New data structures and algorithms for the efficient management of large spatial datasets

Autor: Guillermo de Bernardo Roca

Tesis Doctoral

Directores:
Nieves Rodríguez Brisaboa
Gonzalo Navarro Badino

Departamento de Computación

A Coruña, 2014

PhD thesis supervised by
Tesis doctoral dirigida por

Nieves Rodríguez Brisaboa
Departamento de Computación
Facultad de Informática
Universidade da Coruña
15071 A Coruña (España)
Tel: +34 981 167000 ext. 1243
Fax: +34 981 167160
brisaboa@udc.es

Gonzalo Navarro Badino
Departamento de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120 Santiago (Chile)
Tel: +56 2 6892736
Fax: +56 2 6895531
gnavarro@dcc.uchile.cl

Nieves Rodríguez Brisaboa y Gonzalo Navarro Badino, 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 Guillermo de Bernardo Roca cuya firma también se incluye.

Acknowledgements

First I would like to thank my advisors, Nieves and Gonzalo, for their support throughout this work, the knowledge they transmitted to me, and also their advice and dedication from the first ideas to the final review of this thesis. I also want to thank all the members of the Databases Laboratory, my partners and friends, who made every day of work easier with their support. Thanks also to all the people who collaborated with me in different research lines, and all those that I met during research stays for their support and hospitality.

Finally, my biggest gratitude to my friends and family, who encouraged me and helped me in the difficult moments. Thanks to all who were closer to me in those moments, and especially to María, who always believed in me and without whom all of this would not have been possible.

Agradecimientos

En primer lugar me gustaría agradecer a mis directores, Nieves y Gonzalo, su apoyo a lo largo de este trabajo, por el conocimiento que me han transmitido, y también por sus consejos y dedicación, desde las primeras ideas hasta la revisión final de la tesis. También quiero agradecer a todos los miembros del Laboratorio de Bases de Datos, compañeros y amigos, quienes hicieron cada día de trabajo más fácil con su apoyo. Gracias también a todos los que colaboraron conmigo en las diferentes líneas de investigación y a todos a quienes conocí durante mis estancias de investigación, por su apoyo y hospitalidad.

Finalmente, mi mayor gratitud para mis amigos y familia, quienes me animaron y ayudaron en los momentos difíciles. Gracias a todos los que estuvieron más cerca de mi en esos momentos, y especialmente a María, que siempre creyó en mí y sin la que todo esto no habría sido posible.

Abstract

In this thesis we study the efficient representation of multidimensional grids, presenting new compact data structures to store and query grids in different application domains. We propose several static and dynamic data structures for the representation of binary grids and grids of integers, and study applications to the representation of raster data in Geographic Information Systems, RDF databases, etc.

We first propose a collection of static data structures for the representation of binary grids and grids of integers: *1)* a new representation of bi-dimensional binary grids with large clusters of uniform values, with applications to the representation of binary raster data; *2)* a new data structure to represent multidimensional binary grids; *3)* a new data structure to represent grids of integers with support for top-k range queries. We also propose a new dynamic representation of binary grids, a new data structure that provides the same functionalities that our static representations of binary grids but also supports changes in the grid.

Our data structures can be used in several application domains. We propose specific variants and combinations of our generic proposals to represent temporal graphs, RDF databases, OLAP databases, binary or general raster data, and temporal raster data. We also propose a new algorithm to jointly query a raster dataset (stored using our representations) and a vectorial dataset stored in a classic data structure, showing that our proposal can be faster and require less space than the usual alternatives. Our representations provide interesting trade-offs and are competitive in terms of space and query times with usual representations in the different domains.

Resumen

En esta tesis estudiamos la representación eficiente de matrices multidimensionales, presentando nuevas estructuras de datos compactas para almacenar y procesar *grids* en distintos ámbitos de aplicación. Proponemos varias estructuras de datos estáticas y dinámicas para la representación de matrices binarias o de enteros y estudiamos aplicaciones a la representación de datos raster en Sistemas de Información Geográfica, bases de datos RDF, etc.

En primer lugar proponemos una colección de estructuras de datos estáticas para la representación de matrices binarias y de enteros: *1)* una nueva representación de matrices binarias con grandes grupos de valores uniformes, con aplicaciones a la representación de datos raster binarios; *2)* una nueva estructura de datos para representar matrices multidimensionales; *3)* una nueva estructura de datos para representar matrices de enteros con soporte para consultas top-k de rango. También proponemos una nueva representación dinámica de matrices binarias, una nueva estructura de datos que proporciona las mismas funcionalidades que nuestras propuestas estáticas pero también soporta cambios en la matriz.

Nuestras estructuras de datos pueden utilizarse en distintos dominios. Proponemos variantes específicas y combinaciones de nuestras propuestas para representar grafos temporales, bases de datos RDF, datos raster binarios o generales y datos raster temporales. También proponemos un nuevo algoritmo para consultar conjuntamente un conjuto de datos raster (almacenado usando nuestras propuestas) y un conjunto de datos vectorial almacenado en una estructura de datos clásica, mostrando que nuestra propuesta puede ser más rápida y usar menos espacio que otras alternativas. Nuestras representaciones proporcionan interesantes *trade-offs* y son competitivas en espacio y tiempos de consulta con representaciones habituales en los diferentes dominios.

Resumo

Nesta tese estudiamos a representación eficiente de matrices multidimensionais, presentando novas estruturas de datos compactas para almacenar e procesar *grids* en distintos ámbitos de aplicación. Propoñemos varias estruturas de datos estáticas e dinámicas para a representación de matrices binarias ou de enteiros e estudiamos aplicacións á representación de datos raster en Sistemas de Información Xeográfica, bases de datos RDF, etc.

En primeiro lugar propoñemos unha colección de estruturas de datos estáticas para a representación de matrices binarias e de enteiros: *1)* unha nova representación de matrices binarias con grandes grupos de valores uniformes, con aplicacións á representación de datos raster binarios; *2)* unha nova estrutura de datos para representar matrices multidimensionais; *3)* unha nova estrutura de datos para representar matrices de enteiros con soporte para consultas top-k. Tamén propoñemos unha nova representación dinámica de matrices binarias, unha nova estrutura de datos que proporciona as mesmas funcionalidades que as nosas propostas estáticas pero tamén soporta cambios na matriz.

As nosas estruturas de datos poden utilizarse en distintos dominios. Propoñemos variantes específicas e combinacións das nosas propostas para representar grafos temporais, bases de datos RDF, datos raster binarios ou xerais e datos raster temporais. Tamén propoñemos un novo algoritmo para consultar conxuntamente datos raster (almacenados usando as nosas propostas) con datos vectoriais almacenados nunha estrutura de datos clásica, amosando que a nosa proposta pode ser máis rápida e usar menos espazo que outras alternativas. As nosas representacións proporcionan interesantes *trade-offs* e son competitivas en espazo e tempos de consulta con representacións habituais nos diferentes dominios.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

The compressed representation of information has been a basic need almost in every domain of Computer Science for a long time. Even though the total amount of storage space is not an important issue nowadays, since external memory (i.e., disk) can store huge amounts of data and is very cheap, the time required to access the data is an important bottleneck in many applications. Accesses to external memory have been traditionally much slower than accesses to main memory, which has led to the search of new compressed representations of the data that are able to store the same information in reduced space.

A lot of research effort has been devoted to the creation of compact data structures for the representation of almost any kind of information: texts, permutations, trees, graphs, etc. Compressed data structures are usually able to perform better than classical equivalents thanks to being stored in upper levels of the memory hierarchy, taking advantage of the speed gap among levels. The goal of a compact data structure is to allow processing the information over the compressed form, supporting algorithms that process the information directly from the compact representation, that will ideally be stored completely in main memory.

In this thesis we focus on the representation of multidimensional data, and especially on the representation of bi-dimensional data that appears in multiple domains in the form of grids, graphs or binary relations. Different data structures exist on the state of the art to efficiently represent and query this kind of data. The wavelet tree [GGV03] is a data structure that can represent a grid where each column contains a single element. Another data structure, called K^2-tree [Lad11], is able to represent general binary grids in compact form, taking advantage of the sparseness and clusterization of 1s in the grid.

Our main goal in this thesis is the development of new and efficient representa-

tions of spatial data, typically in the form of bi-dimensional or general n-dimensional grids. Particularly, we focus on different problems where the information is typically represented using a binary grid (grid of 0s and 1s) or a grid of integer values. We aim to obtain efficient representations that take advantage of different characteristics of the grids in order to achieve compression, while efficiently supporting query operations over the compressed form.

One of the main areas of application of our techniques is the representation of raster data in Geographic Information Systems (GIS), where data is viewed as a grid of values with the usual characteristic that close cells tend to have similar values. This property is very frequent in spatial data and is exploited by typical representations in this domain. However, general data structures like the K^2-tree do not take advantage of this kind of regularities in spatial data.

1.2 Contributions

In this thesis we present a group of data structures that are designed for the compact representation of multidimensional data in different domains. Many of our proposals are based on a compact representation of sparse binary matrices called K^2-tree [Lad11], that was originally designed as a Web graph representation. We describe each contribution separately, detailing the problem it addresses:

1. Our first contribution is the design, analysis, implementation and experimental evaluation of data structures for the compact representation of binary matrices or grids, taking advantage of clusterization of similar values. Our proposal is based on the K^2-tree and is able to compress efficiently binary grids with large regions of 0s and 1s. We call the conceptual proposal "K^2-tree with compression of ones" or K^2-tree1 and introduce several variants or implementations of the proposal. All of them support access to any region of the matrix over the compressed representation. We compare our proposal with alternative representations to show that our variants achieve space competitive with state-of-the-art compact quadtree representations and efficiently support multiple query operations, particularly the efficient access to regions of the grid. A preliminary version of some of the new proposals was published in the 20th String Processing and Information Retrieval Symposium (SPIRE 2013) [dBAGB$^+$13].

2. Our second contribution is the design and implementation of a new compact data structure for the representation of multidimensional binary matrices, or n-ary relations. Our proposal, called K^n-tree, is a generalization of the concepts in the K^2-tree to higher dimensionality problems. We compare the K^n-tree with similar alternatives proposed for the representation of ternary relations using K^2-trees, namely the MK^2-tree and the IK^2-tree, originally designed to represent RDF databases [ÁGBF$^+$14]. We compare our proposal

with these alternatives and study the strengths of each of them and their applicability to different problems. Finally, as a proof of concept of our analysis we apply a K^3-tree (a K^n-tree specific for 3-dimensional problems) to the representation of temporal graphs. As part of this experiment we devise a variant of the IK^2-tree, called diff-IK^2-tree, that is a contribution in itself since it saves much space in comparison with the IK^2-tree when the ternary relation has some regularities, providing an interesting space/time tradeoff. The K^n-tree data structure and its first applications to spatial data were published in the 20th String Processing and Information Retrieval Symposium (SPIRE 2013) [dBAGB+13]. The IK^2-tree and our variant diff-IK^2-tree for temporal graphs were published in the Data Compression Conference (DCC 2014) [ÁGBdBN14].

3. Our third contribution is the design and implementation of a compact representation of multidimensional matrices with efficient support for top-k queries. Our new data structure is called K^2-treap and provides an efficient mechanism to compactly store information in multidimensional databases, supporting general top-k queries or range-restricted top-k queries. We experimentally evaluate our representation in comparison with a state-of-the-art representation based on wavelet trees as well as a simpler representation based on K^2-trees, showing that our data structure is able to overcome state-of-the-art approaches in most cases. A preliminary version of this contribution was published in the 21st String Processing and Information Retrieval Symposium (SPIRE 2014) [BdBKN14].

4. Our fourth contribution, covered in Part II, is the design, implementation and experimental evaluation of new dynamic data structures for the compact representation of dynamic binary relations. Our proposal, the dK^2-tree or dynamic K^2-tree, is a dynamic version of the K^2-tree that supports all the query algorithms provided by static K^2-trees and at the same time supports update operations over the binary matrix, namely changing the value of existing cells and adding/removing rows/columns of the matrix. We experimentally evaluate our proposal to show it has a small space overhead on top of the static representation. Additionally, we experimentally evaluate different query algorithms in static and dynamic K^2-trees to show that the time overhead due to the dynamic implementation can be very small in many real-world queries. Using the dK^2-tree as a basis we devise new dynamic representations of RDF graphs, temporal graphs and raster data and we experimentally evaluate all of them to show that the overhead required by the dynamic representation is reasonable in most cases. The new data structure was presented in a preliminary form in the Data Compression Conference (DCC 2012) [BdBN12].

5. Our fifth contribution is the design, implementation and experimental

evaluation of new representations to solve different problems in GIS related to the storage and querying of raster data. We can divide this contribution in three, since we study three different problems:

(a) We design techniques for the compact representation of general raster data (non-binary raster images) using proposals based on the previous contributions. From this research two new data structures are created, the AMK^2-tree1 (a variant of the MK^2-tree specific for this problem) and the IK^2-tree1 (combination of the IK^2-tree and our first contribution the K^2-tree1). We also study the application of other of our proposals, including a K^3-tree. All our representations are compact data structures that support efficient navigation in compressed form, requiring less space than state-of-the-art alternatives and being able to work efficiently in main memory.

(b) We design techniques for the compact representation of temporal raster data (moving regions) using similar proposals to those in the previous contribution. We develop specific algorithms to answer spatio-temporal queries in our proposals, and we experimentally evaluate all of them in this new context, showing that they are very compact and still able to efficiently answer relevant queries.

(c) We present new algorithms to simultaneously query (join queries) raster datasets stored using our compact data structures and vectorial datasets stored using an R-tree. We study the problem in binary rasters (using the K^2-tree1), showing that our algorithm significantly reduces the number of accesses required in the R-tree. We also extend our evaluation to general rasters (using an AMK^2-tree1 to represent the raster data), where our experimental evaluation shows that our representation requires less space and is faster than simpler algorithms based on sequential traversals of the raster dataset.

Preliminary versions of some of the proposals for the representation of general raster data have been published in the 20th String Processing and Information Retrieval Symposium (SPIRE 2013) [dBAGB$^+$13].

1.3 Structure of the Thesis

First, in Chapter 2, we introduce some basic concepts about data compression and succinct data structures, along with different well-known universal compression techniques. Then, in Chapter 3, we introduce several data structures that are extensively used and referred to throughout this thesis, particularly a compact representation of grids called K^2-tree. After this, in Chapter 4 we introduce basic concepts in different domains where our representations will be applied: the

representation of time-evolving or temporal graphs, the representation of RDF graphs and the representation of spatial and spatio-temporal data in Geographic Information Systems, focusing mostly in the raster model of representation and existing proposals for the representation of raster images.

After these introductory chapters, the contributions of the thesis are grouped in three main parts:

- ***Part I*** explains the new data structures developed for the compact representation of static multidimensional data, including binary and n-ary relations:
  - Chapter 5 introduces our proposals for the compact representation of binary matrices. Several representations are proposed for the compact representation of binary matrices taking advantage of the clusterization of regions of 0s and 1s. Our proposals can be seen as an evolution of the K^2-tree, and they are compared with original K^2-trees and among themselves to show their application in different contexts. Considering the representation of binary raster images as the key application domain, our proposals are compared also with other existing representations to show their compression properties and query efficiency.
  - Chapter 6 introduces alternative approaches for the representation of ternary or n-ary relations. We introduce the K^n-tree, a new compact data structure to represent n-ary relations. We also study existing alternatives, also based on the K^2-tree, called MK^2-tree and IK^2-tree. As a proof of concept, we propose approaches for the compact representation of temporal graphs using a K^n-tree and a new variant of the IK^2-tree proposed in this thesis, to study the differences between them, and we compare them with existing alternatives to show their applicability.
  - Chapter 7 introduces the K^2-treap, a data structure for the compact representation of multidimensional grids with efficient support of top-k queries and range-restricted top-k queries (or *prioritized range queries*). Several proposals are introduced for the representation of general multidimensional datasets as well as for taking advantage of regularities in the data, particularly clusterization of values.
- ***Part II*** presents a new data structure called *dynamic K^2-tree (dK^2-tree)* that can compactly represent binary relations while supporting different update operations:
  - Chapter 8 details the dK^2-tree data structure and the implementation of query and update algorithms on top of the dynamic representation and provides an experimental evaluation of the data structure, comparing its space and query efficiency with the equivalent static representation and testing its update capabilities.

  - Chapter 9 presents an experimental evaluation of the dK^2-tree in comparison with the static K^2-tree in different domains where the K^2-tree has already been used: RDF databases, raster images and temporal graphs. The comparison is focused on the space and time overhead required by the dK^2-tree to answer the same queries, and its efficiency compared with existing alternatives.

- ***Part III*** presents the application of our proposals to the representation of raster data in Geographic Information Systems:

  - Chapter 10 introduces new proposals for the compact representation of general raster images, including direct applications of the different data structures presented in Part I and new variants of previous contributions especially designed for this problem. An experimental evaluation of the proposals is presented, providing a comparison with state-of-the-art alternatives.
  - Chapter 11 presents data structures for the compact representation of time-evolving region data, or multiple overlapping binary images. Again, the proposals introduced in this chapter are variants of the data structures in Part I especially tuned to support the typical queries over spatio-temporal data. An experimental evaluation of the different proposals is presented, and a comparison with alternative representations is provided.
  - Chapter 12 introduces new algorithms to simultaneously query a raster dataset represented using our proposals and a vectorial dataset represented using an R-tree index. We provide an algorithm for binary rasters and another for general rasters, and we experimentally evaluate both of them to show the ability of our proposal to reduce the number of accesses to the R-tree representation and its efficiency in comparison with simpler algorithms based on uncompressed representations of the raster dataset.

Finally, the thesis is completed in Part IV, that summarizes the contributions of all our work and gives a general overview on future lines of research. Appendix A also shows a list of international publications and other research activities related to this thesis.

Chapter 2

Basic Concepts

In this thesis we propose new compressed data structures for the compact representation of information in different domains. In order to understand many concepts used in the thesis we need to introduce some basic concepts about compression and information retrieval. In addition, the data structures we propose make use of existing, well-known succinct data structures, hence we also need to introduce some of these data structures for a better understanding of our proposals.

This chapter presents some of the basic concepts that are needed for a better understanding of the data structures in this thesis. Sections 2.1 and 2.2 introduce some concepts related with Information Theory in order to understand the basis of Data Compression, and some popular compression techniques. Sections 2.3 and 2.4 introduce succinct data structures and describe existing static and dynamic succinct data structures to solve *rank* and *select* operations in binary and general sequences and binary relations.

2.1 Information Theory and data compression

2.1.1 Basic concepts on Information Theory

Information Theory deals with transmission of information through communication channels. Shannon's work [SW49] is considered to be the basis of the field and provides many interesting concepts that are widely used nowadays. The goal of Information Theory is to *measure* the information considering the minimum amount of space required to encode it.

Given a discrete random variable X with domain d_X and probability mass function $P(X)$, the amount of information or "surprise" associated with an outcome $x \in d_X$ is defined by the quantity $I(x) = \frac{1}{\log P(x)}$. The idea behind this formula is that outcomes that are more likely to appear contain a smaller amount of

information. For example, if $P(x) = 1$ no information is obtained from the outcome because the result was already expected. On the other hand, if $P(x)$ is close to 0, the outcome is a "surprise" and therefore the amount of information in the outcome is large.

The *entropy* of X is a measure of the expected amount of surprise in X. It is defined as:

$$H(X) = E[I(X)] = \sum_{x \in d_X} P(x) \log_2 \frac{1}{P(x)},$$

where I is the amount of information or *information content* described above. Essentially, the entropy $H(X)$ measures the average amount of information that can be obtained from each outcome of the random variable.

A *code* C of X is a mapping from elements in d_X (*source symbols*) of X to sequences in a target alphabet D. Hence, a code determines how each source symbol is transformed into a sequence of symbols $d_1 \dots d_l$, $d_i \in D$. For each symbol $x \in d_X$, the *codeword* $C(x)$ returns the *encoding* of x according to C. A usual target alphabet is $D = 0, 1$, so that codes using this target alphabet encode each symbol as a sequence of bits. In the general case, the code can yield sequences of symbols in any alphabet: for example, in many codes the alphabet is $D = 1, \dots, 256$. The size of the target alphabet $|D|$ determines the number of bits that are required to represent each symbol in D: for instance, if the target alphabet is binary each symbol is represented using a single bit ($\log_2 |D|$) and the code is said to be bit-oriented, while a target alphabet of 256 symbols requires $\log_2 256 = 8$ bits (1 byte) to represent each output symbol, and the code is called a byte-oriented code.

A *message* is defined as a sequence of source symbols $x_1 x_2 \dots x_n$. The *encoding* process of the message consists of applying the code to each symbol in the message and concatenating all the resulting codewords. The result of the encoding is the sequence of target symbols $C(x_1)C(x_2) \dots C(x_n)$. The decoding process is the reverse process, that obtains the source symbol corresponding to each codeword to rebuild the original message.

A code is called a *distinct code* if the mapping of two different source symbols is always different, that is, $\forall x_1, x_2 \in d_X, x_1 \neq x_2 \rightarrow C(x_1) \neq C(x_2)$. A code is *uniquely decodable* if every codeword is identifiable from a sequence of codewords. A uniquely decodable code is called a *prefix code* if no codeword is a proper prefix of any other codeword. Prefix codes are *instantaneously decodable*, that is, each time a codeword is found we can know its value without reading further symbols, hence any encoded message can be decoded without performing any *lookahead*. An instantaneously decodable code is an *optimal* code if, given the source symbols and their probabilities, the code has the minimum average code length among all instantaneously decodable codes.

Example 2.1: Consider the source alphabet $\Gamma = \{a, b, c\}$ and the following encodings:

C_1	C_2	C_3
$a \leftrightarrow 0$	$a \leftrightarrow 00$	$a \leftrightarrow 0$
$b \leftrightarrow 10$	$b \leftrightarrow 01$	$b \leftrightarrow 10$
$c \leftrightarrow 1$	$c \leftrightarrow 011$	$c \leftrightarrow 11$

The code C_1 is a distinct code but not uniquely decodable, because the sequence 10 can be decoded as b or as ca. C_2 is a uniquely decodable, but it is not instantaneously decodable, since the sequence 01 can be decoded as b or c depending on the next bit (we need to perform *lookahead*). C_3 is a prefix code, since no code is prefix of any other. As a consequence of this, it is also instantaneously decodable: given any input sequence, we can immediately decode each new symbol as we read the encoded input.

2.1.1.1 Entropy in Context-dependent Messages

Symbols can be encoded based on their *context*. In general, the probability of a symbol can be modeled taking into account the symbols that have appeared before it. The *context* of a source symbol x is a fixed-length sequence of source symbols preceding x. The length m of this sequence, also called length of the context, determines the *order* of the model. The entropy can be defined depending on the order of the model. Considering a source alphabet of size n, formed by symbols $s_1, \ldots, s_n$, the kth-order model entropy is denoted H_k and is defined as follows:

- *Base-order models* consider that all source symbols are independent and equally like to occur. The entropy for these models is denoted as H_{-1}, results $H_{-1} = \log_2 n$.

- *Zero-order models* assume that source symbols are still independent, but with different number of occurrences. In this case, the zero-order entropy is defined as $H_0 = -\sum_{i=1}^{n} p(s_i) \log_{|D|} P(x)$.

- *First-order models* consider the probability of occurrence of a source symbol s_j conditioned by the symbol s_i ($p_{s_j|s_y}$). The *entropy* is obtained as $H_1 = -\sum_{i=1}^{n} p(s_i) \sum_{j=1}^{n} P_{s_j|s_i} \log_{|D|}(P_{s_j|s_i})$.

- *Second-order models* obtain the probability of occurrence of a source symbol s_k conditioned by the previous occurrence of the sequence $s_i s_j$ (that is, $P_{s_k|s_j s_i}$). Hence, for these models, the *entropy* is computed as $H_2 = -\sum_{i=1}^{n} p(s_i) \sum_{j=1}^{n} P_{s_j|s_i} \sum_{k=1}^{n} \log_{|D|}(P_{s_k|s_j,s_i})$.

- *Higher-order models* work in a similar way, considering the probability of the current symbol conditioned by the previous k symbols.

2.1.2 Data Compression: basic concepts

Data compression is a ubiquitous problem in computer science. Compression techniques are used nearly everywhere to allow the efficient storage and manipulation of large datasets. The huge amount of data (in the form of text, image, video, etc.) that has to be processed and transmitted everyday makes clear the necessity of compression techniques that reduce the size of the data for a more efficient storage and transmission.

Compression is strongly related with entropy. Given a message, the goal of compression is to reduce its size while keeping all the information it carries. Entropy represents the average space required to store a symbol from a given source. Hence, the goal of compression is to minimize the space required in addition to the entropy of the source, that represents the theoretical minimum. The difference between the length of a given code and the entropy of the source is called *redundancy*. Given a message and a code C, and let us call $l(c_i)$ the length of the codeword c_i assigned to the source symbol s_i, then the *redundancy* is described as:

$$R = \sum x \in d_X p(x) l(c_i) - H = \sum_{i=1}^{n} p(x) l(c_i) - \sum_{i=1}^{n} -p(x) \log_{|D|} p(x)$$

According to this formula, the redundancy of a code for a given source is minimized reducing the average codeword length. A code is called a *minimum redundancy code* if it has the minimum possible average codeword length.

2.1.2.1 Classification of compression techniques

Compression requires two different steps: an "encoding" step transforms a message into a "compressed" version, and a "decoding" step allows us to retrieve the original message (or an approximate version of the original message) from the compressed version. We distinguish two main categories of compression methods according to the result of compression/decompression:

- *Lossy* compression applies a transformation to the original message that is not reversible. Using lossy compression it is impossible to recover the original message from the compressed version, and a "similar" message may be obtained instead. Lossy compression is widely used for the compression of audio, images and video, since a similar version of the original suffices in many cases for the receiver of the data. Examples of lossy compression techniques are JPEG[1], the basis of the JPG file format, and the MPEG family of standards[2] used in audio (.mp3) and video (.mpg) files.
- *Lossless* compression is completely invertible: from the encoded message we can recover the original message. This is the usual approach in text

[1] http://www.jpeg.org/jpeg
[2] http://mpeg.chiariglione.org/

compression. We will focus in this section in lossless compression techniques usually applied to text compression.

We can categorize compression techniques according to the model used for compression or according to how the encoding process is realized. According to the model used we distinguish 3 categories of techniques:

- *Static or non-adaptive models.* In these models source symbols are assigned predefined codes, that are completely independent of the message. Although these codes can be obtained from a previous study of probability distribution for similar messages, there is no guarantee that the predefined frequency distribution will fit the current message. These techniques are simple to implement, but the compression obtained is usually poor. The *Morse* code is an example of static model.

- *Semi-static models.* In these models the encoding is adapted to the frequency distribution of the source. Usually in these models encoding is performed in two steps (*two-pass techniques*): in a first step, the message is scanned and the frequency distribution of the symbols in the source alphabet is computed. Then, codewords are assigned to each symbol in the source alphabet attending to the frequency distribution. In a second pass, the message is processed again, replacing each source symbol with the corresponding codeword. The encoded message must contain, in addition to the sequence of codewords, the mapping between source symbols and codewords. Semi-static techniques can obtain much better compression than static models. As a counterpart, semi-static techniques need to know and process the complete message before compression can start, so they are not suitable for the compression of data streams. Some representative semi-static compression techniques are Huffman codes [Huf52] or End-Tagged Dense Code (ETDC) and (s, c)-Dense Code, from the family of Dense Codes [BFNP05].

- *Dynamic of adaptive models.* These methods, also known as *one-pass* techniques, encode the message in a single pass over the text without preprocessing it. To do this, dynamic methods usually start with an empty *vocabulary*, or empty matching between symbols and codewords, and update their vocabulary as they process new symbols in the message. Each time new symbols appear, they are added to the vocabulary, and the frequency distribution of the previously found symbols can be updated with each new symbol, adapting the encoding process to the content of the message. In these techniques the compressor and decompressor must define strategies for updating the vocabulary that allow the decoder to rebuild the vocabulary used by the encoder from the encoded sequence. To do this, the encoder and the decoder usually share the same representation of the vocabulary and the same update algorithms. In addition, new symbols or changes in the vocabulary

can be added to the encoded sequence so that the decoder can update its representation. Examples of dynamic or adaptive models are the Lempel-Ziv family [ZL77, ZL78, Wel84], Prediction by Partial Matching [CW84] (PPM) and arithmetic coding [Abr63, WNC87, MNW98]. The Dynamic ETDC and Dynamic (s, c)-Dense Code, from the family of Dense Codes [BFNP05], also use adaptive models.

According to how the encoding process is performed, we distinguish two main families of techniques:

- *Statistical techniques*: these methods assign a codeword to each source symbol whose length depends on its probability. Compression is achieved by assigning shorter codewords to more frequent symbols. Well-known statistical compressors are Hufmman codes [Huf52], Dense Codes [BFNP05], and arithmetic codes [Abr63, WNC87, MNW98].
- *Dictionary techniques*: these techniques use a *dictionary* of substrings that is built during compression. Sequences of source symbols are then replaced (encoded) by small fixed-length pointers to an entry in the dictionary. Therefore, compression is obtained as long as large sequences are substituted by pointers with less space requirements. The Lempel-Ziv family of compressors are the main example in this category [ZL77, ZL78, Wel84].

2.2 Popular compression techniques

In this section we describe some compression techniques that are widely used, either alone or in combination with other methods. These compression techniques are frequently used as the building block of many compact representations. All the compression techniques introduced in this chapter are lossless techniques.

2.2.1 Run-Length Encoding

Run-Length encoding (RLE) is a classic and simple compression technique. A message is encoded representing sequences of equal symbols with a representation of the symbol and the number of repetitions of the symbol. For example, given a message *aaabbbbbaaaccccbbbb*, a run-length encoding of the message would be 3*a*5*b*3*a*4*c*4*b* (3 *a*'s, 5 *b*'s, 3 *a*'s, and so on).

Even though RLE is an extremely simple encoding that only takes advantage of repetitions of identical symbols (also known as *runs*), it is widely used for compression in different domains due to its simplicity. It is particularly useful to compress binary data where long runs of 1s and 0s are expected. Because of this, RLE is used as a simple compression technique for bitstrings in the field of compressed data structures, and for the compression of black and white images

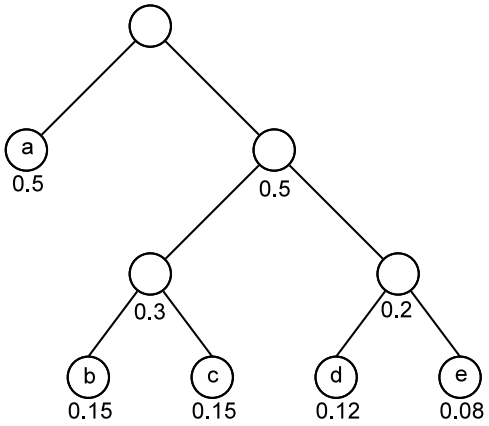


Figure 2.1: Example of Huffman tree.

in multiple standards for binary image compression in fax machines, usually in combination with other techniques.

2.2.2 Huffman

The classic Huffman encoding [Huf52] is one of the best-known compressors, with many applications to compression of text, images, etc. It is a statistical semi-static compressor. An important characteristic of Huffman encoding is that Huffman codes are optimal prefix codes for any source. The process to generate the codewords for a set of source symbols, given their frequency distribution, is based on the construction of a binary tree. Each node in this tree, known as Huffman tree, will have either 0 or 2 children. The Huffman tree is built from the leaves to the root as follows: first, a node is created for each source symbol, and the frequency of the source symbol becomes the *weight* of the node. Then, the two nodes with lower weight are selected and a new internal node is added to the tree containing those nodes as children. The weight of the new node will be the sum of its children's weights. In the next iterations the nodes already merged will not be considered: instead we will consider the new created node. The process continues merging the two nodes with smaller weight until all nodes are merged and the Huffman tree is complete. Once the tree is created, codewords are assigned to source symbols (leaf nodes) according to their path in the tree (a left branch is a 0, a right branch is a 1). Because nodes with smaller weight (less frequency) are merged first, they will belong to lower levels of the tree, and therefore have longer codewords.

Figure 2.1 shows an example of Huffman tree built for an alphabet $\{a, b, c, d, e\}$. In the first step, nodes e and d are combined. The new node has weight 0.2. In the second step, nodes b and c have the smallest weight, so they are combined to

create another internal node with weight 0.3. Then, the two newly-created internal nodes are merged, and finally node a is merged with the rest of the tree. The final Huffman codes would be $a \leftrightarrow 0$, $b \leftrightarrow 100$, $c \leftrightarrow 101$, etc. Notice that when several nodes have the same weight we can choose different methods for generating the Huffman tree, what would lead to different codes.

A Huffman-encoded message must include, in addition to the sequence of codewords, the shape of the Huffman tree created from the source symbols. Using this information, the decompressor is able to decode the symbol from the codeword: each time it reads a bit, it advances on the huffman tree (left if the bit was 0, right otherwise). When a leaf is reached, the codeword is translated into the symbol associated to the leaf, and the decoder moves back to the root of the Huffman tree.

2.2.3 The Lempel-Ziv family

The Lempel-Ziv compressors are probably the best-known and more used dictionary-based and adaptive compressors. LZ77 [ZL77] and LZ78 [ZL78] are the basis of widely used compressors such as *gzip*[3] and *p7zip*[4]. LZW [Wel84] is a variant of LZ78 that is used in Unix's *compress* and the *GIF* image format among others. This family of compressors does not achieve in general the best compression values, being worse that other adaptive techniques like arithmetic encoding. Their main advantage is their compression and decompression speed, and particularly their fast decompression speed is one of the reasons of their wide use in general-use compressors and file formats.

Even though the complete family of compressors is widely used, we will describe only the LZW that will be referred to later in this thesis. LZ77 and LZ78 are similar in the basics but they differ in the way the dictionary is built and how the output is created.

LZW is a widely used variant of LZ78. It uses a dictionary containing substrings of the source message previously encoded. First, the dictionary is initialized to contain all the symbols in the source alphabet. Because all the symbols are initially in the dictionary, LZW output will consist entirely of pointers to dictionary entries (that is, it does not need to output symbols). Compression is performed reading the source message from left to right. At each compression step, LZW reads the source message while there is a matching sequence in the vocabulary. When a non-matching symbol is found, LZW outputs the entry in the dictionary that corresponds to the longest matching sequence, and adds to the dictionary the result of concatenating the next symbol to the matching sequence. The next processing step will start at the non-matching symbol.

[3] http://www.gzip.org
[4] http://www.7-zip.org

Input	Next symbol	Output	Dictionary
			$entry_0 = a$ $entry_1 = b$ $entry_2 = c$
a	b	0	$entry_3 = ab$
b	a	1	$entry_4 = ba$
ab	a	3	$entry_5 = aba$
ab	b	3	$entry_6 = abb$
b	b	1	$entry_7 = bb$
b	c	1	$entry_8 = bc$
c	a	2	$entry_9 = ca$
ab	c	3	$entry_{10} = abc$
c	$\emptyset$	2	–

Table 2.1: Compression of the message *abababbbcabc* using LZW.

Table 2.1 shows the compression steps in LZW and the evolution of the dictionary entries for the message *abababbbcabc*. Notice that the dictionary is initialized to store all the symbols in our source alphabet $\Sigma = \{a, b, c\}$. In the first processing step, we read the input message until we find a non-matching sequence. Since our vocabulary consists only of symbols, the longest matching sequence will be "a", located at $entry_0$ in the vocabulary, and the non-matching symbol (*next* symbol) will be b. Hence we output 0, the position of the sequence a in the vocabulary, and add to the vocabulary the entry ab, resulting from concatenating $entry_0$ with the next symbol b. The next processing step would start from the symbol read in the previous step, the b symbol at the second position. Table 2.1 shows the complete process for the message, that repeats the basic step until until the complete message has been processed.

The maximum size of the dictionary and the strategy used to add/replace substrings to it is a key element in LZW. Once the maximum dictionary size is reached, we can choose between multiple alternatives: not allowing new entries and continue compression with the current one, using a least recently used (LRU) policy to determine which entries of the dictionary may be removed to add the new ones, dropping the current dictionary and building a new (empty) one, etc.

2.2.4 End-Tagged Dense Code and (s,c)-Dense Code

End-Tagged Dense Code (ETDC) [BINP03, BFNP07] is a semi-static statistical byte-oriented encoder/decoder. It is a simple encoding method that achieves very

good compression and decompression times while keeping similar compression ratios to those obtained by Plain Huffman [SdMNZBY00], the byte-oriented version of Huffman that obtains optimum byte-oriented prefix codes.

Consider a sequence of symbols $S = s_1 \dots s_n$. In a first pass ETDC computes the frequency of each different symbol in the sequence, and creates a vocabulary where the symbols are placed according to their overall frequency in descending order. ETDC assigns to each entry of the vocabulary a variable-length code, that will be shorter for the first entries of the vocabulary (more frequent symbols). Then, each symbol of the original sequence is replaced by the corresponding variable-length code.

The key idea in ETDC is to mark the end of each codeword (variable-length code): the first bit of each byte will be a flag, set to 1 if the current byte is the last byte of a codeword or 0 otherwise. The remaining 7 bits in the byte are used to assign the different values sequentially, which makes the codeword assignment extremely simple in ETDC. Consider the symbols of the vocabulary, that are stored in descending order by frequency: the first 128 (2^7) symbols will be assigned 1-byte codewords, the next 128^2 symbols will be assigned 2-byte codewords, and so on. The codewords are assigned depending only on the position of the symbol in the sorted vocabulary: the most frequent symbol will have code `10000000`, the next one `10000001`, and so on, until `11111111` (that is, sequential numbers using the rightmost 7 bits in the byte, while the first bit is a flag to mark that this codeword only contains a single byte). After them will come the 2-byte codewords, starting with `00000000 10000000`, `00000000 10000001` and so on (notice the flag is set to 0 in the first byte to mark that the codeword continues). Table 2.2 shows the complete code assignment for each symbol depending on its position.

The simplicity of the code assignment is the basis for the fast compression and decompression times of ETDC. In addition, its ability to use all the possible combinations of 7 bits to assign codewords makes it very efficient in space. Notice also that ETDC can work with a different chunk size for the codewords: in general, we can use any chunk of size b, using 1 bit as flag and the remaining $b - 1$ bits to assign codes, hence having 2^{b-1} codewords of 1 chunk, $2^{2(b-1)}$ codewords of 2 chunks and so on. Nevertheless, bytes are used as the basic chunk size ($b = 8$) in most cases for efficiency.

Example 2.2: Consider the sequence $afadabaccaadaabbbbabae$, and consider for simplicity that we use a "chunk size" of 3 bits (i.e. $b = 3$). To represent the sequence using ETDC we sort the symbols according to their frequency $a(10) > b(6) > c(2) > d(2) > e(1) > f(1)$ (the order of symbols with the same frequency is not important). The most frequent symbols would be assigned sequential 1-chunk codes: $a \leftrightarrow 100$, $b \leftrightarrow 101$, $c \leftrightarrow 110$, $d \leftrightarrow 111$. The less frequent symbols would be assigned 2-chunk codes: $e \leftrightarrow 000100$, $f \leftrightarrow 000101$.

Position	Codeword	length	
0	10000000	1	
1	10000001	1	
2	10000010	1	2^7
...	...	...	
127 (2^7-1)	11111111	1	
128 (2^7)	00000000 10000000	2	
129	00000000 10000001	2	
...	...	...	$2^7 \times 2^7$
255	00000000 11111111	2	
256	00000000 10000000	2	
257	00000000 10000001	2	
...	...	...	
65511	00000000 10000001	2	
65512	00000000 00000000 10000000	3	
65512	00000000 00000000 10000001	3	
...	...	...	$(2^7)^3$

Table 2.2: ETDC codeword assignment.

(s,c)-Dense Code. The *flag* used in ETDC to mark the end of a codeword means that ETDC uses 128 (2^{b-1}) values (from 0 to 127) to represent symbols that do not end a codeword, called *continuers* (c), and other 128 values (from 128 to 255) for symbols that end a codeword, called *stoppers* (s). An enhancement (or generalization) of ETDC has been proposed where the number of stoppers and continuers can be any value as long as $s+c=2^b$. This proposal, called *(s,c)-Dense Code* or *(s,c)*-DC [BFNP07] considers that all the bytes with values between 0 and s are stoppers, while the remaining bytes are continuers. The codeword assignment is very similar to ETDC: the first s words are given one-byte codewords, words in positions s to $s+sc-1$ are given two-byte codewords, etc.

2.2.4.1 Dynamic End-Tagged Dense Code

Dynamic End-Tagged Dense Code (DETDC [BFNP05]) is an adaptive (one-pass) version of ETDC that is also generalized to a Dynamic (s,c)-DC. As an adaptive mechanism, it does not require to preprocess and sort all the symbols in the sequence before compression. Instead, it maintains a vocabulary of symbols that is modified according to the new symbols received by the compressor.

The solution of DETDC for maintaining an adaptive vocabulary is to keep the

vocabulary of symbols always sorted by frequency. This means that new symbols are always appended at the end of the vocabulary (with frequency 1), and existing symbols may change their position in the vocabulary when their frequency changes during compression.

The process for encoding a message starts reading the message sequentially. Each symbol read is looked up in the vocabulary, and processed depending on whether it is found or not:

- If the symbol is not found in the vocabulary it is a new symbol, therefore it is appended at the end of the vocabulary with frequency 1. The encoder writes the new codeword to the output, followed by the symbol itself. The decoder can identify a new symbol because its codeword is larger than the decoder's vocabulary size, and add the new symbol to its own vocabulary.

- If the symbol is found in the vocabulary, the encoder simply writes its codeword to the output. After writing to the output, the encoder updates the frequency of the symbol (increasing it by 1) and reorders the vocabulary if necessary. Since the symbol frequency has changed from f to $f + 1$, it is moved to the region where symbols with frequency $f + 1$ are stored in the vocabulary. This reordering process is performed swapping elements in the vocabulary. The key of DETDC is that the encoder and the decoder share the same model for the vocabulary and update their vocabulary in the same way, so changes in the vocabulary during compression can be automatically performed by the decoder using the same algorithms without transmitting additional information.

ETDC stores a hash table H that contains all the symbols in the vocabulary. For each symbol, its value, current frequency and position in the vocabulary are stored in a node of the hash table. In addition, an array *posInH* contains an entry for each symbol in the vocabulary, sorted in descending order by frequency. Each entry in *posInH* is a pointer to the entry of the hash table that contains the symbol. Finally, another array *top* is used: $top[i]$ contains the position in the vocabulary of the first symbol with frequency i.

Let n be the current size of the vocabulary (initially $n = 0$). To encode a symbol, the encoder first looks up the symbol in H. If it does not exist, it is inserted in H with frequency $f = 1$, $posInH[n]$ is updated to point to the new entry and n is increased by 1. If the symbol already existed in H, let f be its previous frequency and p its position: the encoder first encodes the symbol and then increases its frequency to $f + 1$ in H. Then the positions p and $top[f]$ are swapped in *posInH* (that is, we swap the current symbol with the first element of the vocabulary with frequency f), and the value of $top[f]$ is increased by 1. In this way, the new *top* value still points to the first element with frequency f, and the symbol encoded is now in the region of symbols with frequency $f + 1$ (it is actually the last symbol in that region).

DETDC and its variants are able to obtain very good results to compress natural language texts, obtaining compression very close to original ETDC without the first pass required by the semi-static approach.

2.3 Succinct data structures

Succinct data structures aim at representing data in space close to the information-theoretic lower bound while still being able to efficiently solve the required operations over the data. The reduced space achieved by succinct data structures allows them to work in faster levels of the memory hierarchy. Even though algorithms to access succinct data structures are usually more complex, their reduced space allows them to provide efficient access times, even surpassing the results of simpler compressed or uncompressed representations. A lot of work has been devoted to obtain succinct representations of trees [Jac89, MR97, FM11], texts [GGV03, FM05], strings [GGV03, GMR06, HM10], graphs [Jac89, MR97, FM08], etc.

2.3.1 Rank and select over bitmaps

One of the first presented succinct data structures consisted of bit vectors (often referred to as bitmaps, bit strings, etc.) supporting *rank* and *select* operations [Jac89]. These basic operations constitute the basis of many other succinct data structures.

Let be $B[1,n]$ a binary sequence of size n. Then *rank* and *select* are defined as:

- $\text{rank}_b(B,p) = i$ if the number of occurrences of the bit b from the beginning of B up to position p is i.
- $\text{select}_b(B,i) = p$ if the $i-th$ occurrence of the bit b in the sequence B is at position p.

Given the importance of these two operations in the performance of other succinct data structures, like *full-text indexes* [NM07], many strategies have been developed to efficiently implement *rank* and *select*.

Jacobson [Jac89] proposed an implementation for this problem able to compute *rank* in constant time. It is based on a two-level directory structure. The first-level directory stores $\text{rank}_b(B,p)$ for every p multiple of $s = \lfloor \log n \rfloor \lfloor \log n/2 \rfloor$. The second-level directory holds, for every p multiple of $b = \lfloor \log n/2 \rfloor$, the relative *rank* value from the previous multiple of s. Following this approach, $\text{rank}_1(B,p)$ can be computed in constant time adding values from both directories: the first-level directory returns the *rank* value until the previous multiple of s. The second-level directory returns the number of ones until the previous multiple of b. Finally, the number of ones from the previous multiple of b until p is computed sequentially

over the bit vector. This computation can be performed in constant time using a precomputed table that stores the *rank* values for all possible block of size b. As a result, *rank* can be computed in constant time. The *select* operation can be solved using binary searches in $O(\log \log n)$ time. The sizes s and b are carefully chosen so that the overall space required by the auxiliary dictionary structures is $o(n)$: $O(n/\log n)$ for the first-level directory, $O(n \log \log n/\log n)$ for the second-level directory and $O(\log n \log \log n)$ for the lookup table. Later works by Clark [Cla96] and Munro [Mun96] obtained constant time complexity also for the *select* operation, using additional $o(n)$ space. For instance, Clark proposed a new three-level directory structure that solved $select_1$, and could be duplicated to also answer $select_0$.

The previous proposals solve the problem, at least in theory, of adding *rank* and *select* support over a bit vector using $o(n)$ additional bits. However, further work was devoted to obtain even more compressed representations, taking into account the actual properties of the binary sequence [Pag99, RRR02, OS07]. Pagh [Pag99] splits the binary sequence into compressed blocks of the same size, each of which is represented by the number of 1 bits it stores, and the number corresponding to that particular subsequence.

Raman et al. [RRR02] represent the compressed binary sequence as a collection of blocks of size $u = \frac{\log n}{2}$. Each block is encoded as a pair (c_i, o_i), where c_i indicates the number of 1 bits it contains (the *class* of the block), and o_i, represents the *offset* of that block inside the sorted list of blocks with c_i $1s$. In this way, blocks with few (or many) 1s require shorter identifiers and the representations achieves the zero-order entropy of the binary sequence. This approach supports *rank* and *select* in constant time for both 0 and 1 bits, but is more complex than previous representations.

Okanohara and Sadakane [OS07] propose several practical alternatives. Each of their variants has different advantages and drawbacks and may be applied in different cases.

Another alternative study, called *gap encoding*, aims to compress the binary sequences when the number of 1 bits is small. It is based on encoding the distances between consecutive 1 bits. Several developments following this approach have been presented [Sad03, GGV04, BB04, GHSV06, MN07].

2.3.2 Rank and Select over Arbitrary Sequences

The operations *rank* and *select* have been extended to sequences of symbols. Given a sequence $S = s_1 s_2 \dots s_n$ of symbols from an alphabet Σ, *rank* and *select* can be described as:

- $\text{rank}_s(S, p)$ is the number of appearance of symbol s in S until position p.
- $\text{select}_s(S, i)$ is the position of the i-th occurrence of symbol s in S.

The sampling solutions proposed for bitmaps are difficult to use here, so the typical solution to provide *rank* and *select* in sequences over general alphabets is to reduce the problem to *rank* and *select* on bitmaps. A trivial solution would be to build a bit vector for each $s \in \Sigma$, thus being able to solve $\text{rank}_s(S, p)$ and $\text{select}_s(S, i)$ in constant time using any of the solutions for bitmaps. However, this solution is unfeasible for general sequences as it would require $n \times |\Sigma|$ bits, much more than the size of the sequence.

2.3.2.1 Wavelet Trees.

The *wavelet tree* [GGV03] is a data structure that allows us to compute *rank* and *select* over arbitrary sequences of symbols. A wavelet tree is simply a balanced binary tree that stores a bit vector B_v in each node v. The root of the tree contains a bitmap B_{root} of length n, where n is the length of the sequence. $B_{\text{root}}[i]$ will be set to 0 if the symbol at position i in the sequence belongs to the first half of the alphabet, and 1 otherwise. Now the left child of the root will process the symbols of the first half of the alphabet, and the right child will contain the others. The same process is applied recursively in both children, halving the alphabet at each step. The decomposition ends when the alphabet cannot be subdivided. Figure 2.2 shows the wavelet tree representation for an example sequence (notice that only the bitmaps shown for nodes N1, N2 and N3 will be actually stored, the sequences of symbols at each node are shown only for a better understanding of the decomposition process).

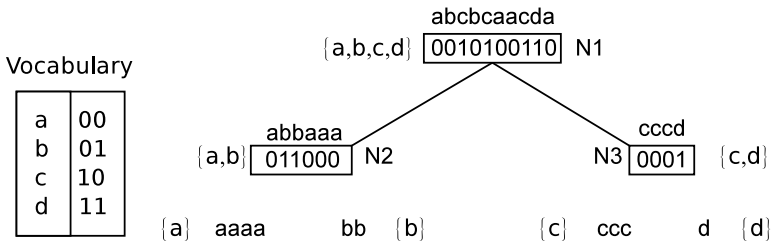


Figure 2.2: Wavelet tree of the sequence *abcbcaacda*.

In a wavelet tree the *rank* and *select* operations over the original sequence are solved efficiently in top-down or bottom-up traversals of the tree. Assume we want to find, in the wavelet tree of Figure 2.2, the number of occurrences of symbol `b` in the sequence until position 8 ($\text{rank}_{\text{b}}(S, 8)$). We know that `b` belongs to the first half of the vocabulary, so that occurrence should be a 0 in the bitmap of N1. We compute $\text{rank}_0(B_{\text{N1}}, 8) = 5$, which gives us the number of elements in the left half of the alphabet until position 8. Hence, since the left child of N1 contains only the symbols of the left half, we move to N2 and our new position will be 5. In N2 we repeat the binary rank operation. In this case `b` is in the right half of the vocabulary of the node, so we perform $\text{rank}_1(B_{\text{N2}}, 5) = 2$. Since N2 is a leaf node

we return immediately the result 2. If we want to compute $\text{select}_c(S, 2)$ we perform the inverse operation, a bottom-up traversal of the tree. From the position of `c` in the alphabet we can know its position in the leaves of the wavelet tree. In this case, our symbol is represented using 0s in node N3. We compute $\text{select}_0(B_{\text{N3}}, 2) = 2$, which gives us the position we need to check in the upper level. When we move to N1, we know we come from its right child, so the operation we need to compute is $\text{select}_1(B_{\text{N1}}, 2) = 5$. Since we have reached the root, we have obtained the position of the second `c` in S.

Many variants of the wavelet tree have been proposed to obtain better space results. The wavelet tree can achieve the zero-order entropy of the sequence abandoning the balanced tree representation and using instead the shape of the Huffman tree of the sequence, or using the compressed bitmap representation of Raman et al. in each node [GGV03, NM07]. Other representations based on the wavelet tree have also been developed to reduce the space requirements of the tree or improve queries [CNO15]. The wavelet tree has also been generalized to use a a tree of arity ρ, where each node contains a sequence over a vocabulary of size ρ, instead of the usual binary tree with bit vectors in its node. This variant, usually called *generalized wavelet tree* or *multi-ary* wavelet tree, reduces the height of the tree and also the query time, based on constant-time *rank* and *select* support in the sequences of each node.

2.3.3 Compact binary relations

The compact representation of general binary relations, even in a static context, has not received much attention until recent years. However, specific applications of binary relations have been extensively studied. In particular, several approaches for the representation of binary relations have been proposed using wavelet trees [GMR06], taking advantage of the flexibility and theoretical guarantees of this data structure.

One of the most studied methods for the compact representation of general binary relations is the reduction of the problem to the representation of sequences. Consider a binary relation $\mathcal{R}$ between a set of n objects and σ labels that contains t pairs. Let us represent the binary relation as a binary matrix M, of size $\sigma \times n$, whose columns are the n objects and the rows the σ labels. In this matrix, a cell will have value 1 only if the object is related with the corresponding label. Therefore, the number of ones in M will be equal to t, the number of pairs in $\mathcal{R}$. The matrix can be represented using a bitmap B of length $n + t$ and a string S of length t over alphabet Σ. The bitmap B stores the number of elements in each row of the matrix in unary. Figure 2.3 shows an example of this transformation.

Barbay et al. [BGMR07] first showed how to solve *rank* and *select* operations on the objects and labels of the binary relation represented in this way, given that S and B support *rank* and *select* operations themselves. Therefore, their solution

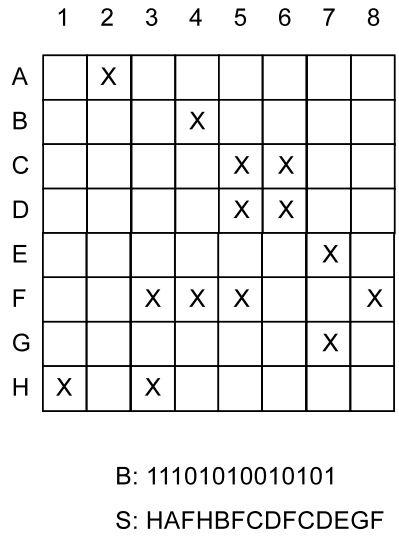


Figure 2.3: Reduction of binary relation to strings.

provides support for a relevant set of operations in a binary relation relying on the existing implementations of bit vectors and sequences.

The reduction of basic operations over the binary relations to operations on S and B is quite straightforward. For instance, to retrieve all the labels related with object 5 in the example of Figure 2.3, we simply need to access $S[\text{select}_1(B, 5), \text{select}_1(B, 6) - 1]$. To retrieve only the second label related with object 5 we would access directly the second element counting from $\text{select}_1(B, 5)$. To find the number of occurrences of the label c up to position 7 we can simply compute the number of occurrences of that symbol in S before the start of column 7: $\text{rank}_c(S, \text{select}_1(B, 7) - 1)$. More complex operations, involving for example ranges of objects and labels, can also be performed using specific algorithms supported by wavelet trees.

2.3.4 Maximum and top-k queries

2.3.4.1 Range minimum queries

Given a sequence of values $S[1, n]$, range minimum (maximum) queries (RMQ) ask for the minimum (maximum) value in a subsequence $[s_i, s_e]$ of S. The problem of efficiently locating the minimum/maximum for any arbitrary subsequence has been extensively studied since it has multiple applications in more elaborate queries. Several data structures have been proposed to compute the range minimum/maximum in constant time using as few additional space as possible.

Currently the best solutions to answer range minimum(maximum) queries is able to answer queries in constant time using only $2n + o(n)$ bits [Fis10]. The representation is based on a succinct tree of n nodes, that is queried to locate the position of the minimum for any range in constant time. An important fact of this and other RMQ data structures is that they point to the *position* of minimum, so they must be stored together with the sequence if we are interested in the value of the maximum. However, if we are interested only in the location of the maximum we can discard the sequence and store only the RMQ data structure.

2.3.4.2 Top-k queries on sequences and grids

Top-k queries, in general, are concerned with locating the most *relevant* items from a collection. Given any comparable value that estimates the relevance of an item, a top-k query asks for the k items with higher relevance value in the collection, or the k maximum values.

In a sequence of values, top-k queries can be answered easily using an RMQ data structure over the sequence. The process outputs elements one by one, as follows: first, the maximum in the range is found using the RMQ data structure and added to a priority queue, together with the interval of which it was the maximum. Then, elements are extracted iteratively from the priority queue. Each time an element is extracted from the priority queue, it is emitted as a new result, and its associated range is divided in two sub-ranges to its left and its right, computing the top elements for each of them and adding them to to the priority queue. The process finishes after k iterations or when the priority queue is empty.

Consider now a $[m \times m]$ grid where each column can contain a single 1. This grid can be represented using a wavelet tree exactly like a sequence, but in this grid we want to answer 2-dimensional top-k queries. The top-k problem is reduced to locating the position in the leaves of the wavelet tree where the maximum values occur. Assume that we add RMQ data structures to all the nodes of the wavelet tree, that will allow us to locate the position in each node where the maximum value of the corresponding subsequence occurs. This suffices to answer 2-dimensional top-k queries on a wavelet tree for any arbitrary range [NNR13]. First we locate all the nodes of the wavelet tree that intersect the query range $R = [x_1, x_2] \times [y_1, y_2]$, and the corresponding intervals in those nodes that fall within the range. Starting at the root of the wavelet tree with the interval $[x_1, x_2]$, we recursively add the children of each tested node, keeping track of the corresponding interval in each node that intersects with $[x_1, x_2]$ and stopping traversal when this interval becomes empty. Each leaf of the wavelet tree reached will only be considered if it intersects the interval $[y_1, y_2]$. Once this collection of nodes is found, the process to answer top-k queries is simple: the top element in each node is found, and the maximum among all the nodes is selected as the first result. To compute this, the top elements for each node are added to a priority queue, and the maximum is then extracted. Each time an element is extracted from the priority queue, the interval where it

was found is divided into two sub-intervals to its left and its right, and the top elements for each of these sub-intervals is added to the priority queue. For each result, its x and y coordinates can always be found traversing the wavelet tree up or down in $O(\log m)$ time. It can be proved that the number of wavelet tree nodes intersecting a range will be $O(\log m)$, so the wavelet tree is able to answer top-k queries in $O((k + \log m)(\log km))$ time.

2.4 Succinct dynamic data structures

Many proposals exist for the succinct representation of static bit vectors, sequences, etc. However, the development of practical and efficient dynamic versions of the same data structures is still an open problem. In this section we will introduce some succinct dynamic data structures proposed for the representation of dynamic bit vectors or sequences. Some of these data structures are able to obtain in theory optimal space and query times. Nevertheless, most of these dynamic representations are not very practical, so practical implementations of dynamic data structures need to build upon simpler solutions that work well in practice but do not fulfill the theoretical optimal bounds.

2.4.1 Dynamic bit vector representations

The representation of a fully dynamic bit vector, or *dynamic bit vector with indels* problem, consists of storing a bit vector B of length n, supporting the basic operations *rank*, *select*, *access* as well as the following update operations:

- *flip*(i): change the value of the bit at position i.
- *insert*(i, b): insert a new bit at position i, with value b.
- *delete*(i): delete the bit at position i.

This problem and its simplified variants have been studied many times in the past. The *subset rank* problem, that only considers *rank* and *flip* operations, was studied in early works. A lower bound of $\Omega(\log n / \log \log n)$ amortized time per operation was proved for the subset rank problem in the cell probe model of computation [FS89] and can be extended to the RAM model and the dynamic bit vector with indels problem.

Dietz presented a first solution for the subset rank problem [Die89] as a special case of the *partial sum* problem. The partial sum problem involves storing a sequence A of length l, storing k-bit integers and supporting two operations: *sum(i)*, that returns $\sum_{p=1}^{i} A[p]$, and *update(p,δ)*, that sets $A[p] = A[p] + \delta$ for

$\delta = O(\log^{O(1)} n)$. The subset rank problem is equivalent to the partial sum problem with $k = 1$.

The main idea in Dietz's approach is to solve the problem for "short" lists and then build a complete solution over this. For the short lists (of size $l = O(\log^{\epsilon} n)$, where $\log n$ is the word size and $0 < \epsilon < 1$), the partial sums are represented using two arrays: B stores an "old" version of the partial sums, and C stores the differences at each position. The *sum(i)* operation is computed as $B[j] + \sum_{p=1}^{i} C[j]$. Updates are executed in C, and the complete structure is rebuilt after l updates. The size of the list allows C to be represented in $O(\log^{\epsilon} n \log\log n)$ bits, so updates and sums can be performed in constant time using table lookups with some precomputation. For the representation of general sequences of length n, a tree with branching factor $b = \Theta(\log^{\epsilon} n)$ is built and the sequence is stored in its leaves, from left to right. The *weight* of a node v is defined as the sum of all the leaves in the subtree rooted at v. At each internal node in the tree its weight is stored together with the partial sums of the weights of its children (using the data structure for short sequences). The height of the tree is $O(\log n / \log\log n)$, and operations at each internal node can be computed in constant time.

Raman, Raman and Rao [RRR01] propose later a solution similar to Dietz's but supporting also *select* operations. Again, they use a solution based on a proposal for "short" lists and a b-ary tree that stores the previous structure at each internal node to provide constant-time operations in internal nodes. Building a tree with branching factor $b = O(\log^{\epsilon} n)$ they provide support for all operations in $O(\log n / \log\log n)$ time.

Hon, Sadakane and Sun [HSS03] add *insert* and *delete* operations to those supported by Raman et al., thus supporting the complete set of operations involved in the dynamic bit vector. They propose a modified weight-balanced B-tree [Die89], a tree that stores a sequence in its leaves and whose internal nodes are required to be balanced in *weight* (the total number of 1s in the descendants of the node): each internal node of the tree stores the number of 1s (LW-value) and the number of bits (LN-value) under all its left siblings. Their proposal relies on properties of weight-balanced B-trees to ensure succinct space and efficient query times, but this requires the utilization of additional data structures to support constant-time *rank* and *select* operations in the nodes of the tree. For the dynamic bit vector problem they obtain $n + o(n)$ bits and provide the following tradeoff: for any $b = \Omega(\log n / \log\log n)^2$, they can answer *rank* and *select* in $O(\log_b n)$ time, while updates require $O(b)$ time.

Chan, Hon, Lan, Sadakane: Chan et al. [CHLS07, Section 3.1] proposed a data structure based on a binary tree to represent a dynamic bit vector with *rank* support as part of a solution for managing a dynamic collection of texts. Their data structure is based on a binary search tree. The bit vector is partitioned into

segments of size $\log n$ to $2 \log n$. The nodes of the tree will store the segments, so that the segments read in an inorder traversal of the tree yield the original bit vector. To support the required operations, each node of the tree stores the following information:

- A segment of bits.
- Two integers *size* and *sum* that contain the number of bits and 1s in the subtree.

In this tree, query operations can be performed using the *size* and *sum* counters during traversal to locate the node for a given position and count the number of 1s to its left. The *sum* in each node is used to locate the node that contains a position p, and the *sum* counters are used to determine the number of 1s to the left of any node. Update operations require a search, updating a constant number of segments and also updating the counters in the tree branch. All operations involve at most $O(\log n)$ nodes of the tree, and using universal tables operations in each node are constant-time. This proposal requires $O(n)$ for the structure and solves all operations in $O(\log n)$. Even though its space requirements are high, this representation is used as a simple building block for more elaborate solutions (for instance, the next proposal by Mäkinen and Navarro).

Mäkinen and Navarro enhanced Chan's proposal to obtain a a succinct dynamic bit vector [MN08]. They build upon a balanced binary tree storing *size* and *sum* counters in its nodes, but propose a representation that only requires $n + o(n)$ bits of space using a tree with leaves of size $\omega(\log n)$. This requires a careful partition of the bit vector and additional data structures to answer queries efficiently. The authors also propose a *gap-encoded* version, which allows them to achieve the zero-order entropy of the binary sequence $nH_0 + o(n)$.

2.4.2 Dynamic sequence representations

The representation of dynamic sequences is a very studied problem because it appears frequently in different domains, such as text compression, where the need to efficiently process large datasets is combined with the desire to manage dynamic datasets. In Section 2.3.2 we described how the problem of the static representation of sequences supporting *rank* and *select* was usually based on an efficient solution of the same problem for binary sequences (bitmaps). Something similar occurs in the dynamic version, where the representation of dynamic sequences is also very related with the problem of dynamic bit vectors. Several of the solutions presented for dynamic bit vectors can be extended to handle general sequences.

The proposal of Mäkinen and Navarro [MN08] for bit vectors can be generalized for small alphabets. To handle a general sequence, they simply reduce the problem

to the previous one: given a sequence over a large alphabet, they build a *generalized wavelet tree* where each node contains a sequence of symbols over a small alphabet using their previous solution. Hence, the solution is essentially identical to the process followed in static data structures: a wavelet tree is built and all operations in the sequence are translated into a series of operations at the different levels of the wavelet tree.

Another representation of dynamic sequences is due to Navarro and Nekrich [NN13]. Their proposal is based on a different interpretation of the well-known wavelet tree. They abandon the idea of performing a *rank* operation per level. Instead, they *track* the positions that need to be accessed at each level, thus providing a way to navigate the wavelet tree top-down or bottom-up without actually known the *rank* and *select* values at each level. Their idea is based on a different representation of the wavelet tree nodes. Each node is partitioned in fixed-size blocks, and each block is represented using a special data structure that does not actually store the bitmap. The key in their representation is that block operations are constant-time, and they are able to compute from any block and offset the corresponding positions in the parent/child node. Their proposal achieves the theoretical lower bounds for access and update operations in the sequence.

2.4.3 Compact dynamic binary relations

The proposals based on wavelet trees can be made dynamic using a dynamic wavelet tree representation to support changes in the labels/objects of the binary relation. If B is replaced with a dynamic bit vector representation and S is represented with a dynamic wavelet tree, we can insert easily a new pair in $\mathcal{R}$. The new element (a, b) in $\mathcal{R}$ will be a new entry en S, therefore the insertion of the pair requires to insert the new label in the appropriate position in S and to increase the length of the corresponding column in the binary matrix (insert a bit in B). New objects can also be added very efficiently, as we only need to add a new bit to B to represent the new column with no pairs.

Chapter 3

Previous work

Most of the representations proposed in this thesis are based on an existing data structure, the K^2-tree, that was originally proposed as a compact representation of sparse graphs. The K^2-tree itself and several other data structures used later in the thesis also make use of a technique to provide direct access to sequences of variable-length codes called Directly Addressable Codes. In this chapter we introduce Directly Addressable Codes and the K^2-tree, two data structures that are essential to understand most of the proposals in this thesis. Section 3.1 introduces Directly Addressable Codes, that will later be used in other data structures including the K^2-tree. Then, Section 3.2 provides a conceptual description of the K^2-tree, its structure and basic navigation algorithms, that are necessary to understand most of the contributions in the next chapters.

3.1 Directly Addressable Codes (DAC)

Consider a sequence of symbols encoded using End-Tagged Dense Codes (ETDC), described in Section 2.2.4: the encoded sequence is a sequence of variable-length codes, probably smaller than the original message. However, in order to access a specific symbol we must decode the sequence from the beginning, since we have no way to know where the k-th symbol starts. Directly Addressable Codes [Lad11, BFLN12](DAC) provide a way to represent the sequence using almost the same space used by ETDC (or (s, c)-DC, from the same family of Dense Codes), but providing efficient access to any position in the sequence.

Recall from Section 2.2.4 that codewords in ETDC were variable-length codes of chunk size b, where a bit in each chunk was used as a marker to indicate whether the codeword contained more chunks or not. The key idea in DAC is to separate the chunks of the codewords using the following data structures:

- Three different arrays $L_i, i = 1..3$ are set up, that will contain the i-th chunk

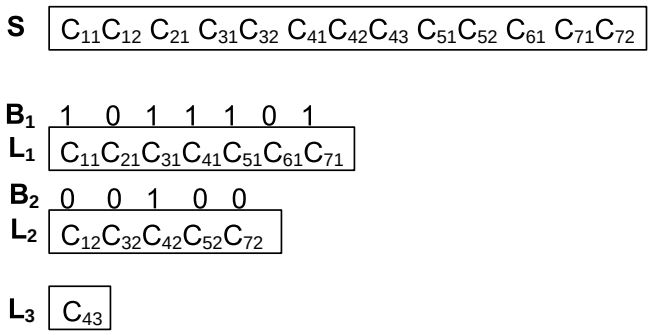


Figure 3.1: Sequence of variable-length codes represented using DACs.

of each codeword (only the $b-1$ bits that correspond to the offset). Each L_i will contain as many entries as codewords exist in the sequence with at least i codewords.

- Three bitmaps B_i contain, for each chunk in L_i, a bit indicating if the corresponding codeword has more chunks or not.

Example 3.1: Figure 3.1 shows an example where a sequence S of variable-length codes is partitioned using DACs. The sequence contains 7 variable-length codes, each of which may contain a different number of chunks: for example, codeword 1 would consists of the chunks C_{11} and C_{12}, while codeword 2 only contains a single chunk C_{21}. These codewords would be the result of creating a vocabulary of symbols using ETDC. In the DAC representation of the sequence, shown below the sequence, the first list L_1 we store the first chunk of all codes, i.e. C_{i1} for all i. In addition, the bitmap B_1 contains, for each chunk in L_1, a 1 if the corresponding code has more chunks or a 0 otherwise: for example, the first bit is a 1 because C_1 has two chunks, but the second bit is 0 because C_2 only contains a chunk. In the second list L_2 we store all the second chunks of the codes that contain at least two chunks, and again a bitmap to know which codes contain a third chunk. The arrays are built from left to right adding the corresponding chunks of each codeword in the sequence.

This scheme in DAC supports efficient access to any code in the sequence. If we want to access the k-th, we can find its first chunk in L_1 simply indexing the array, since the elements are now fixed-size chunks. Then we check $B_1[k]$ to know if the code contains more chunks. If $B[k] = 0$ we return immediately, otherwise we retrieve its second chunk in L_1. Computing $p' = \text{rank}_1(B_1, p)$ we can determine how many codes have at least two chunks until position p, hence we obtain the position p' in L_2 where the second chunk of the code is located. Repeating this process recursively we can obtain the complete code accessing the appropriate positions in

all the L_i's until we find a 0 in B_i. A rank structure supports the *rank* operations in each B_i in constant time. For example, if we want to retrieve code C_5 in S we would start by accessing $L_1[5]$ and find C_{51}. Now we check $B_1[5] = 1$, so we need to find a second chunk. The second chunk will be located at position $p' = \text{rank}_1(B_1, 5) = 4$ in L_2. In $L_2[4]$ we find the second chunk of the code, so now the code is $C_{51}C_{52}$. We access now $B_2[4]$, and we find a 0, so we have finished the computation of the code and return the value $C_{51}C_{52}$.

3.2 The K^2-tree

The K^2-tree [BLN09] is a compact data structure for the representation of sparse binary matrices. It was originally designed as a compressed representation of Web graphs, representing their adjacency matrix in reduced space. In later works, the K^2-tree has been applied to the compression of other kinds of graphs or binary relations, such as social networks [CL11] or RDF databases [ÁGBFMP11]. In this section we will present the K^2-tree as a method for the compression of a binary matrix, that will usually be the adjacency matrix of a simple graph or binary relation.

3.2.1 Data structure and algorithms

Given a binary matrix of size $n \times n$, the K^2-tree represents it using a conceptual K^2-ary tree that represents the complete matrix, for a given K. The size of the matrix is assumed to be a power of K[1]. The root of the conceptual tree corresponds to the complete matrix. Then, the matrix is partitioned in K^2 equal-sized submatrices of size $\frac{n}{K} \times \frac{n}{K}$. The K^2 submatrices are taken in left-to-right and top-to-bottom order and a child node is added to the root of the conceptual tree for each of them. Each node of the tree is labeled with a bit: 1 if the corresponding submatrix contains at least a *one* or 0 otherwise. Then the process continues recursively only for the submatrices that contain at least a *one*. The recursive subdivision of the matrix stops when the current submatrix is full of *zeros* or when the cells of the original matrix are reached. Figure 3.2 shows a 10×10 binary matrix, virtually expanded to size 16×16 (left), and the conceptual K^2-tree that represents it, for $K = 2$.

Following the example in Figure 3.2, in order to access the highlighted cell (row 9, column 6) the conceptual tree can be traversed from the root. The cell (9,6) is in the bottom left quadrant of the matrix, that is, the third child of the root node. This child is labeled with a 1 so the traversal continues recursively for the current submatrix. The traversal continues until a 0 is found (the value of the cell is 0) or the last level of the tree is reached. In our example, the highlighted branches in the conceptual tree are traversed until we finally reach a 1 in the last level, showing that the cell was set to 1 in the matrix.

[1] If n is not a power of K, we use instead $n' = K^{\lceil \log_K n \rceil}$, the next power of K. Conceptually, the matrix is expanded with new zero-filled rows and columns until it reaches $n' \times n'$.

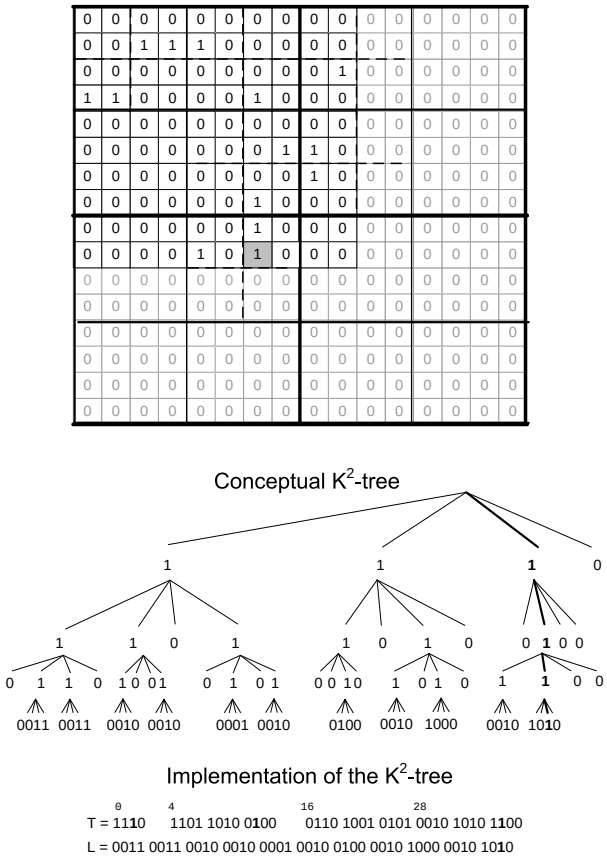


Figure 3.2: Representation of a binary matrix using a K^2-tree, for K =2.

The actual K^2-tree representation of the binary matrix is a succinct representation of the conceptual tree. The conceptual K^2-tree is traversed levelwise and the bits from all the levels except the last one are stored in a bitmap T. The bits of the last level are stored in a second bitmap L. Figure 3.2 shows the bitmaps T and L for the example binary matrix. The conceptual tree is not actually stored, and the actual representation of the K^2-tree consists only of the bitmaps T and L.

In order to retrieve the values of the binary matrix top-down traversals of the conceptual K^2-tree are required. These traversals can be performed efficiently over the K^2-tree taking advantage of a property of the representation: for any 1 located at position p in T, its K^2 children start at position $\text{rank}_1(T, p) \times K^2$ in $T : L$ ($T : L$ represents the concatenation of T and L). This property is due to the fact that leaf nodes (labeled with 0) have no children, and all internal nodes (labeled with 1) have exactly K^2 children. In order to provide efficient navigation, a *rank structure* like those explained in Section 2.3 can be added to the bitmap T, hence supporting each step in the top-down traversal in constant time. Nevertheless, K^2-tree implementations used for most applications use a simpler implementation of *rank* based on a single directory and sequential scan that is not constant-time but is very fast in practice [GGMN05].

Using the basic traversal algorithm based on *rank* operations, K^2-trees can efficiently solve single cell queries, row/column queries or general range reporting queries (i.e., report all the 1s in a range) by visiting all the subtrees that store cells belonging to the queried region. The traversal algorithms start at the root of the tree (position 0 in T) and access at each level all the submatrices that intersect with the row/column or region of interest.

Example 3.2: We want to retrieve the value of the cell at row 9, column 6 in the matrix of Figure 3.2. The path to reach the cell is highlighted in the conceptual K^2-tree in the bottom. To perform this navigation, we would start at the root of the tree (position 0 in T). In the first level, we need to access the third child (offset 2), hence we access position 2 in T. Since $T[2] = 1$, we know we are in an internal node. Its children will begin at position $\text{rank}_1(T, 2) \times K^2 = 12$, where we find the bits `0100`. In this level we must access the second child (offset 1), so we check $T[12 + 1] = T[13] = 1$. Again, we are at an internal node, and its children are located at position $\text{rank}_1(T, 13) \times K^2 = 9 \times 4 = 36$. We have reached the third level of the conceptual tree, and we need to access now the second child (offset 1). Again, $T[36 + 1] = 1$, so we compute the position of its children using $p = \text{rank}_1(T, 36) \times 4 = 80$. Now p is higher than the size of T (40), so the K^2 bits will be located at position $p - |T| = 80 - 40 = 40$ in L. Finally, in L we find the K^2 bits `1010`, and need to check the third element. We find a 1 in L and return the result.

The K^2-tree is able to efficiently represent sparse matrices by compressing in small space large regions of zeros and taking advantage of the clustering of the sparse ones. In the worst case, the total space in bits is $K^2 m \left(\log_{K^2} \frac{n^2}{m} + O(1)\right)$, for an

$n \times n$ matrix with m ones. However, the results are much better in practice. Query times do not have a worst-case guarantee in K^2-trees. The time to retrieve the value of a single cell is proportional to the height of the conceptual tree $\log_K n = O(\log n)$. The time to retrieve all the 1s in a row/column is $O(n)$ in the worst case, even if no 1s are returned. Despite the worst-case analysis, K^2-trees are able to obtain good query times in practice in real datasets.

3.2.2 Improvements

Some improvements have been proposed to obtain better compression results in the K^2-tree. In this section we describe the most relevant modifications that can be added to the basic K^2-tree. This modifications affect the query algorithms but the computations to take them into account are straightforward in most cases, so we will explain them shortly.

3.2.2.1 *Hybrid* K^2-tree: multiple values of K

Higher values of K reduce the number of levels, and therefore improve query times, in the K^2-tree, at the cost of potentially increasing the final size of the structure. However, a careful selection of different values of K depending on the level of the conceptual tree can obtain significant improvements in query time without actually increasing the size. *Hybrid* K^2-tree representations have been proposed [Lad11], using higher values of K in the first levels of decomposition only, and proved to overcome classic representations. The computations required to find the children of a node are slightly different, since we must take into account the different values of K in different levels of the tree, but it is easy to see that we can still compute the children of a node using a *rank* operation if we store additional information to determine the position of the bitmap where the value of K changes. In practice, the computations in the lower levels of the tree need to add an *adjust factor* to correct the rank value and all the navigation is essentially the same.

3.2.2.2 Compression of L: matrix vocabulary

Another major improvement proposed over basic K^2-trees is the use of a matrix vocabulary to compress the lower levels of the conceptual tree [Lad11]. In this variation, the decomposition process is stopped when submatrices of a predetermined size are reached (for example, we can stop decomposition when the submatrices are of size $K \times K$, $K^2 \times K^2$, etc.) This is equivalent to have a different value K' that is used to partition the last level of the tree. Once the value of K' is defined, the conceptual tree is built like any other K^2-tree, where the matrix is decomposed recursively in K^2 submatrices, but in the last level the decomposition creates a $K' \times K'$ submatrix. The enhancement proposed is to represent the sequence of submatrices that appear in the last level using a statistical compressor instead

of the plain bitmap representation. The sequence of submatrices is statistically encoded using Dense Codes (in practice (s, c)-Dense Codes for better compression). Hence, the original sequence of submatrices is replaced by the sequence of variable-length codes that encodes the sequence plus a *matrix vocabulary* that stores the plain representation of each matrix only once, in decreasing order of frequency. The advantage of the statistical encoding is that when the represented matrix contains low-level regularities, this leads to a skewed distribution of the possible submatrices, so this approach can reduce the space requirements since each matrix is only stored once in plain form (in the vocabulary) and the most frequent matrices are represented using shorter codewords.

Example 3.3: In the conceptual tree in Figure 3.2 we want to use a matrix vocabulary to represent the matrices in the last level of the tree. We choose to stop decomposition when we reach submatrices of size 2×2. The new representation will contain a bitmap T identical to the original version. However, the representation of the matrices in the last level will be a sequence of variable-length codes. Following the usual steps of Dense Codes we would build a matrix vocabulary where elements are stored in decreasing order by frequency:

$$0010(5) > 0011(2) > 0001(1) > 0100(1) > 1000(1) > 1010(1)$$

The plain bitmap L is replaced by the matrix vocabulary V and the sequence of variable-length codes S:

$$V = [0010\ 0011\ 0001\ 0100\ 1000\ 1010]$$

$$S = c_1 c_1 c_0 c_0 c_2 c_0 c_3 c_0 c_4 c_0 c_5,$$

where c_i is the i-th codeword in the (s, c)-Dense code scheme selected. If we consider chunks of 2 bits and an ETDC encoding, or (2,2)-DC encoding, the codewords $c_0 = 10$ and $c_1 = 11$ would consist of a single chunk and we would be saving 2 bits per occurrence of the matrices 0010 and 0011. In large datasets with skewed distribution of submatrices the use of variable-length codes to represent the matrices can obtain important savings.

The direct representation of L using a sequence of variable-length codes and a submatrix vocabulary does not allow random access to positions in L, since now we should decompress it sequentially to reach a position. Since random access to L is necessary to perform any query, DACs are used to store the sequence of variable-length codes. Recall from the precious section that the DAC representation is a reorganization of the variable-length codewords that requires little additional space and allows direct access to any position in the sequence. This direct access allows us to access the sequence at any position, reading the corresponding codeword and retrieving the actual matrix from the matrix vocabulary.

Chapter 4

Previous work in application areas explored in this thesis

In this chapter we introduce the main application areas where our representations will be used. This chapter provides an introduction to the specific requirements, problems and state-of the art solutions to each domain, focusing on the problems solved by our representations.

In Section 4.1 we introduce the notion of temporal graph and some existing models and data structures for their representation. Then, Section 4.2 introduces RDF graphs, a graph representation of knowledge that has gained a lot of popularity in the Semantic Web community in recent years, and whose storage and querying is an active line of research. Finally, in Section 4.3 we introduce Geographic Information Systems, the main characteristics and problems in the GIS domain and state-of-the-art representations of geographic data, focusing mainly on the alternatives for the compact representation of raster datasets.

4.1 Temporal graphs

We can define a temporal graph, or time-evolving graph, as a tuple $G = (V, E)$, where V and E are a set of nodes/vertices and edges respectively, and each edge $e \in E$ has somehow associated a collection of *valid times*, that indicate the times at which the edge existed in the graph. In this sense, we can see a temporal graph as the temporal union of its *snapshots* $G_i(V_i, E_i), \forall t_i \in T$, where T is the set of possible time instants. In this way, we consider $V = \cup_i V_i$ and $E = \cup_i E_i$, so that each edge in the temporal graph would be labeled with the timestamps t_i corresponding to the time instants where the edge was active.

In this thesis we will focus on the representation of a specific kind of time-evolving graphs. We assume that the graph is relatively large and it has many

timestamps t_i, instead of just a few snapshots. Additionally, we will focus on graphs that change gradually between timestamps, in the sense that the state of the graph in consecutive time instants is not too different (i.e. the majority of edges keep their state between consecutive time instants).

We will focus on the representation of temporal graphs with efficient support to retrieve the state of the graph at any time point, providing support for different queries associated to time instants or time intervals. Essential queries in this kind of models are successor and predecessor queries (retrieving the nodes related with the given one, because an edge points from the current edge to the successor or from the predecessor to the current node), checking the state of a specific edge or more advanced queries such as range queries (e.g. compute all the messages sent from a group of users to another group of users). Additionally, according to the temporal dimension of the graph we can categorize queries in two main groups:

- *Time-slice queries* (time point, timestamp queries) ask for the state of the graph at a given time instant, that is, they try to recover the state of the temporal graph at a given point in time.
- *Time-interval queries* ask for the state of the graph during a time interval, that is, during a period of time that spans many snapshots of the graph. In this kind of queries the goal is to discern whether each edge was active or not during the complete interval. Two different semantics can be applied to time-interval queries, depending on the meaning associated with "being active" during a time interval:
  - *Weak* time-interval queries ask for edges that have been active at any point within the time interval. For example, if an edge was active only at t_2 and we ask for the time interval $[0, 4]$ the edge was active according to the weak semantics.
  - *Strong* time-interval queries ask for edges that have been active during the complete time interval. For example, if we ask for the time interval $[0, 2]$ a cell that was active at $t = 0, t = 1, t = 2, t = 3$ would be active according to the strong semantics, but a cell that was active at $t = 0, t = 1, t = 3$ would not be active in the interval, because there is at least a point in the interval ($t = 2$) where it was not active.

4.1.1 Alternatives and existing proposals for representing temporal graphs

Research on temporal graphs has led to the development of multiple approaches to model time-evolving data. The simplest strategy is possibly the representation of the temporal graph as a collection of snapshots. This strategy is simple but is somehow limited in its ability to represent the temporal graph in reduced space

(since the complete state of the graph is stored in each snapshot) and makes it difficult to analyze the evolution of the graph along time, since the information of the evolution of edges is scattered among the different snapshots. However, the storage of time-evolving graphs using snapshots is still widely used in domains such as Web graphs or social networks, where very large graphs are managed and snapshots are created with low frequency.

Another simple model for the representation of temporal graph is the timestamping approach. This approach considers time as an additional dimension of the graph: a series of temporal values will be associated with each edge, which allows a representation based on this model to efficiently find all the information associated with a node or edge, querying the time dimension.

More advanced models for the representation of temporal graphs take into account the notion of *changes* in the graph, and build their models over this concept. Change-based representations are based on an initial snapshot of the complete graph and then they store only changes with respect to the initial state. This helps reducing the amount of information stored, since (ignoring the initial snapshot) information is only stored when an actual change exists in the graph. A variation of this approach stores not the changes themselves, but the events that produce these changes: this approach is called an event-based representation [Wor05, GW05].

The FVF Framework [RLK+11] combines a change-based strategy with a clustering step. In this approach, similar snapshots are grouped in clusters and only two representative graphs are kept for each cluster, corresponding to the union and the intersection of all the snapshots inside the cluster. In order to obtain a compact representation of the graph, the collection of snapshots grouped inside a cluster are encoded using the change-based approach. The authors propose two different methods that provide a tradeoff between query efficiency and space utilization. In both cases, only a representative of the cluster is stored completely. In the first method of storage, the remaining snapshots are encoded differentially with respect to the representative. However, in the second method, the sequence of snapshots in the cluster is stored differentially with respect to the previous snapshot. This second method reduces very significantly the space utilization, since the authors focus on graphs that model evolving graphs, so consecutive snapshots are expected to be very similar. On the other hand, a purely differential representation requires a reconstruction step to rebuild the original value in a snapshot, processing the differences in all the intermediate snapshots.

4.1.1.1 A representation of temporal graphs based on compact data structures

The ltg-index [dBBCR13][1] is a compact data structure based on a collection of *snapshots* and *logs of changes* to store a temporal graph in reduced space. Each *snapshot* is a simple unlabeled graph representing the complete state of the graph at a given time instant. The logs of changes, one for each node in the graph, contain the sequence of changes in the adjacency list of the node between consecutive snapshots. A parameter c manages the number of snapshots that is created, so that, if the first snapshot has e edges, a new snapshot is created after $e \times c$ changes in the log.

The ltg-index is based on the combination of several existing compact data structures to provide a very compact representation of the temporal graph and support time-slice and time-interval queries. Figure 4.1 shows a conceptual representation of the components in the ltg-index. An additional element in the figure, the *delgraph*, is required to efficiently answer reverse (predecessor) queries, since the logs store changes in successors of each node.

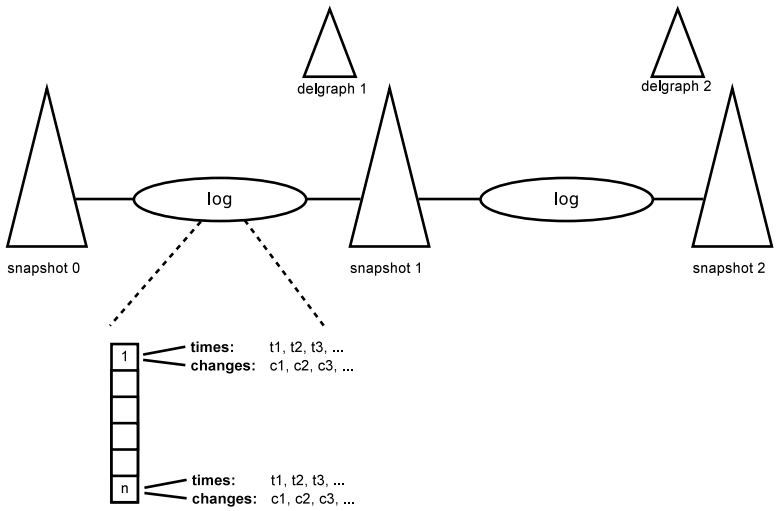


Figure 4.1: Conceptual representation of the ltg-index.

The conceptual representation depends on efficient representations for both logs and snapshots, but does not impose severe restrictions on the actual data structures used to store them, since the operations performed in both of them are relatively simple. However, the ltg-index is proposed using a specific set of data structures

[1] A preliminary version demonstrating the ltg-index was published together with other representations that are included as contributions on this thesis. The ltg-index itself was developed by our coauthor Diego Caro and is part of his ongoing PhD work related to the representation of temporal graphs

that are designed to provide the best performance. A table with n entries stores the log for each node. Each entry contains two lists: a time-ordered sequence of changes, storing the id of the successor that changed, and another sequence that stores the time instants or timestamps when each change occurred. The first sequence is encoded using ETDC to take advantage of repetitive changes, and the sequence of timestamps is encoded using PForDelta [ZHNB06] on the gaps between timestamps, since it is a fast and compact representation to store long sequences containing many small integers. The *delgraph* associated to each snapshot is a graph that stores all the removed edges within the time interval of the log. Both snapshots and delgraphs are stored using K^2-trees.

The ltg-index can easily retrieve the successors (direct neighbors) of a node at a time instant of time interval, simply finding the previous snapshot and sequentially traversing the change log of the corresponding node. For each successor in the log, it suffices to count the number of times it appears in the log to know whether it has the same state as in the previous snapshot (an even number of changes) or a different state. Hence, the successors of the node computed in the previous snapshot are completed with the changes in the log to answer the query.

The process to find the predecessors (reverse neighbors) of a node is more complicated: the predecessors of the node are computed in the previous and next snapshot, as well as in the delgraph associated to it. The union of the three lists is the list of candidate predecessors. For each of them, the algorithm for successors is used to find their actual state at the time instant or time interval of the query.

4.2 RDF graphs

The Resource Description Framework (RDF [MM04]) is a standard for the representation of information. It models information as a set of triples (S, P, O) where S (subject) is the resource being described, P (predicate) is a property of the resource and O (object) is the value of the property for the subject. RDF was originally conceived as a basis for the representation of information or metadata about documents.

In recent years, a new interpretation of the web has gained a lot of influence: the Semantic Web, or Web of Data, is a data-centric interpretation of the web. It tries to provide a framework for the exchange of semantic information on the web. The Linked Data project[2] is a project that defines a way to share information between nodes in a standard way, based on HTTP and RDF. The growth of the Linked Data perspective in recent years has made of RDF the *de-facto* standard for the representation of information in the *Web of Data*.

RDF provides a framework for the conceptual representation of information. As we said, it represents the information as a set of triples (S, P, O). These triples can

[2]http://linkeddata.org

J.R.R. Tolkien	born in	South Africa
J.R.R. Tolkien	author of	The Lord of the Rings
J.R.R. Tolkien	author of	The Silmarilion
South Africa	capital	Pretoria
South Africa	capital	Cape Town
South Africa	capital	Bloemfontein
The Lord of the Rings	author	J.R.R. Tolkien
Nelson Mandela	born in	South Africa

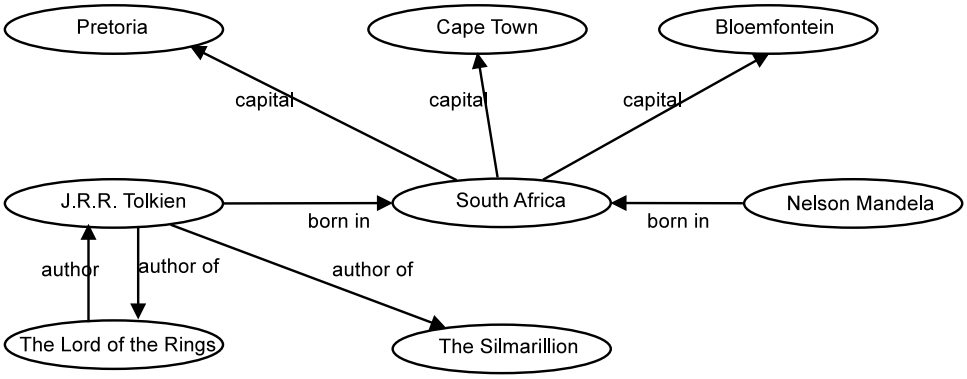


Figure 4.2: RDF representation: triples (left) and RDF graph (right).

also be seen as edges in a labeled directed graph. The vision of a set of RDF triples as a graph is called RDF graph in the original recommendation [MM04]. Figure 4.2 shows a small example of RDF representation that models some statements about J.R.R. Tolkien. For example, the first triple states that Tolkien was born in South Africa; the second one shows that Tolkien wrote The Lord of the Rings; etc. The same information shown by the triples can be seen in the labeled graph in the figure.

RDF datasets can be queried using a standard language called SPARQL [PS08]. This language is based on the concept of triple patterns: a triple pattern is a triple where any of its components can be unknown or variable. In SPARQL this is indicated by prepending the corresponding part with `?`. Different triple patterns are created simply changing the parts of the triple that are variable. The possible triple patterns that can be constructed are: (S, P, O), $(S, P, ?O)$, $(?S, P, O)$, $(S, ?P, O)$, $(?S, P, ?O)$, $(S, ?P, ?O)$, $(?S, ?P, O)$ and $(?S, ?P, ?O)$. For instance, $(S, P, ?O)$ is a triple pattern matching with all the triples with subject S and predicate P, therefore it would return the values of property P for the resource S. $(S, ?P, ?O)$ is a similar query but it contains an *unbounded predicate*: the results of this query would include all the values of *any* property P of subject S.

SPARQL is a complex language, similar to the SQL of relational databases, and

```
SELECT ?o
WHERE {
    J.R.R. Tolkien    author of    ?o
}
```

```
SELECT ?s
WHERE {
    J.R.R. Tolkien    born in    ?o
    ?s                born in    ?o
}
```

Figure 4.3: Example SPARQL queries.

it supports multiple selection clauses, ordering and grouping, but its main features are based on the triple-pattern matching and join operations, that involve merging the results of two triple patterns. For example, $(?S, P_1, O_1) \bowtie (?S, P_2, O_2)$ is a join operation between two triple patterns where the common element $?S$ is the join variable. The result of this join operation would contain the resources S whose value for property P_1 is O_1 and their value for P_2 is O_2.

Like simple triple patterns, multiple join operations can be constructed varying the variable (or unbounded) elements that appear in the join: $(?S, ?P_1, O_1) \bowtie (?S, P_2, O_2)$, $(?S, ?P_1, O_1) \bowtie (?S, ?P_2, O_2)$, etc. Independently of the number of variable elements, different join categories can be determined depending on the elements of the triple patterns that act as join variable: in *subject-subject (S-S)* joins the join variable is the subject of both triple patterns (e.g. $(?V, P_1, O_1) \bowtie (?V, P_2, O_2)$), in *subject-object (S-O)* joins the variable is the object in one pattern and the subject in the other (e.g. $(S_1, P_1, ?V) \bowtie (?V, P_2, O_2)$) and in *object-object (O-O)* joins the variable is the object in both patterns (e.g. $(S_1, P_1, ?V) \bowtie (S_2, P_2, ?V)$).

Example 4.1: Figure 4.3 shows two examples of SPARQL queries over the triple patterns in Figure 4.2. In the first query the triple pattern used shows that we are asking for the books written by J. R. R. Tolkien. The second query is a join query that asks for all the people born in the same place as J. R. R. Tolkien (this is due to the value $?O$, that is the same in both triple patterns and therefore behaves as join element).

4.2.1 Representation of RDF graphs

RDF is only a conceptual framework that does not force any physical representation of the data. The recent popularity of RDF has led to the appearance of many different proposals for the actual storage of information in RDF, known as *RDF stores*. They can be divided in two main categories: those based on a relational database and those based on specific data structures designed to adjust to the characteristics of RDF.

4.2.1.1 Relational proposals

Many of the most popular solutions for the representation of RDF are based on relational databases. In these representations, and depending on the tables created

for the storage of the triples, SPARQL queries can be easily translated into relational queries that can be easily interpreted by the database system. Virtuoso[3] stores RDF triples essentially in a table with three columns for the subject, predicate and object respectively. Other systems, such as Jena [Wil06], try to represent the information in a more structured way by categorizing the subjects and creating tables in the database that represent the common set of properties for a list of objects. Other solutions are based on the concept of *vertical partitioning*. The vertical partitioning approach builds a table for each different predicate in the RDF dataset, so the triples are partitioned according to their P value. In a relational approach, the result of vertical partitioning is typically the creation of a separate table for each different P_i, containing all the pairs (s, o) such that a triple (s, P_i, o) exists in the dataset.

4.2.1.2 Specific solutions

Moving away from relational solutions, several other proposals have been proposed for the compact representation of RDF datasets.

- Hexastore [WKB08] is based on the creation of multiple indexes over the triples. It basically builds 6 different indexes, one for each of the possible orders (S, P, O),(S, O, P),(P, S, O),(P, O, S),(O, S, P),(O, P, S). This leads to easy access to triples independently of the triple pattern used, but increases a lot the space requirements. Additionally, Hexastore is designed to work in main memory, so the space requirements limit this approach to work with smaller datasets.

- RDF-3X [NW10a] is also based on building multiple indexes to represent the set of triples. In this approach, however, the indexes are stored in compressed B^+-trees. The procedure used to build the triples is specifically designed so that the entries in the indexes can be easily compressed, and custom encoding methods are applied to reduce the space of the indexes. This reduces the space requirements, and the B^+-tree storage of the indexes allows RDF-3X to work from external memory. Even if the space requirements are still high, the ability to work from external memory in large dataset makes of RDF-3X a better indexed solution. Additionally, further work has evolved the tool to optimize update operations in the dataset [NW10b].

4.2.1.3 Representation of RDF graphs using K^2-trees

A new proposal, based on the K^2-tree, has been proposed recently for the compact representation of RDF datasets [ÁGBFMP11, ÁGBF$^+$14]. This proposal, called K^2-triples, is based on a vertical partition of the triples, and takes advantage of the K^2-tree as the basis for efficient query support.

[3] http://www.openlinksw.com/

The first step to represent an RDF dataset in K^2-triples is the *dictionary encoding*. In this step, the triples (S, P, O), where the components of the triple are URIs, are translated into triples of ids. Each URI is replaced by its id in the corresponding dictionary, so that the triples that will be actually represented are of the form (v_s, v_p, v_o), where each v is an integer value. They use 4 different groups of URIs, that may be encoded independently, containing respectively elements that are both subjects and objects, only subjects, only objects and only predicates.

After the dictionary encoding has been applied, a vertical partitioning is applied to the triples. For each different predicate, a K^2-tree is created to store all the triples corresponding to that predicate. Conceptually, each K^2-tree represents a binary matrix where the rows are all the subjects in the RDF dataset, the columns are all the objects, and the ones in the matrix are the pairs (S, O) that correspond to existing triples (S, P_i, O) for the current predicate. The matrices for all the different predicates contain the same rows and columns in the following order: the first rows/columns correspond to elements that are both subjects and objects; then, the next rows correspond to the elements of the vocabulary that are only subjects, and the next columns correspond to the elements of the vocabulary that are only objects.

Triple patterns can be answered very efficiently in K^2-triples. Each possible triple pattern is translated into row/column/range operations in one or all the K^2-trees in the dataset. For example, a pattern $(S, P_1, ?O)$ can be translated as a row query in the K^2-tree that stores the triples for predicate P_1; a pattern $(S, ?P, ?O)$ requires to compute the same row query in all the K^2-trees; a pattern $(?S, P_1, ?O)$ simply involves obtaining all the cells in the K^2-tree for predicate P_1.

Join operations can also be solved in K^2-triples. The authors propose different strategies to solve different join operations involving two triple patterns can be solved:

- *Independent evaluation* simply computes both patterns independently and then merges the results: for example, to answer a query of the form $(?S, P_1, O_1) \bowtie (?S, P_2, O_2)$ this strategy would compute the results for $(?S, P_1, O_1)$ and the results for $(?S, P_2, O_2)$ and then intersect the sequences of results obtained from the first and second queries.

- *Chain evaluation*, computes the "easiest" pattern first and then runs the second pattern limited to the results in the first one. Following the previous example, to answer a query of the form $(?S, P_1, O_1) \bowtie (?S, P_2, O_2)$ using chain evaluation we would first compute the results for $(?S, P_1, O_1)$ and then run a set of simpler queries looking for (S_i, P_2, O_2) for each S_i obtained in the first query.

- A third strategy, called *interactive evaluation*, performs a synchronized traversal of the K^2-trees involved in both patterns, filtering the results depending on the values on each side of the join. To answer a query of the

form $(?S, P_1, O_1) \bowtie (?S, P_2, O_2)$ we would execute the queries $(?S, P_1, O_1)$ and $(?S, P_2, O_2)$ in parallel. Whenever we find a 0 in one of the K^2-trees corresponding to a range of subjects S, we know that the join operation cannot yield any result for this range so we stop traversal in the second K^2-tree. In these operations the branches traversed in each K^2-tree are not necessarily the same (in the previous example, in the first K^2-tree we are looking for column O_1 and in the second one we are looking for column O_2). Additionally, even the operations required in each K^2-tree may be different. Consider for example a join $(S_1, P_1, ?V) \bowtie (?V, P_2, ?O_2)$: in the first K^2-tree we would look for all elements in the row corresponding to S_1, and in the second K^2-tree we would traverse the complete matrix. If we find a 0 for a range of values of V in the first K^2-tree we can stop traversal in all regions intersecting $?V$ in the second K^2-tree. This kind of join operation requires some computation to keep count of the mapping between nodes in the left part of the join and nodes in the right part that "cancel each other".

In K^2-triples, triple patterns or join operations with variable predicates must query all the K^2-trees for each predicate, which becomes very costly if the number of predicates is high. An enhanced version of K^2-triples, called K^2-triples$^+$, has also been proposed to speed up operations involving variable predicates [ÁGBF$^+$14]. The authors propose the creation of additional indexes that indicate, for each subject and object in the dataset, which predicates are associated with that subject/object. This allows them to limit the execution of queries with variable predicates to the predicates associated with the corresponding subject(s) or object(s). The authors create two compressed indexes SP and OP that return the list of predicates associated to a given subject or object. In practice, to obtain good compression they do not store the lists directly: instead, Directly Addressable Codes are used to statistically encode the lists and store the resulting sequence of variable-length codes providing direct access to any of them.

K^2-triples have proved to be very competitive with other state-of-the-art RDF stores in most queries involving triple patterns and simple join operations [ÁGBF$^+$14], that are the building blocks of SPARQL. However, the static nature of the K^2-tree is an important drawback for K^2-triples, as RDF datasets have a dynamic nature. The improved K^2-triples$^+$ approach is able to improve times in queries with variable predicate in datasets with a large number of predicates, at the cost of a significant increase in space and the introduction of new static indexes that are not suitable for a dynamic environment.

4.3 GIS

4.3.1 Geographic Information Systems and spatial data

A Geographic Information System (GIS) is a "computer-based information system that enables capture, modeling, storage, retrieval, sharing, manipulation, analysis, and presentation of geographically referenced data" [WD04]. Essentially, the goal of GIS is to provide support for any activity that involves any kind geographic information. Following this definition, the applications of GIS are immense. Cartography or meteorology are immediate domains of application where clearly we need to manage geographic information, but Geographic Information Systems are also used in many other domains, like traffic monitoring, disaster management, wildlife monitoring and plague control, etc.

Advances in technology have made the utilization of GIS available even to small organizations in recent years. Many countries in the world put to the disposition of the public increasingly large volumes of geographic data corresponding to topographic information, weather information, land use, etc. In addition, there has been an important effort to provide the community with standard specifications for the interoperability of these systems. The Open Geospatial Consortium[4] (OGC) is an international consortium formed by public agencies and private companies that has published many open standards for the publication of geographic information in the web. The combination of all these factors leads to a huge volume of spatial data that is accessible to the public and has many applications for small business as well as for large companies and government agencies.

4.3.1.1 Models for geographic information

The data obtained from the real world in a geographic information system needs to be conceptualized using *models* that allow us to represent the data and perform the required operations over it within a well-defined framework.

There are two main conceptual models for the representation of geographic information: *field-based models* and *object-based models*.

- In a *field-based model* the space is seen as a surface over which different *features* or *spatial attributes* are represented. Features in a field-based model can be seen as a mapping from coordinates in space to a value for the coordinate, that is, the space itself is considered as an entity that can be described by attributes. Examples of information that would suit a field-based model are temperature, elevation, pressure, etc., that can be measured at any point of space. The typical operations of interest in field-based models involve computing the maximum, minimum or average value in a zone, computing slopes, etc.

[4] http://www.opengeospatial.org/

- In an *object-based model*, the space is simply a frame over which a set of discrete objects appear. Object-based models describe a set of objects that have spatial attributes, and these objects are themselves perfectly identifiable. The space, however, is now simply a frame of reference. Object-based models describe different types of objects, such as points, curves and regions, and the usual operations are based on the topological relationships between objects (check if two objects intersect, one of them is contained in the other, etc.).

The two conceptual models are somehow opposed, in the sense that field-based models understand the information as a set of "attributes of the space", while object-based models identify specific objects that have a given geometry and a location in the space.

The conceptual representation of geographic information does not enforce a particular way of representing the elements defined. The logical models determine the way of translating the abstractions of the conceptual models into an actual representation in a system. There are two main logical models for the representation of geographic information systems: the *vector* model and the family of tessellation models, of which the *raster* model is the most important representative.

In the vector model, geographic information is represented by a collection of points and segments. A point is represented by its coordinates and a segment is represented by its endpoints. Abstractions such as curves and surfaces are approximated using multiple connected segments. The representation of objects using a vector model requires a previous step to discretize the actual shapes in order to obtain an approximation that can be stored in the system. Figure 4.4 (left) shows an example of objects represented using a vector model.

Tessellation models are based on a decomposition of the space in a collection of non-overlapping polygons. Tessellations can be regular (the decomposition uses regular polygons) or irregular. The most usual irregular tessellation is the Triangulated irregular network (TIN). Regular tessellations can use triangles or hexagons, but the most used tessellation is the grid or *raster* model. In a raster model, the space is partitioned in a grid of disjoint cells. Each cell stores a set of values, typically the value of the feature or features that are being represented. Figure 4.4 (right) shows an example of application of the raster model, compared with the equivalent representation in the vector model.

In general, any kind of data can be represented using any of the logical models. However, spatial features such as temperature or elevation, that fit better with a field-based model, are usually represented using a raster model. On the other hand, domains involving discrete objects and following an object-based conceptual model are usually represented using the vector model. Additionally, there are several advantages and drawbacks of both methods that should be considered. In general, information represented using the vector model tends to require less space, provides efficient support for spatial operations such as rotations and supports the definition of explicit relationships between geographic objects. On the other hand,

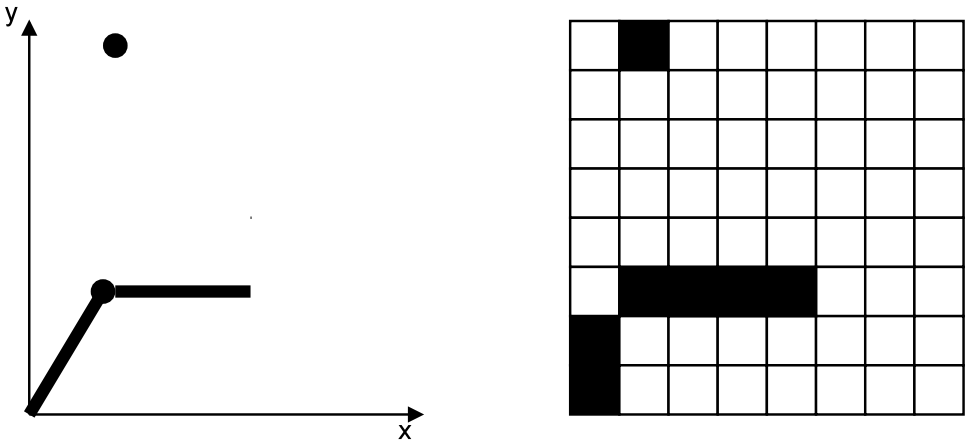


Figure 4.4: Graphical comparison of the vector model (left) and the raster model (right).

the raster model is much simpler to implement and simple algorithms for displaying and filtering can be used over the data. Additionally, a lot of information coming from real-time measurements is originally represented using the raster model. We will devote the remaining part of this chapter to give an overview of the state-of-the-art representations of geographic information under the raster model and the vector model, focusing especially in representations of raster data.

4.3.2 Representation of raster data

As explained in the previous section, the representation of geographic information usually requires a choice of representation model depending on the characteristics of the data and the operations required. The vector model is the most used model, as it is of application in most contexts and provides interesting applications to work with well-defined discrete objects. However, the representation of spatial features that are continuous in space is still based on the raster model. Geographic Information Systems usually provide some support for the direct storage and manipulation of raster data, and many specific data structures have been designed for storing raster dataset.

Conceptually, a raster is simply a matrix or grid containing different values of a spatial feature. However, in practice we may distinguish several kinds of rasters according to the nature of the data:

- The simplest raster is a binary matrix containing the existence or not of a spatial feature at each possible cell of the matrix. A binary raster can be used to determine, for instance, cloud cover or the evolution of fires, plagues, etc.

- A general raster contains *values* of a spatial feature. Each cell of the raster will store a value with a predefined precision, thus we may differentiate a raster depending on the data type required for storing each value. Raster representations may require bytes, 32-bit integers or even double-precision floating-point values to store each value.

- In GIS a general raster is usually understood as a grid that may store a collection of spatial features for the covered space. Each *layer* of the raster corresponds to a given feature. Therefore, a general raster in this case may be seen as a collection of rasters, each storing the value of a single layer for the same spatial region.

The raster model of representation, although applied here to the regular decomposition of geographic information, is also extended to the representation of images in general. Many image representations are based on a raster decomposition of the image and the raster representation of its values. Furthermore, one of the simplest but most used application of raster datasets is efficient visualization of data. Because of all this, the strategies for representing and compressing raster data are also of application to represent general images, and many well-known image file formats are widely used to store raster datasets in small space. We will refer in general to raster images as a matrix of values corresponding to the raster decomposition of spatial information.

Raster datasets in GIS usually contain spatial data in addition to the matrix of values, including for instance the actual size of the raster, the coordinates it covers and the size of the cell. Furthermore, it is usual in raster datasets to have cells where the value is not known, cannot be computed or was simply not measured (for example, in a land use map we would not have values for cells in the sea, in many raster datasets we may not have data outside the boundaries of our own country, etc.). These cells are usually stored with a special NODATA value that has to be managed in most queries.

The set of operations that are usually applied to raster data are usually based on *map algebras*: given a set of raster datasets, the operations are designed to obtain results or compute new raster datasets based on the values in one or more of those raster representations. We can distinguish the following categories of operations over a raster dataset:

- *Local* operations are applied to a single cell of the raster(s) to obtain or compute a result. Different operators can be used, for example, to compute whether the value in a raster is greater/smaller than a given value, or to compute the maximum/minimum/average/sum of the values of the same cell in multiple rasters.

- *Focal* operations are applied to a cell and its *neighborhood* (the set of adjacent cells) to compute the new value. A typical example of this is the computation

of slopes from elevation values, where the slope in a cell depends on the elevation in the current cell and its adjacent cells.

- *Zonal* operations involve cells that share the same *zone* or category, usually determined by the cell value. For example, in a map representing land usage a zonal operation may want to retrieve only the cells corresponding to pasture zones in order to perform some computation (simply compute the total area of pastures, or select only the pasture regions in a different raster dataset).
- *Global* operations may involve all the cells in the raster dataset to compute the output.
- Any other application-specific operation that involves a combination of multiple of the previous operations.

A typical requirement in spatial queries is to restrict the query to a subregion of the space covered by the raster. For example, consider a land usage map for a country stored in a raster dataset. An operation that tries to obtain pasture regions only in a region of the country requires to perform the zonal operation restricted to the *spatial window* indicated in the query. Window queries are one of the most basic and widely used queries over raster datasets.

In many cases, the processing of raster data is performed sequentially in a pass over the complete representation. This requires simple data structures to store the information, and allows efficient compression since access is sequential, but it is highly ineffective considering the kind of queries that are usually performed in spatial data: a local operation that asks for the value of a single cell should not need to access the complete raster dataset. In order to support spatial queries efficiently, a representation of raster data should provide at least some level of *spatial indexing* of the data, that is, efficient access to specific regions of the space without the need to process the complete raster. Compression and query efficiency must be taken into account in a space/time tradeoff to provide an efficient representation of raster data.

4.3.3 Common strategies for compressing raster data

The representation of huge volumes of raster data requires a strategy for *compressing* the raster dataset. An uncompressed raster, that stores the value of each cell directly, will require a large amount of space to represent large datasets. For example, a $50,000 \times 50,000$ raster image that stores a 32-bit integer for each cell would require roughly 10 GB of space to be stored. Even simple operations over such a raster (for instance, sequential processing) would require reading 10 GB, which would slow down significantly all operations involving the raster.

Most of the compression techniques used for raster data are based on well-known properties that are expected in most raster datasets corresponding to different

kinds of spatial information. The most important of these properties is the spatial continuity of the values in the raster. Raster datasets represent the value of spatial attributes that tend to be clustered: elevation, land usage, temperature, extension of a plague, habitat of animals, oil spills... All of them tend to contain well-defined regions of identical or at least similar values. The raster representation of such information will be a matrix with many uniform regions that should be efficiently compressed.

The clusterization of close values in raster datasets leads to approaches that tackle raster compression relying on this property. The *tiling* technique consists of partitioning the complete raster into a collection of equal-sized tiles. This reduces the problem to the representation of smaller matrices, that are usually compressed independently. The locality of values in most raster datasets makes each tile easier to compress using simple techniques than the complete matrix, since the number of different values and the differences between values within a tile will be much smaller than on the overall matrix. Furthermore, the tiling approach can also provide efficient access to individual tiles of the raster for partial decompression in window queries, even if each tile must be completely decoded.

The simplest techniques for compressing raster data combine the tiling technique with simple and effective sequential compression algorithms that are applied to the individual tiles, to take advantage of the regularities in each tile. This approach reduces significantly the size of the raster at the cost of losing efficient direct access to any position in the raster (direct access is restricted to the tile, and the tile must be decompressed sequentially to access any cell in it). However, when combined with the tiling approach, the use of sequential compression usually provides very good space results while providing some level of spatial indexing in the raster. Many different compression algorithms can be used to compress the values in a raster (recall Section 2.1): run-length encoding is a simple and fast approach; LZW-compression takes advantage of many regularities to compress even further while providing good decompression times; arithmetic encoding is used in some representations to obtain very good compression at the cost of higher decompression times. Again, the compression method used provides a tradeoff between compression and query efficiency. The combination of tiling and efficient sequential compression of each tile provides a tradeoff between efficient random access to regions of the matrix and space requirements of the compressed dataset.

The current representations of raster data can be coarsely divided in the following categories:

- *File-based representations.* This approach covers a large collection of file formats that are designed for the representation of spatial raster data or are designed for the general representation of images, arrays, etc. but can be used for the storage and compression of raster data by a GIS. In these representations the raster data is simply stored in a (possibly compressed) file and stored on disk. Many of these formats are thought mainly for the

compressed storage of raster images and only support a sequential access to the data, while other representations follow strategies like the tiling approach to enhance compression and support local access to tiles. Examples of file-based representations of raster data include GeoTIFF and general image formats like JPEG or PNG.

- *Relational representations.* This is the approach followed by many geographic information systems for the representation of raster data. A set of database tables is prepared for the storage of raster data. Tiling approaches are used to provide some level of spatial indexing and database representations are used to store the tiles independently. Most commercial databases, including Oracle and PostgreSQL, contain modules for the management of spatial information that support raster data.

- *Fully-indexed representations.* In this group of representations we include specifically designed data structures, or *spatial access methods*, that efficiently support spatial access to the data and at the same time are able to compress the data. The families of data structures based on a recursive decomposition of the space are probably the most successful representatives of this group, and are widely used for the representation of spatial data and also in other fields related to image representation and processing. The family of *quadtree* data structures is widely used for the representation of spatial information. A similar family of data structures, based on a binary tree structure called *bintree*, is based on a binary decomposition of the space and has also been used for the compact representation of raster images.

In the rest of this section we will briefly describe some file formats and relational representations for raster data, that are mostly based on the general techniques already explained: efficient compression methods and *tiling* to optimize compression and provide random access to regions of the image. The quadtree-based and bintree-based data structures share many similarities with our proposals in this thesis and their structure and applications to the representation of raster images will be studied in detail in Sections 4.3.4 to 4.3.6.

4.3.3.1 File formats

GeoTIFF[5] is a standard proposed for enriching a Tagged Image File Format (TIFF) image file with specific annotations regarding the geographic characteristics of the image, such as the coordinate system and projection used. The TIFF format is very flexible and many extensions to it have been proposed since its first specification. It supports binary, grayscale and color images with different depths. It allows a decomposition of the image in *strips* (one or more rows) or *tiles* that are compressed

[5] http://geotiff.osgeo.org

```
ncols         4
nrows         4
xllcorner     0.0
yllcorner     0.0
cellsize      30.0
NODATA_value  -999
4 4 5 6
3 3 4 4
1 2 3 4
-999 -999 1 4
```

Figure 4.5: An example of ArcInfo ASCII file.

independently from each other. Different compression methods are supported, including run-length encoding, LZW or DEFLATE[6] (a combination of LZ77 and Huffman coding). Its extension capabilities and the ability to even add custom headers to the format make of TIFF a perfect candidate for the extension to geographic images. Support for accessing and manipulating GeoTIFF images is provided by existing libraries such as libgeotiff[7], that provides elementary spatial operations on top of the common libtiff library for manipulating TIFF images.

NetCDF[8] is a standard of the Open Geospatial Consortium (OGC) for the encoding of geospatial data. It provides a representation of data as sets of arrays and can represent efficiently different kinds of multidimensional data. It provides metadata that can store the spatial and temporal properties of the stored data. The data itself is organized in arrays for the different dimensions, including attributes for the specification of relevant properties. To obtain compression, NetCDF files can use DEFLATE compression to represent the data.

Esri grid is a proprietary binary format developed by Esri as the file format for the storage of raster data in ArcGIS, a proprietary system. It decomposes the raster in rectangular tiles. The tiles are stored independently either in plain or compressed form (run-length encoded). A plain variant of the ArcInfo grid format using ASCII also exists for the representation of raster data. It is called ArcInfo ASCII Grid and provides a simple representation of raster data based on plain text storage of the matrix of values. It only defines the essential information of the raster image and encodes it in plain text, as shown in Figure 4.5. It is not a practical representation due to its high space requirements but is used as an intermediate format because of its simplicity.

Many common image file formats are used frequently to store raster data, separating the spatial information (that can be stored as metadata) from the

[6] http://www.ietf.org/rfc/rfc1951.txt
[7] http://download.osgeo.org/geotiff/libgeotiff/
[8] http://www.opengeospatial.org/standards/netcdf

actual raster data, that can be easily compressed using general techniques of image compression. JPEG2000[9] is a standard of the Joint Photographic Experts Group, proposed as an evolution of the JPEG standard[10]. It is designed for the compression of general images and can be used for the storage of raster data. JPEG2000 supports both lossy and lossless compression and obtains better compression than the previous JPEG standard. *PNG* (Portable Network Graphics) is another example of file format for the storage of general images. It provides good compression of the resulting file but does not provide any kind of random access. However its wide use and good compression make it a candidate for the transmission and visualization of raster images. Other file formats that are sometimes used for the storage of raster data are the Graphics Interchange Format (GIF) or JPEG. In all of these formats we can see some limitations: access to the raster is limited and there is no support for geographic information.

4.3.3.2 Relational solutions

Oracle Spatial is an extension of the Oracle database management system[11] for the management of geographic information. It provides a GeoRaster data type that is used for the storage of raster data. In this representation, a GeoRaster object in the database stores the basic information of the raster (the dimensionality, whether it contains multiple layers, its object identifier within the database) and a pointer to a table T that stores it and some metadata. The raster is partitioned in tiles, corresponding to *blocks*, and each tile is placed as a BLOB (custom binary object) in a row of T, together with its MBR and its tile position. Database indexes allow efficient access to each tile to provide support for window queries.

Similar representations are used in other database management systems: *PostGIS*, the geographic extension of PostgreSQL, provides utilities for importing and exporting raster files into a database schema based in tiling the source raster and storing each tile in a different row.

4.3.4 Representation of binary raster images

Binary images are the simplest form of raster, where a single feature is collected and we only need to store whether it exists in a given region or not. Despite the simplicity of the raster, binary images are used in many GIS applications to represent region features (for example, extension of plagues or oil spills) and can be combined with other raster images to obtain more complex results. In this section we present two families of representations for binary images that are widely used: *quadtrees* and *bintrees*. Both structures are very similar conceptually and the different variants also share similarities. Both quadtrees and bintrees and all their

[9] http://www.jpeg.org/jpeg2000/
[10] http://www.jpeg.org/jpeg
[11] http://www.oracle.com/

variants achieve good compression of binary raster images and efficiently support different operations depending on the variant, including random access to regions in the image, image manipulations and set-theoretical operations that compute new images from a logical combination of one or more input images.

4.3.4.1 Quadtrees

The term "quadtree", first used by Finkel and Bentley [FB74], is generalized nowadays to describe a class of hierarchical data structures that are based on the recursive decomposition of the space into smaller blocks (typically in quadrants) until the data in each block satisfy some condition. Quadtrees have been widely used in different domains due to their simplicity and ability to represent efficiently clustered data. Particularly, quadtree representations are widely used for the representation of geographic information (either for the representation of raster data or to store points in an object-based model) and for the general representation of images (we can consider a raster dataset as an image where each cell corresponds to a pixel of the image).

Conceptually a quadtree is a tree data structure in which each node represents a region of the space. The root node covers the complete space. All internal nodes have 4 children, whose regions correspond to a partition of the parent's region.

Many different variations of quadtrees have been proposed to represent points, lines, polygons or regions. The rules for decomposition may follow fixed rules or be dependant on the input data. Different decomposition rules lead to a regular partition of the space in equal-sized parts (following the embedding space hierarchy) or an irregular decomposition (following the data space hierarchy). The number of times the decomposition process is applied (known as the *resolution* of the raster) may also be predefined or depend on the actual input).

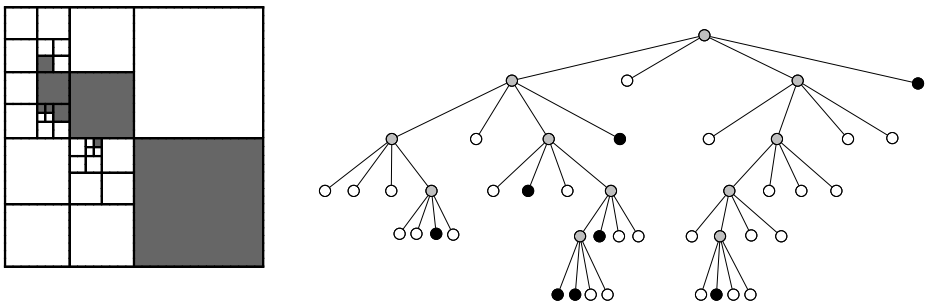
An extensive explanation of many quadtree variants can be found in [Sam90b]. We will focus on the quadtree variants used for the representation of region data, particularly in the region quadtree and the linear region quadtree.

Region quadtree

The region quadtree [Sam90b], also named Q-tree [KD76], is one of the most popular data structures of the quadtree family. The popularity of this structure is such that the general term "quadtree" is used frequently to refer to region quadtrees. Further references to the term "quadtree", unless otherwise specified, refer to the region quadtree.

The region quadtree, as its name states, is designed to store region data, and has been extensively used to store and query binary images in image processing and geographic information systems. Consider a binary matrix of size $n \times n$, where n is a power of two, that represents a binary image. Each cell in the binary matrix represents a *pixel* in the binary image. A cell with value 1 represents a *black pixel* in the binary image and a cell with value 0 represents a *white pixel*. The

region quadtree uses a fixed-size decomposition to partition the image. If the image is not *uniform* (all cells are 0 or all cells are 1), it is divided in four quadrants. The decomposition process continues recursively until all the obtained blocks are uniform. Figure 4.6a shows an example of binary matrix and the decomposition process followed by the region quadtree.



(a) Binary matrix **(b)** Region quadtree

Figure 4.6: A binary matrix decomposition and the resulting region quadtree.

The blocks resulting of the decomposition are stored in a 4-ary tree. The root of the tree corresponds to the complete matrix. Each child of a node represents one of its subquadrants (the children are usually labeled NW, NE, SW, SE to identify the corresponding quadrant). All internal nodes are said to be *gray nodes*. Leaf nodes are called *black nodes* or *white nodes* depending on the value of their associated block. Figure 4.6b shows the region quadtree generated for the binary matrix in Figure 4.6a. Notice that large regions of black or white pixels are represented with a single node in the tree, and only regions that contain both black and white pixels need to be decomposed again. This property is the key of the compression achieved by quadtree representations.

The decomposition process followed to build a region quadtree can be easily explained using a top-down recursive approach:

- The quadtree for a matrix full of zeros (completely white image) is a single *white node*.
- The quadtree for a matrix full of ones (completely black image) is a single *black node*.
- The quadtree for a matrix with ones and zeros is a *gray node* with four children. The children will be the quadtree representations of the 4 quadrants of the current matrix.

Despite this definition, the quadtree construction is usually performed in a bottom-up fashion. Additional details on the construction algorithms used for quadtrees can be found in [Sam90b].

The region quadtree can represent binary images in very small space depending on the distribution of black and white pixels. In general, binary images with large regions can be efficiently represented using a region quadtree, as many uniform square regions will be found during the decomposition process.

Region quadtrees efficiently support many simple queries, such as retrieving the value of a single pixel, thanks to their fixed-size space decomposition. The process to perform a search in a region quadtree involves a top-down traversal of the tree. At each level of the tree, we must check all the children of the current node that intersect with our query: if we are looking for a cell we must check the quadrant that contains the cell, if we are looking for a large region we must check all the quadrants that intersect with the region. The selection of the appropriate children is very simple because all the nodes at the same level represent equal-sized square regions. Additionally, some transformations are specially easy to perform in a region quadtree: 90° rotations are obtained applying a permutation to the children of each node; scaling by a power of two just requires a change in resolution. Set operations are also very simple to implement in a region quadtree: union or intersection of two binary images can be performed by a synchronized traversal of their quadtree representations, determining the color of the union/intersection quadtree from the color of the equivalent branch in each of the source quadtrees. A complete explanation of the supported operations and additional details about them can be found in [Sam84].

The original representation of the region quadtree is an in-memory tree. Each internal (gray) node stores four pointers to its children and a pointer to its parent if desired. Leaf nodes store a flag that determines their color. This in-memory representation provides easy navigation over the tree and facilitates some operations, but is clearly not very space-efficient. A lot of space is devoted to store the internal nodes of the tree and their pointers, hence adding a lot of overhead to the overall representation.

Linear quadtree

The linear (region) quadtree [Gar82] (linear quadtree from now on) is a secondary memory implementation of the region quadtree. Linear quadtrees try to optimize the representation of the data stored by following two main principles:

- The representation does not need to store a complete tree representation. In a linear quadtree, only the black nodes of the conceptual tree are represented.
- Each (black) node is represented in a format that implicitly determines the position and size or the corresponding black block.

Hence, a linear quadtree stores only a sequence of codes that identify the black

blocks of the image. The sequence of codes can be easily stored in secondary memory using a B-tree or any of its variations (usually a B^+-tree).

In order to represent a block, the usual approach is to use a sequence of "location codes" that determine the path in the tree that leads to the node corresponding to that block. The fixed length (FL) and the fixed length-depth (FD) representations, that have fixed code length, are the most popular representations for the location codes. Variable length representations can also be used to reduce space requirements at the cost of increased complexity in handling the sequence of codes.

- The FL encoding, proposed originally by Gargantini [Gar82] represents each block using a base 5 number with exactly h digits, where $h = \log n$ is the height of the conceptual quadtree. Each digit of the number is either a "location code" that indicates the quadrant corresponding to the block (we will use 0, 1, 2, 3 to represent the NW, NE, SW and SE quadrants respectively) or a "don't care" code (we will use the `X` symbol) that indicates that the block belonged to an upper level and covers all the quadrants in this level. Thus, this fixed-length code suffices to know the position and size of the corresponding block.

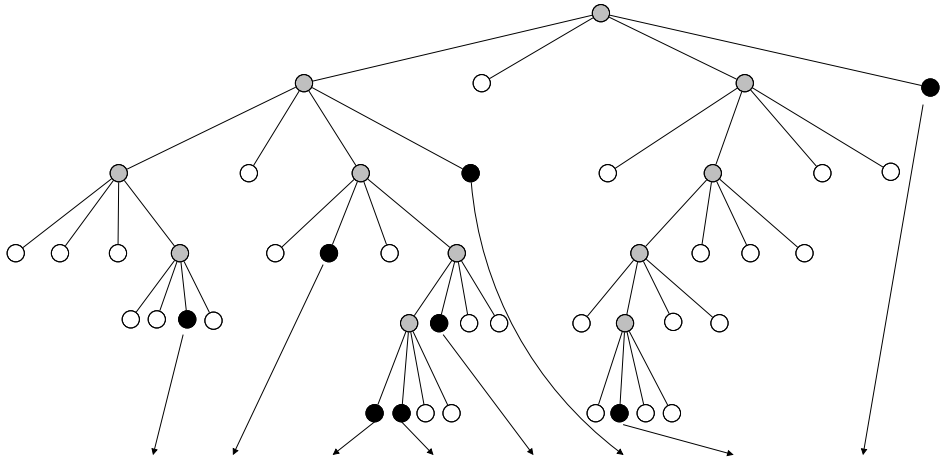
- The FD encoding uses a pair of numbers to represent each block. The first number is a sequence of h base 4 digits that indicates the position of the block using the locations codes 0,1,2,3 as in the FL encoding. To represent the node level, and therefore the block size, a second number is used. This second number requires $\lceil log\ h \rceil$ bits to store the depth of the node in the quadtree.

- The variable length encoding (VL) uses base 5 digits to represent each node, where the quadrants are represented by the digits 1, 2, 3 and 4. Each node is represented using only as many digits as its depth. The digits are chosen in a way that allows efficient computation of the depth of a node. Additional details on this and the other linear representations can be found in [Sam90a].

Figure 4.7 shows the linear quadtree codes generated for the example in Figure 4.6. Notice that the resulting sequence, independently of the actual encoding used, is built in a preorder traversal of the tree. The resulting sorted sequence of quadcodes can be easily placed in a B-tree or B^+-tree data structure and stored in secondary memory. Notice also that the traversal of the tree already generates a sequence of sorted codes.

The linear quadtree representation has some desirable properties that make it very efficient. First, thanks to representing only black nodes it can obtain important savings in space. Thanks to avoiding the use of pointers it is suitable for external memory.

Algorithms The linear quadtree efficiently supports many query operations. A simple query that involves retrieving the value of a cell can be performed in a single search in the B-tree data structure. The procedure to search a value in a linear quadtree starts by building the corresponding quadcode for the cell we are



FL	0032X	021XX	02300	02301	0231X	03XXX	21011	3XXXX
FD	(0032,4)	(021,3)	(02300,5)	(02301,5)	(0231,4)	(03,2)	(21011,5)	(3,1)

Figure 4.7: Linear quadtree location codes for the example in Figure 4.6.

looking for. Since the quadcodes for all black nodes are stored in the B-tree in order, we can search in the B-tree the corresponding code in a single top-down traversal, performing the usual binary search in each node. If the exact quadcode for the cell we are looking for is found, we can return immediately. If the quadcode is not found, we need to look at the quadcode that is located in the position where the one we are looking for would be placed. The quadcode we find could be a quadcode that contains the query cell (hence the cell is set to 1) or any other quadcode (hence the cell is set to 0). The ordering of the quadcodes guarantees that if the previous quadcode does not contain the query cell no other node in the linear quadtree representation will.

The procedure to perform window queries in a linear quadtree is quite similar to the single-cell query. The usual approach to solve a window query is to decompose the query window in a collection of *maximal blocks*, that is, the largest square blocks that would be generated if the window query was decomposed following the quadtree strategy. This decomposition returns a number of blocks proportional to the sides of the window and can be computed very efficiently. For each of the quadblocks the same procedure explained for single-cell queries is used: the quadcode is searched in the B-tree to find it or one of its ancestors, determining for each of them its value in the quadtree representation.

Compact quadtree representations for binary images

The linear quadtree is a very efficient representation of raster images that supports a wide range of operations. However, other quadtree representations

have been proposed to represent binary images that require significantly less space. These representations, like the linear quadtree, are based on encoding the nodes of the quadtree in a sequence that can be efficiently traversed. The different proposals are mainly designed for the compact storage of the image and usually support image manipulation operations such as union and intersection of multiple images, but are more limited than the linear quadtree (or the simple pointer-based quadtree) to access random regions of the image without processing the compressed representation.

Several proposals of compact quadtrees follow a representation scheme that stores the nodes of the quadtree following a breadth-first traversal instead of the usual depth-first traversal, hence reducing the average space required to encode each node in the quadtree.

The *Fixed Binary Linear Quadtree* (FBLQ [CC94]) distinguishes two different kinds of codes: "position" codes serve to indicate the position of the nodes, and "color" codes are used to know the color of a node. The FBLQ assigns a position code of 1 bit to each node (1 if the node is internal/gray, 0 if it is a leaf) and a color code of 1 bit to each leaf (0 for white leaves, 1 for black leaves). Therefore, each internal node in the quadtree is represented using a single bit and each leaf node requires two bits. An exception is the last level of the tree, where only color codes are used. Codes are stored together for groups of siblings: for each four siblings, their 4 position codes are stored and then the color codes of the nodes that required them are placed immediately afterwards. The bit placement is shown in Figure 4.8: the complete FBLQ representation is shown at the bottom, and the codes used for the nodes are indicated over the tree to the left of each node (the first number is the position code, the color code follows between parentheses).

The *Constant Bit-Length Linear Quadtree* (CBLQ [Lin97b]) uses a different code assignment, distinguishing between white nodes, black nodes and two types of internal nodes: internal nodes whose children are all leaves and any other internal nodes. Each node is then assigned a base-4 number: white nodes are encoded with a 0, black nodes with a 1, internal nodes with a 2 and internal nodes whose children are all leaves with a 3. Figure 4.8 shows the codes assigned to the nodes of a quadtree (to the right of each node) and the final sequence in the bottom.

These representations based on the breadth-first traversal of the tree are in general less efficient than depth-first representations such as the linear quadtree, and they do not support advanced queries. Hence, these representations can be seen as an alternative for the compact storage or transmission of a binary image, but are usually replaced by other approaches in order to perform queries or other operations. Because of this, some research effort has been devoted to the efficient transformation between depth-first and breadth-first linear quadtree representations [Che02].

Some of the compact quadtree representations that follow a depth-first traversal of the tree are described next:

- The *D*F-expression [KE80] encodes all the nodes in the quadtree, following

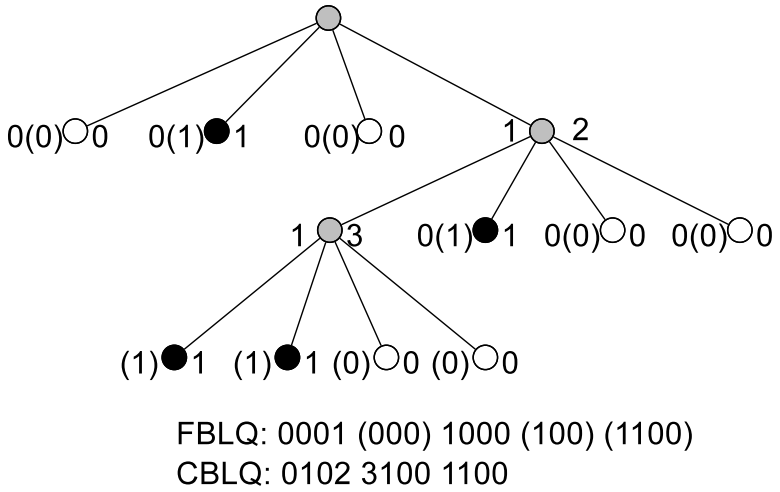


Figure 4.8: Examples of breadth-first quadtree codes.

a depth-first traversal, using three different symbols: "(" encodes an internal node, "0" encodes a white leaf, and "1" encodes a black leaf. Different methods can be used to represent the actual codes. The authors propose 3 different encodings for the symbols: *i)* using 2-bit codes for all the symbols, *ii)* encoding "0" with 1 bit and the other symbols with 2 bits, and *iii)* encoding "(" using 2 bits, "0" using a single bit 0 and "1" using a single bit 1 if it is in the last level or two bits 10 if it is in the upper levels. Figure 4.9 shows the DF-expression for a simple tree.

- The *Compact Improved Quadtree* (Compact-IQ [YCT00]) encodes only the internal nodes of the quadtree, using a representation based on the type of all their children. Considering node type 0 for white nodes, 1 for black nodes and 2 for gray nodes, each internal node is encoded using the formula $C = \sum_{i=0}^{3} C_i \times 3^i$, that is, the types of its four children are considered as digits in a base-3 number that is converted to decimal. The basic representation, called simply "Improved Quadtree", stores this values in a sequence. Figure 4.9 shows an example of this representation, including the computation of the values for the internal nodes and the final sequence. The Compact-IQ is able to further compress the sequence of integers using statistical properties of the nodes. A dictionary is stored that assigns a variable-length code to each number depending on its level, so numbers that are statistically more probable to appear in a given level are assigned shorter codewords. The dictionary uses the *random pixel model* to estimate the probabilities of each number at each level, and uses different Huffman codes for each level of the tree. The final

Compact-IQ representation is formed by the compressed sequence of variable-length codes plus the representation of the vocabulary.

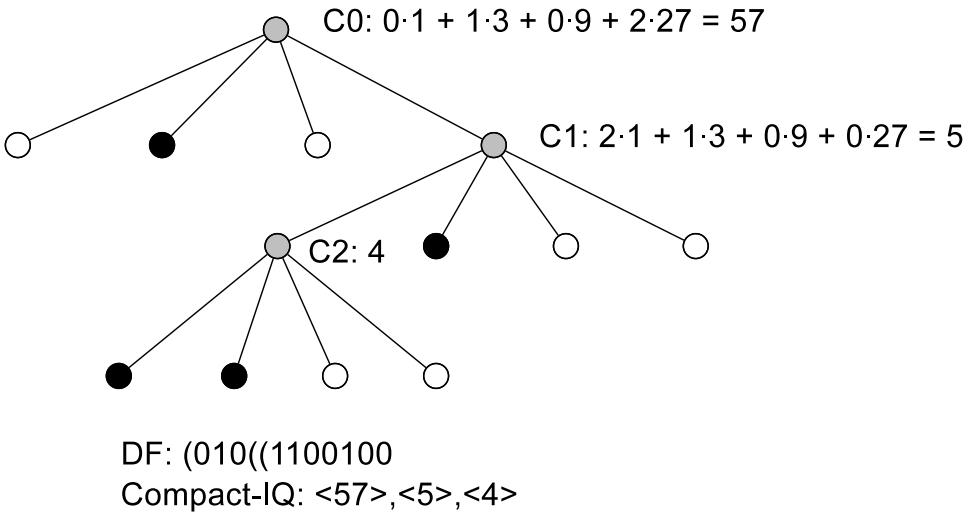


Figure 4.9: Examples of depth-first quadtree codes.

4.3.4.2 Binary tree representations of binary images

The *bintree* is a data structure conceptually very similar to the quadtree. A bintree, like a quadtree, is based on a fixed decomposition of the space. Given a binary matrix like the one shown in Figure 4.10, a bintree partitions the image in half at each decomposition step. Following the usual notation in quadtrees, a bintree first partitions the image in two rectangular sub-images W and E (west and east), and then each of the subimages in (now square) subimages N (north) and S (south). Figure 4.10 shows the bintree decomposition of a small binary image. As we can see, the decomposition is first in rectangular subimages for the east and west half of the image and then each rectangular subimage (if it is not uniform) is decomposed again to obtain new square subregions. Leaf nodes in odd levels of decomposition correspond to rectangular subimages (for example, node `d` is the rectangular block at coordinates $(0, 0)$), while nodes at even levels of decomposition are associated with square regions.

The bintree, like the region quadtree, can be implemented as a pointer-based structure. Algorithms for performing searches in a bintree are very similar to those of a linear quadtree: at each step in the top-down traversal of the tree, a simple computation can be used to determine which branches need to be traversed to find

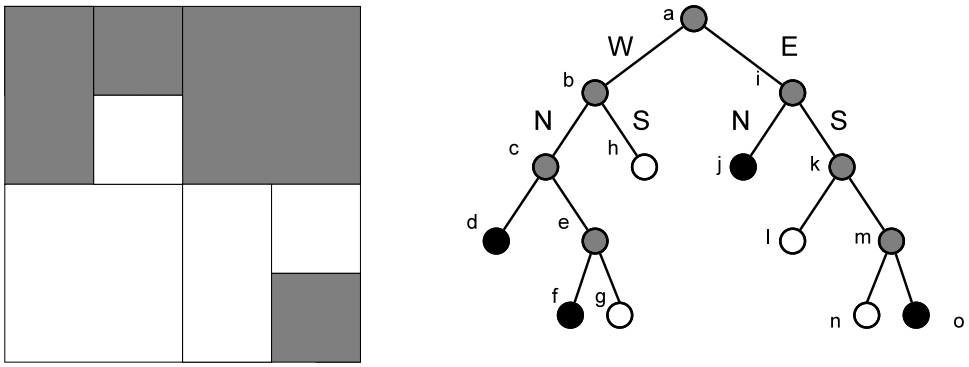


Figure 4.10: Bintree decomposition of a binary image.

a specific region. The size and position of the region covered by a black (white) node in a bintree are implicitly stored in the path followed from the root to the leaf node. Theoretical and experimental comparisons show that in general a bintree representation of a binary image should require less space than the equivalent quadtree representation [SJH93], although the space requirements may be more dependent on the actual representation chosen for each data structure than on the choice between quadtrees and bintrees.

Compact bintree representations for binary images

Compact representations of a bintree have also been proposed following the same strategies used in different quadtree representations:

The **Interpolation-Based Bintree** [OY92] or IBB represents the bintree as a sequence of encoded leaf nodes, following a strategy similar to the encoding used in the linear quadtree. In an IBB each black node is represented by a *bincode* that encodes its coordinates and depth. Given a $2^N \times 2^N$ image, the bincode of a black node at coordinates (x, y) and found at depth d is computed interleaving the bits of x, y and a representation of d. For example, consider the bincode corresponding to the square located at $(2, 0)$ in the image of Figure 4.10. Its associated node in the bintree is node `j`, and can be found at depth 2 following the path E-N in the tree. To assign it a bincode, we consider the binary representation of $x = (10)_2$ and $y = (00)_2$, and the bit sequence $s = 1100$, that is an N-bit sequence of the form $1 \ldots^{d} \ldots 10 \ldots^{2N-d} \ldots 0$ (the d first bits are 1 and the remaining bits are 0). The bincode for the node in our example is formed interleaving bits of x, y and s: first we write the leftmost bit of x, then the leftmost bit of s, then the leftmost bit of y and then the next bit of s. We repeat the process to obtain a final bincode $C = x_{n-1}s_{2n-1}y_{n-1}s_{2n-2}x_{n-2}s_{2n-3} \ldots x_0 s_1 y_0 s_0$, in our case $C = 11010000$. All this formulation is translated visually in a path traversed in the bintree to reach the node: the bincode of node `j` contains the path followed to reach the node (`10`, for a

sequence of "right", "left" child) interleaved with 1s. Once the path is complete, the code is simply filled with 0s. The complete sequence of bincodes for the bintree in Figure 4.10 would be: <01010100><01011101><11010000><11111111>. The actual representation of the bincodes is a sequence of integer numbers corresponding to the binary sequences.

The IBB, or bincode representation, efficiently supports different operations over the stored image. The strategy followed for the code assignment makes easy searching for a specific bincode: to check for a region, we generate its bincode and binary search in the IBB (i.e. in the sorted sequence of bincodes). If the bincode is found the region is black. If the bincode is not found, we can still know whether the bincode is in a black region thanks to the ordering of bincodes. From the position where the searched bincode should be, we check the previous bincode: if it corresponds to a region containing the current bincode, the bincode is in a black region; otherwise, the bincode cannot be in a black region. Additionally, the IBB efficiently supports other operations such as the intersection or union of binary images, computing the complement of a binary image or finding the neighbors of a given node [CCY96].

Logicodes and *restricted logicodes* [WC97] are based on a reinterpretation of bincodes as logical operations. Logicodes build a logical expression that represents each bincode, so the original image can be seen as a sum of the expressions for each individual code. The goal of logicodes is to minimize the size of the sum of all the individual codes.

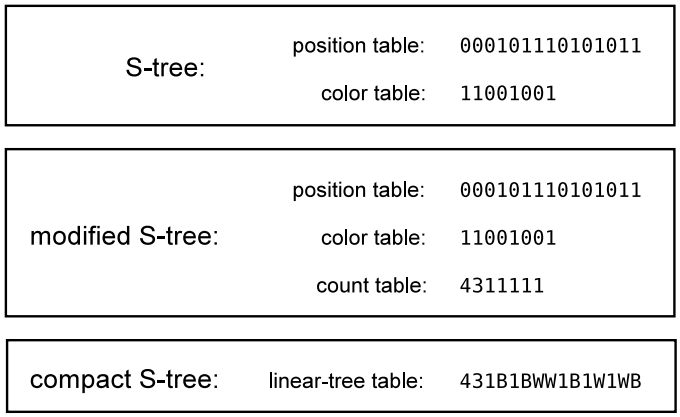


Figure 4.11: Different S-tree representations of the bintree in Figure 4.10.

The **S-tree** [DSS94] is a linear representation of the bintree based on the concept of position codes and color codes. The S-tree for a bintree is built traversing the tree in preorder and appending a specific code for each node found (including internal and white nodes): for each internal node a "0" is stored, and for each leaf node a "1"

is stored in a position table. Additionally, for each leaf node its "color" is added to a color table, encoded with a single bit (1 for black nodes, 0 for leaf nodes). In the S-tree representation searching for a specific region requires to traverse the sequence until the region is found. Figure 4.11 shows the S-tree representation of the image in Figure 4.10. Notice that the encoding procedure of the S-tree is identical to the procedure explained for the FBLQ.

The **modified S-tree** [CW95] adds to the S-tree a third table (the *count* table) that stores, for each internal node, the number of leaves in its left subtree. The addition of the *count* table to the modified S-tree allows more efficient searches. In a modified S-tree we can look up a given code without traversing the complete sequence, using the *count* values to jump forward in the sequence: if we want to go to the second child of a node, instead of traversing the complete first child we use *count* to jump directly to the second child. This makes navigation faster but the space requirements are significantly higher.

The **compact S-tree** [CWL97] is another variant of the S-tree that uses a single table (the *linear-tree table*) to represent the same information stored in the three tables of the modified S-tree. The compact S-tree is built from a bintree traversing the tree in preorder. For each internal node found, the number of leaves in its left subtree is added to the table. For each leaf node found, its color is added to the table. The final result is a sequence that contains a single symbol per node of the bintree: the number of leaves covered by each internal node and the color of each leaf node. An example of compact S-tree is shown in Figure 4.11. Notice that the symbols "B" and "W" that denote the color of a leaf node are finally encoded as integers (-1 for "B" and 0 for "W"), so the final representation would be a sequence of integer values.

The **S^+-tree** [DSS94] is a secondary-memory implementation of the S-tree, designed for efficiently storing the S-tree representation of a binary image in secondary memory. An S^+-tree stores the S-tree representation in blocks, where each block contains a fragment of the position table together with the corresponding fragment of the color table. Each block is built appending bits until it is full: bits in the position table are added from the beginning of the block, and bits of the color table are added from the end of the block. The construction process forces the last node in a block to be a leaf node, and when no more leaves fit in a block, a new one is created. The value used to index the next block is the encoded path from the root of the bintree to the first leaf node in the block (encoding with "0" a left child and with "1" a right child). Figure 4.12 shows an example of S^+-tree representation. Notice that the values used in the internal nodes of the B^+-tree to index the leaf blocks are actually the path in the bintree to the first leaf node in each block: the first node in block B2 is node `g`, corresponding to path `0011`; the first node in block B3 is node `k`, corresponding to path `11`. To make easier the computations in the leaf nodes, the S^+-tree adds a prefix and suffix to each node, that is shown separately in Figure 4.12: the prefix represents the separator of the

node, storing each 0 with 0 and each 1 with 01, while the suffix stores a 0 for each 1 in the separator. If we add the prefix at the beginning of the node table and the suffix at the end of the node, the representation of the node becomes a complete bintree representation of the branches in the current node (each 01 in the prefix and 0 in the suffix acts as a *dummy left branch* in the bintree). This addition is mainly for convenience, to make easier the computations inside the node, since the values can be easily computed from the separator.

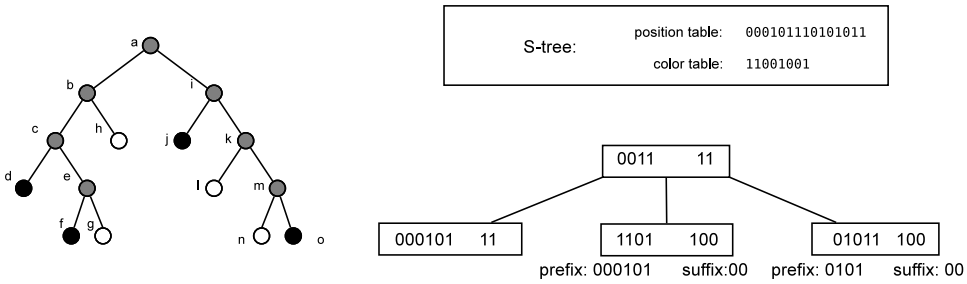


Figure 4.12: S^+-tree representation of the S-tree in Figure 4.11.

4.3.5 Representation of general raster data

The spatial access methods presented in the previous section are mainly oriented to the representation of binary images. However, as we explained earlier in this chapter, general raster datasets may need to store more complex information at each cell of the raster. In this section we present spatial access methods specifically designed to store and query raster images with multiple non-overlapping features. This includes thematic maps or general raster datasets where we are interested in performing queries attending to the values stored in the cells.

Most of the representations are evolutions or variants of the data structures presented in the previous section, that are enhanced with additional information or modified to represent efficiently the values stored in the raster. Particularly, these representations are designed to efficiently solve the following usual operations when representing general raster data:

- *exists(f,w)*: check whether any cell in the window w contains the value f.
- *report(w)*: return all the different values (features) contained inside window w.
- *select(f,w)*: report all the occurrences of value (feature) f inside window w.

4.3.5.1 Quadtree representations

The **Simple Linear quadtree** [NP95], or SL-quadtree, is a straightforward generalization of the linear quadtree representation. Consider a raster dataset where each cell contains a *value* (a specific feature in a thematic map or the value of a numeric feature like elevation). We can decompose the raster recursively using the same strategy followed in binary quadtrees, but we stop decomposition when we find *uniform* regions. The leaf nodes in this tree will store the actual *value* of the associated region. In an SL-quadtree, each uniform region is encoded with two numbers: its locational code and the *value* of the corresponding region. The sequence of values is stored in a B^+-tree and indexed according to the locational codes exactly like in a linear quadtree. Figure 4.13 shows an example of simple linear quadtree representation, where three different "colors" are possible for each region. The sequence of locational codes in the leaf nodes corresponds to the locational code of each block in the image, independently of its color.

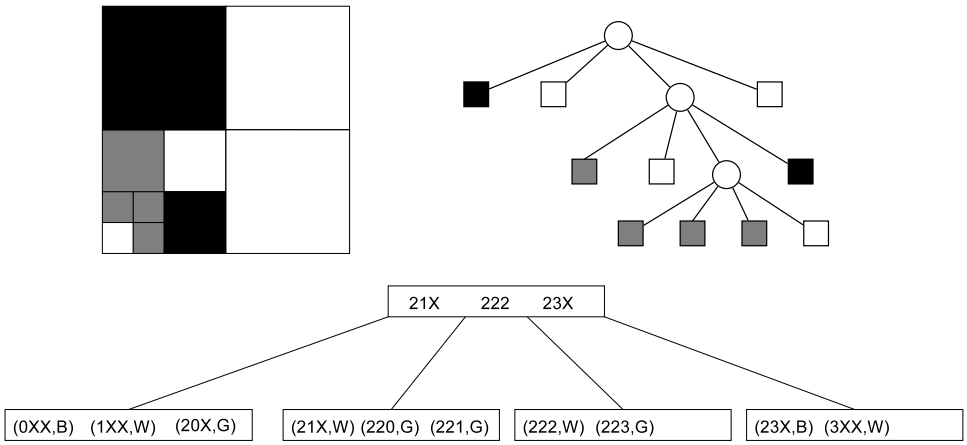


Figure 4.13: Simple Linear Quadtree representation.

Notice that spatial operations in an SL-Quadtree are identical to the operations in a binary raster. However, it is difficult to answer queries involving the values stored: a complete traversal of the raster (or the spatial window) is required to retrieve the regions with a specific value.

The **Hybrid Linear Quadtree** [NP97], or HL-quadtree, tries to obtain a better representation for managing different values in a raster image. In the HL-quadtree, all the nodes of the quadtree decomposition of an image are physically stored. Given a colored image, it is first decomposed into uniform regions as explained for the previous method. Then, a pair $\langle loc, feat \rangle$ is stored for each node in the quadtree (including internal nodes), where loc is the locational code of the corresponding node and $feat$ is a bit array that indicates, for each different feature in the raster image,

whether it exists in the region covered by the current node (bit set to 1) or not (bit set to 0). The sequence of pairs, as in the previous linear quadtree representations, is then placed in the leaf nodes of a B^+-tree that indexes the content by locational codes. According to this, the sequence of codes that would be stored by the HL-Quadtree representation of the image in Figure 4.13 would be (considering that the features are sorted as WGB):

```
(XXX,111) (0XX,001) (1XX,100) (2XX,111)
(20X,010) (21X,100) (22X,110) (220,010)
(221,010) (222,010) (223,100) (23X,001)
(3XX,100)
```

The first pair corresponds to the root node, that contains all the possible values. The remaining codes follow a preorder traversal of the conceptual tree. Notice that, in this case, internal nodes may contain more than one feature (the bit array contains many 1s), but leaf nodes will store a single 1 in their bitmap.

In this representation spatial operations can be solved as in the SL-Quadtree, since the B^+-tree is still indexed by location codes. The advantage of the HL-Quadtree is in the specific queries for values/features: to check whether feature f exists inside a window, the window can be decomposed into maximal blocks and these blocks searched in the B^+-tree. Each block (or ancestor) found will contain the explicit list of all the features contained in the current region, so we can answer immediately. Similar simple algorithms can be used to *report* all the features contained in a window of to *select* all the locations of a feature inside a window.

4.3.5.2 Bintree based representations

The S^*-tree [NP99] is an evolution of the S^+-tree for the representation of multiple non-overlapping features. The S^*-tree is based on the same idea proposed in the S^+-tree, to store position and color bits together in nodes of a B^+-tree. To store information of the different features, the S^*-tree stores, for each internal node, a color string of f bits (where f is the number of different features) that indicates features contained in the region. For leaf nodes a color string of $\lceil \log f \rceil$ bits is used to indicate the feature it contains.

Figure 4.14 shows an example image, its bintree decomposition (top), the corresponding position and color tables (middle) and the final (simplified) S^*-tree representation. In this example, node `a` (the root node) contains all features, so its color string is `111`; node `e` contains only white and gray leaves, so its color string is `110` (the order of the colors is white-gray-black, as in previous examples). Leaf nodes contain 2-bit color codes (00 for white nodes, 01 for gray nodes, 10 for black nodes). The position codes and color codes are placed in the S^*-tree in the leaves of a B^+-tree. The partition of the blocks occurs between nodes `e` and `f` and between

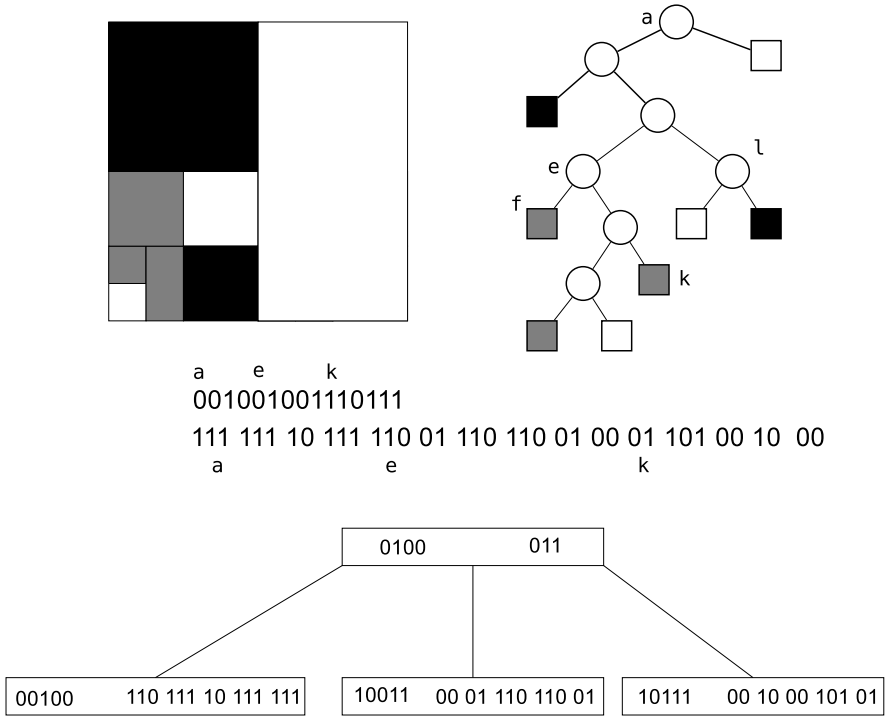


Figure 4.14: S*-tree representation of a colored image.

nodes `k` and `l`. Hence, the indexes stored in the internal nodes of the B^+-tree are the representation of a path from the root of the bintree to nodes `f` and `l` respectively: 0100 and 011. A small but important difference between the S*-tree and the S^+-tree is that the S*-tree does not force the last node in each block to be a leaf node (in our example, the last node in the first block of the B^+-tree is `e`, an internal node in the bintree). This change reduces the unused space in the blocks of the B^+-tree and has small implication in queries.

An enhancement in the B^+-tree data structure used in the S*-tree has been proposed to optimize the efficiency of queries by feature [MNPT01]. This enhancement simply adds to the entries in internal nodes of the B^+-tree a bitmap that indicates, for each different feature, whether the pointed node contains at least

an entry with the feature. This bitmap can be easily generated computing the `or` of the bitmaps that would be associated with the entries in the child node. This modification allows us to stop navigation in the internal nodes of the B^+-tree if its bitmap indicates that there are no entries in the corresponding leaf for a given value. This enhancement can be also applied to SL-quadtrees and HL-quadtrees, leading to a "subfamily" of spatial access methods called BSL-tree (SL-quadtree over enhanced B^+-tree), BHL-tree (HL-quadtree) and BS^*-tree.

4.3.6 Representation of time-evolving region data

The representation of geographic information also needs to deal with the *evolution* of spatial objects or features over time. In many real-world GIS applications, information does not only contain a spatial component, but also a temporal component. The modeling and compact representation of spatio-temporal datasets has been widely studied. In this section we will focus on the representation of spatio-temporal information under the raster model, or *time-evolving region data*. Notice that the representation of sequences of raster images can also be seen as a special case of a more general problem: the representation of *multiple overlapping features*. Many of the proposals for the representation of time-evolving raster images are similar in structure and functionalities to the representations for general raster data (multiple non-overlapping features)

Most spatio-temporal data require a set of common queries that involve filtering information depending on a set of spatial and temporal constraints. Spatial constraints are similar to those explained in previous sections. We will distinguish the following categories of temporal constraints (the same categories are used in Section 4.1 for temporal graphs):

- *Time-slice queries* are concerned with the state of the dataset at a specific point in time.
- *Time-interval queries* refer to a longer time interval. The simplest query asks for the values at each time point in the interval, but we also use two special semantics: *weak* queries return all the cells that fulfilled a condition (e.g. having a value higher than a threshold) at some point in the query interval, while *strong* queries return only the cells that fulfilled the condition during the complete interval.

The simplest approach to represent a temporal raster dataset is to store a different data structure with the state of the raster at each time instant. This approach is usually unfeasible due to the high storage requirements to represent the complete raster at each time instant. Because of this, several proposals have been presented to reduce the cost of storing temporal raster data, while supporting time-slice and time-interval queries. In the remaining of this section we will describe

representations of time-evolving region data under the raster model, based mostly on the linear quadtree.

4.3.6.1 Quadtree-based proposals

Inverted Quadtrees

The Fully-Inverted Quadtree [CT91] (FI-quadtree) is a data structure for the efficient access to a collection of images or binary raster datasets. It is based on the idea of grouping a collection of images in a single quadtree. For a collection of n images, an inverted quadtree is built creating a *full* quadtree (i.e. a quadtree that contains all possible branches, no leaf nodes exist in the inner levels). Each node v in the FI-quadtree contains a bitmap B_v of length n, such that $B_v[i] = 1$ iff node v is black for image i. Figure 4.15 shows the conceptual full quadtree generated for a collection of 3 different images, where for each node we indicate the sequence of images where this node would be black (recall, in the FI-quadtree this information is finally actually stored in a bitmap of length n in each node). The nodes are traversed in preorder and placed in secondary memory.

Since the quadtree is full and the nodes are of equal size we can know the actual offset of each node easily. To search a given region (i.e. a node or set of nodes in the quadtree), all the nodes in the path to the desired region are checked. In each node it is very easy to check the value of a given image accessing its bitmap. Because of this, the FI-quadtree supports efficiently spatial query operations and also queries involving specific features. Nevertheless, the FI-quadtree requires too much space to represent a collection of similar images: a full quadtree is created even if most of the nodes do not exist for any image, and each node requires as many bits as images in the collection. Hence, this representation is only suitable for small collections of images, and does not take advantage of the regularities between images.

The Dynamic Inverted Quadtree [VM95] (DI-quadtree) is an evolution of the FI-quadtree that tries to solve some its drawbacks. It is based on the same idea of building a full quadtree to store a collection of images, but each node in a DI-quadtree points to a list of identifiers that contain the images for which the node is black. This makes the representation dynamic (new images can be added to the lists) and reduces the space requirements of the FI-quadtree since the space required by each node is now proportional to the number of images where it is a black node. To query a DI-quadtree we use the same strategy of FI-quadtrees, accessing at each node the list of images instead of the bitmap.

MOF-Tree

The *Multiple Overlapping Features Quadtree* (MOF-tree [MNPP97]) is essentially a combination of multiple quadtrees in a single data structure. It can be seen as an enhancement of some of the ideas in the FI-quadtree and the DI-quadtree. A region quadtree is built, and for each node we store for which features (or, in our case, time instants) the region is covered. When a region is not covered for

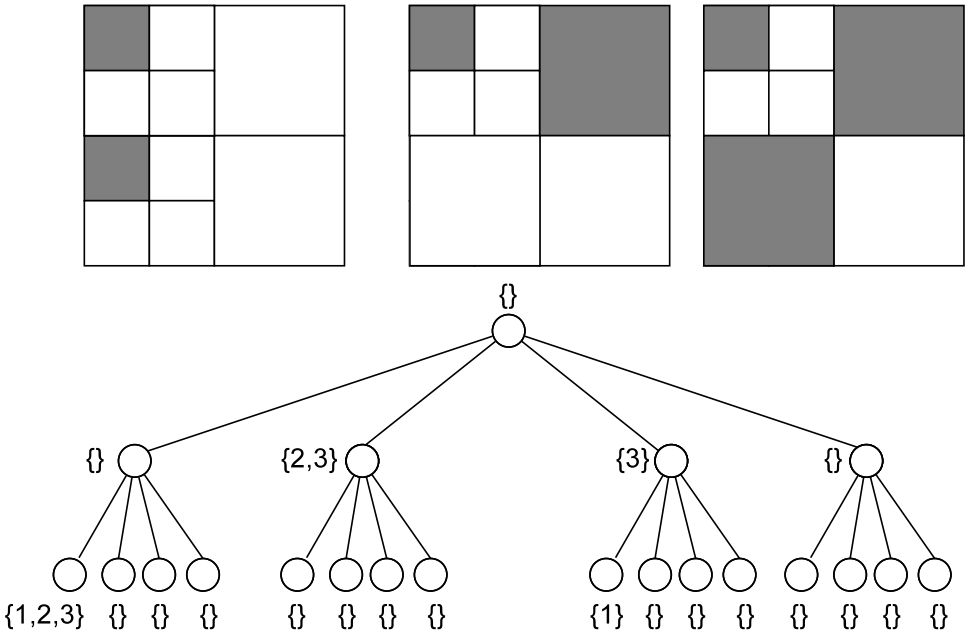


Figure 4.15: Conceptual representation of an inverted quadtree.

any time instant or is covered in all of them, decomposition stops. A conceptual MOF-tree representation is shown in Figure 4.16. Notice that the MOF-tree is not a complete tree, therefore the number of nodes may be much smaller than in an inverted quadtree.

The MOF-tree can be implemented as a linear quadtree representing each node with its locational code. Each node must store, in addition to its code, the information of the features that cover it. A bit vector *feature* indicates, for each feature, whether it overlaps (covers partially) the region of the node. A second bit vector *cover* indicates, for each feature, whether the region is completely covered (the bitmap *cover* can be omitted in the leaves since it would be equal to *feature*: a leaf node is either entirely covered or entirely uncovered by all the features). Additionally, a flag is added to each node to indicate whether it is a leaf node.

Spatio-temporal queries are solved in a MOF-tree like in inverted quadtrees, checking in this case the *feature* bit vector to know whether the region was covered by a specific feature.

Overlapping Linear Quadtrees

Overlapping Linear Quadtrees (OLQ [TVM98]), as their name states, are based on the *overlapping* technique. The idea of overlapping, that can be applied to many data structures, is to reduce the redundancy in the representation of a collection of

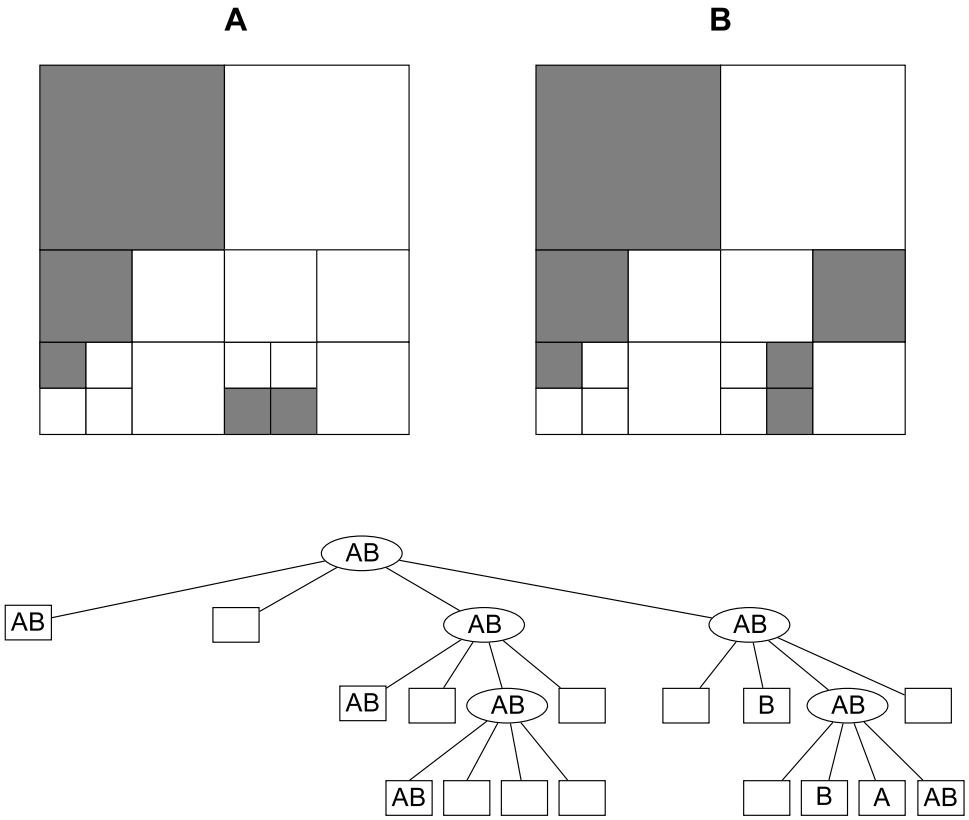


Figure 4.16: Conceptual MOF-tree.

datasets by reusing the common parts of the data structures used to represent each of them. Overlapping linear quadtrees are built combining ideas from Overlapping B^+-trees [TML99] with a linear quadtree representation.

Consider a sequence of linear quadtrees that store the state of the raster at each time instant. We will assume in our examples that the linear quadtree implementation uses *fixed length (FL)* encoding of the blocks, and the sequence of block codes can be stored in a B-tree variant in secondary memory. In overlapping linear quadtree, consecutive B-trees are "overlapped" so that the common branches are represented only once. This means that, if a node in a B-tree would store the same codes than another node in the previous quadtree, we do not create a new node but point to the node in the previous B-tree. Figure 4.17 shows the overlapping linear quadtrees representing two binary images.

The access to a single quadtree in overlapping linear quadtrees has no overhead:

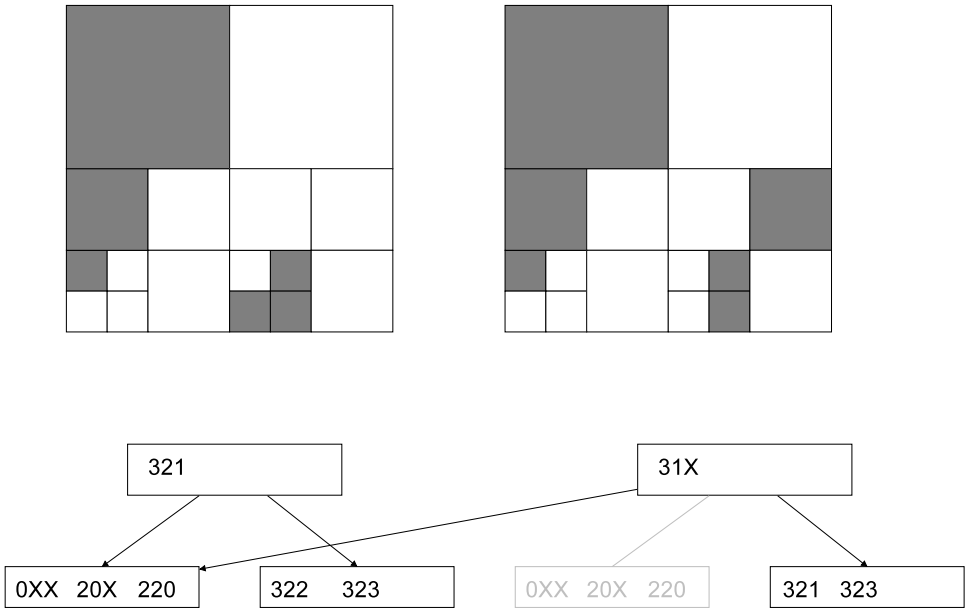


Figure 4.17: Overlapping linear quadtrees.

time instant queries can be answered exactly as they would if independent linear quadtrees were stored for each time instant, since the linear quadtree data structures are not affected by the OLQ (the only difference is that pointers between nodes may point now to "shared" nodes.

To efficiently answer time-interval queries, the OLQ also stores a set of pointers between nodes of different trees. These pointers allow us to locate the nodes storing "future" and "past" versions of the current node, therefore making it easier to retrieve the state of the raster in a long time interval without querying all the time instants.

Time Split Linear Quadtree and Multiversion Linear Quadtree

The Time Split Linear Quadtree (TSLQ [TVM01]), based on the Time-Split B-tree [LS89], combines spatial and temporal information in the entries of a single B^+-tree data structure. All nodes in a TSLQ have two additional fields *StartTime* and *EndTime* that indicate their valid times. The leaves of the TSLQ store a set of entries including a locational code and its creation time. This additional information is used in spatio-temporal window queries to only traverse nodes corresponding to the query times. To speed up time-interval queries, the leaves of the TSLQ can be enhanced with backward- and forward-pointers, like Overlapping Linear Quadtrees, to efficiently navigate between "versions" of a node.

The Multiversion Linear Quadtree (MVLQ [TVM00]) also combines temporal

and spatial information. It is based on the Multiversion B-tree [BGO+96], that can be seen as a collection of B^+-trees storing information about different time intervals. In the MVLQ, information is stored in a collection of *version trees*, where nodes combine locational codes with validity intervals.

The Multiversion Access Structure for Evolving Raster Images [TVM04] (MVASERI) can be seen as a modified MVLQ, with an identical data structure but different algorithms to manage update operations.

Overlapped CBLQ

The overlapped CBLQ [Lin97a] is based on the overlapping mechanism explained in Overlapping Linear Quadtrees: the quadtree representation of an image is encoded using only the differences with the quadtree representation of the previous one. Overlapped CBLQs are designed to compress groups of similar images by using a modified CBLQ to represent the overlapped quadtrees. They use special codes to indicate changes in a node between a version and the previous ones, and only the fragments of the quadtree that change have to be represented.

Overlapped CBLQs can obtain very good compression of a temporal raster dataset or any set of similar binary images, thanks to combining the overlapping technique with a compact representation of each quadtree. However, this data structure is not very efficient to access a specific image, since to retrieve a given image we have to sequentially compute all the previous ones, rebuilding each image from the previous one using the "differential" CBLQs at each step.

4.3.7 Representation of vector data

The efficient representation of spatial objects in the vector model is probably one of the most studied problems in GIS. Many different data structures have been proposed to index different kinds of objects. Some of them are specifically designed to represent objects of a specific kind (for instance, they store only points), while others are mainly designed to be able to represent any kind of complex object. For example, the k-d-tree [Ben75] (and its variant K-D-B-tree [Rob81]), the HB-tree [LS90] are mainly designed to store points. To store more complex objects, the R-tree [Gut84] and its variants are most widely used. We will not describe in detail most of these representations, since the work in this thesis is mainly oriented to representations based on the raster model (a complete explanation of these and other data structures can be found in different surveys [GG98, BBK01]). In the remaining of this section we will describe the R-tree and its variants, that are the most popular access methods and will be used in the last part of this thesis.

4.3.7.1 The R-tree

The R-tree [Gut84] is a data structure designed mainly for the representation of regions, or in general any spatial object that can not be represented using points.

Later, it has been widely used to the representation of all kinds of spatial objects, including points. The R-tree can be used to represent n-dimensional objects.

The key idea in the R-tree is to group together objects that are "close" in space and to index objects according to their *minimum bounding box*. The minimum bounding box of an object is the smallest n-dimensional rectangle that contains the object (it is usually called minimum bounding rectangle, or MBR).

The R-tree is based on a balanced tree structure similar to a B-tree. Each node of the tree is a block that may contain a predefined number of entries. Each node is usually stored in a disk page and has a maximum capacity determined by the disk page size and also a minimum capacity that determines the space utilization of the tree. In a leaf node, each entry stores information about the objects stored: typically, the information for each object is the MBR of the object and a unique identifier of the object (OID). Internal nodes contain entries that point to other nodes: each entry in an internal node contains a pointer to the corresponding child, together with the MBR for the child node (that is, the smallest rectangle that contains all the MBRs in the child node). This guarantees that all the objects in the descendants of a node are contained in the MBR of the node. An important characteristic of the R-tree is that the MBRs of nodes at the same level may *overlap* (that is, for any given node, many of its children can contain a given point at the same time).

Example 4.2: Figure 4.18 shows an example of R-tree representation. The rectangles R1,...,R12 are the MBR of 12 different objects indexed by the R-tree. The rectangles RA,...,RF are the MBRs generated in internal nodes of the R-tree. The right side of the image shows the actual tree structure, considering nodes with 3 entries as maximum capacity. Notice that the MBRs for objects may overlap (R7 and R10 overlap), and MBRs in internal nodes also overlap.

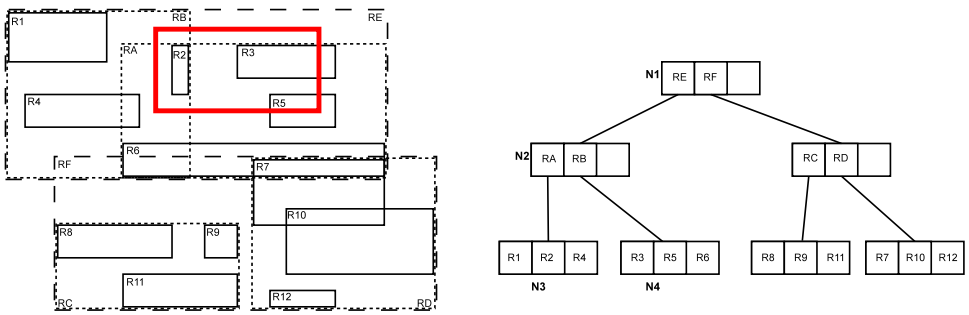


Figure 4.18: R-tree space decomposition and data structure.

The procedure to search in an R-tree, given a query rectangle (in general, the MBR of the region searched), performs a top-down traversal of the tree. At each internal node (starting in the root node), all the entries are checked to determine

if they intersect with the query rectangle. For all the entries that intersect with the query rectangle, the child node pointed to by this entry must be checked also. When a leaf is reached, all the entries in the leaf are also checked. Each entry in the leaf whose MBR intersects with the query rectangle is a candidate result. Only the elements in the candidate list would have to be checked later to see if they actually intersect with the query, comparing their actual geometries.

Example 4.3: In the R-tree of Figure 4.18, if we want to search objects within the region defined by the highlighted rectangle, we would start at the root node. The only entry that overlaps the query rectangle is RE, so we would navigate to the first child. Now two entries (RA and RB) intersect with the query rectangle, so both children N3 and N4 must be checked. Finally, we would need to check all the entries in N3, where R2 intersects with the query rectangle, and N4, where R3 and R5 intersect with the query rectangle.

The R-tree is a completely dynamic representation. Objects may be added and deleted from the representation at any time. To insert a new object we compute its MBR. Then, starting at the root, a recursive procedure tries to find the best child of the current node to add the new entry. To do this, at each step, we check all the entries in the current node. The entry whose MBR needs less enlargement to contain the MBR of the new entry is chosen, and we repeat the process recursively in the pointed child. When a leaf node is reached, the new entry is simply added to the node. If the maximum capacity of the leaf node is exceeded, the leaf is split and the entries in internal nodes are adjusted recomputing their MBRs. Splits in internal nodes are handled similarly. Deletions are implemented in a similar way, merging nodes as necessary.

Many variants of the R-tree have appeared to enhance the capabilities of the original R-tree. R*-trees [BKSS90] are similar to R-trees but use a different algorithm during insertion, trying to obtain at the same time MBRs of the smallest area and with reduced overlap. Many other data structures based on the R-tree have been used in related problems: for example, the HR-tree [NST99], MV3R-tree [TP01] are examples of data structures based on the R-tree designed for the representation of moving objects in spatio-temporal databases.

Part I

New static data structures

Chapter 5

K^2-trees with compression of ones

The K^2-tree, introduced in Section 3.2, is a compact representation of binary matrices that has proved its efficiency in different domains for the representation of different real-world matrices. However, the application of the K^2-tree is somehow limited by some characteristics in the matrices it represents, particularly their sparseness and clusterization: the K^2-tree compression is based on compressing groups of "close" 1s with as few bits as possible.

In many real-world applications, data can be viewed as a binary grid where the regions of zeros and ones are heavily clustered but the number of ones and zeros is approximately the same. Typical examples of this are black and white (or *binary*, or bi-level) images, where the image can be represented as a matrix where each cell corresponds to a *pixel* of the original image.

In this chapter we present our proposal for the compact and indexed representation of clustered binary grids, focusing on the compression of large regions of zeros and ones. Our proposal is conceptually an extension of the K^2-tree (originally designed to compress sparse binary matrices) to handle a wider range of datasets where clusterization exists but sparseness is not required. To obtain a better representation of images with many ones, we extend K^2-trees to behave like classic region quadtrees in the sense of compressing efficiently large regions of ones as well as large regions of zeros. We design several simple but effective variants that behave like the K^2-tree but compress efficiently regions full of ones, making our variants useful to compress binary raster images.

The rest of the chapter is organized as follows: in Section 5.1 we summarize some concepts and state-of-the-art proposals for the representation of binary images. After this, Section 5.2 presents the conceptual description of our proposals and the different implementations of the conceptual representation, including the basic

navigation operations. In Section 5.3 we present a theoretical analysis comparing the space requirements of our representation with state-of-the-art proposals based on recursive space decomposition. Then, Section 5.4 introduces a method to improve compression results in our representations. Section 5.5 describes the implementation of different query algorithms and typical operations over our representations. Section 5.6 presents an experimental evaluation of our proposals, showing the differences in compression and query times among our representations and comparing them with alternative representations. Section 5.7 introduces some other applications of our encodings. Finally, Section 5.8 summarizes the contributions on this chapter and refers to additional related contributions that will appear later in the thesis.

5.1 Representation of binary images

The design of data structures for the compact representation and efficient manipulation of images has been studied for a long time in different fields of computer science, from the pure storage and manipulation of images in image databases to specific applications in GIS (visualization of raster data). Binary images are usually easier to compress, and are widely used to represent digitized documents (i.e. scanned text documents stored as images without OCR processing), in image communications (fax transmissions of text documents and drawings) and in many other documents related to cartographic information (maps, urban plans and all kinds of spatial information in Geographic Information Systems). Many different proposals exist for the efficient compression of binary images that take into account the expected properties of some kinds of binary images. For example, representations designed primarily for the transmission of digitized text documents aim to compress efficiently line patterns and should provide efficient compression and decompression but do not need to provide advanced operations on images. On the other hand, representations primarily designed for computer graphics or GIS do need to support efficiently the usual operations in the domain: efficient access to regions of the image, image manipulations such as rotation, scaling or set operations involving multiple images, etc.

Some of the simplest strategies for binary image compression involve a sequential compression of the pixels in the image in row-major order: techniques such as run-length encoding, LZW or Huffman encoding can be used to compress a binary image simply using the bit sequence determined by the pixels. Many other techniques are based on different variants of classic compression techniques. For example, *JBIG1*[1] and its evolution *JBIG2*[2] are standards of the Joint Bi-level Image Experts Group, published as recommendations of the International Telecommunication Union (ITU) for the encoding and transmission of bi-level images. These standards are designed

[1] `http://www.jpeg.org/jbig`
[2] `http://jbig2.com`

for the compression of binary images that may contain large portions of text, and are widely used for the transmission of information in fax machines. They are based on an arithmetic coder specially tuned and use different compression techniques to compress text regions, images or other content.

A common approach for the representation of binary images with support for image manipulations and queries in compressed form are pixel tree approaches, that have been presented in Section 4.3.4: *quadtree* representations decompose an image into quadrants, while *bintree* representations decompose the image alternating in dimensions. Our proposals in this chapter can be seen as compact and querieable pixel tree representations, in the sense that they build a conceptual tree similar to the conceptual quadtree of an image.

5.2 Our proposal: K^2-tree with compression of ones (K^2-tree1)

Our proposal, as its name states, is conceptually an evolution of the K^2-tree explained in Section 3.2, to enhance it with the ability to efficiently compress uniform regions of ones, as it already compresses regions of zeros.

In the original K^2-tree representation, a K^2-ary conceptual tree is built to represent an adjacency matrix. The conceptual tree generated can be seen as a form of quadtree that only stops decomposition when uniform regions of 0's are found. This conceptual representation chosen in the K^2-tree comes from its original purpose: it was designed to compress Web graphs, which have an extremely sparse adjacency matrix. In our proposal we focus on obtaining an efficient representation for binary matrices that contain large clusters of 0's and 1s.

To fulfill this goal, we propose to change the conceptual tree used by the K^2-tree representation. In our proposal, the conceptual tree used to build the K^2-tree data structure will use a different method for partitioning the matrix: decomposition of the binary matrix will stop when uniform regions are found, be them full of 0's or 1s. This simple change will reduce the number of nodes in the conceptual tree when large regions of 1s appear in the binary matrix. Figure 5.1 shows an example of binary matrix with large clusters of ones and its conceptual tree representation, for $K = 2$. We use the typical quadtree notation to show the nodes of our conceptual trees: internal nodes are *gray* nodes, containing mixed regions of 0s and 1s, regions of 1s are marked with *black* nodes and regions of 0s are marked with *white* nodes. Notice also that, in this example, the conceptual tree built is similar to a region quadtree representation of the binary matrix, explained in Section 4.3.4.1, since our proposal for $K = 2$ is based on the same partitioning method used in quadtrees. However, the K^2-tree may use different values of K in different levels of the tree, and the K^2-tree1, as we will show later in this chapter, can also use this and other techniques that make the conceptual K^2-tree1 different from a classic region

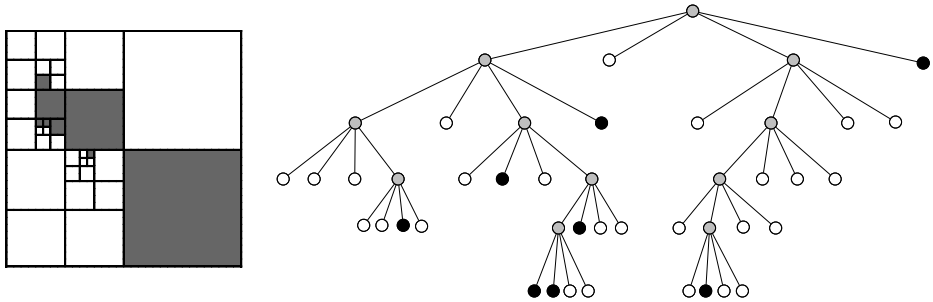


Figure 5.1: K^2-tree with compression of ones: changes in the conceptual tree.

quadtree.

In the remaining of this section we present our proposals for the physical representation of the conceptual tree. In all the approaches we focus on keeping the navigational properties of original K^2-trees, thus providing efficient traversal over the conceptual tree using only *rank* operations. We will propose the different representations and show how they can be used to perform basic navigation over the conceptual tree. We note that to perform queries over the underlying matrix we will rely on three basic navigational operations that must be provided by our representations:

- $internal(n)$ to know whether a node is internal or leaf.
- $children(n)$ to locate the K^2 children of the current internal node. The K^2 siblings will be in consecutive positions in all our representations, so we only need to find the position of the first child.
- $color(n)$ to know the color (white or black) of a leaf node[3].

If we support these operations, the basic navigation algorithms in our representations can be implemented as if our representation was a pointer-based region quadtree. For example, to find whether a cell (x, y) is black or white, we start at the root of the tree. From the cell coordinates we can determine the child of the current node that should contain it, so we find the location of the children using the *children* operation, move to the corresponding offset in the K^2 siblings and check whether the child is an *internal* node. If it is not internal, we can return immediately the

[3]Notice that these operations can be combined in different ways. For example, we may consider an operation *color* that returns three values: gray, white or black, and thus replaces both *internal* and *color*. The goal of this partition in basic operations is to show conceptually the ability of our proposals to answer high-level queries using only the low-level operations. In practice, our proposals will actually combine computations of the different operations to improve efficiency.

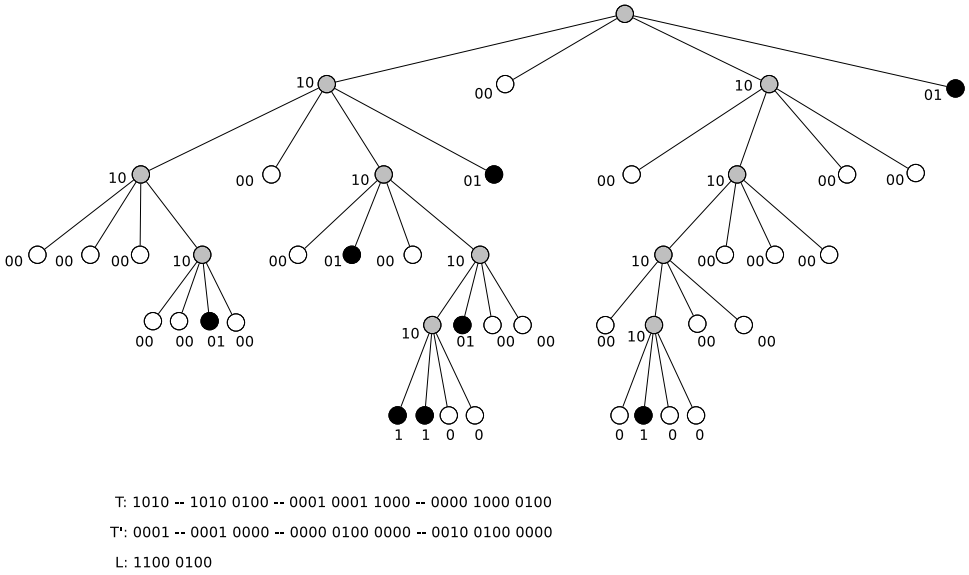


Figure 5.2: K^2-tree1$^{2\text{bits}-\text{naive}}$ representation.

color of the node, that will give us the actual value of the cell. If the node is internal, we repeat the process recursively until we reach a leaf node. General window queries can also be solved using the basic functions as a replacement for the pointer-based tree navigation.

5.2.1 Encoding the colors of nodes with three codes

5.2.1.1 Naive coding: the K^2-tree1$^{2\text{bits}-\text{naive}}$

The simplest approach for the representation of the conceptual tree is simply to use 2-bit codes to represent each node in the conceptual tree. We assign 10 to gray nodes (internal nodes), 01 to black nodes and 00 to white nodes. In the last level of the conceptual tree we can use simply the codes 1 or 0 for black or white nodes respectively, since no gray nodes can appear. Figure 5.2 shows the bit assignment for the conceptual tree of Figure 5.1. Notice that the bit assignment is not arbitrary, but chosen to maintain the ability to navigate the K^2-tree efficiently: the first bit of each node indicates whether it is internal (1) or a leaf (0), and the second bit determines whether a leaf represents a white or a black region.

Once the codes have been assigned, we perform a levelwise traversal of the conceptual tree to obtain all the codes. We place them in 3 different bitmaps, as follows:

- T will contain the first bit of each node in all the levels except the last one.
- T' will contain the second bit of each node in all the levels except the last one.
- L will contain the (only) bit of each node in the last level.

In this representation, each node is identified by its position in T or L. In L each node represents a single cell and is either black or white. The bitmap T contains a 1 for each gray (internal) node and a 0 for each leaf node, so $internal(p) = true \leftrightarrow T[p] = 1$. Only gray nodes have (exactly K^2) children. Therefore, given a (gray) node at any position p in T, its K^2 children will begin at position

$$children(p) = \text{rank}_1(T, p) \times K^2 = p'$$

If p' is higher than the length of T (we will denote this using $|T|$), the access is instead performed in L at position $p'' = p' - |T|$. This operation is identical to the navigation required in original K^2-trees, and uses a rank data structure over T to support the $rank_1$ operation in constant time.

Finally, to provide the support for the operations we also need to provide a way to tell apart white from black nodes (*color* operation). This operation is immediate in a K^2-tree1$^{2\text{bits}-\text{naive}}$: given a (leaf) node at position p in T, its *color* is given by $T'[p]$ (white nodes have $T'[p] = 0$, black nodes $T'[p] = 1$).

In terms of space the K^2-tree1$^{2\text{bits}-\text{naive}}$ is a completely naive proposal: internal nodes do not need to use 2-bit codes to be differentiated. However, it is presented here because it leads to the simplest implementation of the data structure with virtually no overhead in terms of query time (recall from Section 3.2 that the rank_1 operation needed to find the children of a node was already required in the original K^2-tree). We will show later that this naive representation using 2 bits per tree node is already competitive in space with some compact quadtree representations that do not support direct navigation like our representations. The rest of this section is devoted to present more space-efficient alternatives for encoding the conceptual K^2-tree1.

5.2.1.2 Efficient encoding of leaf nodes: the K^2-tree1$^{2\text{bits}}$

To reduce the space overhead of the K^2-tree1$^{2\text{bits}-\text{naive}}$, a simple improvement is obtained by noticing that using 2 bits for each node in the upper levels of the K^2-tree1 representation is not necessary, since we only have 3 possible node types. We can use a single bit 1 to mark an internal node and only use 2-bit codes 00 and 01 to distinguish between white and black leaves. Figure 5.3 shows the code assignment and resulting bitmaps following this strategy. The only difference with the previous proposal is that the bitmap T', that still contains the second bit of each 2-bit code, now only contains a bit for each 0 in T. The encoding used determines that T' will contain a bit with the color of each leaf node, and all leaf nodes are encoded with a 0 in T.

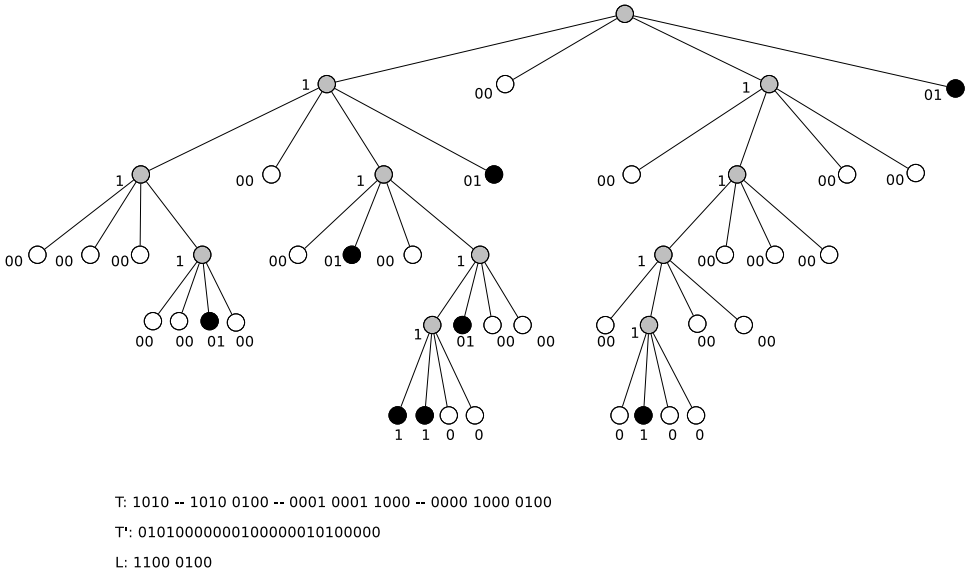


Figure 5.3: K^2-tree1$^{2\text{bits}}$ representation.

To provide basic navigation, we note that the basic property of K^2-tree navigation still holds: the bitmaps T and L are identical to those of the K^2-tree1$^{2\text{bits}-\text{naive}}$, and therefore the children of a node at position p are located at $\text{rank}_1(T, p) \times K^2$. Additionally, we can know that a node is internal simply checking $T[p]$. The operation to know the color of a leaf node is still very efficient: given a leaf node at position p in T, its color can be obtained using the formula $color(p) = T'[rank_0(T, p)]$. The new *color* operation is shown in Algorithm 5.1, considering also accesses to the bitmap L. It only differs from the K^2-tree1$^{2\text{bits}-\text{naive}}$ in the additional rank_0 operation to compute the position in T' that stores the color of the leaf. We can compute this $rank_0$ operation in constant time using the same rank structure that is already necessary for the *children* operation[4].

We note at this point that the bitmaps stored in the K^2-tree1$^{2\text{bits}}$ can be seen as a reordering of the FBLQ, described in Section 4.3.4.1: our K^2-tree1$^{2\text{bits}}$ turns out to use the same encoding for the nodes of the conceptual tree than the FBLQ representation, and simply places them in a way that allows us to perform efficient traversals over the tree. On the other hand, the FBLQ representation was designed for storage and can not be navigated easily like a pointer-based representation. Our representation is also similar in structure to the variants of S-tree presented in Section 4.3.4, in the sense that it stores two bitmaps equivalent to the position list and the color list in the S-tree variants. However, our representation provides an

[4]Notice that in a binary sequence each bit is either 0 or 1, so $\text{rank}_0(T, p) = p - \text{rank}_1(T, p)$

Algorithm 5.1 *color* operation in K^2-tree$1^{2\text{bits}}$.

```
 1: function COLOR(tree, p)
Output: nodeType: BLACK, WHITE
 2:     T ← tree.T
 3:     T' ← tree.T'
 4:     L ← tree.L
 5:     isLast ← false
 6:     if pos ≥ T.size then
 7:         p ← p − T.size
 8:         isLast ← true
 9:     end if
10:     if isLast then
11:         if L[p] = 0 then
12:             return WHITE
13:         else
14:             return BLACK
15:         end if
16:     else
17:         p' ← rank_0(T, p)
18:         if bitget(T', p') then
19:             return BLACK
20:         else
21:             return WHITE
22:         end if
23:     end if
24: end function
```

efficient method to navigate the quadtree representation using only a small rank structure in addition to the bitmaps, while S-tree variants either must be traversed sequentially (the basic S-tree representation) or require significant additional space (the modified and compact S-tree representations store the size of the left subtree for each internal node).

5.2.1.3 Reordering codes: navigable DF-expression

Following the observation in the previous subsection, we present here another alternative that is based on an already existing encoding. In this case, we use the most efficient encoding proposed for the DF-expression [KE80], explained in Section 4.3.4.1. This encoding is similar to the K^2-tree1$^{\text{2bits}}$, but uses 1 bit for white nodes and 2 bits for black and gray nodes, based on the observation that white nodes may be usually the most frequent nodes in the quadtree representation of real-world binary matrices. We call this representation K^2-tree1$^{\text{df}}$, and propose it as an alternative representation that shows the flexibility of our techniques.

In a K^2-tree1$^{\text{df}}$ a white node will be encoded with "0", a gray node with "10" and a black node with "11" (except those black nodes in the last level of the tree that can be represented using a single bit "1"). We also use the bitmaps T, T' and L like in the previous proposals: T will store the first bit of all the nodes except those in the last level, T' the second bit of all the nodes except those in the last level and L will store the bits for the nodes in the last level.

In a K^2-tree1$^{\text{df}}$ the navigation can still be performed using simply rank operations but is slightly more complicated. The used codes affect the navigational properties of the K^2-tree. In a K^2-tree1$^{\text{df}}$, the nodes that have children are those labeled with a "1" in T and a "0" in T'. Therefore, to find the children of a node at position p we need to count the number of ones in T until that position (this accounts black nodes and gray nodes), and subtract the number of black nodes (that is, we need to count how many 1s appear in T' that correspond to the nodes until position p). Following this reasoning, the formula to locate the children of a node is $children(p) = \text{rank}_1(T, p) - \text{rank}_1(T', rank_1(T, p))$. The procedure to check whether a node is internal is also slightly changed: a node is internal if $T[p] = 0$ (it is black or gray) and the corresponding bit in T' is "0": $T'[rank_1(T, p)] = 0$. Finally, to compute the color of a leaf node at position p we check $T[p]$: if $T[p] = 0$ it is white, otherwise it is black.

This representation, although it is based on a possibly better code assignment, presents some drawbacks when implemented as a K^2-tree1. The first one, related to space, is that we must add a rank structure not only to T but also to T'. This additional space, although sublinear in theory, may lead to worse compression than the K^2-tree1$^{\text{2bits}}$ in some cases. The second drawback, related to time efficiency, is that the children operation requires now two *rank* operations. Again, the *rank* operation is theoretically constant-time but it is the most costly operation in our representation. Nevertheless, the K^2-tree1$^{\text{df}}$ is presented as an example of the

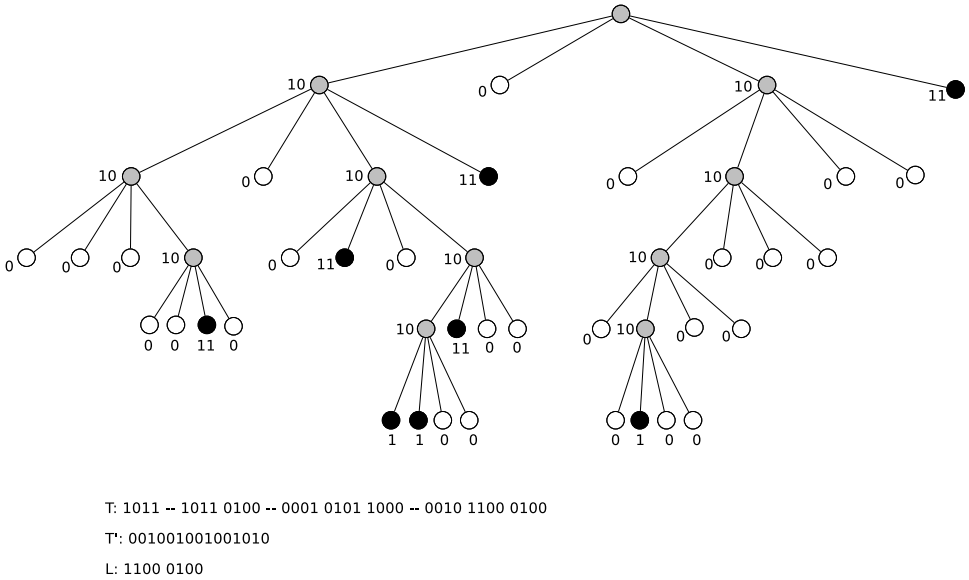


Figure 5.4: K^2-tree1$^{2\text{bits}}$ representation.

versatility of our conceptual proposal. As we will see later in this chapter, our representations can be compared in terms of space with some of the most compact quadtree representations while supporting advanced navigation over the conceptual tree as if we had a pointer-based representation.

5.2.2 A custom approach: K^2-tree1$^{1-5\text{bits}}$

Our last variation is a proposal designed to compress regions of ones but making as few changes to the original K^2-tree as possible. This implementation, K^2-tree1$^{1-5\text{bits}}$, fulfills two objectives: *i)* it never uses more space that the original K^2-tree, and *ii)* it tries not to change the original K^2-tree structure (actually only the query algorithms will be changed).

The key idea in the K^2-tree1$^{1-5\text{bits}}$ is the following observation: in a K^2-tree, a node is set to 1 if and only if some of its children are set to 1. Therefore, given any node that is set to 1, at least one of its K^2 children must be set to 1 in order to obtain a valid K^2-tree. The K^2-tree1$^{1-5\text{bits}}$ takes advantage of this property to obtain a compact representation of regions of ones using the same data structure of original K^2-trees.

In a K^2-tree1$^{1-5\text{bits}}$ the bitmaps are constructed almost like in a simple K^2-tree. However, a K^2-tree1$^{1-5\text{bits}}$ is built over a slightly modified tree representation of the binary matrix. In this representation we use a conceptual tree similar to those

in the previous variants, but we represent regions of ones using the "impossible" combination shown below: a region of ones will be represented with a 1 with K^2 0 children. Figure 5.5 shows an example with a conceptual tree where regions of ones are marked with black nodes: in the K^2-tree$1^{1-5\text{bits}}$ variant, the conceptual tree contains a node with K^2 "white" children for each region of ones. The conceptual K^2-tree$1^{1-5\text{bits}}$ is now identical to the original K^2-tree, in the sense that it uses a single bit to encode each node and it can be represented using the same bitmaps T and L of original K^2-trees. The only difference is that the "impossible" combination in original K^2-trees is given a special meaning in this representation and the query algorithms must be adapted to take this into account.

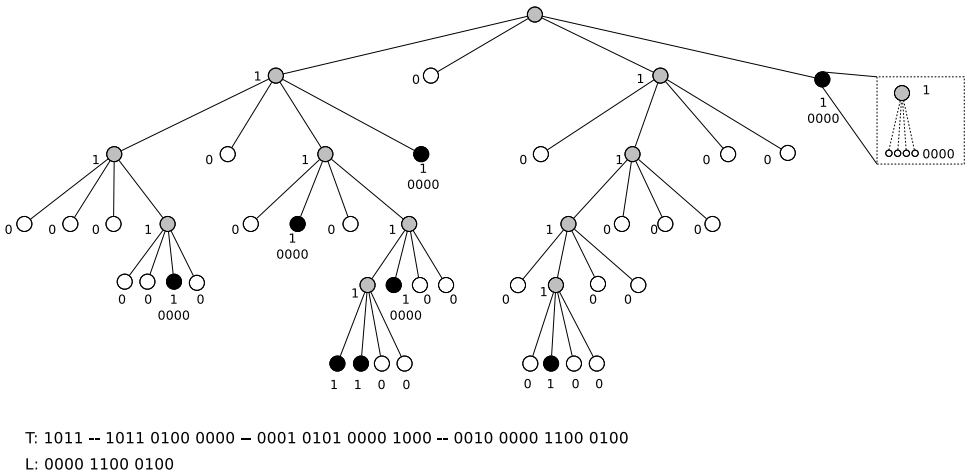


Figure 5.5: K^2-tree$1^{1-5\text{bits}}$ representation.

To perform the basic navigation in this approach, we can use the navigational properties of K^2-trees: the children of an internal node will be located at position $children(p) = \text{rank}_1(T, p) \times K^2$, as usual. The method to know whether the current node is internal requires a more complicated check: a node is internal if $T[p] = 1$ and at least one of its children is a 1 ($T[children(p), children(p) + K^2 - 1] \neq 0 \ldots 0$). Additionally, whenever we find a leaf node we can check its *color* easily: a leaf node is white if $T[p] = 0$ and black if $T[p] = 1$.

The code assignment in the K^2-tree$1^{1-5\text{bits}}$ is very asymmetric, it uses much more space to represent a black node than to represent a white one, particularly for large K. This fact is inherited from the K^2-tree conceptual design, that is tuned to compress matrices with few ones. This characteristic of our proposal may be a drawback in some contexts but also an interesting feature for the representation of images with a different rate of black and white pixels. Trivially, the codes can be swapped depending on the characteristics of the image to compress better white or

black nodes as needed.

The main advantage of the K^2-tree$1^{1-5\text{bits}}$ is that the K^2-tree data structure is not modified at all, because the structure we build is actually a K^2-tree representation of a slightly different conceptual tree. This makes it easier to apply changes or enhancements of the K^2-tree to a K^2-tree$1^{1-5\text{bits}}$, since only the algorithms need to be adjusted to deal with regions full of ones.

Another advantage of the K^2-tree$1^{1-5\text{bits}}$, derived from the first one, is that a K^2-tree$1^{1-5\text{bits}}$ is guaranteed to use at most the same space as the original K^2-tree. To demonstrate this property we look at the conceptual tree and the codes used. In the K^2-tree$1^{1-5\text{bits}}$, a region full of 1s is encoded as a 1 with K^2 0s as children; in the original K^2-tree, the same region full of 1s would be encoded as a complete subtree of 1s: a node with K^2 children, each of them with K^2 children, and so on until the last level of the tree was reached. Therefore, the K^2-tree$1^{1-5\text{bits}}$ saves the space required by the complete subtree (except its first two levels); in the worst case no (aligned) regions of 1s larger than K^2 bits exist and the K^2-tree$1^{1-5\text{bits}}$ representation is identical to an original K^2-tree. Because of this property we consider the K^2-tree$1^{1-5\text{bits}}$ a *conservative* approach in the sense that it is designed to never require more space than a classic K^2-tree, even if the represented matrix does not contain regions of ones. In the next section we will show a theoretical analysis of the space utilization of all our variants.

5.3 Space Analysis

In this section we will analyze the space required by all our representation as a function of the number of nodes in the tree representation of the binary matrix. We will focus on the basic case where $K = 2$, hence our conceptual tree representations are similar to region quadtrees and we can analyze them in terms of the bits required per node of the conceptual quadtree. We assume a binary matrix $M[2^n \times 2^n]$, and its quadtree representation Q. In this context, we call *level* of a node the distance from the root to the node. Without loss of generality, we will consider the case where the quadtree decomposition reaches the n-th level in at least one of its branches (i.e., some of the quadtree leaves correspond to cells of the original matrix). Hence level n is the last level of the quadtree[5]. At each level i, $1 \leq i \leq n$, Q contains w_i, b_i and g_i white, black and gray nodes respectively, adding up to a total of $w = \sum_{i=1}^{n} w_i$ white nodes, $b = \sum_{i=1}^{n} b_i$ black nodes and $g = \sum_{i=1}^{n} g_i$ gray nodes. We analyze the space occupation of the different K^2-tree representations considering the number of bits devoted to represent each node. We will consider only the size of the bitmaps, ignoring for now the lower order terms added for the rank structures.

[5]In the general case, the height of the quadtree may be smaller than n if the smallest uniform region found is of size $2^b \times 2^b$ with $b > 0$. The nodes in the quadtree representation would not change if we consider each $2^b \times 2^b$ submatrix as a cell in a submatrix of size $2^{n-b} \times 2^{n-b}$. The quadtree for the new submatrix would have now height $n - b$.

For simplicity, we will not take into account the fact that the root node is omitted in all our representations, which would change all our formulas in just 1-2 bits.

A K^2-tree uses 1 bit for each gray node and white node, but for black nodes at level l it would require a complete subtree to represent all the cells covered by the node ($\sum_{i=l}^{n}(K^2)^{i-l}$ bits). The total space used by a K^2-tree is therefore $w + b + g + \sum_{i=0}^{n} b_i \sum_{j=i}^{n}(K^2)^{j-i}$. A K^2-tree1$^{\text{2bits-naive}}$ uses 2 bits for each node in the upper levels, and 1 bit for each node in the last level, for a total space of $2(w+b+g)-(w_n+b_n)$. A K^2-tree1$^{\text{2bits}}$ uses 1 bit for gray nodes and 2 bits for each black or white node, except those at the last level of decomposition, that can be represented using a single bit. The total space used is simply $g+2(w+b)-(w_n+b_n)$. A K^2-tree1$^{\text{df}}$ uses 1 bit for white nodes and 2 bits for black or gray nodes (except black nodes in the last level, encoded with a single bit), for a total $w+2(g+b)-b_n$ bits. A K^2-tree1$^{\text{1-5bits}}$ uses 1 bit for each gray or white node, and 5 (K^2+1) bits for each black node that is not in the last level of decomposition. The space is $w + g + 5b - 4b_n$.

The exponential cost of indexing regions of ones in a K^2-tree makes clear the advantage of our variations. However, if a matrix does not contain regions of ones, the additional bits used to represent white nodes in the K^2-tree1$^{\text{2bits-naive}}$ and the K^2-tree1$^{\text{2bits}}$, or gray nodes in the K^2-tree1$^{\text{df}}$, make these alternatives more expensive than a classic K^2-tree. On the other hand, it is easy to see that a K^2-tree1$^{\text{1-5bits}}$ can never use more space than the classic structure, since its space can be rewritten as $w+b+g+\sum_{i=1}^{n-1} b_i K^2$. This makes this alternative a good candidate for indexing data whose structure is not known.

To provide a more comprehensive comparison between our proposals, we use the equality $b + w = 3g + 1 \Rightarrow b + w \approx 3g$ to reduce our formulas. The total space for a K^2-tree1$^{\text{2bits-naive}}$ is

$$2g + 2(3g + 1) - (w_n + b_n) \approx 8g - (w_n + b_n)$$

The K^2-tree1$^{\text{2bits}}$ further compresses the gray nodes, obtaining

$$g + 2(3g + 1) - (w_n + b_n) \approx 7g - (w_n + b_n)$$

The K^2-tree1$^{\text{df}}$ obtains similar results, yielding

$$w + 2g + 2b - b_n \approx 5g + b - b_n$$

Finally, the K^2-tree1$^{\text{1-5bits}}$ space is

$$w + g + 5b - 4b_n \approx 4g + 4b - 4b_n$$

In Table 5.1 we show these results together. As a rough estimation of an average case, we add in the third row an estimation that provides a direct comparison between the proposals. For this estimation we use $b = w$ (and thus $g = \frac{2b}{3}$), which

Approach	General	Simplified	Average estimation
K^2-tree1$^{\text{2bits}-\text{naive}}$	$2(w+b+g)-(w_n+b_n)$	$8g-(w_n+b_n)$	$\frac{13b}{3}$
K^2-tree1$^{\text{2bits}}$	$g+2(w+b)-w_n-b_n$	$7g-(w_n+b_n)$	$\frac{11b}{3}$
K^2-tree1$^{\text{df}}$	$w+2(g+b)-b_n$	$5g+b-b_n$	$\frac{23b}{6}$
K^2-tree1$^{1-\text{5bits}}$	$w+g+5b-4b_n$	$4g+4b-4b_n$	$\frac{14b}{3}$

Table 5.1: Space analysis for all the approaches. The average estimation assumes $b=w$ and $b_n=w_n=b/2$.

is the average case for random images. Additionally, we follow the steps in [Che02] and let $b_n=w_n=\frac{b}{2}$ to get rid of the level-dependant terms. The result is a rough estimation of the efficiency of the proposals in terms of the number of black nodes.

Trivially, the K^2-tree1$^{\text{2bits}}$ representation will always perform better than a K^2-tree1$^{\text{2bits}-\text{naive}}$ (the codes used for each node are always equal or shorter in the K^2-tree1$^{\text{2bits}}$). The K^2-tree1$^{1-\text{5bits}}$ appears in our formulae close to the K^2-tree1$^{\text{2bits}-\text{naive}}$ but with slightly worse results in the average estimation. However, we note that the estimation uses a simplification ($b=w$) that has no significant effect in the K^2-tree1$^{\text{2bits}-\text{naive}}$ or the K^2-tree1$^{\text{2bits}}$, but can be considered as a worst case scenario for the K^2-tree1$^{1-\text{5bits}}$ or the K^2-tree1$^{\text{df}}$, because their asymmetric codes can take advantage of the skewed distribution of black and white nodes.

To give a more generalized estimation of the efficiency of our proposals, we characterize different cases depending on the actual rate of 1s and 0s in the matrices. We use a pixel oriented random model, in which an inary matrix (image) is composed of $2^n \times 2^n$ pixels, each of which is black with probability p and white with probability $1-p$. We call the quadtree representing a $2^n \times 2^n$ matrix a class-n quadtree. Given the value n and the black pixel probability p, the average number of black, white and gray nodes at each level of a class-n quadtree can be obtained easily [VM93]. This yields an average case estimation for all our proposals as a function of the black pixel probability. In Table 5.2 we show, for different values of p, the average overall rate of black/white/gray nodes in the quadtree and the average rate of black/white nodes in the last level of the quadtree, as fraction of the total number of nodes. These values are computed following the formulae in [VM93]. In our case the values are computed for class-10 quadtrees, but the rates of nodes are similar for most values of n.

Using the values in Table 5.2, we compute the expected space in bits for all our proposals, in bits per node of the conceptual quadtree. In Table 5.3 (top) we show the expected space, in bits per node of the conceptual tree, of all our proposals for matrices of size $2^{10} \times 2^{10}$ following the random pixel model. For matrices with many ones ($p \geq 0.5$), the K^2-tree1$^{\text{2bits}}$ representations obtains the best results in average. In this estimation, for matrices with few ones the K^2-tree1$^{1-\text{5bits}}$ representation appears to be the best, using just 1 bit/node.

Black pixel probability	b	w	g	b_n	w_n
0.99	0.67	0.08	0.25	0.23	0.08
0.9	0.59	0.16	0.25	0.39	0.16
0.75	0.51	0.24	0.25	0.43	0.24
0.5	0.375	0.375	0.25	0.36	0.36
0.25	0.24	0.51	0.25	0.24	0.43
0.1	0.16	0.59	0.25	0.16	0.39
0.01	0.08	0.67	0.25	0.08	0.23

Table 5.2: Average rate of nodes in the quadtree as a function of the black pixel probability.

At this point we must remark that this analysis, although it provides some insight to the efficiency of our proposals, cannot be taken as a basis for results in realistic scenarios. All our representations are designed for contexts in which the bits are clustered, so that the representations can take advantage of that. In this random pixel model, our representations would achieve a poor compression of the input matrix. As a proof of concept, Table 5.3 (bottom) shows the expected space results for the same examples in the top part of the figure but now in bits/one. The values for sparse matrices go well above 10 bits/one in all the representations. K^2-trees are designed to take advantage of clusterization and other regularities, and they typically achieve much better compression ratios when clusterization exists. In the experimental evaluation of our proposals we will show the actual performance of our proposals in real matrices that contain some clusterization of ones and zeros.

5.3.1 Comparison with quadtree representations

In Table 5.4 we compare our approaches with several existing representations for quadtrees. We perform a theoretical comparison with the three encodings proposed for the DF-expression, and different codes designed for linear quadtrees: Gargantini codes in a classic linear quadtree (using FL-encoding), the breadth-first oriented CBLQ and FBLQ representations and the Improved Quadtree (the Compact-IQ without additional compression of the bitmaps). We make some simplifications in the formulae: the cost of additional rank structures is not considered in our proposals, and the additional cost of the B^+-tree in the linear quadtree (LQT) approach is also not considered. The last columns of the matrix show the space estimation as a function of the number of black nodes in the quadtree and also an average estimation for all the proposals. Additionally, for each representation we show the memory model of application (in-memory representations are designed as

Black pixel probability	K^2-tree1$^{2bits-naive}$ (bits/node)	K^2-tree1^{2bits} (bits/node)	K^2-tree1^{df} (bits/node)	K^2-tree1$^{1-5bits}$ (bits/node)
0.99	1.69	1.44	1.69	2.76
0.9	1.45	1.20	1.45	1.80
0.75	1.33	1.08	1.33	1.32
0.5	1.28	1.03	1.27	1.06
0.25	1.33	1.08	1.25	1.00
0.1	1.45	1.20	1.25	1.00
0.01	1.69	1.44	1.25	1.00
Black pixel probability	K^2-tree1$^{2bits-naive}$ (bits/one)	K^2-tree1^{2bits} (bits/one)	K^2-tree1^{df} (bits/one)	K^2-tree1$^{1-5bits}$ (bits/one)
0.99	0.21	0.18	0.21	0.35
0.9	1.02	0.84	1.02	1.26
0.75	1.79	1.46	1.79	1.78
0.5	3.09	2.49	3.06	2.56
0.25	5.38	4.37	5.05	4.04
0.1	9.15	7.57	7.88	6.31
0.01	21.27	18.12	15.73	12.58

Table 5.3: Average space results for all approaches, in bits/node (top) and bits/one (bottom).

compact representations for main memory; external representations are designed to be indexed and accessed in external memory). Additionally, we mark the representations that support efficient access to regions of the image without sequential decompression: notice that in general the in-memory representations are designed for an efficient processing of the complete image, and as such do not provide support for algorithms to access regions of the image.

The third representation for the DF-expression is probably the most compact representation in many realistic cases, although in our estimation it behaves slightly worse that our K^2-tree1^{2bits}. This is because this representation, like our K^2-tree1$^{1-5bits}$, is not symmetric, and should behave better when the image contains mostly white pixels. Notice that the code assignment in our representations is very similar to the encoding used in other quadtree representations, and this leads to identical space results. The FBLQ is equivalent to the K^2-tree1^{2bits} representation in terms of space, but our proposal is designed to support efficient navigation. The same occurs comparing the K^2-tree1^{df} representation with the most efficient encoding of the DF-expression. The IQ representation does not obtain good results

Representation	Model	Spatial access	Space	Space$_{est}$
K^2-tree1$^{\text{2bits}-\text{naive}}$	In-memory	✓	$8g - (w_n + b_n)$	$13b/3$
K^2-tree1$^{\text{2bits}}$	In-memory	✓	$7g - (w_n + b_n)$	$11b/3$
K^2-tree1$^{\text{df}}$	In-memory	✓	$5g + b - b_n$	$23b/6$
K^2-tree1$^{1-\text{5bits}}$	In-memory	✓	$4g + 4b - 4b_n$	$14b/3$
DF-expression(1)	In-memory		$8g$	$16b/3$
DF-expression(2)	In-memory		$5g + b$	$13b/3$
DF-expression(3)	In-memory		$5g + b - b_n$	$23b/6$
LQT(Gargantini)	External	✓	$3nb$	$3nb$
CBLQ	In-memory		$8g$	$16b/3$
FBLQ	In-memory		$7g - (w_n + b_n)$	$11b/3$
IQ	In-memory		$7g$	$14b/3$

Table 5.4: Space results for the different approaches. Space$_{est}$ assumes $b = w$ and $b_n = w_n = b/2$.

since we do not consider additional compression. Finally, it is easy to see that our representations are comparable in space with any compressed in-memory quadtree representation but support random access to the contents of the images thanks to the simple navigation using *rank* operations.

5.4 Enhancements to the K^2-tree1

Several improvements have been proposed for original K^2-trees to improve their space utilization and query efficiency. Most of them can be directly applied to the K^2-tree1 with similar results.

The first improvement in original K^2-trees is the utilization of different values of K in different levels of the conceptual tree, particularly larger values in the upper levels of the tree and smaller values in the lower levels. The utilization of different values of K in the K^2-tree does not change the navigation algorithms, but the computations to find the children of a node must be adjusted. This simply requires to add an adjustment factor to the formula that computes the *children* of a node. This change is not affected by the encodings used in our variants, so this improvement can be directly applied to the K^2-tree1, obtaining *hybrid* implementations of the K^2-tree1 with different values of K per level.

Matrix vocabulary: The most important enhancement over original K^2-trees is the addition of a matrix vocabulary to compress small submatrices according to their

frequency. This enhancement, explained in Section 3.2.2, yields a representation in which the subdivision is stopped at a fixed level, and all the submatrices corresponding to the nodes at that level are represented in a different manner.

In a K^2-tree1$^{2\text{bits}-\text{naive}}$, K^2-tree1$^{2\text{bits}}$ or K^2-tree1$^{\text{df}}$, the matrix vocabulary can be added following exactly the same procedure. The bitmaps T and T' will represent upper levels of the conceptual K^2-tree1, and the last level will be represented by the matrix vocabulary that replaces the bitmap L. This allows all these variants to take advantage of any small-scale regularities in the binary matrix that can be exploited using the statistical encoding of the submatrices. In a K^2-tree1$^{1-5\text{bits}}$, the addition of a matrix vocabulary can also be implemented like in the original K^2-tree: the bitmap T is stored in plain form and L is replaced by a statistically-encoded sequence stored using DAC.

Notice that the improvements added to the basic K^2-tree representation increase significantly its ability to compress regions of ones: if large regions of 1s appear in the binary matrix and we use a matrix vocabulary for submatrices of size $K' \times K'$, a short codeword can be used to represent a complete submatrix full of 1s, thus reducing the cost of storing a complete subtree full of 1s in the K^2-tree by a significant fraction. This means that, when using the improvements proposed in this section, and particularly a matrix vocabulary, original K^2-trees should also improve their ability to represent clustered binary images. Later in this chapter we will compare original K^2-trees with our K^2-tree1 variants, also studying the effect of the matrix vocabulary in compression.

5.5 Algorithms

All the main operations supported by original K^2-trees are also supported by our representations. The algorithms to find the value of a single pixel have been covered in the basic traversal explanation. Other queries that look for all the ones in a row/column of the binary matrix are solved filtering the children traversed at each step. In this section we will present the resolution of different operations of interest in image processing where quadtrees are typically used.

5.5.1 Window queries

Window queries are a relevant query in different domains of spatial data. In a pointer-based quadtree representation, a window query can be efficiently answered performing a top-down traversal over the tree and collecting the set of nodes that correspond to regions that overlap the query window. Notice that in our proposals a window query can be performed using exactly the same procedure followed in a pointer-based quadtree, since we have the only required operations: know the *color* of a node and find the *children* of an internal node.

All our representations are variants of the K^2-tree, that already supports *range* queries. Essentially a window query is a range query, with the only difference that in our context we may not be interested in obtaining all the cells in the binary matrix contained in the window. Instead, our queries may want to retrieve all the maximal blocks that are strictly contained in the window or the maximal blocks that overlap the window. In any case, all operations can be performed using the same top-down traversal followed in pointer-based quadtrees or original K^2-trees.

5.5.2 Set queries

Our proposals efficiently support queries involving multiple K^2-tree1s at the same time. We call these *set queries*, understanding them as a variant of the set operations that are widely used in image processing and are supported by most quadtree representations. A set operation is a transformation that takes the representation of two images and returns the representation for a combination of those images. In general, each operation simply takes each pixel in the original images, applies the corresponding set operation to them and sets the corresponding pixel of the resulting image to the obtained value. Typically the result is the representation of the result image instead of the image itself. We will show how our representations can answer queries (for example, any window query) that involve multiple trees, returning directly the results of the query (the sequence of pixels or blocks) instead of building a compact representation as output. Then we will extend the problem to returning a compact representation in a set operation. We will consider four different set operations (most operations are based on logical tables in each node, so similar operators can be implemented following the same principles):

- Intersection ($A \cap B$): each pixel will be black iff that pixel is black in both of the original images.
- Union ($A \cup B$): each pixel will be black iff that pixel is black in any of the original images.
- Subtraction ($A - B$): each pixel will be black iff that pixel is black in A and white in B.
- X-or ($A \oplus B$): each pixel will be black iff the pixel is black in A and white in B or vice versa.

Given any query Q (for instance, a window query) that can be executed in a single K^2-tree1, we can efficiently support the same query on the union, intersection, etc. of two (or more) K^2-trees. Q determines, at each level of the conceptual K^2-tree1, which of the branches would have to be traversed (*candidate* nodes). In a query involving a simple K^2-tree1, at each level of the conceptual tree, the bitmaps of the K^2-tree1 are checked to find which of the candidate nodes are internal nodes

that can be traversed and which of the candidate nodes are actually leaf nodes that can be returned or discarded immediately. The process to answer queries on multiple K^2-tree1 will be similar, but it will require a synchronized traversal of all the involved K^2-tree1.

To perform a query Q over a combination of two K^2-tree1s K_x and K_y we simply need to traverse both K^2-tree1 representations at the same time and run the checks at each level for the corresponding nodes of each K^2-tree1. We can define a simple table, shown in Table 5.5, that computes the process to perform depending on the operation and the *color* of the node in each K^2-tree1. If we already know that the result is a black or white node, we *return* it as a result of Q. If the result is still unknown, we must keep traversing the K^2-tree1s. In this case we must distinguish three cases: *1)* when no information can be derived from the colors, both K^2-trees must be traversed (*traverse* operation). *2)* If a leaf node has been found in K_x, then its color c_x is stored and only K_y must be traversed ($traverse^y(c_x)$). *3)* If a leaf node has been found in K_y, we store its color c_y and only K_x is traversed ($traverse^x(c_y)$).

K_x	K_y	$K_x \cap K_y$	$K_x \cup K_y$	$K_x - K_y$	$K_x \oplus K_y$
w	w	$return(w)$	$return(w)$	$return(w)$	$return(w)$
w	g	$return(w)$	$traverse^y(w)$	n.p.	$traverse^y(w)$
w	b	$return(w)$	$return(b)$	n.p.	$return(b)$
g	w	$return(w)$	$traverse^x(w)$	$traverse^x(w)$	$traverse^x(w)$
g	g	*traverse*	*traverse*	*traverse*	*traverse*
g	b	$traverse^x(b)$	$return(b)$	n.p.	$traverse^x(b)$
b	w	$return(w)$	$return(b)$	$return(b)$	$return(b)$
b	g	$traverse^y(b)$	$return(b)$	$traverse^y(b)$	$traverse^y(b)$
b	b	$return(b)$	$return(b)$	$return(w)$	$return(w)$

Table 5.5: Table for set queries on K^2-tree1s.

As an example, given the two conceptual trees K_x and K_y shown in Figure 5.6, and a window query Q highlighted in the figure, we want to compute Q on the intersection of both trees. Starting at the root of both trees, we check the children that intersect the window (in this case, the second and fourth children). For each of them, we find out the *color* of the node in each of the K^2-tree1s. The fourth child of the root is black in both trees, so according to Table 5.5 we can add the corresponding submatrix to the result. The second child in both trees is gray, so we keep traversing both representations (finding the *children* of the node). In the next level all the submatrices are contained in the query window, so we check all children in both trees. The relevant cases are c (c') and k (k'), since the other intersections result in white nodes. For c and c', we find two gray nodes and keep traversing the

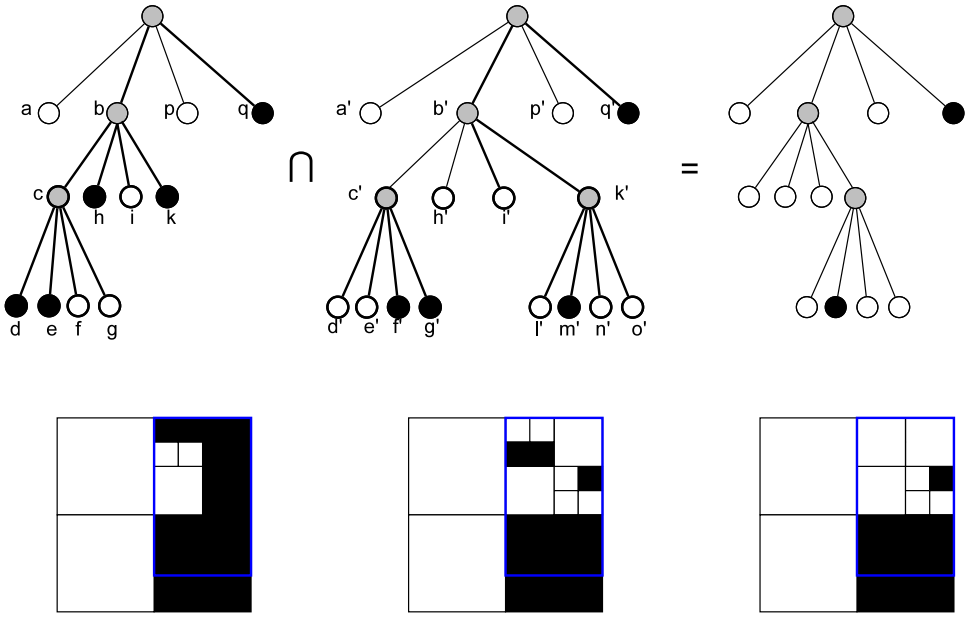


Figure 5.6: Intersection of two K^2-tree1s.

tree. Finally, in the children of c and c' we find that all the pairs are black-white nodes, so no blocks are added to the result. When comparing k and k' we find that k is black and k' is gray, so we only need to keep traversing the right tree and considering that the corresponding left node is black ($traverse^y(black)$). The children of k are checked and compared with the fixed black value in the left side. Comparing all nodes in Table 5.5, only m' returns a black block.

The actual implementation of the set queries, as we have seen, is simply based on our ability to check the color of a node and find its children. Hence, the conceptual query algorithm is directly applicable to all our proposals simply using the appropriate implementation of the basic operations.

5.5.2.1 Returning a compact representation

Set operations on compressed images, as we have seen, are usually considered to return also a compressed representation of the image. A proper set operation involving two of our tree representations should also return a new compact representation of the resulting image. In most cases, the support of our proposals for set operations during queries will suffice. However, in order to implement proper set operations that return the same K^2-tree1 representation of the resulting image more complex operations are needed.

K_x	K_y	$K_x \cap K_y$	$K_x \cup K_y$	$K_x - K_y$	$K_x \oplus K_y$
white	white	$return(w)$	$return(w)$	$return(w)$	$return(w)$
white	gray	$return(w)$	$copy(K_y)$	n.p.	$copy(K_y)$
white	black	$return(w)$	$return(b)$	n.p.	$return(b)$
gray	white	$return(w)$	$copy(K_x)$	$copy(K_x)$	$copy(K_x)$
gray	gray	$undef$	$undef$	$undef$	$undef$
gray	black	$copy(K_x)$	$return(b)$	n.p.	$complement(K_x)$
black	white	$return(w)$	$return(b)$	$return(b)$	$return(b)$
black	gray	$copy(K_y)$	$return(b)$	$complement(K_x)$	$complement(K_y)$
black	black	$return(b)$	$return(b)$	$return(w)$	$return(w)$

Table 5.6: Table for set operations on K^2-tree1s.

To build an output K^2-tree1 representation we consider that the result is precomputed completely before returning it (that is, we do not return partial representations until the complete image has been processed). We will use a table that determines the action to follow depending on the value in both trees; this table is very similar to the one used for queries, but this one explicits the operations needed to build the tree representation of the result.

We store a set of auxiliary lists $L_1, \ldots, L_n$, one for each level of the conceptual tree, that will be filled from left to right as needed. Then, the procedure starts traversing both K^2-tree1s as follows: we start at the root, and follow a depth-first traversal over the tree, using the *children* operation to reach the lower levels. At each level, we check the corresponding node in both representations. If the result for the node is already known (black or white) according to Table 5.6, we append the appropriate result to the list of the current level and stop traversal in the current branch, going to the next node. If Table 5.6 returns a *copy* or *complement* operation, we continue traversal only through the specified tree: operation $copy(K_i)$ indicates that the result for all the children of the current node will be the value in K_i, while $complement(K_i)$ indicates that the result will be the complement of each node in K_i (that is, white and black nodes are interchanged). When we find a copy or complement operation we do not use Table 5.6 until we have finished processing the subtree of the node that generated the copy. Finally, when the result is unknown in Table 5.6 ($undef$), we do not have any information about the current node, since its value depends on its subtree. When we find and $undef$ value we append it to the corresponding list and keep traversing. When the traversal of the subtree finishes and we reach back the node we will already know the color of the node, that can be easily computed from its K^2 children: if the K^2 children are leaves of the same color, they are removed and the $undef$ node is stored as a leaf of that color; otherwise the node is gray.

Nodes processed	L_1	L_2	L_3	
a	$[w]$	[ ]	[ ]	
b	$[w, u]$	[ ]	[ ]	
c	$[w, u]$	$[u]$	[ ]	
d	$[w, u]$	$[u]$	$[w]$	
e, f, g	$[w, u]$	$[u]$	$[w, w, w, w]$	
c^*	$[w, u]$	$[w]$	[ ]	$L_2.last : u \rightarrow w.$
h, i	$[w, u]$	$[w, w, w]$	[ ]	
k	$[w, u]$	$[w, w, w, g]$	[ ]	$copy(K_y)$
l, m, n, o	$[w, u]$	$[w, w, w, g]$	$[w, b, w, w]$	
k^*	$[w, u]$	$[w, w, w, g]$	$[w, b, w, w]$	
b^*	$[w, g]$	$[w, w, w, g]$	$[w, b, w, w]$	$L_1.last : u \rightarrow g.$
p, q	$[w, g, w, b]$	$[w, w, w, g]$	$[w, b, w, w]$	

Table 5.7: Construction of output K^2-tree as the intersection of the K^2-trees in Figure 5.6.

Depending on the K^2-tree1 variant used, the actual storage of values in the L_is should use the same codes used in the corresponding representations. For instance, in a K^2-tree1$^{\text{2bits}}$ each L_i should actually store 2 bitmaps corresponding to T_i and T'_i, the fragment of bitmap corresponding to level i. In a K^2-tree1$^{\text{df}}$ or K^2-tree1$^{\text{2bits-naive}}$ we would use similar 2-bit representations for the nodes. Once all the nodes have been traversed, we have in the L_is the sequence of node values using already the encoding corresponding to the different levels of the tree. The final representation will be the concatenation of the corresponding bitmaps in each L_i to obtain the final bitmaps T, T' and L.

Our implementation is not as efficient as other image representations based on quadtrees, that are specifically designed to answer these queries and return a compact representation. The operations in this section are provided as a proof of concept to show that our proposals can still provide set operations in compact space, and the processing time is still proportional to the number of nodes in both trees (we perform all operations in a single traversal of both trees, and the final concatenation of the lists requires time proportional to the number of nodes in the conceptual tree of the resulting image).

Example 5.1: Following the example representations in Figure 5.6, to return a compact representation of the intersection we would perform a synchronized depth-first traversal of both conceptual trees. Table 5.7 shows the step-by-step construction of the L_i tables as the nodes are processed. We start at the root of the tree, where all our L_is are empty. The first node processed is a/a', and the

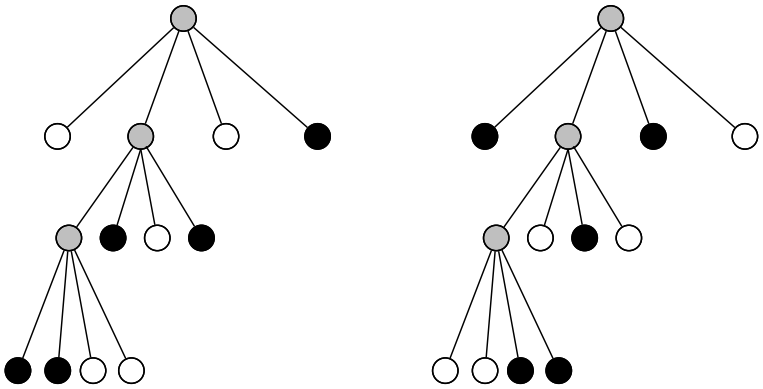
intersection is a white node. Hence w is appended to L_1. The second node processed is b, that is gray in both trees. We set the node as *undefined*, appending u to L_1. The value of the node will be computed after the complete subtree is traversed. Then we process c, that is again undefined, so we append u to L_2. Next, the four children of c and c' are processed, obtaining for each of them a white node that is appended to L_3. After we finish processing g and g' we go back to c (marked as c^* in Table 5.7). Since the last value in L_2 is u (undefined), we need to compute it from the values of its children. We check the last K^2 values in L_3, and find that they are all white nodes, so node c is actually a white leaf: we remove the last K^2 bits in L_3 and replace u with w in L_2. Then we would process h, i, k, obtaining white leaves for h and i. When we process k, k' (black and gray respectively), Table 5.6 tells us to *copy* the right subtree. Hence, we add g to L_2 and process the complete right subtree (nodes l', m', n', o'), appending all the values to the corresponding list (in this case, only L_3). Then we have to move up in the tree, going back to k (that is already *gray*, so no action is needed) and then to b (step b^* in Table 5.7). Since the last value in L_1 is u we would again compute it using the K^2 last values in L_2, hence setting it to g. The intersection is completed processing nodes p and q. At this point, the L_i contain the sequence of nodes for each level. The output K^2-tree is built concatenating the encodings for all the L_is in order. Notice that the lists are only accessed at their end points, and almost all the operations only require to append elements (at most we need to remove the last K^2 elements from a list).

5.5.3 Other manipulations

Other image manipulations that are usually simple in quadtree representations can be performed easily in our proposals. In most cases, the efficiency depends on the variant used (for example, some operations that are implemented with a simple left-to-right traversal of the bitmaps in the other proposals are not feasible in a K^2-tree1$^{1-5\text{bits}}$ due to the fact that black nodes are stored in two different levels). Nevertheless, all our variants are mainly designed for efficient querying, so queries that are supported very efficiently are expected in general to be more complex to implement if we want to return a K^2-tree1 variant as output. Next we present a sketch of the implementation of some usual image manipulations on top of our K^2-tree1 representations that would return a K^2-tree1 as output.

The *complement* operation of an image simply requires to swap black and white nodes in any quadtree. This operation can be performed with maximum efficiency in a K^2-tree1$^{2\text{bits}-\text{naive}}$ and a K^2-tree1$^{2\text{bits}}$, since the color of leaves is stored only in T'. A simple traversal of T' and L flipping all the bits suffices to obtain the complement K^2-tree1$^{2\text{bits}-\text{naive}}$ or K^2-tree1$^{2\text{bits}}$ [6]. This operation can be performed

[6] Actually, the code assignment explained in the K^2-tree1$^{2\text{bits}-\text{naive}}$ was "10" for internal nodes, so flipping all the bits in T' internal nodes would become "11" instead. This would have no effect on the query algorithms since the second bit of internal nodes is never accessed. For the purpose of validating this operation we can redefine the code of an internal node in a K^2-tree1$^{2\text{bits}-\text{naive}}$



2bits-naive	T : 0100 1000 T': 0001 0101 L: 1100	T : 0100 1000 T': 1110 1010 L: 0011
2bits	T : 0100 1000 T': 001 101 L: 1100	T : 0100 1000 T': 110 010 L: 0011

Figure 5.7: Complement operation in a K^2-tree1$^{2\text{bits}-\text{naive}}$ or K^2-tree1$^{2\text{bits}}$.

very efficiently in a sequential traversal of both bitmaps. If a matrix vocabulary is used instead of L, the procedure for flipping the bitmap is even easier: we only need to flip the bits in the matrix vocabulary, and the sequence of encoded matrices does not change. Figure 5.7 shows the representation of a quadtree and its complement using a K^2-tree1$^{2\text{bits}-\text{naive}}$ and a K^2-tree1$^{2\text{bits}}$: as we can see, only T' and L are changed.

To *rotate* an image 90 degrees we will use a recursive definition of the rotation of a quadtree: a rotation by 90 degrees of a quadtree node is obtained by first rotating its K^2 children and then rearranging them, so that the old sequence $NW - NE - SW - SE$ becomes $SW - NW - SE - NE$ (that is, we permute them in order 3–1–4–2. The rotation of a leaf node does not change the node. This operation can be performed in a K^2-tree following a depth-first traversal of the conceptual tree. Each time we find an internal node, we first process its children. For each node processed, we keep track of its offsets in T and T' or L: for a leaf, we store simply its position, for an internal node we store its position and also the intervals in T and T' or L where its descendants are stored. Notice that, for any internal node, the positions occupied by its descendants will be a contiguous interval in each level of the conceptual tree below the level of the node (with some possibly empty

as "1x" so that the resulting codes are still valid.

intervals). Notice also that the interval covered by a node is easily computed while its children are processed, since the intervals for a node are the union of the intervals for its K^2 children. Using this property, we can easily rearrange K^2 siblings if at least one of them is an internal node. For each sibling node we have computed the intervals of the bitmaps T, T' and L that store the node and its descendants. The intervals are arranged according to the previous permutation, moving the bit sequences in the corresponding intervals to their new positions. This operation is performed independently for each level of the conceptual tree, permuting the bit sequences in the intervals for each level. The intervals for the parent node of the K^2 sibling nodes just processed is computed as the union of the intervals in the children nodes. After the rotation operation has finished (i.e. the conceptual root node has been rotated), we must recompute the *rank* directories. If we use a compressed version of L the rotation of K^2 siblings is performed similarly, but rotating the submatrices in the vocabulary. Then, groups of K^2 codewords are rearranged using the same permutation explained. After the rotation is complete, the sequence of variable-length codes must be encoded again using DACs to provide direct access to the rotated representation.

Notice that, even though the rotation operation in the K^2-tree1 variants is relatively complex, a *rotated query* is performed with no penalty by our K^2-tree1 representations: if a query requires us to compute results over a rotated representation, we can use the same traversal algorithms and support the same operations, if we consider the appropriate permutation of the children of each node during traversal. The same occurs with complement queries: we can use any of our variants as a *complemented* variant, returning regions of 0s instead of regions of 1s, without affecting the query algorithms.

5.6 Experimental evaluation

5.6.1 Experimental framework

In this section we will test the efficiency of our proposals for the representation of different kinds of binary matrices. As we have seen earlier in this chapter, our encodings are expected to obtain good compression results in images that are highly clustered, and their results depend heavily on this property of the datasets. In order to test the efficiency of our representations, we will provide two sets of test cases: the first group will be a collection of Web graphs, that have some level of clusterization but are very sparse, so the number and size of the regions of ones found in them should be small. Table 5.8 shows some basic information about the Web graphs used. All the datasets are publicly available datasets provided by the Laboratory for Web Algorithmics (LAW)[7]. Since several variants of each dataset are provided, we use the representations where the nodes are in natural order, that is, the rows/columns of

[7] `http://law.di.unimi.it/datasets.php`

the adjacency matrix correspond to Web pages in lexicographical order according to their URL. The first two datasets are small graphs according to current Web graph standards, while the latter two datasets can be considered good representatives of a Web graph.

Dataset	#Rows	#Columns	#Ones	% Ones
cnr-2000	325,557	325,557	3,216,152	0.0030
eu-2005	862,664	862,664	19,235,140	0.0026
indochina-2004	7,414,186	7,414,186	194,109,311	0.0004
uk-2002	18,520,486	18,520,486	298,113,762	0.0001

Table 5.8: Web graphs used to measure the compression of ones.

To measure the compression capabilities of our proposals in a more suited scenario, we also use a collection of raster datasets corresponding to elevation data in several regions of Spain. All the datasets are taken from the Digital Land Model (Modelo Digital del Terreno MDT05) of the Spanish Geographic Institute[8], which contains altitude information of Spain with 5 meter resolution. We take several fragments of the model, shown in Table 5.9. The first datasets are numbered after the corresponding pieces in the MDT dataset. Dataset *mdt–A* is a combination of four different fragments in a single raster, and finally *mdt–B* contains the elevation data for the complete Galician region. From these datasets, that are actually grids of values, we will build different binary matrices with different percentage of 1s for our experiments. To build the binary matrices we process the complete raster dataset and sort the cells by value. Then, we determine the threshold value that is at percentile $p\%$ in the sorted list of values, where p is the percentage of 1s desired. We use this threshold value to build the binary matrix, that will contain 1 for all cells of the original dataset whose value is not greater than the threshold.

We run all the experiments in this chapter on an AMD-Phenom-II X4 955@3.2 GHz, with 8GB DDR2 RAM. The operating system is Ubuntu 12.04.1. All our implementations are written in C and compiled with gcc version 4.6.2 with full optimizations enabled.

5.6.2 Comparison of our proposals with original K^2-trees

In this section we compare our proposals with original K^2-trees for the representation of different kinds of binary matrices. First we will show how our proposals behave in matrices will small (or no) clusters of ones, where a classic K^2-tree should obtain better results. To do this, we build a representation of all the Web graphs in Table 5.8 with each of our encodings, as well as the equivalent classic

[8] `http://www.cnig.es`

Dataset	#Rows	#Columns
mdt–200	3,881	5,461
mdt–400	3,921	5,761
mdt–500	4,001	5,841
mdt–600	3,961	5,881
mdt–700	3,841	5,841
mdt–900	3,961	6,041
mdt–A	7,721	11,081
mdt–B	47,050	48,266

Table 5.9: Raster datasets.

K^2-tree representation (*base*). For all the Web graph datasets we use the hybrid K^2-tree approach with 2 different values of K, $K = 4$ in the first three levels of decomposition and $K = 2$ in the lower levels. The same values of K are used in the original K^2-tree and all our variants.

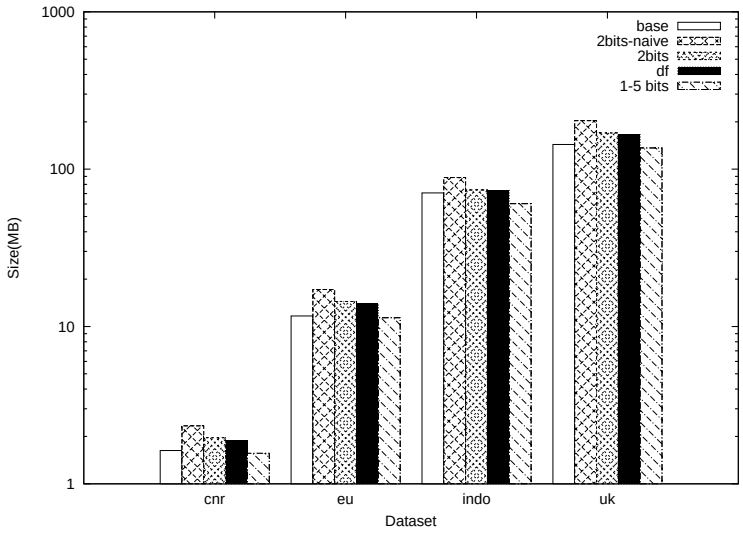


Figure 5.8: Space results of K^2-tree variations for Web graphs.

Figure 5.8 shows the space results for all the variants and the original K^2-tree in the studied Web graphs. The comparison results are very consistent among all the datasets, showing that our encoding based on 2 bitmaps obtain consistently worse compression in Web graphs than the baseline. As we said, this is expected since

all the Web graph datasets are very sparse and their adjacency matrices contain only small clusters of ones. As expected from the theoretical analysis, the K^2-tree1$^{2\text{bits}-\text{naive}}$ obtains the worst compression in all cases, being considerably larger than the baseline and all other encodings (notice the logarithmic scale in the y-axis). The K^2-tree1$^{2\text{bits}}$ and K^2-tree1$^{\text{df}}$ are far more competitive, and in all the studied datasets the K^2-tree1$^{\text{df}}$ is smaller than the K^2-tree1$^{2\text{bits}}$, even if slightly. Finally, and as expected from theoretical analysis, the K^2-tree1$^{1-5\text{bits}}$ obtains compression results very similar to the baseline, being smaller than the baseline in all the datasets and particularly in the *indochina* dataset. As we can see, the K^2-tree1$^{1-5\text{bits}}$ is able to take advantage of the small clusters of ones in the adjacency matrices of Web graphs and provide better compression than the baseline even though the graphs are very sparse.

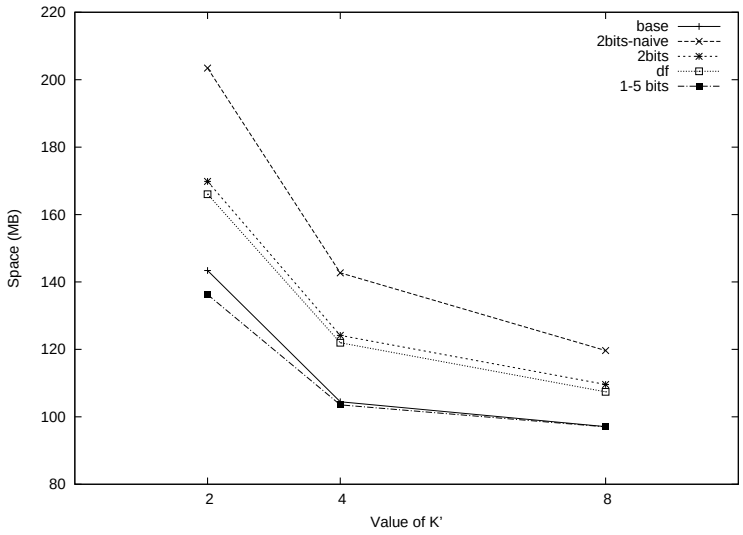


Figure 5.9: Space results for dataset *uk* for different leaf (K') sizes.

In the previous comparison we intentionally ignored the compression of submatrices in the lower levels of the K^2-tree, an enhancement that leads to much better compression in Web graphs. This makes the results in Figure 5.8 a good comparison between our approaches, but does not give the best estimation of the results that can be obtained by our encodings in Web graphs, since the compression of L significantly reduces the space requirements of K^2-trees. In order to properly test the efficiency of our encodings applied to Web graphs, we compare our approaches with original K^2-trees compressing the bitmap L in all the proposals. We measure the space requirements of all the proposals as K', the size of the submatrices in the last level of the tree, increases. In Figure 5.9 we show the results for the dataset *uk*, having

submatrices of 2×2, 4×4 and 8×8 in the last level. It is easy to see that the K^2-tree1$^{1-5\text{bits}}$ outperforms the original K^2-tree, but the difference becomes much smaller as K' increases. This is due to the fact that the clusters of ones are relatively small, so if we use 8×8 leaves most of the clusters are contained in a few leaves and further compression reduces at most a tiny fraction of the overall space.

5.6.2.1 Compression of clustered binary images

Next we test the efficiency of our representations using heavily clustered datasets, in which our representations should obtain their best compression results. To do this, we compare again all our encodings with the original K^2-trees, using in this case the datasets of Table 5.9. To obtain an estimation of the compression achieved by our proposals, we first test them using images with 50% of ones. We build a binary image with 50% of ones from each of the raster datasets filtering out the points with elevation above the median. To build the different approaches we use a hybrid variant with $K = 4$ in the first level of decomposition and $K = 2$ in the remaining ones, and 4×4 leaves for better compression. The results obtained are shown in Figure 5.10. As expected, all our encodings are significantly smaller than the baseline. Focusing on the comparison between our proposals, the K^2-tree1$^{2\text{bits}-\text{naive}}$ obtains again the worst compression results, and the K^2-tree1$^{1-5\text{bits}}$ is just slightly smaller than the K^2-tree1$^{2\text{bits}-\text{naive}}$. The K^2-tree1$^{2\text{bits}}$ obtains consistently the best compression results, but the K^2-tree1$^{\text{df}}$ is very close. Even if the results are very close among all proposals, the K^2-tree1$^{2\text{bits}}$ approach consistently obtains the best results among our proposals.

Notice that, as we explained in Section 5.2, the K^2-tree1$^{\text{df}}$ and K^2-tree1$^{2\text{bits}-\text{naive}}$ approaches are not symmetric, and they can take advantage of a skewed distribution of zeros and ones in the base matrix. In order to test the evolution of the different approaches depending on the number of ones, we analyze the space results obtained by all our proposals applying different thresholds to the raster datasets to obtain binary images with a 1-90% of black pixels (i.e. 1-90% of ones in the binary matrix).

In Figure 5.11 we show the results obtained for the dataset *mdt–*400, although similar results can be obtained in the other datasets. In the top of the figure we show the complete results of our approaches and the baseline; in the bottom we show a more detailed view of the results of our encodings, that, as we have already seen, obtain much better compression than the original K^2-tree. As the figure shows, the results are very similar regardless of the black analogy, with the K^2-tree1$^{2\text{bits}}$ obtaining the best compression results of all our approaches and the K^2-tree1$^{2\text{bits}-\text{naive}}$ obtaining the worst results. The most relevant result is the evolution of the K^2-tree1$^{1-5\text{bits}}$, that is always between the previous encodings but is less competitive when the percentage of ones is around 50%. It is noteworthy that even when the percentage of ones is small the K^2-tree1$^{1-5\text{bits}}$ obtains slightly worse results than the best representation, the K^2-tree1$^{2\text{bits}}$. This result is due to

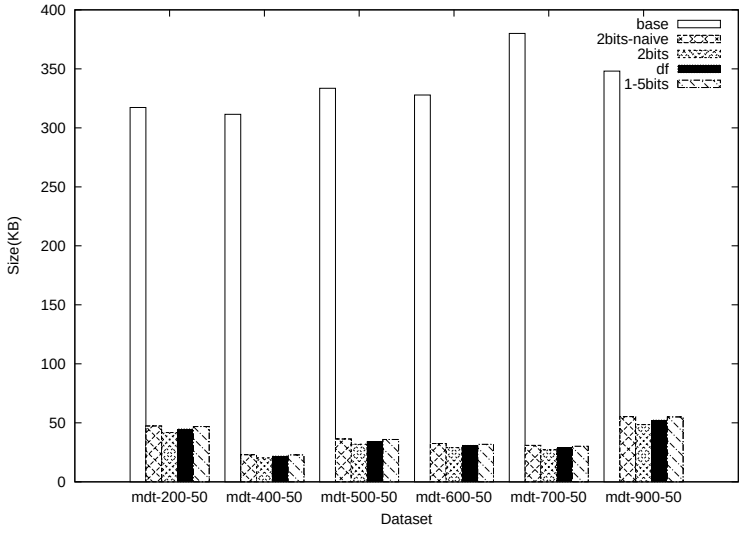


Figure 5.10: Space results of K^2-tree variations for raster datasets with 50% of 1s.

the nature of the datasets: independently of the number of ones in the matrix, they will be so highly clustered that the K^2-tree decomposition produces a relatively high number of black nodes in upper levels of the tree. Therefore, the K^2-tree$1^{1-5\text{bits}}$ and K^2-tree1^{df} representations will obtain worse results than the K^2-tree$1^{2\text{bits}}$ due to the additional cost to represent black nodes in the tree.

5.6.2.2 Query times

Finally, after demonstrating the efficiency of our representations to compress binary matrices by exploiting the clusterization of large regions of ones and zeros, we experimentally test their query efficiency. We compare our encodings with original K^2-trees when applied to the two main domains studied: the representation of Web graphs and the representation of binary raster images. In each domain, and for each dataset, we run a set of 10 million cell retrieval queries (that is, each query asks for the value of a single cell). The query sets are built selecting random cells within the boundaries of each dataset.

First we test the efficiency of our encodings in comparison with original K^2-trees to represent Web graphs. As we have already seen, our encodings are not well suited for the compression of such sparse matrices, since Web graphs contain few and relatively small clusters of ones. Therefore, when applied to Web graph datasets our encodings do not reduce the number of nodes in the conceptual tree

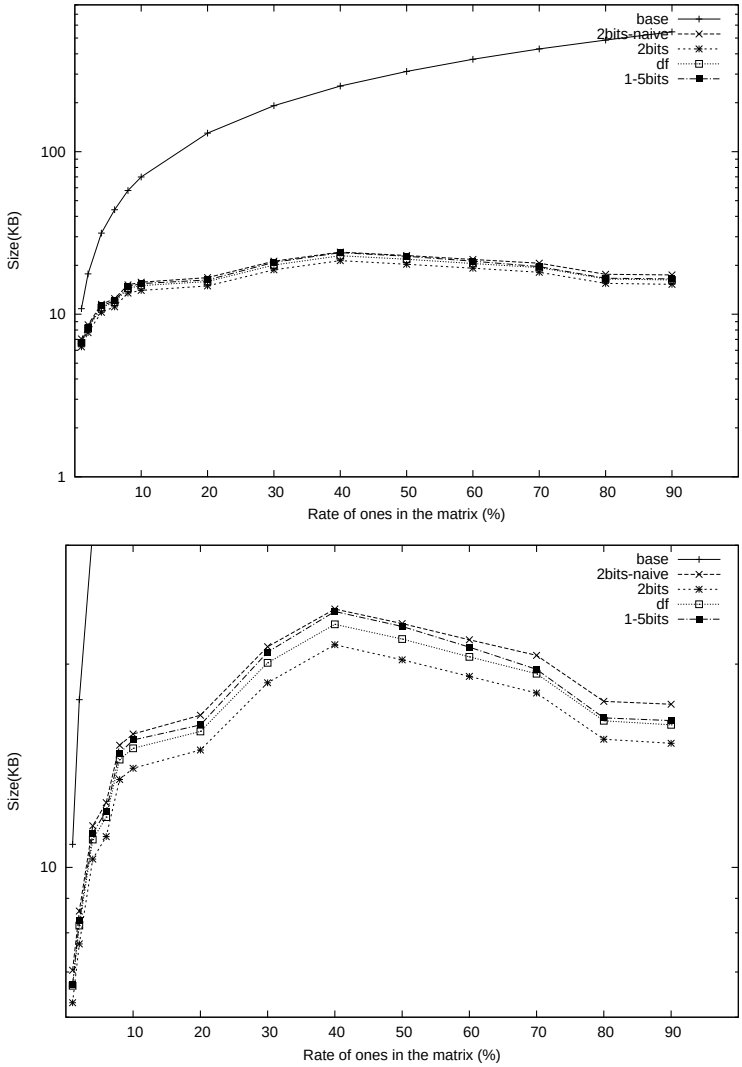


Figure 5.11: Space results of K^2-tree and K^2-tree1 variants for the dataset mdt-400 with varying % of 1s. Complete results are shown in the top plot and the detail of K^2-tree1 variants in the bottom plot.

and require essentially the same number of navigation steps over the conceptual tree.

Figure 5.12 shows the average query times for the different Web graph datasets

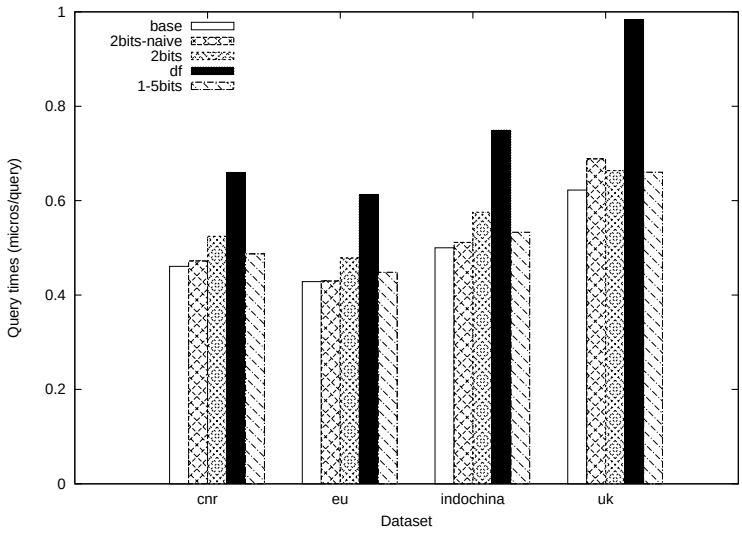


Figure 5.12: Query times to access a cell in different Web graph datasets using all our encodings (times in μs/query).

studied. All our encodings, as expected, obtain higher query times than original K^2-trees, due to the additional computations required to keep track of the "color" of tree nodes. The K^2-tree1$^{2\text{bits}-\text{naive}}$ obtains the best query times of all our approaches, since the cost of checking the color of a leaf node is simply a bitmap access. The K^2-tree1$^{1-5\text{bits}}$ also obtains good query times in all the Web graphs, because the main overhead in this approach is the computation to differentiate between gray and black nodes, so for many queries that reach a white node in upper levels of the conceptual tree the overhead is very small. Recall that the K^2-tree1$^{1-5\text{bits}}$ was also the only approach that was always competitive in space with original K^2-trees, always overcoming them by a tiny fraction. The K^2-tree1$^{1-5\text{bits}}$ combines space results, that are guaranteed to never be worse than original K^2-trees, with relatively close query times, making it a good alternative for the representation of graphs where the level of clusterization is not known or is very irregular. The K^2-tree1$^{2\text{bits}}$ and K^2-tree1$^{\text{df}}$ encodings obtain the worst query times, due to the additional *rank* operations required to navigate the conceptual tree, and the K^2-tree1$^{\text{df}}$ is significantly slower than all our other encodings. This is due to the fact that in a typical traversal we need to traverse many internal nodes until we finally reach the leaves, and the K^2-tree1$^{\text{df}}$ requires an additional *rank* operation in both internal nodes and leaves, while the K^2-tree1$^{2\text{bits}}$ for example only requires and additional rank operation at the leaves of the conceptual tree, to distinguish between black and white nodes.

Next we present a comparison of our representations in a better suited scenario: the compression of highly clustered raster images. We build two sets of binary images, the first with 50% of black pixels and the second with 10% black pixels, based on the usual raster datasets. For each dataset we build a set of 10 million cell retrieval queries, corresponding to random cells within the dimensions of each dataset. Figure 5.13 shows the results obtained by all our representations and original K^2-trees in the different datasets. Focusing on the top of the figure, corresponding to images with 50% black pixels, we can see that the results in raster datasets are significantly different from the comparison in Web graphs: most of our approaches are faster than original K^2-trees, and the relative comparison between them is also changed. The fact that most of our encodings are faster than original K^2-trees is due to the reduction in the number of nodes in the conceptual tree: in original K^2-trees, all queries that ask for a cell that is set to 1 need to traverse the complete tree; on the other hand, in our representations most of these queries are stopped in upper levels of the conceptual tree when a uniform region of ones appears in the matrix. The reduction in the average height of the tree is enough to compensate the additional cost of bitmap accesses and rank operations in most cases. The K^2-tree1$^{2\text{bits}-\text{naive}}$ is again the fastest approach but the K^2-tree1$^{2\text{bits}}$ and K^2-tree1$^{1-5\text{bits}}$ also obtain close query times. We consider of particular interest the comparison between the K^2-tree1$^{2\text{bits}-\text{naive}}$ and the K^2-tree1$^{2\text{bits}}$, that provides a space-time tradeoff in this kind of data: the K^2-tree1$^{2\text{bits}-\text{naive}}$ is a larger representation but is in general faster than the K^2-tree1$^{2\text{bits}}$; depending on the final application of the representation, they can be easily interchangeable to provide slightly faster access or a smaller data structure.

Figure 5.13 (bottom) shows the query times obtained for the same query sets in datasets with 10% black pixels. If we compare the results in the top and bottom plots of the figure, it is easy to see that the query efficiency of our encodings depends heavily on the existence of large regions of zeros or ones in the image that can reduce the average height of the conceptual tree. With 10% black pixels, original K^2-trees become competitive in query times with most of our representations (even if they are far from competitive in space, as we have shown in Figure 5.11). As the number of ones reduces, so does the size of the black regions in the datasets, improving the efficiency of the K^2-tree1$^{1-5\text{bits}}$ representation that becomes slightly faster than the K^2-tree1$^{2\text{bits}}$. The K^2-tree1$^{2\text{bits}-\text{naive}}$ is still the fastest approach, and the K^2-tree1$^{\text{df}}$ is still significantly slower in all cases.

5.6.3 Comparison with linear quadtrees

In the theoretical analysis of our proposals we have shown that they are competitive in space with some of the most compact quadtree representations, and also provide efficient access to regions of the represented matrix/image. In this section we demonstrate the capabilities of our representations comparing our best overall encoding in highly clustered data, the K^2-tree1$^{2\text{bits}}$ (it obtained the best

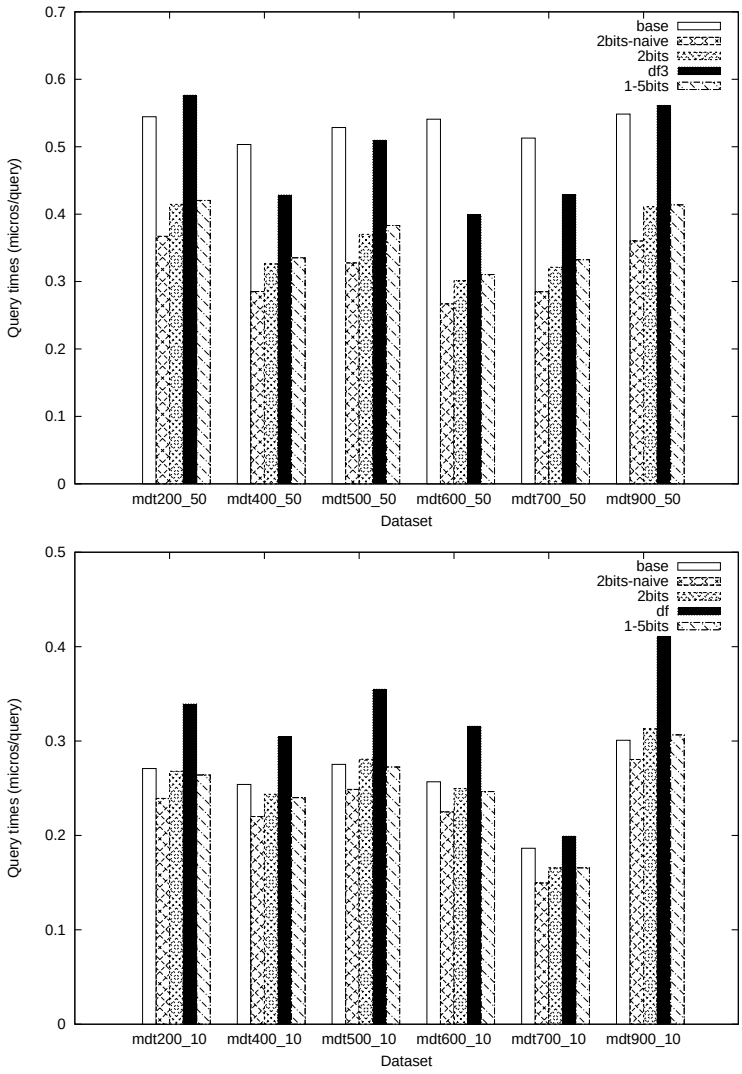


Figure 5.13: Query times to access a cell in raster datasets with 50% (top) or 10% (bottom) black pixels(times in μs/query).

compression in general, even though its query times were a bit slower than the K^2-tree1$^{\text{2bits}-\text{naive}}$), with the linear quadtree representation proposed by Gargantini, a widely used compact quadtree representation that provides efficient support for most operations, including random access to pixels or regions of the image as well as image

manipulations (set operations, rotations, etc.). Since the linear quadtree (LQT) is designed to operate in external memory, we implemented an in-memory version of the LQT that provides a fairer comparison with our K^2-tree1 representation. Our in-memory implementation, that we call LQT-array, stores the sequence of quadcodes (represented using FL encoding) in an array in main memory, so searches for quadcodes can be implemented using a simple binary search on the sequence of quadcodes.

We perform our comparison using a subset of the raster datasets and some Web graphs in order to measure the differences in different kinds of data. As in the previous sections, the raster datasets ($mdt-$) correspond to the expected use case of our encodings, matrices with relatively large clusters of zeros and ones. On the other hand, we also experiment with Web graph datasets to provide a baseline for comparison. In all cases, we compare our K^2-tree representation with the equivalent LQT-array and measure the compression obtained in bits per one of the binary matrix.

Table 5.10 shows some basic information about the datasets used, including the size of the matrix and the number of ones in it. For the raster datasets, in these experiments we build a binary matrix for each of them with roughly a 50% of ones. The last two columns of the table show the space results obtained by our encoding and the equivalent linear quadtree. As expected, our representation is always much smaller than the linear quadtree: in all the raster datasets and Web graphs studied, our encodings are approximately 10 times smaller than a linear quadtree.

Dataset	rows × cols	#ones	K^2-tree	LQT-array
mdt-600	3961×5881	11,647,287	0.02	0.25
mdt-700	3841×5841	13,732,734	0.02	0.17
mdt-A	11081×7821	50,416,771	0.01	0.22
cnr	325,557×325,557	3,216,152	3.14	41.32
eu	862,664×862,664	19,235,140	3.81	49.92

Table 5.10: Space utilization of K^2-trees and LQTs (in bits per one).

The space results confirm that our representations are indeed much smaller than the linear quadtree, as expected from the theoretical analysis. Next we compare the efficiency of our representations performing random queries against the different datasets. For each dataset we executed a set of queries asking for the value of a cell, selecting the cell uniformly at random within the matrix in each query. We build a different set of random queries for each dataset, according to its dimensions, and execute the same set of queries in our representation and the LQT-array representation. In our case, the query in the K^2-tree$1^{2\text{bits}}$ is implemented with a simple traversal of the tree that returns immediately when a leaf node is

found. In the LQT-array implementation, the query is implemented with simple binary searches for the corresponding quadcode in the array. The results of our experiments are shown in Table 5.11. The query times, in μs/query, show that our representation is not only smaller but also faster than a simple LQT representation of the data. In all the examples our encodings are on average roughly 3 times faster than the LQT-based approach to access a cell of the raster.

Dataset	K^2-tree	LQT-array
mdt-600	0.25	0.84
mdt-700	0.28	0.88
mdt-A	0.26	0.98
cnr	0.77	2.08
eu	1.10	2.62

Table 5.11: Time to retrieve the value of a cell of the binary matrix, in μs/query.

5.7 Other applications of our encodings

An interesting point of our encodings is their ability to maintain the navigational properties of the classic K^2-tree while adding more information to the conceptual tree represented. Even though our representations have been proposed for the compact representation of regions of ones, we believe they provide the basis for multiple enhancements of the K^2-tree. The new encodings allow us to distinguish 3 different kinds of nodes in the K^2-tree without destroying its navigational properties. Using a K^2-tree1$^{\text{2bits}-\text{naive}}$, we could also build a tree with 4 different kinds of nodes.

Essentially, our encoding techniques provide the basis to add information to a K^2-tree while maintaining its navigational properties. The K^2-tree1$^{\text{2bits}-\text{naive}}$ provides the simplest method to mark "special" nodes in a K^2-tree without affecting the structure and with a guaranteed upper bound in space requirements determined by the K^2-tree size. However, the encodings based on using an additional bitmap provide better support to not only mark special nodes but also associate additional information to them. Next we introduce a technique to improve compression in K^2-trees by shortening some paths of the conceptual tree, called "unary paths", that contain a single 1 of the matrix. In this technique, a long path is replaced by a special leaf node and it is compressed in a separate data structure. A similar technique can be used to add any information to some nodes of a K^2-tree and retrieve that information efficiently.

Compression of unary paths: Consider the classic K^2-tree representation of a sparse binary matrix, as explained in Section 3.2. The K^2-tree is able to compress efficiently large regions of 0s, and takes advantage of the clusterization of 1s in the adjacency matrix to obtain the best compression. However, if we consider a very sparse binary matrix where the 1s are not really clustered, the K^2-tree obtains its worst compression results: for each 1 in the binary matrix, we need to encode a complete path from the root of the conceptual tree to the cell, thus storing $K^2 \times \log_K n$ bits to represent a single cell in a matrix of size $n \times n$. We call these paths *unary paths*, since they are formed by K^2 nodes in multiple levels to represent a single cell in the matrix. The K^2-tree decomposition becomes very inefficient if there are many unary paths. A unary path of length l (i.e. a unary path that spans l levels of the conceptual tree) represents the cell using space $K^2 \log_K l$ bits instead of the $K \log_K l$ bits required to simply address the cell coordinates.

An optimization that can be added to the K^2-tree to enhance its compression is to remove unary paths from the representation and store them separately. This requires a special encoding of the K^2-tree nodes: a leaf node in the modified K^2-tree can correspond to a region full of 0s or to a unary path. We can use our encodings to encode this information in a K^2-tree. Then we can encode the sequence of unary paths in the tree in a different data structures using $K \log_K l$ bits for each path. If the unary paths at each level of the conceptual tree (i.e. the unary paths with equal length l) are sorted in a separate list $paths_l$, we can index the list easily since all elements have the same size. To know the index of the current unary path in the corresponding list we can use the K^2-tree$1^{2\text{bits}}$ encoding (where each unary path is represented using a 0 in T and a 1 in T'). First, we store the number of unary paths before each level of the conceptual tree in a separate array $unary[1, \log_K n]$. In this array, $unary[1] = 0$ and $unary[i] = \Sigma_{j=1}^{i-1} unary[j] + nPaths[i]$, where $nPaths[i]$ is the number of unary paths at level i. Since each unary path is stored in T' with a bit 1, and the nodes are stored levelwise, we can compute the offset of a unary path easily. If we find a unary path at position p in T, p' in T', corresponding to level l in the K^2-tree, we know it will be stored in the list $paths_l$, at offset $\text{rank}_1(T', p') - unary[l]$. Adding *rank* support to T' we can perform the operation very efficiently, and access immediately any unary path found during traversal.

This approach for the compression of unary paths may not be significant in most applications of the K^2-tree, due to the additional space required to encode leaves of the tree. As we will see, the same ideas presented here can be applied in higher dimensionality problems, and the effect of compression of unary paths may be much more significant in these problems[9].

[9]The compression of unary paths has been implemented and is currently being used to develop new data structures for the compression of temporal graphs in Diego Caro's ongoing PhD work, in combination with another of our proposals, the K^n-tree, an extension of the K^2-tree to multidimensional data that will be introduced in Chapter 6.

5.8 Summary

At this point, we have shown that our representations are able to compress very efficiently binary matrices with large clusters of zeros and ones, and we provide a set of efficient algorithms to query the data in compressed form. Our theoretical analysis and experimental evaluation of the different K^2-tree1 variants show that they obtain space results competitive with the state-of-the-art quadtree representations in different datasets with high clusterization, while providing efficient support for selective access to regions of the image over the compressed in-memory representation. We compare the compression and query times of our proposals with the well-known linear quadtree, showing that the K^2-tree1 can be both smaller and faster than the linear quadtree when representing clustered binary images. Additionally, we have introduced a number of different operations like set queries that are efficiently solved by K^2-tree1 variants. We have also presented some other applications of our encodings to improve original K^2-trees with additional functionality, particularly the compression of unary paths that is useful to compress very sparse matrices.

In order to fully demonstrate the efficiency of our representations for the general representation of binary images, some other points will be studied later in this thesis. Firstly, all the encodings presented here are based on an essentially static data structure. We have introduced some algorithms to perform some manipulations on top of our compressed representations, but the static nature of K^2-trees limits their applicability to domains, such as GIS, where some parts of the images may be modified dynamically. In Part II we will introduce a dynamic alternative to the K^2-tree, and we will also show that all the encodings developed in this chapter can be applied to the dynamic representation as well. We will also expand on the current experimentation to demonstrate that a dynamic version of the encodings introduced in this chapter is also competitive with alternative quadtree representations in terms of space and query times.

Our variants of K^2-tree1 can also be of application in domains where data is not binary, as long as a clusterization of similar values still exists. In following chapters of this thesis we will study the representation of multidimensional grids and grids of integers, and in each of these problems we will provide a solution based on the K^2-tree1 to take advantage of the clusterization of values.

We will also study the representation of raster data, a specific area of application where clusterization usually appears. In Part III we will study different applications of our variants to the representation of raster images and temporal or time-evolving raster data. The simplest of our proposals to represent this kind of data will be the storage of a simple collection of K^2-trees representing multiple equal-sized submatrices corresponding to each different value or time instant, where each K^2-tree will be actually based on our new encodings.

Chapter 6

Representing multidimensional relations with the K^2-tree

The extension of spatial data structures to high-dimensionality problems has been studied for most of the existing representations. As we have shown in Section 4.3, many of the spatial representations are directly designed to handle multidimensional data, while others are originally designed for bi-dimensional space but later extended to manage multidimensional data. The k-d-tree [Ben75] and the R-tree [Gut84] are examples of data structures that are conceived in general for the representation of n-dimensional data. On the other hand, the quadtree and its variants have been proposed for 2-dimensional data, but they can be extended to n-dimensional problems: the octtree is a straightforward generalization of the quadtree to 3-dimensional data, and the same name is used for a general extension to represent n-dimensional data.

In this chapter we present a new data structure for the representation of n-dimensional binary matrices. Our proposal is an extension of the K^2-tree, explained in Section 3.2, for the representation of n-ary relations. We call our proposal K^n-tree. The K^n-tree is based on a generalization of the partitioning algorithm that maintains the simple navigation operations of the K^2-tree and provides a fully-indexed representation of n-ary relations. Our representation efficiently supports queries that involve constraints in all the dimensions, including fixing a value or range of values in any of them. We also introduce another data structure, the IK^2-tree, that was originally proposed for the compact representation of RDF graphs[1].

[1]The IK^2-tree was first devised to improve the representation of RDF databases based on classic K^2-trees explained in Section 4.2. The IK^2-tree data structure, along with some variants and its applications to RDF databases and temporal graphs, was first published as a result of joint

In this thesis we will propose new applications of the IK^2-tree for the representation of data in specific domains. Particularly, in this chapter we will show how to use a K^n-tree and an IK^2-tree for the compact representation of temporal graphs, as a proof of concept of the applicability of both data structures and differences between them.

6.1 Representation of ternary relations as a collection of binary relations

A simple method to store a ternary relation is to reduce the problem to a collection of binary relations. Given a ternary relation $R \subseteq X \times Y \times Z$, we can transform it into a set of $|Y|$ binary relations R_y, one for each different value $y \in Y$. Following this approach, each $(x, y, z) \in R$ becomes a pair (x, z) in R_y. Let us denote the dimension Y *partitioning dimension* or *partitioning variable*. The decomposition of a ternary relation in multiple binary relations simplifies the problem of storing the original data, that can now be stored efficiently using any representation for binary relations.

6.1.1 The MK^2-tree

This approach of partitioning a ternary relation into a collection of binary relation has been studied to take advantage of the functionalities of K^2-trees. This has been applied to represent RDF graphs using K^2-trees (recall Section 4.2), considering the RDF graph as a collection of unlabeled graphs and representing each of them using a K^2-tree. We call this simple approach, based on multiple K^2-trees, MK^2-tree.

The MK^2-tree of a ternary relation $R \subseteq X \times Y \times Z$, where Y is the partitioning dimension, is simply defined as a collection of K^2-tree data structures K_y that represent each induced binary relation R_y. Each K_y is built over the adjacency matrix of its corresponding binary relation, that will contain a 1 for all positions (x, z) such that $(x, y, z) \in R$.

The utilization of multiple K^2-trees to represent a ternary relation provides a method to efficiently answer queries that are restricted to a single value in the partitioning dimension (*fixed-value partitioning dimension* queries, for example finding all the pairs (x, z) for a fixed $y = 3$): these queries are answered accessing a single K^2-tree.

The main drawback of the MK^2-tree, caused by the decomposition of a ternary relation in multiple binary relations, is the fact that indexing capabilities in the partitioning dimension are usually reduced. When our query is not restricted in the partitioning dimension (*unbounded partitioning dimension* queries), we must

work with Sandra Álvarez-García [ÁGBdBN14]. An extensive description of the data structure and its applications in RDF datasets is part of Álvarez-García's PhD thesis [ÁG14].

query multiple K^2-trees to obtain the final results. For example, to find all the values with $y \in [1, 10]$ we must access all the K_ys corresponding to values in $[1, 10]$ independently to obtain the final results. This causes the efficiency of these queries to degrade significantly when the partitioning dimension is large.

6.1.2 The IK^2-tree

The IK^2-tree [ÁGBdBN14] was originally devised as an improvement of the MK^2-tree, particularly oriented to solve the problem of the representation of RDF databases. Given the decomposition of a ternary relation into $|Y|$ adjacency matrices, the IK^2-tree represents all those matrices simultaneously, "merging" the multiple K^2-trees used in the MK^2-tree in a single data structure. Instead of using a single bit to know whether the matrix is empty or not, the IK^2-tree uses a variable number of bits.

Figure 6.1 shows an example of IK^2-tree construction, where the partitioning variable Y can take three different values. The top of the figure shows the three adjacency matrices that represent the different R_y, where the black cells correspond to valid relations. Below each matrix we show the conceptual K^2-tree representation of the matrix, and in the bottom of the figure the IK^2-tree representation of the complete collection. The IK^2-tree representation is a reorganization of the multiple K^2-trees into a single data structure, created merging the equivalent branches in all of them. Figure 6.1 shows this process visually, highlighting the bits of each K^2-tree in a different color. The IK^2-tree representation, shown below the K^2-trees, contains all the branches in the different K^2-trees, but each node stores all the bits corresponding to the equivalent nodes in all the K^2-trees that contain that branch. For example, node labeled N_0 contains 3 bits corresponding to the 3 K^2-trees, but node N_2 contains only 2 bits, because only the second and third K^2-trees contain that branch.

Like a single K^2-tree, the IK^2-tree is stored in two bitmaps T and L, that are built in a levelwise traversal of the tree. The bitmap representation of the IK^2-tree shown at the bottom of Figure 6.1 contains all the bits of the IK^2-tree. As we can see, the final bitmap representation contains the bits of the different K^2-trees for each matrix, but they are interleaved in the bitmap so that the bits of all K^2-trees for the same node are stored together in the IK^2-tree. The bitmaps T and L are the actual data structure used to store the tree.

The IK^2-tree representation has some interesting properties derived from its construction:

- An IK^2-tree uses the same number of bits that the equivalent representation based on multiple K^2-trees: as we can see in Figure 6.1, the IK^2-tree can be built rearranging the bits of the individual K^2-trees.
- Even if nodes in different branches may have a different number of bits, nodes belonging to a group of K^2 siblings always have the same number of bits. This

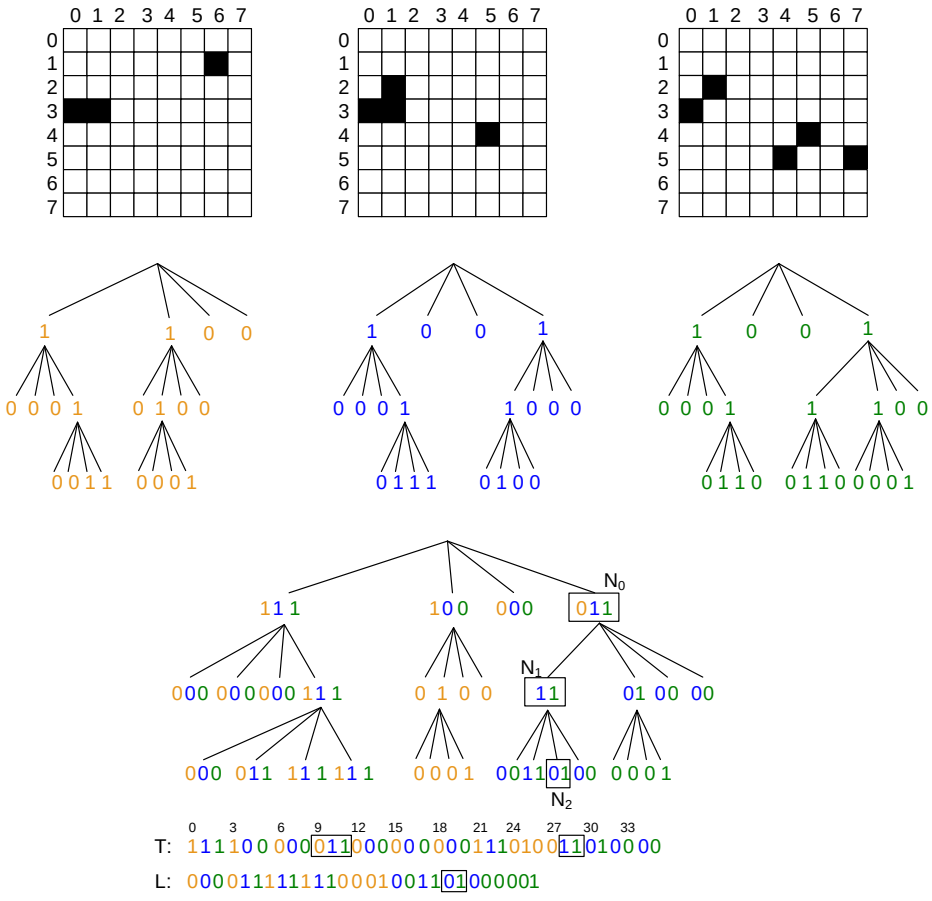


Figure 6.1: Representation of a ternary relation using MK^2-tree and IK^2-tree.

is because all K^2-tree nodes have either 0 or K^2 children.

- Another property, important for navigation, is that each 1 in a level generates K^2 bits in the next level (even if the K^2 bits are not consecutive now, but interleaved with bits from other K^2-trees), and a 0 in a level does not generate any bits in the next level. In the first level of decomposition, all K^2 nodes contain m bits, where $m = |Y|$. Moreover, only nodes that contain at least a 1 have children, and the number of 1s in a node determines the number of bits in each of its K^2 children. These properties allow us to perform basic navigation over the bitmap representation:
  - In the first level of decomposition, we have K^2 nodes of m bits each. To determine if a node has children we need to check if there is at least a 1 among its m bits.
  - Given a node that contains b bits starting at offset p in T (i.e., the node covers positions $[p, p + b - 1]$), its children will begin at position $(\text{rank}_1(T, p - 1) + m) \times K^2$, where m (the number of matrices) is a correction factor. Each child will have o bits, where $o = \text{rank}_1(T, p + b - 1) - \text{rank}_1(T, p - 1)$ is the number of 1s among the bits of the current node.

These properties provide the basis to navigate the conceptual IK^2-tree. Next we will show how to answer queries using the basic navigation operations.

6.1.3 Query support

The IK^2-tree lacks some filtering capabilities in the partitioning dimension, but is more efficient in practice than the MK^2-tree in many queries, especially if the size of the partitioning dimension is large. In this section we will show how to answer queries in the IK^2-tree depending on the filter applied in the partitioning dimension.

Fixed-value partitioning dimension: When the partitioning dimension is fixed (a single value) we must perform in the IK^2-tree the equivalent of a query in a single K^2-tree. This is independent of the filters in the other dimensions, that only determine the branches that are explored. The key to answer these queries is simply to track the position in each node that corresponds to the desired value. Consider, for example, that we want to find the value of $(x = 4, y = 1, z = 5)$ in the example of Figure 6.1, where Y is the partitioning variable (that is, we want access row 4, column 5 in the second matrix). At the first level of decomposition, we go to the last quadrant (node N_0), that will be located at $T\,[|Y| \times 3, |Y| \times 4 - 1] = T[9..11]$. We check the second bit, located at $T[p + y] = T[9 + 1]$, and find that it is set to 1. Therefore, we need to continue traversal to its children. We compute $(\text{rank}_1(T, 9 - 1) + |Y|) \times K^2 = 28$ to locate the offset of its children, and $\text{rank}_1(T, p + m - 1) - \text{rank}_1(T, p - 1) = \text{rank}_1(T, 11) - \text{rank}_1(T, 8) = 2$ to determine the number of bits that each child will contain. Additionally, we need to keep track of the offset

corresponding to $y = 1$ in the children. This can be computed easily as the number of ones in N_0 prior to the current bit: $y' = \text{rank}_1(T, p+y-1) - \text{rank}_1(T, p-1) = 0$. Therefore, in the next level we will check offset 0 in the corresponding node. The process is repeated in all the nodes until we reach the last level of the conceptual tree.

Any other queries involving a fixed partitioning dimension are solved in the same way, applying the K^2-tree basic navigation techniques to answer queries involving ranges in the other two dimensions. Notice that queries with fixed Y may be significantly slower in an IK^2-tree representation than in the MK^2-tree, due to the additional *rank* operations required at each step during navigation.

Unbounded partitioning dimension: When we are interested in all the possible values for the partitioning dimension (unbounded partitioning dimension) we must check all the bits of each node to determine which of them are set to 1, and keep track of all of them during traversal to answer the query. In order to answer queries with unbounded partitioning dimension, we simply keep a list of "active" y's for each IK^2-tree node. The list of active values A is initialized with all the possible values in Y. In each node traversed, we check which of the bits are set to 1 and build a new list of active values containing only those elements. The list of active values A used in each node contains an entry per bit in the node, corresponding to the y value represented by the bit. Finally, when we reach the leaves of the tree, each bit set to 1 found in the node is mapped to a y value using the active list A.

Example 6.1: In the IK^2-tree of Figure 6.1 we want to access row 4, column 5 and retrieve all the values of y. At the first level of decomposition, we go to the last quadrant (node N_0). Our list of active matrices is $A = 0, 1, 2$. Since all nodes in this level have $m = 3$ bits, the fourth child will be located in $T[9..11]$. We now check all the bits in N_0, and find that only the second and third bit are set to one. This means that its K^2 children will have 2 bits, corresponding to the second and third matrices in A. We set $A = 1, 2$ and find the children of N_0, located at $p = 28$. We now access the first child N_1, whose bits are all set to 1, so A remains the same, and the children of N_1 will have 2 bits each. We compute the *rank* again to find the children of N_1 at position $(\text{rank}_1(T, 27) + 3) \times 4 = 13 \times 4 = 52$. Since $|T| = 36$, the children will begin at position $52 - 36$ in L. Finally, we want to access the third child in this level. We locate it in $L[20, 21]$, and check its bits: the first bit is set to 0, so we discard it. The second bit is set to 1, so the cell was active for the matrix at the second position in A. In our case, the second element in A was 2, so we can know that the cell was set to 1 only in matrix 2.

Like in fixed-y queries, the same procedure can be applied to row/column/range queries using the basic traversal techniques of the K^2-tree. In this type of queries the IK^2-tree is very efficient thanks to the fact that for each tree node the different y values for a node are stored in consecutive bits. Notice that in the MK^2-tree the same query would require $|Y|$ rank operations, one in each K^2-tree, to be completed. This difference makes the IK^2-tree an interesting approach to represent

ternary relations replacing the MK^2-tree, especially in domains where queries with unbounded Y are very frequent in comparison with queries with fixed partitioning variable.

Fixed-range partitioning dimension: The third type of query that may be of interest in a ternary relation involves a range of values in the partitioning dimension. These queries can be solved using a combination of the procedures explained for the fixed-value and unbounded case. We need to keep track of the offsets in the bitmap of each node that correspond to the query range. This can be easily computed using two *rank* operations at each node. Additionally, we need the same list of active values A to keep track of the y values that correspond to each bit in the node in order to properly answer the query. Like in the case of fixed-value queries, in the case of very short ranges the overhead required to perform navigation in the IK^2-tree may be higher than the costs of maintaining independent K^2-trees. However, as the length of the range increases the query in the IK^2-tree will become more efficient than an approach based on separate K^2-trees.

6.2 The K^n-tree

The K^n-tree is a generalization of the K^2-tree explained in Section 3.2 to higher dimensionality problems. The goal of a K^n-tree is to represent a binary n-dimensional matrix using the partitioning strategy and physical representation used in a K^2-tree while providing efficient access to the relation and supporting constraints in all the dimensions simultaneously.

Consider an n-dimensional matrix. Without loss of generality, we can assume that all its dimensions are of the same size s and that s is a power of K. If a matrix does not fulfill these conditions we can "virtually" expand it, considering that all dimensions are enlarged to size $s' = 2^{\lfloor \log s \rfloor}$, where s is the size of the largest dimension, and considering that all the added space is filled with 0s. The matrix is subdivided in K^n equal-sized submatrices as follows: for each of the dimensions, $K-1$ hyperplanes divide the matrix at positions $i\frac{s}{K}, i \in [1, K-1]$ across that dimension. After all the dimensions have been partitioned, K^n submatrices are induced by the hyperplanes, each of them of size $\frac{s}{K} \times \frac{s}{K}$. The submatrices can be numbered using their relative ordering in each dimension. In the general case, we can choose to take the submatrices in row-major order (start sorting by the first dimension) or column-major order (start sorting by the last dimension). If we consider $(v_1, v_2, \cdots, v_n)$, the relative position of the submatrix in each dimension, and we sort starting by the first dimension, the submatrices that share the same v_1 will be placed consecutively, while submatrices sharing v_n will be placed at every K^{n-1}-th position. If we start sorting by the last dimension, the submatrices that share the same v_n are consecutive, while the submatrices that share the same v_1 are isolated, separated by K^{n-1} positions. As we will see later, the method used to order the submatrices has little effect in the algorithms, so in practice any choice of

ordering of the K^n submatrices will be supported by the K^n-tree simply adjusting the basic operations.

Example 6.2: Consider a 3-dimensional matrix of size $s \times s \times s$, like the one shown in Figure 6.2. Notice the placement of the axis that determines the dimensions x, y, z. To partition the matrix we cross the matrix with the planes $x = i\frac{s}{K}$, $y = i\frac{s}{K}$, $z = i\frac{s}{K}$, for $i = 1..K-1$. K^3 submatrices are induced by these planes, with positions $(1,1,1)$ to (K,K,K). As for the ordering of the submatrices, we chose a left-to-right, top-to-bottom, front-to-back traversal of the matrix, therefore sorting first by z, then by y (row) and finally by x (column). The actual numbering of the submatrices for the usual case $K = 2$ is shown in the right part of Figure 6.2. Notice that in this example we choose the ordering so that the traversal is easy to understand visually, but in the general case the ordering of the dimensions can be arbitrary.

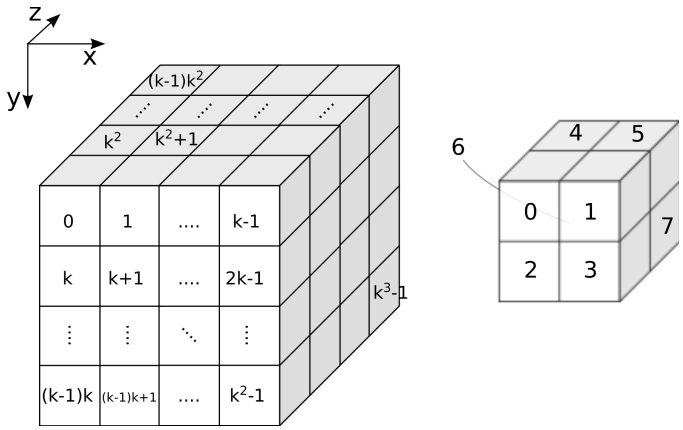


Figure 6.2: K^3-tree partition of a matrix: general case (any K) and typical partition with $K = 2$.

A K^n-tree is built from a binary matrix following the same recursive procedure used in the K^2-tree. A conceptual tree is created, whose root corresponds to the complete matrix. Then, the matrix is subdivided as explained, and each of the submatrices will become a child node of the root. This node will be labeled with a 1 if the corresponding submatrix contains at least a 1, or 0 otherwise. For all the nodes labeled with a 1, the decomposition process is repeated recursively until the cells of the matrix are reached.

Example 6.3: Figure 6.3 shows an example of K^3-tree representation for a small 3-dimensional matrix and $K = 2$. The original matrix, of size 8×8, is shown on the left side. The matrix is represented as an empty cube in which the cells with value 1 are highlighted in black. The root node of the conceptual tree (right) has 8 children, corresponding to the 8 submatrices of the partition following the order explained

in Figure 6.2. Only the second submatrix has at least a 1, so the decomposition only continues in that node. The decomposition proceeds, as shown in the matrix, until the individual cells are reached. The last level of the tree corresponds to all the $2 \times 2 \times 2$ submatrices that contain at least a 1.

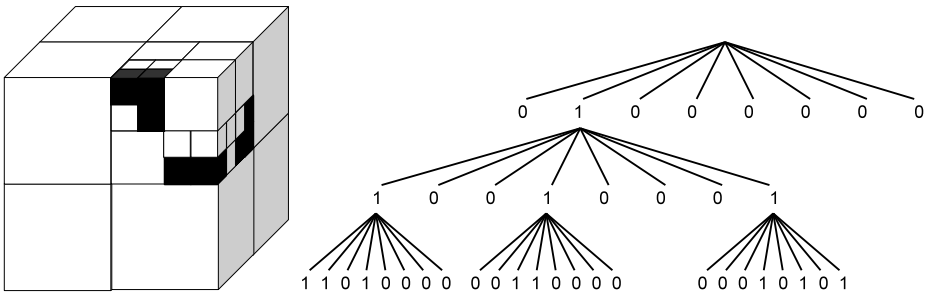


Figure 6.3: 3-dimensional matrix and its K^3-tree representation.

6.2.1 Data structure and algorithms

The conceptual K^n-tree is stored using the same bitmaps used in the K^2-tree: a bitmap T stores the bits in all the levels except the last one, and a bitmap L stores the values of the last level. The basic navigational property of the K^2-tree also holds with a small modification: each internal node will have exactly K^n children, so the children of the node at position p in T will be located at $p' = \text{rank}_1(T, p) \times K^n$.
Example 6.4: The bitmaps for the K^3-tree shown in Figure 6.3 are:

```
T: 01000000 10010001
L: 11010000 00110000 00010101
```

The children of the node at position 1 (we start numbering in 0) begin at position $p' = \text{rank}_1(T, 1) \times K^3 = 1 \times 8 = 8$. The children of the node at position 8 begin at position $p'' = \text{rank}_1(T, 8) \times K^3 = 16$. As $p'' \geq |T|$, they are actually located in $L[p'' - |T|]$.

The navigation needs to be adjusted to take into account the different dimensions. Each node of the K^n-tree will have exactly K^n children. The formula to determine the actual offset of a child depends on the numbering selected for the submatrices. If we order the submatrices by the dimensions in reverse order (from n to 1), then the offset for a submatrix at position $(v_1, v_2, \cdots, v_n)$ can be computed with the following formula:

$$offset_{ROW-MAJOR} = \Sigma_{i=1}^{n} v_i \times K^{i-1}$$

$$offset_{COLUMN-MAJOR} = \Sigma_{i=1}^{n} v_i \times K^{n-i}$$

Any other ordering of the dimensions can be translated into a similar formula that allows us to find the corresponding child given its partition in each dimension.

Using this generalization of the formulas, a K^n-tree supports advanced queries involving all its dimensions, extending the algorithms of the K^2-tree to work in multiple dimensions. Recall from Section 3.2 that the K^2-tree is able to answer unconstrained queries, queries that ask for a specific row or column or queries that involve a range of rows/columns. The K^n-tree supports the same constraints in all the dimensions represented at the same time.

6.2.1.1 Check the value of a cell

To answer this query, we need to determine at each step which of the submatrices contains the desired cell. Starting at the root, we check the coordinates of the cell $(c_1, \cdots, c_n)$ to find the submatrix that contains it. This simply requires to compute, for each c_i , the value $\frac{c_i}{s/K}$, where s is the size of the current submatrix (the total size at the root, divided by K at each level). With this we obtain a tuple $(v_1, \cdots, v_n), v \in [0..K-1]$ that we can use to find the position of the child, using the *offset* formula: $offset = v_1K^{n-1} + v_2K^{n-2} + \cdots + v_{n-1}K + v_n$. If the corresponding bit is set to 1, we find its children using the *rank* operation and repeat the process recursively. In the child node, the new c_i's will be updated to reflect offsets within the current block. To do this, we simply compute the new c_i as $c'_i = c_i \mod s/K$ and use the new values in the next level. The iteration is repeated until we find a 0 (the cell is a 0) or we reach a 1 in the last level (the cell is a 1).

Example 6.5: In the matrix of Figure 6.3 we want to check the value of the cell at coordinates $(6, 3, 0)$ (row 3, column 6, front layer). In the associated K^3-tree, we start at the root (position 0 in T). Dividing each component by $s/K = 4$, we find that the submatrix that contains the cell is the submatrix $(1, 0, 0)$, thus the child that we must check is at offset $1K^0 + 0K + 0K^2 = 1$. As we can see in T, shown in the previous example, the bit is a 1, so we keep traversing and divide s by K. The children of the node begin at position $\text{rank}_1(T, 1) \times K^3 = 8$. Again, we find which of the submatrices contains the desired cell. In this case, we compute $\left(\frac{6 \mod 4}{2}, \frac{3 \mod 4}{2}, \frac{0 \mod 4}{2}\right)$, which gives us the position of the child $(1, 1, 0)$, at offset $1K^0 + 1K^1 + 0K^2 = 3$. We check the bit at position $8 + 3 = 11$ and find another 1. The K^3 children of the current node start at position $\text{rank}_1(T, 11) \times 8 = 24$. As it is greater than $|T|$, we go to L at position $24 - |T| = 8$. The cell we are looking for is in the submatrix $(0, 1, 0)$, so its bit is at offset 2. Therefore, bit $8 + 2 = 10$ in L contains the value of the cell (1).

6.2.1.2 General range queries

All the queries in a K^n-tree can be seen as a special case of a general n-dimensional range query. For instance, a query that asks for the value of a cell sets a range of length 1 in all the dimensions, therefore at each step in the conceptual tree there is

only a child that may contain the desired cell. In a general range query, we define a set of ranges $(r_1, \cdots, r_n)$ that determine the constraints in each dimension.

To perform a general range query in the K^n-tree we generalize the range queries supported in K^2-trees to higher dimensionality. Starting at the root of the conceptual tree, we check at each step which of the children of the current node contain cells within the specified ranges. Again, this can be easily checked dividing the values of the range by the size of the current node's submatrix to obtain the set of candidate children. The bit of each child is checked, and we keep traversing only the children that have value 1. As an additional computation, the range values must be updated at each node to reflect only the subinterval that falls within the corresponding submatrix.

Example 6.6: In the matrix of Figure 6.3 we want to find all the cells with value 1 in row 3 ($y = 3$), within the three closest layers ($0 \leq z \leq 2$). This query gives us the constraints $(0-7, 3-3, 0-2)$ that have to be checked at each step. At the root of the tree, we first find which submatrices may contain cells in the given region. We divide the extremes of the intervals by $s/k = 4$ and we identify the submatrices $(0/4-7/4, 3/4-3/4, 0/4-2/4) = (0-1, 0, 0)$ as candidates, and their offsets are respectively 0 and 1. Checking the bits in T, we find that only the second submatrix contains ones. We traverse the K^n-tree to position $\text{rank}(T, 1) \times K^3 = 8$ and remove the part of the intervals that falls outside the current submatrix, obtaining a new range $(0-3, 3-3, 0-2)$. In the new node, the submatrices $(0-1, 1, 0-1)$ intersect with the ranges. Their offsets are 2,3,5,7, but only $(1, 1, 0)$ at position $8 + 3 = 11$ and $(1, 1, 1)$ at $8 + 7 = 15$ are set to 1. The process would be repeated for each of the submatrices: the children of 11 are at offset 8 in L, and the new ranges would be $(0-1, 1-1, 0-1)$. The relative offsets of these submatrices are 2,3,6,7. Checking the corresponding bits we would find the cells $(6, 3, 0)$ and $(7, 3, 0)$ at positions 10 and 11 in L. Repeating the same process for the node at position 15, we would find its children and locate the cell $(7, 3, 2)$ as another result of the query.

6.2.2 Enhancements to the K^n-tree

All the enhancements of the basic K^2-tree presented in Section 3.2 can be extended to a general K^n-tree, either with a simple generalization of the methods or with small changes. In this section we sketch the implementation of different enhancements that can be applied to the K^n-tree:

- A *hybrid* K^n-tree can be built with different values of K depending on the level of decomposition. This provides the same space/time tradeoff of K^2-trees (a small K leads to tall but relatively thin trees; a big K leads to shorter trees –therefore reducing the number of traversals– but the number of children per node increases). In general, the value of K used will be smaller (usually 2) for most of the levels of the K^n-tree, but a higher value (4 or 8) may be

chosen for a few top levels. This change requires to adapt the basic traversal algorithms like in the K^2-tree.

- A *matrix vocabulary* can be added to the last levels of the K^n-tree following the same steps of the K^2-tree. The procedure and data structures do not change significantly: at a given level the decomposition stops, and the resulting $K' \times \ldots^n \times K'$ submatrices are assigned variable-length codes according to their overall frequency.

- A *partition of the original matrix* can be performed before constructing the K^n-tree. The original matrix can be converted into a collection of regular fixed-size submatrices. The query algorithms need to be adjusted to take into account the set of submatrices during the navigation.

- The variants with *compression of ones* can be also adapted to higher dimensionality. The proposals presented change the original K^2-tree data structure but the essential navigation algorithms are very similar. The adaptations explained for a basic K^2-tree can be easily extended to any of the proposals with compression of ones presented.

6.2.2.1 Concerns of high dimensionality problems

The generalization of the K^2-tree conceptual tree to higher dimensionality problems leads to some problems derived of the conceptual tree representation. In a conceptual K^n-tree, each node has exactly K^n children if its submatrix contains at least a 1. The increased arity of the tree may lead to a waste of space in some matrices. This was already a concern in simple K^2-trees: the selection of values of K in a K^2-tree had a significant effect in the overall size of the representation. This effect is due to the fact that for bigger K, the lower levels of the tree may require K^2 bits to represent branches almost filled with 0s.

In a K^n-tree the effect of the distribution of the cells in the size of the tree may be significant. For an isolated cell (a cell that is not close to any other cell of the matrix), a long branch in the conceptual K^n-tree must be created. In a worst-case scenario, a matrix with very few cells set to 1, each cell may require almost $K^n \times \log_K s$ bits to be represented, as all the branches created in the K^n-tree are *unary paths* (i.e., paths in the conceptual tree that contain a single 1 at each level). The K^n-tree depends on the clustering of the cells to obtain good compression, and this dependency increases with K and also with n.

The enhancements proposed earlier in this section, although directly applicable to a K^n-tree, may obtain worse results as the number of dimensions increase. For n-dimensional matrices the selection of values of K higher than 2 will not be feasible for all the levels of the conceptual tree; hence the hybrid approach is necessary if we want to reduce the height of the tree. On the other hand, a hybrid K^n-tree may only have high values of K in very few of the top levels of the tree.

The use of a matrix vocabulary in the last levels of the K^n-tree becomes also more difficult as n increases. The compression obtained by the matrix vocabulary depends not only on the frequency distribution of the submatrices but also on obtaining a reduced set of submatrices that can be represented in small space. Even for small K' value of the submatrices, the number of different submatrices may become difficult to manage, as each matrix must be physically stored using $(K')^n$ bits.

Finally, we must note that the implementation of a K^n-tree with compression of ones may also be constrained by the value of n. The proposals based on using a second bitmap for the representation of the three colors of a node, presented in Section 5.2, should be extendible to a K^n-tree without problems. However, the asymmetric proposal presented in Section 5.2.2 degrades clearly as n increases: each black node requires $1 + K^n$ bits to be represented. In further chapters we will make use of variants with compression of ones of the K^n-tree, disregarding the possibility of using this asymmetric proposal due to this overhead.

6.3 Applications

The K^n-tree, the MK^2-tree and the IK^2-tree can be used to index a 3-dimensional dataset, providing access to the data according to filters in any of the dimensions. However, the data structures are significantly different in how they index the data, so one may be better suited than the other for specific kinds of datasets or queries. The K^n-tree provides a simple and efficient way to access a multidimensional dataset filtering by any of its dimensions, its algorithms are symmetric and simple. However, the K^n-tree performance may degrade significantly as the number of dimensions increases, and its compression results depend heavily on the existence of some level of clustering or other regularities in the distribution of values across the dimensions. On the other hand, the MK^2-tree and the IK^2-tree are not symmetric, since the algorithms to filter the partitioning dimension are different from the simple traversals required to filter the other dimensions. The IK^2-tree and especially the MK^2-tree are less efficient to index the partitioning dimension. Finally, we must take into account the fact that the MK^2-tree and the IK^2-tree do not depend on the existence of regularities in the partitioning dimension to obtain good compression (they compress the data as a collection of binary relations), while the K^n-tree compression may degrade significantly if the dataset has no regularities.

In this thesis we will focus on two specific applications of the K^2-tree variants to represent multidimensional data: the representation of temporal graphs and the representation of raster data. The representation of general raster data using variants of K^n-tree and other of our new data structures will be fulfilled in Part III of this thesis, where we will demonstrate how to combine and apply our representations to obtain compact and efficient representations of general raster data as well as spatio-temporal raster data. In the rest of this chapter we introduce simple schemas

to represent a temporal graph using a K^3-tree and a new variant of IK^2-tree, called diff-IK^2-tree, specially designed to store temporal information.

6.3.1 Representation of temporal graphs using a K^3-tree

A temporal graph can be interpreted as a ternary relation, or 3-dimensional grid $X \times Y \times T$, where the first two dimensions represent the origin and destination of edges and the third dimension represents time. Such a representation can be easily stored using a K^3-tree, that will support the usual queries in temporal graphs. Our representation can easily support queries asking for the successors or predecessors of a given node or range of nodes at a specific time point (time-slice queries), or to retrieve the complete state of the graph at a given time point: all these queries are reduced to a 3-d range query where the third dimension T, the temporal dimension, is fixed to a single value.

A K^3-tree representation of a temporal graph can also answer time-interval queries. In this case, the data structure does not provide such a direct access to retrieve the results; instead, some additional filtering is required to obtain the results of the query depending on its semantics:

- A simple time-interval query that asks for all the time points where a cell/range has been active can be answered directly performing a 3-dimensional range query on the K^3-tree.

- A *strong* time-interval query, that asks for the cells that have been active during the *complete* time interval, requires us to perform the same 3-dimensional range query to look up all possible results. In this case, the children of each node must be traversed in ascending order of t. If for an (x, y) pair a t value within the interval has value 0, we stop traversal of the remaining siblings for the same (x, y). Notice however that this process can only be performed efficiently on groups of direct siblings. In order to efficiently filter results for each edge during traversal, we can alter the traversal procedure to check all the results for an (x, y) together, that is, we traverse the tree following that specific order. However, this change makes the traversal slower, so in practice it may be faster to extract all cells in the matrix that fall inside the interval and then extract only (x, y) pairs that are not active during the complete interval.

- A *weak* time-interval query is symmetrical to its *strong* counterpart. In this case, we can avoid looking at siblings for a given (x, y) once we find the first 1 in the K^3-tree for those coordinates.

6.3.2 Representations based on a collection of K^2-trees

6.3.2.1 Naive approach: multiple snapshots of the graph

The simplest representation of a temporal graph consists of storing a complete snapshot of the graph for each time instant. This simple approach has a clear drawback: the amount of space required to store the temporal graph increases with the number of snapshots, and is completely independent of the actual number of changes that occurred in the graph. This means that a graph that remains unchanged for 1000 time instants will require 1000 times the space of the initial graph, because we are storing the complete state of the graph at each time instant.

Even though this approach is unfeasible in practice because of its space requirements, let us present a sketch of its implementation using a MK^2-tree or an IK^2-tree. The construction of both data structures is direct, considering the complete state of the graph at each time instant. To answer queries, the process is similar to the basic algorithms explained:

- We can answer time-slice queries in the MK^2-tree querying only the K^2-tree corresponding to the query time instant. The same query is answered in the IK^2-tree as a fixed-value partitioning dimension query.

- To answer time-interval queries in the MK^2-tree, we have to check all the K^2-trees for the interval. This operation can be especially tuned to the different semantics in the query: to answer *weak* queries, we must check the cell in all the K^2-trees until we find a time instant where the cell was active; to answer *strong* queries we check all K^2-trees until we find a time instant where the cell was not active. Notice that these operations are actually the union and intersection of multiple K^2-trees, studied in Chapter 5 for the K^2-tree1 but that can also be implemented as basic operations in the K^2-tree. To answer time-interval queries in the IK^2-tree we would use the range-restricted partitioning variable strategy, and again we can optimize the query for this case: in *weak* queries, we only need to check that there is at least a 1 in the interval, and we may not store the actual list of active values; in *strong* queries, we may stop traversal if a node does not contain a 1 in *all* the positions in the interval.

The snapshot approach just explained is very efficient performing time-slice queries and can also efficiently answer time-interval queries if we use an IK^2-tree representation. However, in most real-world temporal graphs we expect a relatively small number of changes between consecutive timestamps in relation to the total number of elements in the graph. In this case, the application of the naive multiple-snapshots approach to the compact representation of the graph becomes very inefficient in terms of space, therefore we will use this approach only as a baseline.

6.3.2.2 Differential snapshots: the diff-IK^2-tree

Our proposal is also based on a collection of K^2-trees that will later be merged in an IK^2-tree, like the previous approach. However, in this proposal each K^2-tree will be a *differential* K^2-tree representation of the graph. For the first time instant we will store a complete snapshot of the original state of the temporal graph (that is, the same K^2-tree we used in the naive approach, corresponding to the active edges at t_0). However, each K_i for $i > 0$ will store only the edges in the graph that changed their state between t_{i-1} and t_i. In this new representation, the different K^2-trees require a 1 per *change* in the temporal graph instead of a 1 per active cell at each time instant. Hence, if an edge of the graph was inactive at t_0, became active at t_8 and then inactive again at $t = 120$, we would store a 1 in the corresponding cell in K_8 and K_{120} instead of storing the cell in all the K^2-trees for the interval $[8, 119]$. Notice that, because of how K^2-trees are built, in each differential K^2-tree an internal node in a K^2-tree K_i will be set to 1 if and only if there has been at least one change in the region it covers between t_{i-1} and t_i. Figure 6.4 shows the conceptual decomposition of a temporal graph with only 3 snapshots. In the top of the figure we show the complete snapshots for all the time instants $t = [0, 2]$. For clarity, below the complete snapshots we show the "differential snapshots" that must be built. Finally, at the bottom we display the conceptual K^2-tree representations for each snapshot: the complete snapshot at $t = 0$ and the differential snapshots for the remaining time instants. As we can see in the figure, this optimization is designed for temporal graphs with a relatively small number of changes, where the differential snapshots reduce significantly the number of ones in each K^2-tree, which should also reduce the overall space requirements.

To answer a time-slice query (R, t_i) in the differential K^2-tree representation, we need to rebuild the state of each cell at time t. However, since we only store the changes for each cell, we do not have the state explicitly stored. In order to find the state of a cell we need to *count* the number of changes between t_0 and t_i. Notice that at t_0 a complete snapshot is kept, so a cell is initially active if it was set at t_0; each change is based on the previous state, so even changes are deactivations and odd changes (re)activations of the cell. Hence, if the number of "changes" (the number of K^2-trees where the cell is set to 1) is odd, the cell is active at t_i; otherwise, the cell is inactive. This count operation can be implemented with an x-or operation involving all the K^2-trees corresponding to the interval $[t_0, t_i]$.

To answer weak time-interval queries $(R, I = [t_\ell, t_r])$ we need to perform a similar operation adapted to the semantics of the query. Following the weak semantics, the cell will be returned if it was active at t_ℓ or if it changed its state at any point between t_ℓ and t_r (whether it was active and became inactive or the other way, we would know it was active at some point within I. We can compute this performing an x-or operation to check if the cell was active at t_ℓ and an or operation in the interval $[t_\ell, t_r]$ to check for changes within the query interval. Again, this operation must be checked for each node expanded in the K^2-trees depending on the range R

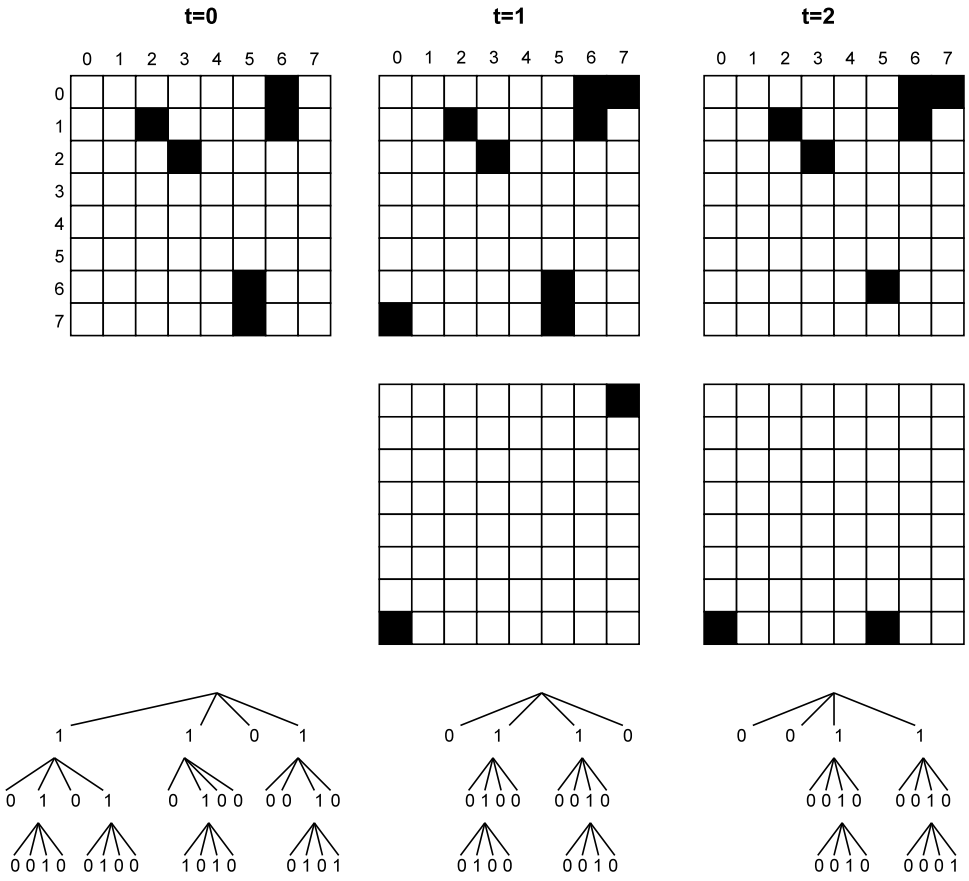


Figure 6.4: Differential K^2-tree representation of a temporal graph.

determined in the other dimensions.

To answer strong time-interval queries $(R, I = [t_\ell, t_r])$ the operations are very similar to those in weak queries. In this case, a cell will be returned if it was active at t_ℓ and no changes occurred within the query interval I. This can be computed using an x-or operation to check the state at t_ℓ and an or operation to check for changes in $[t_\ell, t_r]$.

Notice that in all the queries required a costly operation must be executed involving many K^2-trees: we always need to x-or all the K^2-trees corresponding to time instants before the time instant or interval in our query. In many temporal graphs with a large number of time instants the cost of these operations may become too large for this representation to be feasible. However, we can speed up operations following the same differential approach using an IK^2-tree built over the collection

of K^2-trees explained.

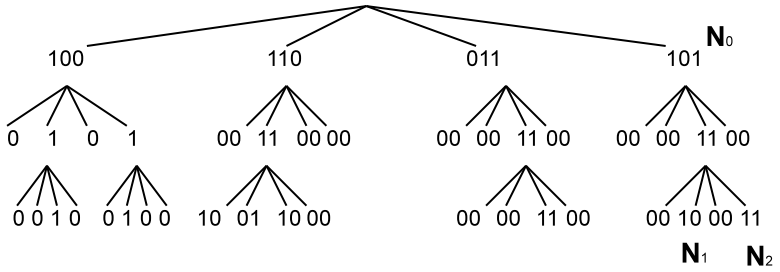


Figure 6.5: diff-IK^2-tree representation of the example in Figure 6.4.

Figure 6.5 shows the equivalent "differential" IK^2-tree representation of the temporal graph in Figure 6.4, that we call diff-IK^2-tree. In each leaf of the diff-IK^2-tree we store a bit for each change in the associated edge. In the internal nodes of the IK^2-tree we store a bit for each time instant where at least one cell covered by that node suffered a change. For example, node N_1 represents row 6, column 5 of the adjacency matrix. This cell is active at t_0 and does not change at any other instant, hence its bitmap contains a single 1 in the first position, corresponding to t_0 (the actual mappings of the bits are t_0 and t_2, as we can see in the bitmap of N_0). On the other hand, node N_2, that represents row 7, column 5, is active at t_0 and becomes inactive at t_2, therefore its bitmap contains two bits set to 1, one for each change. Notice that in this case the diff-IK^2-tree representation provides a huge advantage to perform the counting operations required by queries, since the changes for each node are consecutive in the final bitmap representation. As we will see, all queries can be rewritten in terms of counting operations that are solved by means of *rank* operations in the bitmaps of the K^2-tree:

- To answer a time-slice query, (R, t_i), we navigate the diff-IK^2-tree like in a fixed-value query. In each internal node, we only stop navigation in the branch if the bitmap of the current node has all bits set to 0 until (and including) the bit corresponding to t_i: this case indicates that all cells in the current submatrix were 0 at t_0 and never changed. When we reach a leaf node, we can know whether the cell was active at t_i counting the number of ones between t_0 and t_i, which can be easily computed with 2 *rank* operations on L. Notice that we need to add rank support to L, which is not needed in basic K^2-trees, but the small increase in space provides a much more efficient query method. The cost of a time-slice query does not depend anymore on t_i, the number of time instants before the current one. For example, following the example of Figure 6.5, if we want to check if the cell of N_2 was active at t_2 we navigate from the root of the diff-IK^2-tree mapping the offset in the bitmap corresponding to t_2. In the first level (node N_0), we find some changes before

t_2 so we keep traversing, updating the offset of t_2 to 1. In the next level we keep traversing, and the offset does not change. Finally, when we reach N_2 we know that the bit corresponding to t_2 is the second one. We count the number of ones in the bitmap and obtain a value of 2, meaning the cell is inactive at t_2. If we wanted to check the value at t_1, we would perform the same operation and obtain an odd value (1), meaning the cell is active.

- To answer time-interval queries, we behave exactly like in time-slice queries: we navigate until we reach the leaves, and in the leaves we check the number of changes using just *rank* operations.

6.3.3 Experimental evaluation

In order to demonstrate the efficiency of our proposals to compactly represent temporal graphs, taking advantage of the regularities between consecutive snapshots, we perform an experimental evaluation of several of our proposals in a set of real and synthetic temporal graphs with different characteristics. We will test a representation based on a K^3-tree and the representations based on differential snapshots using a collection of "differential" K^2-trees and a diff-IK^2-tree. We will compare our representations with a baseline approach based on a simple collection of snapshots of the temporal graph, that should be very inefficient in space in graphs with few changes but provides a reference for query times in our other proposals. Additionally, we will compare the space and query times obtained by our representations with the ltg-index, introduced in Section 4.1.

Table 6.1 shows a summary with some information about the datasets used in our experiments. MContact is a real dataset, obtained from the relationships between users in the real social network MonkeyContact. PowerLaw is a synthetic graph whose edges are distributed following a power-law degree distribution. Finally, CommNet simulates a communication network where connections with a short lifespan are established frequently between random nodes. As a first reference of the expected size of the datasets we show for each dataset a "base size" and a "differential size". The base size is the size of the temporal graph represented as a collection of snapshots, where each snapshot is encoded using a pair of integers per edge (that is, the base size uses 8 bytes per active cell and time instant). The differential size gives a better estimation of the amount of information in the graph, as it considers the size of the first snapshot (8 bytes per edge) and then the size of the changes at each successive timestamps (12 bytes/change). The fourth column shows the number of snapshots or time instants contained in each dataset, and the change rate represents the percentage of the total number of edges that change at each timestamp on average. The last two columns are an estimation of the average size of the graph, indicating the number of nodes and edges that are active on average in the different time instants.

Collection	Base size (MB)	Diff. size (MB)	snapshots	change rate	avg.nodes/ snapshot	avg. edges/ snapshot
CommNet	225.9	222	10,000	25%	10,000	20,000
MContact	4,303.9	33.5	220	1%	3,200,000	2,500,000
PowerLaw	22,377.1	716.4	1,000	2%	1,000,000	2,900,000

Table 6.1: Temporal graphs used in our experiments.

6.3.3.1 Comparison of a diff-IK^2-tree with a collection of "differential" K^2-trees

Dataset	Snapshots	Edges/snap.	Multiple K^2-trees	IK^2-tree
CommNet	10,000	20,000	137.28	136.51
Monkey contact	220	2,500,000	4.55	4.54
Power Law	1,000	2,900,000	281.11	281.03

Table 6.2: Space comparison on different temporal graphs (sizes in MB).

In order to prove the efficiency of the diff-IK^2-tree to represent the "differential snapshots" of the temporal graph, we compare the two differential approaches, using a diff-IK^2-tree or multiple K^2-trees, in terms of space and query efficiency. In Table 6.2 we compare the space requirements of both approaches, that as expected are almost identical, since the diff-IK^2-tree is essentially a reorganization of the bits in all the differential K^2-trees.

Next we compare the query times of the diff-IK^2-tree with the multiple K^2-trees. We perform the comparison using two essential and basic queries in any kind of graph: retrieving the direct neighbors (successors) or reverse neighbors (predecessors) of a given node. Both queries are translated easily into a row/column query in the adjacency matrices of the K^2-trees. We build several query sets asking for direct and reverse neighbors of a node at a time instant or time interval, using both strong and weak semantics in time-interval queries. All our query sets are composed of 2000 queries, where nodes and time instants are chosen randomly, while the length of the intervals is fixed for each query set (100).

The results are shown in Table 6.3 for time-slice queries and Table 6.4 for time-interval queries, comparing the query times of the diff-IK^2-tree (IK^2) and the collection of K^2-trees (M.K^2). The experimental results show the superiority of the diff-IK^2-tree when compared with the simpler approach based on multiple K^2-trees in terms of querying efficiency in the context of temporal graphs.

An important consideration is the superiority of the diff-IK^2-tree in time-slice queries in this approach, that must not be confused with the difference between

	CommNet		MContact		PowerLaw	
Query	M.K^2	IK^2	M.K^2	IK^2	M.K^2	IK^2
Direct neighbors	86.78	1.7	0.27	0.14	17.60	1.28
Reverse neighbors	89.07	1.79	0.28	0.15	19.86	1.43

Table 6.3: Time comparison on time-slice queries (times in ms/query).

		CommNet		MContact		PowerLaw	
Query	Semantics	M.K^2	IK^2	M.K^2	IK^2	M.K^2	IK^2
Direct	Weak	87.19	1.84	0.27	0.14	20.72	1.56
neighbors	Strong	88.02	1.82	0.26	0.16	19.73	1.44
Reverse	Weak	90.96	2.07	0.26	0.15	18.36	1.46
neighbors	Strong	93.75	2.05	0.26	0.16	19.98	1.50

Table 6.4: Time comparison on time-interval queries (times in ms/query).

fixed-partitioning-variable queries and fixed-range queries. Notice that we are using "differential" representations, so all our queries (even time-slice queries) require traversing a series of time points to be answered. Particularly, in this "differential snapshot" approach the cost of the query depends more on the time point used in the query than on the length of the query interval: if the time-slice query refers to an early time point t_i (or time interval $[t_\ell, t_r]$), we only need to check the changes in the interval $[0, t_i]$ (or $[0, t_r]$); on the other hand, a time-slice query that asks for a time point $t_j > t_i$ requires us to check more time slices and compute a higher number of changes. This is confirmed in our experiments, since time points and time intervals are selected randomly and the cost of time-slice and time-interval queries is almost the same in Table 6.3 and Table 6.4, with only minor differences.

6.3.3.2 Comparison with other representations

In this section we test the actual efficiency of the two main compact representations introduced in this chapter, the K^3-tree and the diff-IK^2-tree. The K^3-tree has the advantage of providing simpler algorithms that can access all the dimensions similarly, while the diff-IK^2-tree is designed to be very compact and still solve efficiently the relevant queries. We will also compare our approaches with another compact data structure, based on the combination of regular snapshots and explicit logs of changes, explained in Section 4.1. We consider this representation our state-of-the-art alternative for the compact representation of temporal graphs with basic query support. For the ltg-index we use two different setups, one optimized for speed and another optimized for compression, varying the number of snapshots generated.

The "small" approach creates a single snapshot to obtain the best possible space results, while the "fast" approach creates more snapshots to reduce query times.

Dataset	Snapshots (baseline)	diff-IK^2-tree	K^3-tree	ltg-index (fast)	ltg-index (small)
MContact	362.0	4.54	35.0	35.0	35.0
CommNet	666.0	136.51	147.2	194.0	111.0

Table 6.5: Space comparison of the approaches (sizes in MB).

Table 6.5 shows the space requirements of the different approaches in different temporal graph datasets. The diff-IK^2-tree is very competitive with the ltg-index, particularly in the real dataset MContact that has a relatively small number of time instants (220) and a small number of changes between time instants. Another important result is the difference in compression results depending on the compressibility of the dataset: in the CommNet dataset, whose edges are activated randomly and suffer many changes, all the representations designed to take advantage of regularities are smaller than the naive approach based on full snapshots by a factor of 2. On the other hand, in the MContact dataset where the number of changes is relatively small all the compact representations are at least 10 times smaller than the baseline, even when the dataset has a small number of time instants. Finally, the results for the datasets MContact also show that the ltg-index space/time tradeoff does not work in this dataset, because the varying snapshot ratio requires a large number of time instants to provide a significant tradeoff.

Next we compare the query times of all the approaches in time-slice and time-interval queries. The results are shown in Table 6.6. Notice that for the MContact dataset we only show the query times of the "fast" version of the ltg-index, since the "small" version does not reduce the total space and would be slower. The diff-IK^2-tree, that obtained clearly the best results in space, is slower than the K^3-tree in most of the queries. This is due to the additional computation required to rebuild the original state from the differential encoding used in the diff-IK^2-tree. The K^3-tree can answer all queries very fast because it can translate any query in a range query in the 3-d matrix, where at least one coordinate is fixed (row or column) and the time coordinate may be either fixed or restricted to the query interval (in any case a small subset of the overall dataset). On the other hand, the baseline is obviously very fast in time-slice queries but much slower in time-interval queries. Finally, the ltg-index is very competitive in direct queries, that can answer directly, obtaining the best query times of all the representations in all of them. The K^3-tree is still competitive with the ltg-index in time-slice queries, that do not require any post-processing of the results, but the small overhead required by the K^3-tree is enough to make it slower than the fast change logs in the ltg-index, at least when it is optimized for speed. In reverse neighbor queries, where the ltg-index uses a more

Dataset	Query	Semantics	baseline	IK^2-tree	K^3-tree	ltg-index (fast)	ltg-index (small)
CommNet	Direct	Instant	27	1622	96	28	183
		Weak-10	263	1787	183	29	201
		Weak-100	2633	1776	539	47	201
		Strong-10	263	1747	178	29	200
		Strong-100	2640	1745	542	40	202
	Reverse	Instant	29	1718	106	144	32937
		Weak-10	280	1971	211	164	62158
		Weak-100	2788	1989	609	306	61912
		Strong-10	280	1927	210	164	62174
		Strong-100	2782	1943	610	292	62060
MContact	Direct	Instant	57	143	78	45	*
		Weak-10	487	157	118	44	*
		Weak-100	4915	163	149	45	*
		Strong-10	504	153	126	46	*
		Strong-100	4830	145	130	44	*
	Reverse	Instant	59	147	89	147	*
		Weak-10	578	161	162	167	*
		Weak-100	5944	159	180	169	*
		Strong-10	598	153	157	167	*
		Strong-100	5928	149	177	163	*

Table 6.6: Time comparison on time-slice and time-interval queries (times in μs/query).

complex method to obtain the result, the K^3-tree becomes competitive even with the "small-and-fast" version of the ltg-index. The IK^2-tree is in general slower than the other approaches, particularly in the CommNet dataset, and it cannot compete with the query times of the K^3-tree, that relies on a much simpler representation to efficiently index the data.

Summarizing the results, our representations provide interesting alternatives for the compact representation of temporal graphs, providing a space/time tradeoff depending on the characteristics of the datasets and the desired applications. Our diff-IK^2-tree representation of the temporal graph is very competitive in space, and should be much smaller than other representations in datasets with a small change ratio and a reduced number of time instants, like the MContact dataset tested.

On the other hand, the K^3-tree provides a simple and efficient representation of the temporal graph, combining good space results with reasonable and symmetric query times. Both representations provide possible alternatives to non-compact

representations of temporal graphs, and even to compact representations such as the ltg-index when the space utilization is crucial (we can use the diff-IK^2-tree to exploit all the regularities in the graph) or when symmetric query support is of interest (the K^3-tree provides efficient access to direct and reverse neighbors, as well as single cell retrieval and range searches with very simple algorithms).

6.4 Summary

In this chapter we have introduced a generalization of the K^2-tree, called K^n-tree, that can be applied to represent multidimensional data in multiple domains. We analyzed the differences between our new proposal and two existing alternatives called MK^2-tree and IK^2-tree, designed for 3-dimensional data and based on storing the data as a collection of 2-dimensional grids. We give an overview of the differences between the 3 approaches when representing ternary relations. We also showed that the K^n-tree can be used in combination with the encodings developed in Chapter 5 to compress multidimensional grids with highly clustered values.

We have studied a particular application of the K^n-tree, particularly its 3-dimensional variant K^3-tree, to the representation of temporal graphs. We compared the K^3-tree with another proposal introduced in this chapter, the differential IK^2-tree or diff-IK^2-tree, especially designed to compactly represent time-evolving data. Our experimental evaluation shows that the diff-IK^2-tree is able to reduce the space utilization to a small fragment of the original space, at the cost of higher query times. We experimentally compare the K^3-tree and the diff-IK^2-tree with state-of-the-art alternatives, showing that the IK^2-tree can obtain very good space results and the K^3-tree provides simple and symmetric query support.

The proposals in this chapter will be used in Part III to represent raster data. Particularly, the K^3-tree will be used in Chapter 10 to represent raster data, and the variant with compression of ones will also be used in Chapter 11 to represent time-evolving region data. Throughout Part III we will also introduce other representations of raster data using variants of the MK^2-tree and the IK^2-tree.

Chapter 7

The K^2-treap

Top-k queries, that compute the k most relevant results in multidimensional datasets according to a score function, are widely used in a variety of domains. Retrieving the k most relevant documents for a given query pattern in a web search engine, finding the most productive days or vendors in OLAP cubes or locating the highest points in a map are some examples of usual top-k queries. The simplest top-k queries involve finding the top values within a dataset, but many interesting queries require to compute the top-k values in any subset of the data. For example, queries such as "Find the top 10 stores with store_id between x_0 and x_1 that have the highest amount of sales between date y_0 and y_1" can be translated into a two-dimensional query over a range $Q = [x_0, x_1] \times [y_0, y_1]$ in a grid. If we consider the bi-dimensional dataset as an $n \times m$ matrix M, we can formally define top-k range queries as queries to retrieve the top-k points according to their weights within a two-dimensional range Q inside M.

In this chapter we present a new compact data structure, called K^2-treap, that is able to store multidimensional grids in compact space and is designed to efficiently answer top-k queries. Our proposal, as its name states, is inspired by two existing data structures: the K^2-tree and the treap. The K^2-treap is able to answer general top-k queries or top-k range queries, but also supports other simpler queries such as general range searches.

7.1 Conceptual description

Consider a matrix $M[n \times n]$ where each cell can either be empty or contain a weight in the range $[0, d-1]$. We consider a K^2-tree-like recursive partition of M into K^2 submatrices. We build a conceptual K^2-ary tree similar to the K^2-tree, as follows: the root of the tree will store the coordinates of the cell with the maximum weight of the matrix, and the corresponding weight. Then the cell just added to the tree

is marked as *empty*, deleting it from the matrix. If many cells share the maximum weight, we simply pick one of them.

Then, the matrix is conceptually decomposed into K^2 equal-sized submatrices, and we add K^2 children nodes to the root of the tree, each representing one of the submatrices. We repeat the assignment process recursively for each child, assigning to each of them the coordinates and value of the heaviest cell in the corresponding submatrix and removing the chosen points.

The procedure continues recursively for each branch until we reach the cells of the original matrix or we find a completely empty submatrix. Notice that we may find an empty submatrix due to two cases: either the original matrix did not contain any values/weights in the complete region covered by the current node or all the cells that had a weight have been "emptied" in earlier steps of the construction process.

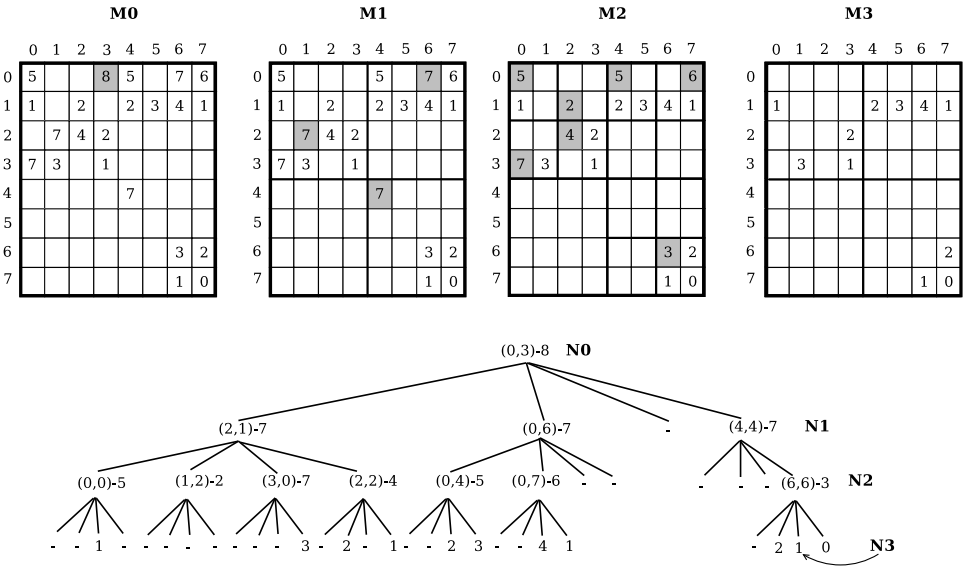


Figure 7.1: Example of K^2-treap construction from a matrix.

Figure 7.1 shows an example of K^2-treap construction, for $K = 2$. On the top of the image we show the state of the matrix at each level of decomposition. $M0$ represents the original matrix, where the maximum value is highlighted. The coordinates and value of this cell are stored in the root of the tree, and the cell would be "emptied", or removed from the matrix. In the next level of decomposition (matrix $M1$), the matrix is decomposed in quadrants and we find the maximum values in each quadrant (notice that the cell assigned to the root is already empty in $M1$). The new local maxima, highlighted in $M1$, are added to the conceptual

tree as children of the root node in the usual left to right, top to bottom order. The process continues recursively, subdividing each matrix into K^2 submatrices. The cells chosen as local maxima are highlighted in the matrices corresponding to each level of decomposition.

The decomposition stops in all branches when empty submatrices are found (empty submatrices are marked in the tree with the symbol "-"). For instance, following the example of Figure 7.1, the third child of $N0$ is empty because the complete submatrix was empty in the original matrix. On the other hand, the first child of $N1$ is also empty, even if the original matrix had a weight in the corresponding submatrix, because that weight (7) was already added to the tree as a local maximum in $N1$ and removed from the matrix. Notice that for each local maximum we store its value and its coordinates, except in the last level of the tree where each node corresponds to a single cell of the matrix, hence the coordinates of the local maximum can be implicitly obtained from the path in the tree leading to the current node.

7.2 Data structure and algorithms

The conceptual tree that composes the K^2-treap is actually partitioned in several data structures that compose the actual K^2-treap data structure. To obtain a good compression of all the components of the K^2-treap, we use 3 different data structures to store the location of local maxima, the weights of the local maxima and the tree topology. Additionally we perform some transformations of the data stored in the conceptual tree that allow us to obtain a more compact representation.

The first step is to apply some transformations to the data, particularly to the numeric values of coordinates and cell weights, in order to obtain more compressible values:

- *Coordinate values:* We transform all coordinates into an offset, relative to the origin of the current submatrix. The coordinates at each level ℓ of the tree are transformed into an offset in the corresponding submatrix. Notice that this can be computed easily during the construction procedure, since we only need to take into account the current level in the conceptual tree and transform each coordinate c_i into $c_i \bmod (n/K^\ell)$. In Figure 7.2 (top) we can see the conceptual tree after this transformation has been applied to the conceptual tree of Figure 7.1. Notice that, for example, the coordinates of node $N1$ have been transformed from the global value $(4, 4)$ to a local offset $(0, 0)$, after computing their mod-4 values. In the next level of the tree, the coordinates of $N2$ are transformed from $(6, 6)$ into $(0, 0)$ again, and so on.

- *Weights:* A second transformation applied to the original data is to differentially encode the values of local maxima. This transformation is performed in a post-order traversal of the conceptual tree, replacing each value with

the difference between its value and the value of its parent. For example, in Figure 7.2 (top) the new value of node $N1$ is computed subtracting the original values of its parent $N0$ and $N1$ itself, that is, $N0$ new value is $8 - 7 = 1$. This transformation, as we said, is applied to all non-empty nodes in the conceptual tree and can be performed in a single traversal of the conceptual tree (either a levelwise bottom-up traversal or a post-order depth-first traversal). The result of the differential encoding is a new sequence of non-negative values, because the value of a node can never be higher than the local maximum stored in its parent. If there are regularities in close values within the matrix, the new sequence may be composed on average of smaller values than the original and therefore it should be easier to compress.

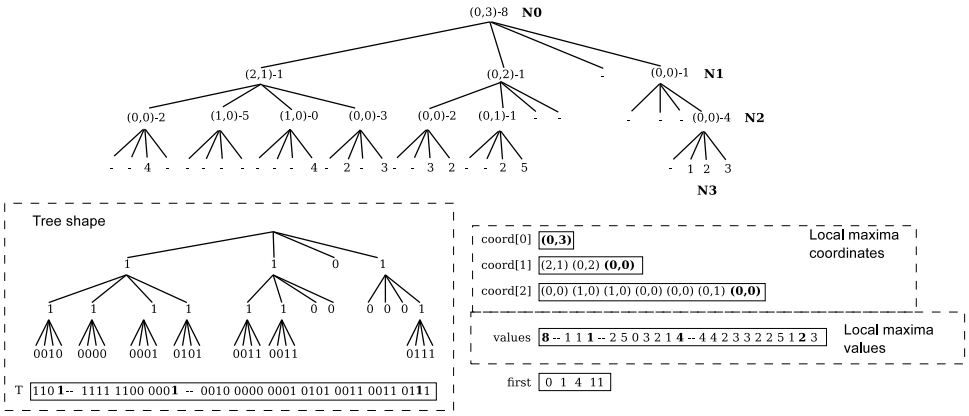


Figure 7.2: Storage of the conceptual tree in our data structures.

After the data transformations we store each type of data in a different data structure to take advantage of their different properties:

- ***Local maxima coordinates:*** The conceptual K^2-treap is traversed levelwise, reading the sequence of cell coordinates from left to right in each level. The sequence of coordinates at each level ℓ is stored in a different sequence $coord[\ell]$. The previous transformation of the coordinate values into relative offsets allows us to use a different number of bits to represent the coordinates in each level. Since each coordinate is now a value not larger than the submatrix size of the current level, the coordinates in each sequence $coord[\ell]$ are stored using a fixed-width representation that uses exactly $\log(n) - \ell \log K$ bits per coordinate. In the bottom of Figure 7.2 we highlight the coordinates of nodes $N0$, $N1$ and $N2$ in the corresponding *coord* arrays. Notice that empty nodes are ignored to build the *coord* arrays, we only store an entry for each non-empty node. Also, in the last level all nodes represent single cells, so there

is no *coord* array in this level. With this representation, the worst-case space for storing t points is $\sum_{\ell=0}^{\log_{K^2}(t)} 2K^{2\ell} \log \frac{n}{K^\ell} = t \log \frac{n^2}{t}(1 + O(1/K^2))$, that is, the same as if we stored the points using the K^2-tree.

- ***Local maxima values:*** The K^2-treap is traversed levelwise and the complete sequence of values is stored in a single sequence named *values*. After the transformation of the original values into *differential* values, the sequence *values* is expected to consist of a majority of small values that should be efficiently compressed using any integer encoding method. In order to exploit the regularities and small values while allowing efficient direct access to the array, we represent *values* with DAC, explained in Chapter 3. Following again the example in Figure 7.2, in the bottom of the figure the complete sequence *values* is depicted and the positions of the example nodes highlighted. We also store a small array $first[0, \lg_K n]$ that stores the offset in *values* where each level starts. This small array adds negligible space to the overall structure and will be used to efficiently navigate the K^2-treap.

- ***Tree structure:*** The final part that will allow us to efficiently query the conceptual representation is a compact representation of the tree structure independent of the values in the nodes. The tree structure of the K^2-treap is stored in a K^2-tree. In the bottom of Figure 7.2 we show the K^2-tree representation of the example tree, where only cells with value are labeled with a 1, while empty cells are labeled with a 0. However, our K^2-tree representation will be slightly different from the original K^2-tree. Our K^2-tree is stored in a single bitmap T with *rank* support, that contains the sequence of bits from all the levels of the tree, instead of the usual two bitmaps T and L. We implement this minor difference because we will need to perform *rank* operations in the last level of the tree to efficiently answer queries, as we will see when we describe the query algorithms.

The separation of the information in independent data structures provides the basis for an efficient compression. Next we will show how the K^2-treap builds on top of the K^2-tree basic navigation algorithms to provide advanced query support and efficient access to the different data structures.

7.2.1 Query algorithms

7.2.1.1 Basic navigation: accessing a cell of the matrix

Assume we want to retrieve the value of a single cell of the original matrix M represented using a K^2-treap. In order to access a cell $C = (x, y)$ we start accessing the root of the K^2-tree. The coordinates and weight of the root node are always stored at $(x_0, y_0) = coord[0][0]$ and $w_0 = values[0]$. If the coordinates of the root node are equal to C, we can immediately return w_0 as the value of the cell.

Otherwise, we need to recursively descend in the conceptual tree to find the answer. To do this, we find the quadrant where the cell would be located and navigate to that node in the K^2-tree using the usual *rank* operations.

Let now p be the position we have just navigated to, that is, the position of the new node in the bitmap T. If $T[p] = 0$ we know that the corresponding submatrix is empty and we can return immediately (the cell was empty). Otherwise, we need to retrieve the coordinates and weight stored for the current node. Since only nodes set to 1 in T have coordinates and weights, we take advantage of the K^2-tree structure to find their offsets. We compute $r = rank_1(T, p)$ in the K^2-tree. The value of the current node will be at $values[r]$, and its coordinates at $coord[\ell][r - first[\ell]]$, where ℓ is the current level in the conceptual tree. Then we compute the absolute values of the coordinates and weights: the weight w_1 is computed as $w_0 - values[r]$ and (x_1, y_1) is computed adding the current submatrix offset to $coord[\ell][r - first[\ell]]$.

With the real values already computed, we check again if (x_1, y_1) are equal to the query coordinates C. If the coordinates are equal, we have found the cell and we return w_1, otherwise we repeat again the decomposition process to find the appropriate quadrant in the current submatrix where C would be located. Notice that the formula to find the children, identical to that of the original K^2-tree, is based on computing $rank_1(T, p) \times K^2$, and the *rank* value r is also needed to access the coordinates and weights, so the cost added by the K^2-tree navigation is very small. The decomposition process is repeated recursively until we find a 0 bit in the target submatrix, we find the coordinates of the cell in an explicit point or we find a 1 in the last level of the K^2-tree. Notice that when we reach the last level of the K^2-tree the current node is already associated with the queried cell C, so we only need to check whether it is empty or not and if it is not empty rebuild its original value.

Example 7.1: In the K^2-treap of Figures 7.1 and 7.2 we want to retrieve the value of cell $(7, 6)$, that corresponds to the highlighted node $N3$. In the root of the tree we find the value $w_0 = 8$ in $values[0]$ and the coordinates $(0, 3)$ in $coord[0][0]$. Since the coordinates do not match, we move to the bottom-right quadrant of the root, located at position $p = 3$ in T (node $N1$). After we check that the node is not empty ($T[3] = 1$) we retrieve the relative weight and coordinates associated with the node: we compute $r = \text{rank}(T, 3) = 3$, and access the relative weight at $values[3] = 1$ and the coordinates at $coord[1][3 - first[1]] = coord[1][2] = (0, 0)$. The absolute weight of the current node is computed as $w_0 - values[3] = 8 - 1 = 7$, and the coordinates are updated adding the offset of the current matrix: since we are at the bottom-right quadrant of the submatrix, we add $(4, 4)$ to the current coordinates, that become $(4, 4)$. Again we have reached a different cell so we keep traversing. We use the value of r to find the children of $N1$ at offset $3 \times K^2 = 12$ in T, and then we access the fourth child, node $N2$, at position $p = 15$. The node is not empty, and we find an absolute value $w_2 = 7 - 4 = 3$ at coordinates $(6, 6)$. We repeat the process again to reach the last level, where we find the third child of node $N2$, node

$N3$, at $p = 42$. We check $T[42] = 1$, so the cell is not empty. Since we are at the last level, we know we are at the right coordinates. We compute and return the final absolute value of the cell as $3 - 2 = 1$.

7.2.1.2 Top-k queries.

The process to answer top-k queries, like all query algorithms in most K^2-tree variants, is based on a tree traversal that starts at the root of the conceptual tree. We consider a general top-k query that asks for the top k values inside a range $Q = [x_1, x_2] \times [y_1, y_2]$ (as we will see, the process to answer top-k queries without range restrictions is identical).

First, we initialize a max-priority queue P that will store K^2-tree nodes according to their associated absolute weight. Each entry in the priority queue will store the *rank* of the node, its coordinates (x_i, y_i) and its weight w_i. In this context the *rank* of the node is not the actual position of the node p in T, that is no longer needed after the coordinates and value have been obtained. Instead, we store the value of $r = \text{rank}(T, p)$, that has already been computed and is used to compute the position of its children in the K^2-tree.

Initially, the priority queue is populated only with the K^2-tree root. The process to answer top-k queries iteratively extracts the first node N in the priority queue (in the first step, the extracted element is the root node). If the coordinates of N fall inside Q we output the cell as the next answer to the query. Regardless of whether or not N was within the range, we check which of its children correspond to (non-empty) submatrices that intersect with the query range Q, and insert into the priority queue only the children that do (including their coordinates and weight). Since all the added elements will be sorted in the priority queue according to their absolute weight, in the next iteration we will extract from the priority queue the next highest value in the matrix, corresponding to a child of a previously extracted node. The process finishes when k results have been found and emitted as output or when the priority queue becomes empty. The latter case occurs only when there are less than k non-empty elements in Q.

For example, assume we want to find the top-3 elements in the range $Q = [4, 7][4, 7]$ in the K^2-treap of Table 7.1 shows the iterations required to answer the query along with the extracted and output nodes in each of them. Initially we have in our priority queue only the root node, with coordinates $(0, 3)$ and value 8 (notice that in this example we omit the *rank* of each node, that is not relevant for the results but must be stored to allow computing the children of a node). In the first iteration we extract the root node and find that its coordinates do not fall within the query range, so we discard the node. We then add to the priority queue P only the children of the root that intersect with Q, in our case only its fourth child $N1$. Hence now P contains only the tuple corresponding to $N1$: $\langle (4, 4), 7 \rangle$. In the second iteration we emit to the output the current maximum and add its only non-empty child to the queue. In the third iteration we emit another result to the

output, corresponding to node $N2$ and add its children to the priority queue. In the fourth iteration we emit the third result to the output and return immediately, since k results have been found already. In we wanted to emit more results, we would continue extracting elements from P in the same way (at most we should find two more results, since the current elements in P do not have children and there were no more values in the range).

Iteration	Extracted node	Output	P
0			$\langle(0,3),8\rangle$
1	$\langle(0,3),8\rangle$		$\langle(4,4),7\rangle$
2	$\langle(4,4),7\rangle$	$\langle(4,4),7\rangle$	$\langle(6,6),3\rangle$
3	$\langle(6,6),3\rangle$	$\langle(6,6),3\rangle$	$\langle(6,7),2\rangle$, $\langle(7,6),1\rangle$, $\langle(7,7),0\rangle$
4	$\langle(6,7),2\rangle$	$\langle(6,7),2\rangle$	$\langle(7,6),1\rangle$, $\langle(7,7),0\rangle$

Table 7.1: Iterations in a top-k query.

Notice that the K^2-treap is especially efficient when no range restrictions are applied (that is, the query range is the complete matrix). In this case, the filtering step that determines whether or not the current maximum is inside the query range is not needed. If no range restrictions are given, the K^2-treap will always emit a result per iteration, finishing computation in at most k iterations.

7.2.1.3 Other supported queries

The K^2-treap can also answer simpler queries that do not involve prioritized access to data.

- Range queries (i.e., report all the points that fall inside the query range Q without a particular ordering of the results). To answer these queries, the K^2-treap can be traversed like a K^2-tree: at each step we compute all the children of the current node that intersect with Q and only continue traversal through them. At each step we must now obtain the coordinates and value of the local maxima, that can be immediately emitted to the output if they fall inside the query range. To perform simple range queries we can simply replace the priority queue used in range queries with a simpler queue or stack where the children of each expanded node are added to be processed.

- We can also answer *interval queries*, which ask for all the points in Q whose weight is in a range $[w_1, w_2]$. We traverse the tree exactly like in a top-k query, but we only output weights whose value is in $[w_1, w_2]$. Moreover, we discard submatrices whose maximum weight is below w_1. Notice that this procedure is much less efficient than top-k queries in the sense that many nodes with

values higher than w_2 are still checked and expanded, while in top-k queries only the children of valid nodes are expanded.

7.3 Better compression of similar values

The K^2-treap is designed to take advantage of similarities in close values in the matrix or grid, thanks to the differential compression of weights in each node of the tree. This differential compression can improve the compression obtained in some datasets, but can be applied in general to datasets where the values in the matrix are not necessarily clustered, or at least not heavily clustered. However, in some application domains we may find grids in which close cells tend to have not only close values but in many cases the same value. In this case the K^2-treap cannot obtain a good compression of uniform regions due to the fact that the K^2-tree that represents its shape only stops decomposition on empty regions. The key to obtain a much better compression in this kind of datasets is to follow the strategies explained in Chapter 5, to compress regions of 1s in the K^2-tree1. In this section we will present two variants of the K^2-treap that take advantage of the clusterization of similar values to improve compression.

K^2-treap-uniform: Our new K^2-treap representations are built similarly to the basic K^2-treap, but the partition process stops when we identify a "uniform" submatrix, either because all its cells are empty or because all of them share the same value. Figure 7.3 shows an example of conceptual tree representation. Matrices $M0$ to $M3$ show the steps of the K^2-treap construction, where the *top* cells selected at each step are highlighted. In this representation when a submatrix is uniform we mark the corresponding node in the K^2-treap (in bold font in the figure) and stop decomposition in that branch, removing all the values in the submatrix. The result is a more compact representation of the dataset if many submatrices with similar values exist in it. Also, notice that in uniform nodes we do not need to store the coordinates of the maximum, since the complete submatrix shares the value.

This representation is stored using the same data structures used in the previous section, with some minor variations. In this case we use a K^2-tree1$^{\text{2bits}}$ representation to store the shape of the tree, so that uniform regions can be distinguished from empty submatrices. Navigation of the K^2-tree1$^{\text{2bits}}$ is identical to the navigation of the original K^2-tree, except that we need an additional operation to check whether a node is uniform but empty or uniform with a value. During the K^2-tree1$^{\text{2bits}}$ traversal we check for uniform regions, and when a uniform region is found we can immediately emit all the cells within the current submatrix using its associated value.

Notice however that we store the value of all the nodes of the K^2-treap, but we only store the coordinates of non-leaf nodes. Hence, to obtain the offset in the lists of coordinates we can use the same formula of the previous version ($\text{rank}_1(T, p) -$

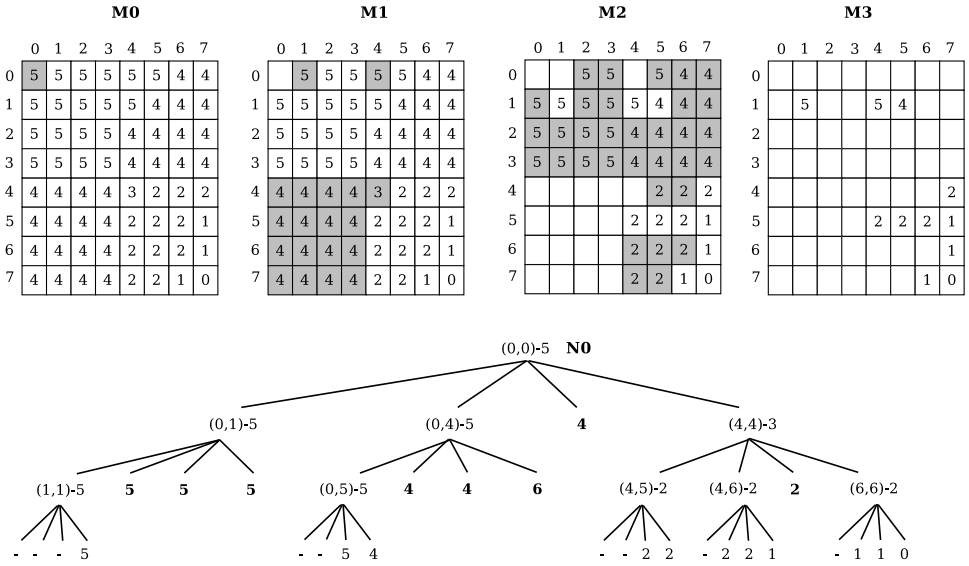


Figure 7.3: Compression of uniform subtrees in the K^2-treap (K^2-treap-uniform).

$first[\ell]$), that computes the number of non-leaf nodes in the current level of the tree. However, to obtain the offset in the list of values we must also take into account the uniform leaf nodes, that are marked with a 1 in T': hence, the offset in the value list for a node at position p in T is computed as $\text{rank}_1(T, p) + \text{rank}_1(T', \text{rank}_0(T, p))$ (i.e., the number of internal nodes plus the number of uniform nodes up to the current position).

K^2-treap-uniform-or-empty: Next we present a slightly different approach that aims to obtain even better compression at the cost of higher query times. In this approach we change slightly the definition of uniform submatrix to include also submatrices that contain both empty and non-empty cells, as long as the non-empty cells share the same value. Notice that in Figure 7.3 we have some submatrices that would become uniform regions according to this definition (for instance the top-left quadrant in $M1$). We can build the previous K^2-treap considering these regions as uniform as well. An example of this variant is shown in Figure 7.4.

This approach clearly obtains a better compression than the previous one in the example, since it can stop decomposition earlier in many branches of the tree. However, in this representation uniform regions and empty regions are not differentiated, which leads to a problem in query time: some cells might be emitted twice using the original algorithms. For instance, cell $(0, 0)$ can be found in the root

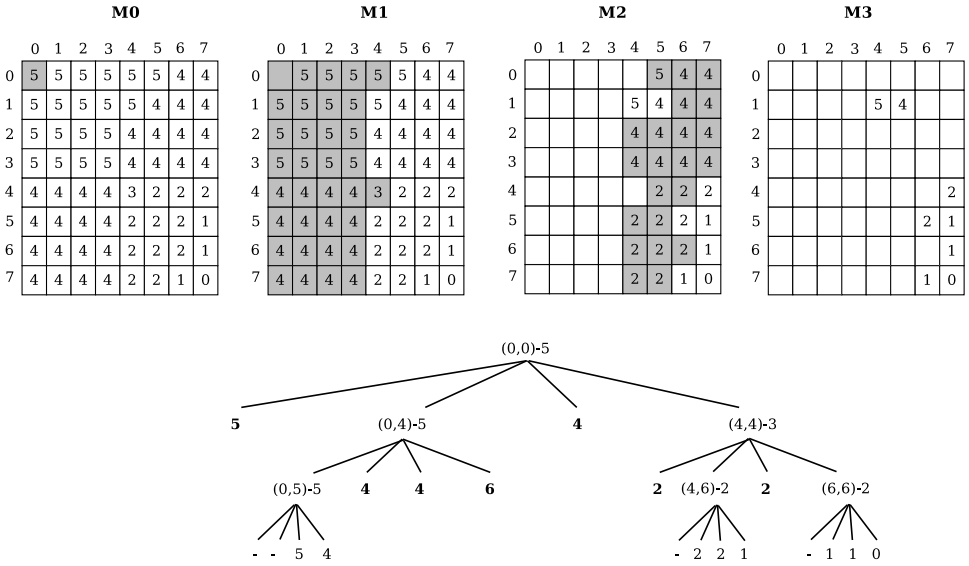


Figure 7.4: Compression of "uniform-or-empty" regions in the K^2-treap (K^2-treap-uniform-or-empty).

of the tree but is also found in its first child, as part of the uniform region identified in that node.

To answer top-k queries using this approach we need to keep track of the already emitted results to avoid emitting a cell more than once. This can be easily done with an additional data structure that stores the already emitted results and is looked up before emitting any result (a hash table or a binary search tree, for example). The effect of this change is that we have an added cost of checking each result before emitting it, in addition to the cost of retrieving the result, and this cost may be very significant for large values of k if we use a data structure like a binary search tree to store previous results. Nevertheless, this alternative should obtain a better compression than the K^2-treap-uniform variant and therefore both variants provide an interesting space/time tradeoff.

7.4 Experiments and Results

7.4.1 Experimental Framework

In order to test the efficiency of our proposal we use several synthetic datasets, as well as some real datasets where top-k queries are of interest. Our synthetic datasets attempt to simulate OLAP data that have little or no regularities in general. All

our synthetic datasets are square matrices where a variable percentage of the cells have a value set. We build different matrices varying the following parameters: the *size* $s \times s$ of the matrix ($s = 1024, 2048, 4096, 8192$), the number of different weights or values d in the matrix (16, 128, 1024) and the *percentage* p of cells that have a weight (10, 30, 50, 70, 100%). The distribution of the weights in all the datasets is uniform in the interval $[0, d-1]$, and the cells with valid points are distributed randomly. For example, the synthetic dataset with ($s = 2048, d = 128, p = 30$) has size 2048×2048, 30% of its cells have a value and their values are follow a uniform distribution in $[0, 127]$.

We also test our representation using real datasets. We extracted two views from a real OLAP database storing information about sales achieved per store/seller each hour over several months: *salesDay* stores the number of sales per seller per day, and *salesHour* the number of sales per hour. Historical logs are accumulated over time, and are subject to data mining processes for decision making. In this case, finding the places (at various granularities) with most sales in a time period is clearly relevant. Table 7.2 shows a summary with basic information about the real datasets. For simplicity, in our datasets we will ignore the cost of mapping between real timestamps and seller ids to rows/columns in the table, and we assume that the queries are given in terms of rows and columns. This process may be necessary to perform queries in real OLAP databases, but is not always necessary and should have a very small cost in comparison with the total cost of the query.

Dataset	#Sellers (rows)	#Instants (columns)	#Values
$SalesDay$	1314	471	297
$SalesHour$	1314	6028	158

Table 7.2: Characteristics of the real datasets used.

We compare the space requirements of the K^2-treap with a state-of-the-art solution based on wavelet trees enhanced with RMQ structures [NNR13] (*wtrmq*), described in Section 2.3.4. Since our matrices can contain none or multiple values per column, we transform our datasets to store them using wavelet trees. The wavelet tree will store a grid with as many columns as values we have in our matrix, in column-major order. Following the steps explained in Section 2.3.3, an additional bitmap is used to map the real columns with virtual ones. Therefore, after the bitmap mappings, top-k range queries in the dataset are translated into top-k range queries in the sequence represented by the wavelet tree.

We also compare our proposal with a representation based on constructing multiple K^2-trees, one per different value in the dataset. In this representation (*mk2tree*), top-k queries are answered by querying consecutively the K^2-tree

representations for the highest values.

All bitmaps that are employed use a bitmap representation that supports *rank* and *select* using 5% of extra space. The *wtrmq* was implemented using a pointerless version of the wavelet tree [CN08] with an RMQ implementation that requires 2.38 bits per value.

We run all the experiments in this section in a machine with 4 Intel(R) Xeon(R) E5520 CPU cores at 2.27 GHz 8 MB cache and 72 GB of RAM memory. The operating system is Ubuntu version 9.10 with kernel 2.6.31-19-server (64 bits). Our code is compiled using gcc 4.4.1, with full optimizations enabled.

7.4.2 Space Comparison

We start by comparing the compression achieved by the representations. As shown in Table 7.3, the K^2-treap overcomes the *wtrmq* in the real datasets studied by a factor over 3.5. The *mk2tree* representation is competitive with the K^2-treap and even obtains slightly less space in the dataset *salesHour*, taking advantage of the relatively small number of different values in the matrix.

Dataset	K^2-treap	*mk2tree*	*wtrmq*
SalesDay	2.48	3.75	9.08
SalesHour	1.06	0.99	3.90

Table 7.3: Space used by all approaches to represent the real datasets (bits/cell).

The K^2-treap also obtains the best space results in most of the synthetic datasets studied. Only in the datasets with a very small number of different values ($d = 16$) the *mk2tree* uses less space than the K^2-treap. Notice that, since the distribution of values and cells is uniform, the synthetic datasets are close to a worst-case scenario for the K^2-treap and *mk2tree*, that could take advantage of a more clustered distribution of the values.

Due to the large number of synthetic datasets created, we provide in Figure 7.5 just a summary of the space results for some of the synthetic datasets used. In the top of the figure we show the evolution of space with the percentage of cells set, p. In all the representations we obtain worse compression in bits/cell (i.e., total bits divided by s^2) as p increases. However, the *wtrmq* increases much faster depending on p. If we measured the compression in bits per point (i.e., total bits divided by t), then the space of the *wtrmq* would become independent of p ($\lg s$ bits), whereas the K^2-treap and *mk2tree* obtain less space (in bits per point/value) as p increases ($\lg \frac{100}{p}$). That is, the space usage of the *wtrmq* increases linearly with p, while that of the K^2-treap and *mk2tree* increases sublinearly.

In the middle plot of Figure 7.5 we show the evolution of compression with the size of the matrix s. As we can see, the K^2-treap is almost unaffected by the matrix size, as its space is around $t \lg \frac{s^2}{t} = s^2 \frac{p}{100} \lg \frac{100}{p}$ bits, that is, constant per cell as s grows. On the other hand, the *wtrmq* uses $t \lg s = s^2 \frac{p}{100} \lg s$ bits, that is, its space per cell grows logarithmically with s. Finally, the *mk2tree* obtains poor results in the smaller datasets but is more competitive on larger ones (some enhancements in the K^2-tree representations behave worse if the matrices are below a minimum size). In any case, the improvements in compression of the *mk2tree* stall once the matrix reaches a certain size and the K^2-treap is still the most competitive approach in larger matrices.

Finally, at the bottom of Figure 7.5 we show the space results when varying the number of different weights d. The K^2-treap and the *wtrmq* are affected only logarithmically by d. The *mk2tree*, instead, is sharply affected, since it must build a different K^2-tree for each different value: if d is very small the *mk2tree* representation obtains the best space results also in the synthetic datasets, but for large d its compression degrades significantly. Summing up, over all the synthetic datasets, the K^2-treap uses from 1.3 to 13 bits/cell, the *mk2tree* from 1.2 to 19, and the *wtrmq* from 4 to 50 bits/cell. The K^2-treap is the most competitive approach and significantly overcomes the *wtrmq* in all cases.

7.4.3 Query Times

7.4.3.1 Top-k queries on synthetic datasets

In this section we analyze the efficiency of top-k queries, comparing the K^2-treap with the *mk2tree* and the *wtrmq*. For each dataset, we build multiple sets of top-k queries for different values of k and different spatial ranges. All query sets are generated for fixed k and w (side of the spatial window or range). Each query set contains 1000 queries where the spatial window is placed at a random position within the matrix. Given that the distribution of the values in the matrices is also random, the queries are expected to have a similar number of results, proportional to the window size and the percentage of ones p in the dataset, but no restrictions were imposed during the construction of the query sets.

Figure 7.6 shows the time required to perform top-k queries in some of our synthetic datasets, for different values of k and w. The K^2-treap obtains better query times than the *wtrmq* in all the queries, and both evolve similarly with the size of the query window. On the other hand, the *mk2tree* representation obtains poor results when the spatial window is small or large, but it is competitive with the K^2-treap for medium-sized ranges. This is due to the procedure to query the multiple K^2-tree representations: for small windows, we may need to query many K^2-trees until we find k results; for very large windows, the K^2-treap starts returning results in the upper levels of the conceptual tree, while the *mk2tree* approach must reach the leaves; for some intermediate values of the spatial window,

the K^2-treap still needs to perform several steps to start returning results, and the $mk2tree$ representation may find the required results in a single K^2-tree. Notice also that, as we explained in the algorithm description, the K^2-treap is more efficient when no range limitations are given (that is, when $w = s$), since it can return after exactly k iterations. Figure 7.6 shows the results for two of the synthetic datasets, but similar comparison results have been obtained in all the synthetic datasets studied: the K^2-treap outperforming the alternative approaches in almost all the cases, except in some queries with medium-sized query windows, when the $mk2tree$ can obtain slightly better query times.

7.4.3.2 Top-k queries on real datasets

Next we perform a set of queries that would be interesting in our real datasets. We start with the same $w \times w$ queries as before, which filter a range of rows (sellers) and columns (days/hours). Figure 7.7 shows the results of these range queries. As we can see, the K^2-treap outperforms both the $mk2tree$ and $wtrmq$ in all cases. Similarly to the previous results, the $mk2tree$ approach also obtains poor query times for small ranges but is better in larger ranges.

We run two more specific sets of queries that may be interesting in many datasets, and particularly in our examples: "column-oriented" and "row-oriented" range queries, that only restrict one of the dimensions of the matrix. Row-oriented queries ask for a single row (or a small range of rows) but do not restrict the columns, and column-oriented ask for single columns. We build sets of 10,000 top-k queries for random rows/columns with different values of k. Figure 7.8 (top) shows that in column-oriented queries the $wtrmq$ is faster than the K^2-treap for small values of k, but our proposal is still faster as k grows. The reason for this difference is that in "square" range queries, the K^2-treap only visits a small set of submatrices that overlap the region; in row-oriented or column-oriented queries, the K^2-treap is forced to check many submatrices to find a few results. The $mk2tree$ suffers from the same problem of the K^2-treap, being unable to filter efficiently the matrix, and obtains the worst query times in all cases.

In row-oriented queries (Figure 7.8, bottom) the $wtrmq$ is even more competitive, obtaining the best results in many queries. The reason for the differences found between row-oriented and column-oriented queries in the $wtrmq$ is the mapping between real and virtual columns: column ranges are expanded to much longer intervals in the wavelet tree, while row ranges are left unchanged. Notice anyway that our proposal is still competitive in the cases where k is relatively large.

The overall results show that the K^2-treap is a very flexible representation and provides the best compression results and query times in most of the datasets and queries. More importantly, its compression and query times are good in all the studied cases, while the alternative representations become very inefficient depending on the characteristics of the query.

7.5 Summary

In this chapter we have shown the space and time efficiency of our proposal, the K^2-treap, to represent multidimensional datasets in small space and efficiently answer top-k range queries. Our experimental evaluation has demonstrated that the K^2-treap achieves small space in synthetic and real datasets, and is faster than state-of-the-art proposals based on wavelet trees in most cases. Additionally, our experiments show that the K^2-treap scales better than alternatives based on the MK^2-tree or the wavelet tree when the size of the dataset or the number of different values changes.

In this chapter we have also introduced some variations of the K^2-treap, inspired by the encodings used in Chapter 5, that are designed to improve compression in datasets where regularities of values exist and are very significant. The most direct application of these variants is the compact representation of raster datasets, that consist of matrices of values with a very high level of clusterization of values. These K^2-treap variants optimized for the compression of uniform regions will be tested in this domain (compression of raster data) in Chapter 10.

Even though we focused in the representation of bi-dimensional data, the K^2-treap can be easily generalized to higher dimensionality problems, replacing the K^2-tree data structure used to manage the shape of the conceptual tree with a K^n-tree. The K^2-treap should scale better than alternatives based on wavelet trees, in terms of space and query efficiency, as the number of dimensions increases.

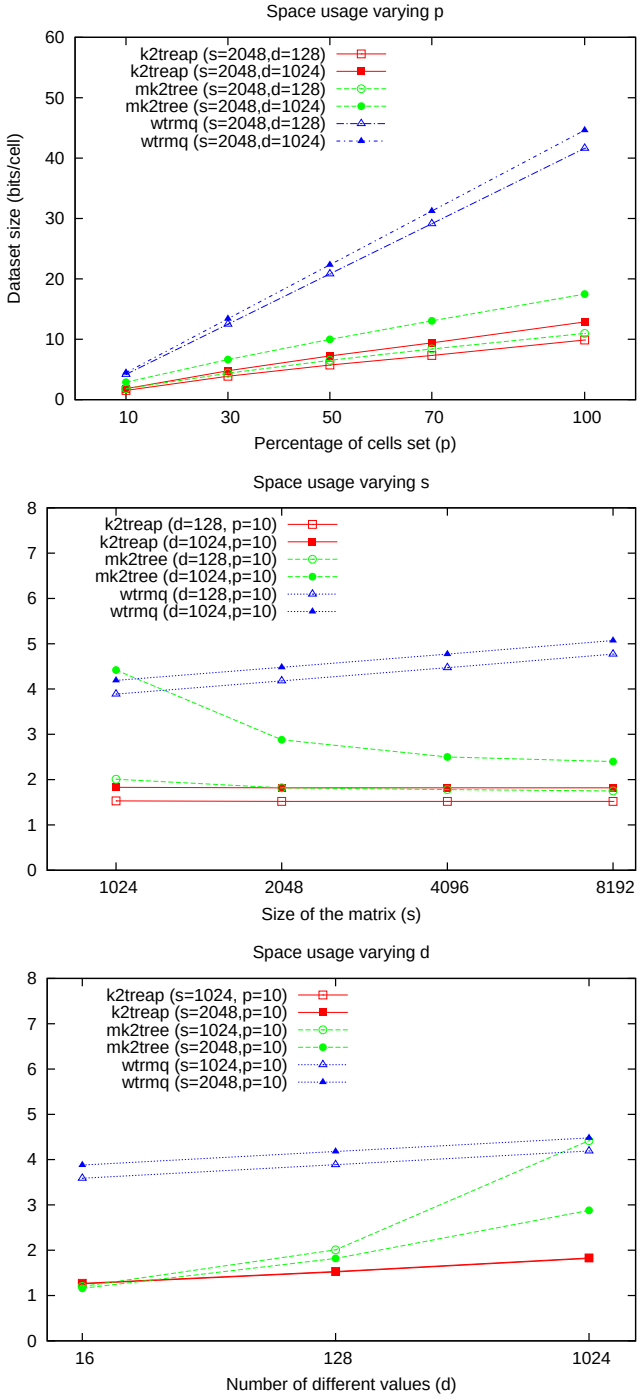


Figure 7.5: Evolution of the space usage when p, s and d change, in bits/cell.

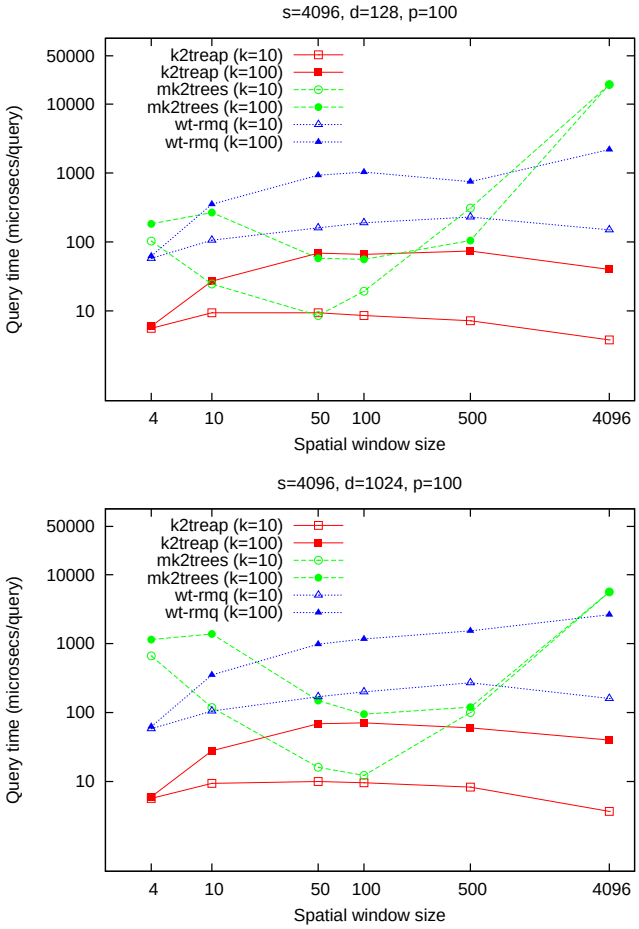


Figure 7.6: Times of top-k queries in synthetic datasets.

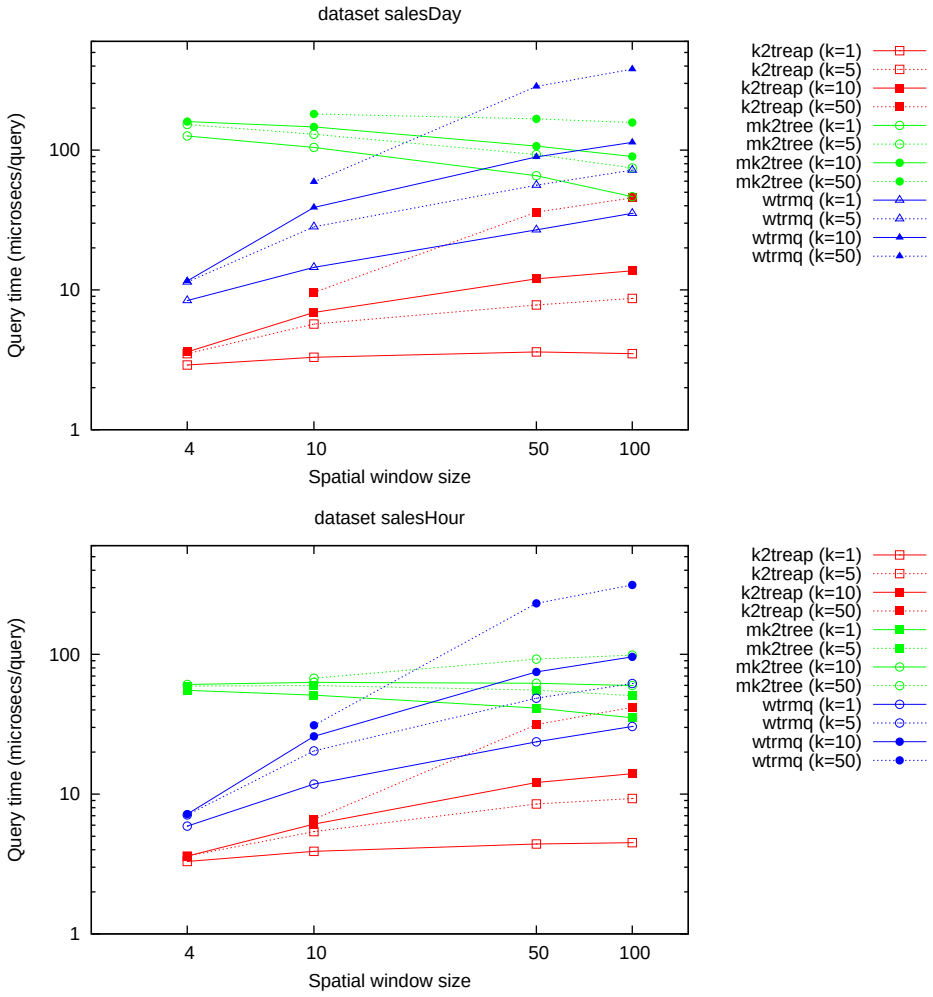


Figure 7.7: Query times of top-k queries in real datasets.

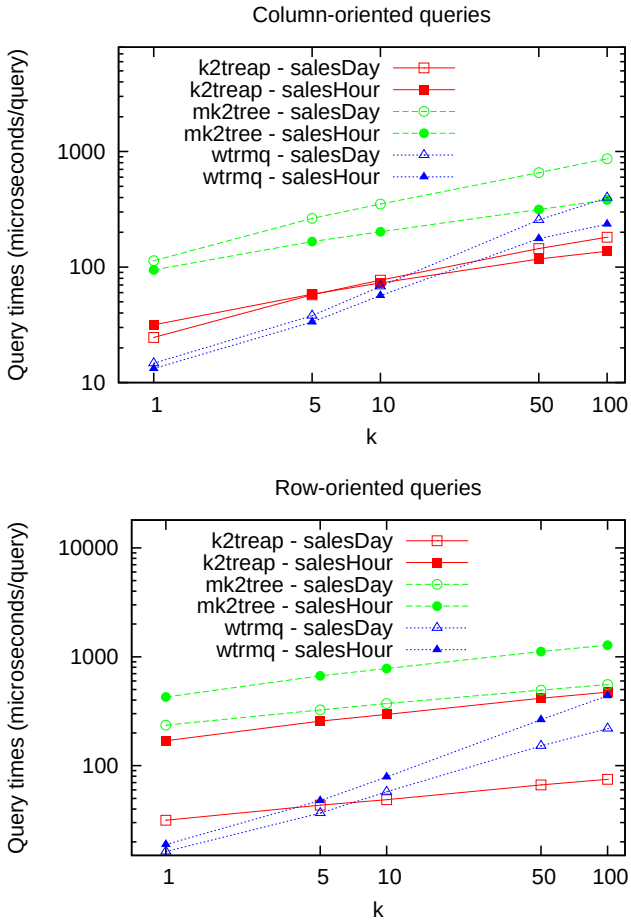


Figure 7.8: Query times of row-oriented and column-oriented top-k queries.

Part II

New dynamic data structures

Chapter 8

Dynamic K^2-tree: dK^2-tree

A lot of effort has been devoted in the literature to obtain efficient dynamic representations for well-known static problems. In particular, the representation of dynamic bit vectors supporting insertions and deletions has been widely studied, as it is the basis for many other succinct data structures. As an example, the representation of dynamic sequences and trees is usually based on dynamic bit vectors.

In Section 2.4 we have introduced theoretical data structures for the representation of bit vectors and sequences that match the theoretical lower bound for their respective problems. However, as we have seen, practical implementations usually abandon the theoretical proposals to obtain good results in practice. The compact representation of dynamic binary relations has also been tackled in the past. The general problem was only recently studied, and several representations that are based on sequence representations can be adapted to a dynamic environment.

In this chapter we aim to provide a dynamic data structure for the compact representation of binary relations that obtains good compression results in different domains. Our proposal is a dynamic version of the K^2-tree, called dK^2-tree, that aims to provide the same compression and query capabilities. Three main goals are pursued in the dK^2-tree:

- *Achieve space efficiency*: the space results of the dynamic version of the K^2-tree should be very close to those of the static K^2-tree. The main advantage of K^2-trees over other binary relation representations is the reduced space and the advanced query support (symmetric cost of queries and support for range queries). Our goal is to obtain a dynamic representation that can obtain space results close to those of a static K^2-tree.
- *Query efficiency*: the dK^2-tree must support the same queries that static K^2-trees. Regarding query efficiency, the overhead in dynamic compressed data structures is usually high due to the theoretical lower bounds and the complex

data structures required to meet them, as shown in Section 2.4. Our goal is to obtain a reasonable overhead in the dynamic data structure.

- *Efficient support for usual update operations*: the dK^2-tree should support the most usual update operations in a binary relation, particularly adding new pairs to the relation and adding new objects to the base sets of the relation.

In order to fulfill these goals, the dK^2-tree replaces the main data structures of the static K^2-tree with dynamic implementations. We focus on obtaining a representation that can work well in practice. To do this, we design data structures that do not aim at providing worst-case bounds in time and space, but are simple to implement and will allow us to fulfill our goals in many real-world applications.

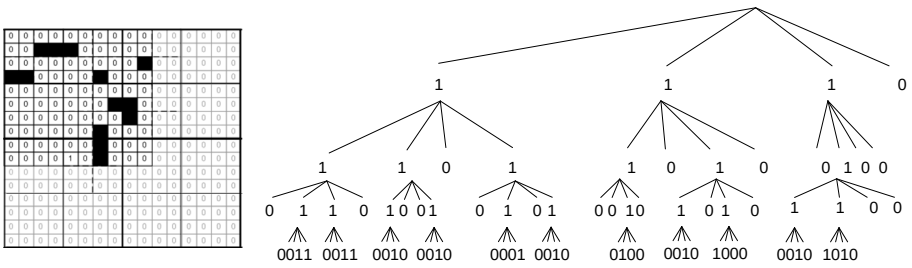
8.1 Data structure and algorithms

Recall from Section 3.2 that a in a static K^2-tree there is a conceptual tree that represents a binary matrix and is stored in two bitmaps T and L, that contain the bits of the upper levels of the tree and the last level respectively. In the dK^2-tree, the bitmaps T and L are replaced by dynamic data structures that support the same operations. Therefore, a dK^2-tree consists of two dynamic data structures, that we call T_{tree} and L_{tree}, that replace T and L respectively, providing the same functionalities of the static bitmaps and allowing at the same time update operations.

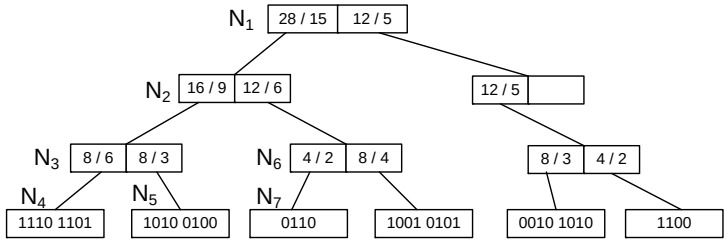
Our T_{tree} and L_{tree} are tree structures similar to some of the state-of-the-art proposals for the representation of dynamic bit vectors. The leaves of T_{tree} and L_{tree} contain chunks of the the bitmaps T and L respectively. The size of the leaves, as well as their maximum and minimum occupancy, can be parameterized. The internal nodes of T_{tree} and L_{tree} allow us to access the leaves for query and update operations.

Each internal node of T_{tree} contains a set of entries of the form $\langle b, o, P\rangle$, where b and o are counters and P is a pointer to the corresponding child node. The counters b and o store the number of bits and ones in the fragment of the bitmap stored in all the leaves that are descendants of the node pointed to by P. Therefore, if P points to a leaf node, the counters will store the number of bits stored in the leaf b and the number of ones o in it. If P points to an internal node, b and o will contain the sum of all the b and o counters in the child node. Internal nodes in L_{tree} are very similar, but they contain entries $\langle b, P\rangle$ since o-counters are not needed because we do not need *rank* support in L.

Figure 8.1 shows a simplified example of a dK^2-tree representation. In this example, we assume that the leaf nodes can store at most 8 bits, while the internal nodes of T_{tree} and L_{tree} can store at most 2 and 3 entries respectively. The entries of T_{tree} contain a pair b/o and the pointer to the child node is represented visually.



T: **1110 1101 1010 0100 0110 1001 0101 0010 1010 1100**



L: **0011 0011 0010 0010 0001 0010 0100 0010 1000 0010 1010**

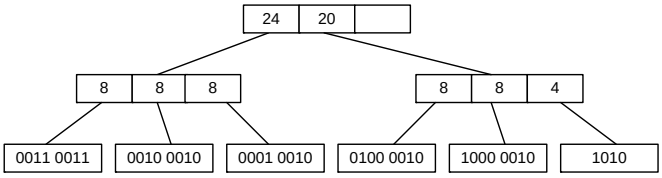


Figure 8.1: A binary matrix, conceptual K^2-tree representation (top) and dynamic K^2-tree representation of the conceptual K^2-tree.

Note that in this example, and in general in any dK^2-tree, the nodes of T_{tree} and L_{tree} may be partially empty. Each node has a maximum and minimum capacity and may contain any number of bits or entries between those parameters. The tree is completely balanced, and nodes may be split or merged when the contents change. The behavior of T_{tree} and L_{tree} on update operations will be explained in more detail later in this chapter.

The actual implementation of this data structure is quite straightforward. The entries in internal nodes of T_{tree} and L_{tree} are stored uncompressed in order to facilitate traversal of internal nodes. The loss in space caused by this choice is very small and, as we will see later, traversing an internal node becomes very easy. All the entries in an internal node have the same size, in order to provide efficient access to them. The leaves of T_{tree} and L_{tree} simply store the bit arrays of their corresponding chunk. Additionally, each node in T_{tree} and L_{tree} contains a small header with its structural information. A flag *leaf* indicates whether each node is an internal or leaf node. To save space, another flag *long* can be used in internal nodes to determine the size of the counters in bits; otherwise, a sufficiently large size can be determined beforehand and used in all the nodes. Finally, a field *size* is used to store the current size of the node contents in order to detect when the node is full. In leaf nodes, *size* contains the number of bits in the stored chunk. In internal nodes, *size* stores the space occupied by all the entries used (since the entries are of fixed size, we can store the total space or just the number of entries). Figure 8.2 shows an example of internal node and leaf node in T_{tree}: the internal node (top) contains fixed-size entries, storing in the field *size* just the number of entries in the node (an additional flag *long* would be added if different nodes may have entries of different sizes). The first entry contains the 3 fields explained: number of bits, number of 1s, and a pointer to the child. The child node (bottom) is a leaf node that stores its size in bits and the plain bitmap representation.

8.1.1 Query support

8.1.1.1 Access and rank operation on the dynamic bitmaps

The dK^2-tree essentially replaces the bitmaps T and L of a static K^2-tree with dynamic bitmap implementations. Therefore, all the queries supported by static K^2-trees are also supported by dK^2-trees simply by providing the same low-level operations offered by T and L in the dynamic bitmaps T_{tree} and L_{tree}. T and L support random access to any position in the bitmap (*access* operation, that is trivial in the static bitmap representation), and T also supports *rank* operations. In this section we will show how to perform *access* and *rank* operations in T_{tree} and *access* operations in L_{tree}.

The procedure to perform *access* and *rank* operations in T_{tree} or L_{tree} is very similar:

- First, we must find the leaf that contains p. This is performed by the *findLeaf*

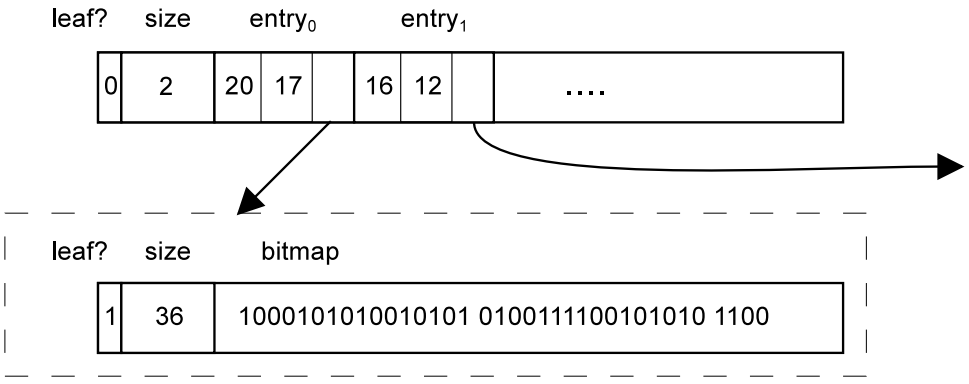


Figure 8.2: Example of an internal node (top) and a leaf node (bottom) in T_{tree}.

operation, that returns the leaf node N_ℓ containing position p and the number of bits b_{before} and ones o_{before} before the beginning of the leaf.

- After *findLeaf* returns, the bitmap operation can be reduced to the same operation in the bitmap of N_ℓ. b_{before} and o_{before} are used to compute the global result for the *access* or *rank* operation.

The *findLeaf* operation starts at the root node of T_{tree} or L_{tree} and uses the counters stored in internal nodes to guide the navigation. The algorithm is quite straightforward and differs only slightly between T_{tree} and L_{tree}, because L_{tree} does not store o counters. Algorithm 8.1 shows the complete implementation of the *findLeaf* operation. It starts at the root of T_{tree} or L_{tree} and traverses the tree until it reaches a leaf node. During traversal, the counters in the internal nodes are used to compute the number of bits and ones to the left of the leaf storing the desired position. At each internal node, all its entries are traversed in order, updating b_{before} and o_{before} until b_{before} surpasses the current position. At this point, we know that the child storing position p is the one pointed by the current entry, so the algorithm proceeds to the next level of the tree. When a leaf node is found, the values of b_{before} and o_{before} contain the accumulated number of bits and ones to the left of the current node.

Example 8.1: We want to *access* position 18 of T_{tree} in the dK^2-tree of Figure 8.1. The *findLeaf* operation starts at the root of T_{tree} (N_1) and traverses all its entries. The first entry of the root node contains 28 bits, so the procedure traverses its pointer to N_2. At N_2, the procedure jumps over the first entry, setting $b_{\text{before}} = 16$ and $o_{\text{before}} = 9$ and goes to the second entry. Now, the accumulated number of bits in the second entry ($16 + 12$) is bigger than the position we are looking for, so the procedure jumps again to the child pointed by the second entry (N_6). In the third

Algorithm 8.1 *findLeaf* operation

```
function findLeafT(N, p, b_before, o_before)
    if not N.leaf then
        nEntries ← N.nEntries
        i ← 0
        e ← N.entries[i]
        P ← e.P
        while b_before + e.b < p do
            b_before ← b_before + e.b
            o_before ← o_before + e.o
            i ← i + 1
            e ← N.entries[i]
            P ← e.P
        end while
        return findLeafT(readNode(P), p, b_before, o_before)
    else
        return (N, b_before, o_before)
    end if
end function
function findLeafL(N, p, b_before)
    if not N.leaf then
        nEntries ← N.nEntries
        i ← 0
        e ← N.entries[i]
        P ← e.P
        while b_before + e.b < p do
            b_before ← b_before + e.b
            i ← i + 1
            e ← N.entries[i]
            P ← e.P
        end while
        return findLeafL(readNode(P), p, b_before, 0)
    else
        return (N, b_before, o_before, 0)
    end if
end function
function findLeaf(tree, p)
    if p < tree.T_tree.size then
        return findLeafT(tree.T_tree.root, p, 0, 0)
    else
        return findLeafL(tree.L_tree.root, p - tree.L_tree.size, 0)
    end if
end function
```

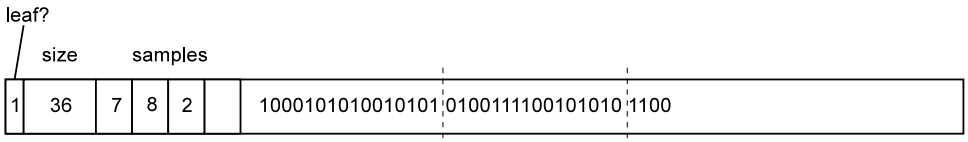


Figure 8.3: Leaf node in T_{tree} with rank directory, for $S_T = 16$ bits.

level, the procedure finds the position in the first entry and jumps to N_7 without changing the values of b_{before} and o_{before}. Finally, the reached node N_7 is a leaf and the procedure stops, returning the current node and the computed values for b_{before} and o_{before}: $(N_7, 16, 9)$.

When *findLeaf* returns the leaf node N_ℓ that contains the desired position, any bitmap operation is reduced to the local operation in N_ℓ. Let T_ℓ be the bitmap stored in N_ℓ. Then, *access* and *rank* operations in T_{tree} can be computed as follows:

$$\text{access}(T_{\text{tree}}, p) = \text{access}(T_\ell, p - b_{\text{before}})$$

$$\text{rank}(T_{\text{tree}}, p) = o_{\text{before}} + \text{rank}(T_\ell, p - b_{\text{before}})$$

The *access* operation in T_ℓ is trivial: we simply need to access the corresponding bit. However, the *rank* operation in T_ℓ is still a costly operation. In order to speed up the local rank operation inside the leaves of T_{tree}, we add a small rank structure to each leaf of T_{tree}. This rank structure is a set of counters that stores the number of ones in each block of S_T bits, where S_T is a parameter for the sampling interval. S_T determines the number of samples that are stored in each leaf and provides a space/time tradeoff for the local rank operation inside the leaves. Figure 8.3 shows an example of T_{tree} leaf with rank samples: the node is divided in chunks of fixed size, and the number of ones in each chunk is stored in the corresponding sample. Notice that the number of samples and the size in bits used to store each sample is prefixed by B and S_T.

Using the modified leaf structure, the operation to compute a local *rank* operation inside a leaf, $\text{rank}_1(T_\ell, p)$ now requires to sum the values of all samples previous to position p and perform a sequential search only on the fragment from the end of the last sample:

$$\text{rank}_1(N_\ell, p) = \sum_{i=0}^{p/S_T} s_i + \mathit{countOnes}(N_\ell[p - p \mod S_T, p])$$

8.1.1.2 Query operations in the dK^2-tree

All the queries supported by K^2-trees are based on *access* and *rank* operations to the bitmaps T and L. Our tree structures T_{tree} and L_{tree} support these basic operations and therefore all the queries supported by static K^2-trees can be directly applied

Algorithm 8.2 Find a cell in a dK^2-tree

```
1: function findCell(tree, r, c)
2:     p ← 0
3:     N_ℓ ← NULL
4:     b_before, o_before, rank ← 0
5:     while p < tree.T_tree.size do
6:         p ← p + computeChild(r, c, p)
7:         (N_ℓ, b_before, o_before) ← findLeaf (tree, p)
8:         T_ℓ ← N_ℓ.data
9:         if access(T_ℓ, p − b_before) then                ▷ access(tree, p)
10:            rank ← o_before + rank_1(T_ℓ, p − b_before)   ▷ rank_1(tree, p)
11:            p ← rank × K^2
12:        else
13:            return 0
14:        end if
15:    end while
16:    return access(T_ℓ, p − b_before)
17: end function
```

to dK^2-trees simply replacing the operations on T and L with their equivalents on T_{tree} and L_{tree}.

As an example, Algorithm 8.2 shows the complete process to access a single cell of the matrix in a dK^2-tree. The complete procedure is essentially equal to the navigation of a static K^2-tree. We assume we have a method *computeChild* that obtains the child that has to be traversed at each level depending on the row and column queried. This method only selects the branch to be traversed in the conceptual tree and does not depend on the underlying data structure, thus the computation is identical in dK^2-trees and static K^2-trees.

Then, at each level, the position p containing the current node is accessed and checked. In the dK^2-tree, the operation *findLeaf* is invoked to obtain the appropriate leaf N_ℓ and the counters b_{before} and o_{before}. The bit corresponding to position p is checked in N_ℓ, and if it is a 1 the traversal continues to the next level. A *rank* operation is performed (notice that o_{before} is already computed, so the *rank* operation must only compute the *rank* in N_ℓ) and the position in the next level is computed like in a static K^2-tree. When the position exceeds the size of T_{tree} we know we have reached the leaves of the conceptual tree and we simply access L_{tree} to retrieve the value of the desired bit. Notice that the only difference between the static and dynamic algorithms is the inner work for computing *rank* and *access*, that in the dK^2-tree is based on the lower-level operation *findLeaf*. All the operations supported by static K^2-trees can be adapted to the dK^2-tree implementation following the same principles.

Example 8.2: We want to check the value of the cell at row 9, column 6 in the dK^2-tree representation shown in Figure 8.1. Starting at the root node ($p = 0$), we compute the child that contains the cell. The first *computeChild* returns us the third child of the root node (offset 2). We call *findLeaf* (2), that returns us $(N_4, 0, 0)$. At N_4 we access the bit at offset $2-0 = 2$, and since it is a 1 we compute $rank = \text{rank}_1(N_4, 2-0) = 2$. The children of the node will be located at position $2 \times K^2 = 8$. The process is repeated again, comparing the new p in line 5 of the algorithm and repeating the process. Now we would traverse the second quadrant of the current node, so we need to *access* position $8+1 = 9$. *findLeaf* will lead us to the corresponding leaf *findLeaf*$(9) = (N_5, 8, 6)$. The procedure would be repeated until we eventually find the cell at position 42 in L_{tree}.

8.1.2 Update operations

In addition to the queries supported by static K^2-trees, dK^2-trees support update operations over the binary relation. First, relations between existing elements may be created or deleted (changing zeros of the adjacency matrix into ones and vice versa). Additionally, dK^2-trees support changes in the base sets of the binary relation (new rows/columns can be added to the binary matrix, and existing rows/columns can be removed as well).

8.1.2.1 Changing the contents of the binary matrix

Changes in the binary matrix are translated into a set of modifications in the conceptual K^2-tree representation of the matrix, leading to the creation or removal of branches in this conceptual K^2-tree. We will first describe the changes required in the conceptual tree and the bitmaps that are the actual representation of the tree. Then we will focus on how these changes in the bitmaps are implemented over the data structures T_{tree} and L_{tree}.

In order to insert a new 1 in a binary matrix represented with a K^2-tree, we need to make sure that the appropriate path exists in the conceptual tree that reaches the cell. The insertion procedure begins searching for the cell that has to be inserted. The traversal of the conceptual tree continues as usual until a 0 is found in the conceptual tree. Two cases may occur:

- The simplest case occurs when the 0 is found in the last level of the conceptual tree. In this case, the 0 is simply replaced by a 1 to mark the new value of the cell and the update is complete.

- If the 0 is found in the upper levels of the conceptual tree, a new path must be created in the conceptual tree until the last level is reached. First, the 0 is replaced with a 1 as in the previous case. Then, groups of K^2 bits must be added in the lower levels. After replacing the 0 with a 1, a *rank* operation is performed to compute the position in the next level. At the new position we

add K^2 0 bits and repeat the procedure, setting to 1 only the bit corresponding to the branch of the cell we are inserting. The procedure continues recursively until it reaches the last level in the conceptual tree.

Figure 8.4 shows an example of insertion in a conceptual tree. At a given level in the conceptual tree a 0 is found and replaced with a 1, and a new path is created adding K^2 bits to all the following levels. The new branch is highlighted in gray and the changes in the bitmaps of the K^2-tree are also highlighted.

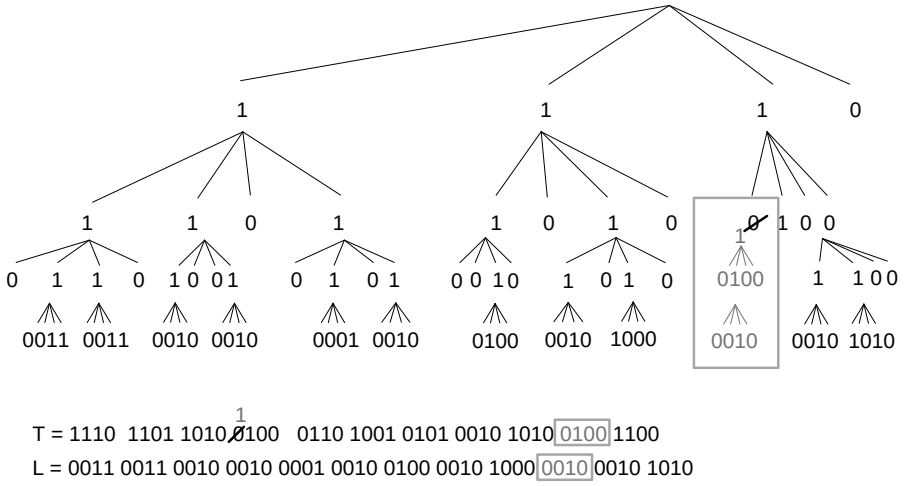


Figure 8.4: Insert operation in a K^2-tree: changes in the conceptual tree.

To change a 1 into a 0 in the binary matrix, we need to set to 0 the bit of the last level that corresponds to the cell. Additionally, the current branch must be checked to ensure that it contains at least a 1 and deleted otherwise. The procedure is similar to an insertion. First, the conceptual tree is traversed until we reach the position of the cell to be deleted. Then, the bit corresponding to that cell is set to 0. After this, we check the $K^2 - 1$ siblings of the current node. If at least one of the siblings is set to 1 the procedure ends. If all of them are set to 0, the current branch is empty so all the K^2 bits are deleted and we move one level up. In the upper level the operation is repeated, setting to 0 the bit of the current branch, checking its $K^2 - 1$ siblings and removing the K^2 bits if necessary.

Implementation

As we have seen, in order to support insertions and deletions in the dK^2-tree we only need to provide three basic update operations in T_{tree} and L_{tree}: flipping the value of a single bit, adding K^2 bits at a given position and removing K^2 bits starting at a given position. For example, Algorithm 8.3 shows the complete process

of insertion of new 1s in the matrix, that is similar to the search algorithm but relies on the additional operations *flip* and *append4*. We will focus in this section on the implementation details of the basic update operations over T_{tree} or L_{tree}.

Algorithm 8.3 Insert operation

```
    function INSERT(tree, r, c)
        p ← 0
        mode ← Search
        for l ← 0 to tree.nlevels - 1 do
 5:         p ← COMPUTECHILD(p, r, c, l)
            (N_ℓ, b_before, o_before) ← findLeafT(T_tree, p)
            T_ℓ ← N_ℓ.data
            if mode = Search then
                if access(T_ℓ, p − b_before) = 0 then
10:                 FLIP(T_ℓ, p − b_before)
                    mode ← Append
                end if
            else
                APPEND4(T_ℓ, p − b_before)
15:             FLIP(T_ℓ, p − b_before)
            end if
            rank ← o_before + rank_1(T_ℓ, p − b_before)          ▷ rank_1(tree, p)
            p ← rank × K^2
        end for
20:     l ← tree.nlevels
        p ← p − tree.T_tree.length
        p ← COMPUTECHILD(p, r, c, l)
        (N_ℓ, b_before) ← findLeafL(L_tree, p)
        T_ℓ ← N_ℓ.data
25:     if mode = Search then
            if access(T_ℓ, p − b_before) = 0 then
                FLIP(T_ℓ, p − b_before)
                mode ← Append
            end if
30:     else
            APPEND4(T_ℓ, p − b_before)
            FLIP(T_ℓ, p − b_before)
        end if
    end function
```

To flip a single bit, we first retrieve the leaf N_ℓ containing the desired position (notice that in all the algorithms for insertion and deletion the position has already

been accessed before we need to change its value). The bit is changed in the bitmap of N_ℓ and its rank directory is updated (adding or subtracting 1 to the value of the appropriate counter). Finally, if we are updating T_{tree}, the o-counters in the ancestors of N_ℓ must be updated to reflect the change. To perform this operation efficiently, the current entry at each level of T_{tree} and L_{tree} is stored during tree traversal, so that after any change the entries in the ancestors can be updated immediately.

To add K^2 bits at a given position, we first compute the leaf N_ℓ that contains that position. The K^2 bits are inserted in the bitmap of N_ℓ directly, and the counters in the rank directory must be updated accordingly. In this case, all the counters from the current position until the end of the leaf must be recomputed to take into account the bits that have been moved. Finally, the b- and o-counters in the ancestors of N_ℓ must also be updated. Every time we insert K^2 bits we are creating a new branch that will contain a single 1 so the b- and o-counters are increased by K^2 and 1 respectively.

When a leaf of T_{tree} or L_{tree} reaches its maximum node size we split it in two nodes, always keeping groups of K^2 siblings in the same leaf. This is propagated to the parent of the leaf, causing the insertion of a new entry pointing to the new node and updating the b- and c-counters accordingly.

To achieve better space utilization the dK^2-tree can store nodes of different maximum sizes. Given a base node size B, we allow a number e of partial expansions of the node before splitting it. In practice, for fixed B and e, T_{tree} and L_{tree} may contain nodes of size B, $B + \frac{B}{e+1}$, $\cdots$, $B + \frac{(e)B}{e+1}$ (we call them class-0, $\cdots$, class-e nodes. If a node overflows and it can be expanded, its contents are reallocated in a node of the next maximum capacity updating the P pointer in its parent if necessary. If a fully-expanded node overflows, it is split into two class-0 nodes and the parent entries are updated accordingly.

Example 8.3: The changes in the conceptual tree described in Figure 8.4 are translated in the actual data structure as shown in Figure 8.5. First the bit at position 12 is flipped. This causes a change in the second leaf of T_{tree} (the rank counter in the leaf node that includes that position must also be updated) and all the o-counters in the path to the root of T_{tree} are increased by one. Then we move to the position where the new 4 bits must be added, that falls in the last leaf of T_{tree}. In this case, we add the new K^2 bits and update both the b- and o-counters in the path to the root. The process is repeated in L_{tree}, adding the corresponding K^2 bits to a leaf node and updating the b-counters in the path to the root. In this example we consider that the leaf node updated in L_{tree} can allocate the new K^2 bits. If the new K^2 bits do not fit in the node, a new node with the next allowed size should be allocated. In the worst case, the leaf node would be split, and a new entry added to its parent. In this case, as all entries in its parent are full, the parent node would also be split and its entries redistributed.

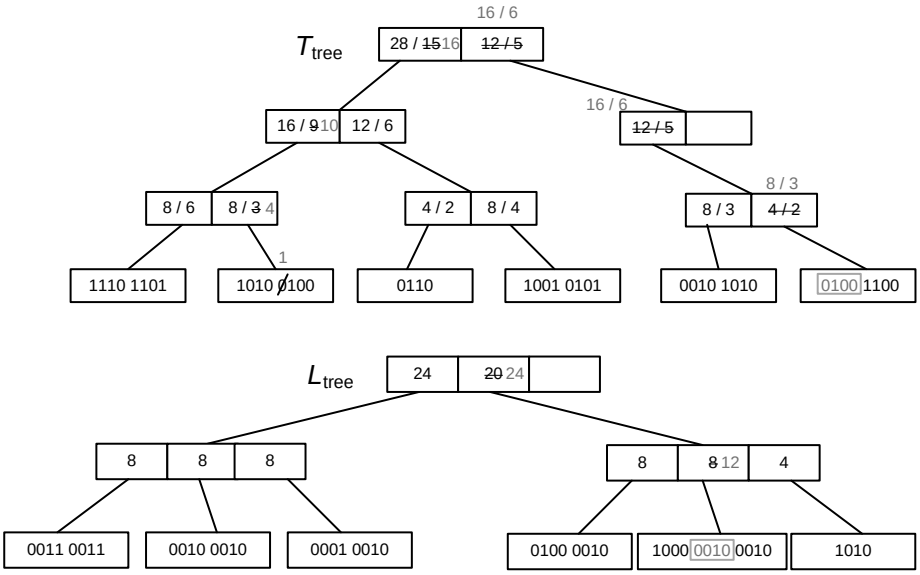


Figure 8.5: Insert operation in a dK^2-tree: changes in the data structures.

8.1.2.2 Changes in the rows/columns of the adjacency matrix

The dK^2-tree supports the insertion of new rows/columns in the adjacency matrix it represents, as well as deletion of existing rows/columns. These operations, although less frequent than changes in the contents of the matrix, are required to provide a fully-functional representation of binary relations. The dK^2-tree supports efficiently insertions of new elements at the end of the current matrix (that is, insertions at unspecified positions).

The insertion of new rows/columns to the adjacency matrix represented by a dK^2-tree can be easily supported using the following property: if the size of the matrix is not a power of K it is virtually expanded to the next power of K (recall Section 3.2). This means that a K^2-tree already contains in most cases a number of unused rows/column at the end of the matrix. In practice, in a dK^2-tree we can explicitly store the actual size of the matrix and consider the remaining rows and columns as *unused*. When a new row (column) is needed, the actual size of the matrix is increased and the first unused row (column) becomes available.

When the size of the matrix becomes exactly a power of K no unused rows/columns are available. In this case, the dK^2-tree is modified to expand the current $K^n \times K^n$ matrix to a size $K^{n+1} \times K^{n+1}$. The current matrix will become the top-left quadrant of the new matrix. In order to do this expansion, we only need to add a new root level to the conceptual tree that represents the matrix, whose

first child will be the current root. This operation requires simply the insertion of K^2 bits 1000 at the beginning of T_{tree}.

To delete an existing row/column, the procedure is symmetric to the insertion. The last row/column of the matrix can be removed by updating the actual size of the matrix and zeroing out all its cells. Rows/columns at other positions may be deleted logically, adding them to a list of deleted rows and columns after zeroing all their cells. In this case, the deleted rows and columns may be reused when new rows are inserted.

The insertion of new columns at specific positions in the matrix is not supported by dK^2-trees. This operation is difficult to implement in the dK^2-tree because of the regular decomposition of space it uses. Additionally, the K^2-tree and its variants rely on the clusterization and regularity of matrices to obtain good compression results, so the logical deletion of rows in the middle of the matrix may have significant effects in compression. This means that a dK^2-tree may need additional data structures to represent binary matrices that require these operations (a permutation can be used, for instance, to map real element numbers to actual row/column positions).

8.2 Matrix vocabulary

The K^2-tree can be improved using a matrix vocabulary to compress the bitmap L, as explained in Section 3.2.2. This improvement replaces the plain bitmap L with a sequence of variable-length codes plus a matrix vocabulary consisting of a single table that stored all the submatrices. Given any code, its offset in the vocabulary could be retrieved and the corresponding submatrix recovered. In dK^2-trees, the leaves in L_{tree} could use the same encoding procedure to store a sequence of matrix codes. In this section we show how a dynamic implementation of the matrix vocabulary can be used to compress L_{tree} using ETDC (recall Section 2.2.4).

First, in order to use a matrix vocabulary in the dK^2-tree we need to handle a dynamic matrix vocabulary. This means that we should be able to add and remove entries from the vocabulary. In order to do this, we need a data structure that allows us to check whether a given matrix already exists in the vocabulary. We built a simple implementation that stores a hash table H to look up matrices. An array V stores the position in H where each matrix is stored. Finally, we add another array F that stores the frequency of each matrix. An additional list V^{empty} stores the codewords that are not being currently used.

In order to store variable-length codes in the leaves of L_{tree} some details must be taken into account: first, the actual number of bits and ones in a leaf is no longer the same as the *logical* values stored in b- and o-counters. This does not affect the tree structure because the actual size of each leaf node is stored in the *size* field of its header, and this is the value used to determine when to expand or split a leaf, while the values in the counters are still used as before to access the appropriate

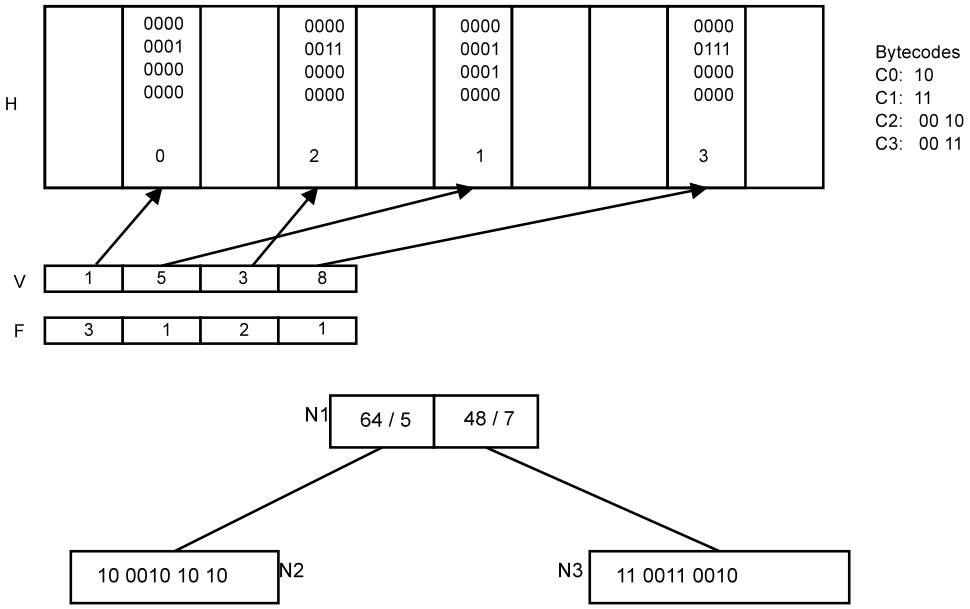


Figure 8.6: Dynamic vocabulary management: vocabulary (top) and L_{tree} (bottom).

leaf.

Example 8.4: Figure 8.6 shows an example with a complete vocabulary representation containing 4 different matrices. The leaves of L_{tree} store now a sequence of variable-length codes represented with ETDC (we use chunk size $b = 2$ in this example to obtain the simpler codewords shown in the figure). Notice that leaves N2 and N3 store only the codewords corresponding to the matrices, but the b- and o-counters in internal node N1 still refer to the logical size of the leaf: the entry pointing to N2 contains 64 bits (4 submatrices of size 4×4) and 5 ones. The submatrices are stored in a hash table that stores for each matrix its offset in the vocabulary (that can be easily translated into a codeword). The array V allows us to find the matrix from the codeword, storing the actual address of the entry in H (in this case, since H is backed by an array, V stores the position in the array of the corresponding entry in H). F stores the actual frequency of each submatrix in the vocabulary.

The algorithm to *access* a position in L_{tree} with matrix vocabulary must be modified to use the encoding. After *findLeaf* returns, we obtain a *logical* offset in the leaf. In order to find the value of that position we must traverse the sequence of variable-length codes stored in the leaf until we find the one that corresponds to the desired position. When we find the code that contains the desired position, we

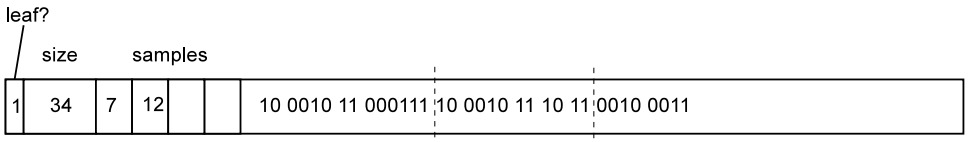


Figure 8.7: Leaf node in L_{tree} with rank directory, for $S_L = 4$.

retrieve the actual matrix it represents: the codeword is translated into an array index, and V is used to retrieve the actual matrix. For example, suppose we want to access position 21 in the example of Figure 8.6. Our *findLeaf* operation would take us to node N2, offset 21. To obtain the actual matrix we would traverse the codes in N2, taking into account the actual size of each submatrix (16 bits), so our offset would be at position 5 in the second submatrix. We go over the code `10` and find the second code `0010`. To find the actual matrix, we convert this code to an offset (2) and access $V[2]$ to locate the position in H where the matrix is actually stored (3, second non-empty position). Finally, in H we can access bit 5 in the matrix bitmap (0).

The main difference with a static implementation is the need to sequentially traverse the list of variable-length codes. We can reduce the overhead of this sequential traversal adding to the leaves of L_{tree} a set of samples that store the actual offset in bytes of each S_L-th codeword. The idea is similar to the sampling used for *rank* in the leaves of T_{tree}, and an example is shown in Figure 8.7. With this improvement, to locate a given position we can simply use the samples to locate the previous sampled codeword and then start the search from the sampled position.

Update operations. Update operations now require to add, remove or modify variable-length codes from the leaves of L_{tree}. All the update operations start by finding the real location in the node where the variable-length code should be added/removed/modified.

To insert groups of K^2 bits we need to add a new codeword. The matrix corresponding to the new codeword is looked up in H, adding it to the vocabulary if it did not exist and increasing its frequency in F. Then, the codeword for the matrix is inserted in the leaf, updating the counters in the node and its ancestors.

To remove groups of K^2 bits in a leaf node of L_{tree} a codeword must be removed: we locate the codeword in L_{tree}, decrease its frequency in F and then we remove the code from the leaf node, updating ancestors accordingly. If the frequency of the codeword reaches 0, the corresponding index in V is added to V^{empty}. When new entries must be added to the vocabulary V^{empty} will be checked to reuse previous entries and new codes will only be used when V^{empty} is empty.

To change the value of a bit in L_{tree} we need to replace existing codewords. First, the matrix for the current codeword is retrieved in H and its frequency is decreased in F. We look up the new matrix in H. Again, if it already existed its

frequency is increased and if it is new it is added to H and its frequency set to 1. Then, the codeword corresponding to the new matrix is inserted in L_{tree} replacing the old codeword.

Following the example of Figure 8.6, suppose that we need to set to 1 the bit at position 21 in L_{tree}. *findLeaf* would take us to N2, where we have to access the second bytecode `00 10` at offset $21 - 16 = 5$. This bytecode (C2) corresponds to offset 2 in the ETDC order. We would access $V[2]$ to retrieve the corresponding matrix. The operation would require us to transform the matrix as follows:

$$\begin{matrix} 0000 \\ 0011 \\ 0000 \\ 0000 \end{matrix} \longrightarrow \begin{matrix} 0000 \\ 0111 \\ 0000 \\ 0000 \end{matrix}$$

The new submatrix created flipping the bit at offset 5 would be checked in H, and we would find that it already exists at position 3 in V with frequency 1. Hence, we would need to update the leaf node replacing the old codeword `00 10` with the new codeword C3: `00 11`. This change is performed in the leaf node, and the vocabulary is also updated: the frequency for C2 would be decreased to 1 and the frequency for C3 would be increased to 2.

8.2.1 Handling changes in the frequency distribution

The compression achieved by the matrix vocabulary depends heavily on the evolution of the matrix frequencies. As update operations are executed in the dK^2-tree, the distribution of the submatrices may change significantly and the efficiency of the variable-length codes will degrade. The simplest approach to obtain a reasonably efficient vocabulary is to use a *precomputed vocabulary*, precomputing a fraction of the matrix to obtain the frequency distribution of submatrices and build a preliminary vocabulary. Alternatively, a tentative vocabulary may be provided by the user during construction if the properties of the matrix to be represented are well known or the matrix is expected to be very regular.

To obtain the best compression results, when the frequency of a submatrix changes too much its codeword should also be changed to obtain the best compression results. This is a process similar to the vocabulary adjustment in dynamic ETDC (DETDC, recall Section 2.2.4.1). However, in a dK^2-tree we are presented with a space/time tradeoff: each time we want to change the codeword of a submatrix, we must also update all the occurrences of the codeword in the leaves of L_{tree}.

To maintain a compression ratio similar to that of static K^2-trees we can use simple heuristics to completely rebuild L_{tree} and the matrix vocabulary: we can rebuild every p updates, or count the number of inversions in F. To rebuild L_{tree}, we must sort the matrices in H according to their actual frequency and compute the

optimal codes for each matrix. Then we have to traverse all the leaves in L_{tree} from left to right, replacing the old codes with optimal ones. Notice that the replacement can not be executed completely in place, because the globally optimal codes may be worse locally, but the complete process should require only a small amount of additional space. After L_{tree} is rebuilt, the old vocabulary is replaced with the optimal values.

8.2.1.1 Keeping track of the optimal vocabulary

An alternative to the simple heuristics to rebuild the matrix vocabulary is to keep track of how good the current compression is. To guarantee that the compression of L_{tree} is never too far from the optimum, we can keep track of the actual optimum vocabulary. In this section we propose an enhanced vocabulary representation in which we maintain updated at all times an optimal vocabulary. This vocabulary representation is similar to the adaptive encoding used in DETDC, in the sense that it uses data structures to maintain the optimal vocabulary on update operations. However, in our case it would be unfeasible to change the actual vocabulary each time the length of a codeword changes (since changing the codeword of a matrix implies rebuilding L_{tree} to replace the old codes with the optimal ones). Hence, we store the optimal vocabulary to keep track of the space efficiency but still use a suboptimal one.

To keep track of the optimal vocabulary we store, in addition to H and F, a permutation VP between the current vocabulary and the optimal one: $VP(i)$ gives the optimal position of the codeword at offset i, while $VP^{-1}(i)$ gives the current position given the optimal position. This permutation will be updated when necessary and will allow us to keep V and F always sorted in descending order of frequency (that is, according to the optimal vocabulary). Figure 8.8 shows the data structures required to represent the same vocabulary of Figure 8.6 using the new method.

In this representation, the procedure to find the codeword for a matrix and to obtain the matrix for a codeword is essentially the same explained for the basic vocabulary: to find the codeword of a matrix, we look up the matrix in the hash table to obtain its offset in the optimal vocabulary; then we use VP^{-1} to compute the offset in the current vocabulary. To find the matrix for a given codeword we compute the optimal offset for the codeword using VP and then use V to find the position of the matrix.

To keep track of the changes in frequencies, we also build an array *Top* that stores, for each different frequency, the position in V of the first codeword with that frequency. The array *Top* is used to perform codeword swaps when frequencies change, as it was performed in DETDC. If the frequency of a matrix at index i in V changes from f to $f+1$, the new optimum position for it would be the position $Top[f]$. For example, following the example of Figure 8.8, if a new codeword C2 (`00 10`, corresponding to offset 2 in the current vocabulary, therefore offset VP (2)=1 in

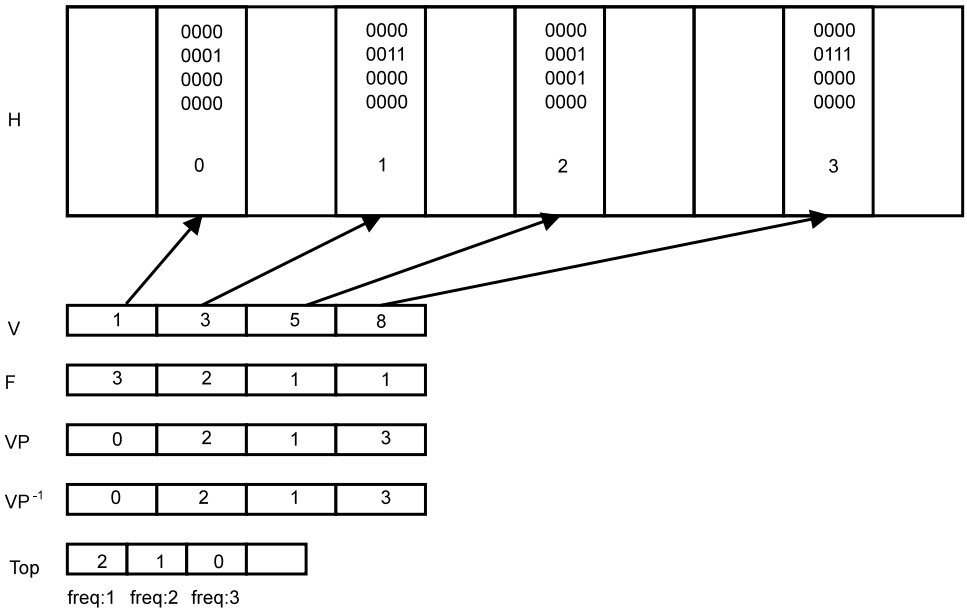


Figure 8.8: Extended vocabulary to keep track of the optimal codewords.

the optimum one) is added to the vocabulary, the value of $F[1]$ must be increased to 3. To keep the vocabulary sorted by frequency we would simply swap the current offset 3 at $V[1]$ and $Top[1] = 1$ (i.e., we exchange the current matrix with the first matrix with frequency 2). The indexes in V, VP and H would be updated to reflect the change. The case when the frequency of a matrix decreases from f to $f-1$ is symmetric: we swap the current position of the matrix with $Top[f-1]+1$, updating the corresponding indexes in F and VP.

The use of the extra data structures allows us to control precisely how much space is being wasted at any moment, since we know the length of the sequence of codewords $size_{cur}$ and we can also know the size the optimum sequence would have ($size_{opt}$). This means that we can set a threshold $reb_{L_{\text{tree}}}$ and rebuild L_{tree} when the ratio $\frac{size_{cur}}{size_{opt}}$ surpasses it.

In order to physically store all the data structures required for the dynamic vocabulary, we resort to simple data structures that can be easily updated. H is a simple hash table backed by an array. V and F are extendible arrays. If we want to set the threshold $reb_{L_{\text{tree}}}$we need additional data structures for VP and Top. Top can be implemented using an extendible array and two extendible arrays can store VP and its inverse. The goal of these representations is to provide efficient update times (recall that each update operation in the dK^2-tree will always lead to a change L_{tree}, that will cause at least one frequency change in the vocabulary).

8.3 Improving accesses to the dK^2-tree

An interesting property of the navigation of K^2-trees is that sequences of consecutive accesses to the bitmaps T and L follow well-defined patterns. All the queries supported by K^2-trees start at the root of the conceptual tree (i.e., at the beginning of bitmap T). The traversal of the conceptual tree jumps to lower levels until the last level of the conceptual tree (bitmap L) is reached. Each traversal operation in the conceptual tree becomes a jump to positions to the right of the current one in the bitmap T, until finally the last level is reached and the bitmap L is accessed instead.

In this section we propose a modification of the access mechanisms of the dK^2-tree to speed up queries. The idea is simply to modify the *findLeaf* operation to be able to search the new leaf from a previously accessed leaf instead of from the root of T_{tree} or L_{tree}. This modification will reduce the cost of traversing the complete tree from the root for each *access* to the dynamic bitmap: instead, we will find the shortest path from a leaf to another. This modification will make accesses more efficient in the dK^2-tree because accesses to positions follow a well-defined pattern, as we explained: each query in the K^2-tree is based on a top-down traversal of the conceptual tree, that translates into accesses to increasing positions in the bitmap T. In the case of range operations, that may access many children of each node, query algorithms are expected to perform accesses to very close positions, at least in the first levels of the conceptual tree.

To be able to start search from a previously accessed leaf node, the new *findLeaf** operation must store the relevant data from the current search that allows us to traverse T_{tree} or L_{tree} and find a path to the next leaf node. For each level of T_{tree} (L_{tree}) some information is stored in order to be able to perform backtracking from the last leaf. An array $levelData[T_{\text{tree}}.depth]$ is kept, that stores for each level of T_{tree} an entry $\langle N, e, s, b, o\rangle$, where N is a pointer to the T_{tree} node accessed at that level, e the entry that was traversed (if N is an internal node), s is the number of bits covered by N[1] and b and o are the values of b_{before} and o_{before}. This information will be kept between consecutive *findLeaf** operations.

To locate the leaf for a new position p, we start from the previously accessed leaf and check if the new position p is in the current leaf (i.e. if $b \leq p < b + s$). If the new position is in the current leaf we can simply return immediately. If p is not in the current leaf we navigate to the upper level of T_{tree} (the previous entry in *levelData*) until at some point the inequality $b \leq p < b + s$ holds (i.e. the new position is in the fragment covered by the current node). At this point, we know the b_{before} and o_{before} until the beginning of the current entry, and we can resort to the top-down traversal used in the basic *findLeaf* operation.

[1] We assume that s is stored for convenience. It can be easily computed: in the root node, s is the total size of the bitmap, that can be physically stored in the dK^2-tree; in lower levels, s is the value of the b-counter in the parent node

The effect of this modification is simply that the cost of each new access operation now depends on the distance to the previous access. In the original algorithm, each access required us to perform a complete top-down traversal of the tree; now, we can start from a leaf of the tree that has been accessed recently and locate the new leaf and offset without traversing all the levels of T_{tree} or L_{tree}. In the best case, the new access is so close to the previous one that the new position falls actually in the same leaf node of the tree. In this case, we do not need to traverse internal nodes and can return immediately the current leaf. This makes our indexed solution a good tool for reducing the cost of the operations: at least in the upper levels of the tree, and in queries that involve ranges of values, the cost of many access and rank operations in T_{tree} can be reduced to the cost of performing the operation in the leaf node, that is comparable to the cost of a static *rank* operation.

8.4 Analysis

In this section we will compare the space and time costs of the dK^2-tree with those of a static K^2-tree. As we explained previously in this chapter, the dK^2-tree aims to provide good results in practice, but it does not provide good theoretical guarantees when compared with the static data structure.

The dK^2-tree stores essentially a dynamic implementation of the bitmaps T and L of the static K^2-tree. The leaves of T_{tree} and L_{tree} store the bits of T and L and may be partially empty, so the overall space required is $O(|T| + |L|)$. The main variable that affects compression in the dK^2-tree is the number of node expansions e, that affects node occupancy. The actual overhead required by the dK^2-tree depends in a smaller factor on the sampling intervals S_T and S_L and the block size B. In practice, the total space can be upper bounded by a linear overhead with a small factor, at least if we do not consider a matrix vocabulary. The additional cost of the matrix vocabulary is difficult to measure, since it depends heavily on the input data. However, we can assume that, if we want to keep track of the optimal vocabulary, the overhead to pay is several times the space of the vocabulary in a static K^2-tree (recall from Section 3.2 that the vocabulary in a K^2-tree only requires the table with submatrices, while our solution requires in addition a hash table H, a permutation VP and the array Top). In general, the dK^2-tree with matrix vocabulary may be competitive with the static K^2-tree as long as the size of the vocabulary is a small fraction of the overall size of the representation.

The cost of query operations in dK^2-trees can be calculated in relation to the cost in a static implementation. We can reduce the overhead required by our dynamic representation to the overhead required by *findLeaf* operations and local *access* and *rank* operations inside leaves of T_{tree} and L_{tree}, compared with the constant-time *access* and *rank* times in T and L. The main overhead in dK^2-trees is the *findLeaf* operation, since we can reduce the cost of local *access* and *rank* operations using samples in the leaves of T_{tree} and L_{tree}. In the worst case, a *findLeaf* operation

requires us to traverse a full path in T_{tree} or L_{tree} from the root to a leaf node, performing $O(B/\log n)$ operations at each internal node. When a leaf node is reached, local *access* operations are $O(1)$ in T_{tree} and L_{tree} or require traversing $O(S_L)$ codewords in a compressed L_{tree}. Local *rank* operations require adding up the values of the samples in the leaf and computing the rank value for the remaining interval, thus $O(B/S_T + S_T)$ time. In our implementation we will consider relatively large values of B so that the space overhead caused by internal nodes is small.

An important consideration regarding practical efficiency is the fact that searches implemented using the *findLeaf** operation can start the search from the previous leaf accessed, thus this search very efficient at least for the first levels. If the queries executed in the dK^2-tree lead to many close accesses in T_{tree} and L_{tree}, the total cost of the operations can be reduced almost exclusively to the cost of the local operations in leaf nodes.

The cost of update operations in the dK^2-tree can also be decomposed in the cost in internal nodes and cost in leaves. In our representation any update operation will require a *findLeaf* operation, exactly like a search, and a modification in a leaf node. To update a bit a leaf node of T_{tree} we need to flip the bit and update its corresponding sample. On the other hand, to insert/remove bits we need to shift the bit sequence stored in the node, requiring $O(B/\log n)$ time, and recompute all the samples in the worst case. However, notice that in practice the cost of recomputing all the samples is amortized by the fact that in the K^2-tree each time a position is updated we usually need to find the *rank* value of that position to navigate to the next level of the conceptual tree.

8.5 Experimental evaluation

In this section we experimentally test the efficiency of our new dynamic representation. Our goal is to demonstrate its capabilities to answer simple queries in space and time close to those of the static K^2-tree data structure. We will start by comparing the dK^2-tree with static K^2-trees to compress Web graphs, the original field of application of K^2-trees. This will serve as a baseline for the expected efficiency of dynamic K^2-trees in comparison with a static version. We choose Web graphs to compare the dK^2-tree with the original K^2-tree because they are the original application of static K^2-trees and all the improvements to K^2-trees have been proved to be very efficient in this kind of graphs, particularly the compression of the last levels using a matrix vocabulary. Nevertheless, this experimental evaluation is a proof of concept and a reference of the overhead required by our dynamic representation that will be completed in the next chapter where we will move to other domains, such as the compression of RDF graphs, raster images or temporal graphs, where representations based on K^2-trees are also efficient but a dynamic implementation that allows changes to the underlying matrix would be useful in many cases.

All the experiments in this section are run on an AMD-Phenom-II X4 955@3.2 GHz, with 8GB DDR2 RAM. The operating system is Ubuntu 12.04. All our code is written in C and compiled with gcc version 4.6.2 with full optimizations enabled.

8.5.1 Parameter tuning

The main parameters used to adjust the efficiency of our structure are *1)* the sampling period s in the leaves of T_{tree} (S_T) and L_{tree} (S_L), *2)* the block size B, *3)* the number of partial expansions e and *4)* the utilization of a matrix vocabulary, and if so the selection of an appropriate value of K' in the last level. This last parameter has a great effect on the overall compression in static K^2-trees, but is highly dependent on the dataset to be represented, its size and the regularities it contains. We will first focus on the effect of the first group of parameters, and leave the study on the effect of the matrix vocabulary for later, since the utilization or not of a matrix vocabulary should be independent of the tuning of the first parameters.

To prove the validity of the parameter selection we show the results obtained by the dK^2-tree representation of a Web graph dataset, eu-2005. It is a small web graph with 19 million edges, already used in Chapter 5, that provides an example, but the parameter tuning is very similar in other datasets used in following sections. We just use this dataset as an example to show the effects of changes in the tuning parameters. We do not use a matrix vocabulary in this example.

Figure 8.9 shows the evolution of the dK^2-tree size and the creation and rebuild time depending on each parameter. The dK^2-tree size, expressed in MB, is the overall memory used by the data structure. The creation time is the time required for building the dK^2-tree structure from an adjacency list representation of the dataset. We build the datasets inserting each edge separately, so this time provides an estimation of the update time of the structure. Finally, the rebuild time is the time required to find all the 1s in the adjacency matrix, thus rebuilding the input dataset from the dK^2-tree representation. This process of rebuilding the original dataset is performed by a single range query that covers the complete matrix, and it is shown again as a rough estimation of the expected evolution of query times. Both times are shown in microseconds per element inserted/retrieved.

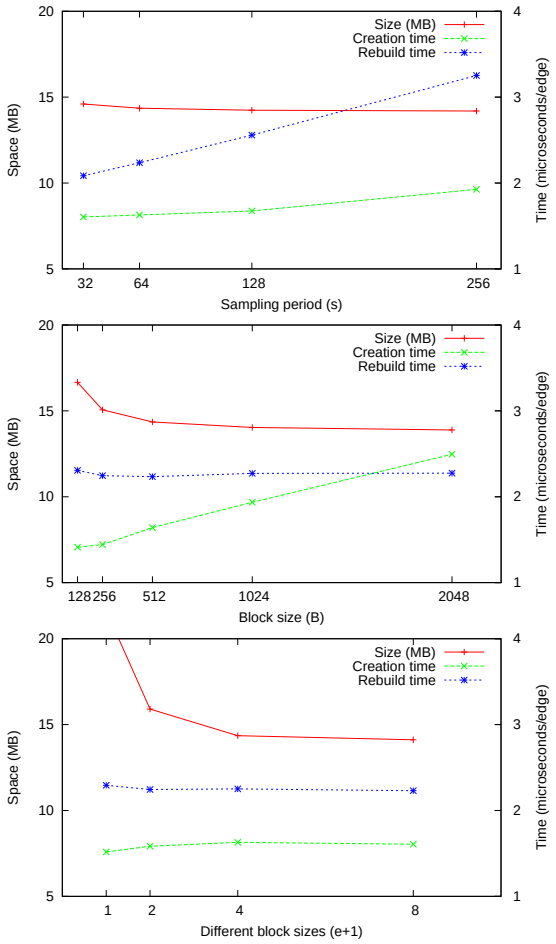


Figure 8.9: Evolution of space/time results of the dK^2-tree changing the parameters s, B and e.

The top plot in Figure 8.9 shows the space/time results obtained for different values of the sampling interval s. The times shown correspond to fixed values $B = 512$ bytes and $\#classes = e + 1 = 4$, but the tradeoff is similar for different values. Notice that the sampling period s has an interesting effect in the dK^2-tree. Smaller values of s increase the size of the trees slightly (more samples are stored). The small increase in size is compensated by a considerable reduction in query time, because the rank operation in leaves (*rankLeaf*) has a cost directly proportional to s. Additionally, update operations can also be improved by using smaller values of s. Blocks with more samples are more costly to update when their contents change,

but the recomputation of the samples is not so expensive and it is only performed if the node contents actually change. On the other hand, the *rankLeaf* operation must always be performed at all the levels of the conceptual tree, and its cost is greatly reduced when using smaller values of s. This effect allows us to choose a sampling period as small as necessary to obtain a $rankLeaf$ operation comparable to a static rank, with only a minor increase in the size of the dK^2-tree.

The middle plot in Figure 8.9 shows the results for different values of B, for fixed $s = 128$ bytes and #classes=e+1=4. The block size B provides a clear space/time tradeoff. Small values of B yield bigger dK^2-trees because the constant overhead added by the node headers and the block manager becomes more relevant. As the value of B increases, the query times are similar thanks to the sampling but updates become more costly. In our experiments we will choose values of $B = 256$ or $B = 512$ to obtain the best space results with small penalties in update times.

Finally, in the bottom plot of Figure 8.9 we show the evolution with e (or, equivalently, with the number of different block sizes $e + 1$), for fixed $s = 128$ and $B = 512$. If we use a single block size ($e + 1 = 1$), the node utilization is low and the figure shows poor space results. For a relatively small number of block sizes the space results improve fast, and we can see only minor changes in the creation and rebuild times. As we can see, using just 4 different block sizes yields a considerable improvement in space and allows us to use simple techniques for the memory management.

8.5.2 Effect of the matrix vocabulary

The creation of a matrix vocabulary to compress the last level of the tree (L) reduced significantly the space requirements of static K^2-trees, as explained in Section 3.2. In this section we experimentally evaluate our proposals for the compression of L_{tree} in a dK^2-tree, that follow the same principles used in static K^2-trees but are more challenging because we must also update the vocabulary.

To test the efficiency of using a matrix vocabulary in the dK^2-tree in comparison with a static K^2-tree, we build the data structure for the same datasets with and without a matrix vocabulary and for different values of the parameter K'. We compare the static and dynamic representations in two different Web graph datasets, already used in Section 5.6: the *indochina-2004* dataset, with 200 million edges, and the *uk-2002* dataset, with 300 million edges. In all cases we build a hybrid variant of the K^2-tree or the dK^2-tree, with $K = 4$ in the first 5 levels of decomposition and $K = 2$ in the remaining levels. We first build a version with no matrix vocabulary as a baseline and then build the same representations with a matrix vocabulary for matrix sizes $K' = 4$ and $K' = 8$. In the dK^2-tree we choose a block size $B = 512$, $e = 3$ (4 different block sizes) and $s = 128$. The static K^2-tree representation uses a sampling factor of 20 for its rank data structures, hence requiring an additional 5% space.

We design an experimental setup to test the efficiency of the simplest version of the matrix vocabulary (that does not keep track of the optimum values) and the more complex one, that keeps track of the optimum and is able to determine how much compression overhead is obtained in L_{tree}. We use in the dK^2-tree the most complex version of the matrix vocabulary, that keeps track of the optimum vocabulary and rebuilds the complete vocabulary when the total size is 20% worse than the optimum. Additionally, and in order to avoid multiple rebuild operations in the first few insertions, we set a threshold of 100 KB for the size of L_{tree}, so that the vocabulary is only checked (and rebuilt if necessary) when L_{tree} reaches that size. An interesting result of this setup is that, when a matrix vocabulary is used in the dK^2-tree, the graphs are only rebuilt once, when the size of L_{tree} reaches the threshold. In this case, the small-scale regularities in the adjacency matrices of Web graphs ensure that, once a small fragment of the adjacency matrix has been built, the resulting matrix vocabulary becomes good enough to compress the overall matrix with a relatively small penalty in space. Given this result, we measure the overall memory utilization of both data structures considering in the dK^2-tree two different scenarios: the space required by the simplest version of the vocabulary (including H and F but not the additional data structures to keep track of the optimum vocabulary) (*dynamic*) and the total space required to keep track of the optimum vocabulary (*dynamic-complete*).

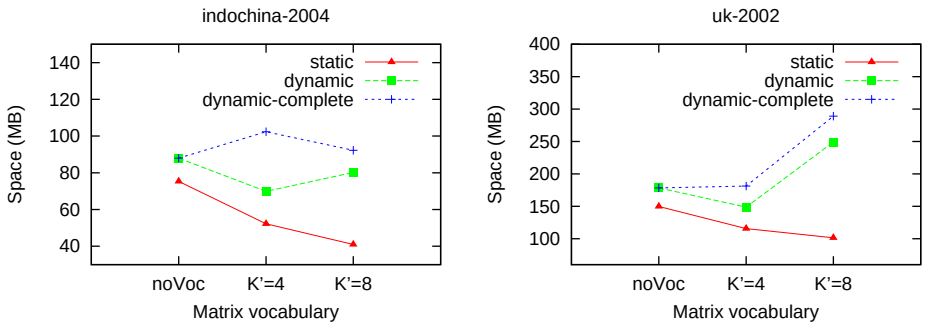


Figure 8.10: Space utilization of static and dynamic K^2-trees with different matrix vocabularies in Web graphs.

Figure 8.10 shows the evolution of the space utilization for both datasets required by original K^2-trees (static) and a dK^2-tree (dynamic). The dK^2-tree space utilization is close to that of the static K^2-tree when no matrix vocabulary is used ($noVoc$). The space overhead of dK^2-trees, around 20%, is mostly due to the space utilization of the nodes. Static K^2-trees obtain better compression for larger values of K', reaching their best space utilization when $K' = 8$. On the other hand, the dK^2-tree improves its space utilization for small K' but it is not

able to take advantage of larger values of K'. However, when we try to keep track of the optimum vocabulary, the overall size of the dK^2-tree becomes worse than the simpler approach. This is due to the additional data structures required, particularly the array *Top*, that add a very significant space overhead to the simpler matrix vocabulary. Considering these results, a simpler strategy to maintain a "good" matrix vocabulary (such as using a predefined matrix vocabulary extracted from experience of simply rebuild after x operations) may be the best approach in many domains. On the other hand, the strategy to keep track of the optimum vocabulary is still of theoretical interest and could be of application in domains where the matrix vocabulary is expected to be very small.

Dataset	K'	Vocabulary size (% of total)	
		Static	Dynamic
indochina-2004	4	0.12	2.77
	8	19.20	46.12
uk-2002	4	0.08	1.30
	8	19.88	59.40

Table 8.1: Percentage of the total space required by the matrix vocabulary in Web graphs.

The space results with no matrix vocabulary and even with matrix vocabulary and small K' show that the main difference between the static K^2-tree and the dK^2-tree when using a matrix vocabulary is the space required to store the vocabulary, that in the dK^2-tree must be searchable and updatable. Table 8.1 shows the evolution of the vocabulary size in relation to the overall size of the representation for the studied datasets. As we can see, the dynamic vocabulary used in dK^2-trees requires a much larger fraction of the overall space for all values of K'. For small K', the difference between the static and dynamic vocabularies is larger due to the larger number of entries in the vocabulary, that must be stored in H and F. For $K' = 8$ the dK^2-tree already uses much more space than static K^2-trees, and the space required to store the complete vocabulary can be more than half the total space. This makes the utilization of a matrix vocabulary difficult for large values of K' in the dK^2-tree. As we have shown, some improvements can be obtained using matrix vocabularies with $K' = 4$ in Web graphs, but even with this improvement in space the space utilization of the dK^2-tree with $K' = 4$ essentially doubles the results for the best static K^2-tree representations using a matrix vocabulary with $K' = 8$.

8.5.3 Access times

As a proof of concept of the efficiency of the dK^2-tree we compare its query efficiency with static K^2-trees in the context of Web graphs. We choose the most usual query in this domain, namely, a *successor* query that asks for the direct neighbors of a specific node (that is, all the cells with value 1 in a specific row of the adjacency matrix). For each dataset we run all possible successor queries, that is, we run a set of queries including all the possible nodes, and measure the average query times in μs/query. We compare the query times between static K^2-trees and the dK^2-tree.

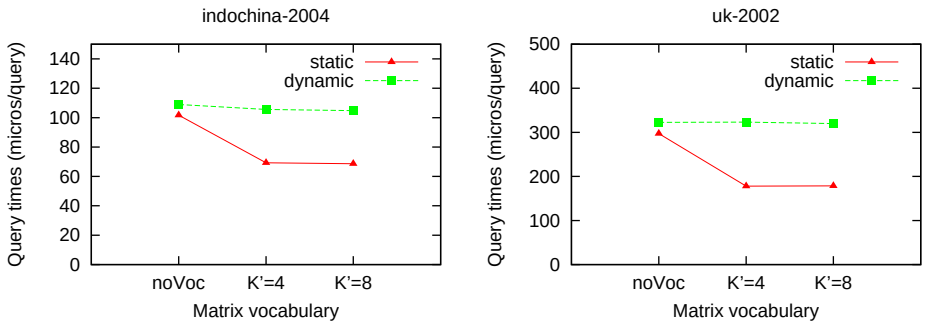


Figure 8.11: Time to retrieve the successors of a node in a static K^2-tree and a dK^2-tree in the Web graphs studied. Query times in μs/query.

Figure 8.11 shows the results obtained. The dK^2-tree is always slower than a static representation, and its query times are very similar independently of whether a matrix vocabulary is used. Comparing the dK^2-tree version that obtained the best space results ($K' = 4$) with the best static K^2-tree version ($K' = 8$), the dK^2-tree is 50-80% slower than the static data structure. This difference in query times is significant, but considering the dynamic nature of the dK^2-tree data structures a result that is below the double of the original times is still a good achievement.

8.5.4 Update times

The cost of update operations depends on several factors. As we have shown, the choice of block size B and sampling parameter s causes significant differences in both query and update times. Apart from this, the characteristics of the dataset also have a great influence in its K^2-tree and dK^2-tree representation, since the clusterization of 1s and thee small-scale regularities in the adjacency matrix lead to a better compression of the data. In the dK^2-tree, the clusterization of 1s and the sparsity of the adjacency matrix also affect update times: when new 1s must be inserted and they are far apart from any other existing 1, the insertion operation must insert K^2 bits in many levels of the conceptual K^2-tree, which increases the

cost of the operation. Therefore, insertion costs are expected to be higher on average when datasets are very sparse.

To show the effect of the distance between 1s on update costs, we build a set of very sparse synthetic datasets where we precisely control the distance between 1s in the adjacency matrix. We create matrices where 1s are inserted every 2^d rows and 2^d columns, so that the K^2-tree representation has a unary path of length d to each edge. Table 8.2 shows a summary with the basic information of the datasets. We choose the separation for the different dataset sizes so that all the datasets have the same number of edges (4,194,304).

Dataset	#rows/columns	Separation between 1s (2^d)	# K^2-tree levels
synth_22	4,194,304	2,048 ($d = 11$)	22
synth_24	16,777,216	8,192 ($d = 13$)	24
synth_26	67,108,864	32,768 ($d = 15$)	26

Table 8.2: Synthetic sparse datasets used to measure insertion costs.

Over these datasets we measure the insertion cost depending on the number of levels ℓ that must be created in the conceptual K^2-tree (that is, the number of levels where K^2 bits have to be added) to insert the new 1. We compare the insertion costs for $\ell \in [0, 10]$. For each dataset and value of ℓ, we create a set of 200,000 cells of the matrix that require exactly ℓ new levels in the conceptual tree (that is, they are separated of the closest 1 by $2^{\ell-1} - 2^{\ell}$ rows/columns). To build the query sets we select random 1s of the original matrix and choose a new 1 adding 2^{ℓ} to the row and/or column of the original 1. This guarantees that the new 1 will be located in the same K^2-tree node of height $\ell + 1$, but it will require a new path of length ℓ from this node down. For each dataset we select the same 1s of the original matrix for all the different values of ℓ (we use a pseudo-random number generator with the same seed), so that insertions access similar positions. Additionally, to limit the effect of other parameters in this measurement, we use a simple setup with no partial expansions (a single block size).

We run the insertions for all different values of ℓ over the synthetic datasets and measure the average insertion times in μs/query. Additionally, and in order to estimate the actual cost of the update operation, we *query* the new cells over the unmodified synthetic datasets. These queries, that return no results, will be used to determine how much of the insertion cost is due to locating the node of the conceptual K^2-tree where we must start the insertion. Notice that since we query the original datasets but each insertion modifies it, the average query time for the cells is just an estimation of the actual cost during insertion.

Figure 8.12 (left) shows the evolution of insertion and query times in the different datasets. When ℓ is small (new 1s are inserted very close to existing 1s), insertion and query times are almost identical, since we only need to update a single bit and

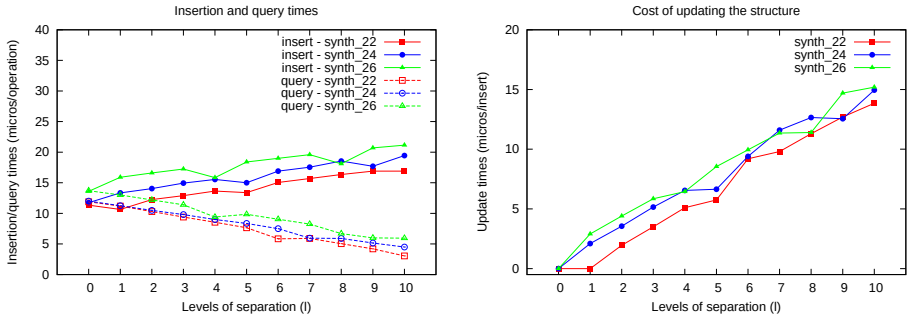


Figure 8.12: Insertion times (left) and estimation of update cost (right) in the dK^2-tree with varying ℓ.

some counters in the dK^2-tree. However, as ℓ increases the insertion cost becomes higher due to the additional cost of inserting data in the blocks of the dK^2-tree and update samples and counters in multiple levels. On the other hand, query times become lower because the first 0 in the conceptual tree (the node where insertion should start) is found in upper levels of the tree. Figure 8.12 (right) shows an estimation of the actual cost devoted to update the tree depending on ℓ. This result is computed subtracting the average insertion times and average query times shown in the left plot. Notice that for very small ℓ the estimation of the update costs becomes 0 (actually in some cases we obtain small negative values that are rounded to 0 due to the fact that we are measuring an estimation of the query cost).

Our overall results show that insertion times in the dK^2-tree can be very close to query times if the represented dataset has some properties that are also desirable for compression, particularly the clustering of 1s in the binary matrix. As expected, insertion times increase with the number of levels that must be updated in the conceptual tree. Varying the level of the tree where insertion begins we can see that insertion times increase slowly as the total cost of insertion becomes dominated by the updates required in the dK^2-tree blocks.

The evolution of insertion times shows that insertions at the upper levels of the tree may be several times more costly than insertions in the lower levels of the tree. However, insertions in the upper levels of the tree should be very infrequent in most of datasets where a dK^2-tree will be used, since compression in K^2-trees also degrades when matrices have no clusterization at all.

Notice that the insertion of new nodes, when it requires the insertion of a root node to increase the size of the conceptual matrix, is the most extreme case of insertion: we are adding a new root node to the conceptual tree and therefore creating a completely new path of K^2 bits at each level of the tree for a new edge. This operation can be several times more costly than the typical insertion, however it is so infrequent that the additional cost is acceptable. Notice also that, in general,

the K^2-tree and therefore the dK^2-tree are mainly designed to work in domains where values are somehow clustered: the compression achieved by the K^2-tree in clustered matrices is much larger thanks to the efficient compression of large regions of 0s and the compression of other regularities in the matrix. A similar argument can be followed in the case of update times in the dK^2-tree: in domains where data is clustered, most of the insertions in the dK^2-tree should be in clusters of 1s, so the average insertion costs should be small. In unclustered matrices, update times of the dK^2-tree may be higher, and compression may also degrade significantly.

8.6 Summary

In this chapter we have presented the dK^2-tree, our proposal for the compact representation of dynamic binary relations. Our proposal is a dynamic version of the K^2-tree, that is built by replacing the static data structures (static bitmaps and a matrix vocabulary) by dynamic versions with the same functionality but supporting updates. We described the techniques used to take advantage of the properties of the K^2-tree to provide an efficient access to the dynamic representation.

In this chapter we also provided an experimental evaluation of the dK^2-tree. First we analyzed the efficiency depending on the different parameters and compression options implemented, including the utilization of a matrix vocabulary. Then we compared the dK^2-tree with a static K^2-tree, showing that the dK^2-tree may require a significant (but reasonable) space overhead. Additionally, the dK^2-tree can answer successor/predecessor or range queries in times relatively close to those of static K^2-trees. Finally, we provided an analysis of the update times in the dK^2-tree, showing that insertion times are not much higher than query times in clustered datasets where insertions occur in positions close to existing 1s.

Chapter 9

Variants and applications of the dK^2-tree

The dK^2-tree, introduced in the previous chapter, provides a simple method to apply static representations based on the K^2-tree to a dynamic environment where data may suffer changes. In this chapter we introduce different specific applications of the dK^2-tree to different domains, in order to show the applicability and flexibility of the new data structure. Some of the proposals will be direct applications of the dK^2-tree data structure, while in other cases we introduce specific variants for a problem.

The rest of this chapter is structured as follows: in Section 9.1 we introduce a dynamic representation of RDF databases based on a collection of dK^2-trees, that enhances an existing static representation with update capabilities. In Section 9.2 we show how to adapt the new encodings proposed in Chapter 5 to the dK^2-tree, therefore providing a dynamic representation of binary images. In Section 9.3 we introduce a new proposal for the dynamic representation of temporal graphs. Finally, Section 9.4 shows a summary of the proposals in this chapter.

9.1 Representation of RDF databases

RDF datasets can be built from snapshots of data and therefore stored in static form. However, in many cases new information is continuously appearing and must be incorporated into the dataset to keep it updated. If we use a static representation of the dataset the complete dataset must be rebuilt when updates of the information appear, even though they may consist of a small fragment in comparison with the complete dataset.

A representation of RDF datasets based on K^2-trees, called K^2-triples, has been presented in Section 4.2. This representation uses a collection of K^2-trees

to represent the triples corresponding to each predicate in the RDF dataset. This representation was proved to be very competitive in space and query times with state-of-the-art alternatives, but was limited to a static context due to the static nature of K^2-trees.

In this section we propose a simple dynamic representation of RDF datasets based on dK^2-trees, that simply replaces static K^2-tree representations with a dK^2-tree per predicate. The goal of this section is to demonstrate that a representation based on the dK^2-tree can obtain query times close to the static K^2-triples approach. Additionally, the dK^2-tree representation provides the basis to perform update operations on the RDF dataset.

9.1.1 Our proposal

Our proposal simply replaces the static K^2-tree representation in K^2-triples with a dK^2-tree per predicate. We consider a partition of the RDF dataset by predicate, and build a dK^2-tree for each predicate in the dataset. We use the same strategy of K^2-triples to store the information of triples: for each predicate we consider a matrix storing all relations between subjects and objects with that predicate. All matrices will contain the same rows/columns, where subject-objects (elements that appear as both subjects and objects in different triples) will be located together in the first rows/columns of the matrix and the remaining rows (columns) of the matrices will contain the remaining subjects (objects). Notice that this organization of the elements is necessary for the representation to efficiently answer join queries without accessing the vocabulary to map between subjects and objects.

Our proposal based on dK^2-trees aims to solve the representation of the structural part of an RDF dataset. We assume in our proposal that additional data structures must be used to store vocabularies of subjects, objects and predicates and map between them and rows/columns of the matrices represented by dK^2-trees. The creation of a dynamic and efficient dictionary representation to manage large collections of URIs and literal values is a complex problem, since the vocabulary may constitute a large part of the total size of an RDF dataset [MPFC12].

9.1.1.1 Query support

As we have shown in Chapter 8, dK^2-trees support all the basic queries supported by static K^2-trees, since our representation supports the basic navigation operations over the conceptual tree with its dynamic bitmap implementations. Hence, our proposal can directly answer all triple pattern queries: (S, P, O) and $(S, ?P, O)$ queries are actually cell retrieval queries (involving one dK^2-tree or all the dK^2-trees in the collection), $(S, P, ?O)$, $(S, ?P, ?O)$, $(?S, P, O)$ and $(?S, ?P, O)$ are row/column queries and $(?S, P, ?O)$ is a full range retrieval query that asks for all the cells in a dK^2-tree.

Join operations can also be answered using a dK^2-tree exactly like in static K^2-trees. Let us consider the three join strategies used in K^2-triples:

- Independent evaluation separates any join operation in two triple pattern queries and a simple merge that intersects the results of both queries. The adaptation to dK^2-trees is immediate using the basic operations explained.

- Chain evaluation is similar, in the sense that a join operation is translated into a collection of triple pattern operations. For any join operation involving two (or more) triple patterns, we can decompose the join into its basic patterns and chain the execution in any order using the basic operations available in dK^2-trees.

- Interactive evaluation is a more complex operation, in which two K^2-trees are traversed simultaneously. The basic elements of this strategy include a synchronized traversal of the conceptual trees, similar to the set operations (specifically the intersection operation) described in Chapter 5 for static K^2-trees and that can be directly applied to dK^2-trees. Regardless of the type or complexity of the join operation, the essential steps of interactive evaluation are based on the access to one or more nodes in the conceptual trees of different K^2-trees to check their value and determine whether or not to continue the navigation in the current branches of the remaining conceptual trees. All these operations are directly supported by dK^2-trees using just the basic traversal operations over the conceptual tree.

9.1.1.2 Update operations using dK^2-trees

Our proposal is able to answer all the basic queries supported by K^2-triples simply replacing static K^2-trees by dK^2-trees and assuming the same organization of the dictionary required to manage the values in the triples. In this section we will show that update operations in RDF datasets can be easily supported by our proposal. This is presented as a proof of concept of the applicability of dK^2-trees to this domain, even though our proposal focuses only on the representation of the triples.

The most usual update operation in an RDF dataset is probably the insertion of new triples, either one by one or, more frequently, in small collections corresponding to new information retrieved or indexed regularly. The insertion of new triples in an existing RDF database involves several operations in our representation based on dictionary encoding:

- First, the values of the subject, predicate and object of the new triple must be searched in the dictionary, and added if necessary. If all the elements existed in the dictionary the new triple is stored as a new entry in the dK^2-tree corresponding to its predicate.

- If the triple corresponds to a new predicate, a new empty dK^2-tree can be created to store the new subject-object pair.

- If the subject and/or object are new, we must add a new row/column to all the dK^2-trees. This operation is usually trivial in dK^2-trees, since the virtual matrices represented by dK^2-trees are always of size power of K and many rows/columns may be unused. In this case to add a new subject/object we simply need to keep count of the new row/column that will be used. When all rows/columns are full, we must increase the size of the matrices, an operation (with a cost comparable to the insertion of a new 1 in the matrix) that must be repeated in all the dK^2-trees. For a matrix of size $n \times n$ this process creates $(K-1)n$ new available rows/columns, so this costly operation should be rarely executed. Larger values of K can be used in the upper levels of the conceptual tree to further reduce the number of times we need to expand the matrices.

The removal of triples to correct or delete wrong information is also a typical update operation in RDF datasets. The possible changes when triples are removed are similar to the insertion case: when new triples are removed we may need to simply remove a 1 from a dK^2-tree or remove a row/column (marking it as unused) if the subject/objects has no associated triples.

Detecting subject-object changes. A particular case that must be considered is the insertion of triples that causes a subject(object) to become a subject-object (or the deletion of triples that transforms a subject-object in just subject or object).

We assume in our representation that subject-object elements are stored together in the top-left region of the matrices, so that join operations do not need to perform additional computations. A simple solution to avoid the problem in the dynamic case would be to use a different setup where rows/columns of the matrices would contain all the elements (subjects and objects) instead of storing only subjects in the rows and objects in the columns. This should have small effect in the overall compression, since K^2-trees and dK^2-trees depend mostly on the number and distribution of the 1s in the matrix than on the matrix size. The effect on query times should also be limited since we would at most double the size of the matrices, adding a single level to their conceptual trees. The selection of this or other organization for the rows/columns of the matrices could be adjusted depending on which organization allows us to efficiently store and query a dictionary of subjects/objects.

In order to follow the original setup with subject-object elements in the first rows/columns of the matrix, when a subject(object) becomes a subject-object we need to move it to the *beginning* of the matrix. This requires finding all the 1s in the corresponding row(column), allocating a new row and column at the beginning of the matrix and inserting the 1s in the same locations in the new row (column).

Notice that, even though we only described the ability of dK^2-trees to add new rows at the end of the matrix, the process can be trivially extended to add rows at the beginning: if we have an $n \times n$ matrix, subject-object elements start at row/column $n/2 - 1$ and are created towards the top-left corner of the matrix, while subjects and objects start at row/column $n/2$ and are created in the usual order. When new rows/columns are necessary at the beginning or the end of the matrix we simple expand it adding a new level to the tree, where the current matrix could be in the top-left corner of the new one (we add new available rows at the end) or in the bottom-right corner (we add new available rows/columns at the beginning for subject-object elements). Simply keeping track of the point in the matrix that divides between types of elements we can still answer all queries. This setup, even though it requires more complex operations (copying several elements), is still feasible in this domain since in many cases elements that are subjects and objects will be detected when adding new sets of triples before adding each element as simple subject or object.

9.1.2 Experimental evaluation

As a proof of concept of the efficiency of the dK^2-tree for the representation of RDF databases, we compare our proposal based on dK^2-trees with the static data structures used in K^2-triples and K^2-triples$^+$ described in Section 4.2. Our proposal is a direct adaptation of the K^2-triples representation where static K^2-trees are replaced by dK^2-trees and all queries are also directly adapted to use our dynamic representation. The goal of these experiments is to demonstrate the efficiency of dK^2-trees in this context, and their ability to act as the basis for a dynamic compact representation of RDF databases. We will compare the space results and query times of dynamic and static representations to show that dK^2-trees can store RDF datasets with a reduced overhead over the space and time requirements of static representations.

9.1.2.1 Experimental setup

We experimentally compare our dynamic representation with the equivalent representation using static K^2-trees. We use a collection of RDF datasets of very different sizes and number of triples, and also include datasets with few and many different predicates[1]. Table 9.1 shows a summary with some information about the datasets used. The dataset *jamendo*[2] stores information about Creative Commons licensed music; *dblp*[3] stores information about computer science publications; *geonames*[4]

[1]The datasets and general experimental setup used are based on the experimental evaluation in [ÁGBF$^+$14], where K^2-triples and K^2-triples$^+$ are tested. We use the same datasets and query sets in our tests.

[2]`http://dbtune.org/jamendo`

[3]`http://dblp.l3s.de/dblp++.php`

[4]`http://download.geonames.org/all-geonames-rdf.zip`

stores geographic information; finally, *dbpedia*[5] is a large dataset that extracts structured information from Wikipedia. As shown in Table 9.1, the number of predicates is small in all datasets except *dbpedia*, that is also the largest dataset and will be the best example to measure the scalability of queries with variable predicate.

Collection	#triples	#predicates	#subjects	#objects
jamendo	1,049,639	28	335,926	440,604
dblp	46,597,620	27	2,840,639	19,639,731
geonames	112,235,492	26	8,147,136	41,111,569
dbpedia	232,542,405	39,672	18,425,128	65,200,769

Table 9.1: RDF datasets used in our experiments.

We will build our dynamic representation following the same procedure used for static K^2-trees, where elements that are both subject and object in any triple pattern are grouped in the first rows/columns of the represented matrices. We use a similar setup to build static and dynamic K^2-trees: we use a hybrid K^2-tree representation, with $K = 4$ in the first 5 levels of decomposition and $K = 2$ in the remaining levels. The dK^2-tree uses a sampling factor $s = 128$ in the leaf blocks (sampling every 128 bytes), while static K^2-trees use a single-level rank implementation that samples every 20 integers (80 bytes). In dK^2-trees we use a block size $B = 512$ and $e + 1 = 4$ different block sizes.

We test query times in all the approaches in all possible triple patterns (except $(?S, ?P, ?O)$, that simply retrieves the complete dataset) and some join queries involving just 2 triple patterns. We use the experimental testbed in [ÁGBF$^+$14] to directly compare our representation with K^2-triples. To test triple patterns, we use a query set including 500 random triple patterns for each dataset and pattern. To test join operations we use query sets including 50 different queries, selected randomly from a larger group of 500 random queries and divided in two groups: 25 have a number of results above the average and 25 have a number of results below the average.

The experimental evaluation of join operations is expected to yield similar comparison results to triple patterns, considering the fact that the implementation of the different strategies to answer join queries is identical in dK^2-trees and static K^2-trees. As a proof of concept of the applicability of dK^2-trees in more complex queries, we will experimentally evaluate dK^2-trees, and K^2-triples to answer join operations $(?V, P_1, O_1) \bowtie (?V, P_2, O_2)$ (*join 1*: only the join variable is undetermined) and $(?V, P_1, O_1) \bowtie (?V, ?P_2, O_2)$ (*join 2*: one of the predicates is variable). Recall from Section 4.2 that each join type can lead to 3 different join

[5] `http://wiki.dbpedia.org/Downloads351`

operations depending on whether the join variable is subject or object in each of the triple patterns: for example, join 1 can be of the form $(?V, P_1, O_1) \bowtie (?V, P_2, O_2)$ (S-S), $(S_1, P_1, ?V) \bowtie (?V, P_2, O_2)$ (S-O) and $(S_1, P_1, ?V) \bowtie (S_2, P_2, ?V)$ (O-O).

9.1.2.2 Space results

We compare the space requirements of our dynamic representation, based on dK^2-trees, with K^2-triples and its improvement K^2-triples$^+$ in all the studied datasets. We build the K^2-tree representations that obtain the best compression results in all cases. The static K^2-trees used in K^2-triples and K^2-triples$^+$ use a matrix vocabulary with $K' = 8$ to obtain the best compression results. In our dK^2-trees we use the version with no matrix vocabulary. Table 9.2 shows the total space requirements on the different collections studied.

Collection	K^2-triples	K^2-triples$^+$	dK^2-trees
jamendo	0.74	1.28	1.61
dblp	82.48	99.24	125.34
geonames	152.20	188.63	242.60
dbpedia	931.44	1,178.38	1,151.90

Table 9.2: Space results for all RDF collections (sizes in MB).

Our dynamic representation is significantly larger than the equivalent static version, K^2-triples, in all the datasets. In *jamendo*, a very small dataset, the dynamic representation requires more than twice the space of K^2-triples. However, the overhead required by the dynamic version is smaller in larger datasets and particularly in *dbpedia*. This result is significant considering the fact that the static representations use a matrix vocabulary with a relatively large value of K' that helps them improve compression. Even so, the dK^2-tree with no matrix vocabulary is able to store the dataset with an overhead below 50% extra in the *dblp* and *geonames* datasets. Even though the overhead is significant, the results must be put into context considering that K^2-triples was proved to be several times smaller than other RDF stores like MonetDB and RDF-3X in these datasets (at least 4 times smaller than MonetDB, the second-best approach in space, in all the datasets except *dbpedia* [ÁGBF$^+$14]).

In the *dbpedia* dataset our proposal has a space overhead around 20% over K^2-triples, and becomes smaller than the K^2-triples$^+$ static representation. This result is mostly due to the characteristics of the *dbpedia* dataset. It contains many predicates with few triples, and a small number of predicates condense 90% of the total triples. The static representations based on K^2-triples store a static K^2-tree representation for each different predicate, each one containing its own matrix vocabulary. These many K^2-tree representations of matrices with very few ones are difficult to compress also in the static data structures, and the utilization of a matrix

vocabulary in these matrices does not improve compression. However, most of the cost of the representation is in the matrices with many triples, so the utilization of a matrix vocabulary still obtains the best results overall.

9.1.2.3 Query times

In this section we aim to determine the relative query efficiency of dK^2-trees and static K^2-trees in the studied datasets. We first measure the efficiency of our dynamic proposal in comparison with K^2-triples to answer simple queries (triple patterns) in all the studied datasets. The results for all the datasets are shown in different tables: Table 9.3 shows the results for *jamendo*, Table 9.4 the results for *dblp*, Table 9.5 for *geonames* and Table 9.6 for *dbpedia*. For each dataset we show the query times of K^2-triples, K^2-triples$^+$ (only in queries with variable predicate) and our equivalent dynamic representation of K^2-triples. The last row of each table shows the ratio between our dynamic representation and K^2-triples, as an estimation of the relative efficiency of dK^2-trees.

	S,P,O	$S,P,?O$	$?S,P,O$	$?S,P,?O$	$S,?P,O$	$S,?P,?O$	$?S,?P,O$
K^2-triples	1.0	4.6	102.8	6954.1	4.9	39.4	29.3
K^2-triples$^+$					1.1	23.6	10.0
Dynamic	1.9	4.8	235.6	12788.5	6.0	34.6	28.4
Ratio	1.88	1.06	2.29	1.84	1.22	0.88	0.97

Table 9.3: Query times for triple patterns in *jamendo*. Times in μs/query.

Solution	S,P,O	$S,P,?O$	$?S,P,O$	$?S,P,?O$	$S,?P,O$	$S,?P,?O$	$?S,?P,O$
K^2-triples	1.2	79.8	1016.4	771061.6	3.6	1294.1	187.5
K^2-triples$^+$					1.7	1102.1	140.8
Dynamic	6.5	92.9	2776.3	1450058.3	14.0	1421.8	247.9
Ratio	5.54	1.16	2.73	1.88	3.87	1.10	1.32

Table 9.4: Query times for triple patterns in *dblp*. Times in μs/query.

Solution	S,P,O	$S,P,?O$	$?S,P,O$	$?S,P,?O$	$S,?P,O$	$S,?P,?O$	$?S,?P,O$
K^2-triples	1.2	59.4	4588.0	1603677.4	2.9	1192.9	273.7
K^2-triples$^+$					1.4	915.9	139.0
Dynamic	9.4	79.6	9544.2	2958262.4	17.9	1514.7	423.5
Ratio	7.71	1.34	2.08	1.84	6.11	1.27	1.55

Table 9.5: Query times for triple patterns in *geonames*. Times in μs/query.

Solution	S,P,O	$S,P,?O$	$?S,P,O$	$?S,P,?O$	$S,?P,O$	$S,?P,?O$	$?S,?P,O$
K^2-triples	1.1	441.4	10.5	1859.5	7960.3	54497.4	29447.7
K^2-triples$^+$					1.4	2216.7	518.3
Dynamic	6.6	561.9	19.1	3870.7	23045.4	83340.2	57051.8
Ratio	6.21	1.27	1.82	2.08	2.90	1.53	1.94

Table 9.6: Query times for triple patterns in *dbpedia*. Times in μs/query.

In most of the datasets and queries, query times of dK^2-trees are between 1.2 and 2 times higher than in K^2-triples. The results in Table 9.3 for the dataset *jamendo* show some anomalies, with the dK^2-tree performing faster than a static representation. However, due to the reduced size of the dataset we shall disregard these results and focus on the larger datasets. The results in Table 9.4, Table 9.5 and Table 9.6 show very different query times but the ratios shown in each table are very similar in all three datasets.

Our results show that dK^2-trees are several times slower than static K^2-trees in triple patterns that are implemented with single-cell retrieval queries (i.e. patterns (S,P,O) and $(?S,P,?O)$). Particularly dK^2-trees are 5.5-7.7 times slower than static K^2-trees to answer (S,P,O) queries. In this queries, the cost of accessing T_{tree} and L_{tree} is very high since a single position is accessed per level of the tree.

In all the remaining patterns, that are translated into row/column or full-range queries in one or many K^2-trees, dK^2-trees are much more competitive with static K^2-trees, obtaining query times less than 2 times slower than K^2-triples in most cases. These differences are mostly due to the indexed representation used in dK^2-trees, that avoids complete traversals of T_{tree} or L_{tree} when many close positions are accessed in each query. In all these patterns, multiple positions are accessed at each level of the dK^2-tree, and in many cases positions accessed consecutively will fall in the same leaf block of T_{tree} or L_{tree}. The *findLeaf** operation, that locates the leaf for a current position from the previously accessed one, can save most of the cost of traversing the internal nodes of T_{tree} and L_{tree} when the next position is in the same leaf block or a sibling of the current leaf block.

After analyzing the relative efficiency of dK^2-trees and static K^2-trees, we focus on the effect of the additional indexes used in K^2-triples$^+$. In the datasets with few predicates, that are the most usual RDF datasets, K^2-triples$^+$ is around 1.5-3 times faster than K^2-triples, hence up to 10 times faster than dK^2-trees in $(S,?P,O)$ queries and up to 3 times faster in the remaining patterns with variable predicate. In the *dbpedia* dataset, the efficiency of K^2-triples$^+$ is much more significant, reducing query times by several orders of magnitude. This makes dK^2-trees much slower than K^2-triples$^+$ in this dataset to answer patterns with variable predicate.

Join operations

Next we test the efficiency of dK^2-trees in comparison with K^2-triples to answer join queries on RDF datasets. Considering that all join strategies used in K^2-

triples can be directly implemented in dK^2-trees, we expect a relative comparison between static and dynamic K^2-trees in the same ranges obtained for simple queries. To test the efficiency of dK^2-trees we compare them with static K^2-trees in the join operations that are more selective, or require more selective operations in the matrices.

We start our experiments with join 1 $((?V, P_1, O_1) \bowtie (?V, P_2, O_2))$. We test the 3 different join strategies applied to this join: *i)* independent evaluation requires two row/column queries in different K^2-trees and an intersection of the results; *ii)* in chain evaluation, we run a row/column query in the K^2-tree corresponding to P_1 and for each result v obtained we run a single cell retrieval in the K^2-tree for P_2; *iii)* the interactive evaluation runs two synchronized row/column queries in both K^2-trees. For each evaluation strategy, we test all the join categories S-O, S-S and O-O and query sets with few results (small) or more results (big).

The results are shown in Figure 9.1. The results for each query set are normalized by the times of static K^2-trees, so each result for dK^2-trees measures the overhead required by the dynamic representation (that is, query times in dK^2-trees divided by query times in static K^2-trees). We also show for each query set the actual query times obtained in K^2-triples, in ms/query. Given the significant differences in query times between datasets, strategies and even query sets, we normalize all the results so that the query times of static K^2-trees are always at the same level.

Results show that dK^2-trees are very competitive with K^2-triples in most of the datasets and strategies: independent evaluation (left plots of Figure 9.1) yields the worst results for dK^2-trees, that are 3-4 times slower than K^2-triples in most of the query sets, except on the larger dataset *dbpedia* where our solution is on average less than 2 times slower than static K^2-trees. In chain evaluation dK^2-trees are much faster in comparison with static K^2-trees, and our solution is less than 2 times slower than a static representation in most of the datasets and query sets. Finally, if we use the interactive evaluation strategy dK^2-trees are able to obtain query times very close to static K^2-trees in most of the cases.

Averaging the results of all the datasets and query sets for each evaluation strategy, dK^2-trees are 3.5 times slower than static K^2-trees in the independent evaluation strategy, 2.6 times slower using the chain evaluation strategy and have a penalty of 27% extra query time using the interactive evaluation strategy. Even though the results vary significantly between datasets and query sets, the overall results show that dK^2-trees are able to support the query operation with a reasonable overhead in space and query times.

Join queries with variable predicate

Next we analyze the effect of the additional indexes in join operations with variable predicates. We compare dK^2-trees with static K^2-triples and K^2-triples$^+$ (static-DACs) to answer the join 2 $((?V, P_1, O_1) \bowtie (?V, ?P_2, O_2))$, that involves a variable predicate. Again, we normalize the results to the query times obtained by K^2-triples and measure all the other query times as a ratio in comparison with

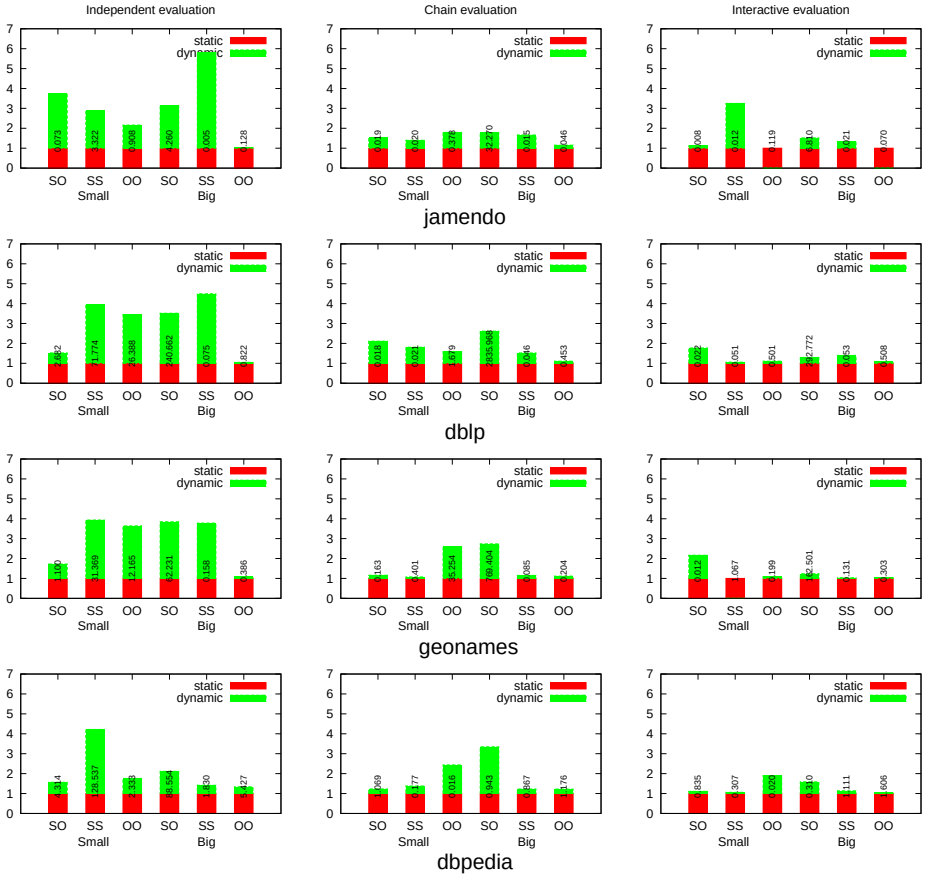


Figure 9.1: Comparison of static and dynamic query times in join 1 in all the studied datasets and query types. Results are normalized by static query times. Query times of the static representation shown in ms/query.

them.

The results are shown in Figure 9.2. Like in the previous tests we obtain diverse results in the comparison depending on the dataset, the evaluation strategy and the query set. In spite of the varying results, dK^2-trees show again a significant overhead compared with K^2-triples, but this time overhead is in general limited by a small factor. Overall, dK^2-trees are between 1.5 and 3 times slower than K^2-triples depending on the query strategy.

If we focus on the comparison between dK^2-trees and the best static representa-

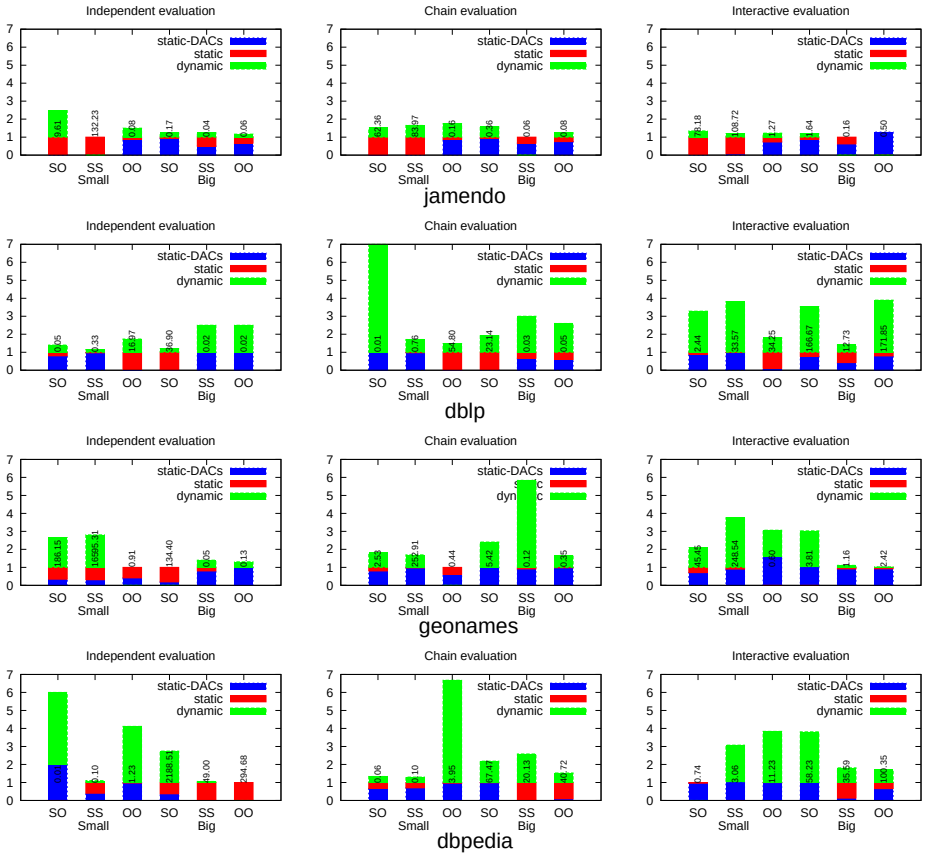


Figure 9.2: Comparison of static and dynamic query times in join 2 in all the studied datasets and query types. Results are normalized by query times of K^2-triples. Query times of the static representation shown in ms/query.

tion, that is now K^2-triples$^+$, the overhead required by the dynamic representation becomes much larger. Notice that in Figure 9.2 there are some query sets for which the query times of K^2-triples$^+$ (*static-DACs*) are so much lower than K^2-triples that the result for K^2-triples$^+$ is not even visible in the plot. This is consistent with the results obtained in triple patterns, where the use of S-P and O-P indexes in K^2-triples$^+$ provided a major improvement in triple patterns with variable predicate. In all these cases our dynamic proposal becomes orders of magnitude slower than K^2-triples$^+$. However, in many cases the improvement obtained by K^2-triples$^+$ is not

so significant and dK^2-trees are still reasonably close in query times. Nevertheless, even in datasets with few predicates the addition of the specific static indexes used in K^2-triples$^+$ leads to much better query times in some cases.

9.2 Image representation with the dK^2-tree

In this section we combine the ideas of Chapter 5 with the dK^2-tree of Chapter 8 to obtain a variant of the dK^2-tree able to compress regions of 1s and 0s in a binary matrix. First, we show that all the techniques used to obtain better compression of ones in the K^2-tree can be directly applied in the dK^2-tree, obtaining different dK^2-tree variants with compression of ones. After this, we give some details on the additional considerations needed to implement the update operations in a dK^2-tree with compression of ones.

Finally, we will show, as a proof of concept of the applicability of our proposal, an experimental comparison between a dK^2-tree with compression of ones, a static K^2-tree with compression of ones and a state-of-the art alternative, Gargantini's linear quadtree (LQT), that supports spatial access to a binary raster as well as update operations that change 0s into 1s and vice versa.

9.2.1 Dynamic versions of K^2-tree variants with compression of ones

The K^2-tree variants with compression of ones explained in Chapter 5 are based on changes in the conceptual tree and changes in the underlying bitmaps that represent the conceptual tree. In all the static variants the basic navigation of the conceptual tree, that is used to answer all the supported queries, is based on the same simple ideas: *1)* a method to determine whether a node is *internal* or a leaf node, *2)* know the *color* of a leaf node (i.e. know whether a leaf node represents a region of 1s or a region of 0s) and *3)* a method to find the K^2 *children* of a node. In all the proposed static variants, all computations are based on *access* and *rank* operations over the bitmaps T or L, that are trivially supported by dK^2-trees. Hence, the changes in the conceptual tree required by all the static variants can be reproduced using dK^2-trees. In this section we sketch the steps required to build a dK^2-tree representation for the different static variants.

- **Dynamic K^2-tree1$^{\text{2bits}-\text{naive}}$, K^2-tree1$^{\text{2bits}}$ and K^2-tree1$^{\text{df}}$.** In all these approaches we use an additional bitmap T' in order to use 2-bit codes for some nodes in the conceptual tree. The differences between the variants arise from the encoding used to assign bits to T and T', since the bitmap L is identical to the original K^2-tree representation:
  - In the K^2-tree1$^{\text{2bits}-\text{naive}}$, we can know whether a node at position p is internal or a leaf, and its color, simply accessing $T[p]$ and $T'[p]$. To

locate the children of an internal node we need to compute $rank_1(T, p)$. Therefore, we need support for *access* in T' and *access* and *rank* in T. We can directly adapt this representation to our dynamic tree representations using T_{tree}, L_{tree} and an additional tree structure T'_{tree} to store the bits in T'. Since we only need to *access* T'_{tree}, we can use the same data structure used for L_{tree} to represent T'_{tree}.

- In the K^2-tree1$^{\text{2bits}}$ the operations to know if a node is internal and to find its children are identical, but we also need to compute *rank* operations in T to know the color of a leaf node. Since *rank* support was already required by the *children* operations, the basic access to T and T' does not change and we can use the same tree structures T_{tree}, T'_{tree} and L_{tree} used in the K^2-tree1$^{\text{2bits}-\text{naive}}$.
- In the K^2-tree1$^{\text{df}}$, slightly more complex operations are required to find the *children* or *color* of a node. The main difference in practice is that we also need *rank* support in T'. Hence, our representation for T'_{tree} in a dynamic K^2-tree1$^{\text{df}}$ should be a data structure similar to T_{tree} instead of L_{tree}.

- **Dynamic K^2-tree1$^{1-\text{5bits}}$.** In the K^2-tree1$^{1-\text{5bits}}$, the data structure of the K^2-tree was not altered. Instead, a special sequence of K^2 0 siblings was used to identify black nodes in the conceptual tree. Since no changes are made to the tree structure, a dynamic K^2-tree1$^{1-\text{5bits}}$ can be built directly replacing the bitmaps T and L with dynamic representations T_{tree} and L_{tree}.

9.2.2 Algorithms

All the typical query algorithms explained for the K^2-tree variants with compression of ones are based on the basic navigation operations that are already supported by dK^2-trees. Therefore, all queries supported by static K^2-trees can by straightforwardly adapted to a dynamic scenario using dK^2-trees.

We focus next on the implementation of update operations in dK^2-tree variants with compression of ones, considering the special cases that must be managed in these variants. Insertion of new rows/columns in the matrix does not change, and similar strategies can also be used to handle the deletion of rows/columns. We focus on the implementation of the basic operations required in a dynamic representation, the insertion of new elements (changing 0s into 1s) and the deletion of elements (changing 1s into 0s). These operations are implemented using algorithms very similar to the algorithms explained in the basic dK^2-tree. We conceptually describe the differences that must be taken into account when compressing regions of ones.

9.2.2.1 Inserting new 1s

In original dK^2-trees the insertion of a new 1 in the matrix is divided in two steps: *1)* we search the node in the conceptual tree where the new 1 should fall. At this point, the value of the node must be changed from 0 to 1. *2)* If the node was not at the last level of decomposition, we must create a new branch in the conceptual tree adding K^2 bits in the remaining levels, all of them set to 0 except the branch corresponding to the new 1.

When we use a dK^2-tree with compression of 1s, the algorithm for insertion of new 1s is similar to the basic case: we first locate the position where the new 1 should fall; at some point we will find a white node (a region of 0s), and from this point on we should start creating new branches in the conceptual tree until we reach the leaves. Again we face two possibilities:

- The simplest case occurs when the white node is in the last level of the conceptual tree: in this case, we simply change its value from 0 to 1.
- If the node was in upper levels of the conceptual tree, the white node must become gray and we continue the insertion procedure exactly like in the basic case: in each new level we add new K^2 nodes, $K^2 - 1$ of them will be white (regions of 0s) and one of them, corresponding to the region where the new 1 belongs, will be gray. Eventually we will reach the last level of decomposition, and we will create K^2 bits, one of which will correspond to the inserted position and will be set to 1, while the others will be 0s (white).

Once the basic insertion procedure has finished, the insertion algorithm must also check if a new region of 1s has been created in the matrix. Trivially, only insertions that find the white node in the last level of the conceptual tree will generate new black regions (otherwise we are creating branches with $K^2 - 1$ white nodes and a gray/black node), so only this case must be checked. After the insertion of the 1 in the last level, we check the K^2-1 siblings of the modified node. If not all of them are black nodes, the current region of the matrix cannot be full of 1s and the algorithm ends. Otherwise, we know the current region is full of 1s, so we must delete all K^2 black nodes and replace their parent by a black node. The process is then repeated in the upper level of the conceptual tree, checking the $K^2 - 1$ siblings of the newly-created black node to check if they are also black nodes. Eventually we will find a mixed region and stop the bottom-up traversal.

9.2.2.2 Deleting 1s

To delete a 1 in the matrix (i.e. change a 1 into a 0), the algorithm in a dK^2-tree with compression of ones is also very similar to the original one. First, in a top-down traversal of the conceptual tree, we locate the (black) node that contains the 1 we want to delete. If the black node is in the last level of the tree, the deletion is identical to the basic algorithm: we replace the black node with a white node and

check its $K^2 - 1$ siblings to determine if all of them are white. If they are, we need to delete them and replace their parent by a white node. The process is repeated in the upper levels until we find that at least one sibling of the modified node is not white.

When we are deleting a 1 from a black node that is not in the last level of decomposition a slightly different algorithm is used. In this case, we are "destroying" a black region by deleting one of its 1s. This eventually transforms the black node corresponding to the region into a gray node with $K^2 - 1$ black children and a single gray child (we are deleting a single element in the complete region). This case is handled exactly like the insertion of new 1s in a white region: first we transform the current node into a gray node, and add in the next level K^2 children: one of them, the one that contains the new 0, will be gray, and the remaining ones will be black. Eventually the last level of the conceptual tree is reached, and K^2 nodes will be created: $K^2 - 1$ black nodes corresponding to the remaining 1s and a single white node corresponding to the element that has been set to 0.

9.2.2.3 Considerations on the implementation

Update operations over the binary matrix in the different variants studied require changing the type of nodes of the conceptual tree, that can be implemented in all the variants simply flipping, adding or removing bits to the different bitmaps used. In the K^2-tree1$^{\text{2bits-naive}}$, K^2-tree1$^{\text{2bits}}$ and K^2-tree1$^{\text{df}}$ variants, that use 3 bitmaps (T_{tree}, T'_{tree} and L_{tree}), changes in nodes will require updating both T_{tree} and T'_{tree}.

In the K^2-tree1$^{\text{2bits-naive}}$ representation most of the changes simply require flipping bits in T_{tree} or L_{tree} except when large white or black regions are destroyed when changing 0s into 1s or 1s into 0s.

Additionally, some optimizations can be used to optimize update operations in a K^2-tree1$^{\text{1-5bits}}$ variant. First, when inserting new 1s, the special case where new black nodes are created can be handled in a single step, since a gray node with K^2 children is transformed into a black node simply setting all its children to 0. A similar case occurs when destroying black nodes to create new gray nodes.

9.2.3 Application: representation of binary images

As a proof of concept of the applicability of our proposals, we compare the efficiency of dK^2-trees with compression of ones with the static variants. We experimentally compare their space and query times to represent sparse binary matrices and binary matrices with large regions of both 0s and 1s. The experimental evaluation in this section is based in the same datasets and setup used in Chapter 5, and compares dK^2-trees and K^2-trees in the compression of Web graphs and binary raster data. The goal of this experiments is to prove the suitability of our encodings for the compact representation of clustered binary images also in dynamic domains.

To demonstrate the efficiency of our variants, we also compare our dynamic and static proposals with an existing quadtree representation. Following the steps in Chapter 5, we implement a completely in-memory version of the LQT that can be compared with our proposals. As a dynamic representation we build a dK^2-tree using the K^2-tree$1^{1-5\text{bits}}$ encoding, that uses just the two original tree structures T_{tree} and L_{tree} and only requires changes in the algorithms. We compare the dynamic representation with a static K^2-tree$1^{2\text{bits}}$, that obtained the best compression results and good query times in our previous experiments. Our dynamic LQT implementation (LQT-BTree) stores the list of quadcodes in a B-tree in main memory. We compare all the approaches for the compression of the same datasets used in our previous experimental evaluation of static variants: binary raster images mdt-600, mdt-700 and mdt-A and the small Web graphs *cnr* and *eu*. In all cases we focus on the space utilization of the representations and the query times obtained.

9.2.3.1 Space results

Dataset	rows × cols	#ones	K^2-tree		LQT-BTree
			Static	Dynamic	
mdt-600	3961×5881	11,647,287	0.02	0.04	0.31
mdt-700	3841×5841	13,732,734	0.02	0.04	0.23
mdt-A	11081×7821	50,416,771	0.01	0.02	0.23
cnr	325,557×325,557	3,216,152	3.14	4.95	41.46
eu	862,664×862,664	19,235,140	3.81	5.86	50.07

Table 9.7: Space comparison of a dK^2-tree with a K^2-tree$1^{2\text{bits}}$ and LQTs (compression in bits per one).

Table 9.7 shows the compression results of all the data structures. For completeness we also include basic information about the size and number of 1s in each dataset. In all the studied datasets the dK^2-tree representation is significantly larger than the static version, but always within a factor of 2. Additionally, we can see in the comparison with linear quadtrees that the dK^2-tree is still at least 5 times smaller than a linear quadtree representation and supports updates in compressed form.

9.2.3.2 Time results

We also test the query times of the dynamic representation of binary images in comparison with the static version and the B-tree linear quadtree implementation. We use the same query sets and experimental setup used in Chapter 5 to test the

static version, measuring the average query times to retrieve the value of random cells. The results are shown in Table 9.8.

Dataset	K^2-tree		LQT-BTree
	Static	Dynamic	
mdt-600	0.25	0.56	0.89
mdt-700	0.28	0.61	0.92
mdt-A	0.26	0.71	1.23
cnr	0.77	2.55	2.28
eu	1.10	3.80	2.94

Table 9.8: Time to retrieve the value of a cell of the binary matrix, in μs/query.

The dK^2-tree is in our results slower than a static K^2-tree by a factor of 2.5 at most in the raster datasets where compression of 1s can be applied. This is a significant result considering the usual overhead in compact dynamic data structures, and is consistent with our previous evaluation of the dK^2-tree. We must note that the query overhead is larger in Web graphs, reaching a factor of 3-4 times slower. In practice, we believe that even an overhead of 3-4 times the static time in cell retrieval queries is acceptable, considering that we have shown that dK^2-trees become more competitive with static K^2-trees in row/column or range queries where multiple elements are accessed at each level of the conceptual tree.

The query times of the dK^2-tree are still competitive with the dynamic in-memory LQT, especially in raster datasets where we can answer the query in upper levels of the conceptual tree when we find a region of ones. Our proposal is faster than LQT-BTree in all the raster datasets studied. Nevertheless, even considering that our advantage in speed is relatively reduced, we believe this result is particularly relevant taking into account the relative simplicity of the LQT in comparison with our dK^2-tree and the fact that dK^2-trees are roughly 5 times smaller than LQTs in all datasets.

9.3 Representation of temporal graphs

In this section we introduce a new approach to store temporal graphs using a dK^2-tree that provides a fully-dynamic compact representation of the temporal graph. Unlike the static representations introduced in Chapter 6, this representation is able to store the complete state of the graph while supporting changes in the current state of the graph as well as changes in past states of the graph.

9.3.1 ttK^2-tree: temporally-tagged dK^2-tree

The new data structure, called ttK^2-tree, models the graph and its versions taking into account the temporal dimension explicitly. It is designed as a dynamic data structure for the management of changes in temporal graphs. Due to its dynamic nature, it can be easily built online, starting with an empty graph and inserting the changes as they appear. This is a common requirement that could also be implemented in simpler approaches based on snapshots. Additionally, the ttK^2-tree provides also the possibility of "correcting" the graph, that is, providing changes not only at the current time but also in past states of the graph.

A ttK^2-tree is conceptually a dK^2-tree in which all the leaves and internal nodes contain a list of timestamps that represent the time instants when the node has changed its value from 0 to 1 or from 1 to 0. These values represent valid time intervals for each position of the dK^2-tree. Figure 9.3 shows an example of the conceptual representation of a ttK^2-tree, with the valid intervals for each node with value 1. The 1s in the last level of the conceptual dK^2-tree are associated with edges of the represented graph, so their temporal information will contain the valid intervals for that edge. The 1s in the inner levels of the dK^2-tree are related, as always, with a submatrix, so their intervals represent the time intervals at which there was at least a valid edge in that submatrix. This means that the valid intervals for an inner node are simply the union of all the valid intervals for its children.

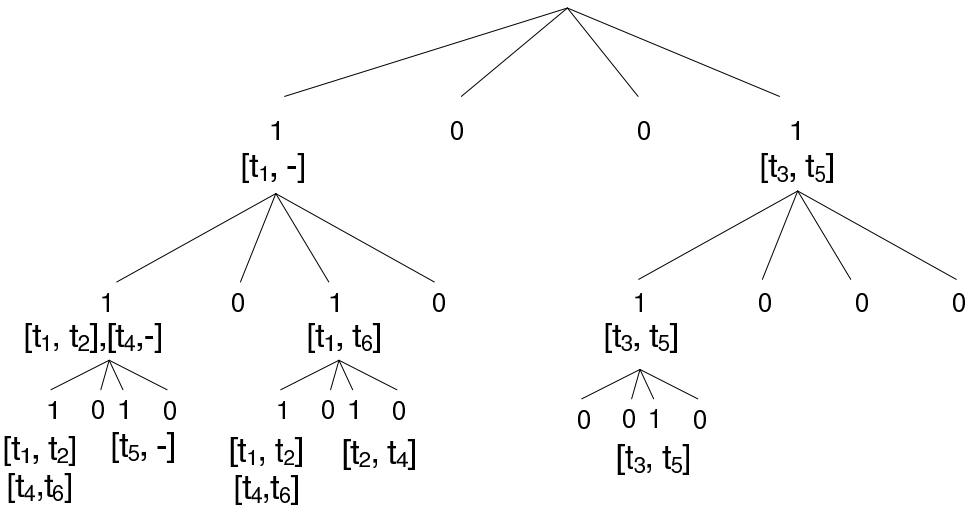


Figure 9.3: Conceptual representation of a ttK^2-tree.

9.3.1.1 Data structure and algorithms

The conceptual dK^2-tree with timestamps is implemented using two independent but related data structures that work together to index spatio-temporal information. A dK^2-tree plays the role of a spatial index, storing all the edges that ever existed in the graph.

To index the temporal information, we store a *change list* for each bit set to 1 in the dK^2-tree. Each change list is represented by a sequence of time instants at which the bit would have changed its value. These time values are encoded differentially and a header is added to each list to indicate its number of instants. For instance, for the leftmost leaf of the conceptual tree in Figure 9.3, we store the intervals $[t_1, t_2][t_4, t_6]$ as the sequence [4; 1, 1, 2, 2]. The values are encoded using variable-byte variations (in which the chunk size can be, for instance, 4 bits instead of 1 byte). We choose this representation because it is fast and easy to update, and because it behaves well even if the length of the list is relatively small.

The change lists are stored consecutively in a dynamic sequence, following the same order of the corresponding nodes. We store them in a tree structure $changes_{\text{tree}}$ similar to L_{tree}, in which internal nodes store counters with the number of elements (lists) in their subtree and the leaves store the actual change lists. In practice, we use a data structure almost identical to the L_{tree} with compressed vocabulary, where leaf nodes have a "logical" size (the number of lists stored in each node) that is indexed in the counters of internal nodes and used to find the appropriate leaf node for each list. Many change lists may be stored consecutively in a leaf node of our tree structure.

The dK^2-tree also suffers a small change, since we need to compute *rank* operations in the last level of the tree as well. Hence, we use the same tree structure used for T_{tree} also in L_{tree}, to obtain a dK^2-tree with *rank* support in its last level.

Query support

In the ttK^2-tree, any query can be reduced to the equivalent query in a dK^2-tree complemented with a set of temporal filters. All queries are implemented starting in the dK^2-tree representation and traversing all the nodes that may intersect with our query. For each node of the conceptual tree we traverse in the dK^2-tree, we retrieve the change list for the position of the node and check if the time instant or interval of the query fits the valid intervals for the node. Since the change list for any node represents the time intervals where at least one node in its associated region was active (i.e. the union of its children's valid intervals), if the query does not fit the valid intervals in the node we can stop traversal in its branch.

Given a node located at position p in the bitmap of T_{tree}, its change list will be at position $\text{rank}_1(T_{\text{tree}}, p)$ in $changes_{\text{tree}}$. For a node in L_{tree}, its change list can be located at position $T_{\text{tree}}.size + \text{rank}_1(L_{\text{tree}}, p)$. The process to locate the change list in $changes_{\text{tree}}$ is similar to a search in L_{tree}, with the only difference being the code to traverse the leaf nodes until we find the appropriate change list. Once the

change list has been found, we traverse it completely and check whether the time intervals stored in the list intersect with the time instant or time interval of the query.

The method to determine whether the query fits the valid intervals for a node is similar for time-slice and interval queries, with strong and weak semantics. In time-slice queries we simply need to check that the time instant is inside a valid interval in the change list. Starting from the beginning of the change list, we simply traverse it sequentially, updating the differential values to obtain the actual time instants that determine the valid intervals. A time instant will belong to a valid interval if the latest point in the change list that is before the time instant is at an odd position. In time-interval queries, depending on the semantics, we need to check that the query interval has a non-empty intersection with the valid intervals (*weak* semantics) or that the query interval is contained in one of the valid intervals (*strong* semantics).

Update operations

Update operations in a ttK^2-tree, similarly to queries, are based on traversing the dK^2-tree and updating the change lists for each node as necessary. The ttK^2-tree supports changes at any time instant, that represent the activation or deactivation of an edge. We will focus on the description of changes in the present state of the graph (that is, changes that affect time instants *after* all the existing time instants in $changes_{\text{tree}}$), describing the two main operations:

Marking active edges: To mark an edge as active we first search the edge in the dK^2-tree using the usual cell-retrieval algorithm. If the cell already existed in the dK^2-tree, we locate its change list in $changes_{\text{tree}}$ and add a new "open" instant marking the current time instant as the beginning of a new valid interval. Then, the change in the valid intervals must be propagated to the ancestors of the current leaf of the conceptual ttK^2-tree: they must also include the newly-created time interval. We check and update if necessary the change lists for all the nodes in the path from the leaf of the ttK^2-tree to the root, so that an open valid interval containing the current time exists in all of them.

Marking inactive edges: The procedure to mark an edge as inactive is similar to the insertion: we locate the position of the edge in the dK^2-tree and its change list. We update its change list in $changes_{\text{tree}}$ to close the last valid interval. After the edge has been marked as inactive, we must update the valid intervals of its ancestors. This operation is similar to the one where nodes are marked as valid. In this case, to determine whether or not the parent of the current node must also be marked as invalid, we must check all the siblings of the current node to see if any of them still has an open valid interval. If this is the case, the parent node must not be modified. If all the siblings have closed valid intervals (even-sized change lists),

the parent node must be marked as inactive at the current time and the process is repeated in the upper level of the tree.

Although we focused on changes in the current state of the graph, the same algorithms can be used to perform updates in past states of the graph. In this case, the assumptions on the size of the change lists are invalid but the techniques to update them and keep the ancestors up-to-date afterwards are identical.

9.3.2 Comparison with static representations

We compare our new proposal, that provides a fully-dynamic representation of the temporal graph, with the static representations introduced in Chapter 6.

Table 9.9 shows the space comparison between the ttK^2-tree and the static representations based on a K^3-tree and an IK^2-tree, using the same datasets and experimental setup of Chapter 6. The ttK^2-tree is competitive in space in the MContact dataset due to the small number of changes in the graph, that can be efficiently encoded in the additional data structure used to store changes in edges. On the other hand, as the number of changes in the dataset increases, the additional information stored by the ttK^2-tree (that includes the changes in all the internal nodes of the conceptual tree) makes it less competitive with other static representations.

Dataset	IK^2-tree	K^3-tree	ttK^2-tree
MContact	4.54	35.0	20.5
CommNet	136.51	147.2	400.0

Table 9.9: Space comparison of the ttK^2-tree with static variants to compress temporal graphs (sizes in MB).

To test the efficiency of the dynamic representation we compare its query times with those of static representations. As we have shown in Chapter 6, the representations we test here (the IK^2-tree and the K^3-tree) are in general slower than the ltg-index approach. Nevertheless, they are closer to the ttK^2-tree in the method for representing information, so we use them as a simple baseline for the query times of the ttK^2-tree. The results are shown in Table 9.10. The ttK^2-tree is slower than both static alternatives, and is particularly several times slower than the K^3-tree.

9.4 Summary

In this chapter we presented several applications of the dK^2-tree to different real-world problems, either using directly the dK^2-tree or building new variants using the dK^2-tree as the basis.

Dataset	Query	Semantics	IK^2-tree	K^3-tree	ttK^2-tree
CommNet	Direct	Instant	1622	96	2328
		Weak-10	1787	183	2970
		Weak-100	1776	539	4756
		Strong-10	1747	178	1525
		Strong-100	1745	542	513
	Reverse	Instant	1718	106	2398
		Weak-10	1971	211	3063
		Weak-100	1989	609	4901
		Strong-10	1927	210	1558
		Strong-100	1943	610	531
MContact	Reverse	Instant	143	78	227
		Weak-10	157	118	226
		Weak-100	163	149	234
		Strong-10	153	126	231
		Strong-100	145	130	212
	Reverse	Instant	147	89	274
		Weak-10	161	162	272
		Weak-100	159	180	285
		Strong-10	153	157	255
		Strong-100	149	177	260

Table 9.10: Time comparison of the ttK^2-tree and static representations of temporal graphs (times in μs/query).

First we proposed a new representation of RDF databases with support for updates using a collection of dK^2-trees. Our proposal is a dynamic version of an existing static representation that was proved to be very competitive with state-of-the-art solutions in single-pattern and join queries. Our experiments show that our proposal can require as little as 20% extra space on top of the static representation in some datasets, and we are less than 2 times slower than the equivalent static data structure in most queries.

We also introduced a variant of the dK^2-tree built on top of the K^2-tree1 presented in Chapter 5. This proposal provides the basis for a dynamic representation of binary images, or any binary matrix with clusters of 1s and 0s. We extend the experimental evaluation of Chapter 5 to show that our dynamic representation of binary raster images is competitive with a classic representation based on linear quadtrees, obtaining better compression and query times.

Finally, we also proposed a new fully-dynamic representation of temporal graphs using the dK^2-tree as the basis, and compared our proposal with the static

representations of temporal graphs introduced in Chapter 6.

Our proposals in this chapter show the flexibility of the dK^2-tree and the applicability of the trees T_{tree} and L_{tree} (the actual data structures that compose the dK^2-tree) to store bitmaps or sequences. This opens the door to the creation of dynamic variants of other data structures based on the K^2-tree, such as the K^n-tree or the IK^2-tree, that can be easily made dynamic replacing their bitmap representations with the tree structures used in the dK^2-tree.

Part III

Applications to GIS

Chapter 10

K^2-trees for general rasters

Most of the operations usually performed in a raster dataset are essentially based on a spatial filtering of the raster: local operations access a specific position in the raster, while a general window query performs an operation over a subregion of the overall raster. However, the group of zonal operations require additional capabilities to be performed efficiently: in an operation that accesses, for instance, all the cells with a given value, general spatial access methods do not provide any advantage over sequential processing, since they do not support filtering the queries by value.

As we have seen in Section 4.3, the representations of raster datasets based on simple tiling (either in compressed file formats or database schemas) provide some query support for spatial queries, but do not provide any support for the kind of queries that access the raster depending on the values stored in its cells. On the other hand, in Section 4.3.5 we have presented different representations that are specifically designed for the representation of this kind of rasters, and have the ability to filter regions according to their spatial location but also according to the values stored in the cells within the region.

In this chapter we will present our proposals for the compact representation of general raster data efficiently supporting typical queries involved in the representation of general raster data:

- *exists*$(v_1, \ldots, v_n, w)$: check whether any cell in the window w contains any of the values $v_1, \ldots, v_n$.
- *exists*$([v_\ell, v_r], w)$: check whether any cell in the window w contains any value in the interval $[v_\ell, v_r]$.
- *report*(w): return all the different values (features) contained inside window w.
- *select*$(v_1, \ldots, v_n, w)$: report all the occurrences of values $v_1, \ldots, v_n$ inside window w.

- *select*$([v_\ell, v_r], w)$: report all the occurrences of values in the range $[v_\ell, v_r]$ inside window w.

Notice that these operations are similar to those defined in Section 4.3.5 for general raster data or multiple non-overlapping features, but we extend them to support multiple values and ranges of values. All the proposals we will introduce in this chapter, are also valid for the representation of multiple non-overlapping features (for example, thematic maps), where each value in the raster represents the main characteristic of the region from a limited set. In the case of thematic maps, the added operations for ranges of values may not be useful because the order of the different features has no meaning to the user. However, when representing general raster data, numeric values in the cells have an explicit order and queries involving ranges according to this order are useful in many real-world queries: for example, in an elevation raster we may be interested in obtaining all the regions below sea level to determine potential flood risks, or finding all regions above a threshold to determine zones where it may snow. Essentially, our main goal is to provide an efficient representation that is able to answer queries restricted to a spatial window and/or to a value or range of values.

Consider a raster dataset that has any number of different values. Our proposals are based on indexing these values in different ways using the K^2-tree variants proposed previously. Our input data will be a matrix M of size $n \times n$ whose values come from a set V of possible values. Moreover, we assume that the values in M have the usual characteristics of spatial data: similar values tend to group in clusters or contiguous regions, and the difference between the values of two close cells will usually be small. Figure 10.1 shows an example of a matrix of this kind, where the set of different values is much smaller than the number of cells in the matrix. In this example matrix, the raster dataset has 5 different values, and cells with the same value are grouped in one or more clusters. Our proposals, in different ways, take into account the different values stored in the raster matrix and are able to provide efficient access to the raster by spatial coordinates and ranges of values.

The rest of this chapter is organized as follows: in Section 10.1 we will present our proposals for the compact representation of general raster datasets, based on the data structures introduced in Part I. In Section 10.2 we discuss the differences and expected strengths of the different proposals. In Sections 10.3 and 10.4 we provide an experimental comparison of our proposals with state-of-the-art representations of raster data to demonstrate the efficiency of our solution. Finally, in Section 10.5 we summarize the contributions and results in this chapter.

1	1	2	2	3	3	3	3
1	1	2	2	3	3	3	3
2	2	2	2	3	3	3	3
2	2	3	3	3	3	3	3
3	3	3	4	5	4	3	3
3	3	3	4	5	4	3	3
3	3	3	3	3	3	2	2
3	3	3	3	3	3	2	2

Figure 10.1: Example of raster matrix.

10.1 Our proposals

10.1.1 Independent K^2-trees: MK^2-tree1

Our first proposal is based on a simple partition of the raster dataset according to its different values. We partition the matrix in a collection of binary matrices M_i. Each M_i will contain the cells of M with value $v_i, i \in [1, |V|]$. This simple step reduces the problem of storing a general raster to the representation of simple binary images. The MK^2-tree1 approach represents each M_i using a different K^2-tree K_i. In order to capture the clustering of similar values the K_i's are K^2-tree1 variants instead of a simple K^2-tree to efficiently compress regions of 1s in the binary matrices (i.e. regions of equal values in the original matrix). Given the raster matrix shown in Figure 10.1, the partition into binary matrices and the corresponding conceptual tree decomposition for each binary matrix would be the ones shown in Figure 10.2. Our representation is essentially the MK^2-tree explained in Chapter 6 but using the K^2-tree1 variants proposed in Chapter 5 instead of classic K^2-trees.

We consider in all our proposals that the set of possible values is an integer range. In a general case, an additional data structure would be necessary to map the actual values to the corresponding offset in V. If we assume we have a manageable number of different values $|V|$, a simple data structure like a sorted array would suffice to provide this mapping: to find the appropriate index i from the value we perform a binary search in the array, with total cost $O(\log |V|)$. In all the queries we will disregard the cost of computing the actual index from the value obtained in the query, assuming that $|V|$ is small enough.

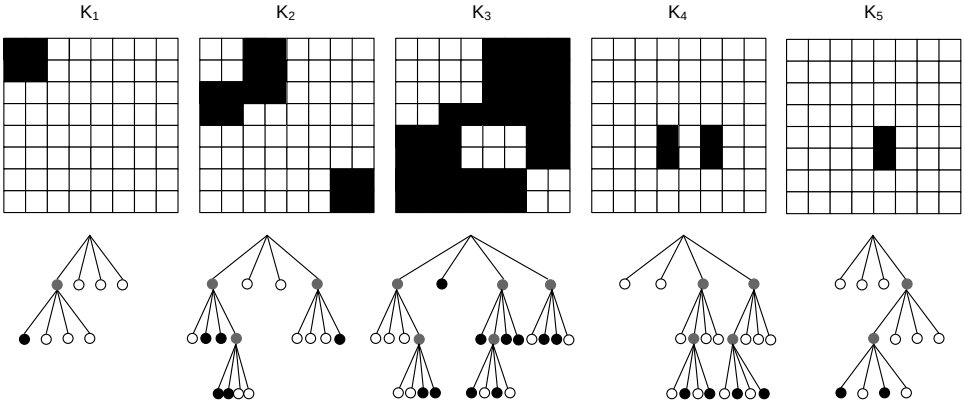


Figure 10.2: Representation of the raster matrix of Figure 10.1 using MK^2-tree1.

The MK^2-tree1 approach is very efficient to solve a subset of queries related to values, but it presents a tradeoff between purely spatial queries and queries involving values:

- To answer *select*(v_i, w) queries, that ask for all the occurrences of the value v_i inside window w, we only need to run a window query in K_i with window w. To obtain the result of *exists*(v_i, w), that asks whether or not at least a cell with value v_i exists inside w, we run the same query but stop after we find the first result.

- To answer general *select* and *exists* queries that involve multiple values $(v_i, v_j, \ldots, v_d)$ or ranges of values $[v_\ell, v_r]$, we need to run the same window query in all the K_is involved. In the MK^2-tree1 this is implemented either running the query sequentially in each K_i or with an *or* (union) operation involving all the K_i's in the range of values. Notice that in order to obtain also the values associated to each cell we can slightly modify the operation to return the index of the K_i that contains each 1 found, instead of just the cell.

- *report* queries, that ask for all the different values inside window w, are solved as a window query w with unbounded value. It can be implemented using a generalized union operation over all the K_i. Notice that this is a very important drawback of this approach, since all purely spatial queries (queries that only want to access positions in the raster independently of the value) become very costly due to the union required of all the K_i.

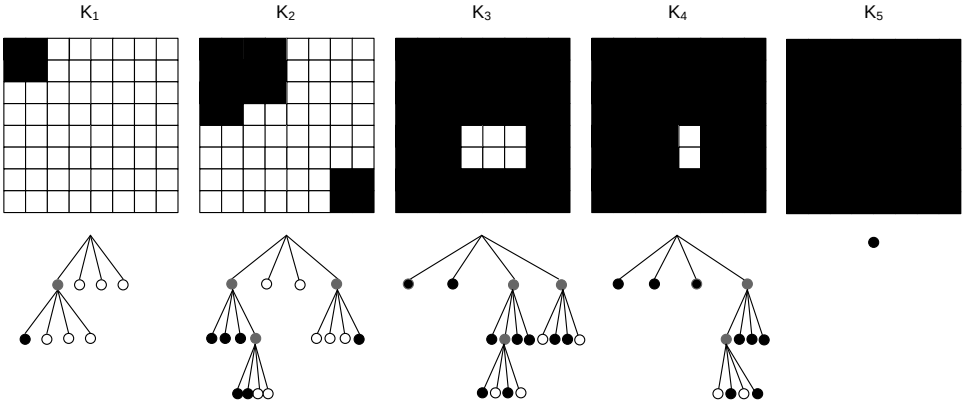


Figure 10.3: Representation of the raster matrix of Figure 10.1 using AMK^2-tree1.

10.1.2 Accumulated K^2-trees: AMK^2-tree1

Our second proposal is based on the same partition used in the MK^2-tree1, but tries to solve some of its drawbacks. The raster dataset is still considered as a collection of binary matrices. However, in this approach we consider *accumulated* values. For each different value v_i, we consider a binary matrix M_i whose cells are set to one if the value stored in the cell is *not greater than* v_i. Then, for each binary matrix M_i we build a K^2-tree K_i to represent it. Again, the K^2-trees will be our variations with compression of ones. Notice that each M_i (and each K_i representation) contains all the cells that were contained in the previous K_i in addition to the cells with value v_i, that is, $K_i^{accum} = \cup_{x=1}^{i} K_x^{base}$. Figure 10.3 shows an example of AMK^2-tree1 representation. Each K_i representation in this approach can be seen as the union of all the previous matrices in a MK^2-tree1.

A direct consequence of the method of representation is that $K_{|V|}$ does not require any space if all the cells have a value, since it should be a completely black matrix. However, we still consider it in our representation because it is needed if the representation contains NODATA cells (i.e. cells without a value). If the raster contains NODATA cells, $K_{|V|}$ contains all the cells in the raster that have any value, and is necessary to recover the original raster from our representation and to perform general range queries.

Another consequence of the construction method in AMK^2-tree1 is that we store many more 1s in the K^2-tree1s than in the previous proposal. However, this should not worsen the overall compression, because the K^2-tree1 variations used will take advantage of the clusterization of ones in the binary matrices. In fact, the use of accumulated values can improve the space results significantly, since it avoids the creation of holes in many matrices that are usual in raster datasets. Typical

examples are elevation or atmospheric pressure, that are sometimes displayed using concentric curves that represent the limits between different values (isobars for pressure in meteorological maps, contour lines for elevation in other maps). We can see an example of the effect of accumulating values in Figures 10.2 and 10.3: the representation of K_3 using MK^2-tree1 (Figure 10.2) needs to represent the border in the NW quadrant of the image, the first child of the root node is gray and a complete branch is generated; on the other hand, the representation of K_3 using AMK^2-tree1 (Figure 10.3) condenses the previous values and thanks to this the shape is simpler. As we said, and like in the previous proposal, the choice of representation for the binary matrices is a K^2-tree1 to take advantage of this property.

10.1.2.1 Algorithms

The main advantage of the AMK^2-tree1 proposal is its ability to efficiently handle queries involving ranges of values. In fact, the method to find all the cells with a single value or with values in a range is exactly the same:

- To answer a query *select* (v_i, w) or *exists* (v_i, w) we need to access two K^2-tree1s. From our construction process, we know that the set of cells that have value v_i are the cells stored in K_i (values $\leq v_i$) that do not appear in K_{i-1} (values $\leq v_{i-1}$). Hence both operations can be performed as a simple window query over $K_i - K_{i-1}$, where the subtraction algorithm is performed as explained in Section 5.5. The only difference between both queries is that *exists* queries return immediately when a result is found, while *select* queries compute the complete window query.

- A query *exists*$([v_\ell, v_r], w)$ can be handled using exactly the same procedure used for a single value, since the cells that have values in the range $[v_\ell, v_r]$ are those that appear in $K_r - K_{\ell-1}$.

- If we want to retrieve all the cells that have values within a range (*select*$([v_\ell, v_r], w)$) we can also perform the window query over $K_r - K_{\ell-1}$ to return the set of cells. Nevertheless, *select* queries implemented using a subtraction only return the set of cells that have values in the range. To compute the actual value of each cell we still need to perform a modified union operation that involves all the K_i in the range. This operation would access all the trees in the range and return for each black region only the first i such that K_i contains that region.

In spite of the fact that a synchronized traversal of many K^2-trees is still required, some operations can still be performed more efficiently in the AMK^2-tree1. Consider the example of obtaining the value of a specific cell (a 1×1 window). A general synchronized traversal needs to access $[K_1, K_{|V|}]$ to obtain the result. Using the AMK^2-tree1 characteristics, we can use a simpler method and perform a binary

search in the sequence of trees: each time we find the cell is black in a K_i we know the value is $\leq v_i$, each time we find the cell is white we know the value is greater. Finally, after $\log |V|$ accesses we obtain the actual value of the cell. For example, following the example of Figure 10.1, assume we want to find the value of the cell in row 3, column 1. The AMK^2-tree1 representation, shown in Figure 10.3, uses 5 matrices to store this raster. To find the actual value, we would first access the middle tree K_3 and find that the cell is in a black region, hence the value is $\leq v_3$, and we need to check the values to the left. We would repeat the process accessing now K_2 and find again a black region. In the third step we access K_1 and find a white region. Since the value appears in K_2 but not in K_1 the value of the cell is 2.

10.1.3 K^3-tree

Our third proposal is to use a K^3-tree to provide an indexed representation of the raster that can be accessed symmetrically by space and by range of values. We consider the raster values as a third dimension in a 3-dimensional binary matrix, that can be represented directly using a K^3-tree. Our K^3-tree representation will store the tuples $\langle x, y, z\rangle$ such that the coordinate (x, y) of the raster matrix has value z. This is essentially equivalent to stacking all the binary matrices used in the MK^2-tree1 proposal into a single 3-dimensional matrix and index this matrix using a K^3-tree.

Because of our construction method, for each coordinate (x, y) there will only be a single cell in the K^3-tree with value 1. A consequence of this is that the matrix will be relatively sparse (for a raster of size $n \times n$ with $|V|$ different values we have n^2 cells set to 1 from a total of $n^2|V|$). However, the binary matrix still has some level of clustering because cells with close coordinates (x, y) are expected to have similar values. Therefore, the K^3-tree compression can still be applied thanks to this locality. A second consequence of our construction is that there will be no 3-dimensional regions of ones in the generated matrix (for each (x, y) there is a single cell set to 1). Therefore, the K^3-tree we use in this approach is a simple K^3-tree *without* compression of ones, unlike all the other representations.

The implementation of the queries required in the K^3-tree is based on the modification of the generic traversal solutions presented in Chapter 6. An *exists* query involving any number of values, or a range of values, is directly translated into a 3-dimensional range query that limits the x and y coordinates to the boundaries of the window w and the z coordinate to the value(s) in the query. As in the previous cases, the query returns immediately when a 1 is found within the limits of the query. A similar implementation that returns all the results is used to answer *select* queries. To *report* all the values inside a given window or to obtain the value of a cell or cells in a window we simply perform another generalized range query, fixing only x and y. The K^3-tree takes advantage of the clusterization of close values to limit the number of accesses, so even if the z coordinate is not limited the properties

of the dataset should limit the number of accesses to a small number of branches of the conceptual tree.

10.1.4 IK^2-tree with compression of ones

The IK^2-tree, explained in Chapter 6, was devised as an alternative to a combination of multiple K^2-trees. Even though the IK^2-tree was originally conceived for classic K^2-trees, using the encodings proposed in Chapter 5 an IK^2-tree with compression of ones (or IK^2-tree1) can be easily built. In this section we propose this new variant, the IK^2-tree1, as an alternative and show how it is built and how it can support the required operations to represent raster data.

The first observation when building an IK^2-tree with compression of ones, or IK^2-tree1, is that if we use the K^2-tree1$^{\text{2bits}-\text{naive}}$ or the K^2-tree1$^{\text{2bits}}$ representation, the algorithms for finding the *children* of a node are identical. The representation of T' is based on the same interleaving technique, placing in each node of the conceptual tree the bits corresponding to the individual K^2-tree1$^{\text{2bits}-\text{naive}}$ or K^2-tree1$^{\text{2bits}}$. Figure 10.4 shows an example of representation of multiple binary images using individual K^2-tree1$^{\text{2bits}}$ representations and the final IK^2-tree1. Notice that, since the bits in T and T' are placed in the same order, the *color* of a node at position p in T can still be retrieved as $T'[\text{rank}_0(T, p)]$. The *internal* operation is not affected by the representation, and the *children* operation is performed like in the basic IK^2-tree representation.

We can use the IK^2-tree1 directly to merge the K^2-tree1 representations in the MK^2-tree1. Figure 10.5 shows the IK^2-tree representation of the K^2-trees shown in Figure 10.2, including the bits associated to each node using the K^2-tree1$^{\text{2bits}}$ representation and the final bitmaps for the IK^2-tree. The main advantage of this representation is that we do not need to traverse multiple K^2-tree representations to answer queries involving multiple values:

- To answer *exists* queries of the form $\textit{exists}([v_\ell, v_r], w)$, $\textit{exists}([v_i, \ldots, v_j], w)$ we now perform a traversal of the IK^2-tree for the nodes that intersect with the window. At each node we find we check the bits corresponding to all the values in our query. If we find a black node that corresponds to any value in our range, we can return immediately that at least a result exists.

- To answer *select* queries we perform the same window query used for *exists* queries, but now we run the complete query returning all the results for cells that intersect with the window.

- To answer *report* queries that ask for the different values inside a window are again performed as a window query in the IK^2-tree. For each node traversed that intersects with the query window, we check all its bits and for each bit that corresponds to a black node we add to the result its corresponding value in A.

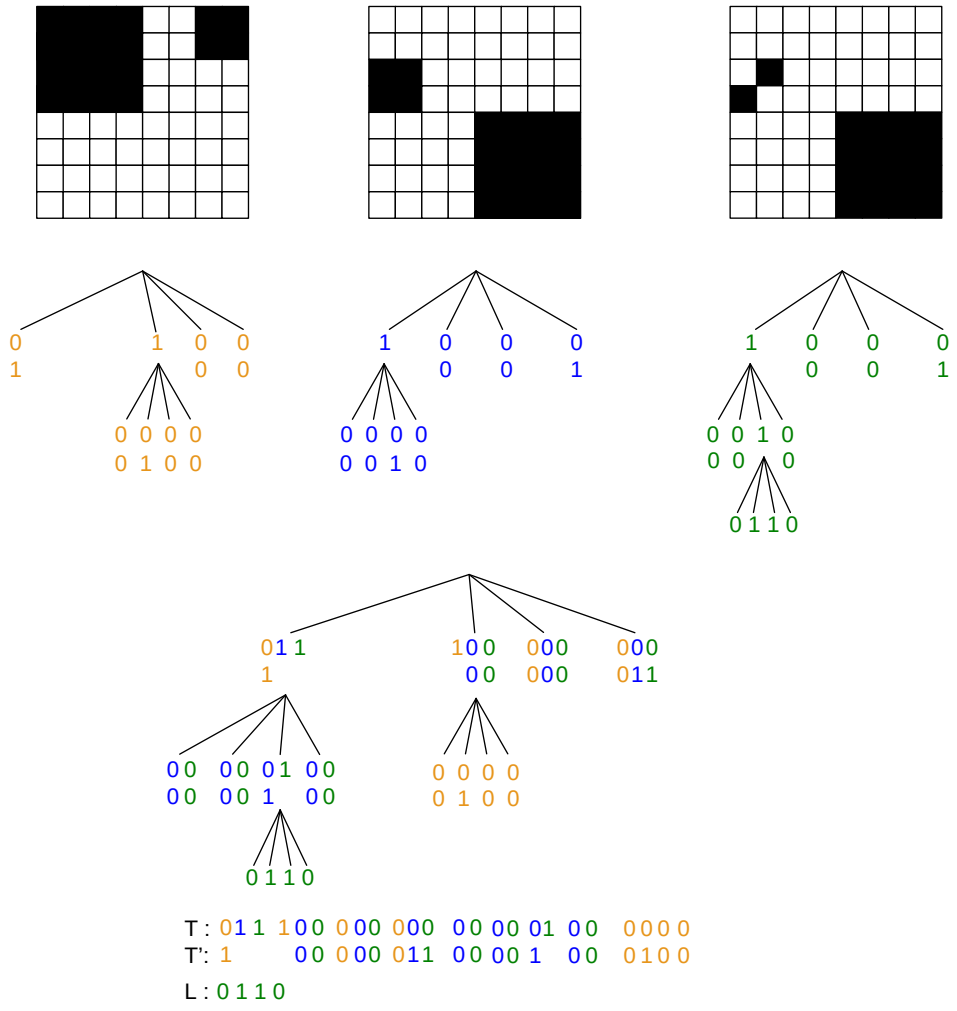


Figure 10.4: Multiple binary images represented using IK^2-tree1.

Example 10.1: Using the example in Figure 10.2, assume we want to *report* the different values that occur in the window $[0–1, 0–7]$ (the first 2 rows of the matrix). To do this in the IK^2-tree representation of Figure 10.5, we check all the branches that intersect with the window. We start at the root node and check the NW and NE quadrants. The NE quadrant (second child) is a leaf node (all the bits in T are set to 0). We check the corresponding bits in T' and find that the third bit,

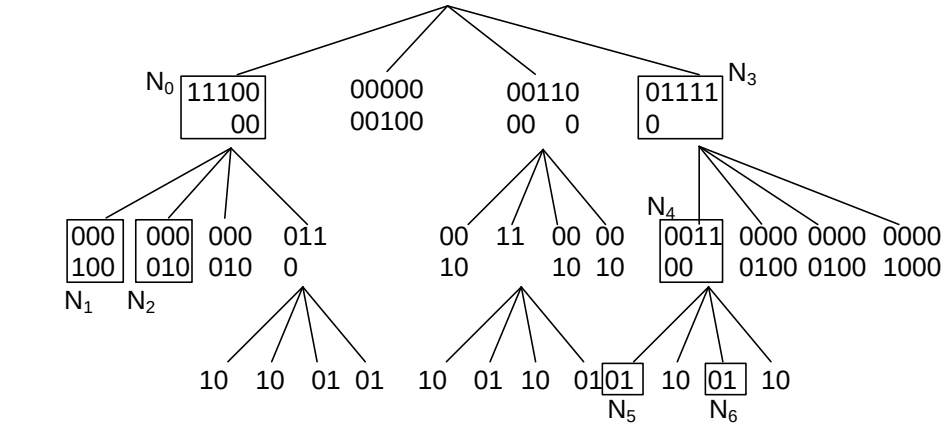


Figure 10.5: IK^2-tree1 representation of the raster in Figure 10.2.

corresponding to v_3, is a 1, hence we add V_3 to our result. The NW quadrant (node N_0) is an internal node, and from its bitmap we know that it contains cells with value v_1, v_2, v_3. We find the children of N_0, of which we only need to check the first two nodes N_1 and N_2 that intersect with our window. N_1 is white for v_2 and v_3 but black for v_1, so we add v_1 to the result. Similarly, the information in N_2 allows us to add v_2 to the result, that is finally $\{v_1, v_2, v_3\}$.

Example 10.2: Using again the example in Figure 10.2 and the IK^2-tree representation of Figure 10.5, we want to *select* all the occurrences of v_5. To do this, we start at the root of the IK^2-tree and check all its children. For each of them, we check the fifth bit corresponding to v_5. In the first three children, the fifth bit is a "white 0" (the bit in T is 0 and the corresponding bit in T' is also 0), so we do not traverse these branches. In the fourth child, the fifth bit is "gray", so we need to find its children. Since we are interested in a single value, we do not need to set A: we simply count the number of 1s in N_3 until the fifth bit (4 1s), and store this offset; the fourth bit in the children will be the one corresponding to v_5. The only child of N_3 whose fourth bit is not "white" is N_4: the fourth bit is "gray", and it is the second 1, so we will look for the second bit in all the children of N_4. Finally, we have reached the last level of the tree. We check all the children of N_4 and find that the second bit is set to 1 in N_5 and N_6. Hence, the only occurrences of v_5 in the raster correspond to the cells represented by these two nodes, in paths SE, NW, NW and SE, NW, SW.

10.2 Comparison of the proposals

Our representations are based on the compact and indexed representation of multiple non-overlapping binary images, specially designed for the representation of thematic maps or general raster data where queries *by value* are usually required. The MK^2-tree1 representation is particularly efficient to answer *exists* or *select* queries involving a single value, since it can query a single K^2-tree1. The AMK^2-tree1 is able to solve *exists* and *select* queries involving a range of values with the same query efficiency than single-value queries, using a synchronized traversal of two K^2-tree1s (this essentially doubles the cost of the MK^2-tree1 queries for single-value queries but is independent of the length of the range of values). Additionally, the AMK^2-tree1 representation may obtain better space results in multiple datasets since it can avoid the pattern of concentric curves that appears in the MK^2-tree1 representation. The K^3-tree representation is able to provide a simple algorithm for all the operations thanks to its symmetrical indexing capabilities in space and values. However, the K^3-tree representation depends a lot on the spatial correlation of values to obtain good compression and to answer queries efficiently. On the other hand, the IK^2-tree1 representation has a smaller dependency on the values and obtains the same compression as the MK^2-tree1. Additionally, the IK^2-tree1 representation is well-suited to answer queries involving ranges of values as well as multiple non-contiguous values, thanks to its structure that groups all the values for a node in contiguous bits in memory.

The MK^2-tree1 and the equivalent IK^2-tree1 representations of binary images are similar in their organization of the information to other state-of-the-art proposals for the representation of multiple overlapping or non-overlapping images, like the S*-tree, inverted quadtrees or the MOF-tree. In all these data structures, as in our representations, a bit is used to mark whether or not a feature/value exists in a region of the space. The MK^2-tree1 representation is mainly oriented to answer queries involving a single value or a small range of values, and may be inefficient in longer ranges. The IK^2-tree1 representation of the trees aims to solve this problem by storing all the values for each node together in a single tree. Even though the IK^2-tree1 navigation is slower than accessing individual K^2-trees (we need to check many bits and perform additional *rank* operations), the cost of the operations is not so dependent on the ranges of values. Particularly, the IK^2-tree1 representation uses a similar approach to the state-of-the-art proposals mentioned before, in the sense that it tries to group the information for a single node in order to answer efficiently queries by value. However, some key differences with those proposals must be noted. First, unlike inverted quadtrees, our representation is not a full quadtree but a tree representation tuned to the characteristics of the image. This means that our representation will have a much smaller number of nodes than inverted quadtrees in the vast majority of real raster datasets. Additionally, the IK^2-tree1 representation is able to store the feature information for all the nodes of the quadtree (both internal and leaf nodes) without explicitly storing m bits

in each of them. This is a significant saving in space: particularly in the case of non-overlapping images, we expect that as we descend in the tree the number of different values stored in the region will decrease significantly due to the properties of spatial raster data. Our representation adjusts the number of bits to store only information about the values that actually exist in the current region of the image. This can lead to huge space savings in comparison with state-of the art proposals. For example, consider a raster representation where a value is only present in a single cell. In a representation that stores m bits in all the nodes we are forced to pay an additional bit that is unnecessary in almost all the nodes of the quadtree. In our representation, the information about the infrequent value is only stored in the branches of the conceptual tree that are actually relevant.

10.3 Experimental evaluation

10.3.1 Experimental Framework

To test the efficiency of our proposals we use several real raster matrices obtained from the MDT05 collection. Table 10.1 gives details about the different fragments taken. The number of different values in each raster is also shown in the table, after rounding the elevation values to a precision of 1 meter.

Dataset	Raster size	#values
mdt-500	4001×5841	578
mdt-700	3841×5841	472
mdt-A	7721×11081	978
mdt-B	48266×47050	2142

Table 10.1: Raster datasets used.

We will compare all our representations in terms of space requirements and query times. As a baseline for comparison we convert all the datasets to the GeoTIFF file format, a widely-used standard for the storage of raster data. GeoTIFF builds on top of the TIFF file format, and provides several options to determine the level of compression applied to the files. We will use two different variants of GeoTIFF files: the first, *tiff-plain*, is a plain representation without compression, that stores all the values in row order (we use short (16-bit) integers as the data type for the representation, so the expected space utilization of this approach is always around 16 bits/cell). The second representation, *tiff-comp*, is optimized for a better compression: the image is divided in tiles of size 256×256 and each tile is compressed using a linear predictor and LZW encoding.

To measure query results for the usual window queries, we build sets of random queries corresponding to each case, each sufficiently large to obtain accurately-measurable results in all the representations. We will consider different typical queries that measure the ability of the representations to filter spatial ranges and stored values: retrieving all the cells with a given value, retrieving all the cells with values in a specified range, retrieving all cells with values in a range and within a spatial window. As an extreme case of *report* query, we also test the efficiency of the representations to retrieve the value of a single cell.

All the time results shown correspond to CPU time. We implement the same queries over the GeoTIFF images on top of the *libtiff* library, version 4.0.3, to retrieve the appropriate fragments from the GeoTIFF images and answer each query type. The libtiff library is not designed to process these queries, and is instead more oriented to process complete images from disk, so some of the comparisons may show worst results than expected when querying the tiff datasets. It must be taken into account that libtiff reads the images from external memory during processing time, however we are measuring only CPU time in all our experiments to provide a fair comparison.

All the experiments in this chapter are run on an AMD-Phenom-II X4 955@3.2 GHz, with 8GB DDR2 RAM. The operating system is Ubuntu 12.04. All our implementations are written in C and compiled with gcc version 4.6.2 with full optimizations enabled.

10.3.2 Space comparison

First we compare the space utilization of all our approaches with the GeoTIFF variants, used as a reference. Table 10.2 shows the space utilization of our proposals and the reference GeoTIFF images, in bits per cell of the raster. As an additional reference, columns 2 and 3 show the base-2 logarithm of the number of different values in each raster and the zero-order entropy H_0 of these values. These columns represent the minimum space that would be required by a representation of the raster as an uncompressed or entropy-compressed sequence, respectively. The K^3-tree clearly obtains the best space utilization among our approaches in all the datasets, being very close to the compressed GeoTIFF representation and using much less space than the zero-order entropy. However, as we will see, the tiff-comp approach provides limited access to the data and makes it hard to perform queries on the data. A significant result is the difference between the MK^2-tree1 and the AMK^2-tree1, where both approaches obtain relatively good compression of the datasets but none of them is clearly better than the other. Finally, the IK^2-tree1 obtains compression results very close to the MK^2-tree1 approach, as expected, but always obtains slightly better space results because some of the redundancies in the independent K^2-trees (particularly, their matrix vocabularies in the lower level of the tree) are eliminated when merging them in a single structure.

Dataset	lg(values)	H_0(values)	MK^2	AMK^2	K^3	IK^2	tiff-plain	tiff-comp
mdt-500	9.17	5.43	2.75	2.21	1.83	2.53	16.01	**1.52**
mdt-700	8.88	4.39	2.07	2.30	1.38	1.84	16.01	**1.12**
mdt-A	9.93	5.86	3.24	2.83	1.94	3.10	16.01	**1.52**
mdt-B	11.06	5.32	3.15	4.36	1.62	3.12	16.00	**1.35**

Table 10.2: Space utilization of all approaches (in bits/cell).

10.3.3 Query times

10.3.3.1 Retrieve the value of a single cell

First we measure the time required to retrieve the value of a single cell of the raster. This operation is the most extreme case of report query, where the query window is limited to a single cell and the range of possible values is not limited. Additionally, this particular query shows the ability of the representations to provide random access to the raster. Table 10.3 shows the results obtained. The K^3-tree obtains the best results in all the datasets for this kind of query. This result confirms the efficiency of the multidimensional index in this domain: the K^3-tree is able to restrict its search only to the coordinates of the cell, and the high locality of values in the real raster datasets allows it to filter many of the branches corresponding to values very different from the value of the cell. The MK^2-tree1, that needs to check multiple independent K^2-tree representations, is much worse than our other representations, since the relative cost of traversing all the K^2-trees is very high in this query. The AMK^2-tree1 obtains very good results, much closer to the K^3-tree than to the MK^2-tree1, thanks to the binary search it performs to answer this query. Finally, the IK^2-tree1 obtains query times much smaller than the MK^2-tree1 but still worse than the AMK^2-tree1 and the K^3-tree: in this case, the overhead required to keep track of the values in the IK^2-tree1 is relatively high compare to the total cost of the query. The approaches based on querying GeoTIFF images obtain very different results, as expected, depending on whether or not compression is enabled. The tiff-plain representation is very fast, obtaining results close to the K^3-tree (the results of this query in the tiff-plain approach should actually be even better, but the libtiff library is designed to process complete rows/tiles of the image at once, limiting the efficiency in this kind of query). On the other hand, the *tiff-comp* representation presents much higher query times compared to the uncompressed version, since it needs to retrieve the appropriate tile of the image and decompress it to recover a single cell. In all cases, the tiff-comp representation obtains much higher query times than our best representations.

Dataset	MK^2-tree1	AMK^2-tree1	K^3-tree	IK^2-tree1	tiff-plain	tiff-comp
mdt-500	123.6	7.2	**2.2**	31.4	2.6	491.7
mdt-700	66.0	5.9	**1.7**	26.5	2.7	461.9
mdt-A	132.4	10.1	**2.6**	45.6	5.2	499.0
mdt-B	421.4	11.1	**2.8**	75.9	87.9	494.8

Table 10.3: Retrieving the value of a single cell. Times in μs/query.

10.3.3.2 Retrieve all cells with a given value

Next we show the efficiency of the representations to select the cells of the raster that contain a specific value, a widely used selection query in raster data. The results for this query are shown in Table 10.4. Not surprisingly, our representations obtain much better results than the *tiff* representations, because the latter must always traverse the complete raster. In this case, the MK^2-tree1 obtains better results in all the datasets: only one K^2-tree is accessed and the regions of the ones in the K^2-tree can be decoded efficiently using the K^2-tree1$^{\text{2bits}}$ representation. The AMK^2-tree1 has to access two K^2-trees, hence its query times are close to doubling the times of the MK^2-tree1. The K^3-tree, like the MK^2-tree1, only needs to perform a query for a single value in its third dimension; however, the existence of other regions with close values forces it to explore many branches that will be then discarded, hence its query times are in general worse than the previous proposals. Finally, the IK^2-tree1 also obtains query times within a factor of 2 of the MK^2-tree1, since it performs a fixed-value traversal of the tree but the computations are more expensive.

Dataset	MK^2	AMK^2	K^3	IK^2	tiff-plain	tiff-comp
mdt-500	**3.9**	5.8	9.4	5.9	39.5	221.4
mdt-700	**3.0**	6.0	7.3	4.5	37.5	199.5
mdt-A	**8.2**	13.6	18.9	12.7	142.6	799.0
mdt-B	**110.2**	255.1	196.6	173.5	3,838.9	19,913.4

Table 10.4: Retrieving all the cells with a given value. Times in ms/query.

10.3.3.3 Filtering values and spatial windows

Next we measure the efficiency of general window-range queries, that is, general *select* queries that involve a spatial window and a range of values. This kind of queries, like the previous single-value-selection queries, are widely used to filter spatial attributes in regions of space, thus avoiding the complete processing of the

Dataset	Window size	Range length	MK^2	AMK^2	K^3	IK^2	tiff-plain	tiff-comp
mdt-500	10	10	10.3	**1.9**	1.9	25.8	33.0	533
		50	43.1	**2.2**	2.6	27.9	26.0	528
	50	10	14.3	**3.6**	5.0	28.9	124.0	694
		50	73.9	**6.0**	16.0	41.5	124.0	694
mdt-700	10	10	9.6	1.9	**1.7**	23.7	33.0	499
		50	46.1	**2.1**	2.2	25.5	24.5	495
	50	10	13.9	**3.9**	4.4	33.4	123.7	649
		50	69.7	**5.5**	13.4	37.1	123.8	649
mdt-A	10	10	10.6	2.6	**2.0**	35.9	82.0	550
		50	43.5	2.7	**2.5**	37.6	48.0	528
	50	10	14.6	**3.4**	4.2	39.2	229.2	699
		50	62.4	**5.1**	11.1	48.5	228.5	703
mdt-B	10	10	13.9	3.9	**2.3**	57.0	285.5	519
		50	60.8	3.9	**2.4**	56.9	927.0	521
	50	10	17.3	4.6	**3.1**	59.5	1,873.0	1,422
		50	55.8	**5.2**	5.7	62.7	1,009.6	691.9

Table 10.5: Retrieving cells inside a window and within a range of values. Times in μs/query.

raster. For different fixed window sizes and ranges of values, we build query sets asking for all the cells found within a square query window of the given size and within the range of values specified. Table 10.5 shows the results obtained for different window and range sizes. As long as queries involve any range of values, the AMK^2-tree1 and the K^3-tree become the fastest of our alternatives, with similar query times in most cases. The MK^2-tree1 query times increase linearly with the length of the range of values, making it very inefficient for ranges of 50 values. On the other hand, the IK^2-tree1 obtains high query times in short ranges, but in longer ranges becomes more efficient than the MK^2-tree1 thanks to its indexed representation of the values. Nevertheless, it is never competitive with the AMK^2-tree1 and K^3-tree, that take advantage of their structure to filter results very early. All our proposals are much faster in all the queries than the tiff-comp representation, overcoming it by at least a factor of 10 on average. The tiff-plain representation, that is not compressed, is much faster than the compressed version and obtains query times close to our less efficient representations, but is still at least 10 times slower than the K^3-tree in all the datasets. Regarding the results in range queries, particularly the comparison of the AMK^2-tree1 with the other representations, we

must note that all the other representations are able to return for this query not only the cells that fall within the range of values, but also the actual value of that cell. However, the query times of the AMK^2-tree1 proposal correspond to the subtraction query that is only able to return the list of cells but not their actual values. As we explained in the description of this proposal, if the actual value of each cell is required the AMK^2-tree1 is no longer efficient in filtering the results and must resort to a sequential traversal of all the K^2-trees in the range.

10.4 Top-k range queries in raster data

In this section we test the efficiency of our proposals to answer a different kind of query that is also usual in raster data: top-k range queries, that ask for the cells with maximum values in a spatial window. We have introduced in Chapter 7 a data structure called K^2-treap that is specifically designed to answer this kind of queries. In this section we will apply the K^2-treap variants for the better compression of uniform regions introduced in Chapter 7 to the representation of raster data, supporting range top-k operations. The goal of this section is first to confirm that our variants obtain reasonable compression and query times in this context, thanks to the fact that close cells usually have the same or close values.

We will test both of the variants introduced in Section 7.3. The first one, K^2-treap-uniform, uses a K^2-tree1$^{2\text{bits}}$ to stop decomposition when uniform regions are found. The second variant, K^2-treap-uniform-or-empty, stops decomposition when all the non-empty cells have the same value (but in the submatrix there may be empty cells and cells with value together). In our experiments we will compare the K^2-treap variants with the MK^2-tree1 representation tested in the previous section. The algorithm to answer range top-k queries in the MK^2-tree1 is a naive iterative method that runs the spatial range query in all the K^2-trees starting in the "highest" one, until k results are found. The reason to choose the MK^2-tree1 instead of the other alternatives studied earlier in this chapter is that the MK^2-tree1 is the fastest representation to answer range queries restricted to a single value, the essential query in the naive algorithm used for top-k queries.

10.4.1 Space comparison

First we measure the compression obtained by the K^2-treap variants in comparison with the MK^2-tree1. The other proposals studied earlier in this section could also answer top-k queries, working with a simple algorithm that first extracts cells from the highest value and continues until no more cells exist or k cells have been found; however, the MK^2-tree1 was already shown to be the fastest representation to retrieve cells with a fixed value, hence it will be the baseline to measure the efficiency of top-k queries in the K^2-treap variants.

Dataset	MK^2-tree1	K^2-treap	
		(uniform)	(uniform-or-empty)
mdt-500	2.75	3.18	2.87
mdt-700	2.07	2.19	1.98
mdt-A	3.24	3.51	3.20

Table 10.6: Space utilization of K^2-tree variants and MK^2-tree1 (in bits/cell).

Table 10.6 shows a space comparison between the K^2-treap variants and the MK^2-tree1. The K^2-treap-uniform does not obtain the same overall compression as the MK^2-tree1, but the K^2-treap-uniform-or-empty is able to obtain even better compression than the MK^2-tree1. In any case the K^2-treap variants provide good compression results, directly comparable to those of the previous representations. Notice however that this does not make the K^2-treap a good candidate to answer general range reporting queries since the navigation cost in the K^2-treap is much higher than in a simpler K^2-tree. However, its space efficiency makes of the K^2-treap a good alternative for domains where top-k queries are of interest.

10.4.2 Query times

Next we compare the average time to answer top-k queries in raster data using the three different approaches. For each dataset, window size and k we generate sets of random square windows within the bounds of the dataset and run each query set using the three proposals to measure the average query time. In Table 10.7 we compare the query times to perform range top-k queries for different ranges (spatial windows) and k values. The K^2-treap-uniform, the less compact of the K^2-treap variants, obtains the best results in the vast majority of the cases, independently of the window size or k value. The K^2-treap-uniform-or-empty is very competitive with the previous one when the value of k is small, since the additional overhead to check for repeated results is very small. Finally, for larger values of k, the MK^2-tree1 is more competitive with the K^2-treap variants, due to the additional navigation cost required in the K^2-treap to retrieve values and coordinates of cells.

10.5 Summary

In this chapter we introduced several compact data structures for the representation of general raster data with advanced query support. We provide representations that are able to store real raster datasets in small space and provide at the same time efficient access not only to regions of the raster but also advanced query capabilities,

such as selecting cells that look for a particular value or range of values, or queries finding the top-k cells with highest value in the raster. Most of the proposals are direct applications of data structures proposed in Part I or are based in those data structures. Additionally, most of the approaches introduced can be transformed into dynamic solutions using the dK^2-tree introduced in Part II.

Our proposals obtain good compression results in the studied datasets and are able to answer queries restricted to a spatial window and/or a value/range of values. We analyze the differences between our representations to show their strengths, and experimentally evaluate all of them to measure their compression and query efficiency. We compare our representations with a classic state-of-the-art representation of raster data, showing that our proposals are very compact and easily overcome classic tools, designed mainly for processing the complete raster. Additionally, the compression results obtained by our representations make them several times smaller than state-of-the-art representations based on linear quadtrees, being able to store and query large datasets in main memory.

We also test different representations to answer top-k queries in raster data. Our experiments confirm the space efficiency of the K^2-treap variants introduced in Chapter 7, that are competitive in space with our other representations of raster data and faster to answer top-k queries.

In our experiments we show the scalability of our representations to efficiently represent rasters with several thousands of different values. Nevertheless, the space efficiency of most of our proposals will degrade if the number of different values in the raster becomes too high. As a consequence, an implicit assumption in our proposals is that the number of different values in the raster dataset is not too high. We claim that in many real-world datasets, even though the values actually stored may have a high precision (most often being stored in floating-point numbers), the precision in the values does not add any quality or accuracy to the dataset after a given threshold: when measuring spatial features such as temperature, elevation, pressure, etc. the actual measurements may be taken with high-precision methods but the interpolation of values, or even the simple averaging of measurements, distorts the precision of the measurements, so we can safely reduce the precision of the values significantly without reducing the quality of the dataset for many purposes.

Dataset	Window	k	MK^2-tree1	K^2-treap	
				uniform	uniform-or-empty
mdt-500	100	10	165	14	16
		100	176	39	49
		1,000	279	234	386
	500	10	133	16	16
		100	143	41	51
		1,000	219	237	375
	1000	10	126	16	16
		100	135	41	52
		1,000	199	217	358
mdt-700	100	10	337	12	13
		100	352	31	37
		1,000	459	176	285
	500	10	310	15	16
		100	338	41	48
		1,000	495	243	383
	1000	10	284	16	17
		100	313	46	54
		1,000	518	283	443
mdt-A	100	10	499	18	21
		100	533	43	51
		1,000	609	239	373
	500	10	399	22	23
		100	456	51	59
		1,000	582	253	395
	1000	10	483	22	22
		100	424	49	57
		1,000	556	256	441

Table 10.7: Query times to answer top-k queries (times in μs/query).

Chapter 11

K2-index for time-evolving region data

Time-evolving region data usually follows some rules that allow its efficient compression. At any point, the binary raster representation of moving regions is expected to contain relatively large clusters of ones and zeros that can be efficiently compressed. Additionally, the changes in the shape and size of a region are progressive in time, so the expected number of changes between consecutive time instants is relatively small. The usual representations of moving regions take advantage of these characteristics. Also, efficient representations of moving regions must be able to answer different types of queries. In Section 4.3.6 we have presented different proposals for the compact representation of time-evolving raster data using data structures based on the regular decomposition of the space. In this chapter we will present our proposals for the compact representation of time-evolving raster data. Our proposals are based on the K^2-tree variants presented before and rely on the compression and query efficiency obtained by K^2-tree representations to provide an efficient and simple representation of the time-evolving features.

The rest of this chapter is organized as follows: in Section 11.1 we present our proposals for the representation of time-evolving raster data or general multiple overlapping features, considering their characteristics and their ability to represent multiple images depending on the level of similarity between consecutive images. In Section 11.2 we present an experimental evaluation of our proposals that shows their space and time efficiency. Finally, in Section 11.3 we summarize the contributions and results of this chapter.

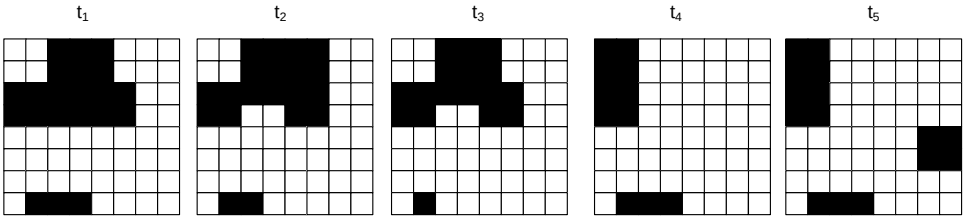


Figure 11.1: Temporal raster dataset.

11.1 Our proposals

Our proposals for the representation of temporal raster data, or time-evolving binary rasters, are based on K^2-tree representations similar to those presented in Section 10.1. The main idea is to consider a time-evolving binary raster as a collection of binary rasters corresponding to the different time instants.

Essentially we end up with a collection of binary matrices that can be efficiently compressed with the representations proposed in previous chapters. We present next our alternatives for the representation of temporal raster data.

Multiple K^2-trees: MK^2-tree1. Our first proposal is to use the MK^2-tree1 presented in Section 10.1 to index the collection of binary images. The data structure used to store the collection of binary rasters is essentially the same: we have a collection of K^2-trees $K_i, i = 1 \ldots t$, each representing the state of the raster at a different point in time. Again, the binary matrix that represents the raster at each time instant should be clustered, so the variants of K^2-tree with compression of ones are used to store each K_i.

This representation allows direct access to the complete state of the raster at each time instant. Each K_i can be queried independently to run window queries at a given time instant.

Time-interval queries can be easily solved using synchronized query operations over multiple K^2-trees: to find all the cells that were covered at any point within an interval $[t_{start}–t_{end}]$ we perform an *or* operation of all the K_is in the interval. To find the cells that were covered at all points within an interval, we perform an *and* operation on the K_is involved.

K^3-tree with compression of ones. Our second proposal for the representation of moving regions is to use a K^3-tree to provide an indexed representation of time and space. The sequence of binary matrices is considered as a 3-dimensional matrix where time is the third dimension, and a K^3-tree is build to represent this matrix, like we showed in Section 10.1.

A time-evolving raster dataset may consist of very slow changes between consecutive time instants, so large regions of ones will be similar in several consecutive snapshots of the raster. This leads to the possibility of compressing the large regions of ones present in the 3-dimensional matrix we generate. To take advantage of the repetitions between consecutive time instants we build in this case a K^3-tree variant with compression of ones. Therefore, large regions of ones or zeros that change slowly over time may be compressed efficiently by the K^3-tree representation.

The most relevant queries in time-evolving region data can be solved using a simple query on the K^3-tree representing the sequence of binary rasters. Time point queries simply ask for the regions of the matrix with fixed third coordinate t. Time interval queries require a more careful implementation to take into account the nature of the intervals. If we want to find the cells that were covered within a time interval together with all the time points where they were covered we simply need to perform a range query on the K^3-tree limiting the time dimension to the given interval. All the cells found within the region will correspond to different points and instants when they were covered.

To find cells in a *weak* interval, we need to perform the same range query explained for simple time interval queries. However, each coordinate (x, y) must be returned only once independently of the number of cells (x, y, t) found in the K^3-tree. A simple solution is to compute all the results in the K^3-tree and then filter out repeated coordinates. A similar solution can be used to solve *strong* interval queries: compute the total number of results and filter out the coordinates that do not have results at all times in the interval. A more elaborate solution for both queries can be obtained arranging the navigation of the K^3-tree algorithms so that the results are returned sorted by (x, y) coordinates. Using a sorted traversal like this, weak and strong queries can be optimized so that the remaining branches for the current (x, y) coordinates are not explored as soon as a 1 or 0 is found respectively: in weak interval queries, the coordinates are added to the result as soon as we find the first 1, in strong interval queries we immediately discard coordinates after we find a 0 in any t in the interval.

IK^2-tree1. The proposal based on an IK^2-tree1 for combining multiple K^2-tree representations into a single tree, presented in the previous chapter, can be applied directly to the representation of time-evolving raster images. The IK^2-tree1 representation does not assume that the images are non-overlapping (although some optimizations can be performed if this is the case). Hence, the application of the IK^2-tree1 representation to time-evolving data is immediate. The process to answer time-slice queries is identical to the *select* query that asks for cells with a given value: $select(v_i, w)$ is equivalent to a time-slice query replacing the v_i by the appropriate time instant. In addition, a general query that asks for all the occurrences of cells in multiple timestamps is equivalent to a $select([v_\ell, v_r], w)$ query in a general raster.

We will focus on the explanation of the changes required to perform *weak* and *strong* interval queries.

To perform a weak interval query we need to check the nodes that were active at least at one instant within the interval. During the window query, we keep track of the bits in each node that fall within our interval (this can be obtained easily computing the *offset* of the first point and the *offset* of the last point with *rank* operations). Then, if the current node contains *at least* a "black bit" in the bits corresponding to our interval, we can return immediately the region covered by that node and stop traversal. Otherwise, we keep traversing its children (if it has any), adjusting the interval of bits that we must check.

To perform a *strong* interval query the algorithm is similar to the previous one. During the top-down traversal, we keep track of the bits corresponding to our interval. We will return only the nodes that only contain "black bits" in our interval. If the node contains at least a "white bit" in our interval, we stop decomposition of that node immediately (no children will be black for that time instant). If the node contains a mix of gray and black bits we must keep traversing to find which of its children are completely black in our interval.

11.2 Experimental evaluation

11.2.1 Experimental Framework

We test the efficiency of our proposals using several different synthetic and real datasets. Table 11.1 shows some details about the datasets we use. The real datasets CFC1 and CFC2 were obtained from the Satellite Application Facility on Climate Monitoring (CM SAF[1]), and they store values of cloud fractional cover. Both are actually temporal rasters that contain daily mean values of cloud cover worldwide, with a spatial resolution of 0.25 degrees, between 1982 and 1985 (CFC1) and between 2007 and 2009 (CFC2). In order to build our input data, we apply a threshold (in this case, require a value greater than 50%) to convert each raster to a binary raster, so that the resulting collections shows the temporal evolution of moving regions (in this case, movements of clouds). The synthetic datasets RegionsA and RegionsB were built using a custom generator, and contain large regions that move and change shape slowly along time. These synthetic datasets are built generating shapes by clustering circles of variable size in a region of space and randomly adding or removing bits to their borders in several iterations. After these iterations irregular shapes are formed, and isolated pixels are cleaned to obtain smoother images.

We run all our experiments in a machine with 4 Intel(R) Xeon(R) E5520 CPU cores at 2.27 GHz 8 MB cache and 72 GB of RAM memory. The machine runs

[1] http://www.cmsaf.eu

Dataset	Raster size	#time instants	#Cells covered	% covered
CFC1	720×1440	1111	778,541,888	67.6
CFC2	720×1440	918	555,708,566	58.4
RegionsA	1000×1000	1000	236,634,823	23.7
RegionsB	1000×1000	1000	242,190,966	24.2

Table 11.1: Datasets used in our experiments.

Ubuntu GNU/Linux version 9.10 with kernel 2.6.31-19-server (64 bits). Our code is compiled using gcc 4.4.1, with full optimizations enabled.

11.2.2 Space Results

We measure the space utilization of our proposals using two different values of K, $K = 4$ in the first level of decomposition and $K = 2$ in the remaining levels. We use a $K' = 4$ in the last level (i.e., we compress 4×4 and $4 \times 4 \times 4$ submatrices in L in the different approaches).

Dataset	K^3-tree	MK^2-tree1	IK^2-tree1	Quadcodes	
				naive	diff
CFC1	1.11	0.71	0.55	6.73	5.01
CFC2	1.37	0.83	0.65	7.53	5.77
RegionsA	0.04	0.09	0.06	0.64	0.16
RegionsB	0.03	0.08	0.06	0.53	0.13

Table 11.2: Space results obtained. Compression in bits per one.

Table 11.2 shows a summary of the results for all the datasets. In the last two columns we provide, as a reference, the space that would be needed to store the quadcodes in a Linear Quadtree representation of the same datasets (we only consider the size of the quadcodes, ignoring overheads added by the data structure). The column *naive* represents the size required to store the complete raster at each time instant. The column *diff* shows the size to store only the *new* quadcodes at each time instant. Notice that this size is a lower bound for an Overlapping Linear Quadtree, that stores at each timestamp a new Linear Quadtree reusing unchanged nodes (however, the OLQ must repeat all the common quadcodes in a node if only one of them changes). We use this value as a reference because the OLQ is in general the most compact approach based on Linear Quadtrees [TVM04]. The results, shown in bits per covered cell (i.e. bits per one) are good in all datasets (ranging from less than 0.1 bits per one to a little over 1 bit per one). Our best representation is always 10 times smaller than a naive representation based on

Linear Quadtrees and 4-8 times smaller than the lower bound for OLQs. However, we can see considerable differences between datasets. In CFC1 and CFC2 the multiple K^2-trees and the IK^2-tree1 obtain the best results, being significantly smaller than the K^3-tree. On the other hand, in the synthetic datasets RegionsA and RegionsB the results are inverted: the K^3-tree obtains the best results among all the representations, and the MK^2-tree1 approach is the worst. In all cases, the IK^2-tree1 is smaller than the MK^2-tree1, because a lot of redundant information is stored in the latter to keep track of the independent vocabularies for each K^2-tree. The discording results in different datasets can be easily explained taking into account the nature of the data: in CFC1 and CFC2, the moving regions are taken from daily snapshots, so the regions change very significantly in a few snapshots. This means that the K^3-tree cannot take advantage of regularities in the temporal dimension, because it needs large clusters of ones to appear in the matrix to obtain its best results. On the other hand, the "multiple K^2-trees" approaches compress each snapshot independently so they can obtain good compression even if consecutive snapshots have nothing in common. In RegionsA and RegionsB, that represent regions changing and moving very slowly, the results are inverted because the K^3-tree takes advantage of the high regularity in the time dimension.

To provide additional insights on the differences between the K^3-tree and multiple K^2-trees depending on the characteristics of the dataset, we analyze the evolution of compression in both approaches taking into account an additional parameter: the change rate of each dataset, that is, how fast the contents change between consecutive snapshots. To study this variability, we use again synthetic datasets. We use the dataset RegionsA and build another 2000×2000 dataset RegionsC following the same principles. In order to obtain datasets with different change rate, we create datasets with only 100 time instants extracted from RegionsA and RegionsC with different granularity: a dataset will contain all the first 100 snapshots, another will contain snapshots $1, 3, \ldots, 199$, another snapshots $1, 4, \ldots, 299$, etc. In this way we build in practice multiple datasets that correspond to the same moving regions seen with different time granularities.

Figure 11.2 shows the evolution of the compression obtained by our different proposals depending on the change rate of the dataset. The change rate is the average number of cells that change value divided by the average number of ones in all the snapshots (i.e. the percentage of the moving region that changes or moves at each time instant). There is a threshold that divides datasets in which a K^3-tree will obtain better compression and datasets that are better represented with multiple K^2-trees. This threshold may depend on the actual dataset, but in our experiments is shown to be relatively small. This is due to the fact that, even for a region changing slowly, after a small number of snapshots a large percentage of the region may have changed. This limits the maximum size of the cubic regions of ones we can find in the binary matrix of the K^3-tree, and therefore its compression efficiency. We believe that for any real moving regions dataset with very slow changes a K^3-

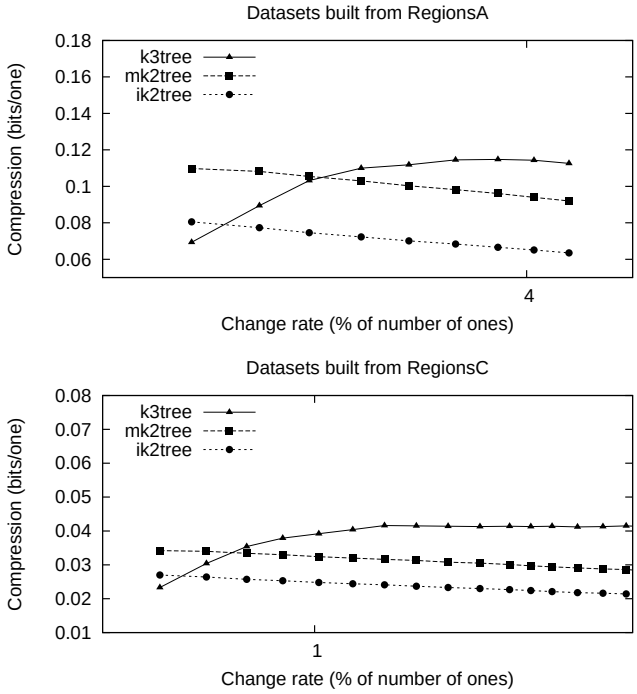


Figure 11.2: Evolution of compression with using a K^3-tree and multiple K^2-trees.

tree should be a better representation. However, the strategy of using multiple K^2-trees or an IK^2-tree1 becomes more efficient in space when changes in the third dimension are significant. Notice that the datasets with high change rate will also be very difficult to compress using any other proposal based on encoding changes, like OLQs, MVLQs or any other of those introduced in Section 4.3.6.

11.2.3 Query times

In order to compare our proposals we measure the query times in two different categories: snapshot queries and time interval queries. We generate sets of 1,000 random queries corresponding to different spatial window sizes, always using square windows. The queries are executed several times on all approaches and we take the average time per query. Figure 11.3 shows the results obtained for all the datasets and approaches. As we can see, the MK^2-tree1 obtains better results in all the queries, since it only needs to access a K^2-tree for the corresponding instant. Additionally, the K^3-tree is forced to go deeper in its conceptual tree because the

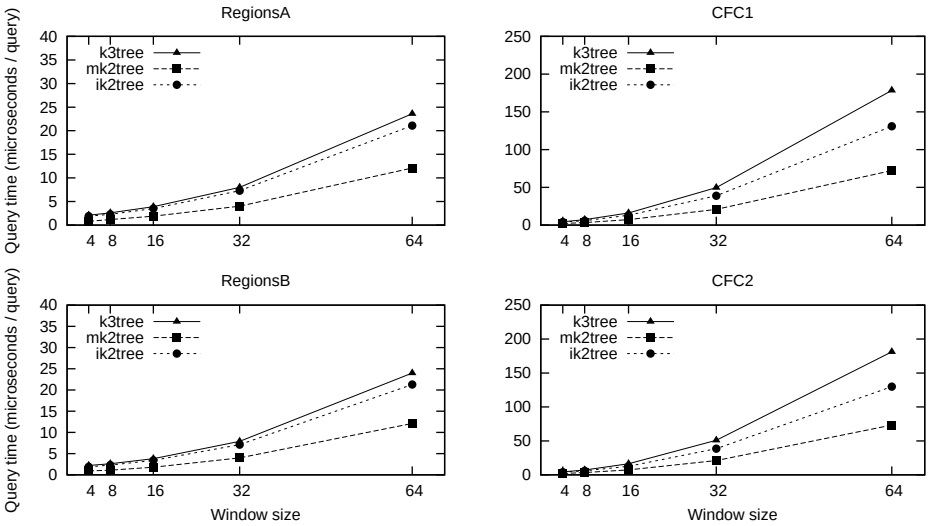


Figure 11.3: Query times for snapshot queries in all the datasets.

uniform regions found in each K^2-tree are in general much larger than the cubic regions the K^3-tree can find, and because of this the K^3-tree compression is limited to fragments of the regions that are covered for a long interval. The IK^2-tree1, as expected, is slower than the MK^2-tree1 by a factor of at most 2, and still faster than the K^3-tree in all the snapshot queries studied.

Next, in order to show the tradeoff between the different proposals, we measure the efficiency of our proposals in interval queries. First we use the simplest interval query, the one that asks for all the cells covered at each time instant within a given time interval. We generate sets of 10,000 random queries, each set with a different interval length. Figure 11.4 shows the query times obtained for a window size 32×32 and interval lengths 1 (snapshot query) to 40. The K^3-tree, that can restrict its search to the region enclosed by the window and the interval, obtains the best query times for all interval queries. The MK^2-tree1 and the IK^2-tree1 obtain very close results in the synthetic datasets RegionsA and RegionsB, but the IK^2-tree1 becomes much slower in the real datasets. This is due to the fact that the IK^2-tree1 must keep track of all the active times that fall inside the query interval. Notice also that this result is very different from the comparison obtained in Chapter 10, where we used essentially the same data structure and algorithm: when compressing raster data, the number of active values for any node in a spatial window will be relatively small, because of the locality of values; on the other hand, in spatio-temporal data many time instants may be active for a spatial window up to the lower levels of the tree, adding a significant overhead to the IK^2-tree1 in mapping all of them.

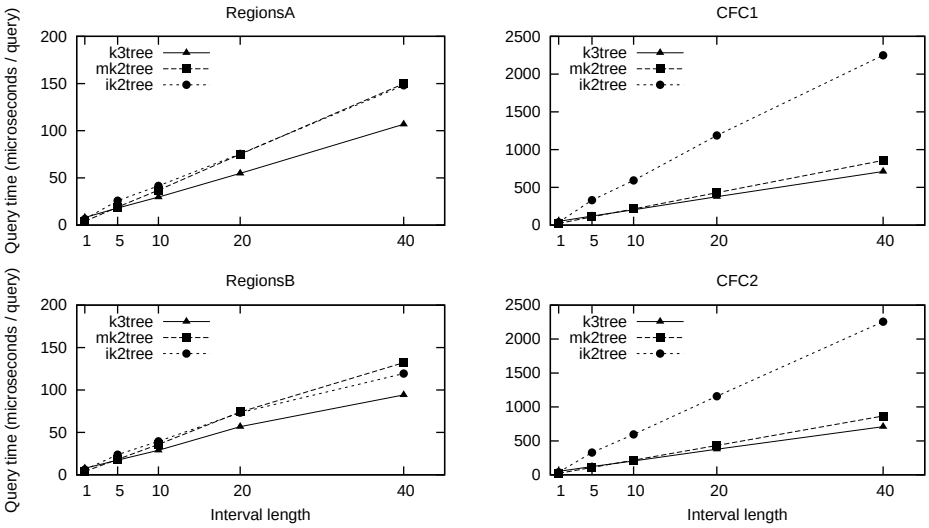


Figure 11.4: Query times for interval queries in all the datasets, for fixed window size 32.

We run the same query sets using now the two semantics associated with time-interval queries: weak queries ask for the cells active at any point within the interval, strong queries ask only for cells that were active during the complete time interval. The results are shown in Figure 11.5 for weak queries and Figure 11.6 for strong queries. The IK^2-tree1 obtains the best query times when the time interval is longer, since it can efficiently check the values for the complete interval in each node. For very short intervals, the MK^2-tree1 is still the best approach, due to the additional computations required to navigate the IK^2-tree1. The K^3-tree becomes slower in general than the IK^2-tree1 because it is not able to filter out nodes in the upper levels of the tree. The results also show that in the synthetic datasets (left side of the figures) query times are better in general and the K^3-tree is more competitive. This is due to the reduced change rate of these datasets, that makes it easier to filter nodes.

11.3 Summary

In this chapter we introduced compact representations for temporal raster data. Our proposals are very similar to those presented in the previous chapter, that are extended to manage the specific interval queries required in temporal data.

Our representations are similar to state-of-the-art compact representations of temporal raster data, and require much less space than alternatives based on classic

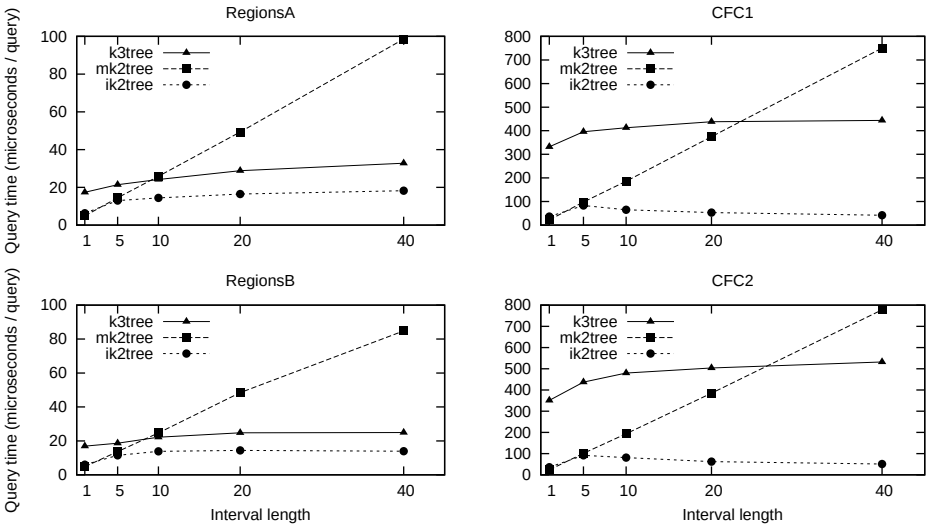


Figure 11.5: Query times for weak interval queries, for window size 32.

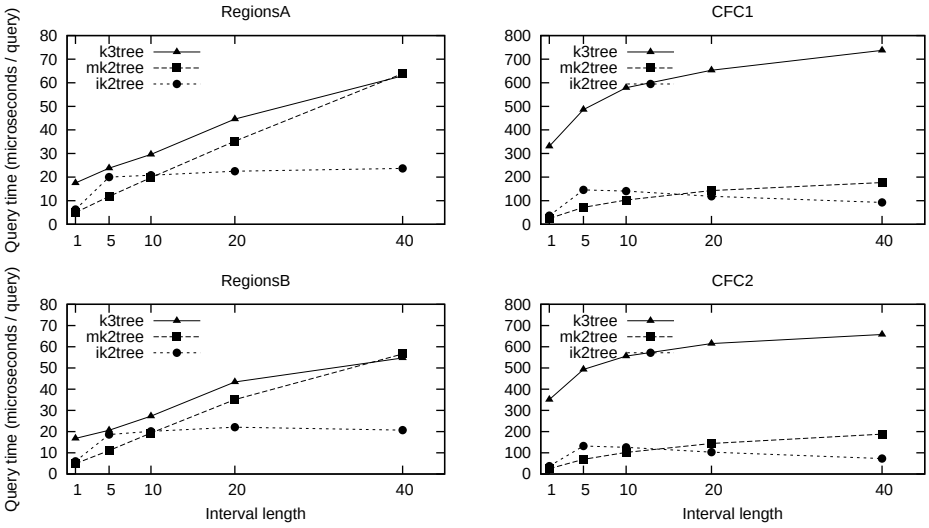


Figure 11.6: Query times for strong interval queries, for window size 32.

data structures like the linear quadtree. Our experimental evaluation shows that our proposals are especially suited for different kinds of datasets: the K^3-tree obtains

the best space results for slowly-changing datasets, but as soon as the change rate increases the approaches based on multiple K^2-trees become smaller.

Our results in this chapter also confirm some of the results obtained in Chapter 6, where the advantages and drawbacks of both representations were studied in a general case. The evolution of the K^3-tree, the MK^2-tree1 and the IK^2-tree1 is consistent with the results obtained for temporal graphs and confirms the idea that the K^3-tree is an efficient alternative to the MK^2-tree and the IK^2-tree, but the characteristics of the datasets should be taken into account to select between the representations, since the change rate of the dataset has significant effect in compression and query times.

Chapter 12

Join operations between raster and vector representations

In recent years the availability of different types of spatial data has increased, as many national geographic agencies are using and publishing digital cartography, and other private organizations make extensive use of geographic information obtained using airborne and satellite technologies. This leads to an increasingly large amount of vectorial and raster information available for its use.

However, many application domains require accessing vector and raster data at the same time. A typical example would be emergency management, where the evolution of fires or floods (typically represented in raster form) must be checked considering road networks or populated areas that may be affected.

Most of the models and languages designed for the representation of spatial data in traditional database models include data types and operators to represent and query vectorial and raster data [SH91, BDF+98, VZ09]. However, in many cases the user is forced to know and use different sets of operations for vector and raster data. While multiple access methods have been developed to query raster or vector data [Sam90b, GG98], not much work has been devoted to the development of algorithms to simultaneously query data structures for vectorial objects and data structures for raster data.

In this chapter we present a method to perform a spatial join between a vectorial dataset represented using an R-tree and a raster dataset represented using the K^2-tree1 variants introduced in Chapter 5. This method can be used to evaluate queries between vector and raster data without having to convert one of the datasets to the other data model. This method is also extended to perform joint queries where the raster dataset is not binary; in this case we adjust our algorithms to work with the

variants covered in Chapter 10, particularly the AMK^2-tree1.

The rest of this chapter is structured as follows: Section 12.1 presents some related work in the combination of representations following the raster and vector models. Section 12.2 presents our proposal, a query algorithm that allows efficient join operations between a raster dataset stored in a K^2-tree1 and an R-tree representation of objects following the vector model. In Section 12.4 we experimentally evaluate our algorithm using different real-world datasets to show the efficiency of the join operation to filter results from the raster dataset. Finally, a summary of the chapter is presented in Section 12.5.

12.1 Querying vectorial and raster data

As we have said, not much work has been devoted to algorithms or data structures that support queries combining raster and vectorial datasets. Some previous work has been presented [CVM99, CTVM08] dealing with binary rasters, as in our case. The first one presents five algorithms for processing joins between an R-tree and a linear region quadtree. Those access methods are based on classic external memory data structures, and their main target is reducing the number of disk accesses.

12.1.1 Join between linear quadtrees and R-trees

In [CVM99] the authors consider the problem of join operations between a linear region quadtree and a R-tree. The authors are based on a typical external-memory implementation of a linear region quadtree using a B^+-tree. A collection of simple and more sophisticated algorithms is presented, focused on reducing the number of accesses to external memory. The simplest algorithms are based on traversing one of the data structures sequentially and performing searches in the other. In a first proposal, called *B^+ to R join*, all the FD-codes in the B^+-tree tree leaves are traversed sequentially, and for each code a range search is performed in the R-tree to find all leaves that intersect with it. Two other simple algorithms, called *R to B^+ joins*, are based on the reverse process, in which the R-tree is traversed sequentially, and for the MBR of each entry found in the leaves of the R-tree a search is performed in the linear quadtree (the B^+-tree) to retrieve all the blocks that intersect with the MBR of the entry. In order to perform the search in the linear quadtree for each MBR the authors propose two strategies: a naive strategy looks for the SW corner of the MBR in the B^+-tree and then traverses sequentially the entries until the SE corner is surpassed, hence accessing probably many blocks that are outside the MBR; a second proposal is based on the decomposition of the MBR in maximal blocks that can be efficiently searched in the B^+-tree independently.

To provide more efficient algorithms for join operations, several heuristics are studied, always oriented to reducing the number of page accesses. A first strategy is based on processing the nodes of the R-tree following an order similar to the

ordering of the regions in the linear quadtree, keeping in main memory only the quadcodes that can actually intersect with the R-tree nodes currently evaluated.

The algorithm to perform the join operations works recursively on the R-tree. It starts at the root node, where all the entries are sorted according to the Z-order of their NW corner (the same ordering imposed by the linear quadtree). Each entry will be processed sequentially in this order. For each entry in the root node, a search in the linear quadtree is performed to retrieve a black block that covers the MBR or as many blocks as possible that intersect with the MBR of the entry. Then, all the children of the current node are traversed recursively and joined with the list of quadcodes: in each internal node, all its entries are again sorted following z-order and processed sequentially adding its children nodes to the list. When a leaf node is reached in the R-tree, all its entries are joined with the quadcodes loaded in main memory.

The join of an internal node returns when all its entries that intersect with the quadcodes loaded in memory have been processed. Eventually the root node will be reached again, and at this point all the entries descending from its first node that intersect with any of the quadcodes in memory have been processed. Then the list of quadcodes is emptied and reloaded with new quadcodes that intersect the entries in the current root entry. After no more quadcodes are found that intersect with the first root entry the process is repeated for the next root entry. To perform these operations, some buffers are necessary to store the list of quadcodes and to keep track of the positions that have been checked for each entry. A similar but improved algorithm is also proposed, that tries to expunge unnecessary quadcodes from memory as soon as possible to reduce even further I/O costs. We will omit the details on the actual implementation and techniques used, that as we said are focused on reducing page accesses. A detailed explanation of the algorithms can be found in the original paper.

The different algorithms proposed for join operations between a linear quadtree and an R-tree obtain significant reductions in the number of I/O accesses required in their experimental evaluation. An interesting conclusion is that the more sophisticated join algorithms proposed by the authors obtain very good results even using limited memory to cache accesses to the structures and quadcodes. However, the authors also show that allowing larger sizes of in-memory structures (mainly LRU caches for accesses in both structures) the performance of the simplest algorithms becomes comparable to the more sophisticated ones.

12.2 Our proposal

In this section we describe our method to perform a join between representations of vector and raster data. Our algorithm accesses an R-tree, that indexes all the minimum bounding rectangles (MBRs) from the vectorial dataset, and a K^2-tree1 that represents the raster dataset. The K^2-tree1 is used to store the complete

raster in main memory, efficiently distinguishing between regions covered by the raster field. The actual representation we will use in our experiments will be the K^2-tree1^{2bits}, but the algorithm is based on simple traversals that could be implemented in any of the variants and we will refer to a K^2-tree1 throughout this chapter. In the vectorial dataset we will consider only the MBRs of the objects, ignoring the additional space and processing time required to check the actual objects. This operation may be very expensive and is independent of the algorithm used to select the MBRs.

Our algorithm is designed in general to return all the MBRs that spatially intersect regions of the raster value the studied spatial field is active (regions of 1s in the binary raster). Our algorithm will return two different lists of MBRs: a list of *confirmed results* and a list of *possible results*. If an MBR is completely contained in a region of 1s, it will be a *confirmed* result. If the MBR intersects, but is not contained in, regions of 1s, it will be a *possible* result. Only the elements in the list of *possible results* have to go through the additional refinement step that checks the actual geometry of the object.

We define two data types for the algorithm:

- Type k $\equiv \langle$x,y,level,offset$\rangle$ nodes represent a K^2-tree1 node: its coordinates x and y, its *level* or depth in the conceptual tree and the *offset* in the bitmap where it is stored.

- Type r $\equiv \langle$MBR, ref, oid$\rangle$ nodes store R-tree nodes. Each tuple represents the MBR of the node, a list of references to children nodes and a list of object ids. Internal nodes of the R-tree will have a `null` value for the list of object ids (i.e. they will be $\langle$ MBR, ref, null $\rangle$) whereas leaf nodes will have a `null` value for the list of references ($\langle MBR, null, oid \rangle$). We also assume that two functions are available to check the type of any node of the R-tree (*isLeafNode()* and *isInternalNode()*).

Our main algorithm is also based on two auxiliary functions that operate on the K^2-tree1: *getMinMatrix* and *rangeSearch*. Both of them are straightforward adaptations of existing K^2-tree1 navigation operations:

- The function *getMinMatrix* receives as input an MBR and a reference to a node *kCur* in the K^2-tree1 and returns a pair $\langle$k,type$\rangle$. The value k is the deepest node descending from *kCur* whose submatrix *completely contains* the given MBR, while *type* describes the "color" of the node: the returned value is 0 for white nodes (regions of 0s), 1 for black nodes (regions of 1s) and 2 for gray nodes (mixed 0s and 1s). Since the MBR is translated into a rectangular region in the raster, *getMinMatrix* simply needs to traverse the K^2-tree1 down until we find a leaf node or the lowest node containing this rectangle. In the worst case, the cost of this operation is proportional to the height of the K^2-tree1.

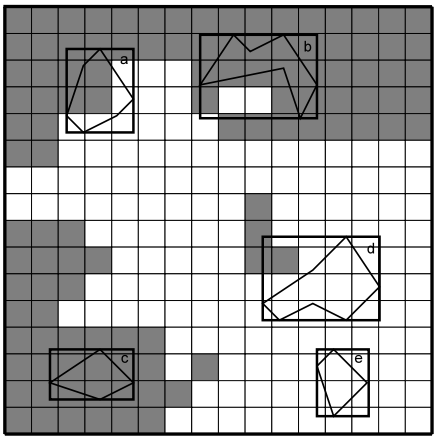


Figure 12.1: A raster dataset and four spatial objects with their MBRs.

- The function *rangeSearch* takes an MBR and a reference of a node $kCur$ in the K^2-tree1 and performs a range search on the K^2-tree1 using the MBR. The function returns the type or "color" of the region, with a similar meaning to the result of *getMinMatrix*:
  - If all cells that intersect the MBR are 0s, then type = 0. An example would be object e in Figure 12.1.
  - If all cells that intersect the MBR are 1s, then type = 1. An example would be object c in Figure 12.1.
  - If the MBR intersects 0s and 1s, then type = 2. Examples of this case would be objects a, b and d in Figure 12.1.

Algorithm 12.1 shows the complete algorithm. We start by defining the lists of confirmed (C) and possible (P) results, and a stack that will be used to keep track of the pairs of nodes that still have to be processed. Then, we set two variables $rCur$ and $kCur$ that point to the root of the R-tree and the K^2-tree1 respectively. This pair of variables is added to the stack S.

After this initialization, the algorithm repeatedly extracts and processes entries from the stack until the stack is emptied. For each $\langle rCur, kCur \rangle$ pair extracted from the stack, we use *getMinMatrix* (line 10) to find the deepest node of the K^2-tree1 that contains the MBR of $rCur$. We will continue depending on the type of the K^2-tree1 node:

- If the K^2-tree1 node is a white node (type = 0), the R-tree node and all its descendants fall within a region not covered by the raster, so we do nothing and advance to the next iteration of the algorithm. Notice that this case is omitted in the algorithm, because nothing is done.

Algorithm 12.1 R-tree × K^2-tree1 join.

```
 1: Let R be an R-tree
 2: Let K be a K^2-tree1
 3: Let C and P be lists with objects of type r
 4: Let S be a stack with objects of type ⟨r,k⟩
 5: rCur ← ⟨MBR(R), R, null⟩
 6: kCur ← ⟨0,0,0,0⟩
 7: push(S, ⟨rCur,kCur⟩)
 8: while S ≠ empty do
 9:     ⟨rCur,kCur⟩ ← pop(S)
10:     ⟨kNew,type⟩ ← getMinMatrix(rCur.MBR, kCur)
11:     if type = 1 then
12:         if isLeafNode(rCur) then
13:             add(rCur.oid, C)
14:         else
15:             addSubTree(rCur.ref, C)
16:         end if
17:     else if type = 2 then
18:         if isInternalNode(rCur) then
19:             for rNew ∈ rCur.ref do
20:                 push(S, ⟨rNew,kNew⟩)
21:             end for
22:         else
23:             type ← rangeSearch(rCur.MBR, kNew)
24:             if type = 1 then
25:                 add(rCur.oid, C)
26:             else if type = 2 then
27:                 add(rCur.oid, P)
28:             end if
29:         end if
30:     end if
31: end while
32: return ⟨C,P⟩
```

- If the K^2-tree1 node is black (type = 1, line 11), the R-tree node and its descendants are covered by the raster. We add the current object (line 13) or the complete R-tree branch (line 15) to the confirmed list.

- If the K^2-tree1 node corresponds to a mixed region (type = 2, line 17) we cannot yet determine whether the intersection is true. In this case, if the R-tree node is internal node we must explore all its descendants (we add all its children to the stack in line 19 and go to the next iteration of the algorithm). If the R-tree node is a leaf, we use function *rangeSearch* to determine if there is an intersection between the object in the R-tree node and the raster (line 23).

Again, our algorithm depends on the *type* returned by *rangeSearch*:

- If type is 0 (region of 0s), we discard the object since it cannot intersect the raster (e.g. object e in Figure 12.1). Again, this case is not shown in the algorithm since nothing is done
- If type is 1 we are in the case of object c in Figure 12.1: it can be added to the confirmed list since it is completely contained in a region of 1s (line 25).
- If type is 2 we are in the case of objects a, b or d in Figure 12.1: the object intersects 0s and 1s and we must use the actual geometry of the object to confirm the result, therefore we add the object to the list of possible results (line 27).

As we have shown, when our algorithm finishes it returns a list of *confirmed results* and a list of *possible results*. An additional step would be necessary for each object in the list of possible results, retrieving its geometry and checking if the result is finally confirmed (this would be the case of objects a and b in Figure 12.1) or if it was a false positive (e.g. object d in Figure 12.1). This additional check, that will not be covered here, is common in spatial access methods that require two-step algorithms composed of a filter step and a refinement step that removes false positives from the result.

12.3 Join between a general raster dataset and a vectorial dataset

We have just described our join algorithm as a solution to perform a join between a binary raster dataset and a vector dataset. Nevertheless, our algorithm can be extended to the case where the raster dataset is not a binary coverage but a general raster instead. In this case, the type of queries of interest would involve a query on the raster dataset, typically a query involving the values stored in it. An example of such a query would be retrieving all the objects in the vectorial dataset that occur in regions with values over or below a threshold, or in regions with an exact value or small range of values. As we have shown in Chapter 10, this kind of queries can be efficiently solved using different K^2-tree1 variants in small space and providing good query times.

Consider now a straightforward generalization of the types described for K^2-tree1 nodes in our algorithm: a node is type 0 if none of its cells fulfill a query condition (originally, be set to 1), type 1 if all the cells fulfill the condition, or type 2 if some cells fulfill the condition but others do not. This simple adaptation allows our algorithm to work with any representation that is able to compute the node type given a query, that is, as long as *getMinMatrix* and *rangeSearch* can work with a general raster. All the representations used in Chapter 10 could be used

to represent a general raster dataset and perform join operations with a vectorial dataset, where the raster dataset could be filtered to search for cells with a given value or any range of values.

Consider for example a general raster and a vectorial dataset, and a join operation asking for all the objects that intersect with regions of the raster with values in $[v_1, v_2]$. The algorithm does not change, only the implementations of *getMinMatrix* and *rangeSearch* do. Consider, without loss of generality, that we are using the AMK^2-tree1 representation for the raster (we can use any of the proposals for general raster: MK^2-tree1, AMK^2-tree1, IK^2-tree or K^3-tree). For each R-tree node, we look for the deepest node in the raster representation that contains its MBR. This traversal is actually performed as a range search in the AMK^2-tree1, traversing simultaneously the K^2-tree1s of v_2 and $v_1 - 1$ to determine the "type" of the node for the condition of the join (i.e. the function *getMinMatrix* would return 0 when there are no values in the raster falling in the range $[v_1, v_2]$, 1 when all the values in the raster fall in the range, and 2 otherwise). When the lowest node containing the MBR is found, the AMK^2-tree1 already has computed its type depending on the range of values. The *getMinMatrix* auxiliary function should then return the "node" (in our case a pair of nodes in K_{v_2} and K_{v_1-1}) and its type. This change, combined with additional information stored in the stack about the nodes in the raster representation, suffices to adjust our algorithm to work with general rasters.

The previous generalization increases the complexity of the *getMinMatrix* operation, because we must do more complex operations to know the actual *color* of the node. In practice, we use a simple algorithm over the AMK^2-tree1 that uses the original *getMinMatrix* algorithm and additional tables to determine the next step depending on the types of the nodes in K_{v_2} and K_{v_1-1}, as shown in Table 12.1. The algorithm acts depending on this virtual "new type", that is not necessarily equal to the type obtained in the synchronized traversal, but computation becomes much easier. In particular, if type1 and type2 are 2, the result can be either 2 (mixed) or 0 (region of 0s), depending on whether K_{v_2} contains any 1 not contained in K_{v_1-1}, but to compute this we must traverse both subtrees completely.

*type*1/*type*2	0	1	2
0	newType=0	newType=1	newType=2
1	Impossible	newType=0	newType=2
2	Impossible	newType=2	newType=2

Table 12.1: Action table for *getMinMatrix*.

Notice however that the *rangeSearch* must still perform the complete computation to know the actual type of the node, otherwise we would not be able to distinguish some results in the general algorithm (in the previous example, and in

Table 12.1, when type1=2 and type2=2 we do not actually know if the result is a gray node or a white node).

An additional observation related with the specific implementation using an AMK^2-tree1 is the fact that queries asking for regions with values above or below a threshold (threshold queries) can be answered more efficiently using the same basic algorithm explained for binary rasters. Since each K^2-tree1 in the AMK^2-tree1 contains cells with values below a given one, this query is answered querying directly the appropriate K^2-tree1: when asking for cells with value above a threshold, we query the complement of the K^2-tree1, which can be easily performed with a special implementation of *getMinMatrix* and *rangeSearch* that returns 0 for type-1 nodes and 1 for type-0 nodes.

12.4 Experimental evaluation

In order to evaluate the performance of the algorithms described in Sections 12.2 and 12.3, we perform several of experiments that use real and synthetic data. We test our algorithm with binary and general raster datasets to demonstrate its query efficiency. First we will test the efficiency of the algorithm showing its ability to filter accesses to the R-tree in a join operation with a binary raster coverage. Then, we will test the extension of our algorithm to query a general raster and an R-tree. We implemented our generalized algorithm using the AMK^2-tree1 representation, that uses a collection of "accumulated" K^2-tree1s to represent the raster dataset.

12.4.1 Experimental setup

Dataset	Size (cells)	%1s	K^2-tree1 size (bytes)
ras1024_20	1024 × 1024	20	2,448
ras1024_60	1024 × 1024	60	3,692
ras2048_20	2048 × 2028	20	6,548
ras2048_60	2048 × 2048	60	8,632
ras4096_20	4096 × 4096	20	23,728
ras4096_60	4096 × 4096	60	33,980

Table 12.2: Binary raster coverages used in the experiments.

Table 12.2 describes the six binary raster coverages used in the experiments. For each raster the table describes its size in pixels, the percentage of 1s in the coverage, and the size of the corresponding K^2-tree1. All the datasets are square subsets of different sizes extracted from the elevation raster *mdt–A* first

Name	Size (cells)	# diff. values
ras1024	1024 × 1024	209
ras2048	2048 × 2048	413
ras4096	4096 × 4096	704
ras8192	8192 × 8192	973
ras16384	16384 × 16384	1,716

Table 12.3: General raster coverages used in the experiments.

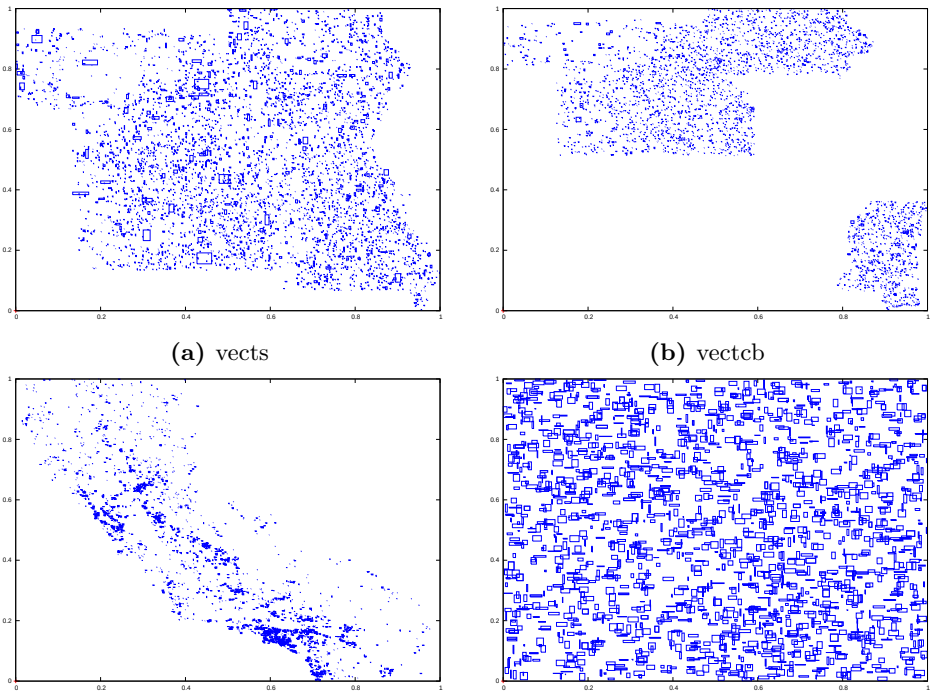
introduced in Chapter 5. We build different binary rasters from the original dataset considering different percentages of black pixels simply varying a threshold on the elevation value used to distinguish black from white positions. Similarly, Table 12.3 describes general rasters used in the second set of experiments. These rasters are fragments from *mdt–B*, a large elevation raster also introduced in Chapter 5. Finally, Table 12.4 describes the different sets of MBRs used as vectorial datasets. For each MBR dataset, we describe whether the dataset is real or synthetic, the number of MBRs it contains and the number of nodes in the resulting R-tree.

Name	Type	# MBRs	*R*-tree size (nodes)
vects	Real	194,971	3,130
vectcb	Real	556,696	9,147
vecca	Real	2,249,727	33,754
vec2m	Synthetic	2,000,000	28,267
vec5m	Synthetic	5,000,000	70,587
vec10m	Synthetic	10,000,000	140,628
vec20m	Synthetic	20,000,000	280,705

Table 12.4: MBR datasets used in the experiments.

To make the implementation of the experiments easier, all coordinates of the MBRs and the rasters are normalized to a $[0, 1] \times [0, 1]$ space. Figure 12.2 shows the distribution of the MBRS in the different real datasets using a representative fraction of the MBRs. The three real datasets, denoted by vects (see Figure 12.2a), vectcb (see Figure 12.2b), and vecca (see Figure 12.2c), are the Tiger Streams (vects), Tiger Census Blocks (vectcb), and California Roads (vecca) datasets from the ChoroChronos.org web site[1]. The synthetic datasets consist of uniformly distributed MBRs with an area between 0.001 and 0.03 square units. Figure 12.2d shows also one of the synthetic datasets.

[1] `http://www.chorochronos.org/?q=node/59`



(a) vects **(b)** vectcb

(c) vecca **(d)** vec2m

Figure 12.2: MBR distributions in the space $[0,1] \times [0,1]$.

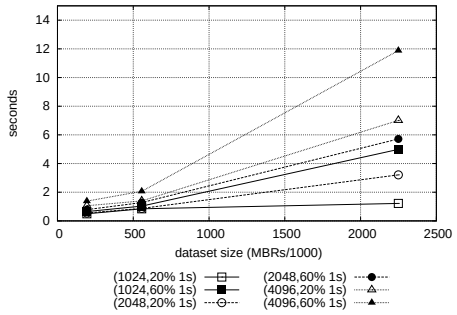
We use Marios Hadjieleftheriou's implementation of the R-tree[2], written in java. The R-tree nodes are of 4KB and have a capacity of 100 entries. To run our tests we use a java implementation of the K^2-tree1, that as we said is based on the K^2-tree1^{2bits} variant. The AMK^2-tree1 representation used for general rasters is also based on the same java implementation of K^2-tree1. In order to provide a pure version of the algorithm, we do not include any of the enhancements to the K^2-tree1 variants used in other chapters to improve compression (there is no compression of submatrices in lower levels of the tree and we use a single value of $K = 2$). This change leads to worse compression results than those obtained in previous chapters but provides a clean basis to test the efficiency of the basic algorithm.

All the tests in this chapter are run in a machine with 4 Intel(R) Xeon(R) E5520 CPU cores at 2.27 GHz 8 MB cache and 72 GB of RAM memory. The operating system is Ubuntu 9.10.

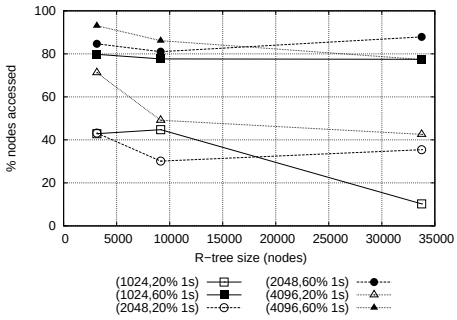
[2] `http://libspatialindex.github.com/`

12.4.2 Experimental results with binary coverages

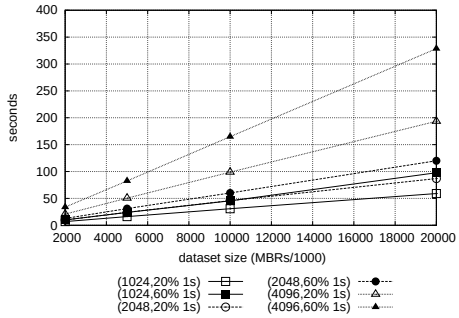
In Figure 12.3 we show the experimental results obtained with the real and synthetic vectorial datasets. Figures 12.3a and 12.3c show that the algorithm requires approximately linear time on the number of MBRs in the vectorial dataset in both real and synthetic datasets. Figures 12.3b and 12.3d show the percentage of the total R-tree nodes that are accessed by the join algorithm. The percentage of nodes accessed clearly depends on the rate of 1s in the raster dataset: for raster datasets with 20% 1s the percentage of R-tree nodes accessed never surpasses 80%, and can be as low as 10%; on the other hand, for raster datasets with 60% 1s the percentage of R-tree nodes accessed ranges from 80-95%. There is also a significant difference between the results in real datasets, shown in Figure 12.3b, and in synthetic datasets, shown in Figure 12.3d: the percentage is much higher and less variable in synthetic datasets, that are built with an uniform distribution; however, in real datasets the locality of the data allows the algorithm to access a lower percentage of the total nodes.



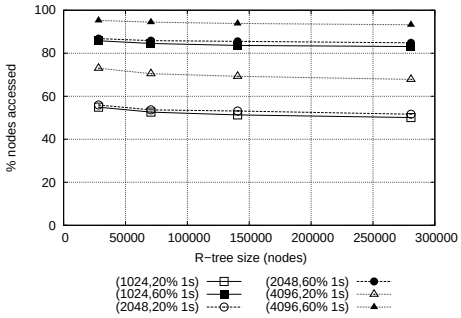
(a) Query times in real datasets



(b) % nodes accessed in real datasets



(c) Query times in synthetic datasets



(d) % nodes accessed in synthetic datasets

Figure 12.3: Processing time (in seconds) and % of R-tree nodes accessed using real (top) and synthetic (bottom) MBR datasets.

Datasets		Number of results		%
Vectorial	Raster	Total	Confirmed	
vects	ras1024_20	27,985	26,483	94.6
	ras1024_60	106,010	103,388	97,5
	ras2048_20	29,480	27,575	93,5
	ras2048_60	115,466	112,366	97,3
	ras4096_20	46,801	42,309	90,1
	ras4096_60	131,707	126,693	96,2
vectcb	ras1024_20	182,852	180,630	98,7
	ras1024_60	375,089	372,831	99,4
	ras2048_20	83,509	80,839	96,8
	ras2048_60	375,315	371,435	98,9
	ras4096_20	123,523	118,560	96,0
	ras4096_60	382,373	375,737	98,2
vecca	ras1024_20	65,622	63,185	96,3
	ras1024_60	1,369,044	1,319,018	96,3
	ras2048_20	516,917	497,309	96,2
	ras2048_60	1,684,579	1,642,619	97,5
	ras4096_20	444,533	407,875	91,7
	ras4096_60	1,350,981	1,296,004	95,9

Table 12.5: Results retrieved by the algorithm using real MBR datasets.

The percentage on R-tree nodes accessed by our algorithm is very high in some cases, especially in synthetic datasets. In order to measure the actual efficiency of the algorithm we show in Tables 12.5 and 12.6 the total results obtained by our algorithm in real and synthetic datasets and the percentage of results that are *confirmed* results (i.e. results that do not require a second comparison between the actual spatial object geometry and the raster dataset to confirm or discard the result). This percentage is above 90% in real datasets, therefore our algorithm can save disk accesses and also much of the costly computation of intersecting the actual geometries of the results with the raster. In Table 12.6 we show the same results for two of the synthetic datasets (the percentages are almost identical in all the synthetic datasets). As expected, in this case the percentages are lower, but we still obtain around 70% of confirmed results.

12.4.3 Experimental results with general raster data

Next we test the extension of our algorithm to deal with general raster data using the AMK^2-tree1 representation. We compare our representation with a simpler approach based on a direct representation of the raster dataset using a matrix of

Datasets		Number of results		%
Vectorial	Raster	Total	Confirmed	
vec2m	ras1024_20	445,161	324,163	72.8
	ras1024_60	1,287,176	1,098,651	85.3
	ras2048_20	485,101	322,654	66.5
	ras2048_60	1,316,354	1,097,920	83.4
	ras4096_20	545,345	261,415	47.9
	ras4096_60	1,411,308	1,004,285	71.1
vec20m	ras1024_20	4,454,095	3,242,037	72.7
	ras1024_60	12,861,243	10,974,984	85.3
	ras2048_20	4,853,557	3,231,135	66.5
	ras2048_60	13,167,282	10,977,876	83.3
	ras4096_20	5,452,352	2,617,038	47.9
	ras4096_60	14,111,225	10,031,539	71.0

Table 12.6: Results retrieved by the algorithm using synthetic MBR datasets.

values. This alternative representation also uses a simpler algorithm to answer the queries: it traverses all the nodes in the R-tree and checks if they fulfill the condition in the raster dataset simply accessing the matrix. In our experiments we use a simple matrix representation where each value is stored using an integer for maximum efficiency in access.

To measure the quality of our algorithms, we will focus on space utilization and query efficiency. We distinguish 2 main groups of queries according to the desired result: *strong* queries that only answer MBRs that completely fulfill the condition (all their points are in the interval or above/below the threshold) and *weak* queries that answer all MBRs that partially fulfill the condition (some points in the MBR fall in the interval). We will also distinguish two different queries according to the filter involved in the query: *interval queries*, that ask for the MBRs in the R-tree that intersect with regions of the raster whose values are in an interval, and two different *threshold queries*, that involve regions of the raster above/below a threshold. Even though threshold queries can be reduced to interval queries, we consider this particular case because it is a frequent query and, as we have explained, our algorithm can be optimized in this kind of queries to access a single K^2-tree1.

12.4.3.1 Space results

Table 12.7 shows the space utilization of the complete AMK^2-tree1 representation (a K^2-tree1 per value in the raster) and the actual space requirements of the matrix representation using an integer per cell. In the last column *matrix-optimum* we

Name	AMK^2-tree1	matrix	matrix-optimum
ras1024	0.31	4	0.96
ras2048	1.53	16	4.34
ras4096	5.77	64	18.92
ras8192	25.61	256	79.41
ras16384	148.51	1,024	343.83

Table 12.7: Space requirements (MB) of AMK^2-tree1 and a matrix representation for all the raster datasets.

show the minimum size that could be achieved by a simple matrix representation, using $\log(\#values)$ bits per element. Our complete representation is clearly much smaller than an uncompressed matrix representation, roughly 10 times smaller. Notice also that some optimizations used in Chapter 10 such as the use of a matrix vocabulary are not used in this chapter. Additionally, in practice our representation only needs to have 2 K^2-tree1s in main memory to answer interval queries, while the naive representation may need to traverse the complete raster in most queries. However, we consider the memory requirements of the complete AMK^2-tree1 to answer queries without accessing external memory.

12.4.3.2 Query times

Threshold queries

First we compare our algorithm with the uncompressed raster representation (*Raster*) to answer threshold queries. Following the same steps of the previous experimentation with binary rasters, we study the evolution of query times with the length of the query interval in the different raster and vectorial datasets. In Figure 12.4 we show the results for the all the vectorial datasets studied. To compare the results obtained for rasters of different sizes, we show in Figure 12.4a the results for all queries using the ras1024 dataset and in Figure 12.4b the results for the largest raster dataset ras16384. The results for the intermediate raster datasets are similar.

Query times differ significantly depending on the vectorial dataset, mainly due to the difference in size between them. On the other hand, the cost of the queries is not very different depending on the semantics (*strong* queries where all the points must be included or *weak* queries where only some of the points need to be included) or the type of threshold used (asking for values above or below a threshold).

In the vects dataset our query times are similar to the query algorithm using an uncompressed raster, and we even obtain worse query times in some cases. The characteristics of the vects dataset are the reason for this result: the MBRs are more uniformly distributed than in the other real vector datasets, as we

can see in Figure 12.2. This fact, combined with the small size of the dataset, causes that the reduction in terms of number of accesses to R-tree nodes is not enough to compensate the added complexity of our algorithm. The results in the larger vectorial datasets, independently of the raster dataset used, demonstrate the efficiency of our algorithm, that is always faster than the baseline in the vectcb and vecca datasets, and especially in the vecca dataset where our query times are 2.5 times faster than the baseline.

Interval queries

In Figure 12.5 we analyze the evolution of query times in interval queries in the two largest vectorial datasets for different raster dataset sizes. The three left plots show the results for the vectcb dataset and the rightmost 3 plots the results for the vecca dataset. We show a selection of results for the raster datasets ras1024, ras4096 and ras16384 (top, middle and bottom plots respectively, for each vectorial dataset).

Our results show that, like in the binary case, the efficiency of our algorithm depends on the selectivity of the query and the characteristics of the datasets studied. While the baseline algorithm obtains consistent query times, that change only slightly in less selective queries due to the additional number of results processed, our algorithm takes advantage of the characteristics of the query to speed up queries that are very selective in their intervals. In particular, Figure 12.5 shows that our algorithm obtains its best results in general when the query interval is very small, thus making the query very selective since few elements will belong to the interval. In this case, our algorithm is able to efficiently filter out a great fraction of the MBRs and reduce significantly the computation costs, while the baseline algorithm still must pay the cost of traversing the complete vectorial dataset. Our results also show that when asking with low selectivity in the raster (the query interval is close to 100% of the values), our algorithm also obtains good results in general, because it can filter out the MBRs not contained in the raster very early and then the computations in the AMK^2-tree1 are very fast because the K^2-tree1 involved are small (since few values are below the query interval or above it).

The results in Figure 12.5 show that, for almost all the vectorial and raster datasets, our algorithm obtains better query times than the baseline, especially in very selective queries. Results for the other vectorial dataset and the remaining raster datasets are similar, with our algorithm consistently obtaining better query times. A particular case occurs in the bottom-left plot of Figure 12.5, corresponding to the vectorial dataset vectcb and the raster dataset ras16384: in this case, our algorithm becomes a bit slower than the baseline in some cases (similar results were obtained for the vects dataset with the ras16384 raster, while in all the remaining combinations our algorithm was always faster). Similarly to the results obtained in threshold queries, the smaller size of the datasets and the fact that our algorithm relies on the distribution of both datasets cause this particular negative result: if the distribution of the MBRs in the vectorial dataset and the regions in the raster

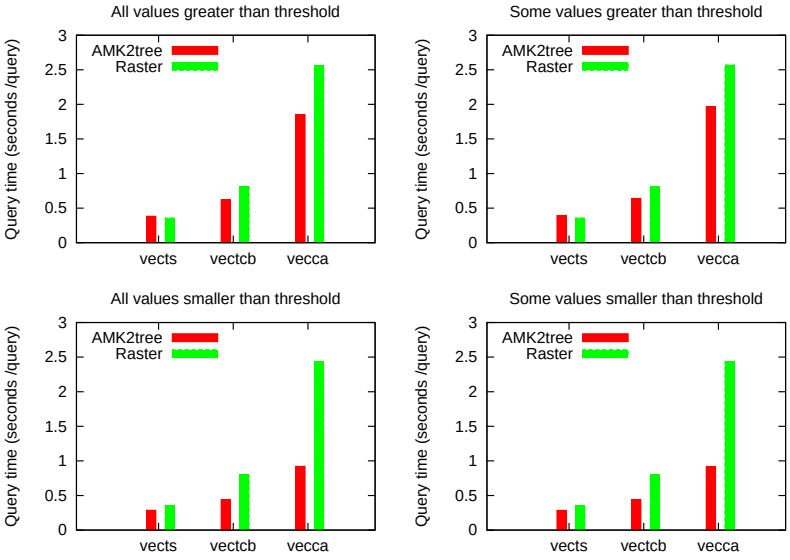
dataset is such that we cannot filter a significant fraction of the total MBRs in our queries, our specific algorithm may become slower than the simpler baseline that simply traverses. Nevertheless, our algorithm is in general much faster than the baseline in interval queries, and especially in selective interval queries where our query times are up to 2.5 times faster.

12.5 Summary

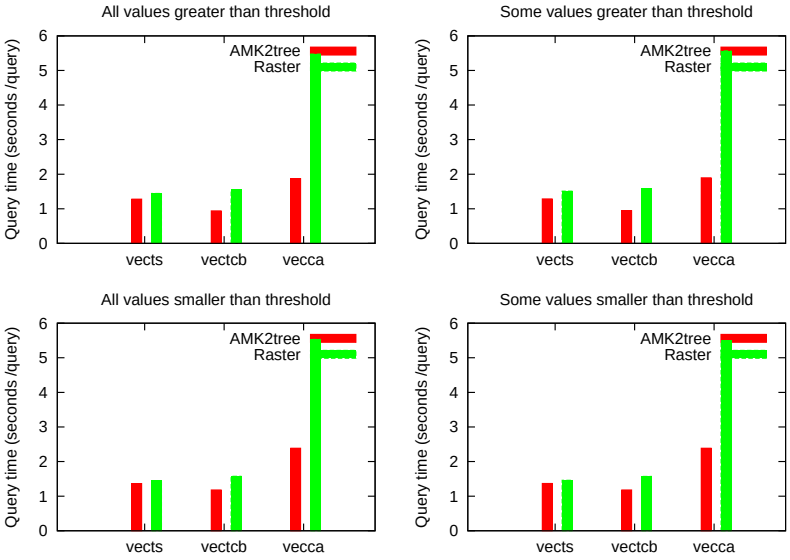
In this chapter we have presented an algorithm to jointly query vectorial and raster datasets, when the vectorial datasets are stored using an R-tree and the raster datasets are stored using the K^2-tree1 variants introduced earlier in this thesis. First we described an algorithm to join a K^2-tree1, introduced in Chapter 5, with an R-tree. Our algorithm is designed to avoid most of the accesses to R-tree nodes. Then, we extended our algorithm to work with the AMK^2-tree1 representation of general rasters, proposed in Chapter 10.

We experimentally evaluated our algorithm to show its efficiency with real and synthetic datasets. We tested its query efficiency in terms of time and number of accesses to R-tree nodes required, showing that it was able to reduce the number of accesses to R-tree nodes significantly. In addition, most of the results returned by the algorithm are confirmed results whose geometries do not have to be checked later.

We also tested our algorithm with general raster datasets, comparing it with a simpler approach based on a sequential algorithm and an uncompressed raster representation. Our experiments show that our representation is not only smaller, as expected, but also faster than the alternative to answer interval and threshold queries in most of the vectorial and raster datasets studied. Our solution is particularly effective in very selective queries, obtaining query times up to 2.5 times faster than the baseline. These results are relevant taking into account the fact that we are comparing our representation with an uncompressed in-memory representation of the raster.



(a) Query times with ras1024 dataset



(b) Query times with ras16384 dataset

Figure 12.4: Query times of threshold queries joining all vectorial datasets with ras1024 (top) and ras16384 (bottom) raster datasets.

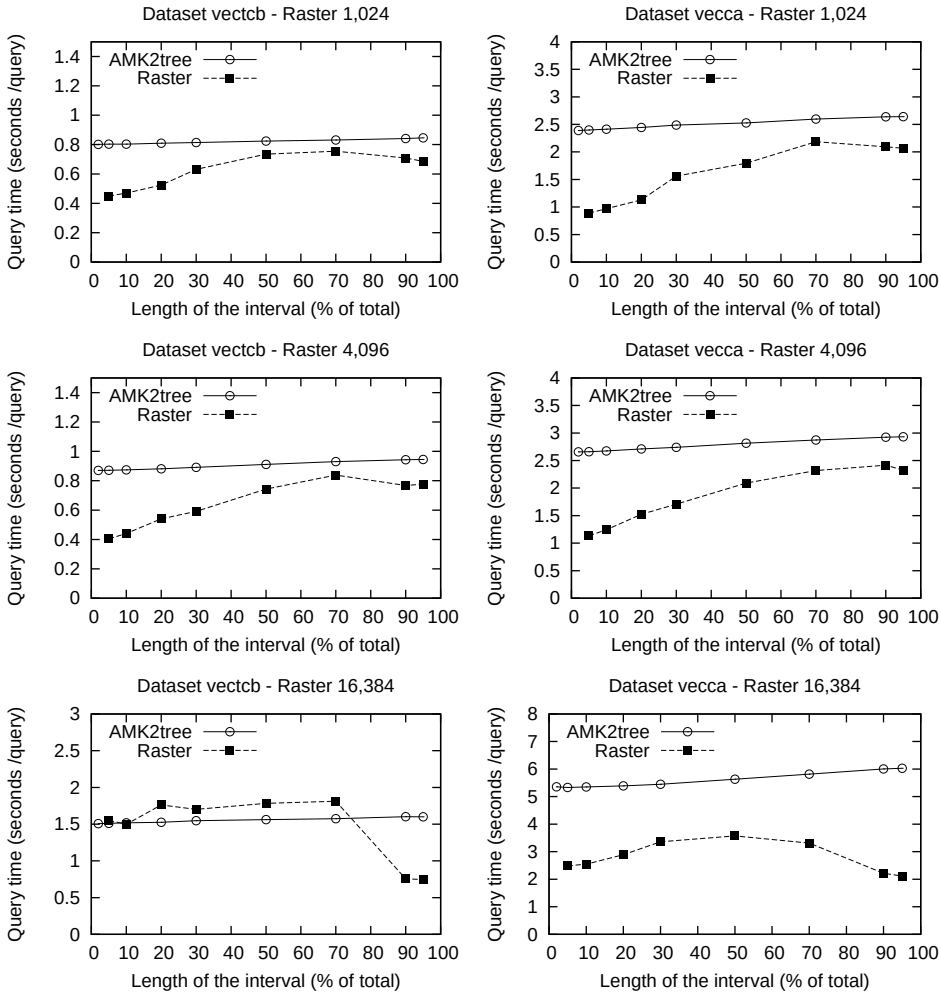


Figure 12.5: Query times of interval queries with different interval length, in with some of the studied vectorial and raster datasets.

Part IV

Summary of the thesis

Chapter 13

Conclusions and future work

13.1 Main contributions

The amount of information stored and processed in digital form has been increasing exponentially for several decades. Nowadays, huge amounts of information are obtained in real time from multiple sources and must be efficiently processed and accessed. Because of this, processing efficiently the increasingly large amounts of information available has become a very important challenge in different areas.

Several strategies are followed to process the large amounts of information available, from the parallel processing of data to the creation of specific hardware architectures designed for specific goals. One of the most fruitful research lines has been the development of compact representations that are able to reduce the space used by the data. These representations allow us to process the data in upper levels of the memory hierarchy, and therefore to process it faster thanks to the speed gap between levels. The development of compact or succinct data structures, that are able to not only store the data but also answer queries in compressed form, has received a lot of attention.

In this thesis we focus on the compact representation of bi-dimensional or n-dimensional data. Our goal is to provide efficient representations for grids that can be easily applied to the representation of binary or n-ary relations, graphs, etc. We are also particularly interested in domains where the data has a "spatial" distribution (e.g. geographic data). We study a set of related problems, associated with the compact representation of grids in different domains.

The main contributions of this thesis are a collection of compact data structures with applications in different contexts. We summarize them in this section. First, we have proposed several general data structures with general application in different domains:

- We presented a collection of representations, called K^2-trees with compression

of ones or K^2-tree1. These representations are an extension of an existing data structure, the K^2-tree, to efficiently represent clustered binary grids with large regions of 1s and 0s. We provide a small set of different encodings that extend classic K^2-trees improving very significantly their compression capabilities in clustered grids. We theoretically and experimentally compare our representation with state-of-the-art representations of binary images, showing that our representations provide efficient query support and achieve compression results very competitive. These representations are also used to develop variants of other proposals studied in this thesis.

- We introduced a new data structure for the compact representation of n-ary relations, called K^n-tree. This representation is a generalization of the K^2-tree to higher dimensionality problems. We study its applicability and its drawbacks in the general case.

- We proposed a new compact data structure to answer top-k queries in multidimensional grids, called K^2-treap.

- We developed a new dynamic representation of grids, called dynamic K^2-tree or dK^2-tree. This representation is able to answer all queries supported by static K^2-trees and supports update operations on the grid, including changing the contents of a cell and the insertion of new rows/columns.

In addition to the proposal of the new data structures, that can be applied in different domains, we have focused on the applications of the new data structures proposed and created new variants of some of them to answer specific problems in each application area:

- We studied the representation of OLAP cubes, and particularly the problem of top-k queries, as a particular query very often required in multidimensional databases. We experimentally evaluated our proposal, the K^2-treap, in comparison with state-of-the-art alternatives based on the wavelet tree. Our experiments show that our representation is smaller and faster than the alternatives in a large majority of cases.

- We also studied the representation of temporal graphs. In this area we have developed a new data structure, called differential IK^2-tree or diff-IK^2-tree, a variant of an existing data structure called IK^2-tree that is able to store time-evolving graphs in very reduced space. We have also shown the applicability of a K^3-tree to the representation of temporal graphs, considering the time as another dimension. Our experimental evaluation shows that our proposals are complementary alternatives for the representation of time-evolving data. In Chapter 9 we also show that a fully-dynamic representation of a temporal graph can be built using our dynamic K^2-tree representation.

- We studied the compact representation of RDF databases, a problem that had already been tackled using K^2-trees [ÁGBF$^+$14] but was limited to a static context. In Part II we make use of the dynamic representation proposed to provide a dynamic alternative that supports the same queries. Our experimental evaluation focuses on the comparison with the static version, that was a very competitive alternative with state-of-the-art proposals. Our results show that the overhead required by our dynamic representation is small in most cases.

- Finally, in the third part of the thesis we study the representation of raster data using the compact data structures developed in the thesis. We study three different problems and provide specific solutions for each of them:

  - We propose several alternatives for the representation of general raster data, based on the different proposals presented in the thesis. All our proposals answer the most important queries in the domain, including queries restricted to a spatial window or to cells with a given value/range of values. We design specific variants of the general data structures for this problem: the MK^2-tree1 is a simple representation based on multiple K^2-tree1 with specific set operations; the AMK^2-tree1 is a variant specific to this problem that optimizes queries involving ranges of values; the K^3-tree is applied directly, with specific algorithms; the IK^2-tree1, developed specifically for this problem, is a combination of the IK^2-tree with the K^2-tree1 to provide better indexing properties. All our representations have strengths in particular queries in this domain. Additionally, we also propose two variants of the K^2-treap especially adapted to answer top-k range queries in raster data.
  - We apply our proposals to the representation of time-evolving region data, or moving regions. The same proposals used for general rasters are applied to the representation of temporal rasters, with specific algorithms and adjustments. Our representations can efficiently retrieve the state of the raster or a spatial window in a given time instant or during a specific time interval. As in the previous case, our proposals provide alternatives for the representation, allowing a space/time tradeoff depending on the characteristics of the dataset and the most relevant queries.
  - We study the problem of querying raster and vectorial data together, or join operations between raster and vectorial data. We provide new algorithms to perform join operations between binary or general raster datasets, represented using our variants, and vectorial datasets represented using classic data structures. Our algorithms are simple to implement and can work with the complete raster stored in main memory thanks to the compact representation used. In addition to being smaller, our solutions are also faster than simple representations that use an

uncompressed representation of the raster even when the uncompressed representation is stored completely in main memory.

13.2 Future work

In this section we devise some of the future work planned after this thesis. Due to the characteristics of the work, with many small contributions and applications, part of the main work has already been introduced in the summary of previous chapters. In this section we summarize the most important lines of future work:

- A field with important challenges for our structures is the representation of datasets with high dimensionality. We have shown that the K^2-tree can be extended to 3-dimensional problems and still provides a compact representation in domains such as the representation of temporal graphs. However, the partitioning algorithm and the need to store always groups of K^n siblings even if few of them are active limits the application of the K^n-tree as the number of dimensions is above 3 or 4. The problem of the arity of the conceptual tree has been reduced with techniques based on our encodings, such as the compression of unary paths, that aim to efficiently reduce the space requirements in very sparse datasets, but we believe that more general solutions would be interesting to guarantee the efficiency of our proposal.

- The K^2-treap has been proved to be a very efficient solution to answer top-k range queries in OLAP data and also in raster data. We believe that the same philosophy of using a K^2-tree-like data structure enhanced with summary information in its nodes, that is also applied in the ttK^2-tree proposed in Chapter 9, can lead to new and interesting applications in other domains. Therefore, we plan to design and test similar variants where K^2-tree nodes include other types of relevant information about the covered submatrix.

- We plan to study the improvement of our dynamic representation dK^2-tree with specific additions depending on the domain. Particularly, the representation of RDF databases in a static context can be improved with additional static indexes that significantly reduce query times in some operations. However, the creation of dynamic versions of these indexes is problematic due to the space requirements: while the static index provides a reasonable increase in size, a dynamic index may require several times the size of the static index to be used. A similar problem occurs in the case of RDF with the management of a dynamic dictionary of terms, that associates each subject, predicate and object with its id; this problem is satisfactorily solved in a static context but an efficient fully-dynamic in-memory dictionary is more challenging.

- The dK^2-tree representation is based on a B-ary tree that could easily be stored in external memory. This would allow our representation to work in domains where the relations are too big to be stored completely in main memory even in compressed form. Our representation can be moved to external memory with minimum changes, as all the information is blocked and the *rank* information for each block is stored in the block itself. However, the minimum access of a block per level of the conceptual tree seems like a big drawback in terms of efficiency. We claim that our representation, even for huge datasets, could be stored almost entirely in main memory. In this case, a simple caching of blocks would allow us to use it when stored partially in secondary memory, as only a reduced number of blocks would have to be actually retrieved from external memory during queries. Nevertheless, we plan to explore alternative representations of the blocks that allow a more efficient retrieval of the tree in terms of I/O costs.

- Regarding the representation of spatio-temporal raster data, a challenge that remains as future work is the possibility of creating *differential* snapshots of the time-evolving region data. This would go in line with the diff-IK^2-tree proposal for temporal graphs introduced in Chapter 6, where differential snapshots were used to reduce very significantly the space utilization at the cost of increasing query times. However, the simple creation of differential snapshots in this domain does not provide a better compression, because compression of each snapshot depends heavily in the existence of large clusters of 1s or 0s. In Chapter 10, the comparison between the MK^2-tree1 and the AMK^2-tree1 showed that the reduction in the number of 1s to store in the dataset does not necessarily imply a reduction in the space utilization in this domain. However, the development of more efficient and compact differential representations is still an interesting line of improvement, following the steps of the OLQ and its family of spatio-temporal data structures but providing a much more succinct representation of the data that can be navigated efficiently like our K^2-tree variants.

- Regarding the join algorithm with vectorial representations, we have proposed our algorithm to join with the R-tree because it is probably the most used representation for this kind of data. However, there are also compact data structures for vectorial data (e.g., [BLNS13]). We believe new algorithms that are able to work with a compressed representation also for vectorial data could provide an interesting alternative for current approaches based on traditional data structures.

- Finally, another future goal that covers all the thesis is the creation and publication of a unified framework containing all our proposals and K^2-tree variants, so that they can be efficiently combined and used in any domain. In this thesis we have combined the encodings developed in Chapter 5 with

other data structures, like the K^n-tree or the IK^2-tree, and we also used them to create new variants of the K^2-treap. However, other combinations of our techniques, that have not been applied yet, can be easily built without affecting the existing data structures and algorithms: the K^2-treap can be easily generalized to n-dimensional data replacing its K^2-tree representation by the appropriate K^n-tree variant, and the adaptations of the IK^2-tree and K^n-tree to the compression of regions of ones can be transparently adjusted to use any of the different encodings proposed. Additionally, most of the representations can be made dynamic directly using the dK^2-tree, but a more careful study of specific solutions is still future work: for example, a dynamic representation of the IK^2-tree can be obtained directly using a dynamic bitmap representation, but specific solutions to efficiently access and update the variable-length nodes could provide much better efficiency. Even if most of the code is currently being reused and is easy to combine, the publication of a complete set of tools that can be easily used is still work in progress and could give more visibility to the new data structures we created.

Appendix A

Publications and other research results

Publications

Journals to be submitted

- Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro: Compressed dynamic binary relations. Manuscript to be submitted.
- Sandra Álvarez-García, Guillermo de Bernardo, Nieves R. Brisaboa, and Gonzalo Navarro. Indexed representations of ternary relations. Manuscript to be submitted.
- Guillermo de Bernardo, Nieves R. Brisaboa, Roberto Konow, and Gonzalo Navarro: Range top-k queries in compact space. Manuscript to be submitted.
- Nieves R. Brisaboa, Guillermo de Bernardo, Gilberto Gutierrez, Miguel R. Luaces, Jose R. Paramá: Join queries between raster and vectorial data using K^2-trees. Manuscript to be submitted.

International conferences

- Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. Compressed dynamic binary relations. In Proceedings of the Data Compression Conference (DCC), pages 52–61, 2012
- Guillermo de Bernardo, Sandra Álvarez-García, Nieves R. Brisaboa, Gonzalo Navarro, and Oscar Pedreira: Compact querieable representations of raster data. In Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE), pages 96–108, 2013.

- Guillermo de Bernardo, Nieves R. Brisaboa, Diego Caro, and M. Andrea Rodríguez: Compact data structures for temporal graphs. In Proceedings of the Data Compression Conference, page 477, 2013.
- Sandra Álvarez-García, Nieves R. Brisaboa, Guillermo de Bernardo, and Gonzalo Navarro. Interleaved k2-tree: Indexing and navigating ternary relations. In Proceedings of the Data Compression Conference (DCC), 2014.
- Nieves R. Brisaboa, Guillermo de Bernardo, Roberto Konow and Gonzalo Navarro: K2-Treaps: Range Top-k Queries in Compact Space. To appear in Proceedings of the International Symposium on String Processing and Information Retrieval (SPIRE 2014)

International research stays

- *February, 2011 - July, 2011.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).
- *February, 2013.* Research stay at Universidad de Chile, Departamento de Ciencias de la Computación (Santiago, Chile).

Appendix B

Resumen del trabajo realizado

La representación comprimida de información ha sido una necesidad básica en casi cualquier área de Ciencias de la Computación, y prácticamente desde sus inicios. Aunque el total de espacio de almacenamiento ya no es un problema tan importante hoy en día, dado que el almacenamiento en memoria secundaria (disco) es barato y puede contener gran cantidad de datos, el tiempo de acceso es todavía un cuello de botella muy importante en muchas aplicaciones. Tradicionalmente el acceso a memoria secundaria ha sido mucho más lento que el acceso a memoria principal, lo que ha llevado al desarrollo de representaciones comprimidas que puedan almacenar la misma información en menos espacio.

Se ha dedicado mucho esfuerzo al desarrollo de estructuras de datos compactas para representar todo tipo de información: textos, permutaciones, árboles, grafos, etc. Las estructuras de datos compactas normalmente permiten un tiempo de procesamiento más bajo que estructuras clásicas, simplemente debido a estar almacenadas en niveles más altos de la jerarquía de memoria, con lo que aprovechan la diferencia de velocidad en el acceso a distintos niveles de esta jerarquía.

En esta tesis nos centramos en la representación de datos multidimensionales, particularmente en datos bidimensionales que aparecen en diferentes dominios en forma de matrices/*grids*, grafos o relaciones binarias. Existen diversas estructuras que permiten representar eficientemente este tipo de información. El wavelet tree [GGV03] puede representar grids en los que cada columna contiene un único elemento. El K^2-tree [Lad11] es una estructura inicialmente propuesta para representar la matriz de adyacencia de grafos Web y que puede representar en general cualquier matriz binaria, aprovechando ciertas características en la matriz (en particular, la agrupación de 1s) para obtener compresión.

El objetivo de esta tesis es el desarrollo de nuevas y eficientes representaciones

de datos espaciales, típicamente grids bidimensionales o n-dimensionales, binarias o de enteros. Para conseguir este objetivo, desarrollamos representaciones eficientes que se aprovechan de características habituales en datos espaciales para reducir el espacio utilizado, proporcionando al mismo tiempo métodos eficientes para procesar la información almacenada sobre la representación comprimida. Gran parte de las estructuras propuestas en esta tesis se orientan a la representación en general de datos espaciales, pero también desarrollamos propuestas específicas para diferentes ámbitos de aplicación.

Entre las aplicaciones de las estructuras propuestas se encuentran la representación de bases de datos RDF, la representación grafos temporales, la representación de datos OLAP y muy particularmente la representación de conjuntos de datos raster en Sistemas de Información Geográfica (SIG), en los cuales la información espacial se almacena como un conjunto de valores en una matriz o rejilla regular. Dentro de este último campo, estudiamos la representación de distintos tipos de datos raster: imágenes raster binarias, utilizadas tanto en SIG como en procesamiento de imágenes para su almacenamiento y transmisión en faxes, rasters generales (en los cuales cada celda de la rejilla contiene un número entero) y rasters temporales, en los cuales se almacena la evolución de una única imagen raster a lo largo del tiempo. Por último, y también dentro del ámbito de SIG, proponemos nuevos algoritmos para consultar simultáneamente conjuntos de datos raster (almacenados usando nuestras propuestas) y conjuntos de datos vectoriales (almacenados usando una estructura clásica, el R-tree [Gut84]).

B.1 Metodología

En este trabajo se han estudiado diversos problemas relacionados con la representación de datos multidimensionales, particularmente datos espaciales, y se han propuesto soluciones a múltiples problemas. Como resultado se han obtenido una serie de algoritmos y estructuras de datos que permiten representar datos espaciales de diferentes características con soporte eficiente para consultas usuales. El planteamiento seguido para obtener estos resultados ha sido el siguiente:

- Se realizó un estudio bibliográfico en relación a las representaciones existentes de datos multidimensionales, tanto las estructuras de datos compactas generales utilizadas para la representación de relaciones binarias como representaciones específicas utilizadas en la representación de bases de datos temporales, espaciales y espacio-temporales. El objetivo de esta parte era adquirir el mayor conocimiento posible del estado del arte en estructuras de datos compactas, la representación de datos espaciales y en particular las soluciones existentes para la representación de datos en ámbitos de aplicación como la representación de grids de valores en SIG. Es de destacar que este estudio se realizó en distintas fases, comenzando por estructuras compactas

para la representación de relaciones binarias como el wavelet tree o el K^2-tree para determinar sus limitaciones y posibles mejoras. A partir de este estudio se analizaron las posibilidades de expansión a otras áreas de aplicación y se estudiaron específicamente soluciones aplicadas en cada una de ellas.

- El análisis de estructuras generales para la representación de relaciones binarias nos llevó a determinar las limitaciones existentes a la hora de obtener una representación general que permita almacenar de forma compacta datos multidimensionales aprovechando sus características específicas y con acceso eficiente a la información. Centrándonos en particular en el K^2-tree, que ya había sido estudiado en diferentes dominios, quedaron claras importantes limitaciones en una estructura de datos que por otra parte había mostrado ser muy competitiva en varios dominios: las características del K^2-tree permiten una muy buena compresión de matrices binarias que sean esparsas, particularmente si los 1s están muy agrupados. La compresión eficiente de valores agrupados es interesante en datos espaciales, ya que es una característica habitual, sin embargo las matrices no son necesariamente binarias ni esparsas. Además, el K^2-tree es una estructura de datos esencialmente estática y sólo se había propuesto para datos bi-dimensionales. Todas estas limitaciones nos llevaron al desarrollo de nuevas propuestas que solucionasen cada uno de los problemas manteniendo las características interesantes y deseables del K^2-tree, como su buena compresión en la práctica y la simplicidad de las operaciones de consulta sobre la representación comprimida.

- Como hemos dicho, el estudio de las limitaciones en el K^2-tree nos llevó al estudio de nuevas representaciones basadas en la misma filosofía (representación de una matriz mediante el particionamiento recursivo siguiendo una estrategia similar a la de los quadtrees [Sam84]). La mejora de las distintas limitaciones del K^2-tree nos llevó al desarrollo por una parte de una versión dinámica del K^2-tree, una generalización para datos n-dimensionales y el desarrollo de nuevas soluciones para la representación de datos espaciales que pueden ser representados por matrices no necesariamente esparsas ni binarias, como es el caso de datos raster en SIG o datos OLAP. El resultado de todo este desarrollo fue una familia completa de estructuras, que pueden combinarse entre si y aplicarse a nuevos problemas como la representación de grafos temporales o datos raster temporales, también estudiados a lo largo de la tesis.

- Cada una de las propuestas se ha validado comparando su eficiencia bien con versiones existentes y más limitadas de la misma estructura bien con alternativas del estado del arte para solucionar el mismo problema. En particular, mejoras al K^2-tree como la generalización a datos n-dimensionales o el K^2-tree dinámico se comparan con el K^2-tree original para evaluar la calidad de las nuevas propuestas y mostrar la variación en relación con la

estructura original, independientemente de sus aplicaciones. En las diferentes áreas de aplicación, las variantes introducidas para la representación específica de datos se evalúan analizando su comportamiento en comparación con estructuras de datos alternativas en cada uno de los dominios.

B.2 Contribuciones y conclusiones

Como ya hemos adelantado, en esta tesis presentamos un grupo de estructuras de datos diseñadas para la representación de datos multidimensionales. La mayor parte de las propuestas están basadas en el K^2-tree, una estructura de datos diseñada para representar de forma compacta matrices de adyacencia muy esparsas como las que aparecen en grafos Web. La mayor parte de nuestras propuestas intentan superar alguna de las limitaciones de esta estructura, mejorando su funcionamiento para la representación de matrices con otras características o ampliando sus funcionalidades para ser utilizada en otras áreas. En esta sección describimos más detalladamente cada una de las contribuciones individuales, incluyendo las áreas de aplicación de cada una de ellas:

1. Nuestra primera contribución es el diseño, implementación y evaluación experimental de estructuras de datos para la representación de grids o matrices binarias, con la capacidad de comprimir de forma eficiente zonas con valores similares (zonas de 0s o de 1s). Nuestra propuesta está basada en el K^2-tree, pero es capaz de comprimir matrices binarias en las cuales existen grandes zonas de 0s y de 1s, por lo que no requiere que la matriz sea esparsa como el K^2-tree original. Nuestra propuesta puede ser denominada por tanto "K^2-tree con compresión de unos" o K^2-tree1, dado que elimina la ineficiencia de los K^2-trees para comprimir matrices con grandes zonas de unos. Nuestra propuesta conceptual permite distintas variantes o implementaciones. Todas ellas permiten el acceso eficiente a regiones o celdas específicas de la matriz que representan. Para mostrar su eficiencia, comparamos nuestra propuesta con representaciones compactas de quadtrees, que son conceptualmente similares a un K^2-tree. Nuestros resultados muestran que nuestras variantes son similares en espacio a las representaciones de quadtrees más compactas en el estado del arte, y soportan una gran variedad de algoritmos para procesar o consultar la matriz que no son habitualmente eficientes en las alternativas más compactas. En particular, nuestras propuestas proporcionan algoritmos eficientes para acceder a regiones específicas de la matriz, obteniendo tiempos de consulta más rápidos que otras representaciones en mucho menos espacio.

2. Nuestra segunda contribución es el diseño e implementación de una estructura de datos compacta para la representación de matrices multidimensionales, o relaciones n-arias. Nuestra propuesta es una generalización de los conceptos

en el K^2-tree a problemas en más dimensiones, de donde sale su nombre K^n-tree. Como parte del desarrollo de esta nueva estructura de datos también analizamos alternativas al K^n-tree usando otras variantes del K^2-tree: el MK^2-tree y el IK^2-tree se basan en la descomposición de una relación ternaria en una colección de relaciones binarias, que son representadas utilizando un K^2-tree por relación binaria (MK^2-tree) o una estructura de datos que agrupa una colección de K^2-trees (IK^2-tree). Analizamos el K^n-tree en comparación con estas alternativas, comparando las áreas de aplicación recomendables para cada una de ellas. Como prueba de concepto de la aplicabilidad de la nueva propuesta, estudiamos la representación de grafos temporales utilizando las diferentes estructuras de datos para relaciones ternarias: un K^3-tree (un K^n-tree para 3 dimensiones), y la representación del grafo en todos sus instantes utilizando un K^2-tree por instante de tiempo. Como parte de esta evaluación también proponemos una variante del IK^2-tree que permite codificar diferencialmente la colección de K^2-trees ahorrando gran parte del espacio utilizado para representar el grafo temporal. Esta estructura, que llamamos diff-IK^2-tree, es una contribución de interés en sí misma ya que es aplicable en general para representar colecciones de datos similares. Nuestra evaluación experimental muestra que las distintas representaciones son aplicables al problema y tienen distintas ventajas e inconvenientes: mientras el K^3-tree es simple y eficiente en consulta, el diff-IK^2-tree puede comprimir mucho más la representación si el grafo temporal sufre pocos cambios.

3. Nuestra tercera contribución es el diseño e implementación de una representación de matrices multidimensionales con soporte de consultas *top-k* de rango (recuperar los k resultados más importantes, en este caso los k resultados con valor más alto, en una región de la matriz). Nuestra nueva estructura de datos se llama K^2-treap y permite representar de forma muy compacta matrices de valores, soportando al mismo tiempo consultas top-k sobre la matriz completa o sobre una región cualquiera de la matriz. Estudiamos empíricamente la eficiencia de nuestra estructura utilizando matrices sintéticas y datos OLAP reales, una de las aplicaciones más directas de este tipo de consultas. Comparamos nuestra propuesta con una representación más simple basada en un MK^2-tree y una representación alternativa basada en wavelet trees. Nuestros resultados muestran que nuestra propuesta es en general más compacta y más rápida que ambas alternativas y obtiene buenos resultados independientemente de las características del conjunto de datos, mientras el funcionamiento de las alternativas es más variable.

4. Nuestra cuarta contribución es el diseño, implementación y evaluación experimental de una nueva estructura de datos para la representación de

relaciones binarias dinámicas. Nuestra propuesta, llamada K^2-tree dinámico o dK^2-tree, supera la limitación del K^2-tree original que sólo podía ser aplicado a conjuntos de datos estáticos. El dK^2-tree soporta todas las operaciones de consulta del K^2-tree pero además permite actualizaciones de los datos en la matriz que representa, en particular cambiar el valor de las celdas y añadir/quitar filas/columnas a la matriz. Comparamos el dK^2-tree con el K^2-tree estático para mostrar que requiere no demasiado espacio adicional y los tiempos de consulta son también bastante próximos a la representación estática. También estudiamos la utilización de variantes del dK^2-tree que permitan compresión de matrices con grandes regiones de unos (proporcionando una variante dinámica de la primera contribución) y proponemos la aplicación del dK^2-tree a la representación de bases de datos RDF y grafos temporales. En la representación de matrices binarias, mostramos que el dK^2-tree puede representar imágenes raster binarias en menos espacio que alternativas basadas en linear quadtrees, permitiendo al mismo tiempo consultas más rápidas. En el caso de bases de datos RDF, mostramos que el K^2-tree dinámico requiere un espacio adicional reducido sobre la representación estática y sus tiempos de consulta son competitivos.

5. Nuestra quinta contribución es el diseño, implementación y evaluación experimental de nuevas propuestas específicas para la representación de datos raster en SIG. Estudiamos 3 problemas específicos en este dominio, que pueden verse como partes de esta contribución:

   (a) Proponemos estructuras de datos para representar datos raster generales, que permiten consultar valores en una región pero también consultas avanzadas como obtener todas las celdas con valor dentro de un rango, una consulta muy habitual en este dominio. Nuestras propuestas son aplicaciones o variantes de propuestas ya presentadas, como el IK^2-tree (que combinamos con nuestro K^2-tree1 para comprimir regiones uniformes) o el K^3-tree. Nuestras propuestas son muy compactas y similares en estructura a alternativas existentes menos compactas, y al mismo tiempo permiten resolver eficientemente consultas filtradas por regiones espaciales o por valor de las celdas. Comparamos nuestras propuestas entre sí y con alternativas existentes, mostrando las ventajas de cada propuesta y demostrando que nuestras propuestas son más eficientes en tiempo y espacio que alternativas existentes.

   (b) Proponemos representaciones de datos raster temporales, similares a las presentadas para datos raster generales, y mostramos que nuestras representaciones son más compactas que las alternativas habituales y permiten realizar consultas espacio-temporales de forma sencilla y eficiente.

(c) Proponemos nuevos algoritmos que permiten consultar de forma sincronizada datos raster y datos vectoriales, utilizando nuestras propuestas para datos raster y un R-tree para datos vectoriales. Estudiamos el problema tanto para datos raster binarios como generales, proporcionando un algoritmo que permite reducir significativamente el número de nodos del R-tree accedidos para realizar una consulta. En el caso de datos raster generales, comparamos nuestra propuesta con alternativas basadas en representaciones no comprimidas del raster, mostrando que nuestro algoritmo es más rápido en general además de utilizar menos memoria.

El resultado global obtenido de estas contribuciones es una familia de estructuras de datos similares, que combinamos entre sí y de las que mostramos gran cantidad de variantes para problemas específicos. Las posibilidades de combinación de nuestras propuestas ya estudiadas y las todavía no aplicadas muestran la flexibilidad de las contribuciones para adaptarse a la representación de datos multidimensionales en diferentes contextos.

B.3 Trabajo futuro

Como hemos dicho, la combinación de las diferentes propuestas en esta tesis da lugar a una pequeña familia de estructuras con posibles aplicaciones en otras áreas todavía no estudiadas. El hecho de proporcionar diversas estructuras estáticas y una variante dinámica que podría utilizarse para obtener de forma más o menos directa versiones dinámicas de otras variantes o combinaciones de las propuestas también es un factor que consideramos puede abrir las puertas a nuevas áreas de aplicación.

En relación con las estructuras propuestas, hay algunos aspectos que consideramos interesante seguir estudiando en el futuro de cara a mejorar el rendimiento o la aplicabilidad de nuestras propuestas:

- La aplicación del K^n-tree a datos con muchas dimensiones, como ya hemos expuesto, puede ser problemática. Algunas mejoras se han propuesto ya para reducir el efecto del número de dimensiones en la compresión del K^n-tree, como la compresión de caminos unarios presentada en el Capítulo 5. Sin embargo la búsqueda de soluciones más generales para este problema es una línea de investigación interesante.

- El K^2-treap ha mostrado ser eficiente para resover consultas de tipo top-k, y hemos propuesto algunas variantes para datos raster. Sin embargo, creemos que el potencial de esta idea de enriquecer un K^2-tree con información adicional sobre las regiones puede ser mucho mayor, y estamos estudiando otras representaciones basadas en la misma idea en las cuales otro tipo de "resúmenes" se almacene en cada nodo, como el total de celdas con valor en la submatriz.

- La aplicación del dK^2-tree en áreas como la representación de RDF es prometedora pero no proporciona una solución completa al problema. La utilización de índices adicionales en un K^2-tree estático hace que la representación sea mucho más competitiva, pero la aplicación a un entorno dinámico es más compleja. Lo mismo ocurre con la representación del vocabulario necesario en una base de datos RDF, un problema complejo en un entorno dinámico. Otra línea de trabajo futura relacionada con el dK^2-tree es la búsqueda de representaciones más eficientes para ser almacenadas en disco.

- En cuanto a la representación de datos raster temporales, la creación de representaciones diferenciales, en la línea de nuestras propuestas para grafos temporales, puede ser una mejora interesante, si bien como hemos mostrado el uso de nuestras variantes con compresión de unos limita la eficiencia de esta aproximación diferencial.

- Una ampliación interesante sobre nuestros algoritmos de consulta sincronizada de datos raster y vectoriales sería la implementación utilizando estructuras de datos compactas también para los datos vectoriales, lo que sería una interesante alternativa a nuestra propuesta utilizando un R-tree.

- Por último, un trabajo pendiente es la creación y publicación de un framework completo que incluya la familia de estructuras completa y proporcione un método sencillo para su combinación y ampliación. Aunque en este punto gran parte del código de las estructuras puede ser reutilizado de forma sencilla, la creación de un entorno que permita el uso y combinación de las propuestas de forma directa por cualquier usuario daría mayor visibilidad a nuestra propuesta y permitiría su uso de forma más sencilla.

Bibliography

[Abr63] N. Abramson. *Information Theory and Coding*. McGraw-Hill, 1963.

[ÁG14] S. Álvarez-García. *Compact and efficient representations of graphs*. PhD thesis, University of A Coruña, 2014.

[ÁGBdBN14] S. Álvarez-García, N. Brisaboa, G. de Bernardo, and G. Navarro. Interleaved k^2-tree: Indexing and navigating ternary relations. In *Proc. 24th Data Compression Conference (DCC)*, 2014.

[ÁGBF+14] S. Álvarez-García, N. Brisaboa, J. Fernández, M. Martínez-Prieto, and G. Navarro. Compressed vertical partitioning for efficient RDF management. *Knowledge and Information Systems*, 2014. To appear.

[ÁGBFMP11] S. Álvarez-García, N. R. Brisaboa, J. D. Fernández, and M. A. Martínez-Prieto. Compressed k2-triples for full-in-memory RDF engines. In *Association for Information Systems Conference (AMCIS)*, 2011.

[BB04] D. K. Blandford and G. E. Blelloch. Compact representations of ordered sets. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 11–19, 2004.

[BBK01] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[BdBKN14] N. R. Brisaboa, G. de Bernardo, R. Konow, and G. Navarro. k^2-treaps: Range top-k queries in compact space. In *Proc. 21st International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS, 2014. To appear.

[BdBN12] N. R. Brisaboa, G. de Bernardo, and G. Navarro. Compressed dynamic binary relations. In *Proc. 22nd Data Compression Conference (DCC)*, pages 52–61, 2012.

[BDF+98] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system rasdaman. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 575–577, 1998.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[BFLN12] N. Brisaboa, A. Fariña, S. Ladra, and G. Navarro. Implicit indexing of natural language text by reorganizing bytecodes. *Information Retrieval*, 2012.

[BFNP05] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Efficiently decodable and searchable natural language adaptive compression. In *Proc. 28th ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 234–241. ACM Press, 2005.

[BFNP07] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Paramá. Lightweight natural language text compression. *Information Retrieval*, 2007.

[BGMR07] J. Barbay, A. Golynski, J. I. Munro, and S. S. Rao. Adaptive searching in succinctly encoded binary relations and tree-structured documents. *Theoretical Computer Science*, 387(3):284 – 297, 2007.

[BGO+96] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *The VLDB Journal*, 5(4):264–275, December 1996.

[BINP03] N. R. Brisaboa, E. L. Iglesias, G. Navarro, and J. R. Paramá. An efficient compression code for text databases. In *ECIR*, pages 468–481, 2003.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *International Conference on Management of Data*, pages 322–331, 1990.

[BLN09] N. R. Brisaboa, S. Ladra, and G. Navarro. k^2-trees for compact web graph representation. In *SPIRE*, pages 18–30, 2009.

[BLNS13] N. R. Brisaboa, M. R. Luaces, G. Navarro, and D. Seco. Space-efficient representations of rectangle datasets supporting orthogonal range querying. *Information Systems*, 38(5):635–655, 2013.

[CC94] H. K. Chang and J. Chang. Fixed binary linear quadtree coding scheme for spatial data. *Visual Communications and Image Processing*, pages 1214–1220, 1994.

[CCY96] K. L. Chung and H. Chin-Yen. Finding neighbors on bincode-based images in O(n log log n) time. *Pattern Recognition Letters*, 17(10):1117–1124, 1996.

[Che02] P. Chen. Variant code transformations for linear quadtrees. *Pattern Recognition Letters*, 23(11):1253–1262, 2002.

[CHLS07] H. Chan, W. K. Hon, T. W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), May 2007.

[CL11] F. Claude and S. Ladra. Practical representations for web and social graphs. In *Proc. 20th ACM International Conference on Information and Knowledge Management (CIKM)*, pages 1185–1190, 2011.

[Cla96] D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Ontario, Canadá, 1996.

[CN08] F. Claude and G. Navarro. Practical rank/select queries over arbitrary sequences. In *Proc. 15th International Symposium on String Processing and Information Retrieval (SPIRE)*, LNCS 5280, pages 176–187, 2008.

[CNO15] F. Claude, G. Navarro, and A. Ordóñez. The wavelet matrix: An efficient wavelet tree for large alphabets. *Information Systems*, 47:15–32, 2015.

[CT91] J. P. Cheiney and A. Touir. FI-quadtree: A new data structure for content-oriented retrieval and fuzzy search. In *Proc. 2nd International Symposium on Advances in Spatial Databases (SSD)*, volume 525, pages 23–32, 1991.

[CTVM08] A. Corral, M. Torres, M. Vassilakopoulos, and Y. Manolopoulos. Predictive join processing between regions and moving objects. In *Proc. 12th Conference on Advances in Databases and Information Systems Conference (ADBIS)*, volume 5207, pages 46–61. 2008.

[CVM99] A. Corral, M. Vassilakopoulos, and Y. Manolopoulos. Algorithms for joining R-trees and linear region quadtrees. In *Proc. 6th Symposium on Advances in Spatial Databases (SSD)*, pages 251–269, 1999.

[CW84] J. Cleary and I. H. Witten. Data compression using adaptive coding and partial string matching. *IEEE Transactions on Communications*, 32(4):396–402, 1984.

[CW95] K. L. Chung and C. J. Wu. A fast search algorithm on modified S-trees. *Pattern Recognition Letters*, 16(11):1159–1164, 1995.

[CWL97] K. L. Chung, J. G. Wu, and J. K. Lan. Efficient search algorithm on compact S-trees. *Pattern Recognition Letters*, 18(14):1427 – 1434, 1997.

[dBAGB+13] G. de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proc. 20th International Symposium on String Processing and Information Retrieval (SPIRE)*, volume 8214, pages 96–108, 2013.

[dBBCR13] G. de Bernardo, N. R. Brisaboa, D. Caro, and M. A. Rodríguez. Compact data structures for temporal graphs. In *Proc. 23rd Data Compression Conference (DCC)*, page 477, 2013.

[Die89] P. F. Dietz. Optimal algorithms for list indexing and subset rank. In *Proc. Algorithms and Data Structures Symposium (WADS)*, pages 39–46, 1989.

[DSS94] W. Dejonge, P. Scheuermann, and A. Schijf. S+-trees: An efficient structure for the representation of large pictures. *CVGIP: Image Understanding*, 59(3):265 – 280, 1994.

[FB74] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[Fis10] J. Fischer. Optimal succinctness for range minimum queries. In *Proc. Latin American Theoretical INformatics Symposium (LATIN)*, pages 158–169, 2010.

[FM05] P. Ferragina and G. Manzini. Indexing compressed text. *Journal of the ACM*, 52(4):552–581, 2005.

[FM08] A. Farzan and J. I. Munro. A uniform approach towards succinct representation of trees. In *Proc. 11th Scandinavian Workshop on Algorithm Theory (SWAT)*, pages 173–184, 2008.

[FM11] A. Farzan and J. Ian Munro. Succinct representation of dynamic trees. *Theoretical Computer Science*, 412(24):2668–2678, 2011.

[FS89] M. Fredman and M. Saks. The cell probe complexity of dynamic data structures. In *Proc. 21st Annual ACM Symposium on Theory of Computing (STOC)*, pages 345–354, 1989.

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

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

[GGMN05] R. González, S. Grabowski, V. Mäkinen, and G. Navarro. Practical implementation of rank and select queries. In *Proc. 4th Workshop on Efficient and Experimental Algorithms (WEA'05)*, pages 27–38, 2005.

[GGV03] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. In *Proc. 14th Symposium on Discrete Algorithms (SODA)*, pages 841–850, 2003.

[GGV04] R. Grossi, A. Gupta, and J. S. Vitter. When indexing equals compression: experiments with compressing suffix arrays and applications. In *Proc. 15th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 636–645, 2004.

[GHSV06] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. In *Proc. 16th Data Compression Conference (DCC)*, pages 213–222, 2006.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 368–373, 2006.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[GW05] A. Galton and M. F. Worboys. Processes and events in dynamic geo-networks. In *Proc. 1st GeoSpatial Semantics (GeoS)*, volume 3799 of *LNCS*, pages 45–59, 2005.

[HM10] M. He and J. I. Munro. Succinct representations of dynamic strings. In *Proc. 17th International Symposium on String Processing and Information Retrieval (SPIRE)*, pages 334–346, 2010.

[HSS03] W. K. Hon, K. Sadakane, and W. K. Sung. Succinct data structures for searchable partial sums. In *Proc. 14th International Symposium on Algorithms and Computation (ISAAC)*, pages 505–516, 2003.

[Huf52] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, 1952.

[Jac89] G. Jacobson. *Succinct static data structures*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1989.

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

[KE80] E. Kawaguchi and T. Endo. On a method of binary-picture representation and its application to data compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(1):27 –35, jan. 1980.

[Lad11] S. Ladra. *Algorithms and Compressed Data Structures for Information Retrieval*. PhD thesis, University of A Coruña, 2011.

[Lin97a] T. W. Lin. Compressed quadtree representations for storing similar images. *Image and Vision Computing*, 15(11):833–843, 1997.

[Lin97b] T. W. Lin. Set operations on constant bit-length linear quadtrees. *Pattern Recognition*, 30(7):1239–1249, 1997.

[LS89] D. Lomet and B. Salzberg. Access methods for multiversion data. *Proc. ACM SIGMOD International Conference on Management of Data*, 18(2):315–324, 1989.

[LS90] D. B. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[MM04] F. Manola and E. Miller. Rdf primer, February 2004.

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

[MN08] V. Mäkinen and G. Navarro. Dynamic entropy-compressed sequences and full-text indexes. *ACM Transactions on Algorithms*, 4(3):32:1–32:38, July 2008.

[MNPP97] Y. Manolopoulos, E. Nardelli, A. Papadopoulos, and G. Proietti. MOF-tree: A spatial access method to manipulate multiple overlapping features. *Information Systems*, 22(8):465 – 481, 1997.

[MNPT01] Y. Manolopoulos, E. Nardelli, G. Proietti, and E. Tousidou. A generalized comparison of linear representations of thematic layers. *Data & Knowledge Engineering*, 37(1):1–23, 2001.

[MNW98] A. Moffat, R. M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, 1998.

[MPFC12] M. A. Martínez-Prieto, J. D. Fernández, and R. Cánovas. Compression of RDF dictionaries. In *Proc. 27th Annual ACM Symposium on Applied Computing (SAC)*, pages 340–347, 2012.

[MR97] J. I. Munro and V. Raman. Succinct representation of balanced parentheses, static trees and planar graphs. In *Proc. IEEE Symposium on Foundations of computer Science (FOCS)*, pages 118–126, 1997.

[Mun96] J. Ian Munro. Tables. In *Proc. 16th Conference of Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 37–42, 1996.

[NM07] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Computing Surveys*, 39(1):art. 2, 2007.

[NN13] G. Navarro and Y. Nekrich. Optimal dynamic sequence representations. In *Proc. ACM-SIAM Symposium on Discrete Algorithms*, pages 865–876, 2013.

[NNR13] G. Navarro, Y. Nekrich, and L. Russo. Space-efficient data-analysis queries on grids. *Theoretical Computer Science*, 482:60–72, 2013.

[NP95] E. Nardelli and G. Proietti. Efficient secondary memory processing of window queries on spatial data. *Information Sciences – Informatics and Computer Science*, 84(1-2):67–83, May 1995.

[NP97] E. Nardelli and G. Proietti. Time and space efficient secondary memory representation of quadtrees. *Information Systems*, 22(1):25–37, March 1997.

[NP99] E. Nardelli and G. Proietti. S*-tree: An improved S+-tree for coloured images. In *Proc. 3rd East-European Conference on Advances in Databases and Information Systems (ADBIS)*, volume 1691, pages 156–167, 1999.

[NST99] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Evaluation of access structures for discretely moving points. In *Proc. International Workshop on Spatio-Temporal Database Management (STDBM)*, pages 171–188, 1999.

[NW10a] T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, 2010.

[NW10b] T. Neumann and G. Weikum. x-rdf-3x: Fast querying, high update rates, and consistency for rdf databases. *Proc. VLDB Endowment*, 3(1-2):256–263, 2010.

[OS07] D. Okanohara and K. Sadakane. Practical entropy-compressed rank/select dictionary. In *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[OY92] M. A. Ouksel and A. Yaagoub. The interpolation-based bintree and encoding of binary images. *CVGIP: Graphical Models and Image Processing*, 54(1):75 – 81, 1992.

[Pag99] R. Pagh. Low redundancy in static dictionaries with o(1) worst case lookup time. In *Proc. 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 595–604, 1999.

[PS08] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Latest version available as `http://www.w3.org/TR/rdf-sparql-query/`, January 2008.

[RLK+11] C. Ren, E. Lo, B. Kao, X. Zhu, and R. Cheng. On Querying Historical Evolving Graph Sequences. *Proc. International Conference on Very Large Data Bases (VLDB)*, (11), 2011.

[Rob81] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[RRR01] R. Raman, V. Raman, and S. S. Rao. Succinct dynamic data structures. In *Proc. Algorithms and Data Structures Symposium (WADS)*, pages 426–437, 2001.

[RRR02] R. Raman, V. Raman, and S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees and multisets. In *Proc. 13th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 233–242, 2002.

[Sad03] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *Journal of Algorithms*, 48(2):294–313, 2003.

[Sam84] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, June 1984.

[Sam90a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[Sam90b] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[SdMNZBY00] E. Silva de Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast and flexible word searching on compressed text. *ACM Transactions on Information Systems*, 18(2):113–139, April 2000.

[SH91] P. Svensson and Z. Huang. Geo-sal: A query language for spatial data analysis. In *Advances in Spatial Databases*, pages 119–140. Springer, 1991.

[SJH93] C. A. Shaffer, R. Juvvadi, and L. S. Heath. A generalized comparison of quadtree and bintree storage requirements. *Image and Vision Computing*, pages 402–412, 1993.

[SW49] C. E. Shannon and W. Weaver. *A Mathematical Theory of Communication*. University of Illinois Press, Urbana, Illinois, 1949.

[TML99] T. Tzouramanis, Y. Manolopoulos, and N. Lorentzos. Overlapping B+-trees: An implementation of a transaction time access method. *Data & Knowledge Engineering*, 29(3):381 – 404, 1999.

[TP01] Y. Tao and D. Papadias. MV3R-tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. 27th International Conference on Very Large Data Bases*, Proc. International Conference on Very Large Data Bases (VLDB), pages 431–440, 2001.

[TVM98] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: a spatio-temporal access method. In *Proc. 6th ACM international symposium on Advances in geographic information systems*, GIS '98, pages 1–7, 1998.

[TVM00] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Multiversion linear quadtree for spatio-temporal data. In *Proc. 4th East-European Conference on Advances in Databases and Information Systems (ADBIS)*, pages 279–292, 2000.

[TVM01] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Time split linear quadtree for indexing image databases. In *Proc. International Conference on Image Processing (ICIP)*, pages 733–736, 2001.

[TVM04] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Benchmarking access methods for time-evolving regional data. *Data & Knowledge Engineering*, 49(3):243–286, June 2004.

[VM93] M. Vassilakopoulos and Y. Manolopoulos. Analytical results on the quadtree storage-requirements. In *Computer Analysis of Images and Patterns*, volume 719, pages 41–48. Springer Berlin Heidelberg, 1993.

[VM95] M. Vassilakopoulos and Y. Manolopoulos. Dynamic inverted quadtree: A structure for pictorial databases. *Information Systems*, 20(6):483–500, September 1995.

[VZ09] A. Vaisman and E. Zimányi. A multidimensional model representing continuous fields in spatial data warehouses. In *ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, pages 168–177, 2009.

[WC97] J. G. Wu and K. L. Chung. A new binary image representation: Logicodes. *Journal of Visual Communication and Image Representation*, 8(3):291 – 298, 1997.

[WD04] M. Worboys and M. Duckham. *GIS: A Computing Perspective, 2Nd Edition*. CRC Press, Inc., 2004.

[Wel84] T. A. Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6):8–19, 1984.

[Wil06] K. Wilkinson. Jena Property Table Implementation. In *SSWS*, pages 35–46, 2006.

[WKB08] C. Weiss, P. Karras, and A. Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endowment*, 1(1):1008–1019, August 2008.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

[Wor05] M. Worboys. Event-oriented approaches to geographic phenomena. *International Journal of Geographical Information Science*, 19(1):1–28, 2005.

[YCT00] Y. H. Yang, K. L. Chung, and Y. H. Tsai. A compact improved quadtree representation with image manipulations. *Image and Vision Computing*, 18(3):223–231, 2000.

[ZHNB06] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar ram-cpu cache compression. In *Proc. 22nd International Conference on Data Engineering (ICDE)*, page 59, 2006.

[ZL77] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[ZL78] J. Ziv and A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory*, 24(5):530–536, 1978.