Information Systems 72 (2017) 179–204

Contents lists available at ScienceDirect

Information Systems

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

Scalable and queryable compressed storage structure for raster data☆

Susana Ladra, José R. Paramá, Fernando Silva-Coira*

Universidade da Coruña, Facultade de Informática, CITIC, Campus de Elviña s/n, 15071 A Coruña, Spain

ARTICLE INFO

Article history:
Received 15 December 2016
Revised 21 July 2017
Accepted 15 October 2017
Available online 18 October 2017

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

ABSTRACT

Compact data structures are storage structures that combine a compressed representation of the data and the access mechanisms for retrieving individual data without the need of decompressing from the beginning. The target is to be able to keep the data always compressed, even in main memory, given that the data can be processed directly in that form. With this approach, we obtain several benefits: we can load larger datasets in main memory, we can make a better usage of the memory hierarchy, and we can obtain bandwidth savings in a distributed computational scenario, without wasting time in compressing and decompressing data during data exchanges.

In this work, we follow a compact data structure approach to design a storage structure for raster data, which is commonly used to represent attributes of the space (temperatures, pressure, elevation measures, etc.) in geographical information systems. As it is common in compact data structures, our new technique is not only able to store and directly access compressed data, but also indexes its content, thereby accelerating the execution of queries.

Previous compact data structures designed to store raster data work well when the raster dataset has few different values. Nevertheless, when the number of different values in the raster increases, their space consumption and search performance degrade. Our experiments show that our storage structure improves previous approaches in all aspects, especially when the number of different values is large, which is critical when applying over real datasets. Compared with classical methods for storing rasters, namely netCDF, our method competes in space and excels in access and query times.

1. Introduction

Geographical information systems can use different data models to manage spatial information [2]. At the conceptual level, there are two possibilities: *object-based models* and *field-based models* [3]. Object-based models consider a space containing discrete and identifiable entities, each with a geospatial position. In contrast, field-based models can be seen as a continuous mathematical function that for each position of the space returns a value. Typically, object-based models represent spaces containing buildings, roads, and other man-made objects. On the other hand, field-based models usually deal with images and physical properties such as land elevation, temperature, atmospheric pressure, etc. At the logical level, there are also two models: *vector models*, which represent the spatial information using points and line segments, and *raster models*, which consider the space as a regular tessellation of disjoint cells, usually squares, each having a value [4]. Any logical spatial model can be used to represent any conceptual spatial model, however, it is common to use vector models to represent object-based models and raster models for field-based models.

This paper deals with spatial information represented with a raster model. This involves images -including remotely sensed imagery-, engineering, modeling, representations of parameters of the land surface such as pollution levels, atmospheric pressure, rain precipitations, land elevation, vegetation indices, etc. Thanks to the advances in remote sensing and instrumentation, the amount and size of raster datasets are increasing rapidly. For example, it has been estimated that each day, remotely-sensed imagery is acquired at the rate of several terabytes per day [5], and the archived amount of raster data of this type is slowly approaching the zettabyte scale [6].

In this field, as usual, compression has been used to save space and bandwidth [7–9]. Long-established compression methods do not allow to process or query compressed data, requiring a previous decompression phase. However, a recent family of storage structures, called *compact data structures*, is changing the way in which compression has been traditionally used. Compact data

☆ A preliminary partial version of this paper was published in [1].

* Corresponding author.
E-mail addresses: susana.ladra@udc.es (S. Ladra), jose.parama@udc.es (J.R. Paramá), fernando.silva@udc.es (F. Silva-Coira).

https://doi.org/10.1016/j.is.2017.10.007
0306-4379/

structures combine in a unique storage structure a compressed representation of a dataset and the mechanisms that allow accessing any given datum without the need of decompressing the data from the beginning [10,11]. The objective is to keep the data always compressed, even in main memory. In this way, in addition to the classical savings in disk space and bandwidth, we obtain several additional benefits: we can process larger datasets within the same memory, we can make a better usage of the memory hierarchy (including reducing costly disk accesses), and we can improve the performance when using parallel processing. Data interchanges between nodes in that scenario are a big issue since they can produce bottlenecks in the network. Compression has been used to reduce bandwidth consumption [12,13]. However, data have to be compressed prior any data exchange and decompressed at the destination node, given that data cannot be processed in compressed form. Nevertheless, using compact data structures, we do not need those compression/decompression processes during data exchanges thanks to the ability to process compressed data.

Another advantage is that, in many cases, compact data structures provide some sort of indexation, which allows answering queries even faster than performing that query over the plain representation and within the same compressed space [11,14,15]. That is, this indexation is not provided by an auxiliary structure, and the index plus the data, kept in the same storage structure, occupy less space than the original data. This characteristic is usually called *self-indexation*.

There exists vast research focused on compressing raster datasets, proposing both lossless [9,13,16,17] and lossy [7] approaches. In addition, there have been efforts in creating indexes on raster data to improve query and processing performance [18–20]. However, there is much less work on data structures capable of compressing and indexing data at the same time. The first exponent is the quadtree [21,22], originally designed as a method to compress images, which allows the manipulation of the compressed image directly in main memory and, in addition, it spatially indexes the values of the raster. However, it does not provide indexation over the values of each cell of the raster. To the best of the authors' knowledge, only two recent compact data structures [23] were designed to represent raster datasets and achieved these three features: a compressed representation, a spatial indexation, and an indexation of the values of the cells. These techniques work well when the number of different values in the raster is low; however, if that number grows, both the space consumption and the query performance degrade dramatically. Our new storage structure scales much better when increasing the size of the input data or when the raster matrix increases its cardinality, that is, when the number of different values grows. Observe that this is an important problem when dealing with rasters, since they are usually obtained from a real continuous phenomenon as temperature, atmospheric pressure, etc.

This work presents a new storage structure designed following a compact data structure approach to represent raster datasets, called k^2-*raster*, and an improved version that we call *heuristic* k^2-*raster* (k^2_H-*raster*). They are based on the k^2-tree [24], a storage structure for representing binary matrices in little space, which can be regarded as a compact version of a quadtree. The basic ideas of k^2-*raster* were already presented in a preliminary work [1]. Here, we described it more in detail, including pseudocodes and examples for more queries. The k^2_H-*raster*, which was not proposed in the original paper, is a variant that significantly improves the spatio-temporal results. We use an entropy-based heuristical approach to compress the last level of the representation, which makes our solution the most space-efficient and scalable compressed and queryable representation up to date.

In this paper, we also enhance the experiments, including more queries. We ran the experiments over new raster matrices extracted from real datasets of different nature (temperatures and elevations). The basic k^2-*raster* is superior to previous approaches in most cases, and only when the number of different values is low, it can be on a par in some parameters, whereas the heuristic variant overcomes them in all aspects, even in the scenario of low number of different values.

We also include a comparison with netCDF [25], a classical method to store rasters. NetCDF includes the possibility of compressing the data with Deflate [26], and by using a simple API, transparently accessing the compressed data. k^2-*raster* obtains compression ratios close to those achieved by netCDF, but differences are not significant. However, k^2-*raster* clearly outperforms netCDF in access and query times, even in some cases when using the uncompressed version of netCDF files. Thanks to the indexing capabilities of k^2-*raster*, queries specifying conditions on the values of the raster are solved orders of magnitude faster than over uncompressed netCDF files.

The rest of the paper is structured as follows. Section 2 presents some related work. Section 3 describes the k^2-*raster* in detail, whereas Section 4 presents the k^2_H-*raster*. Section 5 presents our experimental study. Finally, Section 6 shows the conclusions and future work.

2. Related work

2.1. Quadtress for raster representation or indexation

As we will see, the k^2-*raster* uses some ideas of the k^2-tree [24], which is a region quadtree for binary matrices built with the latest developments in the field of compact data structures. In particular, the k^2-*raster* uses the partition strategy of the region quadtree data structure. Thus, we present here some notions of quadtrees.

There are many different variants of the quadtree and with different purposes [18,27], but the compression of images using region quadtrees was one of its original targets [21,22]. In this scenario, the quadtree was designed as a representation of images not only for storage or transmission purposes but to process them directly in main memory [18]. To fit the structure in main memory, the size is a relevant issue, and thus since it is a tree, pointer-less representations were introduced [28,29]. These pointerless representations use a *locational code* that for each leaf of the tree gives its position in the space [28] or an implicit ordering [29]. For our work, it is of special interest the latter case, denoted as *Treecodes*. The region quadtree is represented by a sequence of numbers, each representing a node of the conceptual region quadtree. Each of these numbers has 5 bits, the most significant bit indicates whether the corresponding node is a leaf or not, and the remaining 4 bits store a value. In the case of a leaf node, that number is the value corresponding to a pixel of the image; in non-leaf nodes, it is the average value of the pixels contained in the region represented by such a node. This average value is used to give a preview of the image during a slow transmission through a network. The quadtree is stored as a sequence of bytes, each storing a 5-bit number, where the correspondence of each byte with the nodes of the conceptual tree is given by the ordering of the sequence, which is a *breadth-first* traversal of the tree. The representation of the image of Fig. 1 is the sequence of bytes: 20, 3, 18, 4, 7, 0, 1, 2, 5. The first 20 corresponds to the root node, which is an inner node signaled with a 1 in the fifth bit, the next 4 bits store the average value of all pixels in the image (4), and thus we have a 10100 (20). The third byte (18) corresponds to the quadrant further divided into subquadrants, therefore it represents an internal node (fifth bit set to one) and the next four bits store the average value (2).

Our work also uses an implicit ordering using a breadth-first traversal, but we separate the topology of the tree (the most sig-