuery* operation, that recovers the values of all cells in a cuboid defined by a spatial window and a time interval. We run a set of 100 queries per dataset, selecting the corners of the

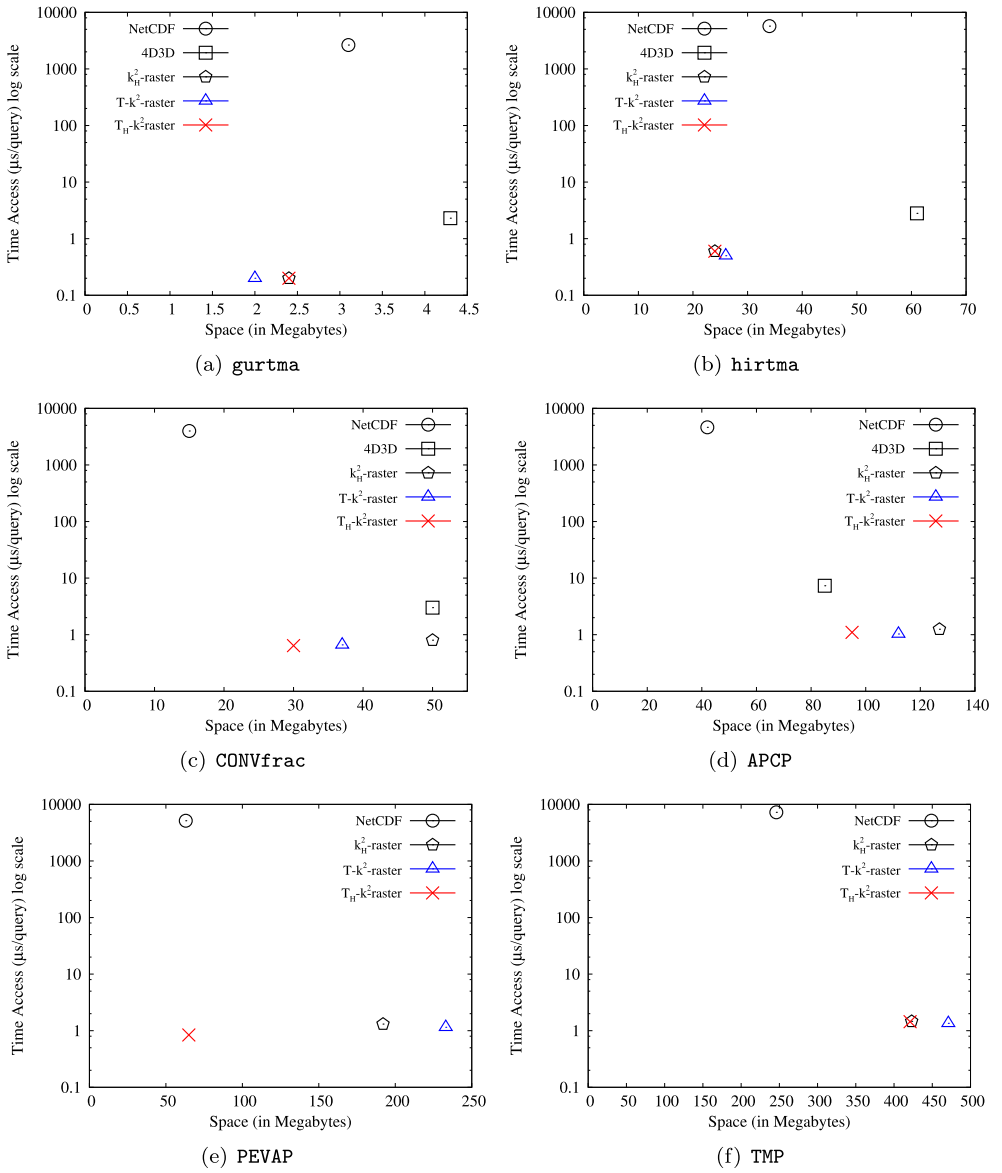

Fig. 11. Time results for *access* over NWC (top) and NLDAS-2 collections. We show the average time per query (in microseconds).

spatial window and the limits of the time interval at random, in order to cover different window sizes and time interval lengths.

Fig. 12 displays the results obtained for all the datasets. In the *agurtma* and *hirtma* datasets, *NetCDF* and 4D3D are 5–10 times slower than any of the other alternatives. In the remaining datasets, 4D3D is still much slower than the other implementations, but *NetCDF* and k^2*raster*h are much more competitive in *windowQuery* than in the simpler *access* operation. *NetCDF* is still slower than *htkdosr* in all the datasets, ranging from 10% extra time to being roughly two times slower. On the other hand, k^2*raster*h is faster than our solutions in most of the datasets, since it has the simplest access operations. Unlike the *access* query, now the rags of the *htkdosr* no longer exceed the costs in most cases, as the access to a single tree structure per time instant of k^2*raster*h is the most decisive factor. The only exception is *agurtma*, where our implementation is still slightly faster.

Focusing on our proposals, $T - k^2$*raster* is much slower than *htkdosr* in general. This difference in performance is relatively small in the datasets of the NWC collection, but much more significant in the NLDAS-2 collection, where $T - k^2$*raster* is 3–5 times slower depending on the dataset. Two factors contribute to this difference: the first one is the adaptive creation of snapshots in *htkdosr*, that yields smaller space and more balanced logs, in which a higher or lower number of snapshots

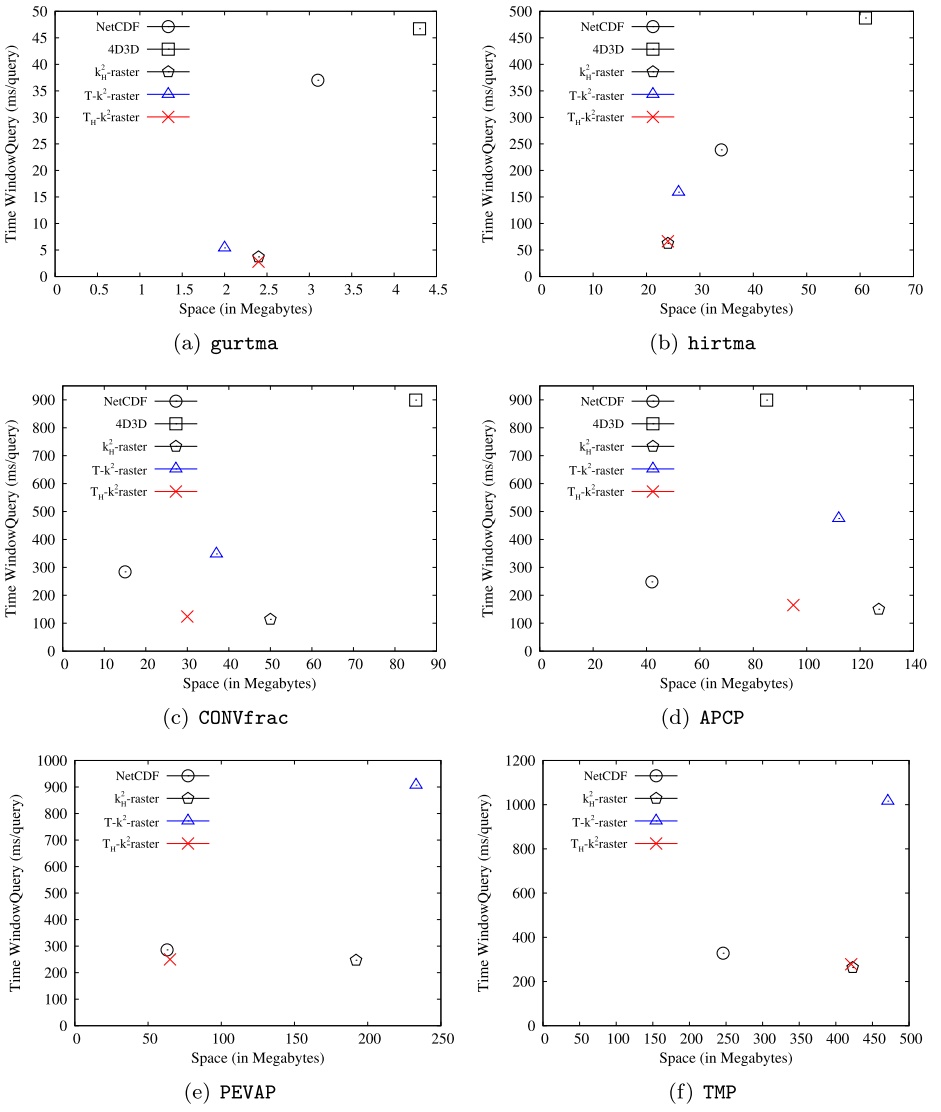

Fig. 12. Time results for *windowQuery* over NWC (top) and NLDAS-2 collections. We show the average time per query (in milliseconds).

may be used if necessary; the second main factor is the difference in the underlying data structure used in each representation: *htkdosr* is built using heuristic k^2*raster*, that is usually smaller but also faster, whereas $T-k^2$*raster* uses the basic k^2*raster*.

Notice that both of our techniques need to perform a synchronised traversal of a k^2*raster* and a k^2*raster*p for all time instants that are not stored as snapshots. This synchronised traversal is expected to be slower than a single-structure traversal. However, our results show that, in practice, *htkdosr* obtains query times very close to those of k^2*raster*h. These results show, firstly, that the cost of synchronised traversal is not too high, and more importantly, that the adaptive selection of snapshots provides enough compensation for the *htkdosr* to be competitive in query times even in complex operations, such as *windowQuery*, where the naive baseline could be expected to be significantly faster.

7.4.3. Times of rangeQuery

In this section, we analyse the performance of all the methods for the *rangeQuery* operation, which adds an additional constraint on the range of values to the previous *windowQuery*. Therefore, this query retrieves cells within the cuboid determined by given spatial and temporal bounds, and only returns the cells in the cuboid whose values fall in a predetermined range of values. We run sets of 100 queries for each dataset, selecting the corners of the spatial window, limits of the time interval and limits of the range of values uniformly at random.

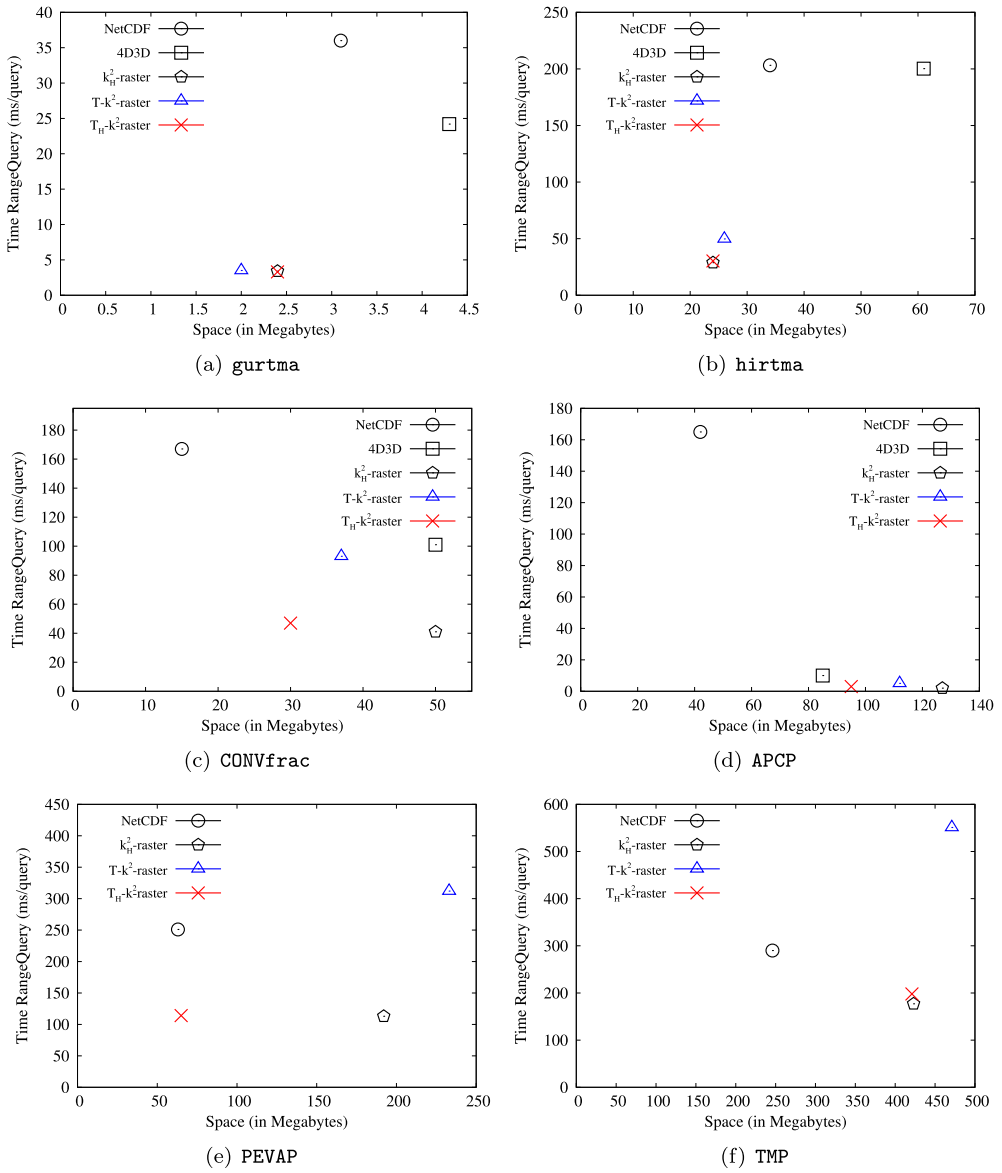

Fig. 13. Time results for *rangeQuery* over NWC (top) and NLDAS-2 collections. We show the average time per query (in milliseconds).

Fig. 13 displays the space/time trade-off of all implementations. The baselines *NetCDF* and 4D3D are very inefficient in general, the first being 1.5–40 times slower than *htkdosr* and k^2*raster*h depending on the dataset, and the latter being 2–10 times slower in the datasets where it could be built. This confirms the results of previous sections regarding 4D3D, that is unable to compete in compression or query performance with our solutions. On the other hand, the difference in performance with *NetCDF* is expected in this query, since our solutions can easily handle filters on ranges of values, whereas *NetCDF* is forced to retrieve and sequentially process large portions of the raster to identify regions of interest. The effect of this additional filter on ranges of values can also be confirmed by comparing the query times obtained by all the implementations in Fig. 13 (*rangeQuery* operation) with those of Fig. 12 (*windowQuery* operation): average query times in *NetCDF* are similar for both operations, whereas the k^2*raster*h baseline and our proposals become up to 5–6 times faster in the *rangeQuery* thanks to being able to efficiently filter out portions of the datasets based on their value.

As in previous experiments, our improved proposal, *htkdosr*, obtains slightly slower query times than the k^2*raster*h baseline on average, but is very competitive and much faster than $T - k^2$*raster*. Even though *htkdosr* is slightly slower than the k^2*raster*h baseline, this difference in query times is below 10% in the worst case, whereas the difference in compression is very relevant in most datasets. The ranking of these three implementations is similar to the one obtained in the previous

section, for the *windowQuery* operation, but the differences between them are reduced due to the ability of the three implementations to filter out regions of the spatial window that do not match the expected range of values. Note also that our basic proposal, $T-k^2raster$, is very inefficient in the *TMP* and *PEVAP* datasets, where it is larger and slower than the *NetCDF* baseline. Again, these results are consistent with those obtained in the previous section for the *windowQuery* operation, and displayed in Fig. 12: the basic $T-k^2raster$, taking snapshots at regular intervals, is not able to improve compression in all cases when compared to the baselines, and the *htkdosr* can obtain a 3x–5x speedup thanks to its adaptive selection of snapshots combined with the use of the heuristic $k^2raster$.

The overall results of our experimental evaluation show that *htkdosr* obtains the most consistent results and a reasonable space–time trade-off in all the test datasets. *NetCDF* is 40–50% smaller in some datasets, but much slower on average. Alternatives like 4D3D also seem to slightly improve our compression in some datasets, but at the cost of higher query times. Finally, our experiments show that *htkdosr* is able to reduce the space of the naive $k^2raster$h baseline by up to 60% depending on the dataset. Moreover, the overhead in query times required by the additional data structure is small enough to be practical even in datasets with high change rate where our proposal is forced to use mostly snapshots.

8. Conclusions and Future Work

We have presented a new solution for the efficient representation of raster time series. Our solution extends existing compact data structures for the representation of raster data, following a strategy of snapshots and logs to take advantage of temporal regularities that arise in many real-world raster datasets. We presented two variants of our solution, a basic implementation $T-k^2raster$ that uses regular-interval sampling, and an improved variant *htkdosr* that is built on an improved *heuristic* $k^2raster$ and uses a heuristic to improve compression, by sampling at irregular intervals that are adaptively selected to improve compression.

We experimentally evaluate our proposal in multiple real raster time series, and compare it with state-of-the-art solutions to show its performance. Our results show that *htkdosr* achieves very good compression, outperforming other techniques based on compact data structures. Our tests include very selective cell-retrieval queries and different window selection queries, and our results confirm that *htkdosr* is in all cases the fastest representation, or very close to it. The only representation that improves our compression in general is *NetCDF*, but it is also much slower in all queries: up to 10,000 times slower in *access* queries, up to 30 times slower in *rangeQuery* queries and up to 10 times slower in *windowQuery* queries, that are the most favorable for their compression scheme. Other solutions based on compact data structures are not competitive with our proposal, which yields the best space–time trade-off across all datasets.

As future work, we plan to explore potential improvements of the heuristic used for the selection of snapshots, in order to provide additional guarantees on compression without significantly affecting construction time. It may also be interesting to study metrics of the regularities existing in raster time series that allow us to better understand the differences in compression ratio between datasets. The Max-Model for 2-dimensional images recently introduced by [1] may be useful for such purpose, but it should be extended to also capture temporal locality, in addition to spatial locality.

CRediT authorship contribution statement

Fernando Silva-Coira: Conceptualization, Software, Resources, Investigation, Writing - original draft. **José R. Paramá:** Conceptualization, Writing - original draft, Writing - review & editing. **Guillermo Bernardo:** Conceptualization, Software, Writing - original draft, Writing - review & editing. **Diego Seco:** Conceptualization, Resources, Writing - original draft, Writing - review & editing.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgement

The data used in this study were acquired as part of the mission of NASA's Earth Science Division and archived and distributed by the Goddard Earth Sciences (GES) Data and Information Services Center (DISC). Funding: CITIC, as Research Center accredited by Galician University System, is funded by "Consellería de Cultura, Educación e Universidade from Xunta de Galicia", supported in an 80% through ERDF Funds, ERDF Operational Programme Galicia 2014-2020, and the remaining 20% by "Secretaría Xeral de Universidades" (Grant ED431G 2019/01). This work was also supported by Xunta de Galicia/FEDER-UE under Grants [IG240.2020.1.185; IN852A 2018/14]; Ministerio de Ciencia, Innovación y Universidades under Grants [TIN2016-78011-C4-1-R; RTC-2017-5908-7; PID2019- 105221RB-C41/AEI/10.13039/501100011033]; ANID - Millennium Science Initiative Program - Code ICN17_002; Programa Iberoamericano de Ciencia y Tecnología para el Desarrollo (CYTED) [Grant No. 519RT0579].

References

[1] A. Abdollahi, N.D.B. Bruce, S. Kamali, R. Karim, Lossless image compression using list update algorithms, in: International Symposium on String Processing and Information Retrieval (SPIRE), 2019, pp. 16–34.
[2] S. Araya, B. Ostendorf, G. Lyle, M. Lewis, Cropphenology: An r package for extracting crop phenology from time series remotely sensed vegetation index imagery, Ecol. Inform. 46 (2018) 45–56.
[3] P. Baumann, P. Mazzetti, J. Ungar, R. Barbera, D. Barboni, A. Beccati, L. Bigagli, E. Boldrini, R. Bruno, A. Calanducci, et al, Big data analytics for earth sciences: the earthserver approach, Int. J. Digital Earth 9 (2016) 3–29.
[4] E.A. Becker, K.A. Forney, P.C. Fiedler, J. Barlow, S.J. Chivers, C.A. Edwards, A.M. Moore, J.V. Redfern, Moving towards dynamic ocean management: How well do modeled ocean products predict species distributions?, Remote Sensing 8 (2016) 149
[5] J.A. Benediktsson, J. Chanussot, W.M. Moon, et al, Advances in very-high-resolution remote sensing, Proc. IEEE 101 (2013) 566–569.
[6] N.R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, G. Navarro, Óscar Pedreira, Extending general compact querieable representations to gis applications, Inf. Sci. 506 (2020) 196–216.
[7] N.R. Brisaboa, A. Gomez Brandon, G. Navarro, J.R. Paramá, GraCT: A Grammar-based Compressed Index for Trajectory Data, Inf. Sci. 483 (2019) 106–135.
[8] N.R. Brisaboa, S. Ladra, G. Navarro, Dacs: Bringing direct access to variable-length codes, Inf. Process. Manage. (2013) 392–404.
[9] N.R. Brisaboa, S. Ladra, G. Navarro, Compact representation of web graphs with extended functionality, Inform. Syst. (2014) 152–174.
[10] A. Cerdeira-Pena, G. de Bernardo, A. Fariña, J.R. Paramá, F. Silva-Coira, Towards a compact representation of temporal rasters, in: International Symposium on String Processing and Information Retrieval (SPIRE), Springer, 2018, pp. 117–130.
[11] N. Cruces, D. Seco, G. Guitérrez, A compact representation of raster time series, in, in: Data Compression Conference (DCC), IEEE, 2019, pp. 103–111.
[12] L.P. Deutsch, RFC 1951: DEFLATE compressed data format specification version 1.3, 1996..
[13] P.N. Edwards, A vast machine: Computer models, climate data, and the politics of global warming, Mit Press, 2010.
[14] D.L. Gall, MPEG: A video compression standard for multimedia applications, Commun. ACM 34 (1991) 47–58.
[15] A. Gholizadeh, M. Saberioon, E. Ben-Dor, L. Boruvka, Monitoring of selected soil contaminants using proximal and remote sensing techniques: Background, state-of-the-art and future perspectives, Critical Reviews Environ. Sci. Technol. 48 (2018) 243–278.
[16] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: Plug and play with succinct data structures, in: International Symposium on Experimental Algorithms (SEA), 2014, pp. 326–337.
[17] R. González, S. Grabowski, V. Mäkinen, G. Navarro, Practical implementation of rank and select queries, in: Poster Proc. of 4th Workshop on Efficient and Experimental Algorithms (WEA) Greece, 2005, pp. 27–38.
[18] M. Hirschmugl, M. Steinegger, H. Gallaun, M. Schardt, Mapping forest degradation due to selective logging by means of time series analysis: Case studies in central africa, Remote Sensing 6 (2014) 756–775.
[19] G. Jacobson, Space-efficient static trees and graphs, in: 30th Annual Symposium on Foundations of Computer Science (FOCS), 1989, pp. 549–554.
[20] A. Klinger, Pattern and search statistics, Academic Press, 1971.
[21] A. Klinger, C.R. Dyer, Experiments on picture representation using regular decomposition, Comput. Graph. Image Process. 5 (1976) 68–105.
[22] W. Kou, Digital Image Compression: Algorithms and Standards, Kluwer Pub, 1995.
[23] S. Ladra, J.R. Paramá, F. Silva Coira, Compact and queryable representation of raster datasets, in: Proc. 28th SSDBM, 2016..
[24] S. Ladra, J.R. Paramá, F. Silva-Coira, Scalable and queryable compressed storage structure for raster data, Inform. Syst. 72 (2017) 179–204.
[25] C. Lee, M. Yang, R. Aydt, NetCDF-4 Performance Report Technical Report, HDF Group, 2008.
[26] Y. Li, T.R. Bretschneider, Semantic-Sensitive Satellite Image Retrieval, IEEE Trans. Geosci. Remote Sens. 45 (2007) 853–860.
[27] J. Mennis, R. Viger, C.D. Tomlin, Cubic map algebra functions for spatio-temporal analysis, Cartography Geographic Inform. Sci. 32 (2005) 17–32.
[28] J.L. Mennis, Multidimensional map algebra: Design and implementation of a spatio-temporal GIS processing language, Trans. GIS 14 (2010) 1–21.
[29] G.M. Morton, A Computer-oriented Geodetic Data Base and a New Technique in File Sequencing Technical Report, IBM Ltd., Ottawa, Canada, 1966.
[30] J.I. Munro, Tables, Proceedings of Foundations of Software Technology and Theoretical Computer, Science (1996) 37–42.
[31] G. Navarro, Compact Data Structures: A Practical Approach, Cambridge University Press, 2016.
[32] M.a. Oliver, Operations on Quadtree Encoded Images, Comput. J. 26 (1983) 83–91, https://doi.org/10.1093/comjnl/26.1.83.
[33] N.C. Operations, Real-time mesoscale analysis (RTMA) products, 2017..
[34] F. Petitjean, P. Gançarski, F. Masseglia, G. Forestier, Analysing Satellite Image Time Series by Means of Pattern Mining, Springer, Berlin Heidelberg, 2010, pp. 45–52.
[35] A. Pinto, D. Seco, G. Gutiérrez, Improved queryable representations of rasters, in: Data Compression Conference (DCC), 2017, pp. 320–329.
[36] G. Proietti, An optimal Algorithm for decomposing a window into maximal quadtree blocks, Acta Informatica 36 (1999) 257–266.
[37] M.G. Quartulli, I. Olaizola, A review of EO image information mining. ISPRS J. Photogramm. Remote Sens 75, 11–28, 2013..
[38] D. Salomon, Data Compression: The Complete Reference, Springer, 2006.
[39] H. Samet, The Quadtree and Related Hierarchical Data Structures, ACM Comput. Surv. 16 (1984) 187–260.
[40] H. Samet, Data structures for quadtree approximation and compression, Commun. ACM 28 (1985) 973–993.
[41] H. Samet, Foundations of multidimensional and metric data structures, Morgan Kaufmann, 2006.
[42] Y. Tao, D. Papadias, MV3R-tree: A spatio-temporal access method for timestamp and interval queries, in: Proc. 27th International Conference on Very Large Data Bases (VLDB), 2001, pp. 431–440.
[43] C. Tomlin, J.K. Berry, Mathematical structure for cartographic modeling in environmental analysis, in, in: Proceedings of the American Congress on Surveying and Mapping annual meeting, 1979.
[44] Y.H. Tsai, K.L. Chung, W.Y. Chen, A strip-splitting-based optimal Algorithm for decomposing a query window into maximal quadtree blocks, IEEE Trans. Knowl. Data Eng. 16 (2004) 519–523.
[45] G.K. Wallace, The jpeg still picture compression standard, Commun. ACM 34 (1991) 30–44.
[46] Y. Xia, K. Mitchell, M. Ek, J. Sheffield, B. Cosgrove, E. Wood, L. Luo, C. Alonge, H. Wei, J. Meng, et al., (accessed June 17, 2020). Nldas primary fosrcing data l4 hourly 0.125× 0.125 degree v002. Goddard Earth Sciences Data and Information Services Center (GES DISC), Greenbelt, MD, USA, Rep. NASA/GSFC/HSL https://doi.org/10.1029/2011JD016048, 2009..
[47] C. Yang, M. Goodchild, Q. Huang, D. Nebert, R. Raskin, Y. Xu, M. Bambacus, D. Fay, Spatial cloud computing: how can the geospatial sciences use and help shape cloud computing?, Int. J. Digital Earth 4 (2011) 305–329.
[48] C. Yang, Q. Huang, Z. Li, K. Liu, F. Hu, Big data and cloud computing: innovation opportunities and challenges, Int. J. Digital Earth 10 (2017) 13–53.