Compact Data Structures for Large and Complex Datasets

Autor: Fernando Silva Coira

Tesis doctoral UDC / 2017

Directores:
Susana Ladra González
José Ramón Paramá Gabía

Compact Data Structures for Large and Complex Datasets

Autor: Fernando Silva Coira

Tesis doctoral UDC / 2017

Directores:
Susana Ladra González
José Ramón Paramá Gabía

Programa Oficial de Doutoramento en Computación

PhD thesis supervised by
Tesis doctoral dirigida por

Susana Ladra González
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1200
Fax: +34 981 167160
susana.ladra@udc.es

José Ramón Paramá Gabía
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1241
Fax: +34 981 167160
jose.parama@udc.es

Susana Ladra González y José Ramón Paramá Gabía, como directores, acreditamos que esta tesis cumple los requisitos para optar al título de doctor internacional y autorizamos su depósito y defensa por parte de Fernando Silva Coira cuya firma también se incluye.

A miña familia

Acknowledgements

I must start these lines by thanking my advisors, Susana and Jose, who supported me over the last few years to do this work, without them this would not be possible. Thanks also for all the knowledge transmitted and the help I had. I also want to thank Nieves for giving me the opportunity to join the research group.

Thanks to all members of the Database Laboratory, especially Alex, Cris, Adrian, Daniil and Tirso, learning (and traveling) partners. And the friends and family that accompanied me all this time. I should also thank Gonzalo for his help in my stay in Chile and for everything I learned during these months.

Finally, my greatest gratitude to Lorena, for being by my side from the first day and helping me when I needed it. Thank you for believing in me and being my traveling partner, I could not do all this way without you.

This thesis has received funding from the European Union's Horizon 2020 research and innovation programme under the Marie Sklodowska-Curie grant agreement No 690941; the European Regional Development Fund (ERDF) [ED431G/01]; Ministerio de Economía y Competitividad (PGE and ERDF) [TIN2016-78011-C4-1-R; TIN2016-77158-C4-3-R; TIN2013-46238-C4-3-R; TIN2013-46801-C4-3-R], Centro para el desarrollo Tecnológico e Industrial Programa CIEN 2014 (co-founded with ERDF) [IDI-20141259; ITC-20151247]; and Xunta de Galicia (co-founded with ERDF) [GRC2013/053].

Agradecimientos

Debo empezar estas líneas dando las gracias a mis directores de tesis, Susana y Jose, que me guiaron a lo largo de los últimos años para realizar este trabajo, sin ellos ésto no sería posible. Gracias también por todo el conocimiento transmitido y los consejos y la ayuda que tuve mientras investigábamos juntos. También quiero agradecer a Nieves por darme la oportunidad de unirme al grupo de investigación.

Gracias a todos los miembro del Laboratorio de Bases de Datos, especialmente a Álex, Cris, Adrián, Daniil y Tirso, compañeros de aprendizaje (y de viajes). Y a los amigos y familia que me acompañaron en todo momento. También debo agradecer a Gonzalo por la ayuda prestada en mi estancia en Chile y por todo lo que aprendí durante eses meses.

Por último, mi mayor gratitud a Lorena, por estar a mi lado desde el primer día y ayudarme cuando lo necesitaba. Gracias por creer en mí y ser mi compañera de viaje, no podría recorrer todo este camino sin ti.

Esta tesis ha recibido fondos del programa de investigación e innovación de European Union's Horizon 2020 en virtud del acuerdo de subvención Marie Sklodowska-Curie No 690941; el Fondo Europeo de Desarrollo Regional (FEDER) [ED431G/01]; Ministerio de Economía y Competitividad (PGE and FEDER) [TIN2016-78011-C4-1-R; TIN2016-77158-C4-3-R; TIN2013-46238-C4-3-R; TIN2013-46801-C4-3-R], Centro para el desarrollo Tecnológico e Industrial Programa CIEN 2014 (cofinancido con FEDER) [IDI-20141259; ITC-20151247]; y Xunta de Galicia (cofinanciado con FEDER) [GRC2013/053].

Abstract

In this thesis, we study the problem of processing large and complex collections of data, presenting new data structures and algorithms that allow us to efficiently store and analyze them. We focus on three main domains: processing of multidimensional data, representation of spatial information, and analysis of scientific data.

The common nexus is the use of compact data structures, which combine in a unique data structure a compressed representation of the data and the structures to access such data. The target is to be able to manage data directly in compressed form, and in this way, to keep data always compressed, even in main memory. With this, we obtain two benefits: we can manage larger datasets in main memory and we take advantage of a better usage of the memory hierarchy.

In the first part, we propose a compact data structure for multidimensional databases where the domains of each dimension are hierarchical. It allows efficient queries of aggregate information at different levels of each dimension. A typical application environment for our solution would be an OLAP system.

Second, we focus on the representation of spatial information, specifically on raster data, which are commonly used in geographic information systems (GIS) to represent spatial attributes (such as the altitude of a terrain, the average temperature, etc.). The new method enables several typical spatial queries with better response times than the state of the art, at the same time that saves space in both main memory and disk. Besides, we also present a framework to run a spatial join between raster and vector datasets, that uses the compact data structure previously presented in this part of the thesis.

Finally, we present a solution for the computation of empirical moments from a set of trajectories of a continuous time stochastic process observed in a given period of time. The empirical autocovariance function is an example of such operations. In this thesis, we propose a method that compresses sequences of floating numbers representing Brownian motion trajectories, although it can be used in other similar areas. In addition, we also introduce a new algorithm for the calculation of the autocovariance that uses a single trajectory at a time, instead of loading the whole dataset, reducing the memory consumption during the calculation process.

Resumen

En esta tesis estudiamos el problema de procesar grandes colecciones de datos, presentando nuevas estructuras de datos compactas y algoritmos que nos permiten almacenarlas y analizarlas de forma eficiente. Nos centramos principalmente en tres dominios: procesamiento de datos multidimensionales, representación de información espacial y análisis de datos científicos.

El nexo común es el uso de estructuras de datos compactas, que combinan en una única estructura de datos una representación comprimida de los datos y las estructuras para acceder a dichos datos. El objetivo es poder manipular los datos directamente en forma comprimida, y de esta manera, mantener los datos siempre comprimidos, incluso en la memoria principal. Con esto obtenemos dos beneficios: podemos gestionar conjuntos de datos más grandes en la memoria principal y aprovechar un mejor uso de la jerarquía de la memoria.

En la primera parte proponemos una estructura de datos compacta para bases de datos multidimensionales donde los dominios de cada dimensión están jerarquizados. Nos permite consultar eficientemente la información agregada (suma, valor máximo, etc.) a diferentes niveles de cada dimensión. Un entorno de aplicación típico para nuestra solución sería un sistema OLAP.

En segundo lugar, nos centramos en la representación de la información espacial, específicamente en datos ráster, que se utilizan comúnmente en sistemas de información geográfica (SIG) para representar atributos espaciales (como la altitud de un terreno, la temperatura media, etc.). El nuevo método permite realizar eficientemente varias consultas espaciales típicas con tiempos de respuesta mejores que el estado del arte, al mismo tiempo que reduce el espacio utilizado tanto en la memoria principal como en el disco. Además, también presentamos un marco de trabajo para realizar un join espacial entre conjuntos de datos vectoriales y ráster, que usa la estructura de datos compacta previamente presentada en esta parte de la tesis.

Por último, presentamos una solución para el cálculo de momentos empíricos a partir de un conjunto de trayectorias de un proceso estocástico de tiempo continuo observadas en un período de tiempo dado. La función de autocovariancia empírica es un ejemplo de tales operaciones. En esta tesis proponemos un método que

comprime secuencias de números flotantes que representan trayectorias de movimiento Browniano, aunque puede ser utilizado en otras áreas similares. En esta parte, también introducimos un nuevo algoritmo para el cálculo de la autocovariancia que utiliza una única trayectoria a la vez, en lugar de cargar todo el conjunto de datos, reduciendo el consumo de memoria durante el proceso de cálculo.

Resumo

Nesta tese estudamos o problema de procesar grandes coleccións de datos, presentando novas estruturas de datos compactas e algoritmos que nos permiten almacenalas e analizalas de forma eficiente. Centrámonos en tres dominios principais: procesamento de datos multidimensionais, representación de información espacial e análise de datos científicos.

O nexo común é o uso de estruturas de datos compactas, que combinan nunha única estrutura de datos unha representación comprimida dos datos e as estruturas para acceder a tales datos. O obxectivo é poder manipular os datos directamente en forma comprimida, e desta maneira, manter os datos sempre comprimidos, incluso na memoria principal. Con esto obtemos dous beneficios: podemos xestionar conxuntos de datos máis grandes na memoria principal e aproveitar un mellor uso da xerarquía da memoria.

Na primera parte propoñemos unha estructura de datos compacta para bases de datos multidimensionais onde os dominios de cada dimensión están xerarquizados. Permítenos consultar eficientemente a información agregada (sumar valor máximo, etc) a diferentes niveis de cada dimensión. Un entorno de aplicación típico para a nosa solución sería un sistema OLAP.

En segundo lugar, centrámonos na representación de información espacial, especificamente en datos ráster, que se utilizan comunmente en sistemas de información xeográfica (SIX) para representar atributos espaciais (como a altitude dun terreo, a temperatura media, etc.). O novo método permite realizar eficientemente varias consultas espaciais típicas con tempos de resposta mellores que o estado da arte, ao mesmo tempo que reduce o espazo utilizado tanto na memoria principal como no disco. Ademais, tamén presentamos un marco de traballo para realizar un join espacial entre conxuntos de datos vectoriais e ráster, que usa a estructura de datos compacta previamente presentada nesta parte da tese.

Por último, presentamos unha solución para o cálculo de momentos empíricos a partir dun conxunto de traxectorias dun proceso estocástico de tempo continuo observadas nun período de tempo dado. A función de autocovarianza empírica é un exemplo de tales operacións. Nesta tese propoñemos un método que comprime secuencias de números flotantes que representan traxectorias de movemento

Browniano, aínda que pode ser empregado noutras áreas similares. Ademais, tamén introducimos un novo algoritmo para o cálculo da autocovarianza que emprega unha única traxectoria á vez, en lugar de cargar todo o conxunto de datos, reducindo o consumo de memoria durante o proceso de cálculo.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Recent advances in hardware and software technology have opened the possibility of developing new applications. One of them is the field called *Big Data*, which involves large-scale data analysis. Big Data introduces several new challenges, because many of the conventional structures and algorithms are not capable of dealing with the *4 V's*: data volume too large (*Volume*), data rate too fast (*Velocity*), data too heterogeneous (*Variability*), and data too uncertain (*Veracity*). In this thesis, we focus on the first V, *data volume too large*.

The straightforward solution for managing huge datasets is to use parallel processing [DG08, KEW13, DXS+15, SETM13], where a good set of tools are available. However, there are other alternatives which involve the use of new data structures and algorithms. Among them, we can highlight the *in-memory data management* [Pla13, PZ12] and the *compact data structures* [Nav16]. The purpose of both approaches is to fit, manipulate, and query much larger datasets in main memory. Although the price of main memory has been reduced significantly, the current size of the datasets requires the use of compression in order to be able to fit them in main memory. Therefore, we need to efficiently process the compressed data in main memory, thus, retrieving a given datum must not require decompressing the dataset from the beginning. This restriction excludes most traditional compression techniques.

By processing compressed data in main memory, we take advantage of the lower latency and higher bandwidth of the upper levels of the memory hierarchy. This applies even if the data without compression fit in main memory or if even the compressed dataset does not fit in main memory.

A compact data structure approach is compatible with the use of parallel techniques, moreover, it can improve the final result. In a parallel scenario, data

interchanges between nodes can slow down the process due to bottlenecks in the network. Compression has been used to reduce bandwidth consumption [BR09] in that scenario. Traditionally, compression methods applied for this context have been designed to perform the compression and decompression processes very fast, since data have to be decompressed prior to any process, and thus data should be compressed before any data exchange and decompressed at the destination node. With a compact data structure approach, data can be interchanged between nodes in compressed form, and processed in that form at the destination node, saving space and time.

Finally, another interesting feature of compact data structures is that many of them are equipped with an index that, in the same compressed space, speeds up the queries. This feature is known as "self-indexation".

Our goal in this thesis is the study and design of new compact data structures and algorithms to represent huge collections of data in three different domains, where the use of large datasets is common. Our methods achieve better space/time results than other techniques of the state of the art in those domains.

1.2 Contributions

Our contributions can be divided into five main blocks. We describe now each of them and the problems they address:

Management of multidimensional data: Compact representation of Multidimensional data on Hierarchical Domains

Our first contribution consists in the design, analysis, implementation, and experimental evaluation of data structures for the compact representation of multidimensional data, where the domain of each dimension is organized hierarchically, as in OLAP systems. We propose a new compact data structure, called **C**ompact representation of **M**ultidimensional data on **H**ierarchical **D**omains (CMHD), that represents multidimensional data in compact space and allows us to improve the query time over the data. Basically, CMHD divides the data according to the hierarchies of the domains in each dimension and builds a tree that indexes aggregate information at the different levels of each partition. This allows us to perform more efficient queries since, using this aggregated information, it is not necessary to access the individual cells to answer them.

In addition, we present a generic multidimensional structure called k^n-treap, an extension to multiple dimensions of a two-dimensional compact summarization structure known as k^2-treap. This is used as baseline for our experiments.

The conceptual description of both structures and the results of the applications were published in the proceedings of the 23th International Symposium on String Processing and Information Retrieval (SPIRE 2013) [BCPLL+16].

Representation of GIS data: k^2-raster

Our second contribution consists in the design, analysis, implementation, and experimental evaluation of a new compact representation for raster data, called k^2-*raster*. Our structure is able to represent raster data in compact space and provides an efficient mechanism to improve queries over the data. The k^2-raster recursively divides the matrix M (the raster dataset) into k^2 equal-sized submatrices, and builds a tree representing this recursive subdivision. Each of those divisions is represented as a node of a tree, which keeps the minimum and the maximum values of the corresponding partition.

We evaluate our new solution with the current state of the art, concretely, the accumulated k^2-tree (denoted as k^2-acc) and the k^3-tree. We show that k^2-raster improves the results of these techniques, especially when the number of different values and the size of the dataset increase, which is critical when applying over real datasets.

A preliminary work was published in the proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM 2016) [LPSC16].

Spatial join: k^2-raster and R-tree

Our third contribution consists in the design, analysis, implementation, and experimental evaluation of a framework that includes the data structures and the algorithm to run a join between a raster and a vector dataset. We use a classical R-tree to index the vector dataset and the k^2-raster for the raster dataset.

In our experiments, we show that our approach obtains important savings in both running time and memory consumption, compared with baselines that represent the raster dataset in plain form. Our proposal is the first solution for solving a full join between raster and vector data using compact data structures, and it shows very good scalability properties.

Representation of scientific data: Compression of Brownian Motion

Our fourth contribution consists in the design, analysis, implementation, and experimental evaluation of a data structure for huge datasets of Brownian trajectories or trajectories of continuous time stochastic processes. Our structure, called **C**ompressed **B**rownian **M**otion (CBM), is oriented to facilitate the efficient computation of the sample autocovariance function. Each trajectory is first processed with differencing encoding and the resulting sequence is compressed with an integer compressor.

We study the behavior of CBM against the R package and a C program that uses uncompressed data, obtaining important savings in running time.

Memory-friendly covariance function

The fifth contribution is a new memory-efficient algorithm to compute the sample autocovariance function where it is only necessary to load a single trajectory at a time, avoiding to keep the whole dataset in main memory. This considerably reduces the space used for computing the covariance compared to the classical algorithm.

1.3 Structure of the Thesis

The structure of the thesis is as follows: First, in Chapter 2, we present some basic concepts about data compression and compact data structures. In Chapter 3, we describe several data structures that are used in this thesis. After that, our contributions are grouped into three parts:

- **Part I** addresses the problem of the efficient representation of multidimensional data over hierarchical domains.
  - Chapter 4 introduces the field of multidimensional data, more concretely, the Data Warehouse databases and the OLAP systems. In addition, we describe the k^n-treap, a straightforward extension of the k^2-treap to manage multiple dimensions, which allows efficient summarization queries along with generic ranges.
  - Chapter 5 presents our new compact data structure called CMHD (Compact representation of Multidimensional data on Hierarchical Domains). By adapting the hierarchy to the domain of each dimension, CMHD allows much more efficient queries than a generic multidimensional structure.
  - Chapter 6 presents the experimental evaluation of the k^n-treap and the CMHD over different datasets, varying the domain hierarchies and the number of dimensions.
  - Chapter 7 discusses the conclusions and some future works for our contribution.
- **Part II** introduces a new compact data structure designed to store raster data, which is commonly used to represent attributes of the space (temperatures, pressure, elevation measures, etc.) in geographical information systems. In addition, this part also introduces a framework that includes the data structures and the algorithm to perform a spatial join between a raster and a vector dataset, which uses a k^2-raster to represent the raster dataset.
  - Chapter 8 introduces the basic concepts of geographic information systems, such as the different raster models or the spatial join. Also two previous compact data structure for the representation of raster data are described, the *accumulated* k^2-tree (k^2-acc) and the k^3-tree.

  - Chapter 9 presents our contribution called k^2-raster. 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 over a raster.
  - Chapter 10 presents a framework to run a join between a raster (k^2-raster) and a vector dataset (R-tree).
  - Chapter 11 includes the experimental evaluation, where k^2-raster is compared with the current state of the art over several real datasets. Besides, the results of different experiments for the spatial join are shown.
  - Chapter 12 summarizes the main contributions and other applications for both proposals.

- **Part III** presents two main contributions. The first is a new compact representation for huge sets of functional data or trajectories of continuous time stochastic processes. The second contribution is a new memory-efficient algorithm to compute the sample autocovariance function.

  - Chapter 13 introduces the motivation of the problem, some notation, and basic concepts of the Brownian motion.
  - Chapter 14 presents our proposal to represent a set of trajectories of Brownian motion, called CBM, and the memory-efficient algorithm to compute the sample autocovariance function.
  - Chapter 15 shows and discusses the experimental results for a set of float point matrices of different sizes.
  - Chapter 16 presents the main conclusions, other applications, and future work.

In order to complete the thesis, Part IV includes the concluding Chapter 17 that summarizes the contributions of our work and gives some general future lines of research. Finally, Appendix A shows a list of publications with their cites and other research activities related to this thesis; and Appendix B presents a summary of the thesis in Spanish.

Chapter 2

Basic Concepts

This chapter introduces some basic concepts, notations, and basic compact data structures that are used during this thesis. Section 2.1 gives some notions about Information Theory. Section 2.1.2 introduces the basis of data compression and different compression techniques for integers. Finally, Section 2.2 presents some basic compact data structures, which are commonly integrated into other structures.

2.1 Information Theory and Data Compression

2.1.1 Basic concepts on Information Theory

The aim of Information Theory is to deal with the transmission of information through communications channels. The basis of this field and many concepts used nowadays were settled in Shannon's work [Sha48]. One of the most important concepts for this thesis is the measuring of the information in term of bits, i.e., the minimum amount of space required to encode a message. This allows us to know how complex to encode a piece of information is and also whether it is very repetitive or not.

Let X be a discrete random variable with a probability mass function $p(X)$ and domain d_x. We define $I(x) = \log \frac{1}{p(x)}$ as the amount of information associated with an outcome $x \in d_x$. With this formula, it is intuited that an outcome with a high probability of occurrence provides less information than one with a lower probability. For instance, if a outcome x has probability $p(x) = 1$, there is no surprise when x appears because it is expected and does not provide new information.

Another measure is the *entropy* of X, which gives us the amount of surprise that is expected from X. The *entropy* of X is defines as:

$$H(X) = E[I(X)] = \sum_{x \in d_x} p(x) \log \frac{1}{p(x)} \tag{2.1}$$

The entropy $H(X)$ measures the average amount of information obtained by observing a random variable.

If the source of information is not an infinite source but just a message S, then we can define the *zero-order empirical entropy* in terms of the Shannon entropy of the observed probabilities of its symbols. Thus, given a sequence $S[1, n]$ over an alphabet $\Sigma = [1 \dots \sigma]$ where each symbol s appears n_s times in S, the zero-order empirical entropy is defined as:

$$H_0(S) = \sum_{1 \leq s \leq \sigma} \frac{n_s}{n} \log \frac{n}{n_s}. \tag{2.2}$$

2.1.2 Data Compression: basic concepts

Data compression appears by the necessity to represent large datasets in less space, improving their manipulation and storage in any system. For example, using data compression we can reduce the number of packets sent over a network in a distributed system.

2.1.2.1 Classification of compression techniques

Compression techniques have two main methods: an *encoding* method that transforms the original message into a compressed version, and a *decoding* method that recovers the original message from the encoded message. We can classify compression techniques in two categories depending of result of the decoding process.

- **Lossy compression techniques** perform an encoding process that does not allow retrieving the original message but an approximate version of it. During the encoding phase, some information of the original message is lost, which implies that the decoded message will be very similar to the original message but not exactly the same. These techniques are used in areas where it is not necessary to receive the original message, but a similar version is sufficient. For example, it is very common to use it to compress audio, images or videos, where humans cannot detect small differences.

- **Lossless compression techniques** return an exact copy of original data. Some scenarios do not allow any type of information loss and therefore these techniques should be used. For example, in text compression we cannot lose or change a character or a word, since the message may become meaningless. In this thesis, we only deal with this type of compression techniques.

2.1.3 Measuring the efficiency of compression techniques

We can use two different types of measures to determine the efficiency of a compression technique:

- The performance of the compression and decompression algorithms. The *theoretical complexity* of those algorithms is a good measure that gives us an approximation of how a technique behaves and allows us to compare it with other techniques. In practice, another way is to measure the compression and decompression times (usually in seconds).
- The compression achieved. The *compression ratio* is a good measure that represents the percentage that the compressed dataset occupies with respect to the original dataset. It is calculated as $\frac{size_{compressed}}{size_{original}} \times 100$.

2.1.4 Compressing Integer Numbers

Let $S = (s_1, s_2, \ldots, s_n)$ be a sequence of symbols over an alphabet Σ. A *code* is an injective function $\mathcal{C} : \Sigma \to \{0,1\}^*$ that assigns a distinct sequence of bits $\mathcal{C}(s_i)$ (codeword) to each symbol $s_i \in \Sigma$. A way to compress S is to order the symbols of Σ by their frequency in S, and assign shorter codewords to the most frequent symbols. This strategy is called *statistical encoding*. One example of this is Huffman coding [Huf52], which is the best code that is univocally decodable.

If Σ is formed by integers, Huffman can be used. However, if the size of Σ is large, since Huffman has to explicitly store the function $\mathcal{C}$, that space may be prohibitive. In this case, *fixed or static codes* can be used. These codes do not use the probabilities, instead, each integer is always mapped to the same codeword, that is, $\mathcal{C}$ does not depend on the exact input sequence S and therefore there is no need to store it.

Still, the main idea is the same, compression is achieved by assigning shorter codewords to more frequent integers, and it is assumed that these numbers are the smallest integers. Therefore, better compression can be obtained if the original sequence is preprocessed with relative or differencing encoding. Each integer, except the first one, is replaced by its difference with the previous one. However, differences can be negative, so this poses a problem as most codes for integers only work with positive numbers. The first solution is to store all the integers in absolute value and add 1 bit per integer to indicate whether the original integer was positive or negative. This additional bit can be avoided by using the ZigZag encoding, which maps signed integers to unsigned integers. The -1 is encoded as 1, 1 as 2, -2 as 3, and so on. The problem with this approach is that numbers with a large magnitude will have a codeword with an even greater magnitude.

Examples of fixed codes are the unary code, Elias codes (γ-codes and δ-codes) [Eli75], or Golomb codes [Gol66]. The unary code, for example, represents a value x as 1^x0, that is, with x ones followed by a zero[1]. From the input sequence, the Golomb encoder chooses one parameter m. The codeword assigned to a source symbol s_i is composed of two parts. The first one is $q = \lfloor s_i/m \rfloor$ encoded in unary.

[1]Notice that the ones and zeros are interchangeable without loss of generality, so x can be represented as 0^11 as well.

The second part is $r = s_i - qm$ encoded in minimal binary: being $c = \lceil \log m \rceil$, the first $2^c - m$ values of r are encoded in binary using $c - 1$ bits, and the rest are encoded in binary using c bits. Rice codes [Ric79] are a special case of Golomb codes, in which the parameter m is chosen to be a power of two. This choice makes their computation faster, and thus Rice codes are extensively used.

The codes shown so far produce codewords of arbitrary bit lengths. This causes bit manipulations that slow down the encoding and decoding processes. To avoid this, there is a family of codes that produce codewords formed by one or more chunks of b bits, usually of 8 bits. The first example is Vbyte [WZ99]. The $\lfloor \log s_i \rfloor + 1$ bits required to represent s_i in binary are split into blocks of $b - 1$ bits. The chunk holding the most significant bits of s_i is completed with a bit set to 0 in the highest position, whereas the rest are completed with a bit set to 1 in the highest position. Therefore, using chunks of 8 bits, the first chunk of a codeword is always a number between 0 and 127, and the rest are between 128 and 255. Therefore, we can split the byte values into two types, the *beginners* (values between 0 and 127) and the continuers (values between 128 and 255), where the beginners signal the begin of a codeword. (s,c)-Dense Code (SCDC) [BFNP07] is similar to Vbyte. It also has two types of chunks, but instead of beginners, it uses *stoppers*. A codeword is formed by one stopper, and zero, one or more continuers. The stopper chunk signals the end of a codeword, and thus, that the next chunk corresponds to the next codeword. However, instead of using 128 values for each set, SCDC decides what is the best distribution of the byte values between stoppers and continuers for a given input sequence, in order to obtain the best compression.

Codes based on chunks are faster, but they pay a price in space. A different family of codes tries to join the good space consumption of the bit-based codes and the fast encoding and decoding of the byte-aligned codes. Instead of encoding and decoding each source symbol, these codes treat short sequences of numbers and read whole computer words from the input. For instance, PforDelta [ZHNB06] encodes a fixed number of integers at a time (typically 128), using for all of them the number of bits needed for the largest one. A fraction of the largest numbers (usually 10%) is encoded separately, and the other 90% is used to calculate how many bits are needed per number.

All codes we have seen do not provide direct access to positions, that is, we cannot directly access the codeword representing the i^{th} symbol in the original sequence without decompressing from the beginning, because they use variable length codewords. The classical solution to this problem is to use absolute pointers to sampled elements, that is, to each h^{th}-element of the sequence. These pointers obviously suppose an overhead. However, there are techniques that avoid the use of pointers. These techniques use a conventional code along with an additional structure to allow direct access. Examples can be based on Elias-Fano codes [Mun96, OS07], on Interpolative coding [Teu11], or on Vbyte [BLN13]. The latter approach is denoted as Directly Addressable Codes (DACs), and they will be presented in more

detail in Section 3.1, since they are used profusely in this thesis.

Other techniques make use of the wavelet tree [GGV03], which can also store the variable length codewords of a sequence, allowing us to retrieve the i^{th} codeword without decompressing the previous codewords. It has been used with Huffman [GGV03], Vbyte [BFLN08], Elias and Rice codes [Kül14], or Fibonacci codes [KS16].

2.2 Compact data structures

The main objective of compact data structures is to represent the data (text, sequences of numbers, trees, etc.) in a compact way, using as little space as possible, while enabling us to retrieve efficiently any datum without the need of decompressing the whole dataset. These structures are designed to keep data always compressed, even when are loaded into main memory, which improves the processing of larger datasets. In addition to the benefit of saving disk space, the compact data structures take better advantage of memory hierarchies, operating at the fastest levels of the hierarchy to improve their performance. In many cases, they also provide indexes that allow us to answer queries even faster than performing those queries over the plain representation.

2.2.1 Rank and select over bitmaps

Let $B[1, n]$ be a bitmap, that is, a sequence of bits. We can define three basic operations:

- $rank_b(B, i)$ returns the number of occurrences of bit $b \in \{0, 1\}$ in $B[1, i]$. Consider a bitmap $B = 110110110$. Therefore, $rank_0(B, 5) = 1$, as only one 0 appears up to the position 5, whilst $rank_1(B, 5) = 4$, since the sequence B[1,4] has four 1s. When omitting b, $rank$ operation returns the number of 1s up to a given position, that is, $rank(B, i) = rank_1(B, i)$.

- $select_b(B, i)$ locates the position of the i^{th} occurrence of b in B. Following the previous example, $select_0(B, 2) = 6$ and $select_1(B, 2) = 2$, which means that the 2^{th} 0 appears at position 6 and the 2^{th} 1 at position two in sequence B.

- $access(B, i)$ returns the bit value at position i. For instance, $access(B, 5) = 1$, the 5th value of sequence B is a 1.

They are basic operations used in most of the compact data structures of the literature [Nav16]. Figure 2.1 illustrates the three previous operations.

Jacobson, whose PhD thesis can be taken as the starting point of the study of compact data structures [Nav16], showed that the $rank$ operation can be answered in constant time over plain bitmaps using a two-level directory structure [Jac89b]. Given a binary sequence $B[1, n]$, the first level stores the result of $rank_1(B, j)$ for

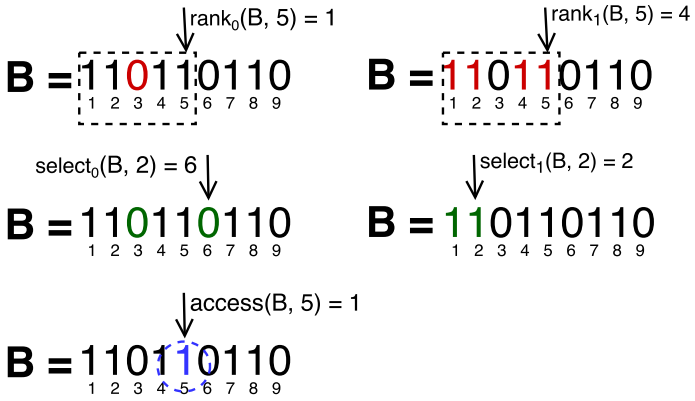


Figure 2.1: *rank*, *select* and *access* over a bitmap $B = 110110110$.

each j multiple of $s = \lfloor \log n \rfloor \lfloor \log n/2 \rfloor$, while the second level holds, for each k multiple of $b = \lfloor \log n/2 \rfloor$, the relative rank within previous blocks of size s. We can compute $rank_1(B, i)$ using those two directories. From the first level, we obtain the *rank* value until the previous multiple of s, while the second level returns the *rank* value until the previous multiple of b. Finally, the number of ones between the previous multiple of b and j is calculated and added up to the final result. This can be obtained in constant time by using a *lookup table*, which stores the result of *rank* for all possible subsequences of size b. However, the *select* operation is computed in $O(\log \log n)$ time using binary searches.

In order to improve queries over the bitmap, Clark and Munro [Cla96, Mun96] proposed a new solution that solves both operations in constant time using just $o(n)$ extra bits, therefore the final total cost is $n + o(n)$.

2.2.2 Compressed bitmap representation

Others solutions were presented to store a bitmap in a very compact way, while they still provide the operations *rank*, *select* and *access*.

Pagh [Pag99] proposed a new approach that splits the bitmap in equal-sized blocks. Then it explicitly stores the *rank* of the first element of each block. In addition, the blocks are compressed with a schema that clusters adjacent blocks into intervals of varying length. Extraction of rank information from this compressed form is also done by using a lookup table and a two-level structure.

Raman et al. [RRR02] also divided the sequence into blocks but each of them has associated a tuple (c_i, o_i), where c_i identifies the class of the block (the number of 1 bits that contains) and o_i is the offset of that block inside the vocabulary of

all possible blocks in the class c_i (blocks with c_i bits). Let b be the size of each block, the cost of representing c_i is $\lceil \log(b+1) \rceil$ bits and o_i uses $\lceil \log(\binom{b}{c_i}) \rceil$ bits. This solution provides *rank* and *select* operations in constant time.

Okanahora and Sadakane [OS07] presented several compact solutions specially designed for sparse bitmaps (those where the number of 1s is much larger than the number of 0s, or vice versa). They can achieve good compression for the bitmap when it is truly sparse, besides providing efficient operations. Each of the solutions is based on different ideas, so its behavior varies depending on the sequence. Other strategy for sparse bitmaps is *gap encoding*, which encodes each 1 bit as the gap between the previous 1 bit [Sad03, GWSV06, MN07].

2.2.3 Compressed tree representations: LOUDS

Level-ordered unary degree sequence (LOUDS) [Jac89a] is a tree representation for ordered trees that appends the degree r of each node in (left-to-right) level order in unary code (1^r0). The sequence of degrees uniquely identifies the tree of n nodes using $2n-1$ bits. For each node we have a 0 bit (when its degree is specified) and, except the root node, a 1 bit (its father has a 1 bit for each child). By adding a false super-root node, we can maintain the property that all the nodes of the tree, including the root, correspond to one 1 bit. This fix only increases the final sequence by 2 bits.

Figure 2.2 illustrates an ordered tree (top) and its LOUDS representation S (bottom). The *fake super-root* node (in color gray) does not belong to the original tree. The process follows a left-to-right level order starting by the *super-root* node. That node has degree $r = 1$ (only one child), therefore, we append 1 in unary code ($1^r0 = 10$) to sequence S. Then, we continue with the next node labeled as "1" (the real root node). Node "1" has degree $r = 3$, so we concatenate sequence of bits 1110 to S. Next, we jump to the next level and process the node labeled as "2", which has degree $r = 2$ and the sequence 110 is appended to S. The process continues analogously with the remaining nodes of the tree and stops when the degrees of all nodes are added to S. The final sequence is 1011101101110101100100110000000 (31 bits long).

This representation is navigated using *rank* and *select* operations. LOUDS allows basic operations such as access to children, obtaining the position of the father or counting the number of children, among others. Given a node x and the position p of its corresponding 1 bit in the LOUDS representation, the first child of x is located at $c = select_0(rank_1(p)) + 1$. For instance, the first child of the node labeled "3", whose 1 bit is at position 3 in the LOUDS sequence, is calculated as $c = select_0(rank_1(3)) + 1 = select_0(3) + 1 = 8 + 1 = 9$, which corresponds to the position of the node labeled "1". In order to access their other children we simply add the offset of that child node. Following the previous example, its third child (offset 2) is located at position $c + 2 = 3 + 2 = 5$. On the other hand, the parent of x is obtained as $f = select_1(rank_0(p))$. The parent of the node labeled "3" is

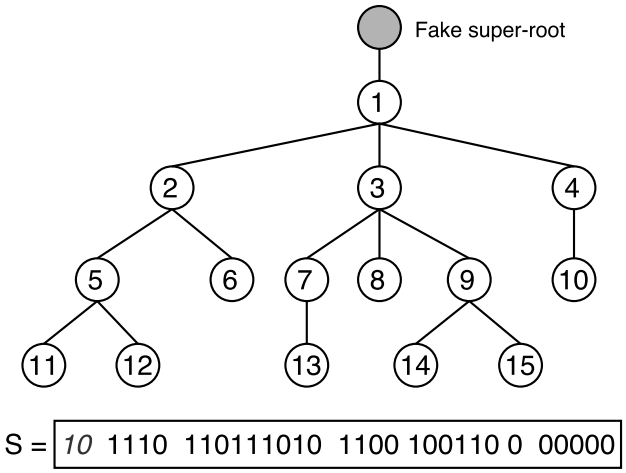


Figure 2.2: LOUDS representation for a tree with 15 nodes.

$f = select_1(rank_0(3)) = select_1(1) = 0$. Finally, to obtain the number of children of node x, we have to locate the position of the first child and count the ones up to the next zero.

Chapter 3

Previous work

In this chapter, we present several data structures highly related or directly included in the contributions proposed in this thesis. Section 3.1 introduces Directly Addressable Codes (DACs), an encoding scheme for sequence of integers that allows direct access to any position of the sequence. Section 3.2 presents the k^2-tree, as some contributions of this thesis are based in this structure. The k^2-tree was designed to compress sparse binary matrices, originally for the representation of Web graphs. Finally, Section 3.3 describes a compact data structure to represent grids of integer values called k^2-*treap*, which is basically constructed as an enrichment of a k^2-tree.

3.1 Directly Addressable Codes (DACs)

Directly Addressable Codes (DACs) [BLN13] is a variable-length code for sequences of integers that supports fast direct access to any given position of the sequence, that is, it allows decoding the i^{th} integer without the need of decompressing the previous integers. It obtains a very compact representation, if the sequence of integers has a skewed frequency distribution, where the number of occurrences of smaller integer values is higher than the number of occurrences of larger integer values.

Given a sequence of integers $X = x_1, x_2, \ldots, x_m$, DACs take the binary representation of that sequence and rearrange it into a level-shaped structure as follows: the first level B_1 contains the first n_1 bits (least significant) of the binary representation of each integer. A bitmap C_1 is added to indicate, for each integer, whether its binary representation requires more than n_1 bits or not. More precisely, for each integer, there is a bit set to 0 if the binary representation of that integer does not need more than n_1 bits and a 1 otherwise. In the latter case, the second level B_2 stores the next n_2 bits of the integers having a 1 in C_1, and a bitmap C_2 marks the integers needing more than $n_1 + n_2$ bits, and so on. This scheme is

repeated as many levels as needed. The number of levels $\mathcal{L}$ and the number of bits n_l at each level l, with $1 \leq l \leq \mathcal{L}$, is calculated in order to achieve the maximum compression. Figure 3.1 shows an example of DACs encoding for the first integers of sequence.

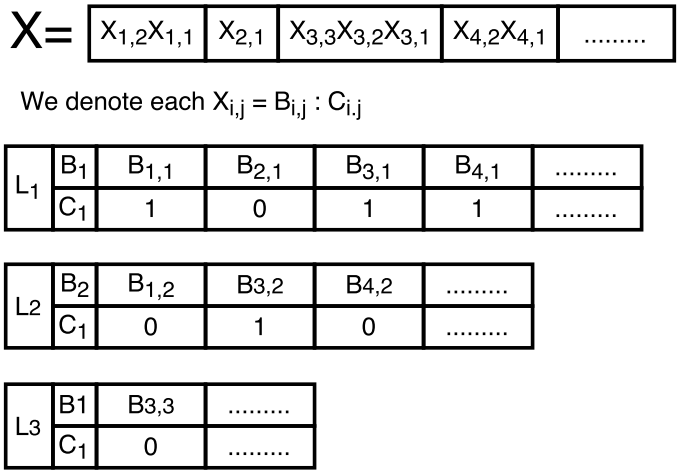


Figure 3.1: Example of DACs encoding.

DACs can efficiently retrieve the integer encoded at given position by obtaining the n_l bits at each level that form the binary representation of the number. That is, to recover the number, a top-down traversal is needed, and thus, the worst case time for extracting a random codeword is $O(\mathcal{L})$, being $\mathcal{L}$ the number of levels used. The position of the corresponding bits at each level is obtained performing *rank* operations over the bitmaps B_l.

We can adjust the number of levels and the number of bits used at each level (n_l) to obtain the maximum possible compression. However, this may lead to slow access times, if it requires a large number of levels. DACs can be configured to obtain the minimum possible space but limiting the number of levels $\mathcal{L}$.

3.2 The k^2-tree

The k^2-tree [BLN14]is a data structure originally designed to compress Web graphs that, as all compact data structures, allows accessing and querying the data without decompressing it. This method follows a region quadtree decomposition [Sam06] to subdivide the binary adjacency matrix of a graph and represents this subdivision using a simplified LOUDS tree representation.

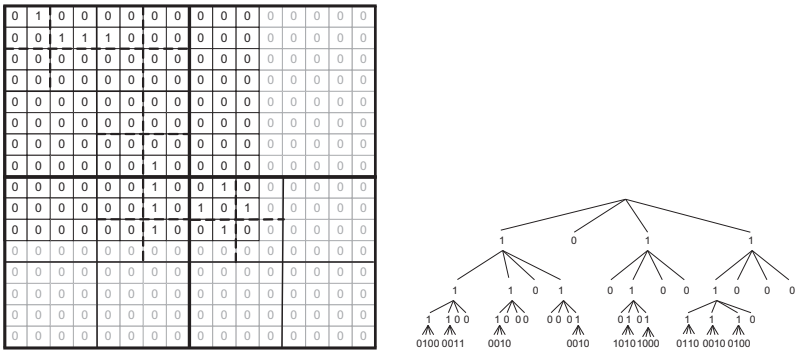


Figure 3.2: Example of binary matrix (left) and resulting k^2-tree representation (right), with $k = 2$.

From a binary matrix of size $n \times n$, and being k an input parameter, the k^2-tree is built as a non-balanced k^2-ary tree, where each node describes a submatrix resulting from a recursive division of the matrix into k^2 submatrices of the same size. The first partition divides the original matrix into k rows and k columns of submatrices of size n^2/k^2. Each of those submatrices generates a child node of the root having only one bit whose value is 1 iff there is at least one 1 in the cells of that submatrix. A 0 child means that the submatrix has all 0s and then, the tree decomposition ends there. The submatrices having at least one 1 are recursively divided into k^2 submatrices, producing each one a child node of the corresponding parent. This process continues until reaching a submatrix full of 0s or until reaching the cells of the original matrix (i.e., submatrices of size 1×1). Figure 3.2 shows an example of this subdivision (left) and the resulting k^2-ary tree (right) for $k = 2$.

Instead of using a pointer-based representation, the tree is compactly represented by just using two bitmaps T and L, whose values are the bit values resulting from a breadth-first traversal of the tree. T stores all the bits of the k^2-tree except those at the last level of the tree, whereas L stores the last level of the tree, thus containing the binary value of (some) original cells of the adjacency matrix. It is possible to navigate this space-efficient representation by just accessing bitmaps T and L. In particular, it is possible to retrieve any cell, row, column or region of the matrix in a very efficient time. This navigation is obtained by means of top-down traversals in the conceptual tree, which are simulated with *rank* operations over T.

The k^2-tree has an excellent performance in both space and time when the binary matrix is sparse, with large zones of 0s and where the 1s are clustered. There also exists a variation of the k^2-tree that compresses areas full of 1s [dBÁGB+13]. In this variation the subdivision ends when the algorithm finds a submatrix full of 0s (*white*

zones) or full of 1s (*black* zones), adding a method to distinguish black and white areas. Therefore the subdivision only continues when the submatrix has a mixture of 0s and 1s (*gray* zones). This representation is more suitable for representing other types of datasets different from Web graphs, such as binary images.

Apart from its original application, the k^2-tree has been used for many different purposes, among others, to support the compact representation of RDF datasets [ÁGBF$^+$15], moving objects [RBR12, BGBNP16], general graphs [ÁGBLP10], and raster data [dBÁGB$^+$13]. The approaches used to represent raster data using the k^2-tree as basis will be explained in Section 8.2.

3.3 The k^2-treap

A k^2-treap [BdBK$^+$16] is a compact data structure designed to represent grids in compact form, while supporting efficient general aggregated queries and other simple range queries. It was originally described for the particular case of obtaining the maximum points in a 2-dimensional matrix, also called top-k queries. This could allow, for example, a GIS system to store a grid, representing a geographic area containing the height of each point, and answer queries about the top-k highest points in a given subregion.

Let M be a matrix of size $n \times n$ where each cell stores either an empty value or a value $v \geq 0$. The k^2-treap recursively partitions M into k^2 equal-sized submatrix, following the same division technique used by the original k^2-tree. We build a conceptual tree representing this recursive partition, where a node stores the maximum value of the submatrix and its coordinates in the corresponding submatrix. Then, that value is removed from the matrix or settled empty. The process is as follow: The root node is created and it keeps the maximum value of the whole matrix and its coordinates, and sets that cell as empty. Then, the matrix is divided into k^2 submatrices and we append a child node to the root for each new partition. We recursively repeat this process for each child until we find a empty submatrix or reach the individual cells. The k^2-treap uses a k^2tree to represent its topology and other two structure to store the values and their positions.

Figure 3.3 shows an example of k^2-treap construction from a matrix of size 8×8, using $k = 2$. The maximum value of each submatrix for the four steps of the recursive subdivision is highlight with color gray (top). On the bottom of the figure, we represent the conceptual tree from the construction of the k^2-treap. Each node stores the maximum value of its submatrix and the coordinate of the corresponding cell. Note that in the last level of the tree, only the maximum value is necessary. In the first iteration (under label "Step 1"), we created the root with the maximum value of whole matrix (10) and its corresponding coordinates (1,1). Then, the cell is removed from the matrix and continue with the next iteration (under label "Step 2"). We divide the matrix into four submatrices of size 4×4 and add a child node to the root for each one. Recall that nodes keep the local coordinates instead of global

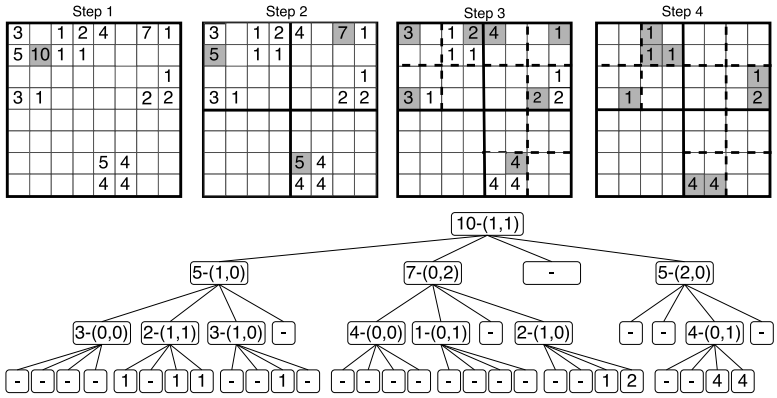


Figure 3.3: Example of the construction of the k^2-treap.

coordinates, for example, the maximum value of the second submatrix (right-top) is 7 and the local coordinates in its submatrix are (0, 2). Also observe that the third submatrix (left-bottom) is empty (marked with the symbol "-" in the tree), thus we do not further subdivide the corresponding submatrix. In the next step (under label "Step 3"), the process continues for the other three non-empty nodes. Nodes of these submatrices are stored in the third level of the conceptual tree. Finally, the process reaches the individual cells (under label "Step 4") and each submatrix has a child node for each of its cells.

In order to obtain the value of an individual cell, we only need to perform a top-down traversal of the tree and compare the coordinates stored in the nodes with the position of our query. If they are equal, we return the corresponding value stored in the node, otherwise we go down through the branch where the cell would be located. To resolve range queries, the process is very similar, but at each step, we calculate all children with some cell in the range of the query.

There are several variants of the k^2-treap, allowing to store, as precomputed values in the internal nodes, the maximum, minimum, or aggregated values (sum, average, number of non empty cells) of the corresponding subtree.

Part I

Multidimensional data

Chapter 4

Introduction

In this Part I, we consider the problem of representing multidimensional data where the domain of each dimension is organized hierarchically, and the queries require summary information at different levels of the hierarchy of each dimension.

We present a compact data structure that partitions the space according to the hierarchies, instead of performing a regular partition like generic multidimensional structures. Therefore, the queries of interest for OLAP applications, which combine nodes of the different hierarchies, will require aggregating the information of just a few nodes, much fewer than if we used a generic space partitioning method.

Our baseline for comparison will be an extension of the k^2-treap to n dimensions, the n-dimensional treap, called k^n-treap.

The k^n-treap will then be extended so that it can follow an arbitrary hierarchy, not only a regular one. The topology of each hierarchy will be represented using LOUDS. This new structure is called CMHD (Compact representation of Multidimensional data on Hierarchical Domains). Although we focus on sum queries, it is easy to extend our results to other kinds of aggregations, for example, the maximum value. This new contribution is presented in Chapter 5.

We experiment with both structures, the k^n-treap and CMHD, with different datasets, by varying their hierarchies and number of dimensions. The results of those experiments are shown in Chapter 6. Finally, in Chapter 7, we present our conclusions and future work.

In this chapter, we introduce the motivation of our contribution and the different domains where it can be applied. Section 4.1 presents a brief description of multidimensional data, including the definition of Data Warehouses, and the description of OLAP systems. Next, we present our baseline, the k^n-treap, in Section 4.2.

4.1 Introduction

In many application domains, data is organized into multidimensional matrices. In some cases, like GIS and 3D modelling, the data are actually points that lie in a two- or three-dimensional discretized space. There are, however, other domains such as OLAP (Online Analytical Processing) systems [CCS93, CD97] where the data are sets of tuples that are regarded as entries in a multidimensional cube, with one dimension per attribute.

In this type of environments, it is very frequent to have to manage huge collections of data, where several dimensions are involved. Moreover, it is necessary to answer complex operations to obtain summaries about very specific regions of the multidimensional cube, which involves retrieving a large number of cells. This presents us two main challenges: we must reduce the space requirements of those datasets and, at the same time, include some type of index to improve the efficiency of summary queries.

4.1.1 Data Warehouses (DWs)

Data Warehouses (DWs) appeared due to the need to process and analyze large amounts of data, since the classic *database management systems* (DBMSs) were not suitable for this type of tasks. The most widespread definition is proposed by Bill Inmon, considered one of the parents of DW [Inm93]:

> A data warehouse is a subject-oriented, integrated, time-variant and non-volatile collection of data in support of management's decision making process.

Unlike classic DBMSs, designed to maintain a balance between data integrity and response time, *Data Warehouses* focus more on the speed of analysis than on the integrity of the data, because these do not change over time. In addition, a *Data Warehouse* stores the temporal evolution of the data, that is, when it receives a new value, it does not delete the previous one, but keeps both. Another feature of DWs is that it is very common to store aggregated values (summaries) of their data to improve the speed of queries.

There are several additional tools that integrate with DW to perform data analysis. The two main families of this type of tools are *data mining* and *online analytical processing* (OLAP).

For this thesis, we are interested in OLAP tools, since our contribution can be applied on this domain.

4.1.1.1 Online Analytical Processing (OLAP)

The term OLAP was proposed by Codd [CCS93], motivated by the design of a new system to perform analysis on the data, with features other than transactional. Codd

also established a series of characteristics, including the multidimensional conceptual view. Although some authors defend the use of the E-R model, the majority of authors are committed to using multidimensional data modeling. For this reason, we will focus on the multidimensional data model.

Therefore, in OLAP system [CCS93, CD97], the data are sets of tuples that are regarded as entries in a multidimensional cube, with one dimension per attribute. The domains of those attributes are not necessarily numeric, but may have richer semantics. A typical case in OLAP [KR02], in particular in snowflake schemes [LL03], is that each tuple contains a numeric summary (e.g., amount of sales), which is regarded as the value of a cell in the data cube. The domain of each dimension is hierarchical, so that each value in the dimension corresponds to a leaf in a hierarchy (e.g., countries, cities, and branches in one dimension, and years, months, and days in another). Queries ask for summaries (sums, maxima, etc.) of all the cells that are below some node of the hierarchy across each dimension (e.g., total sales in New York during the previous month).

A way to handle OLAP data cubes is to linearize the hierarchy of the domain of each dimension, so that each internal node corresponds to a range. Summarization queries are then transformed into multidimensional range queries, which are solved with multidimensional indexes [Sam06]. Such a structure is, however, more powerful than necessary, because it is able to handle *any* multidimensional range, whereas the OLAP application will only be interested in queries corresponding to combinations of nodes of the hierarchies. There are well-known cases, in one dimension, of problems that are more difficult for general ranges than if the possible queries form a hierarchy. For example, categorical range counting queries (i.e., count the number of different values in a range) require in general $\Omega(\log n/\log\log n)$ time if using $O(n\,\mathrm{polylog}\,n)$ space [LvW13], where n is the array size, but if queries form a hierarchy it is easily solved in constant time and $O(n)$ bits [Sad07]. A second example is the range mode problem (i.e., find the most frequent value in a range), which is believed to require time $\Omega(n^{1.188})$ if using $O(n^{1.188})$ space [CDL+12], but if queries form a hierarchy it is easily solved in constant time and linear space [HSTV14].

4.2 Baseline for multidimensional data: the k^n-treap

The k^n-treap is a straightforward extension of the k^2-treap to manage multiple dimensions. It uses a k^n-tree (in turn, a straightforward extension of the k^2-tree) to store its topology, and stores separately the list of aggregate values obtained from the sum of all values in the corresponding submatrix. Figure 4.1 shows a matrix and the corresponding k^n-treap. The example uses two dimensions, but the same algorithms are used for more dimensions. The domains hierarchies are shown (light gray), although these do not affect the construction of the k^n-treap but helps us

to visualize the use of the k^n-treap in this type of environment, that is, an OLAP system.

4.2.1 Construction

Consider a hypercube of n dimensions, where the length of each dimension is $len = k^i$ for some i. If the lengths of the dimensions are different, we can artificially extend the hypercube with empty cells, with a minimum impact in the k^n-treap size. The k^n-trees, which will be used to represent the k^n-treap topology, are very efficient to represent wide empty areas. The algorithm to build the k^n-treap starts storing at its root level the sum of all values of the matrix. It then splits each dimension into k equal-sized parts, thus giving a total of k^n submatrices. We define an ordering to traverse all the submatrices (in the example, rows left-to-right, columns top-to-bottom). Following this ordering, we add a child node to the root for each submatrix. The algorithm works recursively for each child node that represents a non-empty submatrix, storing the sum of the cells in this submatrix, splitting it and adding child nodes. For empty sumatrices, the node stores a sum of 0. The implemented algorithm is recursive and each sum is actually computed only once, when returning from the recursive calls.

As we can see in Figure 4.1, the root node stores 104, the sum of all values in the matrix, and it is decomposed into 4 matrices of size 8×8, thus adding 4 children to the root node. The algorithm proceeds recursively for the four children of the root node. The first child then is subdivide into 4×4 equal-sized matrices of size 4 and so on. Notice that the first submatrix (top-left) is full of zeroes, so this node just stores a sum of 0 and is not further decomposed.

4.2.2 Data structures

The final data structures used to represent the k^n-treap are the following:

- *Values (V)*: It is an integer array containing the aggregated values (sums) for each (sub)matrix, as they would be obtained by a levelwise traversal of the k^n-treap. It is encoded using DACs.
- *Tree structure (T)*: It is a k^n-tree that stores a bitmap T for the whole tree except its leaves. In this case, the usual bitmap L for the leaves in a standard k^n-tree is not used, because the information about which cells have or not a value is already represented in V. Therefore L is not needed.

4.2.3 Queries

The navigation through the k^n-treap is basically a depth-first traversal. Finding the child of a node can be done very efficiently by using *rank* and *select* operations as in the standard k^2-tree. The typical queries in this context are: finding the value

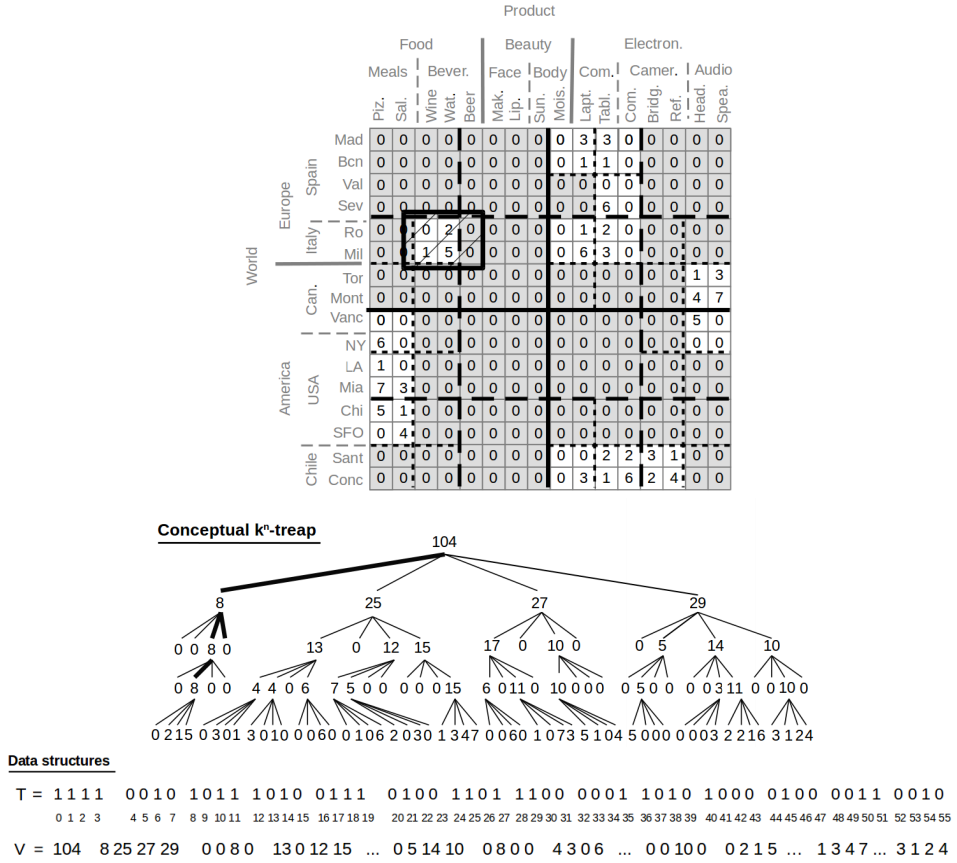


Figure 4.1: k^n-treap with a highlighted range query.

of an individual cell, and finding the sum of the values in a given range of cells, specified by the initial and final coordinates that define the submatrix of interest.

4.2.3.1 Finding the value of a specific cell by its coordinates

In order to find the value of the cell, the search starts at the root node and in each step goes down through the children of the matrix overlapping the searched cell until reaching the individual cell or an empty node. When the search process reaches an empty submatrix, then the query stops and returns the value 0.

For example, for finding the value of cell at coordinates (3,10) in Figure 4.1, which corresponds to *tablets sales in Seville*, the search goes through the second

child node (with value 25 in the figure). The correspond submatrix is not empty, so the process continues through the third child node (with value 13), then through its fourth child (with value 6) and finally through the third child, reaching the leaf node with value 6, which is the value returned by the query.

4.2.3.2 Finding the sum of the cells in a submatrix

The second type of query looks for the aggregated value of a range of cells, like the hatched area in Figure 4.1. This is implemented as a depth-first multi-branch traversal of the tree. If the algorithm finds that the range specified in the query fully contains a submatrix of the k^n-treap that has a precomputed sum, it will use this sum and will not descend to its child nodes. If the process reaches an empty node (with value 0), the process ends for that branch and continues with the remaining branches.

The figure highlights the branches of the k^n-treap that are used for searching the aggregated value of the hatched area (in the OLAP environment, this area corresponds to *sales of beverages in Italy*). Notice that this query completely includes the submatrix with values $\{0, 2, 1, 5\}$, which has its sums (8) explicitly stored at the fourth level of the tree. Therefore, the algorithm does not need to reach the leaf levels of the tree for this matrix. Notice also that there is an empty submatrix that intersects with the region of the query (the fourth child of the first child of the root), so the algorithm also stops before reaching the leaf level in this submatrix.

Chapter 5

Our proposal: CMHD

In this chapter, we present a new compact data structure to represent multidimensional data where the domains in each dimension are hierarchical. It will be called ***C**ompact representation of **M**ultidimensional data on **H**ierarchical **D**omains* (CMHD).

5.1 Our proposal: CMHD

CMHD divides the matrix following the natural hierarchy of the elements in each dimension. In this way, we allow the efficient answer of queries that consider the semantic of the dimensions. This structure is much more efficient than a generic multidimensional structure, since queries are resolved by aggregating much fewer nodes of the tree.

5.1.1 Conceptual description

Consider an n-dimensional matrix where each cell contains a weight (e.g., product sales, credit card movements, ad views, etc.). The CMHD recursively divides the matrix into several submatrices, taking into account the limits imposed by the hierarchy levels of each dimension.

Figure 5.1 depicts an example of a CMHD representation for two dimensions. The matrix records the number of product sales in different locations. For each dimension, a hierarchy of three levels is considered. In particular, cities are aggregated into countries and continents, while products are grouped into sections and good categories. The tree at the right side of the image shows the resulting conceptual CMHD for that matrix. Observe that each hierarchy level leads to an irregular partition of the grid into submatrices (each of them defined by the limits of its elements), having as associated value the sum of product sales of the individual cells

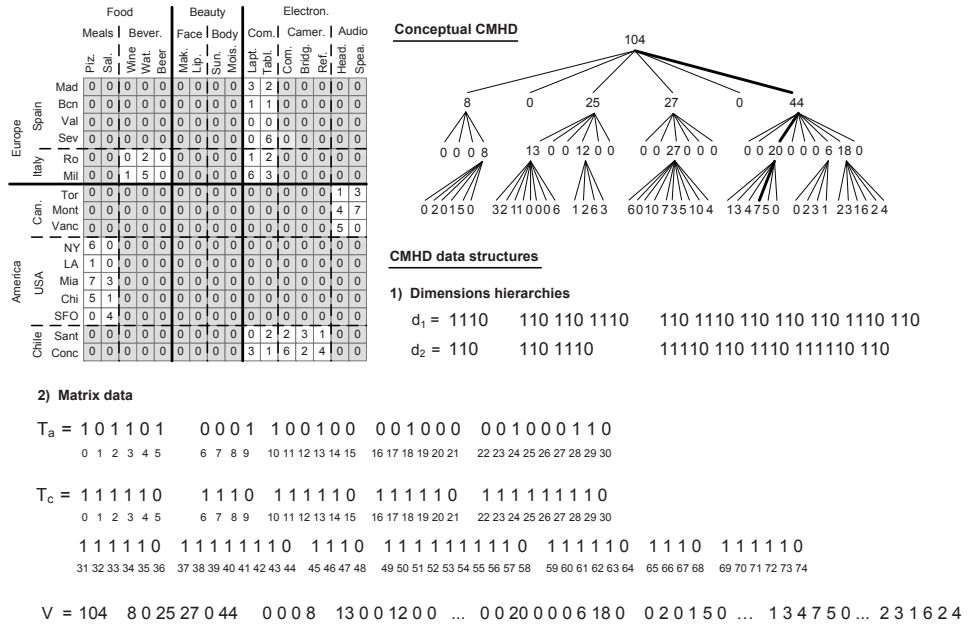


Figure 5.1: Example of CMHD construction for a two-dimensional matrix.

inside it. Thus, the root of the tree stores the total amount of sales in the complete matrix. Then the matrix is subdivided by considering the partition corresponding to the first level of the dimension hierarchies (see the bold lines). Each of the submatrices will become a child node of the root, keeping the sum of values of the cells in the corresponding submatrix. The decomposition procedure is repeated for each child, considering subsequent levels of the hierarchies (see the dotted lines), as explained, until reaching the last one. Also notice that, as happens in the k^n-treap, the decomposition concludes in all branches when empty submatrices are reached (that is, in this scenario, when a submatrix with no sales is found). See, for example, the second child of the root.

Note that CMHD assumes the same height in all the hierarchies that correspond to the different dimensions. Observe that, for each crossing of elements of the same level from different dimensions, an aggregate value is stored. Notice also that artificial levels can be easily added to a hierarchy of one dimension by subdividing all the elements of a level in just one element (itself), thus creating a new level identical to the previous one. This feature allows us to arbitrarily match the levels of the different hierarchies, and thus to flexibly adapt the representation of aggregated data to particular query needs. That is, by introducing artificial intermediate levels where required, more interesting aggregated values will be precomputed and stored. For

example, assume we have two dimensions: (d_1) with levels for *department*, *section* and *product*; and (d_2) with levels for *year*, *season*, *month* and *day*. If we were interested in obtaining the number of sales per *section* for *season*s, but also for *months*, we could devise a new level arrangement for d_1, which will have now the levels *department*, *section*, *section'*, *product*; where each particular *section* of the second hierarchy level results into just one *section'* child, which is actually itself. In this way, aggregated values will be computed and stored considering sales for *section* in each *season*, but also sales for *section'* in each *month*.

5.1.2 Data structures

The conceptual tree that defines the CMHD is represented compactly with different data structures, for the domain hierarchies and for the matrix data itself.

- *Domain hierarchy representation.* The hierarchy of a dimension domain is essentially a tree of C nodes. We represent this tree using LOUDS, which uses $2C - 1$ bits, and can be efficiently navigated, as we have seen in Section 2.2.3. Figure 5.1 illustrates the hierarchy encoding of the dimensions used in that example (see d_1 and d_2). For instance, the degree of the first node for the products dimension (d_1) is 3, so its unary encoding is 1110. Unlike the original LOUDS, we do not represent the last level (leaf nodes generate a single 0, so it would be formed by only 0s.) because we know, in advance, the number of levels of the tree and we can determine where the tree ends. Note that each node (i.e., element of a dimension placed at any level of its hierarchy) is associated with one 1 in the encoded representation of the degree of its parent. We also use a hash table to associate the domain nodes (i.e., labels such as "USA" in Figure 5.1) with the corresponding LOUDS node position. Therefore, the hash table will have an entry for each label in the hierarchy.

- *Data representation.* To represent the n-dimensional matrix, we use the following data structures:

  - *Tree structures (T_a and T_c)*: to navigate the CMHD, we need to use two different data structures in conjunction. First, T_a, a bitmap that, similarly to the k^n-treap, provides a compact representation of the conceptual tree independently of the node values, for all the tree levels, except the last one[1]. That is, internal nodes whose associated value is greater than 0, will be represented with a 1. In other case, they will be labeled with a 0. Observe that, for the k^n-treap, the use of this data structure is enough to navigate the tree, taking advantage of the regular partition of the matrix into equal-sized submatrices. Instead, CMHD follows different

[1] We do not actually need to represent the nodes of the last level in T_a. This data structure will be used to first identify a node whose children will be later located in another bitmap (T_c). But these already constitute matrix cells, with no children.

hierarchy partitions, which results into irregular submatrices. Therefore, a second data structure, T_c, is also required to traverse the CMHD. This is a bitmap aligned to T_a, which marks the limits of each tree node in T_a (this time, it also considers the last tree level). If the next tree node in T_a has z children, we append $1^{z-1}0$ to T_c. Notice that each node of T_a is associated with a 0 in T_c, which allows navigating the trees using *rank* and *select* on T_a and T_c: say we are at a node in T_a that starts at position i; then it has a kth child iff $T_a[i+k-1] = 1$, and if so this child starts at position $select_0(T_c, rank_1(T_a, i+k-1)) + 1$.

– *Values (V)*: the CMHD is traversed levelwise storing the values associated with each node (either corresponding to original matrix cells, or to data aggregations) in a single sequence, which is then represented with DACs.

5.1.3 Queries

Queries in this context give the names of elements of the different dimensions and ask for the sum of the cells defined for those values. Depending on the query, we can answer it by just reporting a single aggregated value already kept in V, or by retrieving several stored values, and then adding them up. The first scenario arises when the elements (labels) of the different dimensions specified in the query are all at the same level in their respective hierarchies. The second situation arises from queries using labels of different levels. In both contexts, top-down traversals of the conceptual CMHD are required to fetch the values. The algorithm always starts searching the hash tables for the labels provided by the query for the different dimensions, to locate the corresponding LOUDS nodes. From the LOUDS nodes, we traverse each hierarchy upwards to find out its depth and the child that must be followed at each level to reach it.

This information is then used to find the desired nodes in T_a. For example, with two dimensions, we start at the root of T_a and descend to the child number $k_1 + a_1 \cdot k_2$, where k_i is the child that must be followed in the ith dimension to reach the queried node, and a_i is the number of children of the root in the ith dimension (a_i is easily computed with the LOUDS tree of its dimension). We continue similarly to the node at level 2, and so on, until we reach one of the query nodes in a dimension, say in the first. Now, to reach the other (deeper) node in the second dimension, we must descend by every child in the first dimension, at every level, until reaching the second queried node. Finally, when we have reached all the nodes, we collect and sum up the corresponding values from V. Note that, if all the queried nodes are at the same level, we perform a single traversal in T_a. Note also that, if we find any zero in a node of T_a along this traversal, we immediately prune that branch, as the submatrix contains no data.

For example, we want to retrieve the total amount of *speaker* sales in *Montreal*, in Figure 5.1. Since both labels belong to the same level in both dimension hierarchies

(the last one), we will have to retrieve a single stored value in that level. The path to reach it has been highlighted in the conceptual tree of the image. To perform the navigation, we must start at the root of the tree (position 0 in T_a). In the first level, we need to fetch the sixth child (offset 5), as it corresponds to the submatrix including the element to search, in that level. Hence we access position 5 in T_a. Since $T_a[5] = 1$, we must continue descending to the next level. Recall that we have a 1 in T_a for each node with children, and that each node is associated with just one 0 in T_c. So the child starts at position $select_0(T_c, rank_1(T_a, 5)) + 1 = select_0(T_c, 4) + 1 = 22$ in T_a. In this level, we must access the third child (offset 2), so we check $T_a[24] = 1$. Again, as we are in an internal node, we know that its children are located at position $select_0(T_c, rank_1(T_a, 24)) + 1 = select_0(T_c, 9) + 1 = 59$. Finally, we reach the third and last level of the tree, where we know that the corresponding child is the fourth one (at $T_a[59 + 3] = T_a[62]$). Recall, however, that this last level is not represented in T_a. To perform this final step, we look directly in array V: $V[62 + 1] = V[63] = 7$ is the answer.

In case of queries combining labels of different levels, the same procedure would apply, but having to get the values corresponding to all the possible combinations with the element of the lowest hierarchy level (e.g., if we want to obtain the number of *meal* sales in *America*, we must first recover the values associated with *meal-Canada*, *meal-USA*, and *meal-Chile*, and then sum them up).

Chapter 6

Experimental evaluation

In this chapter, we analyze empirically the behavior of our proposed data structure, the CMHD. As baseline, we use the k^n-treap, presented in Section 4.2. Both representations have been implemented in C/C++, and the compiler used was GCC 4.6.1. (option -O9). We ran our experiments in a dedicated Intel(R) Core(TM) i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache, and 64GB of RAM. The machine runs Ubuntu 12.04.5 LTS with kernel 3.2.0-99 (64 bits).

6.1 Datasets

We generate different datasets, all of them synthetic, to evaluate the performance of the two data structures, varying the number of dimensions and the number of items for each dimension. These datasets have been labeled as `<dim#>D_<item#>`, thus referring to their size specifications in the own name. For example, dataset `5D_16` has 5 dimensions, and the number of items on each dimension is 16. The total size of this dataset is $16^5 = 1048576$ elements.

In order to show the CMHD advantage of considering the domain semantics, and computing the aggregate values according to the natural limits imposed by the hierarchy of elements in each dimension, the dimensions hierarchies have been generated in two different ways for each dataset. First, the *binary* organization, which corresponds to a regular partition. That is, the hierarchies of each dimension are exactly the same as those produced by a k^n-treap matrix partition into equal-sized submatrices. In this way, both data structures store exactly the same aggregated values. We named it *binary* because we use a value of $k = 2$. Second, the *irregular* organization, which arbitrarily groups data, on each dimension, into different and irregular hierarchies (different number of divisions, and also different size at each level). The last scenario simulates what would be a matrix partition following the semantic needs of a given domain. In this case, the aggregated values stored by

the CMHD will be different from those stored by the k^n-treap, and therefore more appropriated to answer queries using the same "semantic", which means, in our context, queries considering regions that exactly match the natural divisions of each dimension at some level of the hierarchies.

To test the structures behavior, we have also considered four different datasets, with a different number of empty cells, for each size specification: with no empty cells, and with 25%, 50% and 75% of empty cells, respectively.

6.2 Space requirements

First we analyze the space requirements of both data structures for all the datasets (see Table 6.1). Of course, the size decreases as the number of empty cells increases, in both cases. Moreover, we can also observe that the k^n-treap size is slightly lower than the CMHD size. This is expected, because CMHD has to store the LOUDS representation of each dimension hierarchy, while dimensions are implicit for the k^n-treap. Additionally, CMHD uses a second bitmap (T_c) to navigate the conceptual tree, which is not necessary when using the k^n-treap.

We must also clarify a small issue about the sizes of the k^n-treaps: the size of a standard k^n-treap for a specific dataset is always the same, regardless of the organization of its dimensions (binary or irregular). However, Table 6.1 shows some difference in the sizes. For example, for `4D_32`, the size for the binary organization is 588.89, but it is 588.61 for the irregular one. The reason for this variation is that all queries are performed by taking dimension labels as input, so we need a vocabulary to translate each label into a range of cells. We have included that vocabulary (dimension labels and cell ranges) into the size of the k^n-treaps, and the vocabulary for the irregular organization is usually smaller, as it has less levels and less dimension labels (because each node in the conceptual tree can have more than two children in the irregular organization, whereas the binary organization always has two).

6.3 Query times

Regarding query times, we have run several sets of queries for all the datasets. As previously mentioned, queries are posed in this context by giving one element name (label) for each different dimension, as it is the natural way to query a multidimensional matrix defined by hierarchical dimensions. Since the k^n-treap does not directly work with labels, each query has been translated into the equivalent ranged query, establishing the initial and final coordinates for each dimension. The following types of queries have been considered:

- *Finding one precomputed value.* This value can be a specific cell of the matrix (so forcing the algorithms to reach the last level of the tree), or a precomputed

Table 6.1: Space requirements of k^n-treap and CMHD data structures (in KB) for synthetic datasets.

	0% Zeroes				25% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	39.54	47.13	39.54	45.65	34.49	40.77	34.49	40.43
4D_32	588.89	729.39	588.61	689.74	483.86	597.66	483.57	590.15
5D_16	553.79	690.81	553.79	673.41	488.54	608.51	488.54	612.64
5D_32	17608.33	22044.29	17607.79	21205.93	14589.71	18248.49	14589.18	18056.06
6D_16	8526.21	10708.68	8526.26	10529.12	7520.25	9440.07	7520.29	9446.53

	50% Zeroes				75% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	28.25	32.96	28.25	33.35	18.75	21.57	18.75	21.79
4D_32	366.12	450.67	365.84	452.93	232.11	287.18	231.83	295.70
5D_16	388.39	483.51	388.39	496.62	241.62	308.70	241.62	324.82
5D_32	10949.27	13688.31	10948.73	14381.19	6907.18	8783.36	6906.64	9694.66
6D_16	6042.99	7621.09	6043.03	7797.71	3913.45	5124.84	3913.50	5383.60

value that corresponds to an internal node of the conceptual tree.

- *Finding the sum of several precomputed values.* This kind of query must obtain a sum that is not precomputed and stored in the data structure itself. In turn, it must access several of these aggregated values and then add them up. Given that we are specifying the queries by dimension labels, this type of query is defined by using labels that belong to different levels of the hierarchies across the dimensions. The lowest level, which corresponds to individual cells, is not used for this scenario.

Each created set contains $10,000$ randomly generated queries of the two previous types for each dataset. Tables 6.2–6.4 show the average query times (in microseconds per query) for both data structures, taking into account the two different matrix partitions of the datasets (*binary* or *irregular*) and also the percentage of empty cells.

6.3.1 Finding one precomputed values

We first show the results obtained for queries that just need to retrieve one precomputed value, at different levels. On the one hand, Table 6.2 displays query times for specific matrix cells, that is, located at the last level of the conceptual tree. In this case, the k^n-treap performs better than the CMHD in almost all cases. This is an expected outcome as both data structures must reach the leaf level to get the answer, and the depth first navigation of the tree is simpler in the k^n-treap (just

Table 6.2: Average query times (in μs) for queries finding one precomputed value (original matrix cells) for synthetic datasets.

	0% Zeroes				25% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	0.84	2.37	0.85	2.39	0.81	2.29	0.81	2.33
4D_32	1.12	3.18	1.12	3.23	1.06	3.02	1.06	3.11
5D_16	1.04	3.04	1.04	3.09	0.99	2.94	0.98	3.03
5D_32	1.37	4.14	1.37	4.19	1.31	3.93	1.32	4.06
6D_16	1.23	3.78	1.24	3.88	1.20	3.66	1.20	3.78

	50% Zeroes				75% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	0.76	2.18	0.77	2.26	0.72	2.03	0.72	2.08
4D_32	0.97	2.82	0.97	2.93	0.91	2.59	0.91	2.77
5D_16	0.98	2.83	0.99	2.93	0.87	2.63	0.85	2.73
5D_32	1.23	3.68	1.23	3.86	1.19	3.44	1.18	3.73
6D_16	1.20	3.56	1.21	3.72	1.10	3.33	1.11	3.46

products and *rank* operations). In any case, CMHD also performs quite well, using just a few microseconds to answer any of the queries.

On the other hand, Table 6.3 shows the average query times for queries of the same type, but now considering precomputed values stored in nodes of an intermediate level of the tree (in particular, the penultimate level). Note that this fact holds for both data structures when working with a regular partition of the matrix (that is, the *binary* scenario). Thus, in this case, the k^n-treap gets better results than CMHD, but with slight time differences. Yet, observe that this is not the actual scenario when dealing with meaningful application domains, where rich semantics arise. This situation is that corresponding to what we called *irregular* datasets. In this case, CMHD excels, as expected, given that this data structure has been particularly designed to manage hierarchical domains. Results show that CMHD is able to perform up to 4 times faster than k^n-treap (for the best case).

6.3.2 Finding the sum of several precomputed values

Finally, Table 6.4 presents the average query times for the second type of queries (that is, those having to recover several precomputed values and then adding them up to provide the final answer). As results show, the k^n-treap displays a better performance than CMHD for the *binary* scenario. However, again this is not the most interesting situation in real domains. If we observe the results obtained for the *irregular* datasets, we will appreciate that CMHD clearly outperforms the k^n-treap in this scenario, thus demonstrating the good capabilities of our proposal to cope with the aim of this work.

Table 6.3: Average query times (in μs) for queries finding one precomputed value (penultimate tree level) for synthetic datasets.

	0% Zeroes				25% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	0.67	1.82	2.53	1.79	0.67	1.78	2,23	1.83
4D_32	0.93	2.49	4.47	2.48	0.91	2.44	3,87	2.50
5D_16	0.81	2.27	5.72	2.32	0.82	2.26	5.05	2.32
5D_32	1.14	3.20	10.87	3.13	1.15	3.10	9.38	3.15
6D_16	0.96	2.78	13.42	2.81	0.97	2.77	11.89	2.80

	50% Zeroes				75% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	0.70	1.77	1.87	1.84	0.67	1.72	1.73	1.73
4D_32	0.89	2.36	3.24	2.42	0.84	2.24	2.83	2.37
5D_16	0.83	2.26	5.44	2.31	0.78	2.16	3.94	2.22
5D_32	1.11	3.04	7.43	3.12	1.07	2.95	6.73	3.08
6D_16	0.99	2.77	12.96	2.84	0.93	2.68	9.62	2.70

Table 6.4: Average query times (in μs) for queries finding a sum of precomputed value.

	0% Zeroes				25% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	1.76	6.42	10.52	6.70	1.74	6.29	9.16	6.82
4D_32	3.63	14.20	34.89	13.44	3.47	13.42	29.67	13.58
5D_16	2.07	7.73	28.17	8.27	2.08	7.69	24.41	8.36
5D_32	4.30	16.95	98.54	14.91	4.16	16.13	82.80	15.16
6D_16	2.45	9.16	73.32	9.98	2.41	9.01	64.13	9.86

	50% Zeroes				75% Zeroes			
	Binary		Irregular		Binary		Irregular	
name	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD	k^n-treap	CMHD
4D_16	1.92	6.26	7.39	6.81	1.71	5.68	6.72	6.16
4D_32	3.60	12.54	23.30	12.73	3.12	11.01	20.23	11.76
5D_16	2.31	7.66	26.90	8.20	2.03	6.88	18.47	7.48
5D_32	4.38	15.47	62.90	14.77	3.92	13.96	56.52	14.11
6D_16	2.67	9.04	71.32	10.14	2.42	8.40	52.02	9.16

Chapter 7

Discussion

7.1 Main contributions

We have presented a multidimensional compact data structure that is tailored to perform aggregate queries on data cubes over hierarchical domains, rather than general range queries. The structure represents each hierarchy with a succinct tree representation, and then partitions the data cube according to the product of the hierarchies. This partition is represented with an extension of the k^2-treap to higher dimensions and to non-regular partitions. The resulting structure, dubbed CMHD, is much faster than a regular multidimensional k^2-treap when the queries follow the hierarchical domains. This makes it particularly attractive to represent OLAP data cubes compactly and efficiently answer meaningful aggregate queries.

7.2 Future work

As future work, we plan to experiment on much larger collections. This would make the vocabulary of hierarchy nodes much less significant compared to the data itself (especially for the CMHD). We also plan to test real datasets (for example, coming from data warehouses) and real query workloads. We also expect to compare our results with established OLAP database management systems, and to enrich our prototype with other kinds of queries and data.

Part II

GIS data

Chapter 8

Introduction

In this Part II, we present a new compact data structure designed to represent raster datasets, called k^2-*raster*, and an improved version that we call *heuristic* k^2-*raster* (k^2_H-*raster*). This data structure is based on the k^2-tree, which has been previously described in Section 3.2. Unlike previous compact data structures designed to deal with raster data, our proposal exhibits good scalability when the number of different values in the raster increases, in addition to obtaining better memory consumption than the techniques of the literature. In addition, in this part, we also show an algorithm to perform a spatial join between a k^2-raster and a vector dataset.

The next chapters of this part are organized as follow: Chapter 8 introduces a brief description of geographic information systems, previous compact data structures to represent raster information, and a brief description of the spatial join between raster and vector datasets. Chapter 9 describes in detail the k^2-raster and its variations. Chapter 10 includes the definition and implementation of the spatial join between raster data, represented with k^2-raster, and vector data. In Chapter 11 we test our contributions with real data and the structures are compared with the current state of the art. Finally, we summarize the work presented in this part and propose new research lines in the Chapter 12.

This chapter is structured as follows: Section 8.1 introduce some previous concepts of interest for the rest of the thesis. The related work is presented in Section 8.2. Finally, we introduce the spatial join operation in Chapter 8.3.

8.1 Introduction

A geographic information system (GIS) is any computer system that lets us store, manipulate, analyze and display spatial or geographic data [WD04]. Nowadays, GIS tools have been established in many organizations for both commercial and research

use, generating and storing a multitude of geographic data. This means that more and more geographic data are made public and that it is easier to access them for our own analysis and manipulation. In addition, with the increase of devices with GPS and other location systems, and the popularity of their use in many applications, GIS systems need new methods and structures to store and process the data in a much more efficient way.

Research on efficient management of spatial information has produced several spatial data models. On the conceptual level, these models describe the space using two different approaches: *object-based spatial models* and *field-based spatial models* [WD04]. Considering the logical level, spatial data models can be divided into two categories: *vector models* that represent geographic information using finite sequences of points and line segments, and *raster models* that represent geographic information partitioning space into a finite grid of cells and assigning a value to each cell [LGMR05].

The first contribution of this part 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 [LB07], and the archived amount of raster data of this type is slowly approaching the zettabyte scale [QG13].

In this field, as usual, compression has been used to save space and bandwidth [Wal91, LI06]. There exists vast research focused on compressing raster datasets, proposing both lossless [HV14, SJS+12, ZYG15] and lossy [Wal91] approaches. In addition, there have been efforts in creating indexes on raster data to improve query and processing performance [Sam84, Duv09, ZY10]. 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 [Kli71, KD76], which was originally designed as a method to compress images. It 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 [dBÁGB+13] were designed to represent raster datasets and combined these three features: it compresses the data, indexes the space, and indexes 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. 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. Therefore, these previous approaches, which are described in Section 8.2, are not useful in many real

data scenarios.

This big increase in the variety, richness and amount of spatial data has also led to new information demands. Nowadays, many application areas require to combine data stored in different formats [GRS00]. Obviously, combining different data models efficiently becomes harder when huge amounts of data are involved.

Therefore, several challenges arise. First, at the conceptual and logical levels, new data models and query languages can be developed in order to accommodate those new information requirements [SH91, BDF+98, VZ09]. Second, at the physical level, new data structures and algorithms are needed to implement the aforementioned data models and query languages [Kai91, GG98].

Although there is a large body of research regarding those three problems, in most cases that research is focused either in the vector model or in the raster model separately. The two models are rarely handled together. For instance, the usual solution for queries that involve (together) raster and vector datasets is to transform the vector dataset into a raster dataset, and then use a raster algorithm to solve the query. This is the solution for the zonal statistics operation of Map Algebra in, at least, ArcGIS and GRASS [Zon16, GRA16].

However, some previous works focus on a joint approach. In [GRS00], for example, a single data model and language is proposed to represent and query both vector and raster data at the logical level. Even a *Join* operator is suggested, such that it allows to combine, transparently and interchangeably, vector datasets, raster datasets, or both. As an example, the authors propose the query "return the coordinates of the trajectory of an aircraft when it was over a ground with altitude over 1,000". Unfortunately, no implementation details are given.

Other previous contributions deal with the implementation of query operators explicitly defined to query datasets in different formats [CVM99, CTVM08, BdBG+17]. All those contributions include the definition of a *Join* operator between a vector and a raster dataset. Unfortunately, all of them suffer from serious limitations (data structures not complex enough, too restrictive join operations, size problems) that will be explained more in detail in the Section 8.3.

8.1.1 Data model

Geographical information systems can use different data models to manage spatial information [Cou92]. At the conceptual level, there are two possibilities: *object-based models* and *field-based models*.

- *Object-based models* describe a space containing discrete and identifiable entities (objects), each with a geospatial position and well-defined boundaries. It is useful to represent geographic information about objects. Typically, buildings, roads, rivers and other man-made objects are represented with *object-based*

models. The typical operations involve the topological relationship between their objects, for example, if one object intersects with another one.

- *Field-based models* can be seen as a continuous mathematical function that for each position of the space returns a value. It represents a distribution of spatial attributes or phenomenas over a space whose value depends on its spatial position, such as land elevation, temperature, atmospheric pressure, etc. Some basic operations are to obtain the minimum, maximum or average value of an area.

Object-based models and *field-based models* are used for different proposes. The first one is focused on the specific form of the object and its location in the space whilst *field-based models* are suitable for attributes of the space.

The two previous models cannot be used for the digital representation of the space, since they only define the data in a conceptual way and not as the computer processes that information. The logical model allows us to represent the conceptual models in a system. From the logical level point of view there are also two types of models: *vector models* and *raster models*.

- *Vector models* represent the spatial information using points and line segments. A point is defined by its coordinate in the space, whereas for a segment a starting and an ending points are needed. Other more complex representations, such as curves or polygons, are composed of sets of connected segments, so these objects can be defined by a set of points.

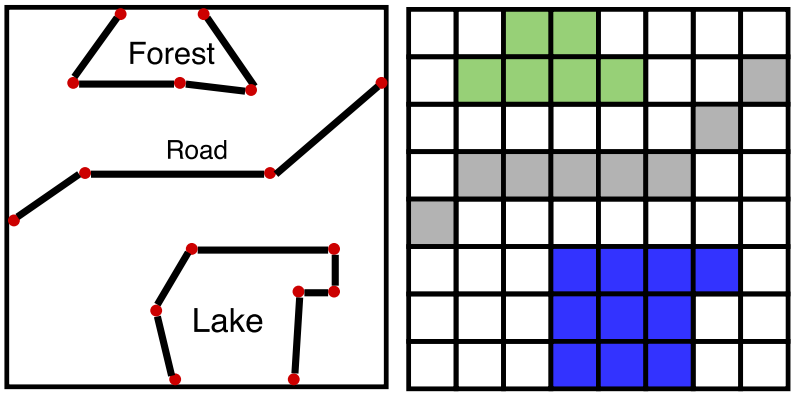
- *Raster models* consider the space as a regular tessellation of disjoint cells, usually squares, each having a value [LGMR05]. Given that any image can be seen as a raster, the use of this data model is massive. Other examples of application of raster datasets could be pollution control, weather forecast, satellite imagery, remote sensing capture, 3D modeling, engineering, etc.

In theory, any logical spatial model can be used to represent any conceptual spatial model, however, in practice it is common to use *vector models* to represent *object-based models* and *raster models* for *field-based models*.

Figure 8.1 shows an example of two representations of the same data with two different models. In Figure 8.1(a) we use the vector model. The lake and the forest are described with two polygons and the road with a line. The output of this model is a set of points for each *object* in the space. In the other Figure 8.1(b), we use the raster mode, dividing the space into cells and assigning one value for each one. In this example, the blue cells mark where the lake is, the green cells indicate the location of the forest, whilst the gray cells define the route of a road.

8.1.1.1 Representation of raster data

The simplest way to represent raster data is with a binary matrix, which determines the existence or not of a spatial feature. Therefore, each cell of the matrix contains



(a) Vector model (b) Raster model

Figure 8.1: Example of a vector model and a raster model for the same data.

the value 1 or 0 depending on whether the corresponding area contains the feature or not. For example, clouds or location of an oil spot on the sea can be represented with a binary matrix. But it is not enough to some kind of spatial features, such the temperature, where we need to keep one value per cell (not only 1s and 0s). For these cases, a general raster is used, where each cell stores a value with a predetermined precision (32-bit or 64-bit integers, floating-points values, etc).

8.1.1.2 Classic formats

Several formats were proposed for representing raster data:

- *GeoTIFF* is a public standard that embeds geographic information (such as the map projection or the coordinate systems used) to TIFF (Tagged Image File Format) files. The GeoTIFF specification defines a set of new tags to describe cartographic and geographical information. It divides the image into separated tiles or strips that support several compression methods such as LZW or RLE (Run-Length encoding).

- *NetCDF* (Network Common Data Form) is a standard proposed by the Open Geospatial Consortium (OGC) for geospatial data. The file contains a metadata header that includes a descriptor of the data and the temporal and spatial properties of the data stored. NetCDF can compress the data using DEFLATE.

- *Esri grid* is a file format designed for representing raster data created by Esri organization. It has two formats, a binary format and an ASCII format. The

binary format is used by Esri programs, as ArcGIS, and divides the raster data in rectangular tiles. Each tile is independently stored and compressed using RLE (Run-Length encoding). On the other hand, the ASCII format stores the data in plain text and is widely used to export and share a raster file. A header is included with basic information of the rater (number of row and columns, max/min value, etc).

Another way to store raster data is to use general image file formats as JPEG or PNG. These formats are only able to represent the values of the raster, therefore, it is necessary to keep the spatial information separately (for example, as metadata).

8.2 Related work

8.2.1 Quadtrees for raster data

There are many different variants of the quadtree and with different purposes [Sam84, Sam06], but the compression of images using region quadtrees was one of its original targets [Kli71, KD76]. 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 [Sam84]. 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 [Sam85, OW83]. These pointerless representations use a *locational code* that for each leaf of the tree gives its position in the space [Sam85] or an implicit ordering [OW83]. 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 Figure 8.2 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 compact structure also uses an implicit ordering using a breadth-first traversal, but we separate the topology of the tree (the most significant bit in the 5-bit number) from the content (the remaining 4 bits). Thanks to this split,

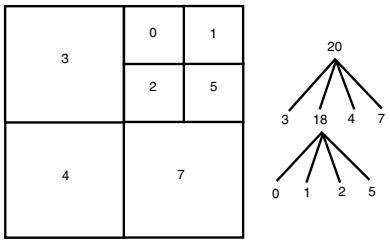


Figure 8.2: An image (left), where a number inside a square means that all pixels in that square have that value, and the corresponding conceptual quadtree showing the byte representation of each node using the Treecodes strategy (right).

we can use more appropriate methods to represent these two types of information. The topology (the bit indicating whether a node is a leaf or not) is represented with a k^2-tree, which uses 1 bit per node and it is a very efficient structure to navigate. With respect to the content, the first difference is that, in non-leaf nodes, instead of storing the average value of the corresponding subquadrant (only useful for pre-visualization purposes), we store the min-max values to index the values at the cells of the raster. The second difference is that the cell values (corresponding to leaves) and the min-max values of inner levels are stored using DACs and differential encoding, which can achieve good compression and allow fast access times.

As explained, the quadtree has been used with different purposes, although the use of the quadtree to compress rasters (including images) [Woo84, Lin97, CC03, CLY06, ZYG15] is one of the main research lines. Another use of the quadtree is to index rasters, although much less effort was devoted to this feature [ZYG10, ZY13]. The best known example is the linear quadtree, which is a disk-resident index for rasters [RSV02].

The region quadtree indexes the space allowing spatial searches, however in [ZYG10], the inner nodes of a quadtree, called Binned Min-Max Quadtree, are enriched with the min-max values of the region represented by such a node, thus indexing the values of the raster dataset as well. However, there are several differences with respect our work. First, the values stored at the leaf nodes and the corresponding min-max values at inner nodes are not the actual values in the raster. They use a binned or histogram strategy, which consists in assigning a code to ranges of possible values, for example, 0 encodes the values between -50 and -10, 1 encodes the values between -9 and 0, and so on. Then to perform searches, first we have to encode our search value, and then use that encoded value to take decisions at the nodes of the quadtree. This implies that the quadtree is simply a classical index and thus we have to store the original raster separated from the quadtree, using a classical representation, that is, the quadtree is an auxiliary structure of the main data file. In addition, the quadtree with binned codes limits the pruning capacity of the tree

to the boundaries of the ranges defined by the binned strategy. Finally, this previous work is mainly focused on search capacity, and thus there are no worries about space, using a naive pointer-based implementation. Later, the same team presented a new data structure, called Cache Conscious Quadtree [ZYG10], which is a quadtree where all nodes are placed in a one-dimensional array to avoid non-continuous memory allocations, in order to improve constructions times. Each node has a field indicating the position of its first child in the array and the min-max values. It uses again a binned strategy, and thus, it is just an auxiliary index.

As a summary, we can point out that the main difference of our approach with respect to these works is that while these works are either focused on representing the raster using compression or on designing an auxiliary index of the raster data, our work focuses on joining these two worlds. We present a structure for representing the raster data in a compressed form, and at the same time, it indexes both the space and the values of cells. Thus, our proposal can be consider a self-index for raster data.

8.2.2 k^2-acc

Another way to represent a raster having values in the range $v_1 < v_2 < \cdots < v_t$ is to use a k^2-tree for each value. Then, the representation is formed by t k^2-trees $K_1, K_2, \ldots, K_t$, where each K_i has a value 1 in those cells whose value is $v \leq v_i$ in the original raster. Observe that the k^2-trees corresponding to the bigger values (those close to v_t) will have large areas full of 1s, therefore the variant of the k^2-tree that compresses also the areas full of 1s is used. In fact, the k^2-tree corresponding to the largest number can be omitted, K_t, as it always represents a matrix full of ones. This approach is called *accumulated k^2-trees* or *k^2-acc*. This is the first compact data structure able to represent raster data and capable of indexing the space and the values stored at the cells.

To obtain the value at a given cell, a binary search over the collection $K_1, K_2, \ldots, K_t$ is needed. This approach is very efficient returning the cells having a value in a given range $[v_b, v_e]$, since it only needs to check K_b and K_e. To obtain the cells having a given value is also solved accessing two k^2-trees.

Figure 8.3 illustrates a raster data (top) and the resulting k^2-acc representation (bottom). This raster has 5 different values, therefore the k^2-acc representation needs four k^2-trees. In this example, the k^2-tree labeled as "K_1" has a one in cells whose original value is 1 or less. Then, cells of the next k^2-tree labeled as "K_2" contains a one if the original value is 2 or less, and so on. The final structure is a set of 4 k^2-trees.

8.2.3 k^3-tree

Another compact data structure for raster datasets is the *k^3-tree*, which is an extension of the k^2-tree where we add one extra dimension. It stores a binary cube

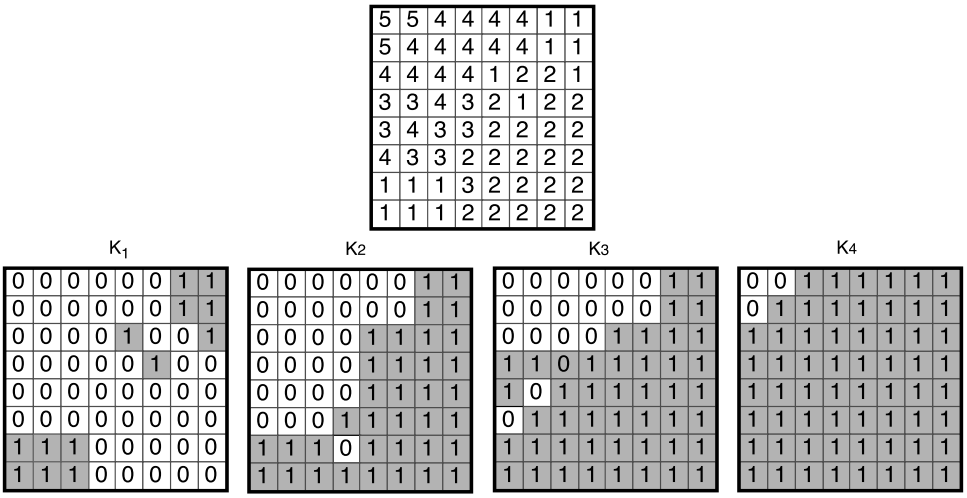


Figure 8.3: Example of raster matrix (top) and resulting k^2-acc representation (bottom).

using the same partitioning and representation strategies used in the k^2-tree. When applied to raster datasets, the k^3-tree stores points $\langle x, y, z \rangle$, where the first two values represent the position in the 2D space, and the third component is the value stored in that cell.

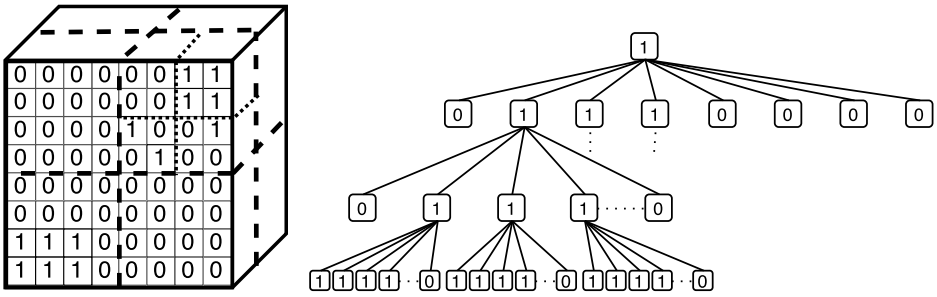
It is possible to efficiently navigate the k^3-tree, basically using the same procedures used in the k^2-tree, but extended to three dimensions. If we want to obtain the value stored at a given position, we just fix that position in the 2D space (x and y) and then we check the corresponding z value. To obtain the cells with a given value or a range of values, we fix the value(s) of z, and we search the values of (x, y) having the given value(s).

An example of k^3-tree is shown in Figure 8.4. The conceptual k^3-tree cube (Figure 8.4(a)) represents a raster dataset as seen before. The corresponding tree is shown in Figure 8.4(b).

Comparing k^2-acc and k^3-tree, the first one is better for retrieving cells containing a given value or range of values, whereas the k^3-tree obtains better space consumption and time results when retrieving the value at a given position.

8.2.4 *R*-tree

An *R*-tree [Gut84] is a tree data structure, similar to *B*-tree, for indexing multi-dimensional information and speeding up queries over a vector dataset. *R*-tree groups spatial objects (lines, polygons or point) using minimum bounding rectangles (MBRs), that is, the smallest n-dimensional rectangle that contains those objects.

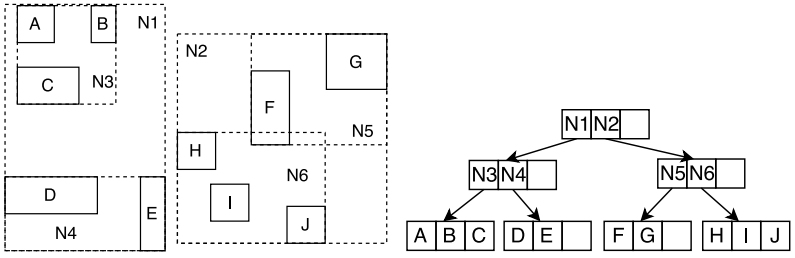


(a) k^3-tree cube (b) k^3-tree tree

Figure 8.4: Example of the k^3-tree decomposition, with $k = 2$.

The R-tree is dynamic, that is, objects can be inserted and removed from the tree at any time.

Each node of the R-tree is stored in a disk page and its capacity is determined by the page size (upper bound) and the space utilization of the tree (lower bound). In the case of leaf nodes, their entries hold information about the indexed object. Each entry usually stores a unique identifier object (called OID) and the MBR that encloses the object. On the other hand, internal nodes contains pointers to other nodes. Each entry of an internal node stores, in addition to the pointer to the child, the corresponding MBR of the child node.



(a) Objects (b) R-Tree

Figure 8.5: R-tree of 3 levels with 10 objects indexed.

Figure 8.5 shows an R-tree of 3 levels and 3 entries as maximum capacity of the nodes that indexes 10 different objects. The MBRs $A, \ldots, J$ enclose the 10 objects and correspond with the leaf nodes of the R-tree. Internal nodes contain other MBRs of the tree. For example, MBR $N3$ have 3 entries (A, B and C) and is the minimum rectangle that contains those 3 entries. Also observed that MBRs $N5$ and $N6$ overlaps.

Given a query defined by a rectangle, the search process performs a top-down traversal of the tree. It starts at the root node and checks its entries to determine if they intersect with the given rectangle. The process continues for each node that intersects with the query and repeats the same steps. If the process reaches a leaf node, each entry of the leaf node whose MBR intersects with the rectangle is added to a list of candidates. Finally, when the process ends, we need to check the actual geometry of each object in the list of candidates to determinate if it actually intersects with the query.

Others variants of the R-tree were presented to improve some characteristics. Beckmann et al. [BKSS90] propose the R^*-tree, which obtains better query performance than the standard R-tree. Nascimento et al. [NST99] introduce the HR-tree, a data structure for the representation of moving objects based on the R-tree. Other example is the $MV3R$-tree described by Tao and Papadias [TP01].

8.3 Spatial join

Few data models consider the possibility of using both the object-based and the field-based spatial models of space. Even international standards separate both views [ISO03, ISO05]. The same situation can be found at the logical level, where international standards [OGC10, OGC12] separate again both views and do not provide languages, data structures and algorithms to perform queries that use information from both data models simultaneously.

Many real systems supporting raster datasets adopt the operations of *Map Algebra* [TB79, Tom90]. In some systems, as for example ArcGIS, QGis, or GRASS, the zonal statistics operation of Map Algebra admits as one of its operands a vector dataset. However, as explained, at least in ArcGIS and in GRASS, the vector dataset is converted to raster before running the query [Zon16, GRA16].

Some research works do provide models and languages that include data types and operators to represent and query vector and raster data in traditional database models [SH91, BDF+98, VZ09]. These models specify two different sets of operations, one for vector data and another one for raster data.

In [GRS00], a single data model and language is used to represent vector and raster data. It includes spatial versions of the relational model operations like *Projection*, *Selection*, and *Join*. These operations can manipulate vector and raster information without having to separate or distinguish the type of operands. In [Bro10], it is presented a model based on multidimensional arrays to manage scientific data that is able to manage vector and raster data.

However, these proposals only describe data types and operators to process raster information, and, in some cases, to manipulate jointly raster and vector information, yet no details of implementation issues are provided (including neither the structures nor the algorithms needed to support the model and the queries).

Corral et al. [CVM99] presented five algorithms for processing a join between an R-tree and a linear region quadtree [Gar82]. In [CTVM08], it is shown the predictive join between regions and moving objects. It uses again a linear region quadtree for the raster information, and for the moving objects it uses the variation of the R-tree called TPR*-tree. Unfortunately, both works deal with binary rasters, and therefore they have a very limited real application.

Recently, Brisaboa et al. [BdBG+17] presented a framework to store and manage compressed vector and raster data, as well as an algorithm to solve a query that, given a vector and a raster dataset, returns the minimum bounding rectangles (MBRs) of the vector dataset overlapping regions of the raster that fulfill a range constraint over the raster dataset. For example, having a vector dataset representing the neighborhoods of a city and a raster storing the amount of nitrogen oxides in the air, a query could be "return the neighborhoods overlapping points where the concentration of nitrogen oxide is above 0.053 ppm". However, their solution does not return the exact cells of the raster fulfilling the range constraint, as the output can only include values of the vector dataset. The vector dataset is indexed with an R-tree and the raster dataset is represented and indexed with a k^2-acc. To solve the query, the algorithm just requires the k^2-trees representing the values at the extremes of the range and the R-tree. The search starts at the root of the three trees and then proceeds in parallel through the three trees using a top-down traversal, pruning the branches of the trees when possible. The k^2-acc has two problems. It works well for range queries, that is, those specifying a range of values of the raster dataset (like the nitrogen oxide example just exposed). However, it obtains modest response times for other queries like, for example, obtaining the value of a given cell. The other problem of the k^2-acc concerns the size of the dataset. It is a compact data structure, and gives good compression rates when the number of different values in the dataset is low. But when the number of different values is large, the dataset occupies much more space than the uncompressed representation, as we will see in Chapter 11.

Chapter 9

Our proposal: k^2-*raster*

In this chapter, we present the k^2-*raster*, a new compact data structure that represents raster datasets in compressed space, and at the same time, indexes the space and the values stored at cells. In Section 9.1, we describe the technique in detail, including algorithms that support several queries over the raster data. In Section 9.2, we propose an improvement of the technique, the *heuristic* k^2-*raster*, or k^2_H-*raster*.

9.1 k^2-raster

Let M be a raster matrix of size $n \times n$, being n a power of k, where each cell M_{ij} stores a value $v \geq 0$. [1]

The k^2-raster uses the same partitioning strategy used by the original k^2-tree, that is, it recursively divides M into k^2 submatrices, and builds a tree representing this recursive subdivision. In k^2-raster, the recursive division stops when all the cells in a submatrix have the same value. The nodes of the tree store the minimum and the maximum values of the corresponding submatrix in order to index the values at cells. Therefore, the k^2-raster puts together the quadtree spatial index, the min/max indexing of rasters, and a compressed representation of the data. As explained, the k^2-raster joins in a unique data structure two desirable properties: indexing capabilities and an efficient representation of the values in the cells and the values at the nodes, making our approach a compact and self-indexed representation.

[1] In case that the input matrix is of size $n \times m$, being n and m any integer, we conceptually extend the input matrix to the right and to the bottom, making it of size $n' \times n'$ such that $n' = k^{\lceil \log_k \max\{n,m\} \rceil}$, that is, we round n and m up to the next power of k of their maximum value. This does not cause a significant overhead because our technique effectively compresses large areas of equal values.

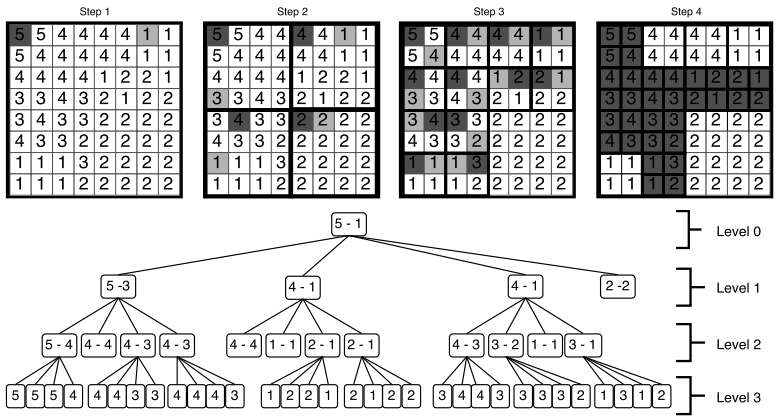


Figure 9.1: Example of raster matrix (top). We indicate the minimum (light gray) and maximum (dark gray) value of each submatrix for the four steps of the recursive subdivision of the construction algorithm, using $k = 2$. Conceptual tree representation obtained from the construction of the k^2-raster (bottom). Numbers at each node indicate the maximum and minimum value of its corresponding submatrix. In the last level, only the maximum is shown.

9.1.1 Construction and data structures

The first step of the construction process is the creation of the root node that stores the minimum and maximum values $(rMin, rMax)$ of the complete matrix. If $rMin$ and $rMax$ are equal, only one value is stored as the maximum, and the process ends here. Otherwise, the two values are stored and the matrix is divided into k^2 submatrices, each adding a child node to the parent, in this case, to the root node. For each generated submatrix, the process is recursively repeated, until the maximum and minimum values become equal, or until the decomposition reaches the last level, that is, when the decomposition of a submatrix obtains submatrices of just one cell. Observe that, being $n \times n$ the size of the matrix, the tree has a height of at most $h = \lceil \log_k n \rceil$ levels.

Figure 9.1 shows an example that illustrates the process. Under the label "Step 1", we can see an 8×8 raster matrix, and below it, the corresponding k^2-raster using $k = 2$. The root node stores the maximum and minimum values of the matrix, since these values are different, the 8×8 raster matrix is divided into 4 submatrices of 4×4 cells. Under the label "Step 2", we can see those submatrices and their maximum (marked in dark gray) and the minimum (marked in light gray) values. For each subdivision, one child node is added to the root node of the tree, storing those values. Observe that in the case of the bottom-right submatrix, the maximum

and the minimum values are the same (2), therefore such node becomes a leaf node and its corresponding matrix is not further subdivided. The other three submatrices are then subdivided, shown under label "Step 3", each displaying its maximum and minimum values. The level 2 contains the nodes corresponding to those submatrices, and again, those containing only one value produce a leaf node, and the rest are further subdivided. Finally, at level 3, the process reaches the cell level, and thus, for each subdivision, its node has one child for each of its cells, storing the value of that cell.

The previous description is a high level description of the k^2-raster. The actual representation uses several *succinct data structures* strategies to obtain compression. More specifically, we represent the topology of the tree and the maximum and minimum values, which make up the k^2-raster, as follows:

- The *topology of the tree* is stored separately from the rest of the information. For this sake, k^2-raster uses a data structure similar to that of a k^2-tree. In contrast to the original k^2-tree, a 0 in a node of a k^2-raster means that all values in the corresponding submatrix are equal, and a 1 means that there are two or more different values. In addition, while the original k^2-tree is divided into two bitmaps T and L, where L represents nodes of the last level and T the rest of nodes of the tree, k^2-raster does not need bitmap L because it would be completely composed of 0s, since the maximum and minimum values of a leaf node will always be equal, as there is just one value at those nodes.

- The *maximum values* of the nodes of the tree are also treated separately. In order to save space, all values, except the value of the root, are encoded as the difference with respect to the maximum value of their parent nodes. Observe that those differences will never be a negative value because the maximum value of a parent node is always equal or greater than the maximum value of its children. We obtain a tree composed of differences, which are stored as a unique array, denoted $Lmax$, where the positions of the values are determined by the breadth-first traversal of that tree. That sequence is composed of differences, which tend to be small, which is precisely the situation where DACs can provide good compression and direct access to any given position. The maximum value of the root ($rMax$) is stored separately as an integer in plain form.

- The construction of the structure for the *minimum values* uses the same technique as for the maximum values. The minimum values are again encoded as differences with respect to the minimum value stored at the parent, again we have always positive values given that the minimum value of a node is always equal or greater than the minimum value of the parent node.[2] The

[2]The minimum value of a node could also be represented as a difference with respect to the maximum value of that node. In fact, since only differences greater than zero are represented, we

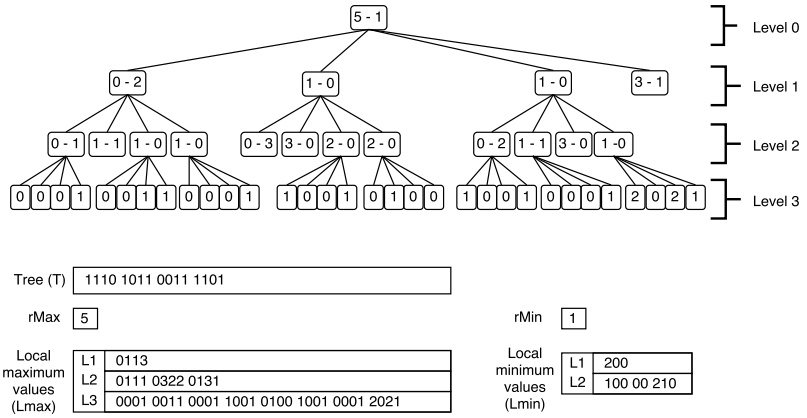


Figure 9.2: Compact representation of the conceptual k^2-raster using differences for the maximum and minimum values (top). Data structures T, $Lmax$ and $Lmin$ used for representing compactly the k^2-raster (bottom). Global maximum and minimum values are also stored separately.

only difference is that we do not need to store the minimum values at the leaf nodes, as the maximum value is enough to represent it. We denote $Lmin$ this array containing the differences for the minimum values, which is also encoded using DACs. The minimum value of the root ($rMin$) is also stored separately as an integer in plain form.

If T has t bits, $Lmax$ has at least t values, and in the first t values of $Lmax$, the i^{th} value corresponds to the maximum value in the submatrix represented by the i^{th} bit of T, as $Lmax$ has one maximum value for each internal node of the tree. That is, they are aligned, since both sequences use the same breadth-first traversal to determine the ordering. However, usually $Lmax$ has more elements, namely the values required for the last level of the tree (which are represented in $Lmax$ but not in T). $Lmin$ only contains values for those internal nodes z with $T[z] = 1$, since nodes with $T[z] = 0$ have a minimum value equal to their maximum value, which is already stored in $Lmax$. Since the first t values of T and $Lmax$ are aligned, given a position z of T, its corresponding value in $Lmax$ is the z^{th} number, and we can easily obtain its position in $Lmin$ as $rank(T, z)$.

Figure 9.2 shows in the upper part a conceptual tree representing a k^2-raster. It corresponds to the same raster used in Figure 9.1. This conceptual tree has an improvement with respect to that in Figure 9.1, namely the maximum and minimum

could subtract 1 to this difference value. This variant has also been proved experimentally and it obtained comparable results.

values stored at each node are now encoded using differences with respect to the values of its parent. The conceptual tree is just shown for illustrative purposes, as we only store the data structures shown in the bottom part of the figure. Observe that when the maximum and minimum values are equal, only the maximum value is stored. Using differences instead of the actual values causes that the final sequence of integers to encode is mostly composed of small numbers (assuming some uniformity among the values of the input raster matrix), and this will be exploited by DACs encoding.

Construction

The construction of k^2-raster can be easily done using a recursive procedure. The algorithm consists in a depth-first traversal of the tree that outputs, separately for each level ℓ of the tree, the bit array of the tree representation T_ℓ and the lists of maximum and minimum values for the nodes of that level ℓ, which we will call $Vmax_\ell$ and $Vmin_\ell$. Then, T can be obtained by concatenating bitmaps T_ℓ for all levels of the tree, and $Lmax$ and $Lmin$ can be obtained from $Vmax_\ell$ and $Vmin_\ell$ respectively, by obtaining the differences between parents and children, concatenating the sequences of all levels, and encoding the final sequences using DACs. The total time of the algorithm is linear in the number of cells of the matrix, that is, $O(nm)$. In fact, it is optimal, since it processes the raster accessing each cell only once.

The algorithm proceeds as follows: for any level except for the last level of the tree, it performs k^2 recursive calls, each one for the k^2 submatrices resulting from a subdivision. When it reaches the last level of the tree, that call processes k^2 leaf nodes of the tree, which correspond to cells of the original matrix. It checks whether the k^2 cells are all equal. If they are all equal, it just returns that value as maximum and minimum values; otherwise, it appends those k^2 values to $Vmax_\ell$, compute their maximum and minimum values and return them as result of the call.

When returning after a recursive call, the algorithm obtains the maximum and minimum values of its k^2 children. For each child, if these values are different, it appends these values to $Vmax_\ell$ and $Vmin_\ell$ lists and sets up a 1 in the T_ℓ of that level. If the maximum and minimum values are equal, it appends the value to $Vmax_\ell$ and sets up a 0 in T_ℓ. After processing the k^2 children, it checks whether all the maximum and minimum values are equal, which indicates that all the children contain the same value. Thus, the algorithm must undo the last operations, as these nodes will not have a representation in the data structure. This can be easily done by removing the last k^2 positions of T_ℓ and $Vmax_\ell$, or just moving the pointer that indicates their last written position, k^2 positions backwards. Finally, the algorithm returns the maximum and minimum values to its parent.

Algorithm 9.1 shows the pseudocode of the construction process. It is invoked as **Build**$(n, 1, 0, 0)$, where the first parameter is the (possibly extended) raster matrix size, the second is the current level, the third is the row offset of the current

Algorithm 9.1 Build(n, ℓ, r, c) computes T, $Vmax$ and $Vmin$ of the k^2-raster representation from matrix M and returns $(rMax, rMin)$

```
1: minval ← ∞
2: maxval ← 0
3: for i ← 0...k − 1 do
4:     for j ← 0...k − 1 do
5:         if ℓ = ⌈log_k n⌉ then                                   ▷ last level
6:             if minval > M_{r+i,c+j} then
7:                 minval ← M_{r+i,c+j}
8:             end if
9:             if maxval < M_{r+i,c+j} then
10:                maxval ← M_{r+i,c+j}
11:            end if
12:            Vmax_ℓ[pmax_ℓ] ← M_{r+i,c+j}
13:            pmax_ℓ ← pmax_ℓ + 1
14:        else                                                    ▷ internal node
15:            (childmax, childmin) ← Build(n/k, ℓ + 1, r + i · (n/k), c + j · (n/k))
16:            Vmax_ℓ[pmax_ℓ] ← childmax
17:            if maxval <> minval then
18:                Vmin_ℓ[pmin_ℓ] ← childmin
19:                pmin_ℓ ← pmin_ℓ + 1
20:                T_ℓ[pmax_ℓ] ← 1
21:            end if
22:            pmax_ℓ ← pmax_ℓ + 1
23:            if minval > childmin then
24:                minval ← childmin
25:            end if
26:            if maxval < childmax then
27:                maxval ← childmax
28:            end if
29:        end if
30:    end for
31: end for
32: if minval = maxval then
33:    pmax_ℓ ← pmax_ℓ − k^2
34: end if
35: return (maxval, minval)
```

submatrix, and the fourth is its column offset. It assumes that k, T_ℓ, $Vmax_\ell$, and $Vmin_\ell$ are global variables, and that T_ℓ, $Vmax_\ell$, and $Vmin_\ell$ have been initialized as empty sequences. In addition, the global variables $pmax_\ell$ and $pmin_\ell$ are used to know the last written position of $Vmax_\ell$ and $Vmin_\ell$ respectively. After running the algorithm, all T_ℓ must be joined to make up T, the same must be done with $Vmax_\ell$ and $Vmin_\ell$ to obtain $Vmax$ and $Vmin$, which, in turn, must be converted into

Algorithm 9.2 getCell$(n, r, c, z, maxval)$ returns the value at cell (r, c)

```
1: z ← rank(T, z) · k²
2: z ← z + ⌊r/(n/k)⌋ · k + ⌊c/(n/k)⌋
3: val ← accessDACs(Lmax, z)
4: maxval ← maxval − val
5: if z ≥ |T| or T[z] = 0 then                                  ▷ leaf
6:     return maxval
7: else                                                         ▷ internal node
8:     return getCell(n/k, r mod (n/k), c mod (n/k), z, maxval)
9: end if
```

$Lmax$ and $Lmin$ by computing the differences and encoding with DACs. Observe that the algorithm returns the maximum and minimum values of the input matrix, that is, $rMax$ and $rMin$, which must be represented in plain form.

9.1.2 Query algorithms

We describe in this section the algorithms that navigate the k^2-raster to solve queries over the raster matrix. We include pseudocodes and examples for some queries to better illustrate the most important procedures.

Obtaining a cell value

To obtain the value of a given cell, the algorithm performs a top-down traversal of the tree. It traverses the node at each level corresponding to the submatrix that contains the queried cell. During the descent through the tree, the algorithm should decode the maximum values stored at the traversed nodes, by subtracting each value from that in the parent. This is needed, since once we reach the queried cell, the stored value is kept as a difference with respect to the maximum value stored at the parent.

Algorithm 9.2 shows the pseudocode of this query. To obtain the value stored at cell (r, c) of the raster matrix, that is, cell M_{rc} at row r and column c, it is invoked as **getCell**$(n, r, c, -1, rMax)$, where n is the size of the matrix, (r, c) is the position of the queried cell, -1 corresponds to the position in T of the node to process (the initial -1 is an artifact because T does not represent the root node), and $rMax$ is the maximum value in the whole raster. T, $Lmax$, and k are global variables. It is assumed that $rank(T, -1) = 0$.

This query has a worst-case time $O(\log_k n \cdot \mathcal{L})$, which corresponds to a full traversal from the root node to the last level of the k^2-raster requiring to decode a value from $Lmax$ at each level. $\mathcal{L}$ denotes the number of levels used in DACs for representing $Lmax$, which depends on the largest number encoded in the sequence.

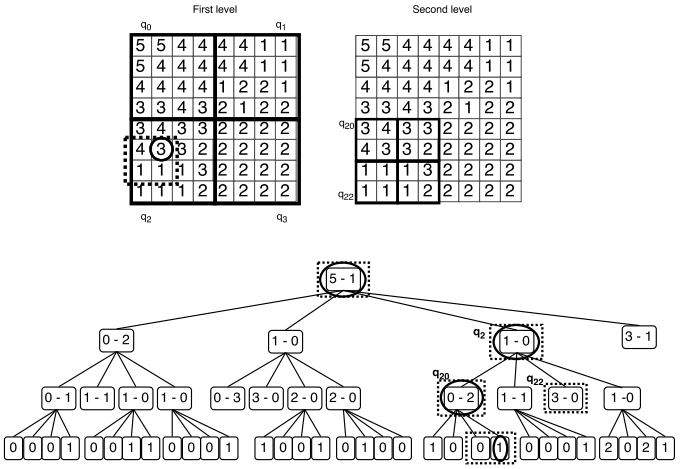


Figure 9.3: Submatrix subdivision and conceptual tree example to illustrate *getCell* and *getWindow* operations. We highlight the nodes used in the examples.

This time will be lower when the queried cell is surrounded by cells with the same value.

To illustrate how this query is computed, we will obtain the value at position $(5, 1)$ of the raster shown in Figure 9.3,which is the cell surrounded with a circle. In the bottom part of the figure we include the corresponding conceptual tree, which is represented using the data structures shown in Figure 9.2. We invoke the algorithm with **getCell**$(8, 5, 1, -1, 5)$. Having as input the node corresponding to the whole 8×8 matrix, the first step (lines 1–2) is to find the position in T (and thus in $Lmax$) of the node corresponding to the submatrix 4×4 that contains the queried cell, which in our example is the submatrix q_2 of the Figure 9.3, that is, $z \leftarrow rank(T, -1) \cdot 4 + 5/4 \cdot 2 + 1/4 = 2$. Then, the maximum value of q_2 is obtained (lines 3–4) as follows. First the algorithm obtains the value stored in $Lmax$ as $val \leftarrow accessDACs(Lmax, 2) = 1$, and then it subtracts that value from the maximum value received as a parameter $maxval \leftarrow 5 - 1 = 4$. Next, the condition of line 5 is checked to determine whether we are in an internal node or not. Since $z = 2 < |T| = 16$ and $T[2] = 1$, it recursively invokes **getCell**$(8/2, 5 \bmod 4, 1 \bmod 4, 2, 4) =$ **getCell**$(4, 1, 1, 2, 4)$. Then, the algorithm repeats the same procedure in the next level, this time having as input the node corresponding to submatrix q_2.

Lines 1–2 find the position in T and $Lmax$ of the submatrix of q_2 containing the queried cell as $z \leftarrow rank(T, 2) \cdot 4 + 1/2 \cdot 2 + 1/2 = 12 + 0 + 0 = 12$, which corresponds to the submatrix q_{20} of Figure 9.3. Then, the algorithm obtains the maximum value of q_{20} as $val \leftarrow accessDACs(Lmax, 12) = 0$, $maxval \leftarrow$

$4 - 0 = 4$. Since $z = 12 < |T|$ and $T[12] = 1$, the algorithm recursively invokes **getCell**$(4/2, 1 \bmod 2, 1 \bmod 2, 12, 4)$ = **getCell**$(2, 1, 1, 12, 4)$.

Having the node corresponding to submatrix q_{20} as input, the algorithm obtains the position of the submatrix that contains the queried cell (this time is a 1×1 submatrix only containing that cell) as $z \leftarrow rank(T, 12) \cdot 4 + 1/1 \cdot 2 + 1/1 = 39$, and its value as $val \leftarrow accessDACs(Lmax, 39) = 1$, $maxval \leftarrow 4 - 1 = 3$. Finally, since $z = 39 \geq |T| = 16$, a 3 is returned, which is the content of cell $(5, 1)$.

In the conceptual tree of Figure 9.3, we highlight the nodes affected by this example with ellipses drawn with solid lines.

Obtaining all the values of a region

Obtaining a region of the raster matrix can be done more efficiently than just obtaining its cells individually using *getCell*, since the same top-down traversal of the tree can be used for extracting values from adjacent positions. Thus, decoding maximum values is performed just once per traversed node, instead of once per cell.

Algorithm 9.3 shows the pseudocode for this query, which is also a recursive procedure. To obtain all the cells contained inside a window $[r_1, r_2] \times [c_1, c_2]$, the algorithm is invoked as **getWindow**$(n, r_1, r_2, c_1, c_2, -1, rMax)$. Again, k, T, and $Lmax$ are considered global variables.

Let us illustrate the algorithm with an example using the raster matrix shown in Figure 9.3. We want to know the cell values in the range $[5, 6] \times [0, 1]$, which is the submatrix surrounded with a square with dotted lines in the figure. In the conceptual tree of Figure 9.3, we highlight the nodes affected by this example with rectangles drawn with dotted lines.

The algorithm is invoked with **getWindow**$(8, 5, 6, 0, 1, -1, 5)$, that is, having as input the whole matrix M, the position of the queried range in M, the position in T of the node representing M (a -1, since the root node is not represented), and the maximum value of M. First, the algorithm computes the position in T and $Lmax$ of the first children of the root node as $z \leftarrow rank(T, -1) = 0$. Then, the algorithm has to determine which submatrices of the first level have to be further inspected to solve the query, that is, which submatrices overlap the queried region. In our case, we only have to inspect the bottom-left submatrix of the current submatrix (corresponding to $i = 1, j = 0$ in line 12), which is the submatrix denoted as q_2. Lines 3–6 and 8–9 give the relative position of the queried range *inside* q_2, in our example, the queried region is the submatrix $[1, 2] \times [0, 1]$ of q_2, that is, it covers rows 1 and 2 and columns 0 and 1 of q_2. Line 12 obtains the position z' in T and $Lmax$ corresponding to the submatrix q_2 as $z' \leftarrow 0 + 2 \cdot 1 + 0 = 2$. Next, the algorithm computes the maximum value of q_2 as $maxval' \leftarrow 5 - accessDACs(Lmax, 2) = 5 - 1 = 4$. Since $T[2] = 1$ and $2 < |T|$, that node is an internal node, and thus the recursive call **getWindow**$(4, 1, 2, 0, 1, 2, 4)$ is launched. That is, to solve our query, it has to return the cells in the region $[1, 2] \times [0, 1]$ of the 4×4 submatrix q_2.

Algorithm 9.3 getWindow$(n, r_1, r_2, c_1, c_2, z, maxval)$ returns all cells from region $[r_1, r_2]$ to $[c_1, c_2]$

```
 1: z ← rank(T, z) · k²
 2: for i ← ⌊r1/(n/k)⌋ ... ⌊r2/(n/k)⌋ do
 3:     if i = ⌊r1/(n/k)⌋ then r'1 ← r1 mod (n/k)
 4:     else r'1 ← 0
 5:     end if
 6:     if i = ⌊r2/(n/k)⌋ then r'2 ← r2 mod (n/k)
 7:     else r'2 ← (n/k) − 1
 8:     end if
 9:     for j ← ⌊c1/(n/k)⌋ ... ⌊c2/(n/k)⌋ do
10:         if j = ⌊c1/(n/k)⌋ then c'1 ← c1 mod (n/k)
11:         else c'1 ← 0
12:         end if
13:         if j = ⌊c1/(n/k)⌋ then c'1 ← c1 mod (n/k)
14:         else c'1 ← 0
15:         end if
16:         if j = ⌊c2/(n/k)⌋ then c'2 ← c2 mod (n/k)
17:         else c'1 ← 0
18:         end if
19:         if j = ⌊c2/(n/k)⌋ then c'2 ← c2 mod (n/k)
20:         else c'2 ← (n/k) − 1
21:         end if
22:         z' ← z + k · i + j
23:         maxval' ← maxval − accessDACs(Lmax, z')
24:         if z' ≥ |T| or T[z] = 0 then                          ▷ leaf
25:             Output maxval ((r'2 − r'1) + 1) · ((c'2 − c'1) + 1) times
26:             return
27:         else                                                  ▷ internal node
28:             getWindow(n/k, r'1, r'2, c'1, c'2, z', maxval')
29:         end if
30:     end for
31: end for
```

This call starts by computing the position of T and $Lmax$ where the children of q_2 start as $z \leftarrow rank(T, 2) \cdot 4 = 3 \cdot 4 = 12$. The **for** of line 2 iterates i over 0..1 and the **for** of line 7 iterates j only over 0. Therefore, at this call, we have to treat two submatrices of q_2, the top-left and the bottom-left submatrices, which we denote as q_{20} and q_{22} in Figure 9.3.

- q_{20}: lines 3–6 and 8–9 give the relative position of the queried range inside q_{20}. Observe that the part of the queried range that overlaps q_{20} is the submatrix $[1, 1] \times [0, 1]$ within q_{20}, which corresponds to submatrix $[5, 5] \times [0, 1]$ in the original matrix. Now, the algorithm obtains the position in T and $Lmax$ of the

information corresponding to q_{20} as $z' \leftarrow 12 + 2 \cdot 0 + 0 = 12$ and we obtain the new maximum value as $maxval' \leftarrow 4 - accessDACs(Lmax, 12) = 4 - 0 = 4$. Given that $T[12] = 1$ and $12 < |T|$, that node is an internal node, and thus the recursive call **getWindow**$(2, 1, 1, 0, 1, 12, 4)$ is performed.

The execution of this call starts by computing the position in $Lmax$ where the children of q_{20} start, $z = rank(T, 12) \cdot 4 = 36$. The **for** of line 2 iterates i only over 1 and the **for** of line 7 iterates j over 0..1. That is, this call has to process the bottom-left and bottom-right submatrices of q_{20}. Those submatrices only contain one cell, that is, they are leaves. For the bottom-left leaf, the algorithm computes its position in $Lmax$ as $z' \leftarrow 36 + 2 \cdot 1 + 0 = 38$, and thus it obtains its value as $maxval' \leftarrow 4 - accessDACs(Lmax, 38) = 4 - 0 = 4$. On the other hand, for the bottom-right leaf, its position in $Lmaxer$ is $z' \leftarrow 36+2\cdot1+1 = 39$, its value is $maxval' \leftarrow 4 - accessDACs(Lmax, 39) = 4 - 1 = 3$.

- q_{22}: lines 3–6 and 8–9 obtain the relative position of the queried range inside q_{22}. Observe that the part of the queried range that overlaps q_{22} is the relative submatrix $[0, 0] \times [0, 1]$ ($[6, 6] \times [0, 1]$, if we consider the whole matrix).

  Recall that at the start of this call, z was set to 12, then we compute the position in T and $Lmax$ of the information associated with the submatrix q_{22} as $z' \leftarrow 12 + 2 \cdot 1 + 0 = 14$. Then we can obtain the maximum value of that submatrix as $maxval' \leftarrow 4 - accessDACs(Lmax, 14) = 4 - 3 = 1$.

  Given that $T[14] = 0$, the node corresponding to q_{22} is a leaf node, therefore line 15 returns the value of $maxval'$ $((0 - 0) + 1) \cdot ((1 - 0) + 1) = 2$ times. That is, since all cells of q_{22} have the same value (1), then it is represented as a leaf node, and the part of q_{22} that overlaps the queried region contains 2 cells, and then this call returns two 1s.

Retrieving cells with a given value or range of values

We describe now how to obtain the positions of all cells within the region $[r_1, r_2] \times [c_1, c_2]$ that contain values in the range $[v_b, v_e]$. If we want to run the query for the whole matrix, we just adjust $[r_1, r_2] \times [c_1, c_2]$ to the complete matrix, and if we want to search the cells having a particular value v, we adjust the range to $[v, v]$.

The algorithm to solve this query combines the functionality of the original k^2-tree to solve range queries, which is able to efficiently obtain cells with 1s within a given rectangle, with the indexing capabilities offered by the k^2-raster, thanks to the storage of the maximum and minimum values at the nodes of the tree. As in previous queries, the search involves a top-down traversal of the tree, but it requires to perform two checks at each level. After obtaining the branches of the tree corresponding to submatrices overlapping the queried region, it has to check whether the maximum and minimum values in those quadrants are compatible with the queried range, discarding those that fall outside the range of values sought.

Algorithm 9.4 shows the pseudocode for this query. It is again a recursive procedure invoked as **searchValuesInWindow**$(n, r_1, r_2, c_1, c_2, v_b, v_e, rMax, rMin, -1)$, if we want to retrieve the cells inside the window $[r_1, r_2] \times [c_1, c_2]$ having values in the range $[v_b, v_e]$. For this algorithm, k, T, $Lmax$, and $Lmin$ are considered global variables.

Lines 1–14 of Algorithm 9.4 are exactly the same as those in *getWindow*. If the condition of line 14 is true, we have reached a leaf node that corresponds to a submatrix that overlaps the queried region. In the case of *getWindow*, the algorithm immediately returns the values of the cells in that region, but now the algorithm *searchValuesInWindow* has to perform the second test (line 16) to check whether the values of the cells in that region have values in the range of values $[v_b, v_e]$. Observe that when reaching this point, all cells in the considered region have the same value, or it is a region with only one cell, and thus the algorithm only has to return the position of the cells of the submatrix that overlaps the queried region.

In case of an internal node (lines 20–29), we have to obtain the minimum value of that submatrix, and compare the maximum and minimum values of the submatrix with the queried range:

- If the minimum and maximum values of the submatrix are within the range $[v_b, v_e]$: then all cells meet the condition of the query; thus, all cells inside the queried region must be returned.
- If the minimum value of the submatrix is greater than v_e or the maximum value is smaller than v_b: then no cell in the submatrix meets the criteria; thus nothing is returned.
- If the values in the cells of the considered submatrix partially match $[v_b, v_e]$: then we have to perform a recursive call to further inspect the submatrix.

Note that this query returns the positions of the values that meet the criteria. If it is required to know the exact values of those positions, they could be retrieved with *getCell*, or in a more efficient way by adding calls to *getWindow* when we report that a submatrix has all its elements within the query range.

Checking the existence of a given value or range of values

Given a value or range of values and a region of the raster matrix, the k^2-raster can determine if inside that region, there exits at least one cell with a value in the queried range or if all cells have values within the queried range. The first case is known as *weak* semantics, whereas the latter is known as *strong* semantics.

This query can be done more efficiently than retrieving all the values of the region and then checking if they lie within the range of values. This is basically a simplification of Algorithm 9.4 that, in the case of weak semantics, as soon as it finds that a submatrix of the queried region has values in the range $[v_b, v_e]$ returns

Algorithm 9.4 searchValuesInWindow$(n, r_1, r_2, c_1, c_2, v_b, v_e\ maxval, minval, z)$ returns all cell positions from region $[r_1, r_2]$ to $[c_1, c_2]$ containing values within $[v_b, v_e]$

```
1: z ← rank(T, z) · k^2
2: for i ← ⌊r1/(n/k)⌋ ... ⌊r2/(n/k)⌋ do
3:     if i = ⌊r1/(n/k)⌋ then x'1 ← r1 mod (n/k)
4:     else r'1 ← 0
5:     end if
6:     if i = ⌊r2/(n/k)⌋ then r'2 ← r2 mod (n/k)
7:     else r'2 ← (n/k) − 1
8:     end if
9:     for j ← ⌊c1/(n/k)⌋ ... ⌊c2/(n/k)⌋ do
10:        if j = ⌊c1/(n/k)⌋ then c'1 ← c1 mod (n/k)
11:        else c'1 ← 0
12:        end if
13:        if j = ⌊c2/(n/k)⌋ then c'2 ← c2 mod (n/k)
14:        else c'2 ← (n/k) − 1
15:        end if
16:        z' ← z + k · i + j
17:        maxval' ← maxval − accessDACs(Lmax, z)
18:        if z ≥ |T| or T[z] = 0 then                                  ▷ leaf
19:            minval' ← maxval'
20:            if minval' ≥ vb and maxval' ≤ ve then   ▷ all cells meet the condition in this branch
21:                Output corresponding region of cells
22:                return
23:            end if
24:        else                                                         ▷ internal node
25:            minval' ← minval + accessDACs(Lmin, rank(T, z))
26:            if minval' ≥ vb and maxval' ≤ ve then   ▷ all cells meet the condition in this branch
27:                Output corresponding region of cells
28:
29:                return
30:            end if
31:            if minval' > ve or maxval' < vb then
32:                return                          ▷ no cells meet the condition in this branch
33:            end if
34:            if minval' < vb or maxval' > ve then searchValuesInWindow(n/k, r'1, r'2, r'1, r'2, vb, ve, maxval', minval', z')
35:            end if
36:        end if
37:    end for
38: end for
```

true. This can be done in a non-leaf node without the necessity of reaching the leaves, with just the minimum and maximum values stored at that node.

In the case of strong semantics, the query is basically the same, but now, as soon as we find that there is, at least, one cell of a submatrix within the queried region that is not within the range, the algorithm stops returning *false*.

The k^2-raster also allows for other efficient queries, such as obtaining the maximum value or the minimum values of region, etc.

9.1.3 Hybrid variant

As seen, most queries require a top-down traversal from the root node to some leaves at the last level of the tree; therefore, the number of levels has an important impact in query times. To reduce the height of the tree, we present a modification of the basic k^2-raster that significantly reduces the time of some queries with the cost of slightly increasing the space requirements of the structure.

This version allows us to modify how the matrix is partitioned during the first levels of the tree, by allowing the use of two different values of the k parameter, k_1 and k_2; k_1 is used in the subdivision of the first levels, and k_2 for the rest. The target is to obtain a smaller tree, by dividing each quadrant into more submatrices in the first levels, that is, we obtain a wider and smaller tree.

Now, instead of k, the construction algorithm needs the two k_1, k_2 parameters and another new parameter n_1, which is the number of levels where the subdivision is done using k_1. More precisely, when creating the first n_1 levels, each submatrix is partitioned into k_1^2 submatrices and for levels $n_1 + 1$ until the leaf nodes is divided into k_2^2 submatrices. From now on, this is the standard version of k^2-raster.

In Figure 9.4, we can see a k^2-raster built with $k_1 = 4$, $k_2 = 2$, and $n_1 = 1$. Observe that in the first level (given that $n_1 = 1$), the matrix is divided into $k_1^2 = 16$ submatrices, each producing a child node of the root and storing the maximum and minimum values in that submatrix. Therefore, the root has 16 children. The second level uses $k = 2$, and thus each submatrix is divided into 4 submatrices, which in this case are individual cells. As it can be seen in the figure, the tree is wider and smaller, thus producing faster top-down traversals.

This hybrid variant can be generalized by using a different k value for each level, such that we subdivide level ℓ into k_ℓ^2 submatrices. Using just two values of k, a larger one for the first levels and a smaller one for the last levels of the tree, works well in practice.

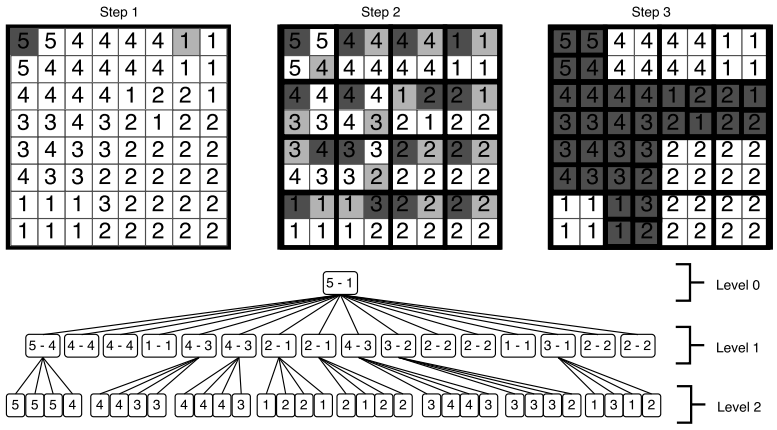


Figure 9.4: Example of using different k values. We indicate the minimum (light gray) and maximum (dark gray) values of each submatrix for the three steps of the recursive subdivision of the construction algorithm (top). Conceptual tree representation obtained from the construction of the hybrid k^2-raster with $k_1 = 4$, $k_2 = 2$ and $n_1 = 1$ (bottom).

9.2 Heuristic k^2-raster: k^2_H-raster

The original k^2-tree structure has a variant that uses a compressed representation of the last level of the tree, which is composed of the submatrices of the original adjacency matrix resulting from the last subdivision. This compression allows the use of a large k in the last level, which shortens the tree and improves navigational times, without increasing the space requirements of the structure. In fact, this compression generally causes an improvement on the space results. Thus, following the same strategy used for k^2-trees, we also propose a variant of k^2-raster that uses a compressed representation of the last level of $Lmax$, that is, the entries corresponding to the submatrices of size $k_{Lst} \times k_{Lst}$ of the original raster matrix resulting from last subdivision, where Lst denotes the last level of the conceptual tree built by the k^2-raster recursive subdivision of the raster matrix and k_{Lst} the value of k used for that level.

More concretely, we will compress $Lmax[Lst]$, which denotes the portion of $Lmax$ representing the cells in Lst, that is, it represents the values of the $k_{Lst} \times k_{Lst}$ non-equal submatrices of the original raster matrix that appear in the last level of the conceptual tree. In the Figure 9.2, $Lmax[Lst]$ is the part of $Lmax$ labeled as $L3$.

Figure 9.5 shows the problem we want to address: we maintain the same

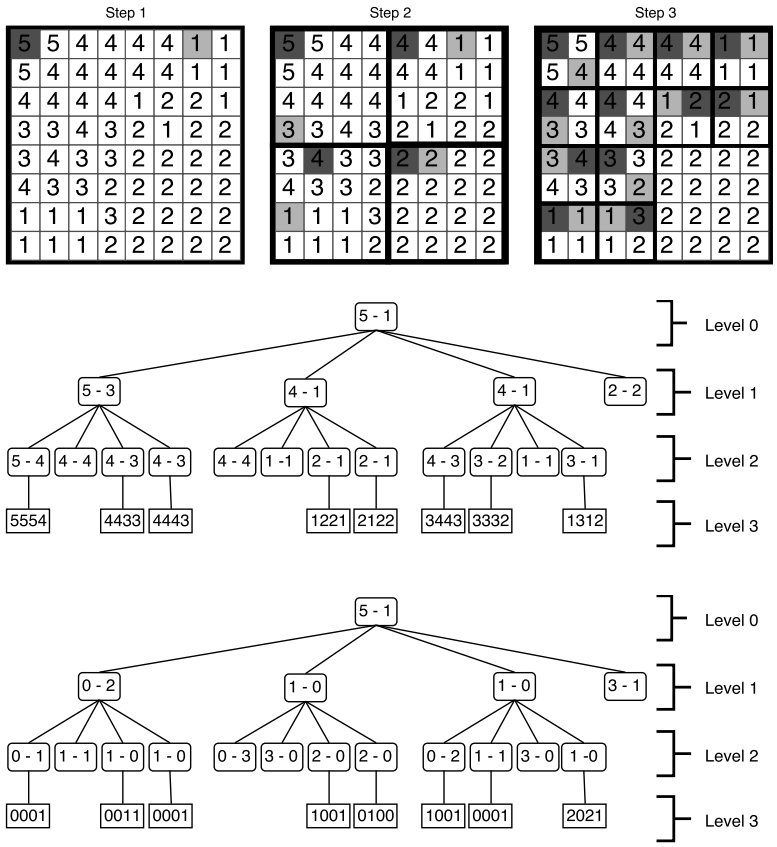


Figure 9.5: Example of raster matrix (top), conceptual tree representation obtained from the construction of the k^2_H-raster (center), and conceptual tree using differential encoding (bottom). The last level is represented using $k_{Lst} \times k_{Lst}$ submatrices, being $k_{Lst} = 2$ for this example.

conceptual representation for the raster matrix except for the last level of the tree, where we want to compactly represent its submatrices of size $k_{Lst} \times k_{Lst}$. We use $k_{Lst} = 2$ for this example, but this technique allows us to use larger k values for the last level of the tree for real datasets.

To compress $Lmax[Lst]$, one possibility is to create a vocabulary by extracting all the different $k_{Lst} \times k_{Lst}$ last-level submatrices, sorting the vocabulary by frequency, and substituting the $k_{Lst} \times k_{Lst}$ contiguous values in $Lmax$ corresponding to each

submatrix by a pointer to its entry in the frequency-sorted vocabulary. However, this strategy, which is the one used for the binary last-level submatrices in the original k^2-tree, is not suitable for k^2-raster, since there are many possible $k_{Lst} \times k_{Lst}$ different integer submatrices, some of them appearing just once, and therefore, the vocabulary becomes very large. Since the compressed representation of $Lmax[Lst]$ consists not only of the pointers but also of the vocabulary, we obtain no compression in case of large vocabularies with many submatrices that are not repeated.

In the example of Figure 9.5, we would have a vocabulary composed of the five 2×2 different submatrices existing at the last level of the conceptual tree. The vocabulary, sorted by frequency, would be $v = \{\langle 0001 \rangle, \langle 1001 \rangle, \langle 0011 \rangle, \langle 0100 \rangle, \langle 2021 \rangle\}$. If we represented $Lmax[Lst]$ with the vocabulary approach, it would require the list of pointers to each vocabulary entry, that is, $p = \{0, 2, 0, 1, 3, 1, 0, 4\}$, in addition to v. Thus, for those submatrices appearing just once in $Lmax[Lst]$, that is, $\langle 0011 \rangle, \langle 0100 \rangle, \langle 2021 \rangle$, we would require their plain representation in v plus a pointer in $Lmax[Lst]$. The basic k^2-raster presented in the previous section would represent these submatrices by simply storing their content, without the overhead of the pointer. Thus, this compression approach would require more space for these submatrices, which may lead to worse space results.

Thus, compressing $Lmax[Lst]$ requires a more refined approach, where we evaluate if including a submatrix in the vocabulary will save space in the final representation. We use an entropy-based heuristical approach to estimate these savings. Thus, we call this improved variant of the technique *heuristic k^2-raster* or *k^2_H-raster*.

More specifically, to obtain the k^2_H-raster of a given raster matrix, we first build the normal k^2-raster, and then:

1. We traverse all $k_{Lst} \times k_{Lst}$ submatrices corresponding to $Lmax[Lst]$, and compute their frequency. Simultaneously, we also compute the frequency for all the individual values that appear in those submatrices.

2. We estimate the average number of bits needed for representing a submatrix using the vocabulary-based approach. Simultaneously, we estimate the average number of bits needed for representing an individual cell value when using DACs to represent them.

3. We sort the vocabulary of submatrices by frequency.

4. For each submatrix of the vocabulary:

   (a) We estimate the cost of representing it as a compressed submatrix using the vocabulary: we multiply its frequency by the average number of bits required for representing a submatrix and we add the space needed to store it in the vocabulary.

(b) We estimate the cost of representing it as individual values using DACs: we multiply the frequency of the submatrix by its size (i.e., k^2_{Lst} cells) and by the average number of bits required for representing an individual number.

(c) We choose the representation with minimum cost. In case of choosing the vocabulary-based approach, we assign a new correlative codeword to the submatrix, which is a pointer to its position in the vocabulary.

We use the zero-order empirical entropy (see Section 2.1.1) to estimate the average number of bits needed to encode the submatrices and the individual values. When we estimate the average number of bits to represent a matrix, the alphabet is the list of different submatrices (the vocabulary of submatrices) in $Lmax[Lst]$. In the case of individual values, the alphabet is formed by the list of different integers appearing in $Lmax[Lst]$.

Then, for a given submatrix s_i having the values $v_1, v_2, \ldots v_{k^2_{Lst}}$ in its cells, we estimate the size (in bits) required to represent that submatrix as $E_{s_i} = (f_{s_i} \cdot H_0(S^s)) + (k^2_{Lst} \cdot w)$, where f_{s_i} is the frequency of appearance of s_i in $Lmax[Lst]$, $H_0(S^s)$ is the average number of bits to represent a submatrix using an alphabet of submatrices, and w is the machine word size. $f_{s_i} \cdot H_0(S^s)$ is an estimation of the size of the pointers that substitute the values of the submatrices in $Lmax[Lst]$. In addition, we also need to store an entry in the vocabulary with the k^2_{Lst} values of the submatrix in plain form.

On the other hand, if we use all the individual values to represent the content of submatrix s_i, we estimate the size as $E_{v_i} = f_{s_i} \cdot k^2_L \cdot H_0(S^v)$, where $H_0(S^v)$ is the average number of bits to represent each individual value. If $E_{s_i} < E_{v_i}$, we represent the occurrences of s_i in $Lmax[Lst]$ with pointers to the entry of the vocabulary of submatrices corresponding to s_i, otherwise we use the original method, that is, we represent its values individually using DACs.

Once we have decided which submatrices will be represented with the vocabulary, we need to create the structures to implement a new $Lmax[Lst]$ where some submatrices are represented as pointers to entries in a vocabulary and others as a list of individual values. For this sake, we create three additional structures:

- Bitmap *isInVoc*, which indicates whether one submatrix is represented with a pointer to the vocabulary or not.

- Array *encodedValues*, which includes the codewords (pointers) for the submatrices that are represented using the vocabulary.

- Array *plainValues*, which includes the encoding for the individual values of the submatrices that are not represented using the vocabulary.

Then, we traverse again $Lmax[Lst]$ and for each submatrix:

- If it is in the list of submatrices to be represented with vocabulary, we set to 1 its corresponding bit in *isInVoc* and append its codeword to *encodedValues*.
- Otherwise, we set to 0 its corresponding bit in *isInVoc* and append all its values to *plainValues*.

Algorithm 9.5 shows the pseudocode of the algorithm that obtains the bitmap $isInVoc$, and the arrays $encodedValues$ and $plainValues$. The inputs of the algorithm are $Lmax$, and the position of $Lmax$ where $Lmax[Lst]$ starts (the parameter P_{Lst}). The value of k for the last level of the tree, that is, k_{Lst}, is a global variable. For the computation of these structures, we create a temporary vocabulary for all the submatrices (s) where we store their values, frequency and codeword. When this procedure ends, we need to add rank support to bitmap *isInVoc* and compact arrays *encodedValues* and *plainValues* using DACs. In addition, we need to create the final vocabulary (Voc) by removing the submatrices that are represented in plain form.

To better understand this variant, we show in Figure 9.6 the compact representation (bottom) resulting from a conceptual tree (top). Notice that the codeword for each submatrix at the vocabulary is implicit and it does not consume space in the representation, as it corresponds to its position in the sorted vocabulary. In this example, only $\langle 0001 \rangle$ and $\langle 1001 \rangle$ have been selected for the vocabulary, as the others only have one appearance in the last level and representing their individual values directly saves more space than including them in the vocabulary and using a codeword.

When processing the last level of leaves from left to right, the first leaf is $\langle 0001 \rangle$, which is one of the leaves that should be represented as a pointer to the vocabulary, therefore the first bit of $isInVoc$ is set to 1. In addition, the algorithm adds the codeword that represents $\langle 0001 \rangle$ in the first position of $encodedValues$, that is, it inserts a 0, since that is the position of $\langle 0001 \rangle$ in the frequency-sorted vocabulary. The second submatrix is $\langle 0011 \rangle$, which should be represented in plain form, then the second bit of $isInVoc$ is set to 0 and the four values ($\langle 0011 \rangle$) are stored in the first four position of $plainValues$. Next, we have a $\langle 0001 \rangle$, therefore the third bit of $isInVoc$ is set to 1, and the second entry of $encodedValues$ is filled with a 0, and so on.

9.2.1 Querying

The navigation over this variant differs from the navigation over the original k^2-raster when accessing the last level of $Lmax$. Instead of directly obtaining its values, the k^2_H-raster requires accessing to the bitmap that indicates whether the submatrix is stored compressed or in plain form, and accessing the corresponding sequence of encoded or plain values. We illustrate how we access the last-level submatrices by showing how the query $getCell$ is done. Algorithm 9.6 shows the pseudocode of the

Algorithm 9.5 $\textbf{Build}_H(Lmax,P_{Lst})$ computes $isInVoc$, $encodedValues$, and $plainValues$

```
1: s ← subMatricesFreq(Lmax,P_Lst,k_Lst)  ▷ Compute the frequency of each different k_Lst × k_Lst submatrix in Lmax
2: v ← valuesFreq(Lmax,P_Lst)  ▷ Compute the frequency of each different value in Lmax
3: H_s ← entropy(s)  ▷ Compute the entropy of the submatrices
4: H_v ← entropy(v)  ▷ Compute the entropy of the values
5: for i ← 0 ... |s| − 1 do
6:     if ((H_s · s[i].freq) + (k²_Lst · w)) < (H_v · s[i].freq · k²_Lst) then
7:         s[i].cdwd ←computeNextCodeword()
8:     else
9:         s[i].cdwd ← −1
10:    end if
11: end for
12: j ← 0
13: posInEncoded ← 0
14: posInPlain ← 0
15: while j < |Lmax| do
16:    s_i ←searchInS(Lmax[Lst][P_Lst + j ... P_Lst + j + k²_Lst − 1]) ▷ Obtains the data in s of the current submatrix
17:    if s_i.cdwd = −1 then  ▷ The submatrix should be stored in plain
18:        isInVoc[j/k²_Lst] ← 0
19:        for t ← 0 ... k²_Lst − 1 do
20:            plainValues[posInPlain + t] ← s_i.values[t]
21:        end for
22:        posInPlain ← posInPlain + k²_Lst  ▷ The submatrix should be stored compressed
23:    else
24:        isInVoc[j/k²_Lst] ← 1
25:        encodedValues[posInEncoded] ← s_i.cdwd
26:        posInEncoded ← posInEncoded + 1
27:    end if
28:    j ← j + k²_Lst
29: end while
```

query. Notice that line 3 from Algorithm 9.2 has been replaced with lines 3–15. For the sake of simplicity, we use k for all levels, but k may have different values at each level of the tree.

To illustrate this with an example, let us obtain the value of the cell at position (5,1) of our running example. The algorithm is invoked as $\textbf{getCell}_H(8,5,1,-1,5)$. Lines 1–2 obtain the position in T and $Lmax$ of the value corresponding to the 4×4 that contains the queried cell, $z \leftarrow rank(T,-1) \cdot 4 + 5/4 \cdot 2 + 1/4 = 2$, which corresponds to the bottom-left submatrix. Since $2 < |T|$, we are in a level that is

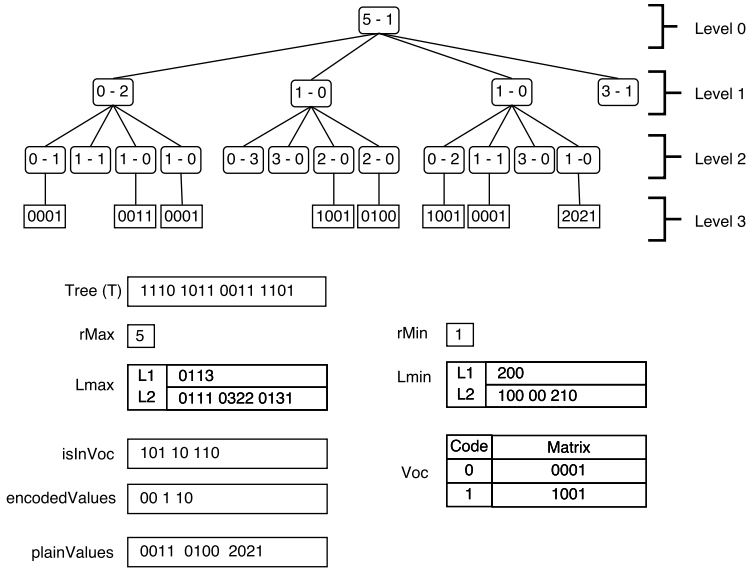


Figure 9.6: Compact representation of the conceptual k^2_H-raster using differences for the maximum and minimum values (top). Data structures T, $Lmax$, $Lmin$, Voc, $isInVoc$, $encodedValues$ and $plainValues$ used for representing compactly the k^2_H-raster (bottom).

not the last one, and thus the flow reaches line 14. Here, the process is the same as in the case of the normal k^2-raster, that is, the algorithm accesses the normal $Lmax$ to obtain the maximum value of that submatrix: $val \leftarrow accessDACs(Lmax, 2) = 1$, and then it subtracts that value from the maximum value received as a parameter $maxval \leftarrow 5 - 1 = 4$. Then, the condition of line 17 is checked, and since the current node is not a leaf, the flow reaches line 20, performing a recursive call $\mathbf{getCell}_H(8/2, 5 \bmod 4, 1 \bmod 4, 2, 4) = \mathbf{getCell}_H(4, 1, 1, 2, 4)$. In the recursive call, lines 1–2 find the position in T and $Lmax$ corresponding to the node representing the submatrix of the second level that contains the queried cell: $z \leftarrow rank(T, 2) \cdot 4 + 1/2 \cdot 2 + 1/2 = 12 + 0 + 0 = 12$. Again, this node is not in the last level of the tree, then the algorithm obtains the maximum value of that submatix $val \leftarrow accessDACs(Lmax, 12) = 0$, $maxval \leftarrow 4 - 0 = 4$ and performs a recursive call $\mathbf{getCell}_H(4/2, 1 \bmod 2, 1 \bmod 2, 12, 4) = \mathbf{getCell}_H(2, 1, 1, 12, 4)$. In the next recursive call, $z \leftarrow rank(T, 12) \cdot 4 + 1/1 \cdot 2 + 1/1 = 39$. Now, $z > |T|$, that is, we are accessing a leaf in the last level. Line 4 obtains the corresponding submatrix among those at the last level: $pos \leftarrow \lfloor (39 - |T|)/4 \rfloor = \lfloor (39 - 16)/4 \rfloor = 5$. Line 5 checks if that position is stored as a pointer to the vocabulary or is stored in plain form. Since $isInVoc[5] = 1$, that submatrix is stored as a pointer to the vocabulary.

Algorithm 9.6 getCell$_H(n, x, y, z, maxval)$ returns the value at cell (x, y)

```
1: z ← rank(T, z) · k²
2: z ← z + ⌊x/(n/k)⌋ · k + ⌊y/(n/k)⌋
3: if z ≥ |T| then                                          ▷ last level
4:     pos ← ⌊(z − |T|)/k²⌋
5:     if isInVoc[pos] = 1 then                              ▷ encoded in Voc
6:         pos ← rank_1(isInVoc, pos) − 1
7:         code ← accessDACs(encodedValues, pos)
8:         val ← Voc[code][x · k + y]
9:     else                                                  ▷ plain form
10:        pos ← rank_0(isInVoc, pos) · k² + x · k + y
11:        val ← accessDACs(plainValues, pos)
12:    end if
13: else                                                     ▷ not last level
14:    val ← accessDACs(Lmax, z)
15: end if
16: maxval ← maxval − val
17: if z ≥ |T| or T[z] = 0 then                              ▷ leaf
18:    return maxval
19: else                                                     ▷ internal node
20:    return getCell_H(n/k, x mod (n/k), y mod (n/k), z, maxval)
21: end if
```

Then the algorithm obtains the position of its codeword in *encodedValues* as $pos \leftarrow rank_1(isInVoc, 5) - 1 = 3$. Next, the algorithm accesses *encodedValues* to obtain the codeword of the submatrix: $code \leftarrow accessDACs(encodedValues, 3) = 1$. Thus, the algorithm must obtain the queried cell from the submatrix encoded at position 1 of Voc as $val \leftarrow Voc[1][1 \cdot 2 + 1] = Voc[1][3] = 1$. With that value, line 14 obtains $maxval \leftarrow 4 - 1 = 3$. Finally, since $z \geq |T|$ $(39 \geq 16)$, a 3 is returned, which is the content of cell $(5, 1)$.

The rest of query algorithms are easily modified in the same way, that is, only modifying the accesses to the last level of $Lmax$ in order to deal with the submatrices in the vocabulary.

Chapter 10

Spatial join: k^2-*raster* and R-tree

In this section, we present an algorithm to compute the join between a raster and a vector dataset, allowing a range constraint on the values of the raster. That is, the query returns the elements of a vector dataset (polygons, lines, or points) and the cells of the raster dataset that overlap each other, such that the cells have values in a given range $[v_b, v_e]$. It is also possible to apply spatial restrictions on both datasets, that is, restricting the join to windows or regions of the vector and the raster dataset.

10.1 Spatial join

The proposed algorithm requires, as input, the R-tree indexing the MBRs enclosing the vector dataset and the k^2-raster that stores and indexes the raster dataset.

In the case of the vector data, the R-tree contains only the MBRs of the spatial objects because their actual representation requires a much larger space and more complex computations, which would make the index useless. Therefore, for the rest of the thesis we will consider the vector dataset as a collection of MBRs. The vector answer of the query will be composed of two lists of MBRs: a list of *definitive results* and a list of *probable results*. The MBRs in the list of definitive results fully intersect a region in the raster with values in the queried range, whereas MBRs in the second list intersect a region in the raster with *some* cells fulfilling the range constraint. Thus, the elements enclosed by the MBRs in the second list require an additional refinement task that uses the complete geometry of each spatial object inside the MBR to check whether the intersection holds or not with the returned cells. This additional procedure is not considered in the algorithm and the experimental evaluation, as usual when spatial indexes are evaluated.

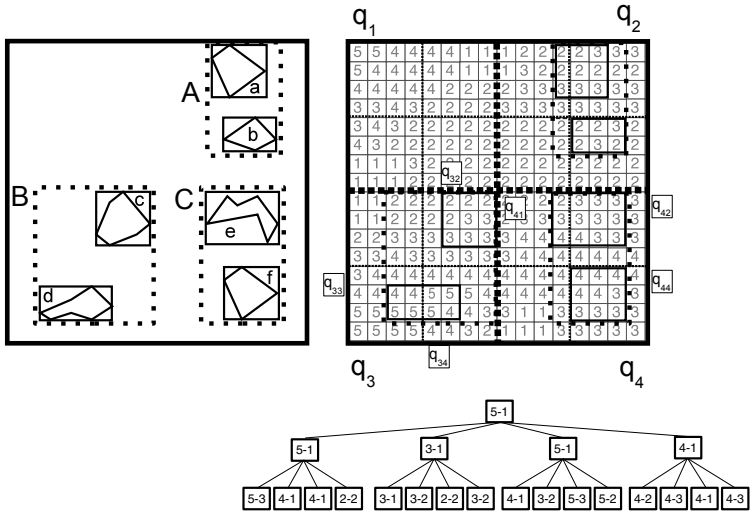


Figure 10.1: The MBRs of an R-tree (left). A raster dataset with the divisions of the k^2-raster and its conceptual tree (right). The k^2-raster uses a hybrid configuration with $n_1 = 2$, $k_1 = 2$, and $k_2 = 4$. The last level of the k^2-raster is omitted for clarity.

On the other hand, in the case of the raster dataset, the k^2-raster stores and indexes the complete data. Thus, the algorithm can return the exact cells of the raster dataset that fulfill the query without any additional task.

Figure 10.1 shows a running example that will help us during our explanations. In the left part of the figure, we can see six polygons and the MBRs of an R-tree enclosing them. The rectangles A, B, and C are the MBRs of the children of the root of the R-tree. These rectangles are depicted with very sparse and thick dotted lines. The rectangles a, b, c, d, e, and f are the MBRs of the children of A, B, and C. These rectangles are depicted with solid lines.

On the right, we can see a raster dataset. The densely dotted lines are the divisions of the space used by the k^2-raster. It uses a k^2-raster hybrid version, where $n_1 = 2$, $k_1 = 2$, and $k_2 = 4$. The first level division creates four quadrants, labeled q_1, q_2, q_3, and q_4 in the figure. These four quadrants are delimited with thick and densely dotted lines. Each of these four quadrants is divided into other four quadrants, for example, q_4 is divided into q_{41}, q_{42}, q_{43}, and q_{44}, and they are delimited with thinner densely dotted lines. These quadrants are 4×4 matrices and they are subdivided, when their minimum and maximum values differ, into 16 submatrices of size 1×1, since $n_1 = 2$ and $k_2 = 4$. Below the raster matrix, we also

show the conceptual tree of the k^2-raster. We intentionally omit the last level of the tree, composed of the 1×1 submatrices, in order to simplify the figure and since the values of the leaves are those shown in the raster. We also depict the MBRs of the vector dataset, superposed on the raster matrix and using solid lines. In this way, we can easily see when an MBR fulfills the query constraints.

10.1.1 Basic components of the algorithm

Next, we present some basic elements that will be used later in the description of the algorithm.

10.1.1.1 Pointers

Being p_r a pointer to an R-tree node, $p_r.MBR$ returns the MBR of that node and $p_r.ref$ returns the list of references to its children. Being p_k a pointer to a k^2-raster node, $p_k.quad$ returns the quadrant of that node.

10.1.1.2 Checking the overlapping

The most frequent operation of the algorithm will be to check whether the MBR of a node of the R-tree overlaps a region of the k^2-raster having cells with values in the queried range. For doing that, we need first to identify the smallest quadrant of the raster completely overlapping the MBR; and then, we need to check if any cell inside the quadrant, and overlapping also the MBR, stores one of the query values. A negative result means that it is possible to prune subtrees of both indexes.

It is critical that such a check be fast. Being a recurrent task in the algorithm, a slow check would spoil the running times. So, we will perform, when possible, a fast and "course-grained" check (*checkQuadrant*), and only when this is not enough, a more thorough, and thus more costly, "fine-grained" check (*checkMBR*).

The operation *checkQuadrant*(p_r, p_k, *Range*) takes a pointer p_r to an R-tree node, a pointer p_k to a node of the k^2-raster, and the query range. It returns a pair $\langle p_{kd}, typeOverQuad \rangle$. The component p_{kd} is a pointer to the deepest descendant of the node pointed by p_k that completely contains $p_r.MBR$. The component *typeOverQuad* can have one the following values:

- *None* means that $p_{kd}.quad$ does not have cells with values in the queried range. Therefore, we can conclude without any further inspection that $p_r.MBR$ does not overlap a portion of the raster having cells with values in the queried range, and thus the subtree rooted at p_r can be pruned.
- *Possible* means that $p_{kd}.quad$ contains cells with values in the queried range, but it also contains cells with values outside of the queried range. This value does not allow to take a decision and thus the algorithm has to perform a deeper analysis.

- *Full* means that $p_{kd}.quad$ contains exclusively cells with values in the queried range. Therefore, this value also allows to take a decision, as all the MBRs in leaves of the subtree rooted at p_r and the overlapping cells are part of the solution, actually of the *definitive* list, and the check of that subtree can be stopped.

The operation *checkMBR*(p_r, p_k,*Range*) takes a pointer p_r to an R-tree node, a pointer p_k to a node of the k^2-raster, and the query range. It returns a value of *typeOverMBR*, with the following meaning:

- *None* means that the geometry of $p_r.MBR$ does not overlap cells having values in the queried range. Therefore $p_r.MBR$ is not part of the solution.
- *Partial* means that $p_r.MBR$ overlaps cells with values in the queried range, but it also overlaps cells with values outside of the queried range. Therefore, $p_r.MBR$ and its overlapping cells with values in the queried range are part of the list of *probable* results.
- *Full* means that $p_r.MBR$ overlaps exclusively cells with values in the queried range. Therefore, $p_r.MBR$ and its overlapping cells are part of the list of *definitive* results.

The operation *checkQuadrant* is very fast, since it only checks the min-max values of the internal nodes of the k^2-raster. From the node of the k^2-raster provided as input, it traverses the tree downwards following the *unique* child that completely contains the input MBR, as long as the query range intersects the range delimited by the minimum and maximum values of the node. Once it reaches a node where none of its children completely contains the MBR or the query range does not intersect the range defined by the minimum and maximum values stored at the node, then the operation ends.

The operation *checkMBR* is more complex because it must navigate downwards *all* the k^2-raster branches that intersect with the MBR and retrieve all the cells in the k^2-raster that intersect the MBR, which may require extracting portions of different quadrants.

Let us illustrate this operation with the example at Figure 10.1. Let us suppose that we have a pointer to the node with MBR b, a pointer to the root node of the k^2-raster (which encloses the whole raster), and the query range [4–5]. Then, *checkQuadrant* checks if the min-max values of the root define a range overlapping or including the queried range. Since $min = 1$ and $max = 5$, then the search continues and it checks if one of its children completely encloses b. This is true for the child corresponding to quadrant q_2. Next, it checks if values from q_2 are contained within the queried range. Since $min = 2$ and $max = 3$ for q_2, therefore, *checkQuadrant* ends and the result is *typeOverQuad= None*. Observe that without checking the actual cells of the raster, and already at the second level of the k^2-raster, we can discard b as part of the solution.

Now, let us suppose another example taking as input a pointer to the node with MBR d, a pointer to the root of the k^2-raster, and the query range [4–5]. As before, the range defined by the min-max values of the root overlaps (but not includes) the query range, then the search continues. The child of the root including d is q_3. Again, we check if the min-max values of q_3 overlap or include the query range. Quadrant q_3 is the deepest node that completely contains d and the min-max values at q_3 are min=1 and max=5, thus, *typeOverQuad = Possible*. Since *checkQuadrant* cannot determine the result of the join for this case, it will be necessary to use *checkMBR*, which takes as input a pointer to the node of d and a pointer to q_3. The answer will be *typeOverMBR= Full*, and thus d and its overlapping cells are added to the *definitive list*.

10.1.2 The algorithm

Algorithm 10.1 shows the pseudocode of the algorithm. It receives a pointer to the root of both trees, and the queried range. Line 1 defines the variables holding the output. Each list is formed by entries with an MBR, the object (or objects) in that MBR, and the list of overlapping cells. Line 2 defines a stack that keeps pairs of pointers that have to be processed.

Lines 3–4 fill the stack with pairs, each of which has a pointer to the root of the k^2-raster and to one of the children of the root of the R-tree. Lines 5–22 are the main loop, which starts extracting the top of the stack and, with the two extracted pointers, calls *checkQuadrant*. If *typeOverQuad* is *Full*, then lines 9–12 add all the descendant leaves of the pointed node of the R-tree, together with their lists of object identifiers and the overlapping cells of the raster having values in the queried range, to the list of definitive results.

If *typeOverQuad* is *Possible*, the search must continue. Line 14 checks if the pointed node of the R-tree is a leaf or an internal node. If it is internal, the algorithm adds the children of that node along with p_{kd}, the pointer to the k^2-raster returned by *checkQuadrant*, to the stack and the flow returns to line 5.

If p_r is a leaf node, as explained, the algorithm has to perform a more detailed analysis in order to take a decision. Then it calls *checkMBR*. If the answer is *Full*, the MBR, its objects ids, and its overlapping cells having values in the queried range are added to the definitive list. If the output is *Partial*, the answer is added to the list of probable results.

Observe that the algorithm tries always to solve the query at the higher possible level of both trees, and that any call to *checkMBR* is delayed as much as possible.

Let us illustrate the algorithm with our running example of Figure 10.1 using [4–5] as query range. The stack starts with three entries containing a pointer to the root of the k^2-raster and, respectively, the pointers to the entries holding the MBRs A, B, and C. *checkQuadrant* with A returns q_2 and *None*, since the min and max values of q_2, which are 1 and 3, do not overlap the queried range, and thus it is discarded. For B, it returns q_3 and *Possible*. Since B is not in a leaf, then Line 16

Algorithm 10.1 *Join* (p_{rRoot}, p_{kRoot}, $[v_b, v_e]$)

```
1: Let D and P be lists of elements ⟨MBR, ListOfOids, ListOfCells⟩ ▷ The list of definitive and probable results
2: Let S be a stack with entries ⟨p_r, p_k⟩ ▷ p_r is a pointer to an R-tree node and p_k a pointer to a k²-raster node
3: for all p_rRef ∈ p_rRoot.ref do
4:     push(S, ⟨p_rRef, p_kRoot⟩) ▷ Inserts in the stack the children of the root node of the R-tree with a pointer to the root of the k²-raster
5: end for
6: while S ≠ empty do
7:     ⟨p_r, p_k⟩ ← pop(S)
8:     ⟨p_kd,typeOverQuad⟩ ← checkQuadrant(p_r, p_k, [v_b, v_e])
9:     if typeOverQuad = Full then
10:        if isLeafNode(p_r) then
11:            add(p_r, ExtractCells(p_r, p_kd), D) ▷ Adds the MBRs, objects, and overlapping cells to the definitive list D
12:        else
13:            addDescendants (p_r.ref, p_kd, D) ▷ Adds all MBRs, objects, and overlappings cells in descendant leaves to the definitive list D
14:        end if
15:    else if typeOverQuad = Possible then
16:        if isInternalNode(p_r) then
17:            for all p_rRef ∈ p_r.ref do
18:                push(S, ⟨p_rRef, p_kd⟩)
19:            end for
20:        else
21:            typeOverExact← checkMBR(p_r, p_kd, [v_b, v_e])
22:            if typeOverMBR = Full then
23:                add(p_r, ExtractCells(p_r, p_kd), D) ▷ Adds the MBRs, objects, and overlapping cells to the definitive list D
24:            else if typeOverMBR = Partial then
25:                add(p_r, ExtractCells(p_r, p_kd), P) ▷ Adds the MBRs, objects, and overlapping cells to the probable list P
26:            end if
27:        end if
28:    end if
29: end whilereturn ⟨D,P⟩
```

adds to the stack the children of B, that is, *d,c*, along with a pointer to q_3. With C, the answer is q_4 and *Possible*, then e and f are added to the stack coupled with a pointer to q_4.

With c and q_3, *checkQuadrant* returns q_{32} and *None*, and thus this pair is discarded. With d, *checkQuadrant* returns q_3 again (no child of q_3 completely contains d) and *Possible*. Since d is already in a leaf, *checkQuadrant* does not allow to take a decision, and a more detailed analysis is needed. Then, in Line 18, the algorithm calls *checkMBR*, which returns a *Full* value, and thus d along with the overlapping cells are added to the list of definitive results.

The *checkQuadrant* call of e and q_4 returns q_4 and *Possible*. Therefore, *checkMBR*

is used again, obtaining a *Partial* value, and thus e is added to the probable list in Line 22. With f and q_4 the output of *checkQuadrant* is q_{44} and *Possible*, so again, the algorithm runs *checkMBR*, which returns *Partial*, and then f is added to the probable list.

Chapter 11

Experimental evaluation

11.1 Raster data compression

We measured the space and time results obtained by the two different versions of the proposed data structure, k^2-raster and k^2_H-raster, and compared them to those obtained by previous compact data structures for raster datasets: k^2-acc and k^3-tree.

11.1.1 Experimental Framework

We ran different experiments to measure the space consumption, construction time, and the navigational time to answer these four types of queries:

- *getCell*: given a position in the raster matrix, this query obtains its cell value. The time was measured by performing 1,000,000 different random queries and we report the average time per query (in microseconds).
- *getWindow*: given a region or window of the raster matrix, this query retrieves all cell values within that window. We measured the time for 100 random queries and report the average time per retrieved cell (in nanoseconds).
- *searchValuesInWindow*: given a range of values and a region of the matrix, this query retrieves all raster positions belonging to the given region whose value lies within that range. We have defined two variants of this query: without any restriction for the range of values and window size, and limiting the range length to 200 and the window size to 500×500. In the first case, we measure the time for 10,000 random queries, and for the second case we measure the time for 100,000 random queries. We report the average time per retrieved cell (in nanoseconds).
- *checkValuesinWindow*: given a region and a range, this query checks if all cell values of the region are within the range of values (we call this variant

strong checkValuesinWindow) or if there exists at least one cell value in the region whose value lies within the range of values (*weak checkValuesinWindow*). The time was measured by performing 1,000,000 random different queries and obtaining the average time per query (in microseconds).

Queries *getCell* and *getWindow* illustrate the impact on the time to access and recover the original information when we represent the raster matrix with each of the techniques, since they keep the information compressed. Queries *searchValuesInWindow* and *checkValuesInWindow* illustrate the indexing capabilities of each representation.

All the experiments were run on a dedicated Intel® Core™ i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache, and 64GB of RAM. It ran Ubuntu 12.04.5 LTS with kernel 3.2.0-115 (64 bits), using gcc version 4.6.4 with `-O9` options. Time results refer to CPU user time. Space consumption was measured in compression percentage, computed as the ratio (in percentage) between the uncompressed size of binary file containing the original raster matrix and the size of the compressed representation.

We used a hybrid configuration for k^2-raster and for k^2_H-raster, with $k_1 = 4, k_2 = 2, n_1 = 4$. They used an implementation for supporting *rank* operations that adds 5% of extra space on top of the bit sequence T and provides fast queries [GGMN05][1]. In addition, *Lmax* and *Lmin* were encoded using the version of DACs that optimizes the space usage while restricting the maximum number of levels. More precisely, we have limited the number of levels to 3. We compared both variants of our proposal with k^2-acc and k^3-tree using the same hybrid configuration. In addition, we configured parameter $S = 14$ for k^2-acc, which is a parameter used to divide the input raster into 2^S subrasters, each one producing a set of k^2-trees.

11.1.2 Datasets

We used real data in our experiments. More concretely, we used data of different nature from the following two sources:

- WorldClim[2] dataset [HCP+05], which provides a set of layers with global climate information. The whole world is divided into equal-spaced tiles, and each cell of a tile is an integer number and has a resolution of about 1 square kilometer. Specifically, we have used the dataset containing the value of the mean temperature, which was measured in degrees Celsius with one decimal, and is represented using integers by multiplying the value by 10.

[1] If more space and less time are desired, one could replace the implementation by another that that uses 37.5% extra space and is much faster.

[2] http://www.worldclim.org/tiles.php

- Spanish Geographic Institute[3] (SGI), which includes several DTM (Digital Terrain Model) data files that contain the spatial elevation data of the terrain of Spain, stored as rectangular equal-spaced tiles with 5 meters of spatial resolution. Each cell of a tile contains a real number of at most three decimal digits.

In our experiments, we analyzed the scalability and behavior of each technique when varying the size of the input raster matrix and the number of different values included in the raster. Thus, we have created several collections of datasets of different nature with different properties of size and number of different values. More specifically, we have joined different adjacent tiles to create raster matrices of different sizes, and we have considered different precision by using different number of decimal digits, in the case of the dataset of spatial elevation values, to obtain variability on the number of different values. Table 11.1 and Table 11.2 show the average values of the main properties (size, number of rows, number of columns and number of different values) for the collection of datasets generated. Specifically, 1×1 matrices were built using just 1 tile, 2×2 matrices were built using 2×2 adjacent tiles, and so on. Each collection was created using a set of different adjacent tiles. For example, $\texttt{cat}_0\texttt{-1×1}$ is composed of 25 datasets, each corresponding to a different tile, and the data shown below represent the mean values obtained by those 25 datasets. This allows us to report the average space and time results obtained in the experiments for each collection, avoiding the dependence on the selection of a unique matrix. The dataset at Table 11.1 will be denoted as `eua` in the experiments, while the datasets at Table 11.2 will be denoted $\texttt{cat}_0$ and $\texttt{cat}_3$. The subscript for these datasets indicates how many digits of the decimal digits were considered. By considering more or less, we increase or decrease, respectively, the number of different values existing in the raster matrix. Thus, we will report the results when using 0 decimal digits ($\texttt{cat}_0$) and 3 decimal digits ($\texttt{cat}_3$). Notice that $\texttt{cat}_3$ corresponds to raster matrices of the original dataset.

In addition, to analyze the behavior of the methods when only the number of different values is varied, but not the matrix size, we generated a collection of matrices from just one random tile (namely, the one denoted as `MDT05-0533-H30-LIDAR`). More concretely, we have first truncated the original values by taking only the two most significant decimal digits. Then we have created other 5 raster matrices $\texttt{MDT05-0533-H30-LIDAR}_{\gg x}$ by shifting x bits of the value of each cell, for $x = 1, 3, 5, 7, 9$. By doing this, we have generated a collection of matrices with the same size and different number of different values.[4] We have not used the original values with all their precision due to the problems of k^2-acc and k^3-tree for running over datasets with a high number of different values. We denote this dataset as $\texttt{MDT}_x$ in the experiments, and show its properties in Table 11.3.

[3] http://www.ign.es

[4] Notice that by shifting x bit each value is divided by 2^x, thus decreasing the number of different values in the raster.

Table 11.1: Properties of dataset `eua`, obtained from WorldClim datasets. It includes raster matrices of different size and number of different values of the input matrix.

Name	size (MB)	#rows	#cols	# different values
`eua-1×1`	49.44	3,600	3,600	252
`eua-2×2`	197.75	7,200	7,200	413
`eua-3×3`	444.95	10,800	10,800	474
`eua-4×4`	791.02	14,400	14,400	498

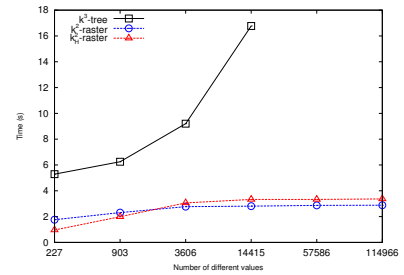
Table 11.2: Properties of datasets $\texttt{cat}_0$ and $\texttt{cat}_3$, obtained from DTM datasets. They include raster matrices of different size and number of different values.

Name	size (MB)	#rows	#cols	# different values
$\texttt{cat}_0$`-1×1`	91.49	4,100	5,849	868
$\texttt{cat}_0$`-2×2`	369.03	8,242	11,737	1,201
$\texttt{cat}_0$`-3×3`	834.76	12,403	17,643	1,503
$\texttt{cat}_0$`-4×4`	1,488.94	16,564	23,564	1,761
$\texttt{cat}_3$`-1×1`	91.49	4,100	5,849	779,405
$\texttt{cat}_3$`-2×2`	369.03	8,242	11,737	1,066,043
$\texttt{cat}_3$`-3×3`	834.76	1,2403	17,643	1,304,704
$\texttt{cat}_3$`-4×4`	1,488.94	16,564	23,564	1,545,248

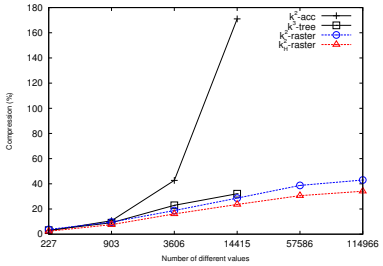
11.1.3 Construction time

Figure 11.1(left) shows the comparison among all the methods when measuring the construction time. Plots only show the results for k^2-raster, k^2_H-raster, and k^3-tree, as the times obtained by k^2-acc were more than 2 order of magnitude slower. Moreover, k^3-tree and k^2-acc were not able to create the compressed representation of those raster matrices with a large number of different values, more concretely, they failed when constructing the compressed representation for $\texttt{MDT05-0533-H30-LIDAR}_{\gg 0}$, $\texttt{MDT05-0533-H30-LIDAR}_{\gg 1}$, and all the matrices from dataset $\texttt{cat}_3$.

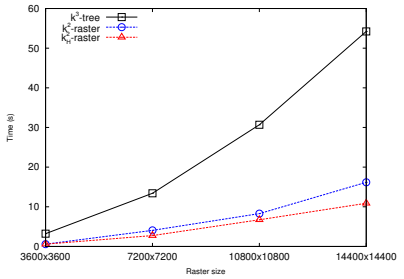
The construction process for k^2-raster and k^2_H-raster is the same, except for the last level of the representation. k^2-raster processes this last level analogously to the rest, whereas k^2_H-raster needs to create a vocabulary to compress the submatrices



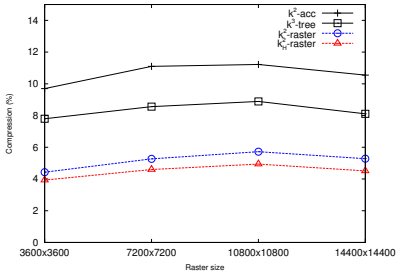
(a) $\texttt{MDT}_x$ – construction time



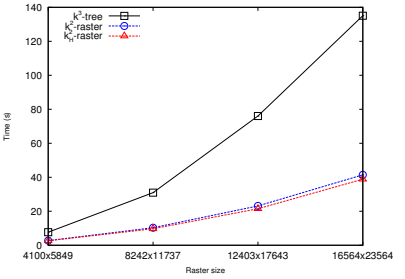
(b) $\texttt{MDT}_x$ – compression



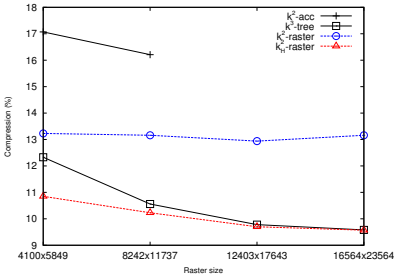
(c) `eua` – construction time



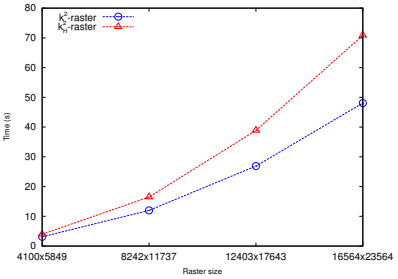
(d) `eua` – compression



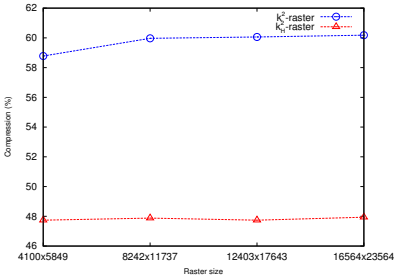
(e) $\texttt{cat}_0$ – construction time



(f) $\texttt{cat}_0$ – compression



(g) $\texttt{cat}_3$ – construction time



(h) $\texttt{cat}_3$ – compression

Figure 11.1: Construction time (left) and compression percentage (right) for datasets of different nature.

Table 11.3: Dataset $\texttt{MDT}_x$, obtained from tile `MDT05-0533-H30-LIDAR`. It includes raster matrices of the same size, but different number of values.

Name	size (MB)	#rows	#cols	# different values
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 9}$	86.48	3,881	5,841	227
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 7}$	86.48	3,881	5,841	903
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 5}$	86.48	3,881	5,841	3,606
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 3}$	86.48	3,881	5,841	14,415
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 1}$	86.48	3,881	5,841	57,586
$\texttt{MDT05-0533-H30-LIDAR}_{\gg 0}$	86.48	3,881	5,841	114,966

corresponding to $Lmax[Lst]$. Thus, at that point, the construction time differs between the two structures. To analyze their behavior, we ran two types of experiments, one where we only varied the number of different values, and another where we also varied the size of the input matrices.

For the first experiment we used dataset $\texttt{MDT}_x$, described at Table 11.3. This dataset contains raster matrices of the same size that differ on the number of different values. The results are shown in Figure 11.1(a). The y-axis shows the time consumption for constructing the compressed representation (in seconds) and the x-axis shows the number of different values for each dataset.

As expected, when increasing the number of different values, the construction time worsens. This happens because it is more likely that a submatrix has more than a single value, which must be divided and processed again. k^2-raster and k^2_H-raster behave similarly, and clearly outperform k^3-tree, which shows a scalability problem. As previously mentioned, k^2-acc was not included in the plot due to its bad performance. k^2_H-raster achieves better results when there is a small number of different values, but it becomes worse than k^2-raster for higher values of different values. This is due to the fact that the size of the vocabulary grows when the number of different values increases, thus, the construction process, explained in Section 9.2, becomes slower. There are no scalability issues due to this parameter for our proposed structures, as times become almost constant when increasing the number of different values.

We show in Figures 11.1(c), 11.1(e), and 11.1(g) the time consumption to build datasets from collections `eua`, $\texttt{cat}_0$, and $\texttt{cat}_3$ respectively. The y-axis shows the construction time (in seconds) and the x-axis shows the size of each dataset. We can also see that k^2-raster and k^2_H-raster obtain similar results, and better than those obtained by the methods of the state of the art. Neither k^3-tree nor k^2-acc are able

to create the compressed representation for collection $\texttt{cat}_3$, which includes raster matrices with a high number of different values. Hence, our proposals show again that are more convenient for real datasets containing a high number of different values.

11.1.4 Space requirements

Figure 11.1(right) shows the compression obtained by the four methods, k^2-acc, k^3-tree, k^2-raster and k^2_H-raster over all the datasets. Figure 11.1(b) shows the results for dataset $\texttt{MDT}_x$, where the number of different values grows while maintaining the size of the raster matrix. With 227 values, the four methods obtained a similar result, around 3% of the original collection size. When the number of different values grows up to 903 values, k^2_H-raster begins to obtain better results regarding the other methods. k^2_H-raster achieves a compression of 7.5% while for the rest of the structures is around of 9%. With the third raster matrix, which contains 3606 different values, the compression of k^2-acc is significantly worse (43%). Again, k^2_H-raster obtains the best compression (16%), followed by the k^2-raster (19%) and k^3-tree (32%). This tendency continues with the fourth raster matrix of the dataset. Thus, using a vocabulary-based approach and a selection heuristic improves the compression up to 9% with respect to the standard k^2-raster and both structures obtain better results than the techniques of the state of the art. In addition, our techniques were able to create the compressed representation for all datasets, including those with a high number of different values.

We also show the comparison of the compression obtained over the other three collections. As expected, k^2_H-raster obtains the best compression for all datasets. Moreover, k^2-acc can only represent the smallest matrices from datasets $\texttt{cat}_0$, and none from $\texttt{cat}_3$, whereas k^3-tree is not able to represent any matrix from dataset $\texttt{cat}_3$. These experiments demonstrate that our solutions can deal with large datasets, rather than the current state of the art. In addition, k^2-raster and k^2_H-raster maintain good compression ratios even when the number of values of the dataset grows.

11.1.5 Query times

In this section we show the results of the experiments for the queries described in Section 11.1.1. Again, we used datasets $\texttt{MDT}_x$, $\texttt{eua}$, $\texttt{cat}_0$, and $\texttt{cat}_3$, and compared the results obtained by our two methods, k^2-raster and k^2_H-raster, to those obtained by the techniques of the state of the art, k^3-tree and k^2-acc.

11.1.5.1 Time of *getCell*

Figure 11.2(left) shows the average time to retrieve the value of a given cell (in microseconds). k^2_H-raster outperforms the rest of the techniques for all cases, followed

closely by the standard version of k^2-raster. Our versions are up to 6 times faster than k^3-tree and 9 times faster than k^2-acc.

The query time of our techniques depends on the height of the tree; in the worst case, it needs to descend up to the last level of the tree, checking one node per level. As we can see in Figure 11.2(a), query times of k^2_H-raster, and also of k^2-raster, are almost constant even when the number of different values is high. In addition, k^2_H-raster uses a vocabulary, which decreases the number of tree levels; thus, the time for retrieving an individual cell becomes smaller than using k^2-raster, since searching inside the vocabulary is very efficient given that the values are kept in plain form.

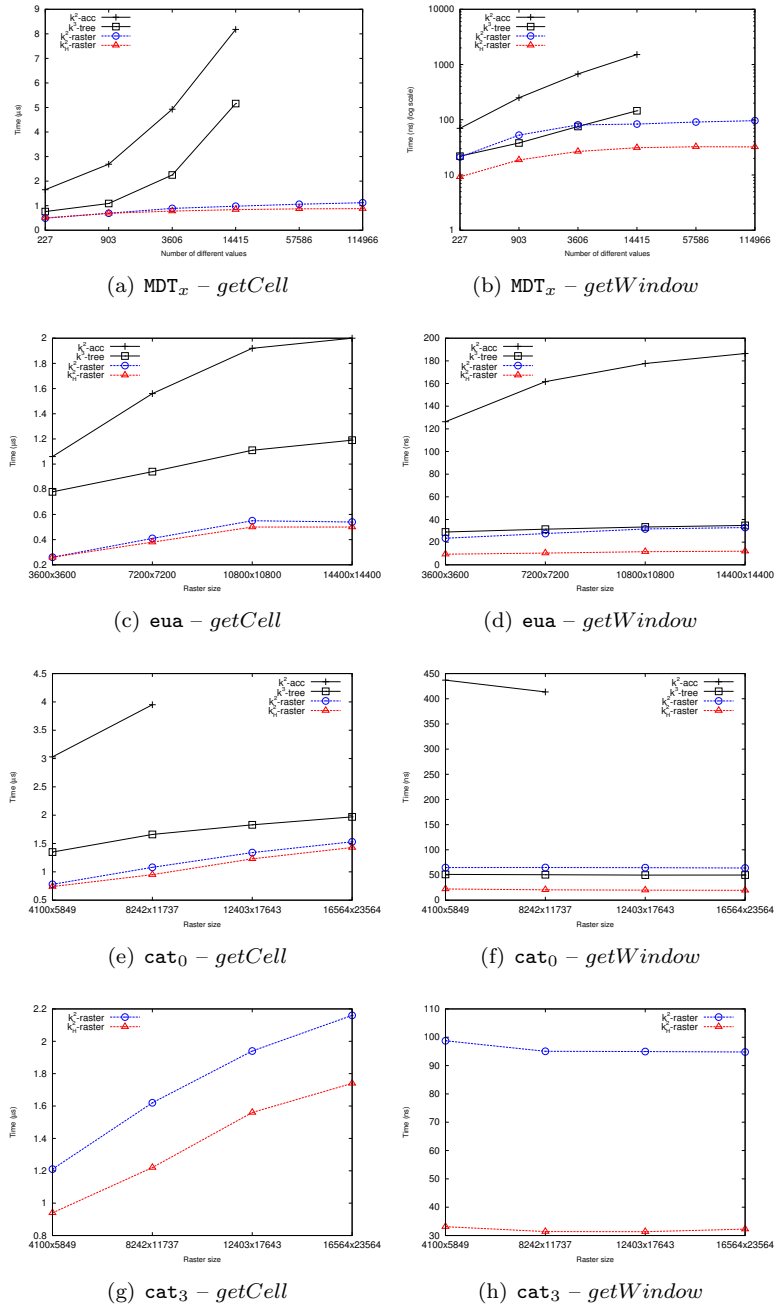
From the results, we can observe that k^2-acc and k^3-tree are not suitable for datasets with a large number of different values. To retrieve a value of a cell, k^2-acc performs a binary search among all its k^2-trees, and the number of k^2-trees depends on the number of different values. Thus, for datasets with a large number of different values, obtaining the cell value is slow. In the case of k^3-tree, the z dimension increases according to the number of different values; thus, the searching time by this dimension also grows.

Figure 11.2(c), Figure 11.2(e), and Figure 11.2(g) show the behavior for datasets with different input size. Again k^2_H-raster and k^2-raster obtain the best results.

11.1.5.2 Time of *getWindow*

Figure 11.2(right) represents the average time consumption to retrieve all values of a window (measured in nanoseconds per retrieved value). k^2_H-raster performs better than the other three methods for all datasets. k^3-tree gets time results similar to those of k^2-raster when the number of different values is small, but k^2-raster obtains better results with a high number of values, as it is shown in Figure 11.2(b). The other three plots of Figure 11.2(right) represent the behavior with different sizes for the input matrix. As expected, k^2_H-raster has the best performance. k^2-raster and k^3-tree obtain similar result whilst the time of k^2-acc is still the slowest by far. While the other methods know where to find the values of each cell, the k^2-acc needs to search all its k^2-trees until it finds the values corresponding to the cells that is searching, which is a very slow process.

An alternative procedure to obtain all the values of a region is to retrieve each value cell per cell. Comparing the results measured in time per cell retrieved by *getCell* at Figure 11.2(left) with those obtained by *getWindow* at Figure 11.2(right), the query *getWindow* takes advantage of the fact that it is possible to obtain adjacent cell values with the same top-down traversal of the tree. Our structures obtain the final value of a cell when reaching a leaf node. If a leaf node, which represents a submatrix with all values equal, belongs to upper levels of the tree, k^2-raster and k^2_H-raster can complete part of the final result in just one step, without obtaining each value cell per cell.



(a) $\texttt{MDT}_x$ – *getCell* (b) $\texttt{MDT}_x$ – *getWindow*

(c) `eua` – *getCell* (d) `eua` – *getWindow*

(e) $\texttt{cat}_0$ – *getCell* (f) $\texttt{cat}_0$ – *getWindow*

(g) $\texttt{cat}_3$ – *getCell* (h) $\texttt{cat}_3$ – *getWindow*

Figure 11.2: Time results for *getCell* (left) and *getWindow* (right) over datasets with different size and number of different values. We show average time per cell retrieved in microseconds for *getCell* and nanoseconds for *getWindow*.

11.1.5.3 Time of *searchValuesInWindow*

This query retrieves all cells whose values lie within a given range. Figure 11.3 shows the time consumption per retrieved cell in nanoseconds. We show the results obtained when we do not limit the size of the window nor the range length (left part of the figure), and when limiting the range length to 200 and the window size to 500×500 (right). We distinguish these two distinct scenarios, as time results show different behaviors. When selecting random ranges without any restriction, these ranges become larger when the number of different values grows. Thus, if the range is large, the query is usually answered in the upper levels of the representation, as there exist a vast amount of valid values that meet the search condition. In addition, the number of retrieved cells is higher, making the time/cell ratio smaller. On the other hand, if we limit the range length to 200, we avoid these two effects; thus, searching times worsen as the number of different values in the raster matrix grows, as the query becomes more selective. Collection `eua` contains very few different values (less than 500 different values); thus, all techniques behave similarly in these two scenarios, as restricting the range length to 200 produces almost no effect.

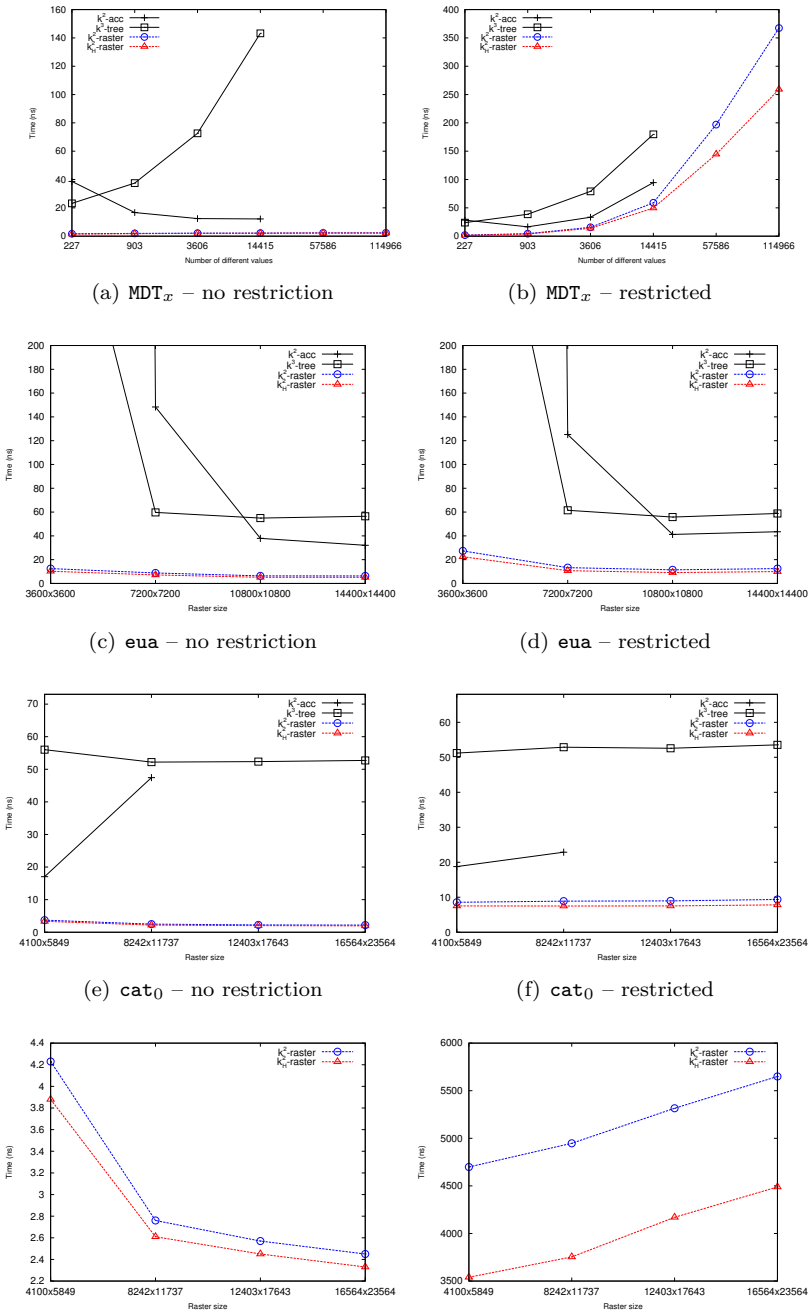
Our solutions perform better than the state of the art in all cases. With the indexation of the minimum and maximum values in the nodes of the tree, our structures are able to determinate if a region has any valid cell or even if all cells lie within the given range of values by only checking one node; in other case, they skip that node and continue the process with the rest of tree. Comparing the techniques from the state of the art, k^2-acc gets better results than k^3-tree in most datasets, especially when the number of different values is high, as it is shown in Figure 11.3(a). This is due to the fact that the k^2-acc only needs to check two k^2-trees, that is, the k^2-tree of the minimum value and the k^2-tree of the maximum value of the given range.

11.1.5.4 Time of *checkValuesInWindow*

Figure 11.4 shows the time to check if there exists at least one cell in the region whose value lies within the range of values (left) or if all cells of the region are within the range of values (right).

For the first case, which corresponds to *weak checkValuesInWindow*, k^2-raster, k^2_H-raster, and k^2-acc obtain very close results. k^2-acc obtains the best time for some datasets containing a small number of different values, more specifically, for datasets from collection `eua`, and some from collection $\texttt{cat}_0$. However k^2-raster, and k^2_H-raster are able to answer this query efficiently over datasets with a large number of different values or a large size. k^3-tree runs up to 40 times slower. This is the unique query where the standard k^2-raster is faster than k^2_H-raster in some case.

In the case of *strong checkValuesInWindow*, k^2_H-raster, k^2-raster and k^2-acc obtain similar results, while k^3-tree behaves constantly worse. Our structures use the information on the nodes (the maximum and minimum values) to check if the



(a) $\texttt{MDT}_x$ – no restriction

(b) $\texttt{MDT}_x$ – restricted

(c) `eua` – no restriction

(d) `eua` – restricted

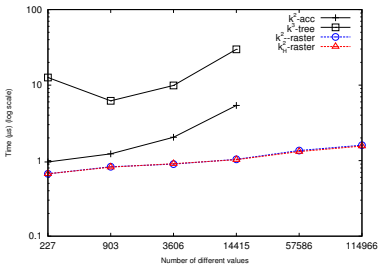
(e) $\texttt{cat}_0$ – no restriction

(f) $\texttt{cat}_0$ – restricted

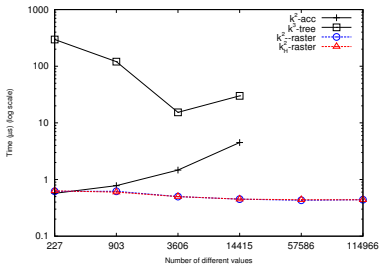
(g) $\texttt{cat}_3$ – no restriction

(h) $\texttt{cat}_3$ – restricted

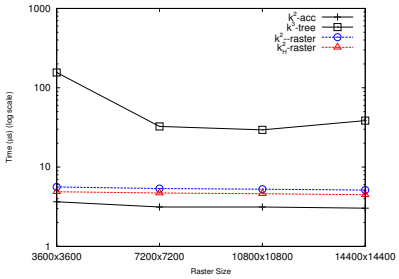
Figure 11.3: Time results for *searchValuesInWindow* using random windows and ranges without any restriction (left) and when restricting the maximum window size to 500×500 and the range length to 200 (right). Time results are measured in nanoseconds per retrieved cell.



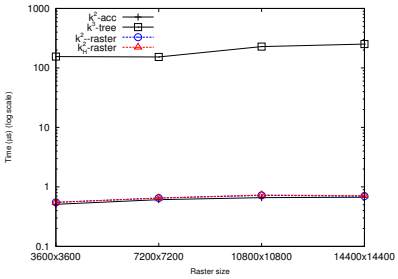
(a) $\texttt{MDT}_x$ – weak $checkValuesInWindow$



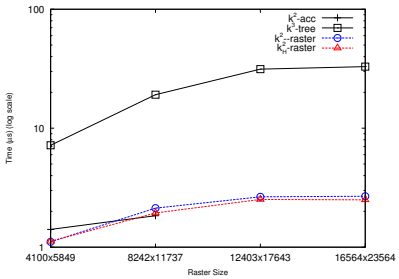
(b) $\texttt{MDT}_x$ – strong $checkValuesInWindow$



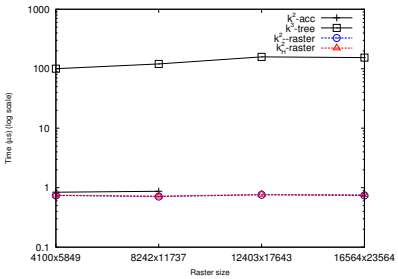
(c) `eua` – weak $checkValuesInWindow$



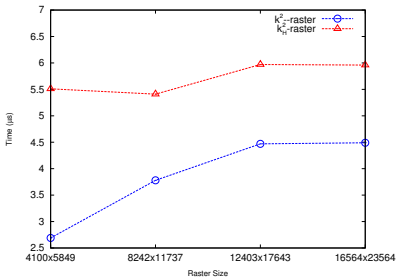
(d) `eua` – strong $checkValuesInWindow$



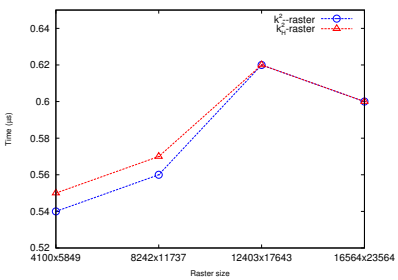
(e) $\texttt{cat}_0$ – weak $checkValuesInWindow$



(f) $\texttt{cat}_0$ – strong $checkValuesInWindow$



(g) $\texttt{cat}_3$ – weak $checkValuesInWindow$



(h) $\texttt{cat}_3$ – strong $checkValuesInWindow$

Figure 11.4: Time results for weak (left) and strong (right) $checkValuesInWindow$. Time results are measured in microseconds per query.

cells meet the conditions of the query. They are generally able to answer a query in the upper levels of the tree, without the need to descend to the last levels. This is the reason why k^2-raster obtains very similar results to those of k_H^2-raster for most of the datasets, as they only differ in the last levels of representation, which is rarely accessed in this query.

11.2 Spatial Join

We ran different experiments to measure the space consumption and processing time for answering join queries over raster and vector data of different nature.

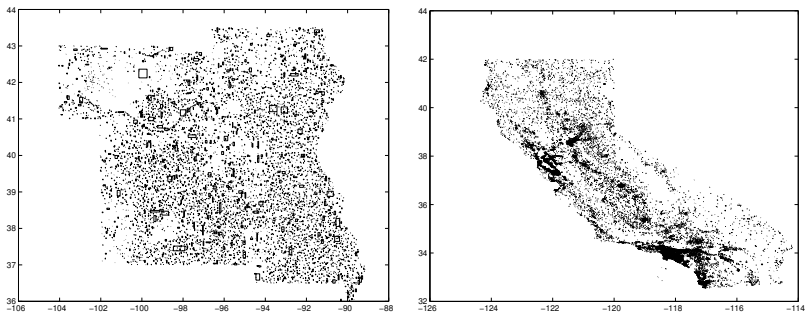
11.2.1 Experimental Framework

All the experiments were run on a dedicated Intel® Core™ i7-3820 CPU @ 3.60GHz (4 cores) with 10MB of cache, and 64GB of RAM. It ran Ubuntu 12.04.5 LTS with kernel 3.2.0-115 (64 bits), using gcc version 4.6.4 with `-O9` options. Time results refer to CPU user time (in seconds). Space consumption was measured as the peak memory usage ((*VmPeak*, in Megabytes). We report the average time and space results after executing 100 queries, were we varied randomly the queried range. We used the *R*-tree implementation from the *libspatialindex* library[5], setting the page size to 4 KB and the fill factor to 70%. We set a hybrid configuration for the k^2-raster, with $n_1 = 4$, $k_1 = 4$, and $k_2 = 2$, that is, it uses $k = 4$ for the first 4 levels and $k = 2$ for the rest. To navigate the tree, we used an implementation for supporting *rank* operations that adds 5% of extra space on top of the bit sequence T and provides fast queries [GGMN05]. In addition, we stored the maximum and minimum values using the version of DACs that optimizes the space usage while restricting the maximum number of levels to 3.

The closest related works cannot be compared against our approach, since the proposal in [CVM99] only considers binary rasters, and the proposal in [BdBG+17] returns just the values of the vector dataset. Therefore, in order to evaluate our framework, we developed also two baselines implementing the join operation over raster matrices which are stored uncompressed and processed directly in main memory (row by row). The first one, denoted as *RasterInt*, used 32-bit integers to represent the value of each cell in the raster. The second one, denoted as *RasterBits*, used $\lceil log(v) \rceil$ bits, being v the number of different values in the original matrix.

We compared our proposal to two baselines where the raster matrices were stored uncompressed and processed directly in main memory (row by row). The first one used 32-bit integers to represent the value of each cell in the raster. The second one used $\lceil log(v) \rceil$, being v the number of different values in the original matrix.

[5] https://libspatialindex.github.io



(a) `vects` (b) `vecca`

Figure 11.5: MBR distributions of the vector datasets `vects` (left) and `vecca` (right).

11.2.2 Datasets

We used real data in our experiments, both for the raster data and also for the vector data. The raster and vector datasets were scaled and translated in such a way that they cover the same space.

To properly analyze the performance and scalability of our approach, we have used two different sets of raster matrices defined in Section 11.1.2:

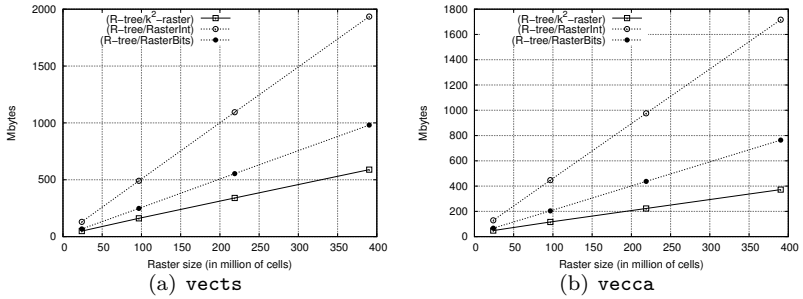
- Scenario I: varying the size of the input raster matrix, which also produces a slight variation of the number of different values appearing in the matrix. Table 11.2 shows the main properties (number of rows, columns, different values, and size with each technique) of this dataset.
- Scenario II: varying only the number of different values. We show their properties in Table 11.3.

The vector datasets were obtained from the ChoroChronos.org[6] web site. More concretely, we used *Tiger Streams* (`vects`) and *California Roads* (`vecca`). Figure 11.5 shows the distribution of the MBRs in these two datasets. We chose them because of the variety on their number of elements: 194,971 and 2,249,727 MBRs, respectively. They have been also used in the previously mentioned related work of Brisaboa et al. [BdBG$^+$17].

11.2.3 Memory usage

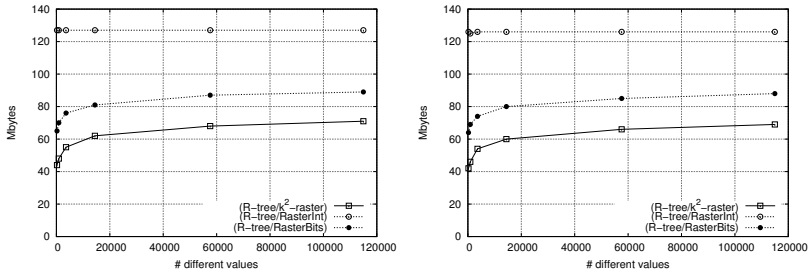
We measured the main memory required to perform a join operation using our approach, and compared it to the memory required by the two baselines for both

[6]http://www.chorochronos.org/



(a) `vects` (b) `vecca`

Figure 11.6: Memory consumption (in Megabytes) for rasters in Scenario I.



(a) `vects` (b) `vecca`

Figure 11.7: Memory consumption (in Megabytes) for rasters in Scenario II.

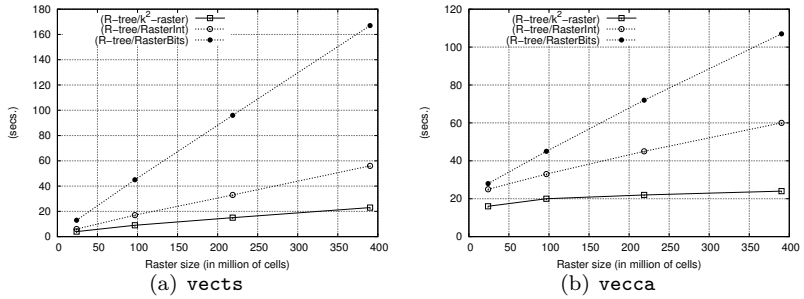
scenarios.

Figure 11.6 shows the results obtained for Scenario I, where we vary the size of the raster matrices. We can observe that our approach obtains always the best space, and it scales better than other approaches. Figure 11.7 shows than, when varying only the number of different values in the raster matrix in Scenario II, our algorithm also outperforms the baselines for both vector datasets, obtaining high compression especially when the number of values is low, and using around half the space than the 32-bit integer representation.

11.2.4 Time performance

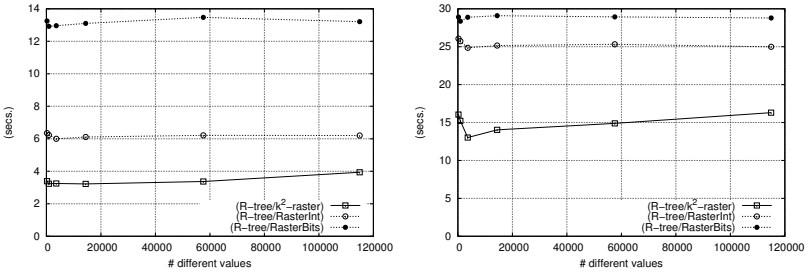
We also measured time performance when computing join operations for Scenario I and Scenario II. As we can see in Figures 11.8 and 11.9, our framework obtains the best time results and scalability properties for both scenarios.

The baseline using 32-bit integers, which requires much more space than the



(a) `vects` (b) `vecca`

Figure 11.8: Processing time (in seconds) with rasters of Scenario I.



(a) `vects` (b) `vecca`

Figure 11.9: Processing time (in seconds) with rasters of Scenario II.

one using $\lceil log(v) \rceil$ bits, obtains the best processing times among both baselines. However, it is clearly outperformed by our approach. Our algorithm using k^2-raster scales much better in Scenario I, when increasing the size of the raster matrix, and obtains also good processing times when increasing the number of different values in Scenario II. These results demonstrate than k^2-raster not only obtains a compressed representation of the raster matrix, but also indexes the data, such that the join query can be computed much faster than using a plain (and fast) 32-bit representation.

We cannot directly compare our framework with the proposal in [BdBG+17] using k^2-acc, as we solve real join queries, returning those raster cells that satisfy each query, and not only the MBRs. But Scenario II, when increasing the number of different values in the raster matrix, allows us an indirect comparison. Remember that k^2-acc requires even more space than the uncompressed data with a large number of different values. Scenario II probes that our framework clearly scales much better than the uncompressed baselines. Moreover, we must take into account that if the framework of [BdBG+17] required the exact cells satisfying the query,

their time results would be even significantly worse, as they would require processing the MBRs to check whether the intersection holds or not for each cell, which is a costly task.

Chapter 12

Discussion

12.1 Main contribution

12.1.1 Raster data compression

In this thesis we propose a new compact data structure, denoted k^2-*raster*, which represents raster data in compressed way and offers efficient indexing capabilities. Our technique supports, within reduced space, fast retrieval of single cell values, decompression of regions of cells and also supports advanced searches, such as retrieving cells inside a region containing some specific value or checking the existence of values inside regions of the raster data. We have also presented a variant of the structure, called k^2_H-*raster*, which uses an entropy-based heuristic to create a vocabulary of common patterns in order to obtain further compression.

We have empirically compared our two variants with existing techniques from the literature, showing that both proposals clearly outperform the techniques from the state of the art. They not only obtain better space usage and query performance, but they also scale better when increasing the size of the input data or when the raster matrix contains a large number of different values. The scalability property is of extreme importance, as these characteristics appear when using real raster data. When comparing the two proposed variants, k^2_H-raster is the clear choice in all scenarios. The simpler proposal, k^2-raster, obtains better construction times, but the heuristic version obtains better spatio-temporal results.

12.1.2 Spatial Join

The possibility of managing raster and vector datasets in geographical information systems is a convenient feature, since it is well-known that each model is more adequate depending on the nature of the spatial data [Cou92]. However, commercial and open-source systems, and even the OGC standard [OGC10, OGC12], separate

both views and do not provide languages, data structures, and algorithms to perform queries that use information from both models.

The exception to this rule could be the zonal statistics operation of Map Algebra [Tom94] that is included in several systems. However, those systems internally translate the vector dataset into a raster dataset before running the operation.

In this part, we presented a framework that includes two data structures and an algorithm for running a join between vector and raster datasets, with no previous transformation of none of them. The good properties shown by this new approach are due to the use of compact data structures, which allows efficient processing in little space.

The closest related work cannot be compared against our approach, since the work in [CVM99] only considers binary rasters, and the proposal in [BdBG+17] does not perform a complete join. Thus, we compared our approach to two baselines that store and process the complete raster dataset in main memory using a row by row filling curve. One of the baselines uses 32-bit numbers for each cell and the other uses $\lceil log(v) \rceil$ bits per number, being v the number of different values in the original matrix. The first is very fast, as it is byte-aligned, whereas the second is space-efficient. In any case, our experiments show that our proposal clearly outperforms the two baselines in both main memory space and running time.

12.2 Future work

As future work, we will extend the proposed structure to other dimensions, for instance, to be used for spatio-temporal or 3D datasets. In addition, we will also study the adaptation of our data structure to distributed or dynamic environments.

Another interesting future research line would be to integrate our structure to perform queries of spatial data in the semantic web using, for example, the standard GeoSPARQL[1]. The current tools for this type of queries have several drawbacks, either they do not implement all the functionality or the query performance is very poor. We believe that our structure could improve both problems.

The election of the R-tree for indexing the vector dataset is a pragmatic choice, since it is the *de facto* standard for this type of data. However, as future work we will consider the use of modern compact data structures as a substitution for the R-tree.

[1] http://www.opengeospatial.org/standards/geosparql

Part III

Scientific data

Chapter 13

Introduction

This part presents two main contributions. The first is a compact representation of huge sets of functional data or trajectories of continuous time stochastic processes, which allows keeping the data always compressed, even during the processing in main memory. It is oriented to facilitate the efficient computation of the sample autocovariance function without a previous decompression of the dataset, by using only partial local decoding. This structure, which we call *Compact representation of Brownian Motion* (CBM), is presented in Section 14.1. The second contribution is a new memory-efficient algorithm to compute the sample autocovariance function, which is described in detail in Section 14.2.

We compare our C++ implementation, which receives as input CBM compressed data, with two baselines: *i)* the R implementation (in fact a C program), and *ii)* our own C implementation, both operating on plain data. The results of our empirical evaluation are shown in Chapter 15. Finally, Chapter 16 presents our conclusions and directions of future work.

The outline of this chapter is as follows. Section 13.1 presents the motivation for the use of compression in the context of empirical autocovariance computation for Brownian motion trajectories. More details about trajectories of Brownian motion are described in Section 13.2. Finally, Section 13.3 shows some related work in the compression field.

13.1 Introduction

In the last decade, we are attending to an exceptionally growing demand for large-scale data analysis, which is linked to the new field called Big Data. The need to process huge collections of data poses several challenges. On one hand, statistics and artificial intelligence communities continue to develop new methods and techniques

to analyze data. On the other hand, computer scientists have to adapt analytical algorithms to datasets with *data volume too large, data rate too fast, data too heterogeneous, and data too uncertain* the so-called *Volume, Velocity, Variability, and Veracity.*

Researchers or professionals working in Big Data must master many different techniques and skills. To facilitate their work, several packages appeared, mainly SAS[1], MATLAB[2], and R[3]. These packages are very useful, but they have scalability problems [KEW13]. For example, the installation and administration manual of R recommends loading into main memory datasets that occupy only 10–20% of the available RAM and warns that if the dataset exceeds 50% of the available RAM, the system will be unusable due to operation overhead, even the simplest ones. The solution to these problems is, in most cases, the use of parallel processing [DG08, KEW13, DXS+15, SETM13]. Parallel processing is a straightforward solution, probably due to the existence of a good set of available tools. However, while putting most of the efforts in this strategy, one is missing chances to improve the scalability by means of other techniques. The use of more evolved data structures and algorithms is losing the role that they had in the past when the hardware technology was more limited.

Compression of floating point numbers has been proven difficult, mainly because the datasets usually contain many distinct values and with few repetitions. These two features make sequences of floating point numbers poorly skewed and, as a consequence, the entropy of those sequences is high, making them virtually incompressible with statistical compressors. Therefore, general purpose compressors may not succeed over sequences of floating point numbers. Instead, there are compressors that take advantage of properties of the data domain. Thus, there are compressors specially designed for images, video, or sound [Wal91, MPFL96, LR04, LI06], for general scientific data [EFF00, RKB06], or for more specific domains [YS08, MHP+11].

Although our method can be used for trajectories of any continuous time stochastic process, in this thesis, we rely on the characteristics of Brownian motion to develop data structures and algorithms especially suited for these data. Since the seminal work by Einstein [Ein06], the Brownian motion has been extensively used to model the movements of particles subject to instantaneous imbalanced combined forces exerted by collisions. Brownian motion and related stochastic processes have been successfully used to model the movement of colloidal particles or the trajectory of pollen grains suspended in water. Over the past forty years, starting with the papers by [BS73], the Brownian motion and related processes have been used to model option pricing and plenty of financial time series (see, for instance, [Hul09]).

Our method is designed to efficiently compute empirical moments from a sample

[1] http://www.sas.com/
[2] http://www.mathworks.com/products/matlab
[3] http://www.r-project.org

of observed trajectories of the stochastic process. One example of these moments is the empirical autocovariance function, which is a very important tool for functional principal component analysis. It can be used for dimension reduction, as in the Karhunen-Loève decomposition.

13.2 Brownian motion and autocovariance estimation

In real life, the Brownian motion can be used to model plenty of phenomena. It can be observed in microscopic particles that, when floating in a fluid, exhibit continuous but very jittery and erratic motion, since they are continuously bombarded by the fluid molecules. This natural phenomenon was formalized by Norbert Wiener in a rigorous mathematical way, as a stochastic process with continuous time.

The Brownian motion is a notion of central importance in probability theory, and it is used as a building block for a number of related random processes that are of great importance in a variety of applications in many fields, in pure Mathematics and in Applied Mathematics. Economics is one of the main applications of Brownian motion. It is used, for example, to predict the prices of financial products. In Medicine, it has been used in image analysis. Brownian motion has many applications in Engineering. For example, it can be used to model noise in electronics and instrument error. In Physics, for example, it is used to model the movement of little particles in a fluid or a gas, like in the aerosol transport phenomena.

13.2.1 Brownian trajectories

A Brownian trajectory (or curve) is just an observation of a Brownian motion stochastic process. In practice, this can be one of the components of a 3D motion, for example, the height of the particle, $X(t)$. These values make up a trajectory, and thus a trajectory is a function that, for every time instant, t, gives a real number. In practice, time is discretized and a trajectory is also discretized as a sequence of floating point numbers. Figure 13.1 shows an example of several Brownian trajectories.

13.2.2 Autocovariance function estimation

Brownian trajectories are randomly observed functions, so statistical analysis of them can be included in the field of functional data analysis. This is a very active research topic in modern statistics that focuses on analyzing complex and high dimensional data structures [RS05, FV06, HK12]. Classical important problems in this field are dimension reduction and supervised classification. These can be addressed using functional principal component analysis and functional data discriminant analysis (see [RS05, LGPBJ$^+$08] among many other). To carry out these techniques,

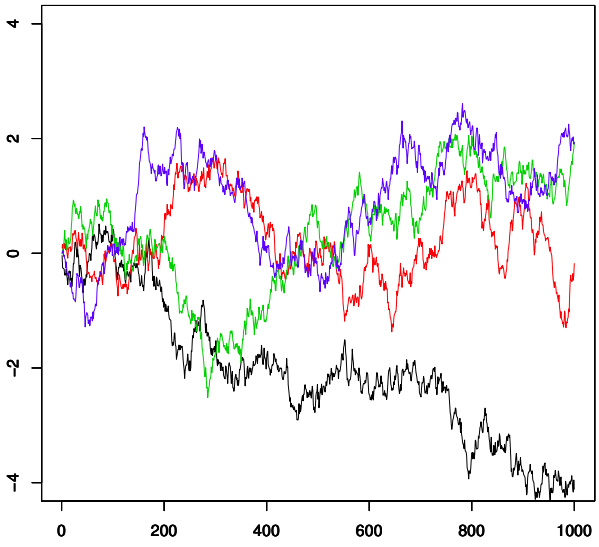


Figure 13.1: Four trajectories (curves) of a Brownian motion.

estimation of autocovariance functions or autocorrelation functions is needed. These are the extension of covariance matrices or correlation matrices to the context of functional data.

In this thesis, as an example of a statistical process that operates on functional data, we consider the autocovariance function estimation for Brownian trajectories. For a collection of trajectories $X_1, X_2, \ldots, X_n$, the value each one has at time $t \in T$, is represented by $X_1(t), X_2(t), \ldots, X_n(t)$, and the autocovariance function is estimated using Equation (13.1):

$$\hat{C}(s,t) = \frac{1}{n}\sum_{i=1}^{n}\left(X_i(s) - \overline{X}(s)\right)\left(X_i(t) - \overline{X}(t)\right), T, \tag{13.1}$$

for every $s, t \in T$, where $\overline{X}(s) = \frac{1}{n}\sum_{i=1}^{n} X_i(s)$. It is clear that direct computation of Equation(13.1), based on a large number of trajectories, n, for a large number, m, of instants $t \in T$ is a time-consuming process and it requires plenty of memory and disk. This is an important problem in Big Data analysis.

13.3 Related Work

13.3.1 Compressing Floating Point Numbers

General-purpose compressors may not perform well over floating point data, thus compression methods specifically designed to compress those numbers were developed following two main strategies. The first one is based on allowing some loss of precision [Wal91, MPFL96, MHP+11]. The second one uses a *predictor* that, before compressing a symbol, obtains a prediction of its value based on previous values, and then stores the difference between the prediction and the actual value [EFF00]. Space saving is obtained because such a difference is a smaller value than the original one and, in addition, it is usually compressed with some sort of encoder. Following this strategy, the well-known ALS compression method [LR04] of MPEG-4 combines two predictors and Golomb-Rice or Block Gilbert Moore coding. Lindstrom and Isenburg [LI06] presented a lossless compression method that uses a predictor, called Lorenzo, and encodes the difference with a two level compression scheme. The method proposed by [FM12] selects the best predictor from an available set, based on the values immediately compressed before the current value. In [YS08], a regression line computed from the last compressed values is used to predict the next value. The works in [RKB06, BR09, BR10] use a forecast system based on jump address predictors for CPUs.

Moreover, most compression methods designed for floating point numbers are not valid for in-memory processing, since they require to decompress from the beginning, and in some cases, they obtain slow compression and/or decompression times.

Several alternatives for compressing and indexing sequences of floating point numbers were presented in [FOP14]. This work uses data structures designed to index and compress text, but adapted to be used with the most significant part of the numbers, whereas the remainder part of the number is stored in plain form.

Concerning Brownian motion values and compression, [ABG+02] is the only related work. In this contribution, compression methods were used, but the target was to predict future values of stock shares, and not space saving.

Chapter 14

Our proposal: CBM

In this chapter, we introduce a new compact data structure, called *Compact representation of Brownian Motion* (CBM), for representing Brownian motion trajectories, which includes mechanisms to improve the calculation of empirical moments (more concretely, the sample autocovariance function). We explain our structure in Section 14.1, describing the construction process as well as the steps that must be followed for the calculation of the autocovariance function.

We also present a new memory-efficient algorithm to compute the sample autocovariance function that can be implemented using each trajectory only once. The pseudocode of this algorithm is described in Section 14.2.

14.1 Compact representation of Brownian Motion (CBM)

CBM is based on a very simple compression method, the differencing encoding. However, this technique cannot be directly applied. Observe that if we subtract two 32-bits floating point numbers, we obtain a new floating point number and therefore we still need 32 bits to represent it. To avoid this problem, we translate the floating point numbers into integers. As we will see later, we only deal with positive numbers, then we simply cast the floating point numbers to the integer that has the exact same 32-bit binary representation.

Next, we have to reduce the size of those 32-bit numbers. Using the typical prediction strategy to compress floating point numbers is a challenge since Brownian motion values are used precisely to model the randomness. Nevertheless, the processed values have an interesting characteristic, they come from a Brownian motion, and thus two consecutive values cannot differ too much. This fits quite nicely with the differencing encoding method, however, we do not use it directly. The differences of CBM are not with respect to the previous number, but to the previous

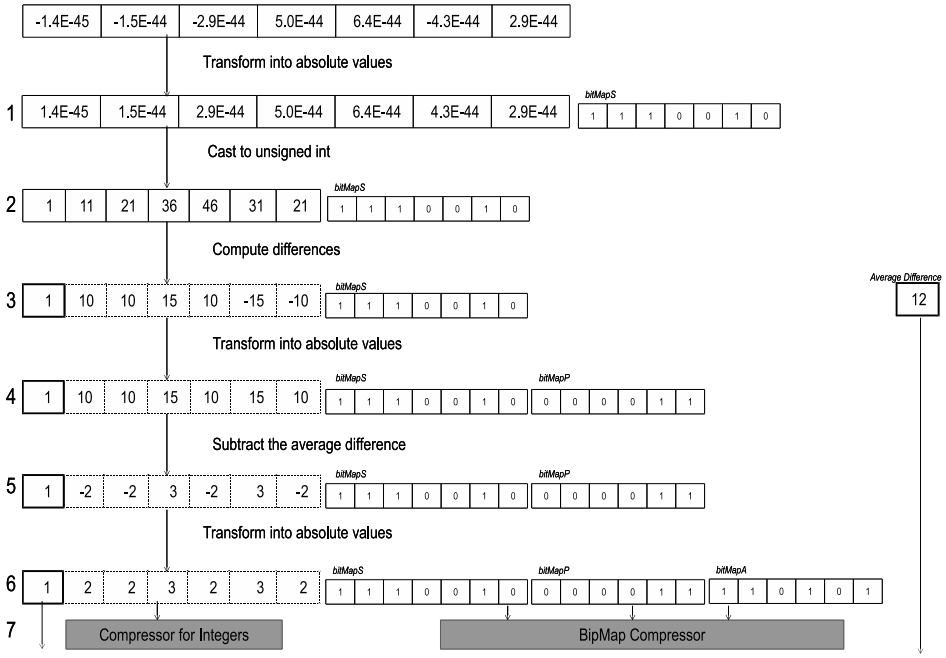


Figure 14.1: Compression process of a trajectory.

number plus the average of the differences between each pair of consecutive numbers of its trajectory. In other words, we use a prediction strategy, where the prediction computed to encode a given number is the previous number plus the average of the differences between each pair of consecutive numbers of the considered trajectory. That is, for each trajectory X_i, we compute the average of the differences between consecutive values at times $t-\delta$ and t for all $t \in T$ ($avgDiff(X_i)$). Then, to compress the value of that trajectory at time $t \in T$, denoted by $X_i(t)$, we make a prediction $P_i(t)$ from the previous value $X_i(t-\delta)$ as: $P_i(t) = X_i(t-\delta) + avgDiff(X_i)$, so we encode $X_i(t)$ as the prediction minus the actual value to encode $(P_i(t) - X_i(t))$.

CBM considers all numbers in absolute value, since the difference between two numbers of different sign yields larger differences. Observe, for example, that the difference between 1 and -2 is 3, whereas the difference between 1 and 2 is only 1.

Therefore, the first step of CBM translates the original numbers into positive values. For this, we have to options:

- To use a bitmap (*bitMapS*) to mark the positions of negative numbers.
- To use ZigZag encoding.

Figure 14.1 shows an example using bitmaps.

Algorithm 14.1 Compression

```
1: function COMPRESSION(t trajectories, #Points)
2:     for each trajectory t_i do
3:         First[t_i] = t_i[1]                                   ▷ Store first value of each trajectory
4:         for p ← 2, #Points do
5:             if t_i[p] < 0 then bitMapS[t_i, p − 1] ← 1        ▷ Mark the positions of negative values
6:             end if
7:             diffs[t_i, p − 1] ← |t_i[p]| − |t_i[p − 1]|       ▷ Compute differences
8:             if diffs[t_i, p − 1] < 0 then
9:                 bitMapP[t_i, p − 1] ← 1                       ▷ Mark the positions of negative values
10:                diffs[t_i, p − 1] ← |diffs[t_i, p − 1]|       ▷ Change of sign
11:            end if
12:        end for
13:        avgDiff ← avg(diffs[t_i])   ▷ Compute the average difference of the current trajectory
14:        for p ← 1, #Points − 1 do
15:            diffs[t_i, p] ←diffs[t_i, p]− avgDiff             ▷ Subtract the average difference
16:            if diffs[t_i, p] < 0 then
17:                bitMapA[t_i, p] ← 1                           ▷ Mark the positions of negative values
18:                diffs[t_i, p] ← |diffs[t_i, p]|               ▷ Change of sign
19:            end if
20:        end for
21:    end for
22:    Compression of bitMapS, bitMapP, and bitMapA
23:    Compression of diffs with a compressor for integers.
24: end function
```

The second step takes the 32-bit number representing each floating point number and casts it into an unsigned integer without any change in the bit number. Step 3 computes the differences between consecutive numbers and obtains the average difference of the trajectory, which in our example is 12. Observe that the first value of the sequence must be kept in plain.

Step 4 transforms the differences into positive values again. Therefore, we can use either a bitmap (*bitMapP*) or a ZigZag encoding.

Step 5 shows the result of subtracting the average difference from each difference. Converting the resulting values to absolute values requires the addition of another bitmap (shown in step 6), or the use of a ZigZag encoding again. Finally, the sequence of differences is compacted with a compressor for integers, the bitmaps (if they exist) are compacted with a bitmap compressor [GGMN05], and the first value of each trajectory and the average difference are stored in plain.

Algorithm 14.1 shows the pseudocode of the compression algorithm, where the bitmaps can be avoided by using ZigZag encoding. Since the compression method is based on differences, it is obvious that the decompression must start at a position with a number in plain form. In our method, we only store in plain form the first number of each trajectory, but we could store numbers in plain form at regular intervals, in order to be able to start the decompression at those points. This feature allows us to decompress portions of the input dataset. In our case, we can decompress trajectories individually. This allows saving main memory during the computation of any algorithm over the compressed sequence, since we can decompress only the trajectories needed at a given step of the algorithm. For this, it is important to have

Algorithm 14.2 Decompression

```
1: function DECOMPRESSION(#trajectories, #Points)
2:     for t ← 1, #trajectories do
3:         Values[t][1] ←First[t]                          ▷ Get first value of each trajectory
4:         lastProcNumber←First[t]
5:         avgDiff ←AvgDiffs[t]                  ▷ Get the average difference of the current trajectory
6:         for p ← 2, #Points do
7:             number←diffs[t, p − 1]                          ▷ Get the difference between points
8:             if bitMapA[t][p − 1] then      ▷ Check if is added or subtracted to average difference
9:                 number←avgDiff−number
10:            else
11:                number←avgDiff+number
12:            end if
13:            if bitMapP[t][p − 1] then     ▷ Check if is subtracted or added to the previous number
14:                number←lastProcNumber+number
15:            else
16:                number←lastProcNumber−number
17:            end if
18:            lastProcNumber←number                 ▷ The number is saved to calculate the next
19:            if bitMapS[t][p − 1] then                    ▷ Check if real number was negative
20:                number←-number
21:            end if
22:            Values[t][p] ←number
23:        end for
24:    end for
25:    return Values
26: end function
```

a fast decompression algorithm; otherwise, the computation times could be harmed.

Algorithm 14.2 shows the decompression algorithm. This process starts by taking the first number of a trajectory in plain. In line 5, we obtain the average difference of that trajectory, which was stored in plain for each trajectory during the compression procedure. Next, for each number in the compressed file, we perform the reverse process of that shown in the compression procedure. For decompressing a given number, we read the difference corresponding to that number and we add (or subtract) the average difference of the trajectory. Then that value is added (or subtracted) to the previous number. Finally, if the number was originally a negative value, then the sign is changed.

14.2 Memory-efficient computation of the sample autocovariance function

To compute the sample autocovariance function shown in Section 13.1, at two time instants, s and t, the values of all trajectories in these two time instants are needed. In addition, the classical implementation of that equation forces the use of each trajectory many times. In this way, the classical algorithms, implemented in R for example, maintain the whole dataset in memory. However, the equation can be implemented using each trajectory only once, accumulating in each entry of the output matrix the contribution to the sum due to the considered trajectory, as shown

Algorithm 14.3 Autocovariance computation trajectory by trajectory

```
1: function Covariance computation(t trajectories, #Points)
2:     for each trajectory t_i do
3:         for p ← 1, #Points do                          ▷ Traverses all the time instants
4:             for q ← p, #Points do                      ▷ Traverses the time instants from p to the end
5:                 cov[p, q] ← cov[p, q] + ((t_i[p] − avg_p) ∗ (t_i[q] − avg_q))
6:                 cov[q, p] ← cov[p, q]
7:             end for
8:         end for
9:     end for
10:    for p ← 1, #Points do
11:        for q ← 1, #Points do
12:            cov[p, q] ← cov[p, q]/#trajectories
13:        end for
14:    end for
15: end function
```

in Algorithm 14.3.

However, the algorithm requires the average value of all trajectories at all time instants (avg_t). To obtain those values, for each time instant, we would require loading the value of all trajectories in that time instant. This process should be performed in advance, which yields to a worsening of the running times. Instead, we compute this value during the compression process, and we add it to the compressed sequence, indeed it is required by the decompression procedure. In this way, the autocovariance computing process can read it from the compressed file.

CBM uses a compact data structure strategy, as it enriches the compact representation to improve the manipulation of the data. It takes advantage of the compression process to perform pre-calculations that will be useful during further processing of the dataset.

Note that this algorithm is not linked to CBM or the Brownian motion, as it can be used for trajectories of any continuous time stochastic process.

Chapter 15

Experimental evaluation

The experimental evaluation of our contributions is presented in this chapter. We compare the CBM (implemented in C++) with two baselines operating over plain data. More concretely, we use the autocovariation function included in the R package and our own C implementation of the algorithm.

This chapter is organized as follows: The setup of the experiments is shown in Section 15.1. Section 15.2 describes the collection of datasets used in the experimental evaluation and an analysis performed on them to give an idea about the behavior and distribution of the data. Next sections present the compression performance of CBM (Section15.3) and the experimental results of the execution of the autocovariance function, that is, the memory consumption (Section 15.4) and the computation time (Section 15.5).

15.1 Setup

Our test machine has an Intel® Core™ i7-3820@3.60GHz CPU (4 cores/8 siblings) and 64GB of DDR3 RAM. It runs Ubuntu Linux 12.04 (kernel 3.2.0- 121-generic). The hard disk was a Seagate ST3000DM001.

As compressor for integers, we used the following techniques: DACs (see Section 3.1), SCDC, PforDelta, and Kulekci using Rice and Elias encoding (see Section 2.1.4). We tested other compressors for integers, but they yield worse values.

By default, we used the three bitmaps: $bitMapA$, $bitMapP$, and $bitMapS$. For DACs and Kulekci with Rice encoding, we also provide the values substituting $bitMapP$ and $bitMapS$ by ZigZag encoding (we denote "-ZZ" these variants). We tested this approach for the rest of techniques, but the results were worse than the full bitmap versions or they did not run. We only substituted bitmaps $bitMapP$ and $bitMapS$, since if we use ZigZag encoding in the step corresponding to the $bitMapA$, the numbers will have a larger magnitude, and thus, since compressors for integers

are designed to compress small integers, most of them did not work, or if they ran, the results were poor.

We compare CBM against: *i)* GNU *gzip*[1], a Ziv-Lempel-based compressor; *ii) p7zip*[2], which is an LZMA compressor with a dictionary of up to 4 Gigabytes; and *iii) fpzip*[3], a compressor specially designed to compress floating point numbers. In the case of gzip, we used the default level of compression.

15.2 Dataset analysis

Table 15.1 shows the details of the datasets used in this study. Several thousands ($n = 10000, 20000, 30000$) of Brownian motion trajectories were simulated in $m =$ 1000, 2000, 10000, 15000, 20000, 30000 time instants in $T = [0, 10]$. The Brownian motion considered has zero mean and covariance function $c(s, t) = \min\{s, t\}$. It has been simulated by using independent normally distributed increments for contiguous time instants.

The first column of Table 15.1 shows the number of trajectories (n) and the number of time instants (m), with the form $n \times m$. The second column shows its size in MBs.

In order to give an idea of the hardness of compressing those datasets, we use Shannon's information theory (see Section 2.1.1) to measure the amount of information in those datasets. In particular, we used the zero-order empirical entropy (in bits/number).

To compute the entropy, we regarded the source file as a sequence of 1-byte, 2-byte, and 4-byte integers. The latter case considers the original numbers, but regarded as integers. For each case, we provide the empirical entropy in bits per number and the value of $\log(|\Sigma|)$, where $|\Sigma|$ denotes the size of the alphabet, that is, the list of distinct values found in the dataset. $\lfloor \log(|\Sigma|) \rfloor + 1$ gives the minimum number of bits required to represent each number using binary codes of the same length, which is adequate for uniform distributions.

When the original dataset is processed considering integers of 1 byte, the empirical entropy is around 7.38 bits per number, whereas $\log(|\Sigma|)$ is exactly 8, since in all datasets, the 256 possible 1-byte values are present. When the dataset is regarded as a sequence of 2-byte integers, the entropy is around 14.22 bits per number and $\log(|\Sigma|)$ is 16. Finally, in the case of 4-byte integers, the values of the empirical entropy are between 22.75–26.50 bits per integer. In this case, $\log(|\Sigma|)$ is not 32, since not all possible 32-bit values are present in the datasets.

Observe that in the case of 1-byte integers and 4-byte integers, the empirical entropy is very close to $\log(|\Sigma|)$, whereas in the case of 2-byte integers, there is a little gap. This is probably due to the nature of Brownian trajectories and the

[1] `http://www.gzip.org/`
[2] `http://p7zip.sourceforge.net/`
[3] `http://computation.llnl.gov/projects/floating-point-compression-zfp-fpzip`

Table 15.1: Dataset sizes and entropy.

		1-byte integers		2-byte integers		4-byte integers	
Dataset size	Size (MBs)	Entropy (bits/ 1b-int)	log(|Σ|)	Entropy (bits/ 2b-int)	log(|Σ|)	Entropy (bits/ 4b-int)	log(|Σ|)
10000 × 1000	38.15	7.38	8	14.22	16	23.14	23.17
20000 × 2000	152.59	7.39	8	14.22	16	24.83	24.94
10000 × 10000	381.47	7.39	8	14.21	16	25.66	25.86
10000 × 15000	572.20	7.38	8	14.22	16	25.93	26.18
20000 × 20000	1525.88	7.38	8	14.22	16	26.34	26.71
30000 × 30000	3433.23	7.39	8	14.22	16	26.50	27.00

Table 15.2: Entropy of the files of differences.

	1-byte integers		2-byte integers		4-byte integers	
Dataset size	Entropy (bits/ 1b-int)	log(|Σ|)	Entropy (bits/ 2b-int)	log(|Σ|)	Entropy (bits/ 4b-int)	log(|Σ|)
10000 × 1000	6.60	8	12.07	16	21.53	21.86
20000 × 2000	6.54	8	11.89	16	21.56	22.39
10000 × 10000	6.37	8	11.44	16	20.79	22.47
10000 × 15000	6.33	8	11.32	16	20.59	22.65
20000 × 20000	6.29	8	11.24	16	20.46	23.30
30000 × 30000	6.24	8	11.13	16	20.25	23.75

internal format of floating point numbers. In a Brownian trajectory, differences between close numbers are small, thus the most significant 16-bits of the 32-bit floating point numbers will vary less since that part includes the sign (1 bit), the exponent (8 bits) and the most significant part of the mantissa (7 bits). Therefore, the first 16-bit numbers will have a more skewed distribution and this is precisely where CBM obtains compression, like all compressors designed for floating point data.

Anyhow, it seems that the chances to compress are low, since the simple binary representation of the numbers is close to the amount of information they carry. This indicates that the original sequence has an almost flat distribution and a large amount of distinct numbers, and this is precisely the worst scenario to achieve compression. This is not surprising, since values simulated from a Brownian motion are inherently random.

Recall that our strategy can be divided in two main steps: first, we preprocess the original dataset in order to obtain a sequence of differences and second, we compress such sequence using a compressor for integers. The sequence of differences (shown at Step 6 in Figure 14.1) is analyzed in Table 15.2. We can see a decrease in the entropy, and thus the integer compressor will be more successful in this new preprocessed file. As an example, if we apply the SCDC compressor over the original sequence, the output is even bigger than the original file, around 125%, whereas if

Table 15.3: Compression ratio.

				CBM						
				DACs			Kulekci			Pfor
Dataset size	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	92.61	80.81	72.42	85.13	84.73	89.78	95.85	99.13	132.88	83.20
20000 × 2000	92.30	79.05	70.82	81.41	81.05	87.58	92.15	91.22	105.33	80.84
10000 × 10000	90.35	75.01	66.85	76.23	75.91	85.63	89.16	84.08	79.20	76.00
10000 × 15000	89.66	73.98	65.79	75.32	75.00	85.43	88.88	83.39	76.30	75.16
20000 × 20000	89.15	73.21	64.96	74.61	74.29	85.26	88.76	83.03	74.59	74.59
30000 × 30000	88.47	72.12	n/a	73.76	73.44	85.07	88.61	82.66	72.67	73.93

we apply it over the sequence of differences, the compression ratio is around 85% in large files, including the auxiliary bitmaps. These auxiliary bitmaps are the price that CBM has to pay, in part, for that decrease of entropy, which in the case of the SCDC version represents $11 - 11.5\%$ of the compressed dataset, and between $11.5 - 13.34\%$ in case of using DACs.

Another interesting effect of the preprocessing is that, when the original dataset is considered as a sequence of 4-byte integers, the entropy grows as the size of dataset increases. However, in the sequence of differences, the entropy decreases as the size of dataset increases, and this will be reflected in the compression ratio, as it will be shown next.

15.3 Compression performance

Table 15.3 shows the compression ratio. As we can see, fpzip obtains the best results, between 15% and 21% better than CBM, except in the largest dataset, where fpzip did not run. p7zip obtains also good results, yet CBM is very close, especially for the largest dataset, where the Kulekci-Elias version is on a par, and the PforDelta version is at most 8% worse. However, neither fpzip nor p7zip support partial decompression, essential to directly use the compressed data in main memory.

Comparing CBM variants, we observe that PforDelta and DACs-ZZ generally outperform the rest, although in the largest dataset the best result is obtained by the Kulekci-Elias version. SCDC and Kulekci-Rice obtain worse compression.

Table 15.4 shows the performance in compression time. Again the best method is fpzip, except in the largest dataset, where it did not run, and then gzip is the fastest. CBM pays the price of performing a compression process per trajectory. If CBM compressed the data of all trajectories in a unique run of the integer compressor, and thus producing a unique compressed dataset, it would be even faster than fpzip. However, in order to be able to load into memory the data of only

Table 15.4: Compression time (seconds).

				CBM						
				DACs			Kulekci			Pfor
Dataset size	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	1.59	6.28	0.74	1208.92	4848.92	14.21	5.30	5.31	6.30	11.46
20000 × 2000	6.31	25.81	3.29	2721.44	10514.35	39.04	14.84	14.93	18.75	44.46
10000 × 10000	17.17	66.07	8.39	1578.49	6191.75	73.53	24.42	24.84	33.46	103.31
10000 × 15000	26.13	98.36	12.83	1596.06	6232.87	102.14	34.99	34.83	47.69	152.22
20000 × 20000	68.38	365.18	37.71	3246.89	13229.90	256.13	91.02	91.31	123.80	401.15
30000 × 30000	155.19	915.98	n/a	4968.80	21806.09	518.64	198.20	197.19	271.69	889.26

Table 15.5: Decompression time (seconds).

				CBM						
				DACs			Kulekci			Pfor
Dataset size	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	0.45	2.02	0.72	0.26	0.24	0.19	0.94	1.59	8.42	0.14
20000 × 2000	2.14	7.97	2.80	1.08	0.89	0.69	4.24	6.14	32.39	0.60
10000 × 10000	5.19	19.36	6.89	2.52	2.07	1.61	8.20	10.49	75.07	1.45
10000 × 15000	7.78	28.74	9.98	3.74	3.14	2.40	10.68	13.62	109.90	2,16
20000 × 20000	13.21	81.34	28.13	10.69	8.14	6.42	28.18	34,81	289.06	5.98
30000 × 30000	50.36	187.55	n/a	22.06	18.24	14.68	59.55	71.64	633.63	13.01

one compressed trajectory, the trajectories have to be compressed isolatedly, which means a slower compression.

For our purpose of using compressed data in main memory, the key feature is the decompression performance, since a slow decompression process will harm the processing. Table 15.5 shows the decompression times. CBM-PforDelta is the fastest technique, between 2.2 and 3.9 times faster than gzip, which compresses between 9 and 15 percentage points less. fpzip, which obtains the best compression, is around 5 times slower than CBM-PforDelta. CBM-SCDC and CBM-DACs-ZZ also obtain good performances.

15.4 Memory consumption during the computation of the sample autocovariance function

Table 15.6 shows the maximum virtual memory[4] consumption during the computation of the sample autocovariance function using the classical algorithm

[4]This includes the complete space of addresses of the process.

(see Section 13.2.2). With this approach, the whole input dataset is stored in main memory.

We compare the R implementation, using plain data (binary representation of numbers), our own C implementation, and our implementations using CBM compressed data as input.

We also performed another experiment, which supposes that the dataset is stored on disk compressed with a classical compressor. Therefore, a previous full decompression is needed in order to obtain the uncompressed version, which is then processed with the normal C program. In this case, we give the highest value of memory consumption between the decompression process and the computation of the sample autocovariance function. These values correspond to the columns labeled as "C+gzip", "C+p7zip", and "C+fpzip".

The C and CBM implementations are basically the same C program. Both programs maintain the whole dataset in main memory, but the CBM version keeps the dataset in compressed form and only decompresses an individual trajectory when the algorithm requires those data in a given step.

The C implementation is on a par with CBM, there are two reasons for this. First, in both cases the output is kept in main memory uncompressed, and this is the biggest component of the memory consumption. Observe in Table 15.6 that when processing the dataset of size 30000×30000, both alternatives consume around 10 GBs. The input dataset requires around 3.3 GBs uncompressed and 2.5 GBs compressed. The output is stored in doubles (in order to be fair with R), and then it occupies 6.6 GBs, the biggest part. The second factor is that the advantage of CBM in the input size (0.8 GBs) is compensated by the fact that, at a given step of the algorithm, the CBM version has to decompress a treated trajectory, and therefore, that trajectory is stored twice in main memory. In addition, some auxiliary data structures are needed to perform the decompression.

Considering the experiment where the data are compressed with a classical compressor, the decompression process of gzip and p7zip has a smaller memory footprint than the computation of the sample autocovariance function. However, fpzip, which is the best compressor in disk space, consumes a large amount of memory, between 1.76 and 3.7 times more than CBM, and indeed it did not run with the largest file.

Finally, the R implementation is the worst one. It consumes between 1.87 and 3.71 times more memory than CBM.

The memory consumption of the memory-efficient version, which uses Algorithm 14.3, is shown in Table 15.7. Except for the small files, where the C implementation consumes less space than CBM, for the larger datasets they are on a par again.

In the case of "C+gzip" and "C+p7zip", even using the memory-efficient algorithm to compute the sample autocovariance function, the decompression process consumes less main memory, except with p7zip in small files. Obviously, now the

Table 15.6: Memory consumption (in MBs) of the classical algorithm.

						CBM						
			C +			DACs			Kulekci			Pfor
Dataset size	R	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	223	50	50	50	254	60	60	60	60	60	60	60
20000 × 2000	648	187	187	187	982	197	197	197	197	197	197	197
10000 × 10000	2545	1149	1149	1149	3204	1159	1159	1159	1159	1159	1159	1158
10000 × 15000	4060	2293	2293	2293	3777	2303	2303	2303	2303	2303	2303	2303
20000 × 20000	9096	4582	4582	4582	12782	4592	4592	4592	4592	4592	4592	4592
30000 × 30000	19358	10305	10305	10305	n/a	10315	10314	10315	10315	10315	10314	10315

Table 15.7: Memory consumption (in MBs) of the memory-efficient algorithm.

					CBM						
		C +			DACs			Kulekci			Pfor
Dataset size	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	12	12	37	254	25	23	25	25	23	25	25
20000 × 2000	35	35	37	982	59	49	60	59	49	59	59
10000 × 10000	767	767	767	3204	814	789	814	814	789	814	814
10000 × 15000	1721	1721	1721	3777	1787	1749	1787	1787	1749	1787	1787
20000 × 20000	3130	3130	3130	12782	3216	3116	3216	3216	3116	3216	3216
30000 × 30000	6919	6919	6919	n/a	7219	6994	7219	7219	6994	7219	7219

gap between "C+fpzip" and CBM is even bigger: CBM consumes between 87% and 5% of the space used by "C+fpzip".

15.5 Time to compute the sample autocovariance function

Table 15.8 shows the time required to compute the sample autocovariance function. In the experiments "C+gzip", "C+p7zip", and "C+fpzip", the displayed values correspond to the addition of the decompression times and the computation times of the sample autocovariance function using the C program over the uncompressed data.

CBM does not improve the results obtained by the C implementation regarding memory consumption, however in this experiment, CBM clearly beats the C implementation. There are two reasons for this. First, the computation of the covariance function requires the average value of all trajectories at all time instants (see Section 13.2.2). In the case of CBM, those values are computed during

Table 15.8: Computation time (seconds) for the sample autocovariance function with the classical algorithm.

						CBM						
			C +			DACs			Kulekci			Pfor
Dataset size	R	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	5	6	7	8	7	4	4	4	5	5	12	4
20000 × 2000	42	50	52	58	53	31	31	30	34	36	63	30
10000 × 10000	515	612	618	632	619	372	375	372	379	383	447	371
10000 × 15000	1155	1374	1382	1403	1384	834	845	834	844	854	942	834
20000 × 20000	4134	4863	4876	4944	4891	2945	2965	2948	2982	2994	3240	2947
30000 × 30000	14006	16488	16538	16676	n/a	9952	10233	9945	9984	10022	10847	9938

compression and stored in the compressed file. However, the C program has to calculate those values before running the main loop implementing Equation (13.1). The second factor is the memory hierarchy. While the output has a big impact in the memory footprint, that space is not critical for the running times. However, the input data is read repeatedly; thus, making the input data available to the processor as quickly as possible has a big impact in the running times. A smaller input has more chances of being stored at higher levels of the memory hierarchy.

With the largest dataset, the C implementation of the classical algorithm consumes up to 10 GBs of virtual memory, while the maximum resident memory[5], is 6.8 GBs. Therefore, the remaning 3.2 GBs must be stored on disk in the swap area. Moreover, those data are interchanged between disk and memory during the computation, implying an important slowdown. Even in the smallest dataset, the virtual memory peak was 50 MBs, while the resident memory peak was 44 MBs. These values are similar for the CBM version, but the number of interchanges between memory and disk could have an important impact. The same applies for the interchanges between memory and the different levels of processor cache, where the number of reads that are successfully solved in low level caches has a big impact.

CBM is between 50% and 66% faster than the C implementation and between 13% and 40% faster than R. Observe that R is faster than the C program. The reason is probably again the memory hierarchy: R uses almost twice as much main memory space as the C program.

If we consider that a previous decompression is needed before running the C program, the improvements are obviously better, CBM is between 1.67 and 2 times faster.

This experiment shows that the decompression required by CBM is really fast and therefore the computation time is not harmed.

[5]The space of RAM used by the process.

Table 15.9: Computation time (seconds) for the sample autocovariance function with the memory-efficient algorithm.

					CBM						
		C +			DACs			Kulekci			Pfor
Dataset size	C	gzip	p7zip	fpzip	bitmap	ZZ	SCDC	Rice-b	Rice-ZZ	Elias	Delta
10000 × 1000	3	4	5	4	6	4	4	5	5	12	4
20000 × 2000	42	45	50	45	44	44	45	47	49	75	43
10000 × 10000	487	493	507	494	490	520	507	519	518	583	488
10000 × 15000	1131	1139	1160	1141	1109	1139	1141	1158	1155	1267	1105
20000 × 20000	4088	4101	4169	4116	4015	4028	4085	4092	3982	4373	3936
30000 × 30000	14258	14308	14446	n/a	13744	15220	13838	14301	13587	15853	13568

Table 15.9 shows the times when using the memory-efficient version of the algorithm. Now, the gap between CBM and the C implementation is shorter. They are almost on a par, and only in the largest files, CBM is around 3.5% faster. In a given step of the algorithm, only one trajectory is loaded into memory, therefore, the C implementation has more chances to fit that trajectory in the upper levels of the memory hierarchy.

Again, resident memory usage supports this explanation. In the largest dataset, the peak consumption of virtual memory was 6,919 MBs, while the maximum resident memory consumption was 6,915 MBs. This implies that interchanges between the swap area and memory are almost inexistent, and then the memory-efficient algorithm using CBM is even faster than the C implementation of the classical algorithm, even though intuition says that should be faster since it has the complete dataset uncompressed in main memory.

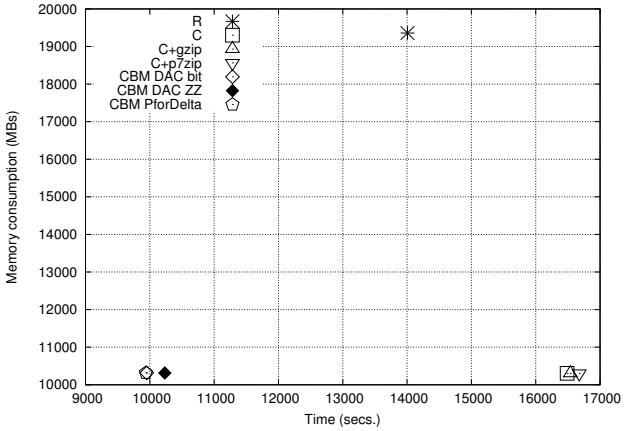
If we consider the experiment where the data were compressed with a classical compressor, then CBM is up to 13 % faster.

In Figure 15.1, we show the trade-off between memory consumption and computation time using the largest dataset with the two algorithms to compute the sample autocovariance function (classical and memory-efficient ones). We only provide the values of some of the versions of CBM to avoid cluttering the figure. In addition, results for "C+fpzip" are not present since fpzip did not run in this dataset.

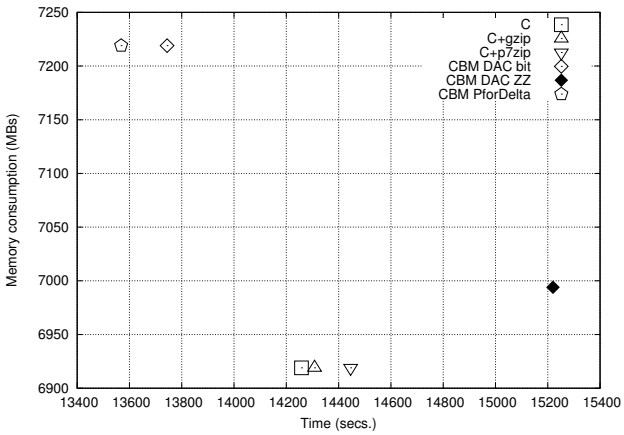
With the classical algorithm, we can see that CBM has clearly the best balance, as it has almost the same memory consumption as the C implementation, but better running times. Observe that R has by far a worse memory consumption.

In the memory-efficient version plot, R is not present. The differences are shorter, having the C implementation a slight advantage in memory consumption, and the CBM-DACs-bitmap and CBM-PforDelta versions a slight advantage in time.

Figure 15.2 shows the trade-off between disk space and the time needed to

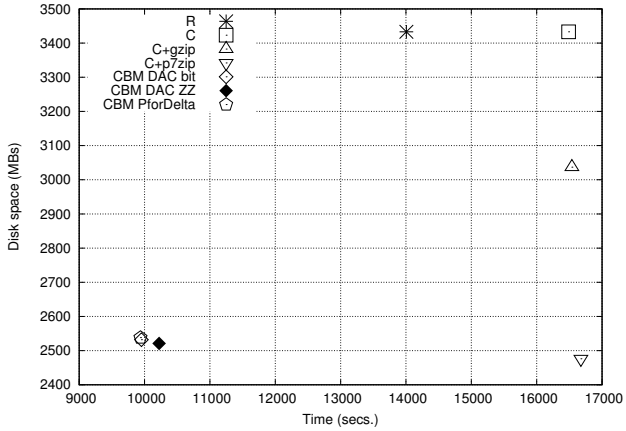


(a) Classical Algorithm

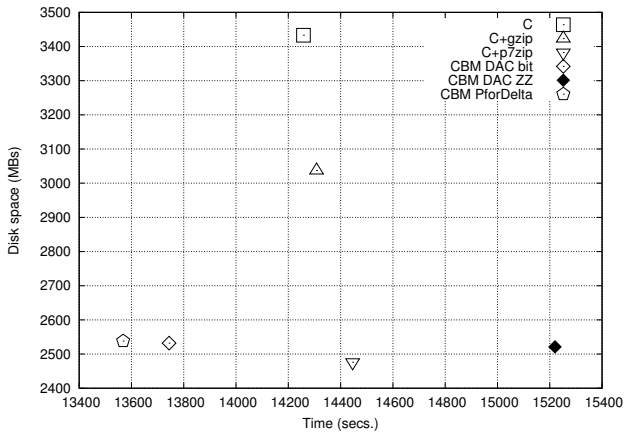


(b) Memory-Efficient Algorithm

Figure 15.1: Memory consumption/computation time trade-off for the dataset of size 30000×30000.



(a) Classical Algorithm



(b) Memory-Efficient Algorithm

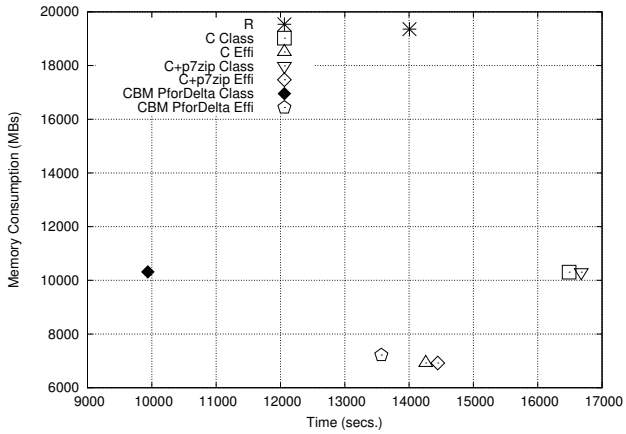
Figure 15.2: Disk space/computation time trade-off for the dataset of size 30000×30000.

compute the sample autocovariance function. With the classical algorithm, CBM is again the best alternative by far in both disk space and time. "C+p7zip" occupies a bit less space in disk, but requires much more time to compute the sample autocovariance function. When using the memory-efficient algorithm, CBM-PforDelta shows the best balance.

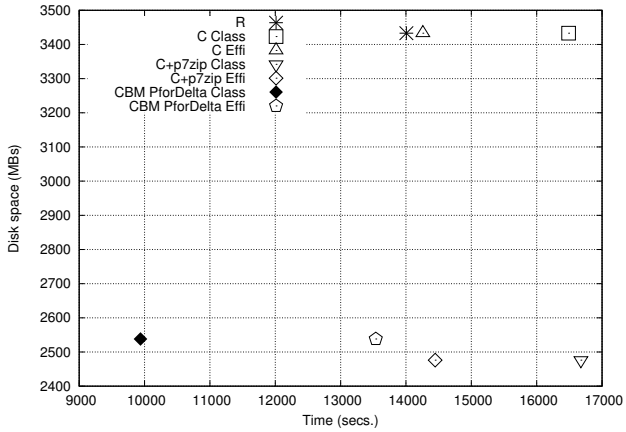
Figure 15.3 shows the final picture, by comparing R and the classical C implementation with our proposal: the use of CBM and a new memory-efficient algorithm to compute the sample autocovariance function. With respect to memory consumption, the efficient versions have the lowest memory consumption. The memory-efficient C implementation consumes 2.8 times less memory than R. In the case of the C program and CBM, the use of the memory-efficient algorithm implies a reduction of around 48% in the memory footprint with respect to the classical implementations.

In disk space, the best option is p7zip, but its fastest version (using the memory-efficient algorithm) is 45% slower than CBM-PforDelta with the classical algorithm, which is the version that shows a best balance in this plot.

Finally, the classical implementation of CBM-PforDelta is 40% faster than R and 65% than the classical C implementation.



(a) Memory consumption



(b) Disk space

Figure 15.3: Overall performance for the dataset of size 30000×30000.

Chapter 16

Discussion

16.1 Main contributions

We have shown that parallel processing is not the only way to improve the scalability of a large-scale analysis system. Even though it is an easy way, as there are many hardware and software tools available, we still have the opportunity to squeeze more by the use of efficient data structures and algorithms. This approach was common in the early times of computer science, where the hardware and software resources were much more limited. More specifically, we apply a compact data structures strategy to improve the scalability of the analysis of Brownian motion trajectories, although our solutions can be applied to any continuous time stochastic process. Our techniques do not prohibit the use of parallelization, on the contrary, they are even more suitable, since each node can process one trajectory independently from the others and data interchanges consume less bandwidth.

CBM is a strategy to represent Brownian trajectories in a compressed way using around 75% of the original space. The novelty is that this saving is also applicable to main memory space, since we can keep the data compressed all the time, decoding an isolated trajectory when needed and keeping the rest of the representation in a compressed form. Moreover, one isolated trajectory can be extracted from the compressed file in disk, loaded into main memory, and decompressed. In this way, the empirical autocovariance function is computed using up to 13 times less main memory space than when using the traditional method on plain data.

The new approach does not only save space in disk and main memory, but it also obtains reductions in running times. We use two strategies for this. First, the average value of all trajectories in all time instants, which is needed in the computation of the autocovariance function, is computed during the compression and stored in the compressed file. Second, the savings in main memory allow a better usage of the memory hierarchy. Therefore, our approach is up to 65% faster than

running a classical C program. Moreover, we have shown that our memory-efficient version of the algorithm is even faster than the classical setup, that is, storing the complete input dataset in main memory

In addition, thanks to the possibility of decompressing parts of CBM compressed data, we can apply the memory-efficient algorithm without a previous decompression. If we have to decompress the input dataset before the application of a C program, that process is up to 67% slower than computing the sample autocovariance function using directly CBM compressed data.

16.2 Future work

As future work, we plan to improve the compression ratio of the method. Another line of research would be to test the method with other continuous time stochastic processes (other than the Brownian motion) and measure the time and memory consumed, comparing the results with those included in this thesis. Finally, we could include other interesting statistics to be computed from the sample of trajectories.

Part IV

Summary of the thesis

Chapter 17

Conclusions and future work

17.1 Main contributions

In recent years, the amount of digital information stored and processed by computer systems has grown constantly. This is due to important advances in sensors and computing power, while Internet makes the data available to many persons or organizations. In addition, the *Big Data* phenomenon has extended analytical techniques such as data mining or OLAP systems, where huge amounts of data are exploited to obtain new data and knowledge. Therefore, designing new techniques to process and access huge amounts of information is a major challenge in many different areas.

There are several lines of research that try to improve the processing of large amounts of data, such as the design of new hardware or the parallel processing. A newly emerging research area is the compact representation of data. We will focus on this, also called *design of compact data structures*, which allows the execution of complex operations directly on compressed data. Compact data structures combine in a unique data structure a compressed representation of a dataset and the access methods that allow us to retrieve any given datum, without the need of decompressing the dataset from the beginning. The idea is to keep data always compressed, even in main memory. The benefits are obvious, in addition to the typical savings in disk space and bandwidth, we can process larger datasets, and moreover, this processing can be more efficient thanks to a better usage of the memory hierarchy. In many cases, compact data structures, in addition to a compressed representation, provide some sort of self-indexation, which allows us to answer queries even faster than performing that query over the plain representation and within the same compressed space.

In this thesis we are focused on the compact representation of large and complex datasets. Concretely, we are interested in three main areas; the representation

of multidimensional data, where the domain of each dimension is organized hierarchically, the management of spatial data, and the efficient execution of complex operations over scientific data.

This section summarizes the main contributions of this thesis:

1. We presented the k^n-treap, a straightforward extension of the k^2-treap to manage multiple dimensions.

2. We designed a new representation of multidimensional data. This structure can be seen as an extension of the k^n-treap which eliminates its strict regular partitioning of the hypercube, by allowing a new flexible division that depends on the domain hierarchies. It is particularly attractive to represent OLAP data cubes compactly and efficiently answer meaningful aggregate queries. This structure was initially designed to support aggregate sums, but it is easy to extend our results to other kinds of aggregations, such as the average or the maximum, or combine several of them. We also studied its applicability and its drawbacks in several scenarios, showing a superior time performance than a generic multidimensional structure.

3. We introduced the k^2-raster, which is a new compact data structure designed to store raster data. This spatial data model 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. We have also included two variants: the *hybrid* k^2-raster and the *heuristic* k^2-raster. The *hybrid* k^2-raster allows us to modify how the matrix is partitioned ateach level of tree. The other variant, the *heuristic* k^2-raster is an improved version that obtains better compression and even faster query times.

   Our experiments show that our methods improve previous approaches in all aspects, especially when the number of different values is large, which is critical when applying over real datasets.

4. We proposed a new framework for running a spatial join between a raster dataset and a vector dataset. It allows us to specify a range constraint on the values of the raster to refine queries. Our spatial join retrieves all elements of a vector dataset and the cells of the raster with a value in the queried range, which overlap each other.

   In our experiments, we show that our approach obtains important savings in both running time and memory consumption.

5. We developed a new compact representation of huge sets of functional data or trajectories of continuous time stochastic processes, called *Compact representation of Brownian Motion* (CBM). In this thesis, we focus on

trajectories of Brownian Motion, but our contribution is able to represent any continuous time stochastic process. CBM includes mechanisms to improve the computation time of empirical moments, for example, the empirical autocovariance function.

6. We presented a new memory-efficient algorithm to compute the sample autocovariance function, where each trajectory is only used once. The major benefit of this algorithm is that it does not keep the whole dataset in memory, thus reducing the memory consumption considerably. The application of this method is not limited to CBM and the trajectories of Brownian motion, it can also be used to computate the autocovariance of any type of trajectories of continuous time stochastic processes.

   Finally, we presented a set of experiments that showed that our approach obtains better space/time trade-offs than the state of the art.

17.2 Future work

In this section, we introduce some interesting future plans. We will present the most important ones for our contributions:

- The CMHD and the k^n-treap have been tested with several synthetic datasets. We plan to extend these experiments with real collections. We also plan to compare our contribution with an established OLAP database management system.

- In this thesis, the k^2-raster was designed for two-dimensional datasets. We plan to extend the structure for datasets with higher dimensionality, for example spatio-temporal datasets. In addition, we believe that our structure can be adapted to other type of environments, such as distributed environments or the semantic web.

  In the case of the semantic web, there is a standard, called GeoSPARQL, for running queries on spatial data. However, current tools for this standard have drawbacks, either they do not implement all the functionality or the query performance is very poor.

- In the framework for the spatial join, the choice of the R-tree to index the vector dataset was because it is the standard for this type of data. A new research line to improve our framework is to substitute the R-tree by modern compact data structures. Another interesting proposal is to increase the power of the framework by adding new operations.

- We plan to add new interesting statistics to expand CBM to include other types of computations, apart from autocovariance. In addition, more experiments with other continuous time stochastic processes can be tested.

Appendix A

Publications and other research results

Publications

Journals

- Nieves R. Brisaboa, Ana Cerdeira-Pena, Gonzalo Navarro, Miguel R. Penabad, and Fernando Silva-Coira Efficient Representation of Multidimensional Data over Hierarchical Domains. Manuscript to be submitted.
- Susana Ladra, José R. Paramá, and Fernando Silva-Coira: Scalable and Efficient Compressed Self-Index for Raster Data. Submitted to *Information Systems*.
- Nieves R. Brisaboa, Ricardo Cao, José R. Paramá, and Fernando Silva-Coira: Scalable processing and autocovariance computation of big functional data. Submitted to *Software Practice and Experience*.

International conferences

- Susana Ladra, José R. Paramá, and Fernando Silva-Coira: Compact and Queryable Representation of Raster Datasets. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 1–12, 2016.

  This paper has been cited by:

  - Alejandro Pinto, Diego Seco, and Gilberto Gutiérrez: Improved Queryable Representations of Rasters. In *Proceedings of Data Compression Conference (DCC)*, 2017

- Nieves R. Brisaboa, Ana Cerdeira-Pena, Narciso López López, Gonzalo Navarro, Miguel R. Penabad, and Fernando Silva-Coira: Efficient Representation of Multidimensional Data over Hierarchical Domains. In *Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 191–203, 2016.
- Susana Ladra, Juan R. López, José R. Paramá, and Fernando Silva-Coira: Efficient Spatial Join between Raster and Vector Datasets. Submitted to *ACM SIG SPATIAL*.

International research stays

- *March, 2017 - May, 2017.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

Appendix B

Resumen del trabajo realizado

En este capítulo se presenta un resumen del trabajo realizado durante la tesis. En la sección B.1 se presenta una breve introducción y la motivación para la realización de esta tesis. Además, se resume brevemente cada una de las áreas donde trabajamos, indicando cada uno de los principales problemas que tratamos de solucionar mediante nuestras contribuciones. En la sección B.2 se exponen y discuten cada una de las estructuras y algoritmos desarrollados. Finalmente, este capítulo se cierra con la sección B.3, donde se abordan diferentes líneas de investigación para mejorar y ampliar en un futuro nuestras contribuciones aquí expuestas.

B.1 Introducción

En los últimos años, con la aparición de un nuevo campo llamado *Big Data*, la demanda de análisis de cantidades enormes de información se ha incrementado considerablemente, como, por ejemplo, la minería de datos (*Data Mining*) o los sistemas OLAP para el análisis de datos sobre almacenes de datos (*Data Warehouses*). Uno de los principales motivos de que esto suceda es el incremento de la capacidad de almacenamiento de los discos y su bajo coste. Pero esto ha creado nuevos retos en el procesado de grandes colecciones de datos, ya que las estructuras de datos y los algoritmos convencionales no están adaptados para esa cantidad de información. Los conjuntos de datos de este campo poseen las llamadas $4V's$: volumen de datos demasiado grande (*Volumen*), velocidad de datos demasiado rápida (*Velocidad*), datos demasiado heterogéneos (*Variabilidad*) y datos demasiado inciertos (*Veracidad*). En esta tesis nos hemos centrado en la primera V: *volumen de datos demasiado grande.*

Actualmente existen varias líneas de investigación que tratan de mejorar el procesamiento de grandes volúmenes de datos, como el diseño de hardware específico para ese propósito o el paralelismo de aplicaciones, donde los datos son distribuidos en diferentes núcleos para ser procesados paralelamente. Por otro lado, existen otras líneas de investigación alternativas que se centran en el diseño de estructuras y algoritmos específicos. Entre ellas podemos destacar dos estrategias distintas pero con un mismo objetivo: la *gestión de datos en memoria* y las *estructuras de datos compactas*. El propósito de ambas es poder manipular un conjunto de datos en memoria principal y evitar los costosos accesos a disco. Mientras que la *gestión de datos en memoria* se centra más en las bases de datos tradicionales, el dominio de las *estructuras de datos compactas* es mucho más amplio, ya que cada una de ellas es diseñada y adaptada al entorno donde opera.

Las estructuras de datos compactas surgieron dentro del campo de la compresión de datos. En la mayoría de los casos, las técnicas de compresión de datos tienen como objetivo principal disminuir el espacio necesario para almacenar los datos y reducir el consumo de banda ancha durante la trasmisión de información. Sin embargo, las técnicas usadas tradicionalmente no permiten procesar o consultar los datos originales cuando están representados en forma comprimida, obligando a ejecutar previamente un proceso de descompresión. Este procesamiento adicional aumenta el tiempo de consulta considerablemente, especialmente en el ámbito del *Big Data*. Es ante este problema donde la nueva familia de estructuras, las denominadas estructuras de datos compactas, ha cambiado el uso clásico de la compresión de información. La representación compacta de datos es una línea de investigación que está ganando mucha importancia ya que permite el acceso y la ejecución de operaciones complejas directamente sobre los datos en forma comprimida, evitando la necesidad de descomprimir todo el contenido. Nuestra tesis se centra en esta línea de investigación, también llamada *diseño de estructuras compactas*. Estas estructuras, al reducir su tamaño, están concebidas para ser ejecutadas en memoria principal, aprovechando los niveles superiores de la jerarquía de memoria, que son mucho más rápidos que la memoria secundaria. Esto permite que a veces las operaciones procesadas sobre la estructura de datos compacta sean más rápidas incluso que en su representación en plano.

B.1.1 Motivación

La principal motivación de esta tesis es diseñar, implementar y evaluar experimentalmente estructuras de datos compactas que permitan representar de forma comprimida y autoindexada grandes volúmenes de datos. Se han estudiado tres escenarios diferentes donde es necesario el procesado y análisis eficiente de grandes cantidades de información:

- **Datos multidimensionales** o almacenes de datos (Data Warehouses). Existen muchas áreas donde los datos son representados en más de dos

dimensiones, por ejemplo, imágenes 3D o en sistemas de información geográfica (GIS). Nosotros consideramos el problema de la representación eficiente de datos multidimensionales donde el dominio de cada dimensión está jerarquizado.

Este es el caso de las bases de datos *Data Warehouse*, concretamente en sistemas de análisis OLAP (siglas en inglés de *procesamiento analítico en línea*). En este tipo de entornos es muy frecuente tener que manipular y analizar grandes colecciones de datos donde están involucradas varias dimensiones. En un sistema OLAP, los datos son conjuntos de tuplas que se consideran entradas en un cubo multidimensional, con una dimensión por atributo. Un caso típico en OLAP es que cada tupla contiene un resumen numérico (por ejemplo, la cantidad de ventas), que se considera como el valor de una celda en el cubo multidimensional. El dominio de cada dimensión es jerárquico, de manera que cada valor en la dimensión corresponde a una hoja en una jerarquía (por ejemplo, países, ciudades y tiendas en una dimensión y años, meses y días en otra). Además, es necesario responder operaciones complejas que requieren resúmenes (sumas, máximos, etc.) a diferentes niveles de cada dimensión (por ejemplo, ventas totales en Nueva York durante el mes anterior o cantidad de ordenadores vendidos en Europa hace 2 años), lo cual puede involucrar acceder a un gran número de celdas.

Esto nos presenta dos retos principales: debemos reducir el tamaño requerido para almacenar este tipo de colecciones y, al mismo tiempo, incluir algún tipo de índice que nos ayude a mejorar el tiempo de respuesta de las consultas. Hasta donde sabemos, no existen antecedentes de estructuras de datos compactas diseñadas específicamente para tratar este tipo de datos.

- **Datos espaciales**. Los sistemas de información geográficos pueden manipular información espacial usando distintos modelos de datos. En el nivel conceptual tenemos dos alternativas: *modelo basados en objetos* y *modelos basados en campos*. Un *modelo basado en objetos* considera que el espacio contiene entidades discretas e identificables, cada una con una posición geoespacial. Por otro lado, los *modelos basados en campos* se pueden ver como una función matemática que por cada posición del espacio devuelve un valor. Por ejemplo, un *modelo basado en objetos* contiene carreteras, edificios y cualquier otro tipo de objeto, mientras que un *modelo basado en campos* está generalmente más relacionado con propiedades físicas, como la elevación del terreno, la contaminación ambiental, la medición de temperaturas, etc. Si consideramos el nivel lógico, podemos encontrar otros dos modelos distintos: el *modelo vectorial* y el *modelo ráster*. El primero, el *model vectorial*, representa la información geográfica usando puntos y segmentos, mientras que un *modelo ráster* representa dicha información dividiendo el espacio, normalmente en cuadrículas de igual tamaño, donde cada celda tiene un valor. Aunque cualquier modelo conceptual puede ser representado mediante cualquier modelo lógico,

es común que los modelos vectoriales sean usados para representar modelos basado en objetos y los modelos ráster se asocian habitualmente a modelos basado en campos.

Nuestro objetivo es diseñar nuevas estructuras para la representación de información espacial, más concretamente, datos espaciales que usan un modelo ráster. Cualquier imagen dada puede ser vista como un ráster, por lo tanto, el uso de este modelo es masivo. Otros ejemplos de aplicación de conjuntos de datos ráster podrían ser el control de la contaminación, el pronóstico del tiempo, la captura de imágenes por satélite, la captación de sensores remotos, la modelización 3D, la ingeniería, etc. Todos estos escenarios tienen algo en común, tienen que tratar con grandes conjuntos de datos ráster. Aunque ya existen estructuras de datos compactas para este tipo de datos, su aplicabilidad sobre conjuntos de datos reales es limitada. El nuevo método propuesto se ha diseñado para evitar los problemas de las técnicas anteriores.

Además, otro reto dentro de este campo es la realización de consultas donde estén involucrados datos ráster y un datos vectoriales. Algunos trabajos anteriores proponen tipos de datos y operadores que pueden utilizar operandos de ambos tipos. Entre ellos, podemos destacar la operación de *join espacial* entre datos representados con un modelo ráster y otro vectorial. Esta operación devuelve los elementos de un conjunto de datos vectorial (polígonos, líneas o puntos) y las celdas del ráster que tienen un valor dentro de un rango definido por el usuario. Sin embargo, pocos trabajos se ocupan de las estructuras de datos y los algoritmos necesarios para implementar esa operación. Nosotros proponemos un framework que permite realizar joins espaciales entre un R-tree (que indexa los objetos con un modelo de datos vectorial) y una de nuestras contribuciones de esta tesis, el k^2-raster (para representar los datos ráster).

- **Trayectorias de procesos estocásticos de tiempo continuo**. Uno de los principales inconvenientes para los investigadores durante el procesado de datos científicos es que los tamaños de las colecciones son muy grandes y los métodos y estructuras convencionales no están adaptados para ellas. Por lo tanto, mientras que la comunidad científica sigue desarrollando nuevos métodos y técnicas para analizar datos, los científicos informáticos tienen que adaptar los algoritmos analíticos a conjuntos de información que poseen las llamadas $4V's$, es decir, los principales problemas del *Big Data*.

  En esta tesis hemos introducido un nuevo método para representar eficientemente trayectorias de procesos estocásticos de tiempo continuo, más concretamente, trayectorias de movimiento Browniano. Lo que proponemos es aplicar una estrategia de estructuras de datos compactas para poder acelerar los cálculos sobre dichas colecciones, como el cálculo de la función de covarianza.

B.2 Contribuciones y conclusiones

En esta tesis nos hemos centrado en la representación compacta y eficiente de conjuntos de datos grandes y complejos, como el propio título indica. Nosotros proponemos nuevas estructuras de datos compactas y nuevos algoritmos que pueden ser aplicados en diferentes dominios para resolver problemas relacionados con la manipulación de enormes cantidades de datos. Como ya hemos comentado en el apartado anterior, nos centramos principalmente en tres áreas: la representación de datos multidimensionales donde cada dimensión está organizada de forma jerárquica, el procesado de datos espaciales (principalmente datos ráster) y la representación compacta de datos científicos que nos permita realizar operaciones complejas eficientemente. En esta sección resumimos las contribuciones más importantes propuestas en la tesis:

1. Nuestra primera contribución es el diseño, implementación y evaluación experimental del k^n-treap, una extensión del k^2-treap para múltiples dimensiones. Además, hemos adaptado esta estructura para que pueda resolver consultas donde la entrada se compone de un conjunto de etiquetas, una por cada dimensión del hipercubo. Esto nos permite realizar consultas como las que se ejecutan en un sistema OLAP o sobre cualquier otra matriz definida por dimensiones jerárquicas. Para ello, antes de realizar la consulta sobre sus datos, el k^n-treap es capaz de traducir las etiquetas de las jerarquías de las dimensiones en una región del cubo definida por un conjunto de puntos. Esta estructura es principalmente usada como línea base para la experimentación con los datos multidimensionales.

   La descripción conceptual del k^n-treap y los resultados de los experimentos han sido publicados en las actas del 23th International Symposium on String Processing and Information Retrieval (SPIRE 2013) [BCPLL+16].

2. Nuestra segunda contribución es el diseño, implementación y evaluación experimental de una nueva estructura compacta, llamada *Compact representation of Multidimensional data on Hierarchical Domains* (CMHD), para la representación de datos multidimensionales, teniendo en cuenta el dominio jerárquico de cada dimensión. Se basa en el k^n-treap, pero elimina la restricción de que sus divisiones sean estáticas, es decir, en vez de partir una matriz en k^2 submatrices de igual tamaño, CMHD utiliza las jerarquías de cada dimensión para decidir cómo realiza dicha división en cada nivel. El uso de esta estructura es particularmente atractiva para representar datos de un sistema OLAP de forma compacta y responder de forma eficiente consultas sobre agregaciones de celdas. Aunque inicialmente esta estructura fue diseñada para soportar consultas de suma de valores de una agregación de celdas, es muy sencillo extender nuestros resultados a otro tipo de consultas, como por ejemplo obtener el valor medio o el valor máximo de una región, o incluso integrar varios tipos

de consultas en la misma estructura. También estudiamos su aplicabilidad y sus inconvenientes en diferentes escenarios.

La estructura propuesta es en general mucho más eficiente que una estructura multidimensional genérica, como el k^n-treap, ya que las consultas son resueltas agregando muchos menos nodos del árbol.

Tanto la descripción conceptual de CMHD como los resultados de los experimentos han sido publicados en las actas del 23th International Symposium on String Processing and Information Retrieval (SPIRE 2013) [BCPLL$^+$16].

3. Nuestra tercera contribución es el diseño, implementación y evaluación experimental de una nueva estructura que permite la representación compacta y eficiente de información espacial, más concretamente de datos ráster, llamada k^2-raster. Los datos ráster se presentan como una matriz de dos dimensiones donde cada celda contiene un valor positivo y representan atributos del espacio (temperatura, presión, elevaciones del terreno, etc.). Nuestra propuesta tiene dos variantes: el *hybrid* k^2-raster y el *heuristic* k^2-raster. El *hybrid* k^2-raster (en español, k^2-raster híbrido) nos permite modificar cómo es particionada la matriz durante los primeros niveles del árbol y durante los últimos niveles del árbol, utilizando dos parámetros (k_1 y k_2) distintos para cada uno de ellos. La otra variante, el *heuristic* k^2-raster (k^2_H-raster, en español k^2-raster heurístico), es una mejora de la versión anterior y usa una técnica de vocabulario para codificar las submatrices del último nivel. Con la ayuda de una función heurística basada en la entropía, determina si la partición es incluida en el vocabulario o, contrariamente, se codifica usando un compresor de enteros llamado DACs. El objetivo es que el vocabulario esté compuesto por las submatrices más frecuentes, cuya frecuencia de aparición compense el coste de la representación de su entrada en el vocabulario de submatrices.

   Nuestro experimentos demuestran que nuestros métodos mejoran el estado del arte actual en todos los aspectos, especialmente cuando el número de valores diferentes contenidos en el ráster es grande, lo cual es crítico cuando se aplica sobre datos de un entorno real.

   Una versión preliminar de este trabajo fue publicada en las actas del 28th International Conference on Scientific and Statistical Database Management (SSDBM 2016) [LPSC16].

4. Nuestra cuarta contribución es el diseño, implementación y evaluación experimental de un framework que soporta la ejecución de la operación *join espacial* entre un conjunto de datos ráster (en nuestro caso utilizamos el k^2-raster mencionado en el punto anterior) y un conjunto de datos vectoriales (indexado mediante un R-tree). Además, las consultas se pueden acotar para un rango de valores concretos. Nuestra operación de join espacial devuelve

todos los elementos del conjunto de datos vectorial y las celdas del k^2-raster que solapan con esos elementos, y donde las celdas del ráster contienen valores dentro de un rango especificado en la consulta.

En nuestros experimentos mostramos que nuestra solución obtiene grandes mejoras, reduciendo tanto el tiempo de ejecución de la operación como la memoria consumida durante el proceso.

5. Nuestra quinta contribución es el diseño, implementación y evaluación experimental de una nueva estructura compacta, llamada *Compact representation of Brownian Motion* (CBM), para la representación de enormes conjuntos de datos compuestos de trayectorias correspondientes al movimiento Browniano, aunque puede ser usada para cualquier tipo de trayectorias de procesos estocásticos de tiempo continuo. Su principal objetivo, además de reducir el tamaño que ocupan los datos, es mejorar el tiempo de cálculo de las operaciones realizadas sobre ellos. Nosotros nos hemos centrado en el cálculo de la autocovarianza. Para mejorar el tiempo de cálculo de operaciones complejas, durante el proceso de codificación se calcula información adicional sobre los datos, como el valor medio de cada instante de tiempo, que será almacenada junto a los datos comprimidos. Para permitir la descompresión parcial de los datos, se codifica cada una de las trayectorias de forma separada.

   Los experimentos demuestran que las trayectorias de movimiento Browniano son muy difíciles de comprimir, ya que por su naturaleza siguen una progresión aleatoria, por lo que las técnicas de compresión no pueden explotar la repetición de valores. Aun así, nuestros resultados muestran que nuestra nueva estructura puede reducir tanto el tamaño del conjunto de datos como el tiempo de cálculo de la covarianza, que es el objetivo principal.

6. Nuestra sexta contribución, diseñada durante el desarrollo de la estructura anterior, es un nuevo algoritmo para el cálculo de la autocovarianza, donde no es necesario cargar todo el conjunto de datos al completo, sino que solo es necesario mantener en memoria una única trayectoria a la vez. Este algoritmo no es exclusivo de nuestra estructura ni tampoco de las trayectorias de movimiento Browniano, sino que puede ser usada para calcular la autocovarianza de cualquier otro tipo de conjunto.

   Hemos comparado nuestra nueva técnica con el algoritmo clásico de cálculo de la covarianza y medido el consumo de tiempo y espacio para ambos casos. Los resultados demuestran que nuestra técnica es mucho más eficiente en el consumo de memoria, e incluso consigue reducir el tiempo de cálculo cuando los conjuntos de datos son más grandes.

B.3 Trabajo futuro

En esta sección proponemos varias consideraciones que pueden ser interesantes para un futuro de cara a mejorar la aplicabilidad y rendimiento de nuestras contribuciones. Entre ellas podemos destacar las siguientes líneas de investigación:

- Tanto el CMHD como el k^n-treap han sido probados con varios conjuntos de datos, variando sus tamaños y el número de dimensiones. Para completar el análisis del comportamiento de ambas estructuras, planeamos extender estos experimentos con nuevos conjuntos de datos aún más grandes, además de añadir nuevos datos reales para no solo probar con datos generados sintéticamente. También creemos que las pruebas se pueden ampliar comparando nuestra estructura con un sistema OLAP y mostrar el rendimiento de ambos para un mismo conjunto de datos. Con estas consideraciones, los experimentos se enriquecerían al cubrir un amplio abanico de casos distintos.

- En esta tesis, el k^2-raster fue diseñado para representar matrices de datos en dos dimensiones. Nosotros proponemos extender nuestra estructura para conjuntos de datos con mayor dimensión, por ejemplo, para ser usado con información espacio-temporal, donde es necesario al menos 3 dimensiones. Además, creemos que nuestra estructura puede ser adaptada para operar sobre otros tipos de dominios, como pueden ser los sistemas distribuidos. Otra consideración que se puede tener en cuenta es la aplicación del k^2-raster en la web semántica para realizar consultas espaciales, por ejemplo, siguiendo el estándar GeoSPARQL. Las herramientas actuales para este tipo de consultas presentan varios inconvenientes, debido a que o bien no implementan toda la funcionalidad o bien el rendimiento de las consultas es muy pobre. Creemos que nuestra estructura podría mejorar ambos problemas.

- La elección del R-tree en nuestro framework de consultas espaciales entre datos de tipo ráster y datos vectoriales se debe a que es el estándar para este tipo de datos. Una nueva línea de investigación para mejorar nuestro framework es añadir estructuras de datos más modernas además del R-tree estándar. Otra propuesta interesante es aumentar la potencia del framework añadiendo nuevas operaciones entre el k^2-raster y un conjunto de datos vectoriales.

- Inicialmente el CBM está diseñado para reducir el tiempo de cálculo de la autocovarianza. Un nuevo plan para esta estructura es ampliarla con nuevos tipos de cálculos aparte de la autocovarianza. Además, sería interesante realizar experimentos con otros tipos de procesos estocásticos de tiempo continuo (otros además del movimiento Browniano). En relación a la compresión obtenida por la técnica, se podría probar la estructura con otros compresores de enteros para codificar las diferencias de cada trayectoria.

Bibliography

[ABG+02] S. Azhar, G. J. Badros, A. Glodjo, M. Y. Kao, J. H. Reif, and H. Reif. Data compression techniques for stock market prediction. In *Proc. of the Data Compression Conference (DCC)*, pages 72–82, 2002.

[ÁGBF+15] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, M. A. Martínez-Prieto, and G. Navarro. Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems*, 44(2):439–474, 2015.

[ÁGBLP10] S. Álvarez-García, N. R. Brisaboa, S. Ladra, and O. Pedreira. A compact representation of graph databases. In *Proc. of the Eighth Workshop on Mining and Learning with Graphs (MLG)*, pages 18–25, 2010.

[BCPLL+16] N. R. Brisaboa, A. Cerdeira-Pena, N. López-López, G. Navarro, M. R. Penabad, and F. Silva-Coira. Efficient Representation of Multidimensional Data over Hierarchical Domains. In *Proc. of the Symposium on String Processing and Information Retrieval (SPIRE)*, volume 9954 LNCS, pages 191–203, 2016.

[BdBG+17] N. R. Brisaboa, G. de Bernardo, G. Gutiérrez, M. R. Luaces, and J. R. Paramá. Efficiently querying vector and raster data. *The Computer Journal*, 2017.

[BdBK+16] N. R. Brisaboa, G. de Bernardo, R. Konow, G. Navarro, and D. Seco. Aggregated 2D Range Queries on Clustered Points. *Information Systems*, pages 34–49, 2016.

[BDF+98] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *Proc. of the ACM SIGMOD Record*, volume 27 of *SIGMOD '98*, pages 575–577, jun 1998.

[BFLN08] N. R. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Reorganizing compressed text. In *Proc. of the International ACM SIGIR Conference on Research and Development in Information Retrieval (SIGIR)*, pages 139–146, 2008.

[BFNP07] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight Natural Language Text Compression. *Information Retrieval*, 10(1):1–33, 2007.

[BGBNP16] N. R. Brisaboa, A. Gómez-Brandón, G. Navarro, and J. R. Paramá. GraCT: A Grammar based Compressed representation of Trajectories. In *Proc. of the 23rd Int. Symp. on String Processing and Information Retrieval (SPIRE 2016) - LNCS 9954*, pages 218–230, 2016.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. of the 1990 ACM SIGMOD international conference on Management of data (SIGMOD)*, volume 19, pages 322–331, 1990.

[BLN13] N. R. Brisaboa, S. Ladra, and G. Navarro. DACs: Bringing direct access to variable-length codes. *Information Processing & Management*, 49(1):392–404, jan 2013.

[BLN14] N. R Brisaboa, S. Ladra, and G. Navarro. Compact representation of Web graphs with extended functionality. *Information Systems*, 39(1):152–174, 2014.

[BR09] M. Burtscher and P. Ratanaworabhan. FPC: A High-Speed Compressor for Double-Precision Floating-Point Data. *IEEE Transactions on Computers*, 58(1):18–31, jan 2009.

[BR10] M. Burtscher and P. Ratanaworabhan. gFPC: A Self-Tuning Compression Algorithm. In *Proc. of the Data Compression Conference (DCC)*, pages 396–405, 2010.

[Bro10] P. G. Brown. Overview of sciDB: large scale array storage, processing and analysis. In *Proc. of the 2010 international conference on Management of data (SIGMOD)*, page 963, 2010.

[BS73] F. Black and M. Scholes. The Pricing of Options and Corporate Liabilities. *The Journal of Political Economy*, 81(3):637–654, 1973.

[CC03] Y. K. Chan and C. C. Chang. Block image retrieval based on a compressed linear quadtree. In *Proc. of the 4th International Conference on Information, Communications and Signal Processing and 4th Pacific-Rim Conference on Multimedia (ICICS-PCM)*, volume 1, pages 31–35, 2003.

[CCS93] E. F. Codd, S. B. Codd, and C. T. Salley. Providing OLAP (On-Line Analytical Processing) to User-Analysts: An IT Mandate. E. F. Codd and Associates, 1993.

[CD97] S. Chaudhuri and U. Dayal. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[CDL+12] T. Chan, S. Durocher, K. Larsen, J. Morrison, and B. Wilkinson. Linear-Space Data Structures for Range Mode Query in Arrays. In *Proc. of the 29th International Symposium on Theoretical Aspects of Computer Science (STACS)*, pages 290–301, 2012.

[Cla96] D. Clark. *Compact {PAT} Trees.* PhD thesis, University of Waterloo, Canada, 1996.

[CLY06] K.-L. Chung, Y.-W. Liu, and W.-M. Yan. A hybrid gray image representation using spatial- and DCT-based approach with application to moment computation. *J. Vis. Commun. Image Represent.*, 17:1209–1226, dec 2006.

[Cou92] H. Couclelis. People manipulate objects (but cultivate fields): Beyond the raster-vector debate in GIS. In *Theories and Methods of Spatiotemporal Reasoning in Geographic Space*, volume 639, pages 65–77, 1992.

[CTVM08] A. Corral, M. Torres, M. Vassilakopoulos, and Y. Manolopoulos. Predictive join processing between regions and moving objects. In *Proc. of the 12th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 46–61, Pori, Finland, 5-7 September 2008. Springer-Verlag Berlin, Heidelberg, Germany.

[CVM99] A. Corral, M. Vassilakopoulos, and Y. Manolopoulos. Algorithms for joining r-trees and linear region quadtrees. In *Proc. of Advances in Spatial Databases (SSD)*, pages 251–269, Hong Kong, China, 20-23 July 1999. Springer-Verlag Berlin, Heidelberg, Germany.

[dBÁGB+13] G. de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, and O. Pedreira. Compact Querieable Representations of Raster Data. In *Proc. of the 20th Int. Symp. on String Processing and Information Retrieval (SPIRE 2013) - LNCS 8214*, pages 96–108. 2013.

[DG08] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communication of ACM*, 51(1):107–113, 2008.

[Duv09] B. Duvenhage. Using an implicit min/max KD-tree for doing efficient terrain line of sight calculations. In *Proc. of the International*

Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa (AFRIGRAPH), volume 1, page 81, 2009.

[DXS+15] S. Dudoladov, C. Xu, S. Schelter, A. Katsifodimos, S. Ewen, K. Tzoumas, and V. Markl. Optimistic Recovery for Iterative Dataflows in Action. In *Proc. of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 1439–1443, 2015.

[EFF00] V. Engelson, D. Fritzson, and P. Fritzson. Lossless Compression of High-Volume Numerical Data from Simulations. In *Proc. of the Data Compression Conference (DCC)*, page 574, 2000.

[Ein06] A. Einstein. On the theory of the Brownian movement. *Annalen der Physik*, 19(4):371–381, 1906.

[Eli75] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, 21(2):194–203, 1975.

[FM12] N. Fout and K. L. Ma. An adaptive prediction-based approach to lossless compression of floating-point volume data. *IEEE Transactions on Visualization and Computer Graphics*, 18(12):2295–2304, 2012.

[FOP14] A. Fariña, A. Ordóñez, and J. R. Paramá. Indexing and Self-indexing sequences of IEEE 754 double precision numbers. *Information Processing & Management*, 50(6):857–875, nov 2014.

[FV06] F. Ferraty and P. Vieu. *NonParametric Functional Data Analysis: Theory and Practice*. Springer, 2006.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, dec 1982.

[GG98] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, jun 1998.

[GGMN05] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical Implementation of Rank and Select Queries. In *Proc. of the 4th Workshop on Efficient and Experimental Algorithms (WEA)*, volume 0109, pages 27–38, 2005.

[GGV03] R. Grossi, A. Gupta, and J. S. Vitter. High-Order Entropy-Compressed Text Indexes. In *Proc. of the 14th annual ACM-SIAM symposium on Discrete algorithms (SODA)*, volume 2068 of *SODA '03*, pages 841–850, 2003.

[Gol66] S. W. Golomb. Run-length encodings. *IEEE Trans. Inform. Theory*, IT-12:399–401, 1966.

[GRA16] Grass gis manual:v.rast.stats. https://grass.osgeo.org/grass72/manuals/v.rast.stats.html, 2016. 6 dec 2016.

[GRS00] S. Grumbach, P. Rigaux, and L. Segoufin. Manipulating interpolated data is easier than you thought. In *Proc. of the 26th International Conference on Very Large Data Bases (VLDB)*, VLDB 2000, pages 156–165, 2000.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD)*, volume 14, page 47, 1984.

[GWSV06] A. Gupta, Wing-Kai H., R. Shah, and S. Vitter. Compressed Data Structures: Dictionaries and Data-Aware Measures. In *Proc. of the Data Compression Conference (DCC)*, pages 213–222, 2006.

[HCP+05] R. J. Hijmans, S. E. Cameron, J. L. Parra, P. G. Jones, and A. Jarvis. Very high resolution interpolated climate surfaces for global land areas. *International Journal of Climatology*, 25(15):1965–1978, 2005.

[HK12] L. Horvath and P. Kokoszka. *Inference for Functional Data with Applications*. Springer, 2012.

[HSTV14] W.-K. Hon, R. Shah, S. V. Thankachan, and J. S. Vitter. Space-Efficient Frameworks for Top-k String Retrieval. *Journal of the ACM*, 61(2):9:1—-9:36, 2014.

[Huf52] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. In *Proc. of the I.R.E. (Institue of Radio Engineers Inc.)*, volume 40, pages 1098–1101, 1952.

[Hul09] J. Hulll. *Options, Futures, and other Derivatives*. Pearson Prentice Hall, 2009.

[HV14] P. G. Howard and J. S. Vitter. Fast and efficient lossless image compression. In *Proc. of the Data Compression Conference (DCC)*, pages 351–360, 2014.

[Inm93] W. H. Inmon. *Building the data warehouse*. Wiley, 1993.

[ISO03] ISO. *Geographic information – Spatial schema*. International Organization for Standardization (TC 211), 2003.

[ISO05] ISO. *Geographic information – Schema for coverage geometry and functions*. International Organization for Standardization (TC 211), 2005.

[Jac89a] G. Jacobson. Space-efficient static trees and graphs. In *Proc. of the 30th Annual Symposium on Foundations of Computer Science (FOCS)*, SFCS '89, pages 549–554, 1989.

[Jac89b] G. Jacobson. *Succinct Static Data Structures*. PhD thesis, Carnegie-Mellon, 1989.

[Kai91] W. Kainz. A Review of: "The Design and Analysis of Spatial Data Structures" by h. samet. *International Journal of Geographical Information Systems*, 5(2):253–253, jan 1991.

[KD76] A. Klinger and C. R. Dyer. Experiments on picture representation using regular decomposition. *Computer Graphics and Image Processing*, 5(1):68–105, mar 1976.

[KEW13] M. Kane, J. Emerson, and S. Weston. Scalable Strategies for Computing with Massive Data. *Journal of Statistical Software*, 55(1):1–19, 2013.

[Kli71] A. Klinger. *Pattern and search statistics*. Academic Press, 1971.

[KR02] R. Kimball and M. Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., 2nd edition, 2002.

[KS16] S. T. Klein and D. Shapira. Random access to Fibonacci encoded files. *Discrete Applied Mathematics*, 212:115–128, 2016.

[Kül14] M. Oguzhan O. Külekci. Enhanced variable-length codes: Improved compression with efficient random access. *Data Compression Conference Proceedings*, pages 362–371, 2014.

[LB07] Y. Li and T. R. Bretschneider. Semantic-Sensitive Satellite Image Retrieval. *IEEE Transactions on Geoscience and Remote Sensing*, 45(4):853–860, 2007.

[LGMR05] P. A. Longley, M. F. Goodchild, D. J. Maguire, and D. W. Rhind. *Geographic information science and systems*. Wiley, 2005.

[LGPBJ+08] F. López-Granados, J. M. Peña-Barragán, M. Jurado-Expósito, M. Francisco-Fernández, R. Cao, A. Alonso-Betanzos, and O. Fontenla-Romero. Multispectral classification of grass weeds and wheat (Triticum durum) crop using linear and nonparametric functional

discriminant analysis, and neural networks. *Weed Research*, 48:28–37, 2008.

[LI06] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, sep 2006.

[Lin97] T.-W. Lin. Compressed quadtree representations for storing similar images. *Image and Vision Computing*, 15(11):833–843, nov 1997.

[LL03] M. Levene and G. Loizou. Why is the Snowflake Schema a Good Data Warehouse Design? *Information Systems*, 28(3):225–240, 2003.

[LPSC16] S. Ladra, J. R. Paramá, and F. Silva-Coira. Compact and queryable representation of raster datasets. In *Proc. of the 28th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 1–12, 2016.

[LR04] T. Liebchen and Y. A. Reznik. MPEG-4 ALS: an emerging standard for lossless audio coding. In *Proc. of the Data Compression Conference (DCC)*, pages 439–448, 2004.

[LvW13] K. G. Larsen and F. van Walderveen. Near-Optimal Range Reporting Structures for Categorical Data. In *Proc. of the 24th Symposium on Discrete Algorithms (SODA)*, pages 265–276, 2013.

[MHP+11] J. Muckell, J.-H. Hwang, V. Patil, C. T. Lawson, F. Ping, and S. S. Ravi. SQUISH: An Online Approach for GPS Trajectory Compression. In *Proc. of the International Conference on Computing for Geospatial Research & Applications (COM.Geo)*, pages 13:1—-13:8, 2011.

[MN07] V. Mäkinen and G. Navarro. Rank and select revisited and extended. *Theoretical Computer Science*, 387(3):332–347, 2007.

[MPFL96] J. L. Mitchell, W. B. Pennebaker, C. E. Frogg, and D. J. Legall. *MPEG Video Compression Standard*. Chapman and Hall, 1996.

[Mun96] J. I. Munro. Tables. In *Proc. of the Conference Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[Nav16] G. Navarro. *Compact Data Structures – A practical approach*. Cambridge University Press, 2016.

[NST99] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. pages 171–189. Springer, Berlin, Heidelberg, 1999.

[OGC10] OGC. *OpenGIS Web Feature Service 2.0 Interface Standard*. Open Geospatial Consortium, Inc., 2010.

[OGC12] OGC. *OpenGIS Web Coverage Service 2.0 Interface Standard - Core: Corrigendum*. Open Geospatial Consortium, Inc., 2012.

[OS07] D. Okanohara and K. Sadakane. Practical Entropy-Compressed Rank/Select Dictionary. In *Proc. of the Workshop on Algorithm Engineering and Experiments, (ALENEX)*, 2007.

[OW83] M. A. Oliver and N. E. Wiseman. Operations on Quadtree Encoded Images. *The Computer Journal*, 26(1):83–91, jan 1983.

[Pag99] R. Pagh. Low Redundancy in Static Dictionaries with O(1) Worst Case Lookup Time. In *Proc. of the International Colloquium on Automata, Languages and Programming (ICALP)*, volume LNCS 1644, pages 595–604, 1999.

[Pla13] H. Plattner. *A Course in In-Memory Data Management: The Inner Mechanics of In-Memory Databases*. Springer, 2013.

[PZ12] H. Plattner and A. Zeier. *In-Memory Data Management: Technology and Applications*. Springer, 2012.

[QG13] M. Quartulli and I. G. Olaizola. A review of EO image information mining, 2013.

[RBR12] M. Romero, N. R. Brisaboa, and M. A. Rodríguez. The SMO-index: a succinct moving object structure for timestamp and interval queries. In *Proc. of the 20th International Conference on Advances in Geographic Information Systems (SIGSPATIAL)*, page 498, 2012.

[Ric79] R. F. Rice. Some practical universal noiseless coding techniques. Technical Report 79-22, Jet Propulsion Laboratory, 1979.

[RKB06] P. Ratanaworabhan, J. Ke, and M. Burtscher. Fast Lossless Compression of Scientific Floating-Point Data. In *Proc. of the Data Compression Conference (DCC)*, pages 133–142, 2006.

[RRR02] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. of the Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[RS05] J. O. Ramsay and B. W. Silverman. *Functional Data Analysis*. Springer, 2005.

[RSV02] P. Rigaux, M. O. Scholl, and A. Voisard. *Spatial Databases: With Application to GIS*. 2002.

[Sad03] K. Sadakane. New Text Indexing Functionalities of the Compressed Suffix Arrays. *ALGORITHMS: Journal of Algorithms*, 48, 2003.

[Sad07] K. Sadakane. Succinct data structures for flexible text retrieval systems. *Journal of Discrete Algorithms*, 5:12–22, 2007.

[Sam84] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(2):187–260, jun 1984.

[Sam85] H. Samet. Data structures for quadtree approximation and compression. *Communications of the ACM*, 28(9):973–993, 1985.

[Sam06] H. Samet. Foundations of Multidimensional and Metric Data Structures. *Order A Journal On The Theory Of Ordered Sets And Its Applications*, di(August):0–1, 2006.

[SETM13] S. Schelter, S. Ewen, K. Tzoumas, and V. Markl. "All Roads Lead to Rome": Optimistic Recovery for Distributed Iterative Data Processing. In *Proc. of the ACM International Conference on Information & Knowledge Management (CIKM)*, pages 1919–1928, 2013.

[SH91] P. Svensson and Z. Huang. Geo-SAL: A query language for spatial data analysis. In *Proc. of the Advances in Spatial Databases (SSD)*, volume 525, pages 119–140, 1991.

[Sha48] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:370–423,623–656, 1948.

[SJS+12] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C.S. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R.t Ross, and N. F. Samatova. ISOBAR Preconditioner for Effective and High-throughput Lossless Data Compression. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, pages 138–149, apr 2012.

[TB79] C. D. Tomlin and J. K. Berry. A Mathematical Structure for Cartographic Modeling in Environmental Analysis. In *Proc. of the 39th Symposium of the American Conference on Surveying and Mapping (ACSM)*, pages 269–283, 1979.

[Teu11] J. Teuhola. Interpolative coding of integer sequences supporting log-time random access. *Information Processing Management*, 47(5):742–761, 2011.

[Tom90] C. D. Tomlin. *Geographic information systems and cartographic modeling*. Prentice Hall, 1990.

[Tom94] C. D. Tomlin. Map algebra: one perspective. *Landscape and Urban Planning*, 30(1-2):3–12, 1994.

[TP01] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. pages 431–440, 2001.

[VZ09] A. Vaisman and E. Zimányi. A multidimensional model representing continuous fields in spatial data warehouses. In *Proc. of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems (GIS)*, SIGSPATIAL 2009, page 168, 2009.

[Wal91] G. K. Wallace. The JPEG still picture compression standard. *Communications of the ACM*, 34(4):30–44, apr 1991.

[WD04] M. Worboys and M. Duckham. *GIS : a computing perspective*. CRC Press, 2004.

[Woo84] J. R. Woodwark. Compressed Quad Trees. *The Computer Journal*, 27(3):225–229, 1984.

[WZ99] H. E. Williams and J. Zobel. Compressing Integers for Fast File Access. *The Computer Journal*, 42(3):193–201, mar 1999.

[YS08] Y. You and M. Y. Sung. Haptic data transmission based on the prediction and compression. *IEEE International Conference on Communications*, pages 1824–1828, 2008.

[ZHNB06] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In *Proc. of the International Conference on Data Engineering, (ICDE)*, pages 59–71, 2006.

[Zon16] Zonal statistics help|arcgis for desktop. http://webhelp.esri.com/arcgisdesktop/9.3/index.cfm?TopicName=Zonal_Statistics, 2016. 1 dec 2016.

[ZY10] J. Zhang and S. You. Supporting Web-Based Visual Exploration of Large-Scale Raster Geospatial Data Using Binned Min-Max Quadtree. In *Proc. of the International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 379–396, 2010.

[ZY13] J. Zhang and S. You. High-performance quadtree constructions on large-scale geospatial rasters using GPGPU parallel primitives. *International Journal of Geographical Information Science*, 27(11):2207–2226, 2013.

[ZYG10] J. Zhang, S. You, and L. Gruenwald. Indexing Large-Scale Raster Geospatial Data Using Massively Parallel GPGPU Computing. *Parallel Computing*, (c):0–3, 2010.

[ZYG15] J. Zhang, S. You, and L. Gruenwald. Quadtree-based lightweight data compression for large-scale geospatial rasters on multi-core CPUs. In *Proc. of the IEEE International Conference on Big Data (Big Data)*, pages 478–484, oct 2015.