Cristina Sirangelo

# Approximate Query Answering on Multi-dimensional Data

– Ph.D. Thesis –

October 21, 2005

*Advisor*: Prof. Domenico Saccà
*Coordinator*: Prof. Domenico Talia

DEIS
Department of Electronics, Computer Science and Systems
University of Calabria, Arcavacata di Rende (CS) Italy

*to Raffaele*

# Preface

The multi-dimensional model represents data as a set of measure values associated to points in a multi-dimensional space. Databases of this type are common in the context of *On-line Analytical Processing* (OLAP), for supporting the decision-making process of enterprizes; they are also used for describing attribute frequency distributions for query optimization in relational DBMSs, and can be found in *Geographic Information Systems* (GIS) for representing data values associated to geographical locations. The main goal of applications operating in these contexts is to extract large-scale summary information from the available data, rather than to inquire detailed data items. In particular, these applications are mainly interested in exploring and aggregating values within specified ranges of the domain. These kinds of aggregate query are called *range queries*.

A common feature in all the cited contexts is the very large size of the explored data sets. This makes exact query answering practically unfeasible: the exact evaluation of complex aggregations, involving large portions of the data set, would require a prohibitive linear scanning. Such a high computational cost is not tolerable in the above scenarios where efficient and promptly data analysis is one of the main requirements. Moreover, a dramatic precision in range query answers is usually not needed; "good" estimates of query results often suffice to accomplish effectively many data analysis tasks. It is therefore very important to find approximate answers to range queries quickly in order to allow flexible, interactive ad effective data exploration.

The most successful approach to the problem of providing "fast" approximate answers to aggregate queries consists in summarizing data into compact structures, and issuing range queries over summary data, rather than over detailed ones. Following this approach, some approximation is introduced in query answers, as summarization is a lossy compression. On the other hand, the amount of data corresponding to the range of a query which must be accessed in the summary structure is much less than the number of elements that should be extracted from the original data set. This implies that com-

puting query answers on summarized data can be much faster than evaluating them on raw ones.

In other application scenarios the need to summarize multi-dimensional data arises also from other issues than making query answering more efficient. In particular, in the context of *data stream* processing, data is produced by "continuous" sources, such as geographically distributed sensors, which send possibly endless streams of readings to a centralized processor. No bound can be given to the amount of information which goes through the data stream management system, whereas the available storage space for representing the received information is bounded. As a consequence, processed data items can be either discarded or partially archived, but exact information about all the stream content cannot be stored, and exact answers to most common queries cannot be computed. A possible solution to this issue consists in dynamically summarizing data, as they are received, into a compact structure which fits in the available storage space, in order to pose queries on summary data. This approach aims at allowing approximate answers on the stream content, by storing as much information as possible on received data.

This thesis gives a comprehensive overview of the issues related to multi-dimensional data summarization for approximate query answering, and defines new techniques for summarizing both static and streaming data, by overcoming the main limitations of state-of-the-art approaches.

## Acknowledgements

There are a lot of people I want to acknowledge for their support throughout my Ph.D.

First of all, I would like to thank my Supervisor, Prof. Domenico Saccà who has inspired my work since the beginning. I'm grateful to him for believing in my potential, for bringing out my qualities, and above all for giving continued fuel to my research with his striking ideas, which have always stimulated and challenged my abilities.
My deep gratitude goes to Prof. Sergio Greco for all his care and friendship, for his exceptional leadership of the Department, for being an example, over these years, with his outstanding energy and enthusiasm. I will be always indebted to him for his deep insight into my aptitudes, his unreserved help and encouragement, which supported me much more than I deserved.
It gives me also great pleasure to thank Prof. Domenico Talia who coordinated my Ph.D and kindly helped me with the completion of my doctoral program.
A big "thank you" to Filippo Furfaro who worked with me in every step of this research. I warmly thank him for these years and for the generosity he has shown in sharing his experience with me. I owe him special gratitude for always being a steadfast source of advise, for his balance and tact, and for his careful and invaluable proof-reading.
Many thanks go also to Francesco Buccafurri for writing the first golden words

in the enjoyable joint work which started the journey of this thesis.
I'm also thankful to Massimo Mazzeo who carried the hard work of implementing the ideas proposed in this thesis. Above all I thank him for offering me his tireless help, and for our long-standing friendship.
In addition I want to thank all the staff of the Department who made it possible to complete this thesis, especially Giovanni Costabile and Franco De Marte for their expertise and timely assistance in technical problems.

It's also a pleasure for me to take this opportunity to thank all the GEMO group members, who are giving me the honor of being part of their prestigious team. I want to gratefully acknowledge Serge Abiteboul for encouraging me and believing in me. Many thanks to Victor Vianu for our enlightening discussions, and to Ioana Manolescu, Tova Milo and Susan Davidson for welcoming me as a family. But especially I owe a debt of gratitude to Luc Segoufin for guiding me through fine thinking, for inspiring my ideas without ever forcing them, for teaching me how to enjoy research, and giving me stronger motivation and enthusiasm than I could ever hope.

I would also like to thank all my colleagues and friends, for their warm friendship and helpfulness. A special "thank you" to Sergio Flesca, Elio Masciari, Andrea Pugliese, Andrea Tagarelli and Pierangelo Veltri for their kindness and advise; to Irina Trubitsyna for being a patient and discreet office mate; to Luciano Caroprese for all our talking; to Ester Zumpano and Carmela Comito for being close to me anytime, especially when I needed most, and for all the shopping and gossiping together, which used to cheer me up and bring good mood.
"Thank you" also to all the friends I met at INRIA, especially Antonella Poggi, Bogdan Cautis, Boris Vrdoljak, Gabriela and Nicolaas Ruberg and Nicoleta Preda, who shared with me the best time I spent in Paris.

Finally and most importantly of all, I wish to express everlasting gratitude to my parents and my brother for all the strength they have given me with their love, and to Raffaele, to whom I dedicate this thesis, for all the music he has brought to my life.

Rende,

October 2005 *Cristina Sirangelo*

# Contents

# Introduction

In the last few years more and more business and scientific organizations have centered their activity and competitive power in the adoption of very large information systems, capable of managing, querying and analyzing huge volumes of data. This trend has been mainly encouraged by the recent advances of information technologies and the collapsing costs of storage and computing resources. Moreover, the spreading of network connectivity has enlarged in scale the perspective of data exchange, thus making the volume of data accessible by every single organization explode. The availability of massive data sets has given rise to lively interest towards applications aimed at extracting useful knowledge from huge collections of detailed data. In this scenario we find many analytical applications which have gained ground in the industry and are also the focus of intense research activity, such as *On-Line Analytical Processing* (OLAP), statistical and scientific data analysis, query optimization in RDBMS, environmental monitoring in Geographic Information Systems, sensor data analysis, etc.

Although heterogeneous, the above-cited contexts share a common main objective: performing advanced data analysis aimed at retrieving non-trivial information, by efficiently executing complex and large-scale data processing. In fact, most data analysis tasks are not concerned with accessing detailed pieces of information, on the contrary they are mainly interested in viewing data at a variety of summary levels and from different perspectives. To this end complex aggregations, involving large portions of the data domain need to be computed. A data model which turns out to be especially well suited for supporting this kind of data exploration is the so called *multi-dimensional model*: it consists in viewing data as a set of values associated to points in a multi-dimensional space. The meaning of either dimensions and data values depends on the application context: OLAP *datacubes* storing measure values associated to attributes of interest, *joint frequency distributions* representing frequencies of relational attributes, GIS data sets recording environmental readings associated to geographical coordinates, are all differing examples of multi-dimensional data sets.

A common operation, at the basis of every complex computation on multi-dimensional data, consists in aggregating values contained in a given range of the multi-dimensional space; these kinds of aggregate query are called *range queries*. The multidimensional view of data is well suited for range query evaluation, nevertheless the amount of data to be accessed for computing exact range query answers can be too large: at least all points lying into the query region must be examined, and this region may involve a significant portion of the overall data set. Thus, computing exact query answers can be a very hard computational task, as it may require large volumes of disk-resident data to be accessed and processed. Such a high computational cost is not tolerable in the above-cited scenarios where efficient and promptly data analysis is one of the main requirements. Indeed, in these contexts, efficiently aggregating data is such a crucial issue, that high accuracy in query answers becomes a secondary requirement. Moreover, a dramatic precision in range query answers is usually not needed; "good" estimates of query results often suffice to accomplish effectively many data analysis tasks.

The most effective approach for providing fast approximate answers to range queries consists in summarizing the data distribution into a compact structure, and issuing range queries on summary data (rather than original ones). Approximation in query answers is due to the "lossy" nature of the adopted summarization strategies, which do not keep all information about the original data distribution. On the other hand, the amount of data that must be accessed to evaluate a range query on the data synopsis is much smaller than the amount of detailed data selected by the query, thus range query answering can be accomplished much faster.

One of the most popular techniques adopted for summarizing multi-dimensional data and for supporting fast query answering, is the histogram-based one. A histogram over a multi-dimensional data distribution is built by partitioning the multi-dimensional domain into a number of hyper-rectangular blocks (called *buckets*), and then storing some aggregate data for each block. Aggregation implies loss of information about the data distribution inside histogram buckets, which will have to be approximately reconstructed when answering queries. Therefore, given a bounded storage space for the summarized representation, a crucial issue is finding the histogram which "best" partitions data values among buckets, so as to allow the most accurate reconstruction of the original data distribution.

Building the "most effective" multi-dimensional histogram (called *V-Optimal*) was shown to be a NP-Hard problem, even in the two-dimensional case. Hence, feasible approaches to the problem of constructing histograms providing reasonable accuracy for query estimates are based on greedy strategies. Several histogram-based summarization techniques, adopting varied heuristics for data partitioning, have been defined in the literature.

The work presented in this thesis is mainly focused on the definition of new histogram-based approaches for summarizing multi-dimensional data.

Our proposals are aimed at overcoming the main limitations of state-of-the-art methods. In particular, the main drawback of existing histogramming strategies is that they do not scale up (in terms of accuracy) to high-dimensionality scenarios. In other words, state-of-the-art histograms, although intended to deal with generic multi-dimensional data, provide satisfiable estimation accuracy only in the low-dimensional case, while their performances tend to worsen dramatically as dimensionality increases. This problem is referred to as the *curse of dimensionality*: as the number of dimensions increases, the size of the data domain grows much faster than the number of data points, thus data become sparser and sparser; as a consequence, the number of buckets needed to achieve a satisfiable degree of accuracy explodes. In high-dimensionality scenarios no technique is known to succeed in constructing histograms yielding "reasonable" error rates within a "reasonable" space bound. At the same time, no technique based on other approaches than histograms (such as wavelets, sampling, etc.) is known to provide satisfiable accuracy in the multi-dimensional scenario.

In this thesis we adopt two possible approaches to cope with this problem. One approach consists in developing ad-hoc techniques tailored on specific number of dimensions. In fact, by exploiting the distinctive characteristics of restricted applications, new solutions can be designed which outperform general methods intended to work for any-dimensionality data.
The other followed direction is to design multi-dimensional summary structures specifically targeted towards data with high dimensionality, by tackling the main issues lowering the estimation performances of state-of-the-art techniques. In fact, the low accuracy in query estimates provided by traditional histograms is also due to the ineffectiveness of the adopted heuristics guiding the histogram construction, and to a poorly intensive usage of the available storage space.

In particular, as to the first approach, in this thesis we consider the specific case of two-dimensional data sets, which are the focus of a number of interesting applications. We design a summarization strategy meant to exploit the peculiarities of two-dimensional data sets: it adopts a quad-tree based data partitioning scheme (a common partitioning strategy for spatial data) on top of which a very compact summary structure is defined (called *Quad-tree Summary* - $QTS$). A $QTS$, by adopting a redundancy-free encoding, stores aggregate information about each block of the quad-tree partition (either blocks corresponding to leaf nodes and internal nodes of the quad-tree) thus resulting in a hierarchical "multi-resolution" data summarization. Greedy criteria guide the partition construction in such a way that the resulting distribution inside quad-tree blocks can be accurately approximated. The intra-bucket estimation is further enhanced by storing, in addition, some compact low-resolution description of the actual data distribution inside buckets (called *index* and designed specifically for two-dimensional data).

For summarizing general multi-dimensional data, we propose several approaches which still provide a satisfiable degree of accuracy in scenarios with

high dimensionality. First we investigate the use of binary hierarchical partitions of multi-dimensional data as a basis for the construction of effective histograms. In particular, we introduce two new classes of histograms, namely *Hierarchical Binary Histograms* (*HBH*) and *Grid Hierarchical Binary Histograms* (*GHBH*). *HBH*s are obtained by recursively splitting blocks of the data domain into two non overlapping sub-blocks. The tree corresponding to the binary partition is exploited to define very specific space-efficient representation models, where bucket boundaries are represented implicitly by storing the partition tree. the saved space is invested to obtain finer grain blocks, which approximate data in more detail. On top of that, in *GHBH*s we introduce a constraint on the hierarchical partition scheme, allowing each block of data to be partitioned only by splits lying onto a regular grid defined on it. We show how the introduction of the grid-constrained partitioning of *GHBH*s can be exploited to further enhance the physical representation efficiency of *HBH*s. The intensive exploitation of the storage space allows either *HBH*s and *GHBH*s to store, within a given amount of memory, a larger number of buckets w.r.t. histograms using a "flat" explicit storage of bucket boundaries. In order to profit by the increase of the number of available buckets, we propose several new heuristics for the data-driven construction of the histograms; the criteria adopted to choose how to split blocks are efficient to compute and enable effective location of inhomogeneous regions where a finer-grain partition is needed.

As a different approach to the problem of summarizing multi-dimensional data, we combine summarization with the use of data clustering techniques. In particular, we propose a new multi-dimensional histogram, called CHIST, whose construction exploits the capability of clustering algorithms to locate dense regions in a data distribution. The idea of isolating dense regions arises from the observation that estimation performances of histogram-based approaches can be significantly poor when dense and sparse regions occur into the same bucket. Intuitively, this is due to the fact that "much" information is lost when replacing data distributions inside buckets with aggregate values. In our proposal a density-based clustering algorithm is first run to locate dense clusters of the input data, and then the data distributions inside clusters – as well as the distribution outside clusters – are summarized separately by means of a grid-based paradigm.

We conduct a thorough experimental analysis comparing all our proposals for summarizing multi-dimensional data with existing approaches (other classes of multi-dimensional histograms, wavelets, etc.). Experimental results show that the techniques proposed in this thesis yield much lower error rates than the state-of the-art ones and (in the multi-dimensional case) are much less sensitive to the increase of dimensionality.

Finally, in this thesis we extend the scope of our approach to the context of *data stream* processing. In this scenario the need to summarize data arises also from other issues than making query answering more efficient. In fact, data is assumed to take the form of a continuous, unbounded flow of information,

generated by special devices used to monitor real life phenomena (such as live weather conditions, network traffic, etc.). The data stream processor is provided with a bounded amount of storage space which is typically very small relative to the (possibly unbounded) stream size. Thus, processed data items can be either discarded or only partially archived, but exact information about all the stream content cannot be stored, and exact answers to most common queries cannot be computed. Possible solutions to this problem are based on the incremental maintenance of a summarized structure approximating the content of the whole stream over time: queries involving the received items are evaluated approximately on the data stream summary. Obviously, this issue shares many similarities with the earlier described problem of summarizing static data sets, but there are some differences. The main one concerns the construction of the summary: compact structures used to summarize data streams have to be constructed and maintained dynamically, as data arrive; while for static data, which is usually historical and very unfrequent to be refreshed, summarization is mostly an off-line task.

In the literature dynamic versions of well known histograms have been proposed for the incremental maintenance of summary information on data streams. Following this line, we propose a dynamic adaptation of our quad-tree based summary structures for the summarization of *sensor network data streams*.

A sensor network is a set of sources producing independent streams of readings; the individual streams converge in a centralized system, where they are combined into a unique data stream for data analysis. In our proposal the overall stream is modelled as a two-dimensional data set where the first dimension corresponds to the set of sources, and the other one corresponds to time. In particular, each reading value is represented as a point in the two-dimensional space whose coordinates are the source generating the reading and the timestamp of generation, respectively. The summarization technique used for this two-dimensional data set is suitably designed to take into account the peculiar nature of time dimension: first of all, time dimension is potentially infinite and data to be summarized arrive dynamically by continuously "updating" the corresponding array locations; moreover, "old" data is likely to be less "interesting" to the user than more recent one. The proposed dynamic summarization strategy divides the sensor data stream into "time windows" of the same size. Each time window is represented separately by a quad-tree summary which is populated dynamically as data arrive. Moreover, as new data is received, "old" windows are progressively compressed (or possibly removed) to release the storage space needed to represent new readings. Thus, recent information (which is usually the most relevant to retrieve) is represented with more detail than old one. Furthermore, an embedded index, allowing fast access to data in specified time intervals, makes the proposed structure especially well suited for both incremental maintenance and fast query answering.

## Thesis Organization

In Chap. 1 we introduce the multidimensional data model. We describe the most important application contexts where data is organized in a multi-dimensional fashion, and we discuss the central issue addressed in this thesis: computing "fast" approximate answers to aggregate queries on multi-dimensional data sets. Finally, we provide a formal framework for dealing with multi-dimensional data.

In Chap. 2 we present summarization of multi-dimensional data as an approach for providing "fast" answers to aggregate queries. We provide an overview of the main state-of-the-art techniques for summarizing multi-dimensional data (namely *histograms*, *wavelets* and *sampling*). In particular, we describe multi-dimensional histograms in detail: we review the main theoretical results about the construction of *V-Optimal* histograms , and we describe the best existing heuristics for constructing effective multi-dimensional histograms.

In Chap. 3 we propose the new histogram-based summary structure specifically designed for two-dimensional data (named *Quad-tree Summary* - QTS). We define several classes of *indices* to provide a low-resolution description of data distribution inside QTS buckets. We extend the definition of V-Optimal histogram to QTSs and we address complexity issues related to its construction. We then propose efficient greedy algorithms for finding effective sub-optimal solutions. We report experimental results showing that the technique yields much better estimation accuracy w.r.t. state-of-the-art methods.

In Chap. 4 we propose the two new classes of multi-dimensional histograms based on hierarchical binary partitions (namely *Hierarchical Binary Histogram - HBH* and *Grid Hierarchical Binary Histogram - GHBH* ). We define their space-efficient physical representation models and we combine them with several new heuristics for guiding the binary partitioning of data. By a complexity analysis, we show that the proposed heuristics are very efficient to compute; on the other hand, we define V-Optimal *HBH*s and *GHBH*s and show that their construction cost is impractical. By means of a thorough experimental analysis, we identify the "best performing" histogram among the proposed versions of *HBH*s and *GHBH*s and we compare it with state-of-the-art approaches, showing its higher accuracy and lower sensitivity to dimensionality.

In Chap. 5 we propose the new technique for constructing multi-dimensional histograms based on data clustering. The technique is orthogonal to any density-based clustering algorithm suitable for the identification of clusters and outliers in multi-dimensional data sets. We describe the summarization of data clusters and outliers by means of a grid partitioning, by showing how it allows possible nesting buckets. We then present a histogram representation scheme which exploits nesting buckets to make query answering more accurate and efficient.

In Chap. 6 we propose the technique for dynamic summarization of sensor data streams. We present the modelling of sensor readings as a two-

dimensional data set and show its representation by means of a sequence of finite time windows. We describe the quad-tree representation of time windows and their embedded index, by showing how they are populated dynamically and summarized progressively, as storage space is needed. We show that the proposed summary structure is suitable for efficient and accurate answering to most common queries on sensor data streams.

# 1

# Multi-dimensional Data: Overview and Basic Notations

*In this chapter we introduce the multidimensional data model. We describe the most important application contexts where data is organized in a multi-dimensional fashion, focusing on the most relevant kind of operation on such data: the computation of aggregations over specified ranges of the domain (*range queries*). Then we introduce the central issue addressed in this thesis: the problem of computing "fast" approximate answers to aggregate queries in order to allow efficient exploration of multi-dimensional data sets. Finally, we provide a formal framework for dealing with multi-dimensional data which will underly the rest of this thesis.*

## 1.1 Introduction

In many application scenarios data can be suitably modelled as a set of measure values associated to points in a multi-dimensional space (*multi-dimensional data*). Databases of this type are common in the context of *Online Analytical Processing* (OLAP) [75] for supporting the decision-making process of enterprizes: measure values define quantities of particular interest for data analysis (such as the total sales or purchases), while dimensions correspond to data attributes (such as the product and the year). Multi-dimensional data are also used for describing attribute frequency distributions for query optimization in relational DBMSs [16, 73]: the selectivity of intermediate query results is estimated by accessing *joint frequency distributions*, storing the number of occurrences of each possible combination of attribute values appearing in a database relation.

Another example of multi-dimensional data can be found in *Geographic Information Systems* (GIS) [5, 62], which store and manage data values associated to geographical locations: dimensions represent geographical coordinates which the reading of different environmental variables (such as pollution or waterfall level) are associated to.

The main goal of applications operating in these contexts is not to inquire detailed data items, but rather to extract large-scale summary information from the available data. In fact, specific applications – such as data mining activities, scientific and statistical data analysis, sensor data analysis, query answering in spatial databases – usually operate on a huge amount of data, but do not return detailed pieces of information: they are mainly interested in exploring and aggregating values within specified ranges of the domain. These kinds of aggregate query are called *range queries*.

For instance, given a data set containing yearly sales of different products, users are likely to be interested in queries such as "*find the total sales for a given product range in a given interval of years*". On data sets recording geographically distributed measurements of pollution level, a typical query is similar to: "*find the average pollution level in a specified space region*". Likewise, query optimizers in RDBMSs, in order to select the most suitable query execution plans, perform preliminary explorations of the content of the relations, asking for queries such as "*the number of relational tuples where a given set of attributes have values in a specified range of their domain*".

A common feature in all the cited contexts is the very large size of the explored data sets; furthermore, queried regions may involve a significant portion of the overall data. This makes exact query answering practically unfeasible: the exact evaluation of complex aggregations, involving large portions of the data set, would require a prohibitive linear scanning. Such a high computational cost is not tolerable in the above scenarios where efficient and promptly data analysis is one of the main requirements. Moreover, a dramatic precision in range query answers is usually not needed; "good" estimates of query results often suffice to accomplish effectively many data analysis tasks. It is therefore very important to find approximate answers to range queries quickly in order to allow flexible, interactive ad effective data exploration.

In this chapter we provide a comprehensive overview of these issues. First in Sec. 1.2 we describe in detail some of the most relevant contexts where data is organized in a multi-dimensional fashion, and is commonly queried by issuing the computation of range aggregates. Then in Sect. 1.3 we discuss the central issue addressed in this thesis: the problem of computing "fast" approximate answers to aggregate queries in order to tackle the high computational cost of processing very large multi-dimensional data sets. Finally, in Sect. 1.4, we present a formal framework for dealing with multi-dimensional data by providing a unified formal abstraction of different types of multi-dimensional data sets and operators.

## 1.2 Application Scenarios

In this section we describe two relevant application contexts which, mostly among the others, have aroused the interest of research community on the

problem of efficiently querying multi-dimensional data: namely *On-Line Analytical Processing* in decision support systems and *selectivity estimation* for query optimization.

### 1.2.1 OLAP

OLAP (*On-Line Analytical Processing*) systems [37, 46, 17] are aimed at supporting the decision-making process of enterprizes, by allowing the end-user to pose complex queries on data gathered from daily business activity. In particular, OLAP services perform flexible, interactive and just-in-time data analysis, by extracting useful information from organization's data.

The architecture of a system providing OLAP services is shown in Fig. 1.1. Its main components are: a *data warehouse* (where data coming from heterogeneous sources are collected and integrated), an *OLAP server* (which extracts information from the data warehouse) and a client, providing interfaces for queries, data analysis, reporting and data mining. As shown in Fig.

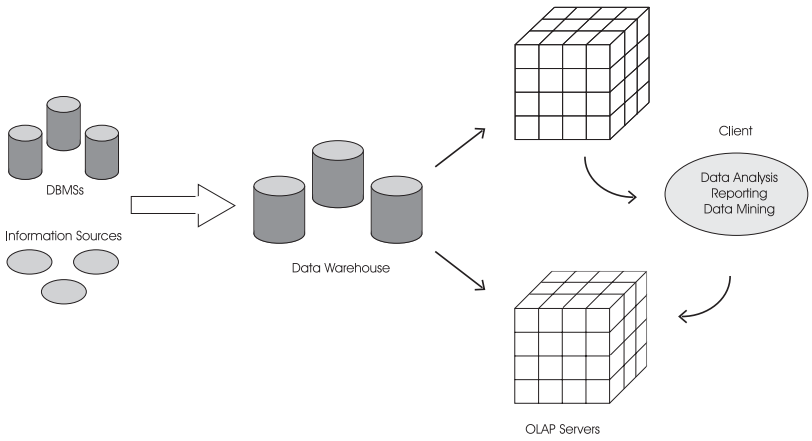


**Fig. 1.1.** Organization of an OLAP system

1.1, OLAP servers do not operate directly on operational data, that is data handled by traditional OLTP (*On-Line Transaction Processing*) systems. On the contrary source data are first extracted, pre-processed and integrated in the data warehouse. During extraction data are re-organized in order to obey a common schema, they are "cleaned" of possible errors and summarized in a less detailed description.

The data warehouse is kept separate from data sources mainly for the following reasons:

1. Data of interest : operational data are detailed, up-to-date, and they are directly retrieved from daily transaction processing; whereas data of in-

terest for supporting decision process are historical and summarized, basically static and integrated from heterogeneous sources.
2. Database design: data coming from operational sources usually follow an application-oriented conceptual model, while data undergoing the analysis process need to be classified by subject of interest.
3. Computational performances: the data warehouse is optimized to support OLAP operations such as aggregations of huge volumes of data and complex computations. On the contrary, operational databases are oriented to handle very frequent and detailed transactions; crucial issues in OLTP systems are concurrency control and data recovery.

After data extraction, the data warehouse keeps an active link with external sources: possible changes in source data are detected, by means of monitoring and integration tools, and propagated to warehouse data by periodical refreshing.
In addition, in some warehouse architectures, some other elements may be present, such as specialized views of the data or *data marts*, which collect data regarding special subjects of interest.

### OLAP datacubes

As already pointed out, OLAP applications rely on a different data organization w.r.t. traditional OLTP systems; this is also reflected by the adoption of a different model for data representation.
The "traditional" relational and object-oriented data models are the basis of the success of OLTP systems, which are able to manage transactions on data collections effectively, guaranteeing the integrity and consistency of information. These two models are especially well suited for those applications whose main objective is continuously updating the collected data and extracting "punctual" information.
On the contrary, the decision-making process uses knowledge extracted from large volumes of "historical data" (i.e. data which cannot be updated any more) by performing aggregation of bulks of tuples, rather than retrieving single data items. Thus, OLAP systems must be able to manage wide domains of data issuing large-scale aggregations efficiently.
DBMSs based on either the relational or the object-oriented data model would be too inefficient in issuing complex aggregation queries, despite the use of indexing structures. This makes traditional DBMSs unsuitable for providing reports (which can support the decision-making process) when the size of the collected data is very large.

It turns out that the the most effective data representation model for supporting flexible inspection and efficient aggregation is the multi-dimensional model. According to it data is organized into *multidimensional relations*, that is relations consisting of:

- one or more *measure attributes*, which represent the values to be aggregated and analyzed (such as the sale volume, the stock, the budget level, etc.)
- a set of $d$ *functional attributes* (or *dimensions*) specifying the context which the measure values refer to; for instance, the set of functional attributes $<$*product, year, area*$>$ may specify the product, the year and the area where a total sale volume has been recorded.

The set of functional attributes is a key for the multi-dimensional relation, thus each context is associated with a unique value of each measure attribute. This conceptual data modelling allows a multi-dimensional view of data: the domains of functional attributes define the dimensions of a multi-dimensional space, and data can be viewed as a set of measure values associated to points in this space.
In particular, given a multi-dimensional relation $R$ having (without loss of generality) one measure attribute and $d$ functional attributes (which will be assumed to be the first $d$ attributes of $R$), the domains of the functional attributes define a $d$-dimensional space, and data is associated to points in this space as follows.
For each point $\mathbf{x}$ of the $d$-dimensional space associated to $R$, let $x_i$ be its coordinate on the $i$-th dimension; then, in the multi-dimensional model:

- a measure value $v$ is associated to $\mathbf{x}$ if the tuple $< x_1, \ldots, x_d, v >$ belongs to $R$;
- otherwise a *null* value is associated to $\mathbf{x}$.

This mapping is shown in the two-dimensional example of Fig. 1.2.

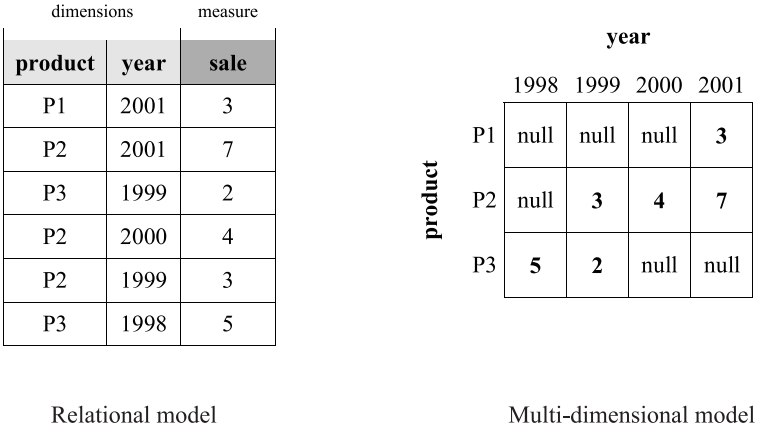


**Fig. 1.2.** Relational versus multi-dimensional data model

On the left-hand side of Fig. 1.2 a relation represents the sales of several products during different years. On the right-hand side of the figure

the same information is represented according to the multidimensional data model: *product* and *year* define the dimensions, whereas *sale* is the (unique) measure attribute.

In general, in the multi-dimensional model, information is logically organized into a multi-dimensional array (called *datacube*) where dimensions define different perspectives for viewing data (such as the *product* and *year* dimensions). Complex OLAP aggregation queries have a straightforward reformulation in terms of array operations in the multi-dimensional model. Consider, for instance, a datacube representing the sale volume of various products in different years and regions. As shown in Fig. 1.3, a query asking for the

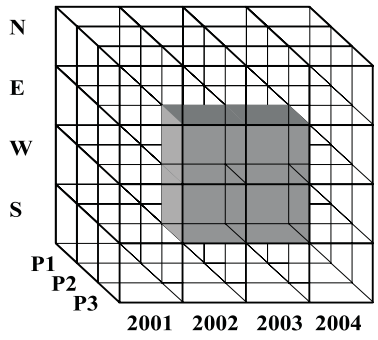


**Fig. 1.3.** Range query over an OLAP datacube

total sale of the product P3 during the years 2002-2003, in the East and West regions, consists in asking for the sum of the datacube values contained in the depicted multi-dimensional range.

On OLAP datacubes every form of aggregation (such as the evaluation of the sum or average of the values contained inside a range, or the computation of the number of occurrences of distinct values) can be translated into visiting sequentially a sub-array corresponding to the range of data that must be aggregated. Thus, in the OLAP context, the multi-dimensional model is suitable to support data exploration, as users can navigate information and retrieve aggregate data by simply specifying the ranges of the data domain they are interested in. Moreover, in order to answer an aggregation query, only cells involved by the query range need to be processed; whereas under the relational model, in the worst case, all the relational tuples (or their indexing structures, if available) must be accessed.

### 1.2.2 Selectivity Estimation

The computation of selectivity estimates for query optimization is another important task where the need to compute efficient aggregations on multi-dimensional data arises. Cost-based query optimizers in relational DBMSs

analyze statistics about the data stored in the database relations in order to compute efficient query execution plans [16, 73]. Each possible query plan has an execution cost depending on the order of execution of subsequent operators. In particular, the cost of each operator is determined by its selectivity, that is by the size (the number of tuples) of its result in the given query plan. Thus, a crucial task for analyzing and comparing query execution plans consists in estimating the selectivity of intermediate results.

The result size of a query involving a single attribute depends on the *frequency distribution* of that attribute in the database relation. The frequency distribution of an attribute $A$ on a relation $R$, whose schema contains $A$, can be represented as a vector, as shown in Fig. 1.4(b). The frequency vector contains, for each value $a_i$ in the domain of $A$, the number of tuples of $R$ whose $A$ value is equal to $a_i$.

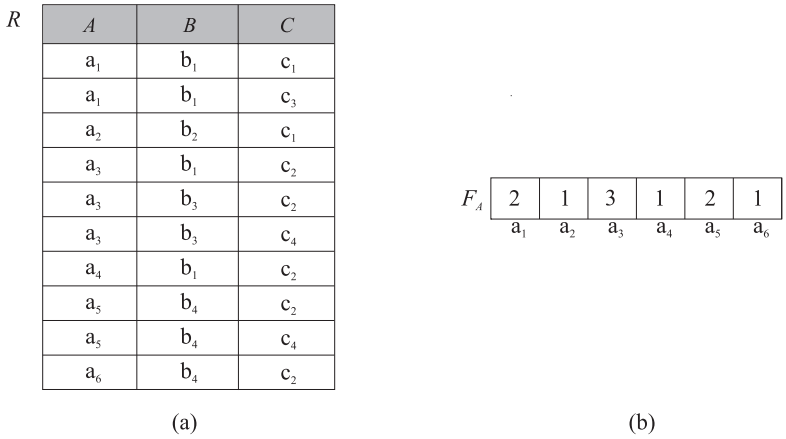


**Fig. 1.4.** (a) Relation $R$ (b) Frequency distribution of the attribute $R.A$

The selectivity of a selection query of type $Q = (a_i < R.A < a_j)$ is given by the sum of the frequencies represented in the domain interval $[a_i..a_j]$ of $A$'s frequency vector. For estimating the selectivity of queries involving multiple attributes, that is queries of the form $Q = (v'_1 < R.A_1 < v''_1) \wedge \ldots \wedge (v'_d < R.A_d < v''_d)$ , most commercial RDBMSs rely on the so called *attribute value independence assumption* [21, 74]. According to this assumption the fraction of database tuples where the attributes $< R.A_1, R.A_2, \ldots, R.A_d >$ take the tuple value $< v_1, v_2, \ldots, v_d >$ can be computed as the product of $d$ probabilities, each probability being the fraction of database tuples where attribute $R.A_i$ takes the value $v_i$, for $i = 1 \ldots d$. In other words, frequency distributions of single attributes are considered independent, and the query optimizer only keeps information about single-attribute distributions.
Unfortunately the attribute value independence assumption is not realistic, as

in real databases attributes may be strongly correlated. Thus, making this assumption may result in very poor selectivity estimates which can dramatically compromise the efficiency of the computed query execution plans.

Indeed, the selectivity of a multi-attribute predicate depends on the *joint frequency distribution* of the involved attributes [73]. The joint frequency distribution of the attributes $< A_1, A_2 \ldots, A_d >$ over the relation $R$ can be viewed as a $d$-dimensional array $\mathcal{F}$ whose dimensions represent the attribute domains, and whose cell with coordinates $< v_1, \ldots, v_d >$ stores the number of tuples of $R$ where $< A_1, \ldots, A_d >= < v_1, \ldots, v_d >$. Figure 1.5 shows the two-dimensional joint frequency distribution associated to attributes $< R.A,\ R.B >$ of the relation shown in Fig. 1.4 (a).

$F_{AB}$

| | | | | | | |
|---|---|---|---|---|---|---|
| $b_4$ | 0 | 0 | 0 | 0 | 2 | 1 |
| $b_3$ | 0 | 0 | 2 | 0 | 0 | 0 |
| $b_2$ | 0 | 1 | 0 | 0 | 0 | 0 |
| $b_1$ | 2 | 0 | 1 | 1 | 0 | 0 |
| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ |

**Fig. 1.5.** two-dimensional joint frequency distribution

The selectivity of the query $Q = (v'_1 < R.A_1 < v''_1) \ \wedge \ldots \wedge \ (v'_d < R.A_d < v''_d)$ is the sum of the frequencies contained in the multidimensional range $\langle [v'_1..v''_1], \ldots, [v'_d..v''_d] \rangle$ of $\mathcal{F}$. This is equivalent to computing a sum range query over the multi-dimensional data distribution represented by $\mathcal{F}$. Thus, the problem of answering aggregate range queries (in particular, sum range queries) over multi-dimensional data turns out to be crucial also in the context of selectivity estimation.

## 1.3 Approximate Query Answering

In the multi-dimensional model, the amount of data to be accessed for computing exact range query answers can be too large: at least all points lying into the query region must be examined and, in many application contexts, queries typically involve large portions of the database. In fact, typical queries on multi-dimensional data are not concerned with processing few values; they consist mainly in large-scale aggregations which analyze and compare huge volumes of historical data in order to extract from them useful information. Thus, computing exact aggregations can be a very hard computational task, as it requires a prohibitive linear scan of the multi-dimensional data set.

A possible solution to this problem is to change its target: instead of computing the exact answer, it is often convenient to estimate an approximate result that can be provided at a significantly lower computational cost, thus matching efficiency requirements. In fact, in most application scenarios the main aim is to retrieve aggregate data efficiently, possibly trading off the computational cost with the accuracy of query answers. Indeed, for either selectivity estimation and range query answering in OLAP systems, as well as for other tasks – such as statistical and scientific data analysis, sensor data analysis, window query answering in spatial databases – efficiently aggregating data within specified ranges of the domain is such a crucial issue, that high accuracy in query answers becomes a secondary requirement. Moreover, in these contexts, "rough" information about the data distribution often suffices for obtaining very useful reports.

For instance, in OLAP Decision Support Systems users are often concerned with performing preliminary explorations of the data domain, to find the portions where a more detailed analysis is needed. In this scenario, high accuracy in less relevant digits of query answers is not needed: "good" estimates of query results usually suffice to locate database regions containing relevant information [75, 80].
At the same time, the main requirement in OLAP systems, in order to guide effectively the decision-making process, consists in the flexibility and interactivity of data analysis. To this end, customized and timely reports must be provided fast, so that users are allowed to focus their explorations quickly and effectively.

Likewise, in the context of query optimization in RDBMSs, an effective query execution plan can be built on the basis of reasonably accurate estimates of the selectivity of intermediate queries. Thus, a dramatic precision in computing aggregate information about attribute frequencies is not needed.
Moreover, also in this context, fast computation of the aggregations is required. In fact, in query optimizers, efficient evaluation of query plans is mandatory: in order for query optimization to be effective, the run-time overhead for choosing the execution plan must be much less than the cost of executing the query itself.

In other application scenarios the need to provide approximate answers arises also from other issues than making query answering more efficient. For instance, in the context of data stream management, data is produced by "continuous" sources, such as sensors which send possibly endless streams of readings to a centralized processor. No bound can be given to the amount of information which goes through the data stream management system, whereas the available storage space for representing the received information is bounded. Thus, processed data items can be either discarded or partially archived, but exact information about all the stream content cannot be stored. Thus, its not feasible to compute exact answers to most common queries. On the other hand, in this context the efficiency requirement in query answering is quite strict: data received by the stream processor usually represent the readings

of environmental variables measuring the conditions of a monitored world; queries need to be evaluated very quickly, in order to allow the stream processor to react timely to possible critical events.

As we will discuss in detail in Chap. 2, one of the most effective approaches for providing fast approximate computation of range aggregates consists in keeping a compact summarized version of the multi-dimensional data set, to be queried instead of the original one.

## 1.4 A Formal Framework for Multi-dimensional Data

In this section we formalize the notion of multi-dimensional data and provide some basic notations which will be considered throughout the rest of this thesis.

A multi-dimensional data distribution will be represented as a $d$-dimensional array of integers with volume $n^d$. That is, without loss of generality, we assume that all dimensions of the array have the same size, and the domain of each dimension is the interval of cardinals $[1..n]$.
According to this representation of multi-dimensional data, values of data points are represented into cells of an array. The array cells which do not correspond to any data point contain the value 0 (that is the *null* value).

Given a $d$-dimensional data distribution $D$, a point in the multi-dimensional space of $D$ will be denoted as a $d$-tuple $\mathbf{x} = \langle x_1, \ldots, x_d \rangle$; following the usual array notation, $D[\mathbf{x}]$ will denote the array element stored in the cell of coordinates $\mathbf{x} = \langle x_1, \ldots, x_d \rangle$. The number of non-zero elements of $D$ will be denoted as $N$.
A *range* $\rho_i$ on the $i$-th dimension of $D$ is an interval $[l..u]$, such that $1 \leq l \leq u \leq n$. Boundaries $l$ and $u$ of $\rho_i$ are denoted by $lb(\rho_i)$ (*lower bound*) and $ub(\rho_i)$ (*upper bound*), respectively. The size of $\rho_i$ will be denoted as $size(\rho_i) = ub(\rho_i) - lb(\rho_i) + 1$.
A *block* $b$ of $D$ is defined as a multi-dimensional range, that is a $d$-tuple $\langle \rho_1, \ldots, \rho_d \rangle$, where $\rho_i$ is a range on the dimension $i$, for each $1 \leq i \leq d$. The ranges $\rho_1, \ldots, \rho_d$ are said *sides of* $b$. Informally, a block represents a "hyper-rectangular" region of $D$. A block $b$ of $D$ with all zero elements is said to be a *null block*.
The volume of a block $b = \langle \rho_1, \ldots, \rho_d \rangle$ is given by $size(\rho_1) \times \ldots \times size(\rho_d)$ and will be denoted as $vol(b)$.
Given two blocks of $D$, $b1$ and $b2$, the intersection $b3 = b1 \cap b2$ is a new block of $D$ such that the $i$-th side of $b3$ is the intersection of the $i$-th side of $b1$ with the $i$-th side of $b2$.
The point $\mathbf{x} = \langle x_1, \ldots, x_d \rangle$ belongs to the block $b$ (written $\mathbf{x} \in b$) if $lb(\rho_i) \leq x_i \leq ub(\rho_i)$ for each $i \in [1..d]$. A point $\mathbf{x}$ in $b$ is said to be a *vertex* of $b$ if, for each $i \in [1..d]$, $x_i$ is either $lb(\rho_i)$ or $ub(\rho_i)$.
Given a block $b$, we denote as $sum(b)$ ($avg(b)$, resp.) the sum (the average, resp.) of the array elements occurring in the block $b$.

A *range query* over a multi-dimensional distribution $D$ is defined by:

1. a multi-dimensional range $r$ of $D$ and
2. an aggregate operator $\mathcal{A}$

it asks for the computation of the aggregate operator $\mathcal{A}$ over all the values contained in the block $r$ of $D$.
In the following we will only consider *sum range queries*, that is queries whose aggregate operator $\mathcal{A}$ is the sum operator. Thus, the result of a sum range query over the range $r$ is defined as $sum(r)$. Our interest in this thesis will be focused on sum range queries as they are relevant in many application contexts (such as selectivity estimation). Anyway most of the results and techniques proposed for evaluating sum range queries can be extended to many other aggregate operators.

# 2

# Data Summarization: Existing Techniques

*A widely accepted approach to the problem of providing "fast" answers to aggregate queries consists in summarizing data into lossy synopses, and approximately evaluating range queries over summary data, rather than over raw ones. In this chapter we provide an overview of the main state-of-the-art techniques for summarizing multi-dimensional data:* wavelets, sampling *and* histograms. *We mainly focus on multi-dimensional histograms, which are the basis for the summary structures proposed in this thesis.*

## 2.1 Introduction

As discussed in Chap. 1, answering aggregate queries on very large multi-dimensional data sets can be computationally very expensive. A widely accepted approach to the problem of providing "fast" approximate answers to aggregate queries consists in summarizing data into lossy synopses, and evaluating range queries over summary data, rather than over raw ones. The amount of data corresponding to the range of a query which must be accessed in the summarized structure is much less than the number of elements that should be extracted from the original data set. This implies that computing query answers on summarized data can be much faster than evaluating them on detailed ones, provided that efficient techniques for estimating the answers directly on the summary structure are available. As expected, the loss of information due to summarization introduces some approximation in query answers but, as already discussed in Chap. 1, some approximation error is usually tolerated, in order to get fast access to data.

Many techniques for summarizing multi-dimensional data and evaluating range queries over their summarized representation have been proposed. In particular, several compression models which had been originally defined and implemented in different contexts have been used to this end. The most significant of these approaches is represented by *histograms*, which we describe in Sect. 2.2. In particular, we review the main theoretical results about histogram

construction, and we describe the best existing heuristics for constructing effective multi-dimensional histograms. We also provide a brief description of some other summarization techniques, namely *wavelets* (Sect. 2.3) and *sampling* (Sect. 2.4).

## 2.2 Histograms

Histograms have been initially designed in the context of selectivity estimation for summarizing single-attribute frequency distributions [55, 50, 72, 69, 16, 51], and are effectively applied in commercial systems (e.g. DB2, Oracle, Microsoft SQL server) for query optimization. They turned out to be quite inexpensive to store and to provide fast and low-error selectivity estimates. Multi-dimensional histograms have been introduced in [65] for approximating joint frequency distributions, and are extensively studied in the literature [73, 5, 15, 44].

In statistical databases [61] histograms represent a method for approximating probability distributions. Indeed, histograms can reach a surprising efficiency and effectiveness in approximating the actual distributions of data starting from summarized information. This has led the research community to investigate the use of histograms also in different fields such as range query answering in OLAP systems, scientific databases, data stream management, etc. [3, 70, 40, 39, 78].

A histogram on a multi-dimensional data distribution is obtained by partitioning the multi-dimensional domain into a set of hyper-rectangular blocks (called *buckets*) and then storing summary information for each block. The summary information associated to each bucket consists in some aggregations over the values occurring in the corresponding range, such as the sum of the values in that range, or the number of occurrences. The meaning of the aggregate value associated to each bucket depends on the data distribution which the histogram summarizes. For instance, when the histogram is constructed on a joint frequency distribution to support selectivity estimation, the sum of the values inside a bucket represents the number of relational tuples whose attribute values are inside the range of the bucket. Likewise, when the histogram is constructed on an OLAP datacube, each bucket stores the result of the computation of aggregate operators (such as SUM, COUNT, MAX, MIN, etc.) over the measure values occurring in the corresponding multi-dimensional range.

Figure 2.1 shows an instance of a histogram built on a two-dimensional data distribution, represented as a two-dimensional array. The histogram is obtained by partitioning the array into some rectangular buckets which do not overlap, and storing, for each bucket, the sum of the values it contains.

Aggregation queries issued on the original data distribution can be estimated on the histogram by exploiting the aggregate values stored in its buckets. In particular, the answer of a range query is estimated on the histogram

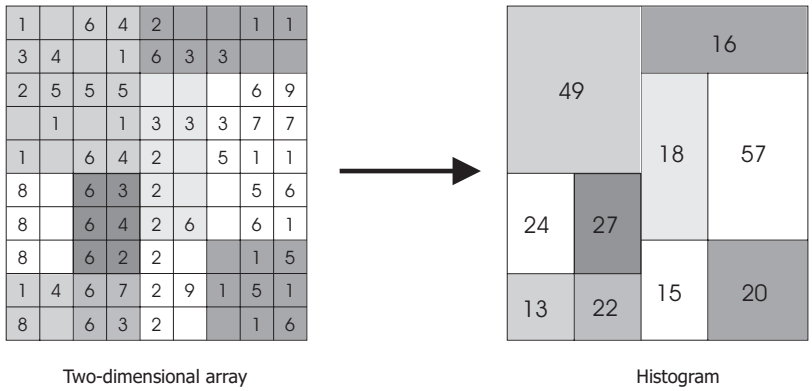


**Fig. 2.1.** A histogram built over a two-dimensional data distribution

by aggregating the contributions of all buckets: every bucket overlapping the query range is located and its contribution to the query answer is evaluated by performing suitable interpolation on its summary information.

In the rest of this thesis we will focus on *sum range queries* as they are relevant in many contexts, especially for problems related to selectivity estimation, which have inspired most of the work on histograms. Thus, we will only consider histograms whose buckets contain sums.

### 2.2.1 Query Estimation on Histograms

A sum range query over the multi-dimensional range $r$ can be evaluated on the histogram by summing the following quantities:

- for each bucket $b$ whose boundaries are completely contained inside $r$, the entire sum value stored in the bucket ($sum(b)$);
- for each bucket $b$ which partially overlaps the query range, an estimate of the portion of $sum(b)$ which lies onto the range of the query;

The latter contribution is in general approximate as the original data distribution inside a bucket cannot be reconstructed exactly from the summary information.

Several strategies have been studied in the literature for estimation within a bucket [55, 72, 74] but the most common technique adopted by histograms consists in performing linear interpolation: data distribution inside each bucket is assumed to be "homogeneous", that is each point inside the bucket is assumed to be associated to a data value equal to the average value inside the bucket. This is known as *Continuous Value Assumption - CVA* and, in the case that the histogram is constructed on a joint frequency distribution to support selectivity estimation, it corresponds to assuming that the joint distribution of attribute values is uniform [74].

Under CVA, the contribution of each bucket $b$ to the sum range query defined over the range $r$, is computed as $\frac{vol(b \cap r)}{vol(b)} \cdot sum(b)$. Figure 2.2 shows the computation of a query over the histogram introduced in the previous example. As shown in Fig. 2.2, the query range involves four buckets of the histogram: two of them give exact contribution, the other two give an approximate contribution computed by linear interpolation.

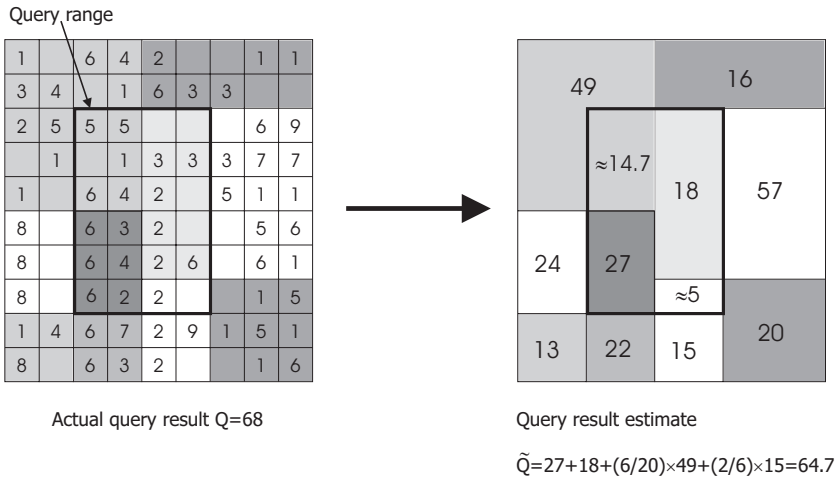


**Fig. 2.2.** Query estimation over a two-dimensional histogram using CVA

In general the contribution of the buckets which partially overlap the range of the query is approximate unless the original distribution of values inside these buckets is actually uniform. Indeed, very recent work [25] has shown that, in the one-dimensional case, linear interpolation gives good estimates even under statistical assumptions which are more general than the uniform distribution assumption. This implies that histograms may behave as good estimators even when the actual data distribution is far from being homogeneous. No extension of this result is currently available for multi-dimensional histograms.

### 2.2.2 Histogram Construction

As already pointed out, on the one hand, querying the histogram rather than the underlying original data reduces the cost of evaluating answers (as the histogram size is much less than the original data size); on the other hand, the loss of information due to summarization introduces some approximation when queries are estimated on the histogram. The effectiveness of a histogram is measured by evaluating the accuracy of estimating queries on it.

Obviously, the more the number of buckets of the histogram, the more detailed (therefore the more accurate) the approximation of the data distribution it may provide. On the other hand, the storage space available for the summarized representation of data is bounded; this defines a limit on the accuracy of query estimation on the histogram. Indeed, even histograms consisting of the same number of buckets may provide very different accuracy: as queries are estimated by performing linear interpolation on the aggregate values associated to the buckets, the more homogeneous the distribution inside the buckets involved in the query, the better the accuracy of the estimation. Therefore the effectiveness of a histogram depends on the underlying partition of the data domain, in that it depends on the degree of homogeneity that the partition provides inside buckets. Thus, different partitions of the multi-dimensional domain, even consisting in the same number of buckets, may lead to dramatically different errors in reconstructing the original data distribution, especially for skewed data.

Therefore, a crucial issue when dealing with histograms is finding the partition which provides the "best" accuracy in reconstructing query answers, given a storage space bound for the representation of the histogram. The problem of determining the "best" histogram for a given storage space bound has been investigated deeply. In [50] the concept of histogram optimality has been formalized by the definition of the *V-Optimal* histogram, which has been shown to minimize the expected error between the actual and approximate answer for several classes of queries.

The V-Optimality is defined on the basis of the *Sum Squared Error* (*SSE*) metric, a widely used metric for measuring the difference between two distributions. In particular, the *SSE* of a histogram consisting of the buckets $\{b_1, \ldots, b_\beta\}$, constructed on the data distribution $D$, is defined as $\sum_{i=1}^{\beta} SSE(b_i)$, where the *SSE* of a single bucket is given by $SSE(b_i) = \sum_{\mathbf{j} \in b_i} (D[\mathbf{j}] - avg(b_i))^2$ (by $\sum_{\mathbf{j} \in b_i}$ we denote that the summation is extended to all the elements of the array $D$ belonging to the block $b_i$). Given a space bound $B$, the histogram on $D$ which has minimum SSE among all histograms on $D$ whose size is bounded by $B$, is said to be *V-Optimal* (w.r.t. $B$).

In [52] the authors propose a polynomial time dynamic programming algorithm for finding the V-Optimal histogram on a one-dimensional data distribution; the algorithm working on a vector of size $n$ and a storage space bound $B$, runs in $O(n^2 \cdot B)$.

In [66] the data partitioning problem has been investigated also for multi-dimensional data. In particular, the authors present a taxonomy of different classes of partitions (see Fig. 2.3):

- *Arbitrary*: arbitrary partitions have no restriction on their structure
- *Grid-based*: grid-based partitions are built by dividing each dimension of the underlying data into a number of intervals, thus defining a grid on the data domain: the buckets of the histogram correspond to the cells of this grid.

- *Hierarchical*: hierarchical partitions are obtained by splitting the overall data domain into two or more sub-blocks and then by recursively partitioning the sub-blocks by hierarchical partitions.

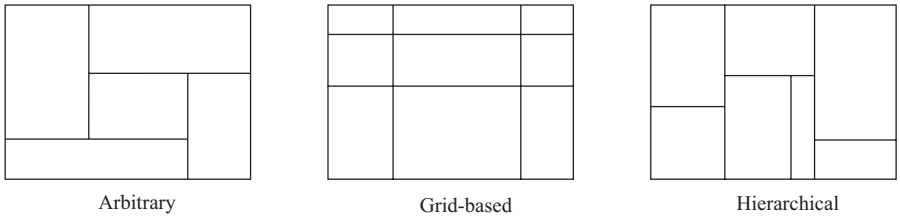


**Fig. 2.3.** Type of partitioning

Obviously, arbitrary partitions are more flexible than hierarchical and grid-based ones, as there are no restrictions on where buckets can be placed. However in [66] the problem of constructing the V-Optimal multi-dimensional histogram based on an arbitrary partition is shown to be NP-hard, even in the two-dimensional case. The authors also prove that this problem remains NP-hard even when the possible data partitions are restricted to obey a grid-based scheme.

On the contrary in [66], the problem of finding the V-Optimal multi-dimensional histogram based on a binary hierarchical partition is shown to be polynomial in the two-dimensional case; the polynomial upper bound is given by a dynamic programming algorithm which, on a two-dimensional data distribution of size $O(n^2)$ and a space bound $B$, runs in $O(B^2 \cdot n^5)$.
Indeed, the problem addressed in [66] is dual w.r.t. the problem of finding the V-Optimal histogram: it consists in finding the multi-dimensional histogram based on binary hierarchical partition which needs the smallest storage space and has an SSE value below a given threshold. But, as we will show in Chap. 4 (Sect. 4.3.1), the dynamic programming algorithm proposed in [66] can be easily adapted to the V-Optimal problem. In Sect. 4.3.1 we will also extend the latter polynomial complexity bound to the multi-dimensional case.
These results imply that it's unlikely to find efficient algorithms for constructing the V-Optimal partitioning: even under appropriate restrictions on the type of partition which make the problem tractable, only high-degree polynomial solutions are known.
In order to cope with this limitation, two kind of approaches have been proposed:

1. One approach consists in developing approximation algorithms for computing histograms which are provably close to the V-Optimal one. Most of these solutions operate on reducing the number of subproblems to be con-

sidered in the dynamic programming, thus achieving significantly faster running time w.r.t. the exact algorithms.
2. another approach consists in defining new heuristics for partitioning data which can be evaluated efficiently, usually according to greedy strategies. The aim of these partition techniques is to build a histogram whose buckets contain values with small "skewness", so that one can estimate a range query inside a bucket assuming that the data distribution is uniform, and thus performing linear interpolation.

Techniques which follow the first approach can be found in [52, 36, 66] for partitioning one- and two-dimensional data distributions. These techniques find sub-optimal solutions with provable quality guarantees; quality guarantees consist in bounds on the "distance" of the provided solution from the optimal one, but they do not provide any measure of the approximation error of the estimated range query answers.

Recently in [42] other error metrics than SSE have been introduced to define histogram optimality. They differ from the SSE which is an absolute error metric as they are a measure of relative error; that is these metrics are suitable functions of the relative errors between the actual data values and their approximate representation provided by the histogram. The problem of constructing optimal histograms w.r.t. these new error metrics has been analyzed in [42] only in the one-dimensional case, for which exact and approximate algorithms have been presented.

As to the other approach, a large number of heuristics for building effective histograms (which can be computed more efficiently than the V-Optimal one) have been proposed in the literature, both for one-dimensional data [72, 26, 51, 10] and for multi-dimensional data [65, 73, 5, 15, 44]. Most of these approaches are not based on arbitrary partitions.
In the rest of this section we briefly describe some of the most representative and effective among these techniques for building multi-dimensional histograms: namely *MHIST*, *MinSkew*, *GenHIST*, *ST-Histograms* and *STHoles*.

### 2.2.3 MHIST and MinSkew

MHIST-p [73] and MinSkew [5] histograms are constructed by means of a multi-step hierarchical partitioning strategy working as follows.

At the first step the histogram consists of a unique bucket whose range coincides with the data domain. At each of the following steps, the bucket of the histogram which is "the most in need of partitioning" (as explained below) is selected and split into a small number ($p$) of buckets along one of its dimensions. Thus, the selected bucket is removed, and its sub-blocks are inserted as new buckets. The algorithm ends when the space needed to store the histogram (that is both the boundaries of buckets and the sum values) saturates the available amount of storage space.

From the experiments in [73], it turns out that MHIST-2 (based on binary partitions) provides the best results, therefore in the rest of this thesis we will refer to MHIST-2 as MHIST.

The block to be partitioned is chosen as follows. First, the *marginal distributions* along every dimension are computed for each block. The marginal distribution of a block $b$ along the $i$-th dimension, denoted as $marg_i(b)$, is given by the "projection" of the internal data distribution of $b$ on the $i$-th dimension. That is, for a block $b = \langle \rho_1, \ldots, \rho_n \rangle$ the marginal distribution along the first dimension is obtained by computing, for each $x \in \rho_1$, the value $sum(\langle x..x, \rho_2, \ldots, \rho_n \rangle)$. Likewise on the other dimensions.

Figure 2.4 shows marginal distributions for a two-dimensional block.

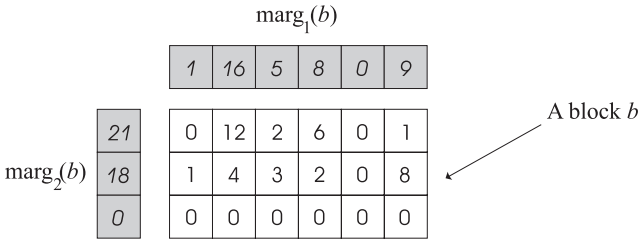


**Fig. 2.4.** Marginal distributions

Both MHIST and MinSkew adopt a greedy criterion based on marginal distributions of blocks to choose and split the "most in need of partitioning" bucket, at each step of the histogram construction.

In the construction of a MHIST histogram the so called *MaxDiff* criterion is adopted: the block $b$ to be split is the one which is characterized by a marginal distribution (along any dimension $i$) which contains two adjacent values $e_j$, $e_{j+1}$ with the largest difference w.r.t. every other pair of adjacent values, in any other marginal distribution of any other block. $b$ is split along the dimension $i$ by putting a boundary between $e_j$ and $e_{j+1}$.

MinSkew adopts a different criterion to select the block to be split and where to split it: it tries all possible splits along every dimension of every block; for each split on a block, it evaluates how much the SSE of the marginal distribution along the splitting dimension is reduced by the split. The block $b$ to be be split is the one where some split produces the maximum such reduction (maximum w.r.t all tried splits on all blocks). The split yielding the maximum reduction is then performed on $b$.

Indeed, instead of storing the actual boundaries of the buckets identified by the partition scheme, both MHIST and MinSkew represent each bucket by storing its *Minimal Bounding Rectangle (MBR)*, that is its minimal (hyper-) rectangular portion of the bucket containing all its non-null elements.

This implies that MHIST and MinSkew histograms do not define a partition of the data domain in the strict sense, as some null regions can be possibly covered by no bucket of the histogram.

MinSkew was originally introduced to deal with selectivity estimation in spatial databases (where data distributions are two-dimensional). In this scenario *MinSkew* first partitions the data domain according to a grid, and then builds a histogram as though each cell of the grid represented a single point of the data source. Moreover, it stores into each bucket some further aggregate data which are useful for spatial selectivity estimation [5].

### 2.2.4 GENHIST

*GENHIST* histograms were proposed in [44] as selectivity estimators for range queries on relations having real attributes. They do not define a partition of the data domain, as they allow bucket overlapping. The idea underlying GENHIST is to progressively locate regions of data which exhibit a non-homogeneous distribution w.r.t. contiguous ones.

At each step, GENHIST algorithm constructs a grid based on a $\xi$-regular partitioning of the multi-dimensional domain and chooses the cells of the grid having average density larger than their neighbors (the average density of a bucket being defined as the overall number of relational tuples in the bucket over the volume of the bucket). The data distribution is made smoother by randomly removing from each selected cell a number of tuples, so that the density of remaining tuples in the cell is the same as the average density of neighbor cells. Removed tuples are considered as belonging to a bucket whose boundaries coincide with those of the corresponding cell.

The value of $\xi$ defining the grid depends on the step of the algorithm. At the first step, an input parameter is used, and at the following steps its value is iteratively decreased, thus making the regular partitioning of data coarser: at each step the grid divides the data domain into about half as many cells as the previous iteration. This follows from the observation that the data distribution processed at the $i$-th step of the algorithm is smoother than that processed at previous steps, as high density peaks have been removed, thus larger buckets suffice to approximate data in detail.

The main difference w.r.t. traditional histograms is that buckets returned by GENHIST algorithm can overlap: buckets constructed at different steps belong to different granularity grids, so they have different size and may overlap. Each bucket, in general, does not contain information on all the tuples within the corresponding multi-dimensional range. tuples which lie in the intersection of many buckets are counted in the average density of only one of them: the bucket which contained the tuple during the iteration in which the tuple was removed.

As in classical histograms, on a GENHIST histogram the selectivity of range queries is estimated by assuming that within each bucket the data density is uniform and equal to the average density stored in the bucket. But,

as buckets represent "layers" of data, in the regions where two or more buckets overlap, the approximate data density is assumed to be the sum of the average data densities associated to all the overlapping buckets. Thus, the density of a region, as described by the histogram, is, in general obtained by summing different combinations of the stored densities. This feature is actually the advantage of the possible bucket overlapping. In fact, the number of regions described by different densities in the histogram can be actually larger than the number of buckets.

### 2.2.5 ST-Histograms

ST-Histograms [2] (self-tuning histograms) have been proposed, in the context of selectivity estimation, as a new incremental approach for histogram construction, alternative to traditional static data partitioning. The main difference w.r.t classical histograms is that ST-histograms are built without looking at the data distribution at all, but only "learning" it by exploiting feedback from the query execution: an initial histogram which describes "roughly" the frequency distribution is built and then progressively refined using information about the actual selectivity of range-selection queries. This is possible in the context of selectivity estimation, as histograms are used to provide preliminary estimations of the query selectivities, which are then computed exactly *on the database*. The actual query selectivities are available from the query execution engine with no extra-cost; thus, the histogram construction is performed incrementally, with very little overhead, by completely avoiding the cost of data scanning (which is the main shortcoming of traditional techniques for constructing multi-dimensional histograms). In more detail, ST-histograms are built according to three steps: *initialization*, *refinement* and *restructuring*, which work as follows:

1. *Initialization*: initially no query feedback is available, the only available information is assumed to be the attribute domains and the total number of tuples in the relation. The initial ST-histogram is built by regularly partitioning the multi-dimensional domain according to a grid, and by evenly dividing the total number of tuples among the equally-sized buckets (this corresponds to assume attribute-independence and uniformity of tuple distribution in the multi-dimensional domain).
   Alternatively, the initial ST-histogram can be built by exploiting possible pre-existing one-dimensional histograms on the attributes of the multi-dimensional joint frequency distribution. In this case each dimension is partitioned by following the bucket boundaries defined by the corresponding one-dimensional histogram. The frequency associated to each resulting bucket is computed from the frequencies associated to the corresponding one-dimensional buckets by assuming independence among attributes.
2. *Refinement*: unlike traditional histograms, in ST-histograms frequencies associated to buckets in general do not represent exactly the number of

tuples whose attribute values fall in the bucket range. Thus, every time a query is issued, the frequency value associated to the buckets of the ST-histogram overlapped by the query range is adjusted by exploiting the query result feedback. In particular, the *absolute estimation error* is computed as the difference between the actual query selectivity and the approximate selectivity evaluated on the ST-histogram. The "blame" for this error is distributed among the buckets involved by the query in proportion to their contribution to the query estimate. Each bucket frequency is then updated by summing to it the (signed) error portion it has been assigned (adjusted by a dumping factor to avoid oversensitive histograms).

3. *Restructuring*: bucket boundaries of the ST-histogram are periodically restructured in order to avoid buckets containing very skewed frequency values. Restructuring basically consists in "moving" some grid splits from smoother regions to more skewed regions of the frequency distribution. In particular, during the restructuring process the current grid partition is analyzed one dimension at a time. Each dimension corresponds to an attribute domain which is partitioned by the grid into a sequence of consecutive intervals; each interval is associated a $(n-1)$-dimensional "slice" of the frequency distribution. On each dimension a greedy strategy is adopted to locate runs of consecutive slices where corresponding buckets have "small" frequency differences. Slices in the same run are merged by merging corresponding buckets in them (by removing the grid splits which separate the slices in the same run). The buckets thus released are reinvested to perform new splits on the same dimension. In particular, the slices to be split are chosen as the ones having the highest marginal frequency on the current dimension (an input parameter is adopted to fix the percentage of slices to be split in any dimension). The number of splits removed in the merging process (on the same dimension) are distributed among the slices to be split in proportion to their marginal frequency. Each of these slices is then evenly partitioned on the current dimension by the number of assigned splits.

Histograms built without any direct knowledge of the overall frequency distribution are expected to provide less accurate estimates w.r.t traditional approaches; in fact, experimental results in [2] show that estimation performances of ST-histograms are comparable to those of MHIST-p only for data distributions without high skew.

### 2.2.6 STHoles

Also *STHoles* [15] histograms follow a query-oriented approach, as they are built incrementally, without examining the data set, but rather by using query result feedback to refine bucket definition. The main difference w.r.t. the idea underlying ST-histograms is the adoption of a new data partitioning scheme

which is far more flexible than grid partitioning and allows bucket nesting. This partitioning structure, as it will be discussed later, is particularly suited to allow histogram refinement by exploiting query result feedback.

In a STHoles histogram buckets are organized according to a tree structure. Each bucket $b$ has a rectangular bounding box, but its actual region is not necessarily rectangular, as it may contain disjoint rectangular holes. Each of these rectangular holes is the bounding box of some other histogram bucket, nested in $b$ and considered as a child of $b$. A frequency value is associated to each bucket, representing the number of tuples contained in the actual bucket region (which is obtained by excluding the regions covered by the holes of $b$ from the region enclosed in $b$'s bounding box).

The initial STHoles histogram can be either the empty histogram, or a pre-existing histogram, or a histogram consisting of one bucket (the root bucket associated to the whole multi-dimensional domain), if the overall number of tuples in the relation is available.

Then, each time a range selection query $q$ is issued on the database, both the frequencies and the layout of the histogram buckets are refined as follows: if $q$ exceeds the bounding box of the root bucket (or the histogram is currently empty) the root bucket is expanded (or created) so as to include $q$. Then the query execution is intercepted and, for each bucket $b_i$ of the current histogram, the exact number of tuples contained into $q \bigcap b_i$ (if not empty) is computed. Intuitively, this is a more precise information about the frequency distribution inside the bucket $b_i$, it says that a known portion of the total frequency associated to $b_i$ is concentrated in the narrower region $q \bigcap b_i$. The general idea is to pull out this region from $b_i$ by building a new bucket $b_n$, nested as a hole of $b_i$, and recording the number of tuples inside $q \bigcap b_i$.

Indeed, the region $q \bigcap b_i$ has not necessarily a rectangular shape, as it can partially overlap holes of $b_i$; thus, the candidature of $b_n$ as a hole of $b_i$ is not straightforward. The solution is to shrink the region $q \bigcap b_i$ along some of its dimensions as much as needed to exclude from it any possible partially overlapping hole of $b_i$. The resulting shrunk region, which will be referred to as $c$, has obviously rectangular shape and does not overlap partially any child of $b_i$ (it may possibly include completely some of them). The number of tuples $T_c$ contained in $c$ is estimated from the known number of tuples inside $q \bigcap b_i$, by assuming uniformity.

Then a new bucket $b_n$ is built having bounding box $c$ and recording the number of tuples $T_c$; $T_c$ is also subtracted from the frequency of $b_i$. The histogram structure is modified accordingly; that is $b_n$ is inserted as a child of $b_i$ and all $b_i$'s children completely contained inside $b_n$ are moved as children of $b_n$ (a few special cases of bucket insertion are handled differently in order to avoid space wasting).

Adding new buckets may make the histogram exceed the fixed storage space bound; when this is the case, some of the existing buckets are merged, so that some storage space is released. In order to choose the buckets to be merged, a *penalty* value is associated to each pair of buckets in the current

histogram. The penalty associated to the pair of buckets $< b_1, b_2 >$ measures the reduction of the histogram estimation accuracy occurring when merging $b_1$ and $b_2$.

When storage space has to be released, the least-penalty pair of buckets is chosen and merged in a single bucket. In particular, two kinds of possible merging are considered: *parent-child* merging, where a parent bucket is merged with one of its child buckets, and *sibling-sibling*, merging where two sibling buckets are merged, possibly including part of the parent volume.

Experiments presented in [15], show that STHoles histograms are competitive with the best multi-dimensional histogramming techniques for low dimensionalities. However the estimation errors of STHoles on queries exploring unseen regions cannot be avoided. In fact, a limitation of STHoles, as well as self-tuning histograms in general, is the fact that histogram accuracy strongly depends on the query workload, as the query results provide the only available views of the actual frequency distribution.

Nevertheless query-aware histograms can be a valid alternative to traditional ones in some application contexts. In fact, in some scenarios queries are posed on remote data sources which may not be accessible by the query optimizer, so that query results are the only information on the original data which can be exploited to build useful statistics. In other contexts data is very frequently updated, and updates involve huge portions of the database relations. Traditional histograms are not suited to describe such data, as they are basically static and need to be rebuilt from scratch when data changes. On the contrary query-aware histograms are intrinsically dynamic: when data is updated, as queries are issued, more and more information on the updated distribution is available, and the histogram is progressively changed accordingly.

## 2.3 Wavelets

Other approaches to the problem of summarizing multi-dimensional data are the wavelet-based ones. Wavelets [77] are mathematical transformations which define a hierarchical decomposition of functions (representing signals or data distributions) into a set of coefficients. They were originally used in different research and application contexts, like image and signal processing [53, 77, 77, 67]. Recent studies have shown the applicability of wavelets to selectivity estimation [64, 31], as well as to the approximation of both range queries [80, 81], and "general" queries [18] (using join operators) over multi-dimensional data distributions.

The summarized representation of a data distribution is obtained in two steps. The first step consists in applying a wavelet transformation to the data distribution, thus generating $N$ wavelet coefficients (the value of $N$ depends both on the size of the data and on the particular type of wavelet transform used).

Techniques such as the one presented in [81] apply the wavelet transform directly on the source data (the approach is mainly oriented at the I/O efficiency of the summarization process), whereas the technique described in [80] performs a pre-computation step: first, it generates an array storing all the partial sums of the source data, then it replaces each of its cells with its natural logarithm (it has been shown that the combination of the logarithm transformation with the approximation technique generally reduces the relative estimation error). Then, the wavelet transform is applied to this array.

After the application of the wavelet transform, no summarization is obtained (the number of wavelet coefficients is the same as the number of data items in the examined distribution), and no approximation is introduced, as the original data distribution can be reconstructed exactly applying the inverse of the wavelet transform to the sequence of coefficients.

The second step introduces summarization: among the $N$ wavelet coefficients, only the $m << N$ most "significant" ones are selected and stored, whereas the others are "thrown away", and their value is implicitly set to 0. For each selected coefficient, two numbers are stored: its value and its position. Thus, denoting the amount of available storage space as $B$, the number of coefficients which can be stored is given by: $\lfloor B/2 \rfloor$. The set of retained coefficients defines the summarized representation, called *wavelet synopsis*.

Issuing a query on the wavelet synopsis of the data set essentially corresponds to applying the inverse wavelet transform to the stored coefficients, and then aggregating the reconstructed (approximate) data values.

Several approaches to the problem of selecting the most effective $m$ wavelet coefficients for approximating the original data set have been proposed in the literature. The simplest one consists in a thresholding method [80, 81]: the $m$ retained coefficients are those with the largest absolute value. This criterion minimizes the overall root-mean-squared error in the data summarization, but cannot provide guarantees on the error of individual approximate query answers. In [31] the authors prove that unpredictable and widely varying errors arise in evaluating approximate query answers on the synopsis obtained using this thresholding method.

Different techniques for choosing an "effective" subset of wavelet coefficients based on a probabilistic framework are described in [39, 32]. The proposed probabilistic thresholding schemes assign each coefficient the probability of being retained according to its importance to the reconstruction of individual data values, and flips coins to select the synopsis. This technique provides quality guarantees on the error of individual queries. It minimizes the maximum relative error of the answers to all possible queries asking for single data values.

Other approaches are based on a more complex deterministic thresholding method [33] with the objective of optimizing relative or absolute maximum-error metrics, and providing error guarantees.

## 2.4 Sampling

Sampling-based summarization techniques represent data distributions by means of a set of random samples (called *sample synopsis*) whose size is smaller than the size of the original data. *Off-line* sampling pre-computes a sample synopsis which is updated when changes occur in the original data; queries are directly evaluated over the pre-computed samples [4, 34, 35].

On the contrary *on-line* sampling evaluates queries on a set of samples which are extracted at run-time among the set of all tuples which give contribution to the exact answers [45, 47].

The latter approach can be used for computing progressive answers to a query. That is, the collection of samples which are accessed for estimating the query answer can be iteratively enlarged (adding at least another sample), until the estimated error is "small" enough. Therefore, the answer to a query can be continuously refined.

The main advantage of the former approach (evaluating every query on the same pre-computed synopsis) is the response time, since the other technique could require several disk accesses to retrieve the samples which will be used for evaluating the answer. The latter approach has the obvious advantage that it can gauge the approximation of each query.

In [4] a technique for evaluating join operations using a pre-computed sample synopsis is also introduced. It is based on the idea of pre-computing samples of a small set of distinguished joins and storing them in the synopsis. This method works well only for queries with foreign-key joins which are known beforehand, and does not support arbitrary join queries over any schema.

# 3

# A Quad-tree-based Approach for Summarizing Two-dimensional Data

*In this chapter we propose a new summarization technique specifically designed for two-dimensional data. It is based on a quad-tree data partitioning combined with the use of* indices*, i.e. compact structures providing a very succinct description of portions of the original data. Experimental results show that the technique yields approximation errors much smaller than other general methods intended to work for any-dimensionality data (such as several types of multi-dimensional histogram and wavelets).*

## 3.1 Introduction

Among the existing summarization techniques, histogram-based ones turned out to be very effective in providing accurate estimates of range queries on one-dimensional data, and have been successfully applied in this context. On the contrary, estimation performances of state-of-the-art histograms for summarizing multi-dimensional data are rather poor. Also the estimation accuracy provided by other approaches, such as wavelet based ones, is far from being satisfactory in the multi-dimensional scenario.

In this chapter we design an *ad-hoc* summarization technique, specifically tailored on two-dimensional data. In fact, rather than searching for a general method which scales up to any dimension of data, we expect that, by exploiting the distinctive characteristics of restricted application domains, higher accuracy can be achieved. Following this direction, we consider specifically two-dimensional data, which are of particular interest in a number of applications:

1. *selectivity estimation in spatial databases* [5, 62]: this problem consists in evaluating the number of objects (triangles, rectangles, etc.) which intersect a query rectangle in a 2-D space. The 2-D space can be approximated as a two-dimensional histogram whose buckets are associated to the *spatial density* of the corresponding regions, i.e. the number of objects which overlap the range;

2. *evaluation of direction queries* [79]: it can be shown that estimating the number of objects which are related by some direction relation (*north*, *north-west*, etc.) to another object can be translated into evaluating 2-D range queries. The opportunity of issuing the query on summary data arises from the fact that the amount of data is often huge, and thus it would be unfeasible to get an exact answer accessing the original tuples;
3. *querying sensor databases*: As we will discuss in Chap. 6, data generated by a set of linearly ordered sources (sensors) can be represented in a 2-D fashion, where one dimension is associated to the sources and the other one to the generation time. The need to aggregate information arises from the fact that sensors produce data which cannot be stored in detail, as they consist of a continuous and "infinite" flow of readings.

Our approach for summarizing two-dimensional data is closely related to histograms: the data distribution is partitioned into buckets, by adopting a quad-tree based hierarchical partition scheme (i.e. by recursively splitting blocks of data into four equally-sized sub-blocks), and aggregate information is stored for each block. Intra-bucket estimation is further enhanced by storing, in addition, some very compactly encoded description of the actual data distribution inside buckets (called *index*, and specifically designed for two-dimensional data).

The chapter is organized as follows. We first provide a formal definition of the problem of summarizing two-dimensional data distributions. In particular, we present the quad-tree based partition schema and introduce the notion of *Quad-Tree Summary* (QTS), the summary structure obtained by applying our partition schema on a given data distribution. We adopt the well known SSE metric for measuring the effectiveness of a QTS w.r.t. the issue of estimating range queries accurately, and discuss the problem of finding the optimal QTS w.r.t. this metric (called *V-Optimal Quad-Tree Summary*). We then present a polynomial time solution for finding the V-Optimal quad-tree summary. The resulting cost function is $O(B \cdot n^2 \cdot log\, n)$ where $B$ is the available storage space for the summary structure, and $n^2$ is the size of the two-dimensional array. As $n$ is in general very high, we cannot afford such a cost, therefore we present a greedy algorithm with cost $O(B \cdot logB)$ for computing a sub-optimal solution, which can be effectively run also on very large two-dimensional data sets (as $B$ is much smaller than $n^2$).

Finally, we enhance the estimation accuracy of the proposed greedy algorithm by introducing *indices* for describing the data distribution inside buckets. In fact, in order to achieve a better estimation of range queries over aggregate data, instead of finding a solution closer to the optimal one, we improve the estimation accuracy inside each block: linear interpolation is replaced with a more accurate technique by exploiting the low-resolution representation of intra-bucket data distribution provided by indices. The experiments we have carried out over a large number of syntectic two-dimensional data sets show that our greedy algorithm combined with indices exhibits much better performances than state of the art "general purpose" approaches.

## 3.2 Summarizing Two-dimensional Data: the Problem

In this section we present our quad-tree based partition schema for summarizing two-dimensional data distributions, and introduce the notion of *Quad-Tree Summary* (QTS) (the summary structure obtained applying our partition schema on a given data distribution). We adopt a well known metric (the SSE) for measuring the effectiveness of a QTS w.r.t. the issue of estimating range queries accurately, and discuss the problem of finding the optimal QTS (called *V-Optimal Quad-Tree Summary*) w.r.t. this metric.

The basic idea underlying the choice of a simple hierarchical schema for partitioning the array of data arises from the following remarks. The main drawbacks limiting the effectiveness of any approach producing an arbitrary partition (i.e. with no constraints on where the boundaries of the blocks can be placed) are related to the amount of space required to store the partition itself. In fact, the advantage of these approaches is that they can derive a very "good" partition avoiding that large differences of values occur in each block of the partition. But, as the space bound is generally "small", this advantage is often deleted by the cost of representing the structure of the summarized data (i.e. the boundaries of the blocks), so that only partitions consisting of a few blocks can be stored.

A way for solving the above problem consists in finding partitions whose representation can be done compactly. A naive solution consists in dividing each dimension into equally sized ranges (*equi-range partition*). In this way, no additional information has to be stored for representing the partition itself, and thus partitions consisting of much more blocks (w.r.t. the arbitrary approach) are obtained. Unfortunately, blocks produced using this technique do not fit any requirement about the variance of contained values, since the partition technique is done "blindly".

Our partition technique is neither too blind nor too arbitrary: it fits the actual distribution of data (defining finer-grain blocks where data is more skewed) and, at the same time, it needs not use a large amount of space for storing the partitioning structure.

### 3.2.1 Quad-Tree Partition

We are given a two-dimensional data distribution $D$ which will be viewed as a two-dimensional array of size $n \times n$.
Given a range $\rho_i$ on the dimension $i$ of $D$, we denote by $lh(\rho_i)$ (*left half*) the range $[lb(\rho_i)..\lfloor (lb(\rho_i) + ub(\rho_i))/2 \rfloor]$ on $i$, and by $rh(\rho_i)$ (*right half*) the range $[\lfloor (lb(\rho_i) + ub(\rho_i))/2 \rfloor + 1..ub(\rho_i)]$.
Given two ranges $\rho_1, \rho_2$ defining the block $b = \langle \rho_1, \rho_2 \rangle$ of $D$, a *quad-split block* of $b$ is any block $\langle \rho'_1, \rho'_2 \rangle$ such that $\rho'_i$ is either $lh(\rho_i)$ or $rh(\rho_i)$, for $i = 1, 2$. Observe that, for a given block $b$ of $D$, there are 4 different quad-split blocks; each of these correspond to one of quadrants of $b$.

Given a block $b = \langle \rho_1, \rho_2 \rangle$ of $D$, we denote by $Q(b)$ the 4-tuple $\langle b_1, b_2, b_3, b_4 \rangle$ such that $b_1 = \langle lh(\rho_1), rh(\rho_2) \rangle$, $b_2 = \langle rh(\rho_1), rh(\rho_2) \rangle$, $b_3 = \langle lh(\rho_1), lh(\rho_2) \rangle$, and $b_4 = \langle rh(\rho_1), lh(\rho_2) \rangle$. $Q(b)$ is said the *quad-split partition* of $b$. Often, with a little abuse of notation we refer to $Q(b)$ as a set. Informally, the quad-split partition of $b$ contains the four quadrants of $b$.

Given a $4-$ary tree $T$, we denote by $Nodes(T)$ the set of nodes of $T$, by $Root(T)$ the singleton containing the root of $T$, $Leaves(T)$ the set of leaf nodes of $T$. We define $Der(T)$ as the set of nodes of $T$ $\{p \in Nodes(T) \mid \exists q \in Nodes(T) \wedge$p is the right-most child node of q$\}$.

A *quad-tree partition* $QTP(D)$ of $D$ is a $4-$ary tree whose nodes are blocks of $D$ such that: 1) $Root(QTP(D)) = \langle 1..n, 1..n \rangle$, 2) for each $q \in Nodes\ (QTP(D)) \setminus Leaves(QTP(D))$ the tuple of children of $q$ coincides with its quad-split partition $Q(q)$, and 3) for each $q \in Nodes\ (QTP(D)) \setminus Leaves(QTP(D))$ it holds that $sum(q) \neq 0$.
Given a quad-tree partition $P$, we denote the set $\{p \in Leaves(P) \mid sum(p) = 0\}$ by $Null(P)$ . From condition 3 in the definition of quad-tree partition, it follows that $Null(P)$ contains all the nodes with sum zero, as there cannot exist any internal node whose sum is zero. Moreover, we denote by $Store(P)$ the set $Nodes(P) \setminus \{Der(P) \bigcup Null(P)\}$.

### 3.2.2 Quad-Tree Summary

A *quad-tree summary* $QTS(D)$ of $D$ is a pair $\langle P, S \rangle$ where $P$ is a quad-tree partition of $D$ and $S$ is the set of pairs $\langle p, sum(p) \rangle$ where $p \in Store(P)$. That is, each pair in $S$ denotes a range of $D$ (belonging to $Store(P)$) and the value of the corresponding sum. Informally, $Store(P)$ represents the set of nodes whose sum must be necessarily stored, whereas $Der(P)$ contains the nodes whose sum can be evaluated using the sums of nodes in $Store(P)$. More precisely, for each node $q$ in $Der(P)$, $sum(q) = sum(p) - \sum_{u \in Children(p) \setminus \{q\}} sum(u)$, where $p$ is the parent node of $q$ and $Children(p)$ represents the set of child nodes of $p$. That is, the sum of a node $q$ which is the right-most child of a node $p$ can be evaluated by summing the values of the three siblings of $q$, and subtracting this sum from the value of $p$.

Given a quad-tree summary $QTS = \langle P, S \rangle$ of $D$, $P$ is said the *partition-tree* of $QTS$, and we denote it by $Part(QTS)$; $S$ is said the *content set* of $QTS$ and we denote it by $Cont(QTS)$. A node $b$ of $P$ is said a *terminal block* if $b \in Leaves(P)$, a *non-terminal block* otherwise.

With a little abuse of notation, throughout the rest of the chapter we will adopt the shortcuts $Root(QTS)$, $Nodes(QTS)$, $Leaves(QTS)$, $Store(QTS)$, $Null(QTS)$ denoting respectively: $Root(Part(QTS))$, $Nodes(Part(QTS))$, $Leaves(Part(QTS))$, $Store(Part(QTS))$ and $Null(Part(QTS))$.

In Fig. 3.1 a graphical representation of a quad-tree summary is reported. White nodes are those of the set $Der(P)$. In the same figure we have also depicted the graphical representation of the partition $P$.

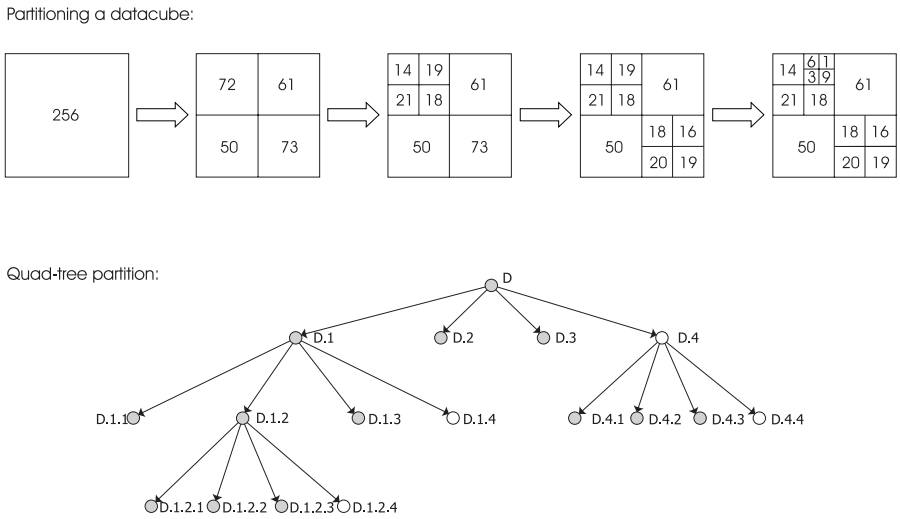


**Fig. 3.1.** A quad-tree based partition

The storage space for a quad-tree summary $QTS = \langle P, S \rangle$ is the space taken by the representations of $P$ and $S$. $P$ can be represented by a string of bits: each pair of bits is associated to a node of $P$ and indicates whether the node is a leaf or not (i.e. whether the block corresponding to the node is split or not) and, if it is a leaf, whether it is null or not. In particular: (1) $\langle 0, 0 \rangle$ means non null terminal node, (2) $\langle 0, 1 \rangle$ means null terminal node, (3) $\langle 1, 1 \rangle$ means split node (i.e. non terminal node). Observe that it remains one available configuration (i.e., $\langle 1, 0 \rangle$) which will be used in Sect. 3.4.2. Clearly, in case (2), the sum of the block is not kept, thus saving 32 bits. Therefore, the string representing the partition $Part(QTS)$ contains $2 \cdot |Nodes(QTS)|$ bits.

The storage space needed for representing $S$ is the space occupied by the set $\{s_i | \exists p_i \in Store(P) \wedge \langle p_i, s_i \rangle \in S\}$. Therefore, $S$ can be efficiently stored by means of an array of size $|Store(P)| \cdot 32$ bits, whose elements are the sums calculated inside each block in $Store(P)$. The order in which the sums are stored in this array expresses their connection to the blocks in $Store(P)$.

Figure 3.2 reports the strings representing the sums and the structure of the quad-tree of Fig. 3.1. Thus, the overall storage space for a quad-tree summary $QTS$ is $size(QTS) = 2 \cdot |Nodes(QTS)| + |Store(QTS)| \cdot 32$. Often, throughout the chapter, we refer to $QTS(D)$ also as *the summarized representation* of the array $D$.

### 3.2.3 Estimating Range Queries on a Quad-tree Summary

We focus our attention on sum range queries. Let $r$ be the range of the query. The estimate is computed by visiting the quad-tree underlying the QTS start-

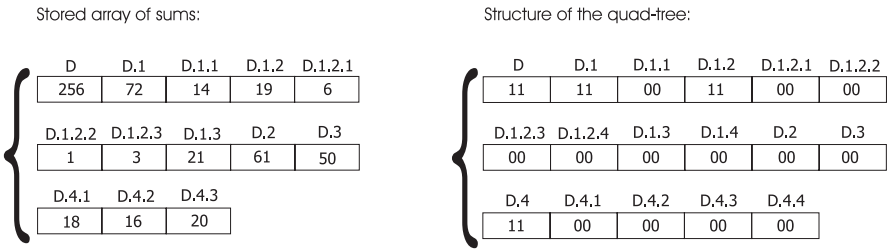


**Fig. 3.2.** Quad-tree structure encodement

ing from its root (which corresponds to the whole data array). When a node is being visited, three cases may occur:
1. *the range corresponding to the node is external to* $r$: the node gives no contribution to the estimate;
2. *the range corresponding to the node is entirely contained into* $r$: the contribution of the node is given by its sum;
3. *the range corresponding to the node partially overlaps* $r$: if the node is a leaf, linear interpolation is performed for evaluating which portion of the sum associated to the node lies onto $r$. Otherwise, the contribution of the node is the sum of the contributions of its children, which are recursively evaluated.

The crucial issue is how to build $QTS(D)$ in order to maintain satisfactory accuracy in (range) query estimation. This is the matter of the next section.

### 3.2.4 V-Optimal Quad-Tree Summary

Let $B$ be the available storage space for representing the quad-tree summary of $D$. The value of $B$ defines the set of all the quad-tree summaries $QTS(D)$ such that $size(QTS(D)) \leq B$. Among this set we could choose the best partitioned array w.r.t. some metrics. The metrics certainly has to be related to the approximation error, but a number of possible ways to measure the error of a summarized representation of a data distribution can be adopted. Following a well-accepted approach in literature, we measure the "goodness" of the summarized representation of a data distribution by using its SSE. We extend the definition of SSE of a histogram (Sect. 2.2.2) to our summary structure. Formally, given a quad-tree summary $QTS$: $SSE(QTS(D)) = \sum_{q_i \in Leaves(QTS)} SSE(q_i)$, where, given a terminal block $q_i$, we recall that: $SSE(q_i) = \sum_{j \in q_i} (D[j] - avg(q_i))^2$, Clearly, the smaller $SSE(QTS(D))$, the "better" the representation provided by $QTS(D)$, in terms of accuracy. We extend to $QTS$s the definition of V-Optimality introduced for general multi-dimensional histograms:

**Definition 3.1.** *Given a two-dimensional data distribution $D$, and a storage space bound $B$ we call* V-Optimal Quad-Tree Summary *on $D$ (for the space bound $B$) a Quad-Tree Summary $QTS^*(D)$ such that, $size(QTS^*(D)) \leq B$*

*and* $SSE(QTS^*(D)) = min_{H \in \mathcal{Q}}\{SSE(H))\}$*, where* $\mathcal{Q}$ *is the set of all Quad-Tree Summaries on* $D$ *with space bound* $B$*.*

## 3.3 Summarizing Two-dimensional Data: Exact and Greedy Solutions

In this section we address the problem of finding the optimal quad-tree summary w.r.t. the SSE metric (V-Optimal QTS). We study the complexity of computing the optimal solution, drawing the conclusion that it is unfeasible on large data distributions. Therefore, we propose a greedy algorithm finding a sub-optimal solution efficiently. We remark that all the complexity results which are provided in this section and in the following one are given under the assumption that, for any block $p$ of a partition, the time complexity of evaluating $sum(p)$ as well as $SSE(p)$ is constant. In other words we are assuming to pre-compute and keep enough information to derive the sum and the SSE of each block of a partition. For instance, given the array of partial sums $F$ of size $n \times n$ such that $F[i,j] = sum(\langle 1..i, 1..j \rangle)$, the sums of the elements of a block of any size can be computed accessing 4 elements of $F$ (see Sect. 4.3.2 and [52] for more details).

**Theorem 3.2.** *Given a two-dimensional data distribution* $D$ *of size* $O(n^2)$*, a* V-Optimal Quad-Tree Summary $QTS^*(D)$ *with space bound* $B$ *can be computed in time* $O(B \cdot n^2 \cdot log\ n)$*.*

*Proof. (Sketch)* Finding the V-Optimal QTS can be reduced to a particular instance of CSP (Constrained Shortest Path [84]). In more detail, if we construct a complete quad-tree whose leaves correspond to the single elements of $D$, we can define a $s-t$ graph having the nodes of this complete quad-tree as vertices (besides the source and destination vertices). The edges of the graph are established in a way such that all the paths from $s$ to $t$ correspond to the borders of all the possible quad-trees partitioning the data array. Moreover, each vertex of the graph is weighted by either a cost and a resource consumption. The cost of a vertex represents the SSE of the corresponding block of $D$, whereas the resource consumption of a vertex represents its contribution to the storage consumption of each QTS having this vertex as a leaf node.

Intuitively enough, the problem of finding the V-Optimal QTS turns into the problem of finding the minimum cost path from $s$ to $t$ having an overall resource consumption bounded by $B$ (the CSP problem). The CSP problem can be solved in $O(m \cdot B)$, where $m$ is the number of edges of the graph. It can be easily shown that our $s-t$ graph has $m = O(n^2 log\, n)$. □

In theory the algorithm could work in exponential time, as $B$ is not bounded. In practice $B = O(n^2)$ since the size of the summarized array (i.e $B$) must be much less than the size of the original one (i.e. $32 \cdot n^2$, assuming that

each value of the array is represented using 32 bits). Therefore, from Theorem 3.2 we have that a V-Optimal Quad-Tree Summary can be computed in polynomial time.

***Remark***. We recall that finding an arbitrary partition (i.e. with no constraints on its structure) minimizing SSE is a NP-Hard problem, as shown in [66] (see Sect. 2.2.2). Our problem is tractable because of the restrictions on the type of partition underlying the summary. Optimization problems on quad-tree partitions, similar to ours, have been studied in the context of motion estimation for video compression. The main difference w.r.t. our optimization problem is the resource bound given on the allowed partitions. In particular, the problem of finding the optimal quad-tree partition w.r.t. a large class of metrics (including SSE) with a bound on the number of leaves has been studied in [63], and an algorithm working in time $O(n^4 \cdot logn)$ has been proposed. However the problem addressed in the latter work is even simpler than ours, since our bound is more "general". That is, our bound on the space available to represent the QTS could be reduced to a bound on the number of leaves only if we were guaranteed that the partition did not identify any null block. Moreover, our approach can work better than $O(n^4 \cdot log\, n)$, as $B$ is often much smaller than $32 \cdot n^2$. We point out that the problem of minimizing the SSE is tractable even with less restricted types of partition, such as binary hierarchical partitions (i.e. hierarchical partitions corresponding to binary trees which are not constrained to split blocks into equal sub-blocks). The problem of finding the binary hierarchical partition which minimizes SSE has been shown to be polynomial in [66], but its bound (i.e. $O(B^2 \cdot n^5)$) is even greater than ours. Indeed, the problem investigated in the latter work is rather different from ours, as the hierarchical partition is not constrained to split blocks into equal sub-blocks; moreover, the issue of re-investing the storage space saved by efficiently representing null blocks is not addressed.

Nevertheless, for large data distributions, the bound $O(B \cdot n^2 \cdot log\ n)$ makes finding the optimal solution too inefficient. In order to reach the goal of minimizing the SSE, in favor of simplicity and speed, we propose a greedy approach, accepting the possibility of obtaining a sub-optimal solution. Our approach works as follows. It starts from the quad-tree summary whose partition tree has a unique node (corresponding to the whole $D$) and, at each step, selects a leaf of the quad-tree (according to some greedy criterion) and applies the quad-split partition to it. Every time a new split is produced, 4 new born nodes are added to the quad-tree. If any of such nodes corresponds to a block with sum zero, we save the 32 bits used to represent the sum of its elements. Anyway, recall that only 3 of the 4 nodes have to be represented, since the sum of the remaining node can be derived by difference, by using the parent node. A number of possible greedy criteria for choosing the block which is the most in need of partitioning can be adopted. For instance, we can choose the block with maximum SSE, or the block whose split produces the

maximum global SSE reduction, or the block with maximum sum, and so on. However, after comparing all the above mentioned greedy criteria by means of experiments, we have chosen to use the greedy criterion of the maximum SSE.
The resulting algorithm is the following:

---

**Greedy Algorithm 1**
Let $B$ be the storage space available for the summary.
**begin**
$Q := \langle\ P_0, \{\langle\langle 1..n, 1..n\rangle, sum(\langle 1..n, 1..n\rangle)\rangle\}\ \rangle$;
$B := B - 32 - 2$;
*// 32 bits are spent for the sum of the whole array*;
*// 2 bits are spent for recording the structure of the partition*;
**while** $(B > 0)$
Select a node $p$ in $Leaves(Q)$ such that:
$SSE(p) = max_{q \in Leaves(Q)}\{SSE(q)\}$;
Let $Q^+(p)$ be the set of nodes obtained by splitting $p$ and selecting its non null children except the right-most one;
$B := B - |Q^+(p)| \cdot 32 - 4 \cdot 2$;
**if** $(B \geq 0)$
$Q := \langle\ Split(Part(Q), p)\ ,$
$Cont(Q)\ \cup \bigcup_{r \in Q^+(p)}\{\langle r, sum(r)\rangle\ \}\ \rangle$;
*// Q is modified according to the split of p*;
**end if**
**end while**
**return** $Q$;
**end**

---

Therein: (i) $P_0$ is the partition tree containing only one node (corresponding to the whole array), and (ii) the function *Split* takes as arguments a partition tree $P_i$ and a leaf node $l$ of $P_i$, and returns the partition tree obtained from $P_i$ by inserting $Q(l)$ (i.e., the quad-split partition of $l$) as children nodes of $l$.

**Theorem 3.3.** *Given a two-dimensional data distribution $D$ of size $O(n^2)$, a space bound $B = O(n^2)$, Greedy Algorithm 1 computes a Quad-tree Summary $QTS(D)$ with space bound $B$ in time $O(B \cdot logB)$.*

## 3.4 Improving the Greedy Solution using Indices

In this section we propose a technique for improving the estimation accuracy of the QTS returned by Greedy Algorithm 1. This is done by storing, beside the overall sum of the elements occurring in each block, further information helping us in reconstructing range queries inside the blocks. The use of this further information, in general, allows us to get a more accurate estimate than

that provided by linear interpolation, as, after partitioning the array of data, we are not guaranteed that blocks contain so uniform data distributions that CVA can be effectively applied. This information is encoded into a 64-bits compact structure (called *index*), and consists of an approximate description of the actual data distribution contained in a block. That is, instead of trying to improve the "quality" of the partition w.r.t. the optimal one, we concentrate on improving intra-block estimation, replacing linear interpolation with a a more accurate technique.

In the following, we first define the structure of indices and describe how they can be used for estimating range queries inside blocks. Then we show how to embed indices in a QTS, thus obtaining a new summary structure called *Indexed Quad-Tree Summary* (IQTS). Finally, we provide an efficient greedy algorithm producing an IQTS and analyze its complexity.

### 3.4.1 Indexing Two-dimensional Data Blocks

Experience acquired in [10, 13] for one-dimensional histograms inspired us in storing approximate sums of internal sub-blocks of a given block $b$ in an hierarchical fashion, by means of a quad-tree partition with a fixed depth.

We define three index types with different organization of sub-blocks, so that we may select the index which better approximates data distribution inside a block: (1) *2/3LT-index*, which is suitable for distributions with no strong asymmetry, (2) *2/4LT-index*, which is oriented to biased distributions, (3) *2/p(eak)LT-index* which is designed for capturing distributions having a few high density peaks. The three types of index use the same amount of storage space, 64 bits, and are next described in detail.

#### 2/3LT-index

The block is partitioned into 4 sub-blocks (its quadrants) which in turn are further divided into other 4 sub-sub-blocks. The aggregation leads to the balanced tree index with 3 levels of Fig. 3.3 where nodes correspond to sub-blocks of the block $Q$ of the figure. The node at level 1 (i.e. corresponding to the sum of the entire block) is explicitly represented by 32 bits (with no approximation). As for the other levels, the simplest approach would be to store the sums corresponding to the grey nodes of the index, whereas the other sums can be derived by difference, using the parent node. We instead use a different storing scheme. At level 2, we keep only approximated sums of regions $A_Q$, $B_Q$ and $C_Q$, as shown in Fig. 3.4.

From the sums of $A_Q$, $B_Q$ and $C_Q$, we can derive sums corresponding to all the nodes of the level 2 of the index:

$$sum(Q_1) = sum(C_Q)$$
$$sum(Q_2) = sum(A_Q) - sum(C_Q)$$
$$sum(Q_3) = sum(B_Q) - sum(C_Q)$$
$$sum(Q_4) = sum(Q) - sum(A_Q) - sum(B_Q) + sum(C_Q)$$

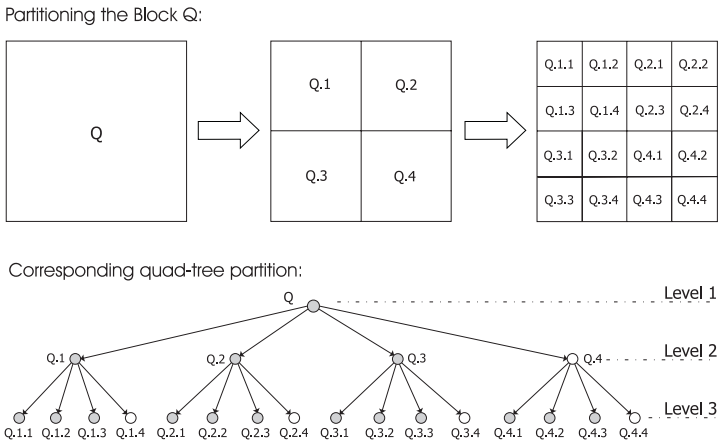


**Fig. 3.3.** 2/3LT-index

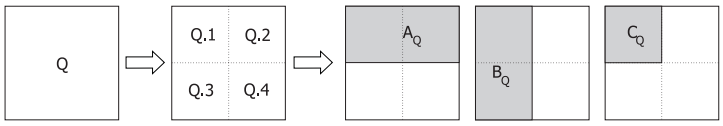


**Fig. 3.4.** $A_Q$, $B_Q$, $C_Q$ regions inside a block

We adopt the same storage scheme at level 3. Thus, for the sub-block $Q_i$ (for $1 \leq i \leq 4$), we keep the sums of $A_{Q_i}$, $B_{Q_i}$ and $C_{Q_i}$, respectively. An example of index for a block with sum 50 is shown in Fig. 3.5.

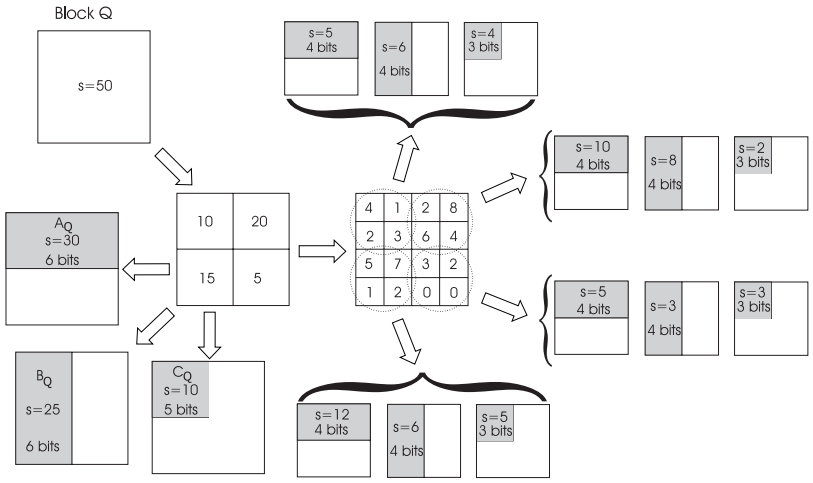


**Fig. 3.5.** Building a 2/3LT-index

The figure also indicates the number of bits used for each sub-block sum. The overall storage space of 64 bits is used as follows. For the region $A_Q$ we use a string of 6 bits, denoted by $L_{sum(A_Q)}$, which represents the sum of $A_Q$ as a fraction of the sum of $Q$. More precisely, $L_{sum(A_Q)} = round\left(\frac{sum(A_Q)}{sum(Q)} \cdot (2^6-1)\right)$. The approximate value $\overline{sum}(A_Q)$ of $sum(A_Q)$ can be obtained from $L_{sum(A_Q)}$ as $\frac{L_{sum(A_Q)}}{2^6-1} \cdot sum(Q)$. We do the same for the region $B_Q$, as the two regions have the same size and we thus expect, on the average, that they contain sums of the same magnitude. For the region $C_Q$ we decrease by 1 the number of employed bits, and exploit them for representing the sum of $C_Q$ as a fraction of the minimum between the sum of $A_Q$ and the sum of $B_Q$ — let $AB_Q$ be this minimum. The 5-bit string associated to $C_Q$ thus contains $L_{sum(C_Q)} = round\left(\frac{sum(C_Q)}{sum(AB_Q)} \cdot (2^5-1)\right)$, and consequently the approximate value $\overline{sum}(C_Q)$ of $sum(C_Q)$ can be computed as $\frac{L_{sum(C_Q)}}{2^5-1} \cdot \overline{sum}(AB_Q)$. The reduction of 1 bit (w.r.t. $A_Q$ and $B_Q$) for representing the sum of $C_Q$ is justified by the observation that the size of $C_Q$ is in the average half of that of $A_Q$ and $B_Q$) and then we expect a sum in $C_Q$ that is half of their sums. For the lowest level, we use 4 bits for $A_{Q_i}$ and $B_{Q_i}$, and 3 bits for $C_{Q_i}$ (for $1 \leq i \leq 4$) – see Fig. 3.5.

In sum, the final storage space balance is $6+6+5+4\cdot(4+4+3) = 61$ bits. Observe that (some of) the 3 remaining bits to two words will result useful for identifying the type of index being used — this issue will be detailed later on.

### 2/4LT-index

This index is unbalanced, and tries to capture "heterogeneous" data distributions. A 2/4LT-index is built as follows. First the block is partitioned into four quadrants. Then, the two quadrants containing the most skewed data distributions are further split. In particular, the more skewed quadrant is split into 16 equally sized portions, and the other one into four quadrants. For instance, the index in Fig. 3.6 describes a block where the region $Q_1$ contains a very skewed data distribution, the region $Q_4$ is less skewed than $Q_1$, whereas the regions $Q_2$ and $Q_3$ contain quite uniform distributions. Observe that, for a given block, there are $2 \cdot \binom{4}{2}$ possible different kinds of 2/4LT-indices (depending on which pair of quadrants is chosen to assign resolution 4 and 3, respectively). Thus, we need 4 bits to identify one 2/4LT-index among all possible ones. The overall storage space required for a 2/4LT-index is $6 + 6 + 5 + 2 \cdot (4 + 4 + 3) + 4 \cdot (2 + 2 + 1) = 59$ bits. Thus, with 4 of the 5 remaining bits we identify the kind of 2/4LT-index. We will see in the following that the remaining bit is enough to identify 2/4LT-index among the other ones (i.e. 2/3LT-index and 2/pLT-index).

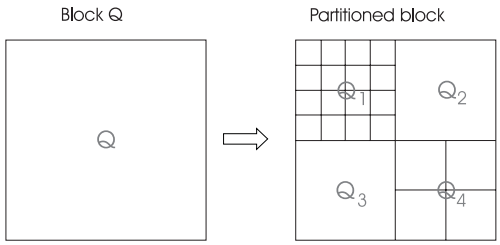


**Fig. 3.6.** The structure of a 2/4LT-index

### 2/pLT-index

This index is designed to capture the case of a few density peaks concentrated in a quadrant of the block $Q$ to which the index is applied. In particular, the 2/pLT-index has levels 1 and 2 as the 2/3LT-index. Moreover, the node of the level 2 corresponding to the quadrant with maximum SSE, say $Q_i$, is associated with 43 bits recording the sum of 5 sub-blocks of the quadrant $Q_i$. Such 5 sub-blocks are the 5 sub-blocks with highest sum among all sub-blocks obtained from $Q_i$ by dividing its sides into 8 equi-size ranges. The 5 sub-blocks are identified by 5 pairs of 3-bit coordinates (each pair, consisting of 6 bit, identifies one sub-block among the 64 possible ones). Each of the 3 highest sums is represented by 3 bits, whereas each of the other 2 sums is represented by 2 bits. Therefore, we have $5 \cdot 6 = 30$ bits for representing the coordinates and $3 \cdot 3 + 2 \cdot 2 = 13$ bits for the sums. Thus, the overall storage space spent for the "internal" description of $Q_i$ is 43. The overall storage space of the 2/pLT-index is 60 bits, obtained by summing 43 bits to the bits needed for representing the level 2, that are $6 + 6 + 5 = 17$. The remaining 4 bits are used, as we shall see, to identify the 2/pLT-index among the other kinds, and to identify the quadrant which is provided with the internal description.

### Overview of the representation of 2/nLT-indices

The 64 bits of the indices are organized as a 2-words frame $F$: 2/3LT-index requires 61 bits, 2/4LT-index 59 bits and 2/pLT-index requires 60 bits. The frame has a header consisting of $F[1..3]$ (i.e. the first 3 bits of $F$) for the 2/3LT-index, of $F[1..5]$ for the 2/4LT-index, and of $F[1..4]$ for the 2/pLT-index. This header is exploited to encode the structure of the index. In particular, $F[1] = 1$ identifies the 2/4LT-index, $F[1..2] = \langle 0, 0 \rangle$ identifies the 2/3LT-index, and $F[1..2] = \langle 0, 1 \rangle$ identifies the 2/pLT-index. For the 2/3LT-index no further information has to be encoded about the structure of the index, so that the bit $F[3]$ is not used. For the 2/4LT-index, the remaining 4-bits portion of the header $F[2..5]$ is used to identify which kind of 2/4LT-index (among the 12 possible ones) is contained in $F$ (that is, which is the quadrant with resolution 4 and which is the quadrant with resolution 3). Finally, for the 2/pLT-index,

the remaining 2-bits portion of the header $F[2..4]$ identifies the quadrant to which the 43-bits internal description is associated.

### Evaluation of a query using a 2/nLT-index

The contribution of a block equipped with an index to the estimate of a range query can be done by visiting the quad-tree underlying the index in the same way as it has been shown in Sect. 3.2.3. Linear interpolation is used on the leaves of the quad-tree. In particular, for a 2/pLT-index, the contribution of the node containing the peaks is evaluated by summing the contribution of every peak inside the query range with the contribution of the remainder portion of the node.

We remark that the choice of the hierarchical partition underlying indices aims to reduce numerical approximation errors deriving from the use of few bits for representing the sums. It can be shown that it produces smaller errors than a *flat* partitioning of the block into a number of sub-blocks [13]. Indeed, in the latter case, the sum of a single sub-block should be represented as a fraction of the entire sum of the block. On the contrary, using the hierarchical approach, the sum corresponding to a node is represented as fraction of the sum of its parent, which, in general, has a smaller value than the sum of the entire block.

### Selection of the best 2/nLT-index

We select the best 2/nLT-index for a block $q$ on the basis of the actual distribution of data inside the block, by measuring the approximation error carried out by the index. As a measure of the approximation error of an 2/nLT-index $I$ we use:

$$\epsilon_q(I) = \sum_{i=1}^{64} (sum(b_i) - sum_I(b_i))^2 \tag{3.1}$$

where $b_i$ represents the $i-$th (among 64 ones) sub-block of $q$ obtained by dividing its sides into 8 equal-size ranges, and $sum_I(b_i)$ represents the estimation of the sum of elements occurring in $b_i$ which can be done by using the 2/nLT-index $I$ and the knowledge of $sum(q)$ (recall that the estimation of such sums can be done as explained above). For a block $q$, we choose the 2/nLT-index $I$ with minimum $\epsilon_q(I)$. Indeed, instead of computing $\epsilon_q(I)$ for all the possible indices of $q$, we consider as candidates only three indices: the 2/3LT-index, the 2/4LT-index which investigates the two quarters of $q$ with largest variance (describing the quarter with maximum variance using the highest resolution) and the 2/pLT-index which investigates the quarter with largest variance. We denote such a set of indices associated to the block $q$ by $Best(q)$. It could be easily shown that choosing the best 2/nLT-index can be done with a number of operations constant w.r.t. the size of the block, under the assumption of Sect. 3.3.

### 3.4.2 A Greedy Algorithm using 2/nLT-indices

In this section we show how the use of the already described 2/nLT-indices can be embedded in the construction of a new type of quad-tree summary in order to improve the estimation accuracy. The new summary structure is called *Indexed Quad-Tree Summary* (IQTS). The basic idea is to embed indices in a quad-tree summary equipping each terminal block with an appropriate 2/nLT index (to be used in intra-block interpolation). Indeed, the application of the 2/nLT-index does not necessarily give a real benefit (w.r.t. CVA) to the estimation accuracy. There might be nodes such that the application of the 2/nLT-index fails. For instance, for a block containing a perfectly uniform data distribution, the use of indices introduce some approximation in the estimates (as values are stored with some loss of precision in every type of index), whereas CVA provides exact answers. To detect such nodes, we need to define how we measure both the error carried out by the (best) 2/nLT-index and the error produced by CVA estimation (used in absence of 2/nLT-index). Concerning the former type of error we evaluate: $\epsilon_q^{nLT} = min_{I \in Best(q)} \epsilon_q(I)$, where $\epsilon_q(I)$ is defined by (3.1) in Sect. 3.4 and $Best(q)$ is defined in Sect. 3.4, just after (3.1). Concerning CVA estimation we define: $\epsilon_q^{CVA} = \sum_{i=1}^{64} (sum(b_i) - sum_{CVA}(b_i))^2$, where $q$ is a non null block of $D$, $b_i$ represents the $i-$th (among 64 ones) sub-block of $q$ obtained by dividing its sides into 8 equal-size ranges, and $sum_{CVA}(b_i)$ represents the estimation of the sum of elements occurring in $b_i$ done by using CVA and the knowledge of $sum(q)$. We evaluate, for each node $q$, the difference: $Benefit_q = \epsilon_q^{nLT} - \epsilon_q^{CVA}$, which will be used for deciding whether $q$ has to be equipped with an index. We expect, in most of the cases, a negative value of $Benefit_q$ as result. But for some blocks, it might happen that CVA works better than the indexing technique, and thus we would have a positive value for the above difference. If so, we decide not to store any index for the block, in order to save storage space that can be reinvested in further splits.

The two bits (per node) describing the structure of the quad-tree summary (see Sect. 3.2.2) can now be used to encode every possible type of node. In particular: (1) $\langle 0,0 \rangle$ means non null terminal node without any 2/nLT-index, (2) $\langle 0,1 \rangle$ means null terminal node, (3) $\langle 1,0 \rangle$ means non null terminal node with 2/nLT-index, and (4) $\langle 1,1 \rangle$ means split node (i.e. non terminal node). Recall that, in case (2), the sum of the block is not kept, saving thus 32 bit. Given an Indexed Quad-Tree Summary $IQTS$, the definitions of the sets $Nodes(IQTS)$, $Store(IQTS)$, $Leaves(IQTS)$ and $Null(IQTS)$ can be trivially extended from the ones given in the context of Quad-Tree Summaries. Also the notion of $SSE(IQTS)$ is analogous to the one introduced for Quad-Tree Summaries. Moreover, we denote by $IndLeaves(IQTS)$ the set $\{q \in Leaves(IQTS) | \ Benefit_q < 0\}$, i.e. the set of leaves which are equipped with an index. The overall storage space for an Indexed Quad-Tree Summary $IQTS$ is: $size(IQTS) = 2 \cdot |Nodes(IQTS)| + |Store(IQTS)| \cdot 32 + |IndLeaves(IQTS)| \cdot 64$. A greedy algorithm for the construction of an indexed quad-tree summary

can be obtained from the one building a QTS by taking into account the storage consumption of the indices needed on the terminal blocks, at each partition step. In more detail, at each step the new algorithm performs a new split. Then, the following quantities are subtracted from the amount of currently available storage space $B$: 1) the space needed to represent the sums of the children of the current node $p$, 2) the space needed to equip every child $q$ with $Benefit_q < 0$ with an index, 3) the space needed to update the quad-tree structure. Finally, 64 bits are added back to $B$ if $Benefit_p < 0$, i.e. if, at some previous step, the space needed to equip $p$ with an index was subtracted from $B$. The resulting algorithm is the following:

---

**Greedy Algorithm 2**

Let $good(C)$ be a function receiving a set of blocks $C$ and returning the maximal subset $S$ of $C$ such that $\forall q \in S\ Benefit_q < 0$ (i.e. the application of a 2/nLT-index is fruitful).

Let $B$ be the storage space available for the summary.

**begin**
  $Q := \langle\ P_0, \{\langle\langle 1..n, 1..n\rangle, sum(\langle 1..n, 1..n\rangle)\rangle\}\ \rangle$;
  $B := B - 32 - |good(\{\langle 1..n, 1..n\rangle\})| \cdot 64 - 2$;
  // 32 bits are spent for the sum of the entire array;
  // $|good(\{\langle 1..n, 1..n\rangle\})| \cdot 64$ counts the bits spent to
  // apply the 2/nLT-index to the entire array;
  // *2 bits are spent to record the structure of* $P_0$;
  **while** $(B \geq 0)$
    Select a node $p$ in $Leaves(Q)$ such that:
      $SSE(p) = max_{q \in Leaves(Q)}\{SSE(q)\}$;
    Let $Q^+(p)$ be the set of nodes obtained by splitting $p$ and
    selecting its non null children except the right-most one;
    $B := B - |Q^+(p)| \cdot 32 - |good(Q(p))| \cdot 64 +$
        $+|good(\{p\})| \cdot 64 - 4 \cdot 2$;
    **if** $(B \geq 0)$
      $Q := \langle\ Split(Part(Q), p)\ ,$
          $Cont(Q)\ \cup \bigcup_{r \in Q^+(p)}\{\langle r, sum(r)\rangle\ \}\ \rangle$;
    **end if**
  **end while**
  Apply the most suitable 2/nLT-index to
  each block in $good(Leaves(Q))$;
  **return** $Q$;
**end**

---

where (i) $P_0$ is the partition tree containing only one node (corresponding to the whole array), and (ii) the function *Split* takes as arguments a partition tree $P_i$ and a leaf node $l$ of $P_i$, and returns the partition tree obtained from $P_i$ by inserting $Q(l)$ (i.e., the quad-split partition of $l$) as children nodes of $l$.

**Theorem 3.4.** *Given a two-dimensional data distribution $D$ of size $O(n^2)$, Greedy Algorithm 2 computes an Indexed Quad-tree Summary $IQTS(D)$ with space bound $B = O(n^2)$ in time $O(B \cdot logB)$.*

***Remark***. We point out that the solution provided by Greedy Algorithm 2 is even worse (w.r.t. the SSE metric) than the one computed by Greedy Algorithm 1. In fact, the space needed to keep indices reduces the number of nodes of the partition that can be stored within a given space bound, thus reducing the number of splits that can be performed while partitioning data. As each split reduces the overall SSE of the partition (SSE is a super-additive metric), the partition computed by Greedy Algorithm 2 has an SSE which is never smaller than the one of the solution returned by Greedy Algorithm 1. However, as we will show in the next section, the index-based approach shows better performances w.r.t. greedy QTS, allowing us to draw the conclusion that *it is better to invest some space for adding quantitative data* (thus improving intra-block estimation), *rather than to use all the available space for producing partitions with finer-grain blocks.*

## 3.5 Experimental Results

In this section we present some experimental results about the accuracy of estimating sum range queries on quad-tree summaries, comparing our method with the state-of-the-art techniques in the context of summarized data. In particular, we compare our technique with the histogram-based technique MHIST, and with the wavelet-based techniques proposed respectively in [80] and [81] (see Chap. 2), which will be denoted respectively as WAVE1 (working on the partial sum data array) and WAVE2.
In order to prove that the usage of 2/nLT-indices improves the accuracy of quad-tree summaries, we have tested both $QTS$ and $IQTS$. The experiments were conducted at the same storage space. We next present the test bed used in our experiments.

### 3.5.1 Measuring Approximation Error

We denote the exact answer to a sum query $q_i$ as $S_i$, and the estimated answer as $\tilde{S}_i$. The *absolute error* of the estimated answer to $q_i$ is defined as: $e_i^{abs} = |v_i - \tilde{S}_i|$. The *relative error* is defined as: $e_i^{rel} = \frac{|S_i - \tilde{S}_i|}{max\{1, S_i\}}$. Our definition of relative error is the same as the one used in [81], and is slightly different from the classical one, which is not defined when $S_i = 0$.

The accuracy of the various techniques has been evaluated by measuring the average absolute error $\| e^{abs} \|$ and the average relative error $\| e^{rel} \|$ of the answers to the range queries belonging to the following query sets:
1. $QS_1$: it contains all the sum range queries defined on a range s.t. one of its vertices coincides with a vertex of $D$;

2. $QS_2(\Delta_1, \Delta_2)$: it contains the sum range queries defined on all the ranges of size $\Delta_1 \times \Delta_2$ (here the vertex of the query does not necessarily coincide with a vertex of $D$);
3. $QS_1^+$ and $QS_2^+(\Delta_1, \Delta_2)$: they contain all the queries belonging to $QS_1$ and, respectively, $QS_2(\Delta_1, \Delta_2)$, whose answer *is not* null;
4. $QS_1^0$ and $QS_2^0(\Delta_1, \Delta_2)$: they contain all the queries belonging to $QS_1$ and, respectively, $QS_2(\Delta_1, \Delta_2)$, whose answer *is* null.

Query sets $QS_1^+$ and $QS_2^+$ have been introduced since it can be meaningful to treat the approximation error of a query whose exact answer is zero differently w.r.t. the error of a query with non-zero answer. That is, when the exact answer is zero, the absolute error of the estimated answer is a good metrics for the approximation error: if $S_i = 0$ it is meaningful to check whether $\tilde{S}_i$ is small or not. Thus, we use different ways for measuring approximation errors: by computing $\| e^{rel} \|$ over $QS_1$ and $QS_2$, we "put together" the relative errors of queries whose answer is not zero with the absolute errors of queries whose answer is zero. By computing $\| e^{rel} \|$ over $QS_1^+$, $QS_2^+$, and $\| e^{abs} \|$ over $QS_1^0$, $QS_2^0$ we consider the case $S_i = 0$ separately from the case $S_i \neq 0$. In the following, the values of the average relative error and the average absolute error evaluated on a query set $QS$ will be denoted, respectively, as: $\| e^{rel}(QS) \|$ and $\| e^{abs}(QS) \|$.

### 3.5.2 Synthetic Data Sets

The synthetic data sets used in our experiments are similar to those of [81]. The synthetic data generator populates $r$ rectangular regions of a two-dimensional array of size $d \cdot d$, distributing into each of them a portion of the total sum value $T$. The size of the dimensions of each region is randomly chosen between $l_{min}$ and $l_{max}$, and the regions are uniformly distributed in the two-dimensional array. The total sum $T$ is partitioned across the $r$ regions according to a Zipf distribution with parameter $z$. To populate each region, we first generate a Zipf distribution whose parameter is randomly chosen between $z_{min}$ and $z_{max}$. Such a distribution contains as many values as the number of cells inside the region. Next, we associate these values to the cells in such a way that the closer a cell to the centre of the region, the larger its value is. Outside the dense regions, some isolated non-zero values are randomly assigned to the array cells.

### 3.5.3 Results

Experiments on synthetic data show the superiority of our technique w.r.t. other methods. We consider the accuracy of the various methods w.r.t. to several parameters, i.e. the *storage space* available for the summarized representation, the *skew* inside each region, the size of the queries (using query set $QS_2$), and we consider both dense and sparse data distributions. The storage

space is expressed as the number of 32 bits integers which are available for the summarized representation of the array.

**Storage space**. We considered several sparse data arrays of size $2000 \cdot 2000$ generated by setting $l_{min} = 25$, $l_{max} = 70$, $z_{min} = 0.5$, $z_{max} = 1.5$, containing about 23000 non zero cells, and dense data arrays of size $500 \cdot 500$, with $l_{min} = 90$, $l_{max} = 130$, $z_{min} = 0.5$, $z_{max} = 1.5$, containing about 97000 non zero cells. The accuracy of the estimates w.r.t. the storage space (i.e. the number of 32 bit words used for representing the summarized data) is depicted in Fig. 3.7 (sparse data) and Fig. 3.8 (dense data). We used a logarithmic scale for $\| e^{rel}(QS_1^+) \|$ and $\| e^{abs}(QS_1^0) \|$, and a linear scale for $\| e^{rel}(QS_1) \|$. In particular, in the picture representing the average relative error on $QS_1$ of Fig. 3.8, only $QTS$ and $IQTS$ are compared, as the errors produced by the other methods are out of scale.

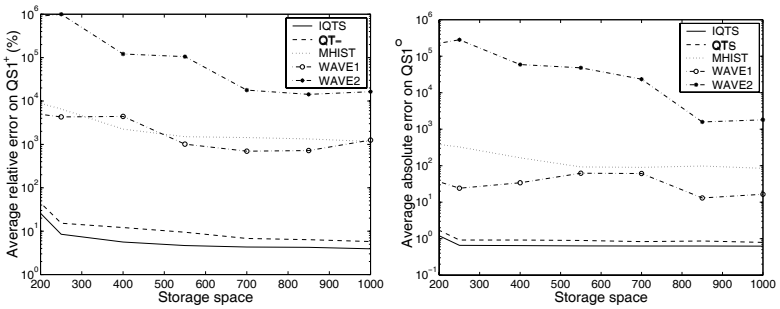


**Fig. 3.7.** Errors of estimates for sparse data

**Skew inside regions**. We considered sparse data arrays of size $2000 \cdot 2000$ with $l_{min} = 25$, $l_{max} = 70$, obtained for different values of the skew inside each region. The accuracy of the estimation (measured using $\| e^{rel}(QS_1^+) \|$) w.r.t. the different skew values is depicted in the picture on the left-hand side of Fig. 3.9. Interestingly, all the techniques are more effective in handling small and large levels of skew than intermediate ones ($z = 1.5$). When the skew is high, only a few values inside each region are very frequent, so that the dense regions contains mainly these values. MHIST and QTS group these values into the same blocks causing small errors, and the wavelet decomposition applied in these regions generates a lot of coefficients with value zero. Analogously, when the skew is small, the frequencies corresponding to different values are nearly the same and thus the data distribution is quite uniform, so that the CVA assumption generates small errors.

**Size of the query**. We considered the same sparse and dense arrays used for measuring the accuracy w.r.t. the storage space, and evaluated the accuracies of the various techniques for different query sizes on the summarized representations obtained using 1600 4-byte integers. In the picture on the right-hand side of Fig. 3.9, the value of $\| e^{rel}(QS_2^+(\Delta, \Delta)) \|$ obtained on

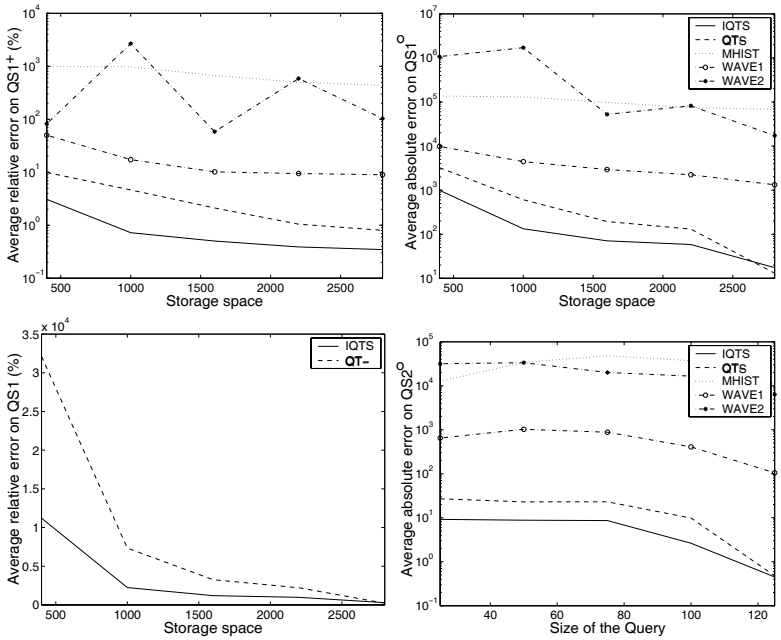


**Fig. 3.8.** Results for dense data

sparse data for different values of the query size (i.e. $\Delta$) is reported. In the picture on the bottom-right corner of Fig. 3.8, values of $\| e^{rel}(QS_2^+(\Delta, \Delta)) \|$ obtained for dense data are shown.

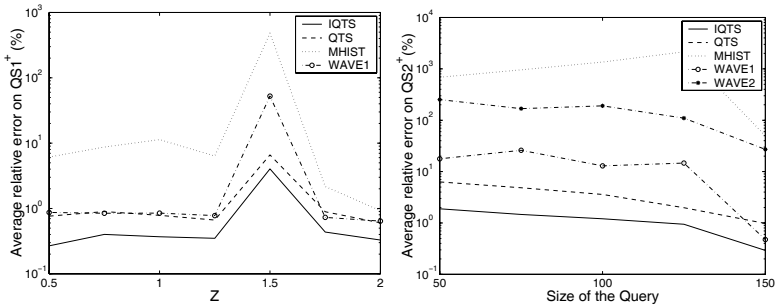


**Fig. 3.9.** Results for sparse data

# 4

# Multi-dimensional Histograms based on Binary Partitions

*In this chapter we investigate hierarchical binary partitions of multi-dimensional data as a basis for the construction of effective histograms. We propose two new classes of multi-dimensional histograms which combine new heuristics for partitioning data with very space-efficient physical representation models. A thorough experimental analysis shows that the proposed approach yields lower error rates than state-of-the-art summarization techniques and is much less sensitive to dimensionality.*

## 4.1 Introduction

A central problem in designing summarization techniques for multi-dimensional data is to retain a certain degree of accuracy in reconstructing query answers from summary data. The problem of effectively summarizing data with multiple dimensions presents intrinsic difficulties. In fact, the accuracy of answering queries on a summary structure depends on how much information on the actual data distribution is retained in the summarization process; as dimensionality increases, this information may require more and more storage space to be represented. For histogram-based summarization techniques this issue is known as the *curse of dimensionality*: as the number of dimensions increases, the size of the data domain grows much more than the number of data points, thus data become sparser and sparser; as a consequence, the number of buckets needed to achieve a satisfiable degree of accuracy explodes. For instance, consider two data distributions $D^2$ (of size $n^2$) and $D^{10}$ (of size $n^{10}$), where the same number of data points are distributed, respectively, on a two-dimensional and ten-dimensional domain. If we use the same number of buckets to partition $D^2$ and $D^{10}$, buckets of $D^{10}$ are likely to be much larger in volume than those of $D^2$. Therefore, the aggregate information associated to buckets of $D^{10}$ is less localized than in buckets of $D^2$ (as the aggregate value associated to each bucket is spread onto a larger volume), thus providing a poorer description of the actual data distribution.

The number of buckets needed to provide an accurate description of the original data distribution grows with dimensionality much faster than traditional histograms can manage. In fact, they are mostly characterized by a poorly intensive usage of the available storage space, and ineffective heuristics for guiding the histogram construction. As a consequence, state-of-the-art histograms, although intended to deal with generic multi-dimensional data, provide satisfiable estimation accuracy only in the low-dimensional case, while their performances tend to worsen dramatically as dimensionality increases. Especially in high-dimensionality scenarios, no technique is known to succeed in constructing histograms yielding "reasonable" error rates within a "reasonable" space bound. At the same time, no technique based on other approaches than histograms (such as wavelets, sampling, etc.) is known to provide satisfiable accuracy in the multi-dimensional context.

In this chapter we present an approach which is an effort in this direction. We study hierarchical binary partitions as a basis for effective multi-dimensional histograms, focusing our attention on two aspects which turn out to be crucial for histogram accuracy: the representation model and the strategy adopted for partitioning data into buckets. As regards the former, we propose a very specific space-efficient representation model where bucket boundaries are represented implicitly by storing the partition tree. Histograms adopting this representation model (which will be said to be *Hierarchical Binary Histograms - HBH*) can store a larger number of buckets within a given amount of memory w.r.t. histograms using a "flat" explicit storage of bucket boundaries (or bucket MBRs). On top of that, we consider the introduction of a constraint on the hierarchical partition scheme, allowing each bucket to be partitioned only by splits lying onto a regular grid defined on it: histograms adopting such a constrained partitioning paradigm will be said to be *Grid Hierarchical Binary Histograms* (*GHBH*). We show how the introduction of the grid-constrained partitioning of *GHBH*s can be exploited to further enhance the physical representation efficiency of *HBH*s. As regards the construction of effective partitions, we introduce some heuristics guiding the data summarization by locating inhomogeneous regions of the domain where a finer-grain partition is needed.

By means of experiments, we provide a thorough analysis of different classes of histograms based on hierarchical partitions: we study the accuracy provided by combining different heuristics (both our new proposals, as well as the "classical" heuristics of MHIST and MinSkew) with either the traditional MBR-based representation model or our specific tree-based ones (both the unconstrained and the grid-constrained one). Interestingly, we show that the impact of either *HBH* and *GHBH* representation models on the accuracy of query estimates is not simply orthogonal to the adopted heuristic. Thus, we identify the best combination of these different features, which turns out from adopting the grid-constrained hierarchical partitioning of *GHBH*s guided by one of the new heuristics.

Finally, we compare this class of *GHBH* with state-of-the-art techniques (MHIST, MinSkew, GENHIST, as well as other wavelet-based summarization approaches [80, 81]), showing that our technique results in much lower error rates. Experiments also show that our histograms still provide a satisfiable degree of accuracy at high-dimensionality scenarios.

## 4.2 Histograms based on Binary Partitions

### 4.2.1 Binary Partitions

Throughout the chapter, a $d$-dimensional data distribution $D$ is assumed. $D$ will be treated as a multi-dimensional array of integers with volume $n^d$ (see Sect. 1.4).
Given a block $b = \langle \rho_1, \ldots, \rho_d \rangle$ of $D$, let $x$ be a coordinate on the $i$-th dimension of $b$ such that $lb(\rho_i) \leq x < ub(\rho_i)$. Coordinate $x$ divides the range $\rho_i$ of $b$ into $\rho_i^{low} = [lb(\rho_i)..x]$ and $\rho_i^{high} = [(x+1)..ub(\rho_i)]$, thus partitioning $b$ into two sub-blocks $b^{low} = \langle \rho_1, \ldots, \rho_i^{low}, \ldots, \rho_d \rangle$ and $b^{high} = \langle \rho_1, \ldots, \rho_i^{high}, \ldots, \rho_d \rangle$. The pair $< b^{low}, b^{high} >$ is said to be the *binary split* of $b$ along the dimension $i$ at the position $x$; dimension $i$ and coordinate $x$ are said to be the *splitting dimension* and the *splitting position*, respectively.

Informally, a binary partition can be obtained by performing a binary split on $D$ (thus generating the two sub-blocks $D^{low}$ and $D^{high}$), and then recursively partitioning these two sub-blocks with the same binary hierarchical scheme.

**Definition 4.1.** *Given a $d$-dimensional data distribution $D$ with volume $n^d$, a* binary partition *$BP$ of $D$ is a binary tree such that:*

1. *the root of $BP$ is the block $\langle [1..n], \ldots, [1..n] \rangle$;*
2. *for each internal node $p$ of $BP$ the pair of children of $p$ is a binary-split of $p$.* □

The root, the set of nodes, and the set of leaves of the binary partition $BP$ will be denoted, respectively, as $Root(BP)$, $Nodes(BP)$, and $Leaves(BP)$. An example of binary partition on a two-dimensional data distribution is shown in Fig. 4.1.

### 4.2.2 Flat Binary Histograms

As introduced in Sect. 2.2, several histogram techniques proposed in literature, such as MHIST and MinSkew, use binary partitions as a basis for building histograms. In this section we provide a formal abstraction of classical histograms based on binary partitions. We refer to this class as *Flat Binary*

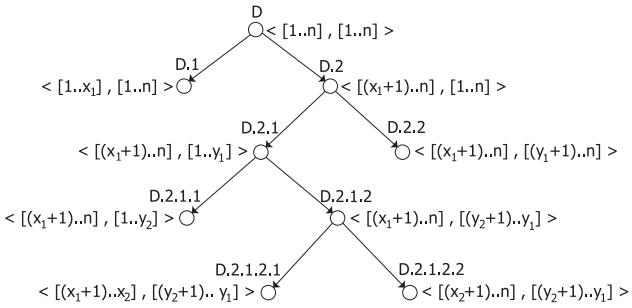


**Fig. 4.1.** A binary partition

*Histograms*, to highlight the basic characteristic of their physical representation model. The term "flat" means that, classically, buckets are represented independently from one another, without exploiting the hierarchical structure of the underlying partition.

**Definition 4.2.** *Given a d-dimensional data distribution D and a binary partition BP on D, the* Flat Binary Histogram *on D based on BP is the set of pairs:*

$$FBH = \{<b_1, sum(b_1)>, \ldots, <b_\beta, sum(b_\beta)>\}$$

*where the set* $\{b_1, \ldots, b_\beta\}$ *coincides with* $Leaves(BP)$. □

In the following, given the flat binary histogram $FBH = \{< b_1, sum(b_1) > , \ldots, < b_\beta, sum(b_\beta)>\}$, the blocks $b_1, \ldots, b_\beta$ will be said to be the *buckets* of *FBH*, and the set $\{b_1, \ldots, b_\beta\}$ will be denoted as $Buckets(FBH)$.

Figure 4.2 shows how the two-dimensional flat binary histogram corresponding to the binary partition of Fig. 4.1 can be obtained by progressively performing binary splits. The histogram consists of the following set:
$\{ < \langle [1..x_1], [1..n] \rangle , 50 >, < \langle [x_1+1..n], [1..y_2] \rangle, 61 >, < \langle [x_1+1..x_2], [y_2+1..y_1] \rangle, 0 >, < \langle [x_2+1..n], [y_2+1..y_1] \rangle, 63 >, < \langle [x_1+1 .. n], [y_1+1 .. n] \rangle, 82 > \}$.

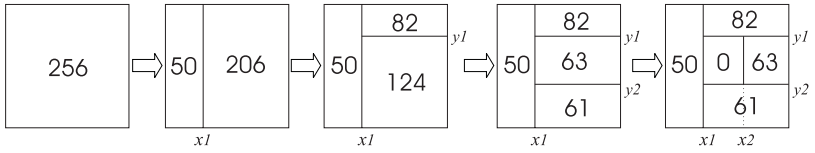


**Fig. 4.2.** Constructing a 2-dimensional *FBH*

A flat binary histogram can be represented by storing, for each bucket of the partition, both its boundaries and the sum of its elements. Assuming that

32 bits are needed to encode an integer value, $2 \cdot d$ 32-bit words are needed to store the boundaries of a bucket, whereas one 32-bit word is needed to store a sum value. Therefore, the storage space consumption of a flat binary histogram $FBH(D)$ is given by: $size(FBH)=(2 \cdot d + 1) \cdot 32 \cdot |Buckets(FBH)|$ bits. Indeed, as cited in Sect. 2.2.3, MHIST and MinSkew algorithms use a different representation of buckets: instead of storing the ranges delimiting the leaves of the binary partition, they store, for each leaf, the coordinates of its MBR (*minimal bounding rectangle*). For instance, consider the case that $D$ is a two-dimensional data distribution with two points in it, placed at the ends of a diagonal. According to this representation model, splitting $D$ will lead to two single-point MBRs. W.r.t. the naive representation model introduced above for *FBH*, this aims at a higher accuracy in approximating $D$, and introduces no spatial overhead. In fact, representing the coordinates of the MBR inside a bucket $b$ has the same cost as representing the boundaries of $b$, but the information provided by the MBR on where non null elements are located inside $b$ is more accurate. However, the storage space consumption of either MHIST and MinSkew histograms is equal to that of an *FBH* histogram having the same number of buckets. It turns out that *FBH* s are a meaningful representative of the class of histogram that are based on a binary hierarchical partition.

In Sect. 4.2.3 we propose an alternative representation scheme, which does not enable MBRs to be stored, but allows bucket boundaries to be represented more efficiently, so that a larger number of buckets can be stored within the same storage space bound.

### 4.2.3 Hierarchical Binary Histogram

The hierarchical partition scheme defined in the previous section can be exploited to define a new class of histogram, which improves the efficiency of the physical representation. It can be observed that most of the storage space consumption of an *FBH* (i.e. $2 \cdot d \cdot 32 \cdot |Buckets(FBH)|$) is due to the representation of the bucket boundaries. Indeed, buckets of a flat binary histogram cannot describe an arbitrary partition of the multi-dimensional space, as they are constrained to obey a hierarchical partition scheme. The simple *FBH* representation paradigm defined in Sect. 4.2.2 does not exploit the hierarchical nature of the partition. In particular, the boundaries of two buckets of an *FBH* corresponding to a pair of siblings in the underlying binary partition could be represented by representing only the boundaries of their father, as well as the splitting position and dimension generating them. For instance, consider two buckets $b_i$, $b_{i+1}$ which correspond to a pair of siblings in the hierarchical partition underlying the histogram; then, $b_i$, $b_{i+1}$ can be viewed as the result of splitting a block $b$ of the multi-dimensional space along one of its dimensions. Therefore the boundaries of $b_i$ and $b_{i+1}$ could be derived from the boundaries of their father if the splitting position and dimension generating

$b_i$, $b_{i+1}$ were available. We expect that exploiting this characteristic improves the efficiency of the representation.

The idea underlying *Hierarchical Binary Histogram* consists in storing the partition tree explicitly, in order to both avoid the explicit storage of bucket boundaries and provide a structure indexing buckets. In particular, storing the structure of the partition enables the boundaries of the buckets (which correspond to the leaves of the partition tree) to be retrieved from the partition itself. Moreover, as storing the partition tree is less costly (in terms of amount of storage space) than storing bucket boundaries (as it will be explained in the following), some storage space can be saved and invested to obtain finer grain buckets.

**Definition 4.3.** *Given a d-dimensional data distribution D, a* Hierarchical Binary Histogram *on D is a pair* $HBH(D) = \langle P, S \rangle$ *where P is a binary partition of D, and S is the set of pairs* $\{\langle p, sum(p) \rangle \mid p \in Nodes(P)\}$. □

In the following, given $HBH = \langle P, S \rangle$, the term $Nodes(HBH)$ will denote the set $Nodes(P)$, whereas $Buckets(HBH)$ will denote the set $Leaves(P)$.

A hierarchical binary histogram $HBH = \langle P, S \rangle$ can be stored efficiently by representing $P$ and $S$ separately, and by exploiting some intrinsic redundancy in their definition. To store $P$, first of all we need one bit per node to specify whether the node is a leaf or not. As the nodes of $P$ correspond to ranges of the multi-dimensional space, some information describing the boundaries of these ranges has to be stored. This can be accomplished efficiently by storing, for each non leaf node, both the splitting dimension and the splitting position which define the ranges corresponding to its children. Therefore, each non leaf node can be stored using a string of bits, having length $32 + \lceil log\ d \rceil + 1$, where 32 bits are used to represent the splitting position, $\lceil log\ d \rceil$ to represent the splitting dimension, and 1 bit to indicate that the node is not a leaf. On the other hand, 1 bit suffices to represent leaf nodes, as no information on further splits needs to be stored. Therefore, the partition tree $P$ can be stored as a string of bits (denoted as $Array_P(HBH)$) consisting in the concatenation of the strings of bits representing $P$ nodes.

The pairs $\langle p_1, sum(p_1) \rangle, \ldots, \langle p_m, sum(p_m) \rangle$ of the set $S$ (being $m = |Nodes(HBH)|$) can be represented using an array containing the values $sum(p_1), \ldots, sum(p_m)$, where the sums are stored according to the ordering of the corresponding nodes in $Array_P(HBH)$. Indeed, it is worth noting that not all the sum values in $S$ need to be stored, as some of them can be derived. For instance, the sum of every right-hand child node is implied by the sums of its parent and its sibling. Therefore, for a given hierarchical binary histogram $HBH$, the set $Nodes(HBH)$ can be partitioned into two sets: the set of nodes that are the right-hand child of some other node (which will be called *derivable nodes*), and the set of all the other nodes (which will be called *non-derivable nodes*). Derivable nodes are the nodes which do not need to be explicitly represented as their sum can be evaluated from the sums of

non-derivable ones. The sums associated to non-derivable nodes are stored into the array $Array_S(HBH)$.

On the right-hand side of Fig. 4.3 this representation paradigm is applied to the *HBH* shown on the left-hand side of the same figure.

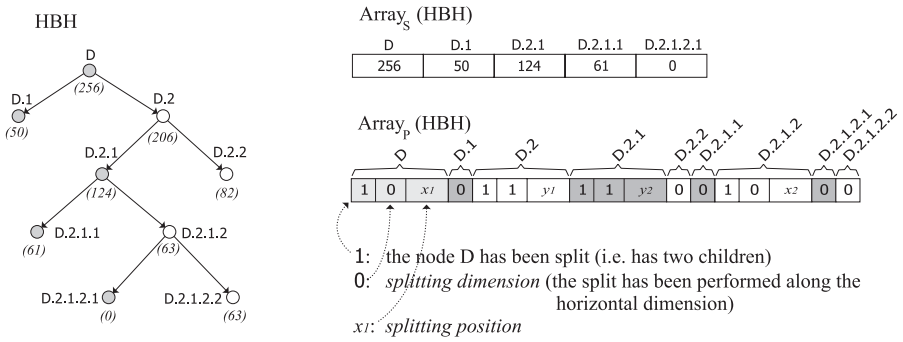


**Fig. 4.3.** Representation of an *HBH*

In Fig. 4.3 non-derivable nodes are colored in grey, whereas derivable nodes are white. Leaf nodes of *HBH* are represented in the array $Array_P(HBH)$ by means of a unique bit, with value 0. As regards non-leaf nodes, the first bit of their representation is 1 (meaning that these nodes are split); the second bit is 0 if the node is split along the horizontal dimension, 1 otherwise.

This representation scheme can be made more efficient by exploiting the possible sparsity of the data. In fact, it often occurs that the size of the multi-dimensional space is large w.r.t. the number of non-null elements. Thus, we expect that null blocks are very likely to occur when partitioning the multi-dimensional space. This leads us to adopt an ad-hoc compact representation of such blocks in order to save the storage space needed to represent their sums. A possible efficient representation of null blocks could be obtained by avoiding storing zero sums in $Array_S(HBH)$ and by employing one bit more for each non-derivable node in $Array_P(HBH)$ to indicate whether its sum is zero or not. Moreover, observe that we are not interested in *HBH*s where null blocks are further split since, for a null block, the zero sum provides detailed information of all the values contained in the block, thus no further investigation of the block can provide a more detailed description of its data distribution. Therefore any *HBH* can be reduced to one where each null node is a leaf, without altering the description of the overall data distribution that it provides. It follows that in $Array_P(HBH)$ non-leaf nodes do not need any additional bit either, since they cannot be null. According to this new representation model, each node in $Array_P(HBH)$ is represented as follows:

- if the node is not a leaf it is represented using a string of length $32+\lceil log\ d \rceil+1$ bits, where 32 bits are used to represent the splitting position, $\lceil log\ d \rceil$ to

represent the splitting dimension, and 1 bit to indicate that the node is not a leaf.
- if the node is a leaf, it is represented using one bit to state that the node has not been split and, only if it is a non-derivable node, one additional bit to specify whether it is null or not.

On the other hand, $Array_S(HBH)$ represents the sums of all non-null non-derivable nodes.
A possible representation of the *HBH* shown on the left-hand side of Fig. 4.3 according to this new model is provided in Fig. 4.4. In particular, both non-leaf nodes and derivable leaf nodes are stored in the same way as in Fig. 4.3, whereas non-derivable leaf nodes are represented with a pair of bits. The first one of these has value 0 (which states that the node has not been split), and the second one is either 0 or 1 to indicate whether the node is null or not, respectively.

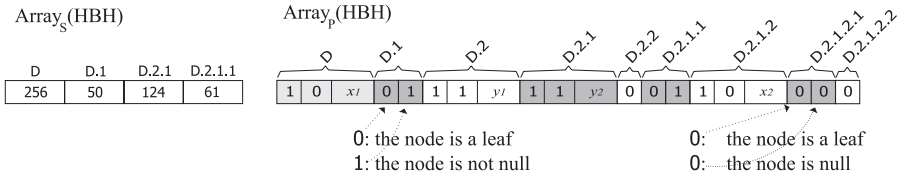


**Fig. 4.4.** Efficient Representation of an *HBH*

An *HBH* contains more information than the corresponding *FBH*, as it represents the sums associated to all the nodes (not only the leaves) of the partition tree. This feature can be exploited to make query answering more efficient, as explained in the following section.

### 4.2.4 Evaluating Sum Range Queries on an *HBH*

When a sum range query is issued on an *HBH*, it is estimated by visiting the partition tree starting from its root to locate the nodes whose range overlaps the query range. Let $r$ be the range of the query. When a node is being visited, three cases may occur:

1. *the range corresponding to the node is external to* $r$: the node gives no contribution to the estimate;
2. *the range corresponding to the node is entirely contained in* $r$: the contribution of the node is given by the value of its sum;
3. *the range corresponding to the node partially overlaps* $r$: if the node is a leaf, linear interpolation is performed to evaluate which portion of the sum associated to the node lies onto $r$. Otherwise, the contribution of the node is the sum of the contributions of its children, which are evaluated recursively.

We point out that the *HBH* could be represented (using the same storage space) by storing the sums of all the leaves of the underlying partition tree, instead of storing the sums of non-derivable nodes. Indeed, the fact that the sums of all the nodes of the partition tree are made available in an *HBH* makes query answering more efficient w.r.t. the case that only the sums of the leaves were available. That is, when a node whose range is entirely contained in the query range is visited, its descending nodes do not need to be visited, so that it is not always necessary to reach leaf nodes.

**Remark.** Observe that the physical representation model introduced above cannot be used to represent the coordinates of the MBRs inside buckets. This is due to the fact that MBRs of two sibling nodes of a binary partition in general do not coincide with node boundaries, because the two partitions can be shrunk to eliminate any null spaces around. This means that our approach can be considered an alternative to the idea of storing MBRs.

### 4.2.5 Grid Hierarchical Binary Histogram

In the previous section it has been shown how the exploitation of the hierarchical partition scheme underlying a histogram yields an effective benefit. That is, a hierarchical binary histogram can be represented more efficiently than the corresponding flat histogram, thus the available storage space can be used to represent a larger number of buckets.

We now introduce further constraints on the partition scheme adopted to define the boundaries of the buckets. The basic idea is that the use of a constrained partitioning enables a more efficient physical representation of the histogram w.r.t. histograms using more general partition schemes. The saved space can be invested to obtain finer grain blocks, approximating data in more detail.

Basically, a *Grid Hierarchical Binary Histogram GHBH* is a hierarchical binary histogram whose internal nodes cannot be split at any position of any dimension: every split of a block is constrained to be laid onto a grid, which divides the block into a number of equal-size sub-blocks. This number is a parameter of the partition, and it is the same for every block of the partition tree. In the following, a binary split of a block $b =< \rho_1, \ldots, \rho_d >$ along the dimension $i$ at the position $x$ will be said to be a *binary split of degree* $k$ if $x = lb(\rho_i) + \left\lceil j \cdot \frac{size(\rho_i)}{k} \right\rceil - 1$ for some $j \in [1 \,..\, k-1]$.

**Definition 4.4.** *Given a d-dimensional data distribution D, a* grid binary partition *of degree k on D is a binary partition GBP of D such that, for each non-leaf node p of GBP, the pair of children of p is a binary-split of degree k on p.* □

**Definition 4.5.** *Given a d-dimensional data distribution D, a* Grid Hierarchical Binary Histogram of degree *k on D is a hierarchical binary histogram*

*k-GHBH* $= \langle P, S \rangle$ *on* $D$ *where* $P$ *is a grid binary partition of degree* $k$ *on* $D$. □

In the following, we will use *GHBH* as an acronym of grid hierarchical binary histogram without specifying the degree $k$ of the partition when $k$ is not relevant. Figure 4.5 shows an example of two-dimensional 4-*GHBH*.

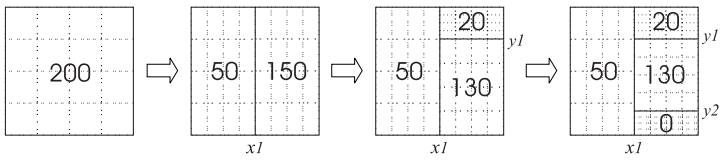


**Fig. 4.5.** Constructing a 4-*GHBH*

Constraining each split of the partition to be laid onto a grid defined on the blocks of the histogram enables some storage space to be saved to represent the splitting position. In fact, for a grid binary partition of degree $k$, the splitting position can be stored using $\lceil log\ (k-1) \rceil$ bits, instead of 32 bits. In the following, we will consider degree values which are a power of 2, so that the space consumption needed to store the splitting position will be simply denoted as $log\ k$. Figure 4.6 shows the representation of the grid hierarchical binary histogram of Fig. 4.5.

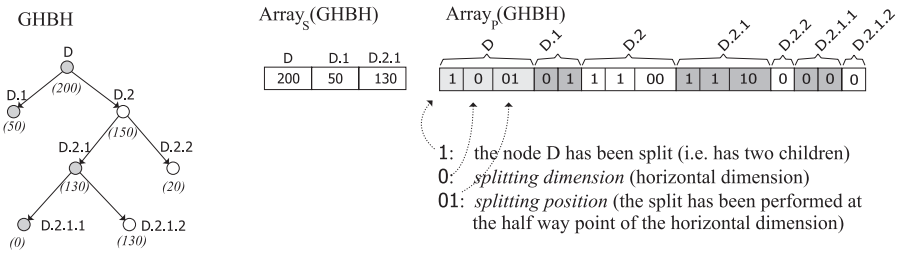


**Fig. 4.6.** Representing the *GHBH* of Fig. 4.5

### 4.2.6 Usage of Storage Space

We now compare the effectiveness of the different physical representation models by evaluating the number of buckets of a histogram $H$ of type *FBH*, or *HBH* or *GHBH* saturating the available storage space $B$. In the following, given a storage space bound $B$, a histogram $H$ will be said to be *B-maximal* if $size(H) \leq B$ and no split can be performed on any bucket of $H$, otherwise the storage space consumption of $H$ would exceed $B$.

**Proposition 4.6.** *Let $D$ be a $d$-dimensional data distribution, $B$ a storage space bound, and $T$ a type of histogram (where $T$ is either FBH, HBH or k-GHBH). The number of buckets $\beta_T$ of a $B$-maximal histogram $H$ of type $T$ on $D$ is in the ranges reported in Table 1.*

| ***Type*** | ***Number of buckets*** |
|---|---|
| *FBH* | $\beta_{FBH} = \left\lfloor \frac{B}{32 \cdot (2 \cdot d+1)} \right\rfloor$ |
| *HBH* | $\beta^{min}_{HBH} = \left\lfloor \frac{B+\lceil log\ d \rceil+34}{67+\lceil log\ d \rceil} \right\rfloor \leq \beta_{HBH} \leq \left\lfloor \frac{B+\lceil log\ d \rceil+2}{35+\lceil log\ d \rceil} \right\rfloor = \beta^{max}_{HBH}$ |
| *k-GHBH* | $\beta^{min}_{GHBH} = \left\lfloor \frac{B+log\ k+\lceil log\ d \rceil+2}{35+log\ k+\lceil log\ d \rceil} \right\rfloor \leq \beta_{GHBH} \leq \left\lfloor \frac{B+log\ k+\lceil log\ d \rceil-30}{3+log\ k+\lceil log\ d \rceil} \right\rfloor = \beta^{max}_{GHBH}$ |

**Table 1**

*Proof.* See Appendix A. □

Observe that while all possible $B$-maximal histograms of type *FBH* have the same number of buckets (for a given $B$), this does not hold for *HBH* and *GHBH*. This is due to the fact that the buckets of an *HBH* (or, equivalently, a *GHBH*) have a different storage space consumption depending on the underlying data distribution. Therefore bounds $\beta^{min}_{HBH}$, $\beta^{min}_{GHBH}$, $\beta^{max}_{HBH}$, $\beta^{max}_{GHBH}$ reported in Table 1 have been computed by considering the case that the available storage space $B$ is equal to the minimum and maximum storage space consumption of an *HBH* and a *GHBH* histogram, respectively (see the proof in the appendix for details).

Comparing the ranges defining the possible number of buckets of the different types of histogram, the main conclusion that can be drawn is that the physical representation scheme adopted for an *HBH* permits us to store a larger number of buckets w.r.t. an *FBH* within the same storage space bound, as the denominator of $\beta^{min}_{HBH}$ (i.e. $67 + \lceil log\ d \rceil$) is less than the denominator of $\beta_{FBH}$ (i.e. $32 \cdot (2 \cdot d+1)$). Analogously, the constraint on the splitting position of a *GHBH* further increases the number of buckets that can be represented within $B$, as we can assume that $log\ k < 32$, thus $67 > 35 + log\ k$.

In order to give an idea of the benefits (in terms of number of buckets) introduced by the efficient representation models of *HBH* and *GHBH*, consider the case of an 8-dimensional data distribution (we consider a 8-*GHBH*). In this scenario, the number of buckets of an *HBH* is between 8 and 14 times the number of buckets of a *FBH* on the same data, whereas the number of buckets of a 8-*GHBH* is between 13 and 60 times the number of buckets of a *FBH* on the same data.

## 4.3 Constructing Histograms based on Binary Partitions

### 4.3.1 Optimal Histograms

As discussed in Sect. 2.2.2, one of the most important issues when dealing with multi-dimensional histograms is building the histogram which approximates "best" the original data distribution, while being constrained to fit in a given storage space bound. The notion of *V-Optimal* histogram, introduced in Sect. 2.2.2, can be trivially specialized to the case of histograms based on binary partitions.

**Definition 4.7.** *Let $D$ be a $d$-dimensional data distribution, $B$ a storage space bound, and $T$ a type of histogram (where $T$ is either FBH, HBH, or k-GHBH). A histogram $H^\star$ of type $T$ on $D$ is said to be* V-Optimal *w.r.t. $B$ if the following conditions hold:*

1. $size(H^\star) \leq B$;
2. $SSE(H^\star) = min_{H' \in \mathcal{H}_T(D,B)} \{SSE(H')\}$

*where $\mathcal{H}_T(D, B)$ is the set of all histograms of type $T$ on $D$ whose size is less than or equal to $B$.* □

**Theorem 4.8.** *Let $D$ be a $d$-dimensional data distribution, $B$ a storage space bound, and $T$ a type of histogram (where $T$ is either FBH, HBH, or k-GHBH). A V-Optimal histogram $H^\star$ of type $T$ on $D$ w.r.t. $B$ can be computed in the complexity bounds reported in Table 2.*

| ***Type of histogram*** | ***Complexity bound for V-Optimal histogram*** |
|---|---|
| *FBH* | $O(\frac{B^2}{d \cdot 2^d} \cdot n^{2d+1})$ |
| *HBH* | $O\left(d \cdot \frac{B^2}{2^d} \cdot n^{2d+1}\right)$ |
| *k-GHBH* | $O\left(d \cdot \frac{B^2}{2^d} \cdot k^{d+1} \cdot n^d\right)$ |

**Table 2**

*Proof.* See Appendix A. □

Results for *FBH*s in Theorem 4.8 can be viewed as an extension of the results presented in [66], where the problem of finding the optimal binary hierarchical partition w.r.t. several metrics (including the SSE) has been shown to be polynomial in the two-dimensional case [1]. We also recall that this result does not hold for arbitrary partitions, where the problem of finding the V-Optimal histogram has been shown to be NP-hard in the two-dimensional

[1] Indeed, [66] addresses the dual problem, see Sect. 2.2.2

case [66]. In the one-dimensional case the classes of arbitrary and hierarchical partitions coincide, and thus our result is consistent with that of [52], where a polynomial-time algorithm for constructing a V-Optimal histogram on a one-dimensional data distribution has been proposed.
Comparing results for *FBH*s, *HBH*s and *GHBH*s in Theorem 4.8 we can observe that the computational complexity of constructing a V-Optimal *FBH* is less than that of computing a V-Optimal *HBH* within the same storage space bound. Essentially, this is due to the more complex representation scheme adopted by *HBH*, whose buckets are represented differently depending on whether they are null or not, derivable or not (see the theorem proof in the appendix for more details). However, the two complexity bounds have the same polynomial degree w.r.t. the volume of the input data; moreover, the aim of introducing *HBH* is not to make the construction process faster, but to yield a more effective histogram. The complexity of building k-*GHBH*$^\star$ is less than that of *HBH*$^\star$ as, in the former case, the number of splits that can be applied to a block are constrained by the grid. Note that if $k = n$ the complexities of the two cases coincide.

### 4.3.2 Greedy Algorithms

From the complexity bounds reported in Table 2 we can draw the conclusion that V-Optimal hierarchical histograms can be built in time polynomial w.r.t. the size of the domain of the input data distribution. In particular, both *FBH*$^\star$ and *HBH*$^\star$ can be constructed in nearly quadratic time w.r.t. $n^d$, whereas k-*GHBH*$^\star$ in linear time (since the grid degree $k$ can be assumed as a constant). Indeed, for high-dimensionality scenarios the size of the domain is so large that finding the V-Optimal becomes unfeasible. In order to reach the goal of minimizing the SSE, in favor of simplicity and speed, we adopt a greedy approach for constructing the histogram, accepting the possibility of obtaining a non-optimal solution. As it will be shown later, this approach can work in linear time w.r.t. $N$ (the number of non-null points inside $D$), which is generally much smaller than $n^d$ (especially in the case of high-dimensionality data).
Our approach can be viewed as an extension of the standard greedy strategy adopted by MHIST and MinSkew. It starts from the binary histogram whose partition tree has a unique node (corresponding to the whole $D$) and, at each step, selects the leaf of the binary-tree which is the most in need of partitioning and applies the most effective split to it. In particular, in the case of a *GHBH*, the splitting position must be selected among all the positions laid onto the grid overlying the block. Both the choices of the block to be split and of the position where it has to be split are made according to a greedy criterion. A number of possible greedy criteria can be adopted for choosing the block which is most in need of partitioning and how to split it. The greedy strategies tested in our experiments are reported in the table shown in Fig. 4.7. Two of them (namely *Max-Red*$^{marg}$ and *MaxDiff*) are not new, as they were used by

other techniques (*MinSkew* and *MHIST*, respectively) to drive the histogram construction.

Criteria denoted as *marginal* (marg) investigate *marginal distributions* of blocks. The marginal distribution of a block $b$, denoted as $marg_i(b)$, has been defined in Sect. 2.2.3. In the following, the term *marginal SSE* will be used to denote $SSE\,(marg_i(b))$ for some $i \in 1..d$.

| **Criterion** | **The node $b$ to be split, and the position $\langle dim, pos\rangle$ where $b$ is split** |
|---|---|
| Max-Var/ Max-Red | the leaf node $b$ having maximum SSE is chosen, and split at the position $< dim, pos >$ producing the maximum reduction of $SSE(b)$ (i.e. $SSE(b) - \big(SSE(b^{low}) + SSE(b^{high})\big)$ is maximum w.r.t. every possible split on $b$) |
| Max-Var$^{marg}$/ Max-Red$^{marg}$ | for each leaf node, the marginal SSE along its dimensions are evaluated, and the node $b$ having maximum marginal SSE is chosen ($dim$ is the dimension s.t. $SSE(marg_{dim}(b))$ is maximum). Then, $b$ is split at the position $pos$ laying onto $dim$ which yields the maximum reduction of $SSE(marg_{dim}(b))$ w.r.t. every possible split along $dim$ |
| Max-Red | the strategy evaluates how much the SSE of every leaf node is reduced by trying all possible splits. $b$ and $< dim, pos >$ are the leaf node and the position which correspond to the maximum reduction of SSE (i.e. $SSE(b) - \big(SSE(b^{low}) + SSE(b^{high})\big)$ is maximum w.r.t. every possible split on all the buckets of the histogram) |
| Max-Red$^{marg}$ used by *MinSkew* | all possible splits along every dimension of all leaf nodes are performed, and the corresponding reductions of marginal SSE (along the splitting dimensions) are evaluated. $b$ and $< dim, pos >$ are the bucket and the position such that the reduction of $SSE(marg_{dim}(b))$ obtained by splitting $b$ at $< dim, pos >$ is maximum w.r.t. the reduction of any $SSE(marg_i(b))$ (where $i \in [1..d]$) which could be obtained by performing some split along $i$ |
| MaxDiff used by *MHIST* | the leaf node $b$ is chosen characterized by a marginal distribution (along any dimension $i$) containing two adjacent values $e_j$, $e_{j+1}$ with the largest difference w.r.t. every other pair of adjacent values in any other marginal distribution of any other leaf node. Then $b$ is split along the dimension $i$ by putting a boundary between $e_j$ and $e_{j+1}$. |

**Fig. 4.7.** Splitting strategies

The resulting algorithm scheme is shown below. It uses a priority queue $q$ where nodes of the histogram are ordered according to their need to be partitioned. At each step, the node at the top of the queue is extracted and

split, and its children are in turn enqueued. Before adding a new node $b$ to the queue, the function $Evaluate(G, b)$ is invoked, being $G$ the adopted greedy criterion. This function returns both a measure of the need of $b$ to be partitioned (denoted as *need*), and the position $(dim, pos)$ of the most effective split, according to the adopted criterion $G$. For instance, if $G$ =*Max-Var/Max-Red*, the function returns the SSE of $b$ into *need*, and the splitting position which yields the largest reduction of SSE into $< dim, pos >$. Otherwise, if *Max-Red* criterion is adopted, the value of *need* returned by $Evaluate(Max\text{-}Red, b)$ is the maximum reduction of SSE which can be obtained by splitting $b$, and the returned pair $< dim, pos >$ defines the position corresponding to this split.

**Greedy Algorithm**
INPUT $D$: a multi-dimensional data distribution;
$B$: available amount of storage space for representing the histogram;
$T$: the type of histogram to be built ($T \in \{FBH, HBH, GHBH\}$);
$G$: the greedy criterion to be adopted;
OUTPUT $H$: a histogram of type $T$ on $D$ within $B$;

**begin**
$q := new\ Queue(\ )$; // the priority queue $q$ is initialized;
$b_0 := \langle [1..n], \ldots, [1..n] \rangle$;
$H := new\ Histogram(T, b_0)$; // a new histogram of type $T$ consisting of the only // bucket $b_0$ is constructed;
<need, dim, pos> = Evaluate($G, b_0$);
$q.Insert(<\ b_0, <$need, dim, pos$>>)$;
**while** ( !$H$.overflow( ) )
$< b, <$need, dim, pos$> >= q.GetFirst(\ )$;
$< b^{low}, b^{high} >= H$.BinarySplit($b, dim, pos, B$);
$q.Insert(< b^{low}$, Evaluate($G, b^{low}$) $>)$;
$q.Insert(< b^{high}$, Evaluate($G, b^{high}$) $>)$;
**end_while;**
**return** $H$;
**end**

Therein function *BinarySplit* takes as arguments a bucket $b$ of the histogram, the chosen splitting position and the available storage space $B$. It returns the pairs of sub-blocks $< b^{low}, b^{high} >$ obtained by performing the specified binary split of $b$. Moreover, it evaluates the storage space consumption of adding $b^{low}$ and $b^{high}$ as children of $b$ (the storage space needed to store these new buckets depending on the histogram type). If the sum of this storage space consumption with the current size of $H$ is smaller than or equal to the space bound $B$, then buckets $b^{low}$, $b^{high}$ are actually inserted into $H$; otherwise, $H$ is not updated and the next invocation of *overflow*( ) will return *true*: this ends the histogram construction.
As regards function *Evaluate*, in the case of *FBH* and *HBH*, the splitting

positions to be evaluated for a bucket $b$ are all the positions between the boundaries of every dimension of $b$, whereas for *GHBH* the function computes only all possible splits laid onto the grid.

The cost of Greedy Algorithm strictly depends not only on the type of histogram to be built, but also on the adopted data model. For instance, data can be stored into a multidimensional array (where each cell is associated to a point of the multi-dimensional space), or by adopting a sparse model, where only non null data are stored. In the latter case, $D$ will be a set of $N$ tuples $< x_1, \ldots, x_d, val >$, where $x_1, \ldots, x_d$ are the coordinates and $val$ the value of non-null points.

In the rest of this section we provide complexity bounds and a workspace analysis for Greedy Algorithm for the different types of histogram (*FBH*, *HBH* and k-*GHBH*), and data models (either sparse and non-sparse). Furthermore, we also consider the use of suitable pre-computed data which serve as auxiliary structures for the evaluation of greedy criteria.

### Evaluating greedy criteria when constructing an *FBH* or an *HBH*

We discuss the complexity of computing $Evaluate(G, b)$ on a block $b$, during the construction of either an *FBH* and an *HBH*, when different greedy criteria $G$ are used, and when either the sparse data model or the non-sparse one are adopted. We show that the order of magnitude of the computational complexity of $Evaluate(G, b)$ does not depend on the criterion $G$. This is due to the fact that the SSE of a block, as well as its reduction due to a split, can be computed by scanning marginal distributions, as explained in the following.

**Max-Red.** Let $b =< \rho_1, \ldots, \rho_d >$ be the block of $D$ on which function *Evaluate* is invoked. If we denote as $<b^l, b^h>$ the binary split of $b$ along the dimension $dim$ at the position $pos$, it can be shown that the reduction of $SSE(b)$ due to this split is given by:

$$\begin{aligned} Red(b, dim, pos) &= SSE(b) - \big(SSE(b^l) + SSE(b^h)\big) = \\ &= \frac{vol(b^l) \cdot vol(b^h)}{vol(b)} \cdot \left(\frac{sum(b^l)}{vol(b^l)} - \frac{sum(b^h)}{vol(b^h)}\right)^2 \end{aligned} \tag{4.1}$$

As $sum(b^l) = \sum_{j=0}^{pos-lb(\rho_{dim})} marg_{dim}(b)[j]$ , and $sum(b^h) = sum(b) - sum(b^l)$, then $Red(b, dim, pos)$ can be computed by accessing $marg_{dim}(b)$. In particular, notice that all possible splits along the dimension $dim$ can be evaluated progressively, starting from $pos = lb(\rho_{dim}(b))$ to $pos = ub(\rho_{dim}(b)) - 1$. That is, if we denote as $b^l(i)$, $b^h(i)$ the sub-blocks of $b$ obtained by performing the binary split at the $i$-th position in this sequence, comparing all possible splits along the dimension $dim$ can be accomplished by first computing

$marg_{dim}(b)$, and then scanning it once, as: $sum\left(b^l(i)\right) = sum\left(b^l(i-1)\right) + marg^b_{dim}[i]$.

The cost of constructing all marginal distributions is either $O(d \cdot n^d)$ (non-sparse data model) or $O(d \cdot N)$ (sparse data model). The cost of scanning all marginal distributions to find the most effective splitting position is $O(d \cdot n)$, so that the complexity of $Evaluate(Max\text{-}Red, b)$ is bounded by either $O(d \cdot n^d)$ (non-sparse data model) or $O(d \cdot N + d \cdot n)$ (sparse data model).

**Max-Red$^{\mathbf{marg}}$.** By applying the definition of $SSE$, the reduction of $SSE(marg^b_{dim})$ due to a split of $b$ at the position $dim, pos$ can be shown to be:

$$Red^{marg}(b, dim, pos) = Red(b, dim, pos) \cdot P_{dim}, \tag{4.2}$$

where $P_{dim}$ is the ratio between the volume of $b$ and the size of its $dim$-th dimension. Therefore the computation of $Red^{marg}(b, dim, pos)$ can be accomplished within the same bound as $Red(b, dim, pos)$, as well as the cost of computing the splitting position yielding the largest reduction of marginal SSE has the same complexity bound as computing the splitting position corresponding to the largest reduction of SSE. This implies that $Evaluate(Max\text{-}Red^{marg}, b)$ can be computed within the same bound as the case that *Max-Red* is adopted.

**Max-Var / Max-Red.** The value of $SSE(b)$ (which is returned as *need*) is given by:

$$SSE(b) = sumSquare(b) - \frac{(sum(b))^2}{vol(b)} \tag{4.3}$$

where $sumSquare(b)$ is the sum of the squares of all values contained in $b$. This implies that $SSE(b)$ can be computed by accessing all non null elements inside $b$. Therefore the cost of evaluating $SSE(b)$ is $O(n^d)$ (non-sparse data model), or $O(N)$ (sparse data model).
The most effective splitting position can be evaluated in the same way as the case that *Max-Red* is adopted, and this cost dominates the cost of computing $SSE(b)$.
Therefore $Evaluate(Max\text{-}Var\ /\ Max\text{-}Red, b)$ can be computed within the same bound as the case of *Max-Red*.

Observe that, when *Max-Var/Max-Red* is adopted, the strategy used by Greedy Algorithm can be modified to make the computation of the histogram more efficient. Instead of computing the most effective splitting position when a bucket is inserted into the queue $q$, the value of $< dim, pos >$ is evaluated only when the bucket is extracted from $q$. In fact, when a new bucket is generated and inserted into $q$, its position inside the queue depends only on its $SSE$; similarly, the bucket which is most in need of partitioning is chosen only on the basis of its $SSE$. Therefore, computing the most effective splitting position $< dim, pos >$ for a bucket $b$ is useful only in the case that $b$ is extracted from the queue. By using this strategy, the most effective splitting position

is evaluated half many times as the previous strategy, as the buckets of the returned histogram (which correspond to the leafs of the underlying partition tree) have never been chosen to be split during the algorithm execution.

**Max-Var$^{\mathbf{marg}}$ /Max-Red$^{\mathbf{marg}}$.** In this case, the marginal distributions of $b$ must be constructed to compute both the value of *need* (that is, the maximum variance of the marginal distributions) and the most effective splitting position. The value of *need* can be computed by scanning all marginal distributions, and the reductions of marginal variance can be evaluated in the same way as the case that *Max-Red*$^{\text{marg}}$ is adopted. Therefore computing $Evaluate(Max\text{-}Var^{marg}/Max\text{-}Red^{marg}, b)$ has the same complexity bound as previous cases.

### Evaluating greedy criteria when constructing a *GHBH*

The main difference w.r.t. the construction of an *HBH* is that splitting positions in a *GHBH* are constrained by the grid, so that the number of possible splits to be compared when processing the block $b$ extracted from $q$ is $d \cdot k$ (instead of $d \cdot n$, as in the *HBH* case). The computation of $Red(b, dim, pos)$ and $Red^{marg}(b, dim, pos)$ corresponding to all the $d \cdot k$ splitting positions can be efficiently accomplished after pre-computing $d$ temporary data structures. Differently from the case of *HBH*, these temporary data structures are not the marginal distributions of $b$, but consist in the marginal distributions of $grid(b)$, which is constructed as follows. $grid(b)$ is a bucket containing $k^d$ elements, where each cell contains the sum of the elements of $b$ located in the corresponding cell of the $k$-th degree grid overlying $b$. The marginal distributions of $grid(b)$ will be denoted as $k\text{-}marg_1, \ldots, k\text{-}marg_d$. Figure 4.8 shows the $k$-marginal distributions associated to a bucket $b$ w.r.t. a 4th degree grid.

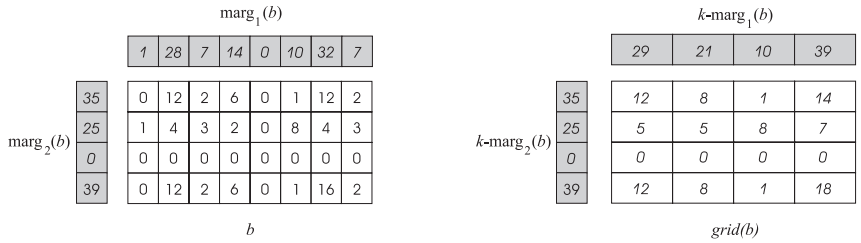


**Fig. 4.8.** $k$-marginal distributions of a bucket

Let $<dim, pos>$ be an admissible splitting position for the bucket $b$, and $i$, $i+1$ be the corresponding cells of $k\text{-}marg_{dim}$ (i.e. the contiguous cells of $k\text{-}marg_{dim}$ which would be separated by performing the split). Then, the reduction of $SSE(b)$ due to this split can be computed using the values $vol(b^l)$,

$vol(b^h)$, $sum(b^l)$ and $sum(b^h)$, as explained for the construction of an *HBH* (see (4.1) and (4.2)). In particular, $sum(b^l)$ and $sum(b^h)$ are:

$$sum(b^l) = \sum_{0 \le j \le i} k\text{-}marg_{dim}[j], \quad \text{and} \quad sum(b^h) = sum(b) - sum(b^l).$$

Obviously, constructing the $k$-marginal distributions has either cost $O(d \cdot N)$ (sparse data model) or $O(d \cdot n^d)$ (non-sparse model), but their scanning has cost $O(d \cdot k)$ (instead of $O(d \cdot n)$, as in the *HBH* case). Therefore, by applying the same reasoning explained in the previous section, it is easy to show that $Evaluate(G, b)$ has cost $O(d \cdot N + d \cdot \alpha)$ and $O(d \cdot n^d)$ for the two data models, respectively, where $\alpha = k$ for all greedy criteria $G$ except *Max-Var*$^{marg}$/*Max-Red*$^{marg}$. When *Max-Var*$^{marg}$/*Max-Red*$^{marg}$ is adopted, $\alpha = n$: in fact, in order to apply this criterion, it is necessary not only to access the $d$ $k$-marginal distributions to establish the most effective split, but it is also necessary to access the $d$ marginal distributions (which have size $n$) in order to compute the maximum marginal $SSE$, that corresponds to the value *need* of the bucket.

### Using pre-computation for evaluating greedy criteria

Each invocation of $Evaluate(G, b)$ can be accomplished more efficiently if the array $F$ of *partial sums* and the array $F^2$ of *partial square sums* are available. $F$ and $F^2$ have volume $(n+1)^d$, and are defined on the multi-dimensional range $< [0..n], \ldots, [0..n] >$ as follows:

- $F[i_1, \ldots, i_d] = \begin{cases} 0, & \text{if } i_j = 0 \text{ for some } j \in [1..d]; \\ sum(< 1..i_1, \ldots, 1..i_d >), & \text{otherwise.} \end{cases}$
- each element $F^2[i_1, \ldots, i_d]$ is either 0 (if $i_j = 0$ for some $j \in [1..d]$) or the sum of all the values $(D[j_1, \ldots, j_d])^2$ where $1 \le j_k \le i_k$ for each $k \in [1..d]$, otherwise.

Figure 4.9 shows an example of arrays of partial sums and partial square sums.

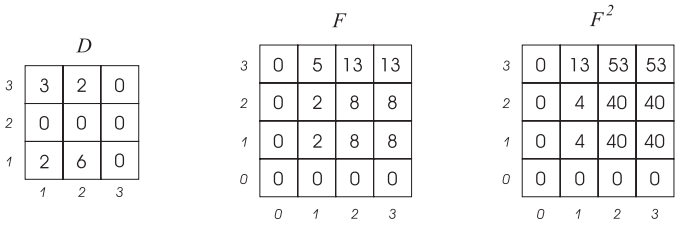


**Fig. 4.9.** Arrays of partial sums and partial square sums

By using $F$ and $F^2$ both the SSE of a block $b$ and the reduction the SSE due to a split of $b$ can be computed efficiently, as both $sum(b)$ and $sumSquare(b)$ can be evaluated by accessing $2^d$ elements of $F$ and $F^2$, instead of accessing all the elements of $b$. For instance, in the two-dimensional case depicted in Fig. 4.9, $sum(< [2..3], [2..3] >) = (-1)^0 \cdot F[3,3] + (-1)^1 \cdot F[1,3] + (-1)^1 \cdot F[3,1] + (-1)^2 \cdot F[1,1] = 13 - 5 - 8 + 2 = 2$. In general, given a block $b =< [l_1..u_1], \ldots, [l_d..u_d] >$, the values of $sum(b)$ and $sum^2(b)$ can be evaluated as follows:

$$sum(b) = \sum_{\mathbf{i} \in b} D[\mathbf{i}] = \sum_{\mathbf{j} \in vrt(\bar{b})} (-1)^{\mathcal{C}(\mathbf{j}, \mathbf{uv}(b))} \cdot F[\mathbf{j}]$$

and

$$sumSquare(b) = \sum_{\mathbf{i} \in b} D[\mathbf{i}]^2 = \sum_{\mathbf{j} \in vrt(\bar{b})} (-1)^{\mathcal{C}(\mathbf{j}, \mathbf{uv}(b))} \cdot F^2[\mathbf{j}]$$

In these expressions:

- $\bar{b} =< [l_1 - 1..u_1], \ldots, [l_d - 1..u_d] >$;
- $vrt(\bar{b})$ is the set of vertices of $\bar{b}$;
- $\mathbf{uv}(b) = \langle u_1, \ldots, u_d \rangle$ is the "upper" vertex of $b$;
- $\mathcal{C}(\mathbf{i}, \mathbf{j}) = \sum_{k=1}^{d} f(i_k, j_k)$, where: $f(a,b) = \begin{cases} 1, \text{if } a \neq b; \\ 0, \text{if } a = b. \end{cases}$

Then, for any splitting position $< dim, pos >$, once $sum(b^l)$, $sum(b^h)$, and $sumSquare(b)$ have been computed, either $Red(b, dim, pos)$, $Red^{marg}(b, dim, pos)$ and $SSE(b)$ can be evaluated using (4.1), (4.2), (4.3).

### Complexity of greedy algorithm

We discuss the time complexity of Greedy Algorithm for constructing histograms based on binary partitions under different greedy criteria, when either the sparse data model or the non-sparse one or pre-computation are adopted.

**Theorem 4.9.** *Given a d-dimensional data distribution $D$ with volume $n^d$ containing exactly $N$ non-null points, the time complexity of the greedy algorithms computing a histogram of type $T$ (where $T$ is either FBH, HBH or k-GHBH) on $D$, adopting either the sparse data model, or the non-sparse data model, or pre-computation, are reported in Fig. 4.10, where $\alpha = n$ if* Max-Var$^{marg}$/Max-Red$^{marg}$ *criterion is adopted, and $\alpha = k$ for all the other greedy criteria.*

*Proof.* See Appendix A. □

The complexity bounds reported in Fig. 4.10 show that Greedy Algorithm, in the case that a sparse data model is used, works in linear time w.r.t. both

| | ***Sparse model*** | ***Non-Sparse model*** | ***Using pre-computation*** | |
|---|---|---|---|---|
| | | | *Cost of pre-computation* | *Cost of computation* |
| *FBH* | $O\left((d \cdot N + d \cdot n) \cdot \beta_{FBH}^{max}\right)$ | $O\left(n^d \cdot \beta_{FBH}^{max}\right)$ | $O\left(2^d \cdot n^d\right)$ | $O\left(2^d \cdot d \cdot n \cdot \beta_{FBH}^{max}\right)$ |
| *HBH* | $O\left((d \cdot N + d \cdot n) \cdot \beta_{HBH}^{max}\right)$ | $O\left(n^d \cdot \beta_{HBH}^{max}\right)$ | $O\left(2^d \cdot n^d\right)$ | $O\left(2^d \cdot d \cdot n \cdot \beta_{HBH}^{max}\right)$ |
| *k-GHBH* | $O\left((d \cdot N + d \cdot \alpha) \cdot \beta_{GHBH}^{max}\right)$ | $O\left(n^d \cdot \beta_{GHBH}^{max}\right)$ | $O\left(2^d \cdot n^d\right)$ | $O\left(\left(2^d \cdot d \cdot \alpha + \log \beta_{GHBH}^{max}\right) \cdot \beta_{GHBH}^{max}\right)$ |

**Fig. 4.10.** Complexity bounds of Greedy Algorithm

the number of non-null points inside $D$ and the size of the dimensions of $D$. Notice that, as these bounds hold for all the considered greedy criteria, the idea of working on the one-dimensional marginal distributions of blocks does not provide a relevant benefit on the efficiency of the histogram construction w.r.t. investigating the actual multi-dimensional distributions of blocks in the greedy criterion.

Furthermore, we point out that the bounds reported in Fig. 4.10 assume that all steps of Greedy Algorithm have the same complexity. In fact, it is unlikely that this case occurs, since as the histogram construction goes on, smaller and smaller buckets are generated, and each of these buckets contain fewer tuples than buckets generated at previous steps. Therefore we expect that, after the very first steps, Greedy Algorithm deals with buckets whose volume is much smaller than $n^d$, whose marginal distributions have size much smaller than $n$, and containing fewer tuples than $N$.

Experimental results comparing the efficiency of the three different approaches (the ones based on the sparse data model, the non-sparse one, and pre-computation, respectively) are provided in Sect. 4.4.7.

### Workspace size for Greedy Algorithm

Implementing Greedy Algorithm with the adoption of pre-computation becomes unfeasible for high-dimensionality data, due to the explosion of the spatial complexity: the space needed to store $F$ and $F^2$ grows exponentially as dimensionality increases, even if the number of non-null values remains nearly the same. In real-life scenarios, it often occurs that $N \ll n^d$, especially for high-dimensionality data: as dimensionality increases, data become sparser and the size of the data domain increases much more dramatically w.r.t. the number of non-null data.

On the contrary, Greedy Algorithm under the sparse data model is much less sensitive on the increase of dimensionality, also from the point of view of the workspace size. In this case, Greedy Algorithm can be implemented by associating to each element of the queue not only the boundaries of the corresponding bucket $b$, but also the set of tuples belonging to $b$. Thus, when

a bucket $b$ is chosen and split into $b^{low}$ and $b^{high}$, tuples of $b$ are distributed among $b^{low}$ and $b^{high}$; then the triplet $< need, dim, pos >$ associated to the new bucket $b^{low}$ [resp. $b^{high}$] is computed by scanning only the tuples belonging to $b^{low}$ [resp. $b^{high}$]. That is, the partition underlying the histogram is used as an index to locate the tuples contained in the buckets. Therefore, the algorithm workspace (i.e. the storage space needed to store $q$) is $O(d \cdot N)$ (instead of $O(n^d)$, as in the case of pre-computation), since each non null point belongs to exactly one bucket of the partition at each step of the algorithm.

## 4.4 Experimental Results

Our experimental analysis investigates thoroughly several issues related to the accuracy provided by histograms based on binary partitions. We focus our attention on different issues, in order to study progressively the impact on the histogram accuracy of the following contributions: the specific tree-based representation model of *HBH*, the grid-constrained partition scheme of *GHBH*, and the heuristics used to accomplish the construction of the histogram. We also analyze the execution times of the proposed greedy approaches for constructing histograms based on binary partitions.

First, in Sect. 4.4.4 we study the impact of adopting different greedy criteria under different representation models. To this aim, we compare *FBH* and *HBH*, when all greedy criteria of Fig. 4.7 are used, in order to establish which combination yields the best accuracy. Then, in Sect. 4.4.5 we study the impact due to the grid-partitioning by comparing *HBH* and *GHBH*. This analysis will show that the best performances among all different combinations *type of histogram/greedy criterion* is provided by histograms of type *GHBH* constructed by adopting *Max-Var/Max-Red*. Then, in Sect. 4.4.6 we compare the accuracy provided by *GHBH* using *Max-Var/Max-Red* with the state of the art, showing that this approach outperforms (in terms of accuracy) all the others. Finally, in Sect. 4.4.7, we provide experimental results comparing construction times of the proposed greedy approaches, under either the sparse data representation model, the non-sparse one, and pre-computation.

### 4.4.1 Measuring the Approximation Error

The *absolute error* of the estimated answer of a query $q$ is defined as: $e^{abs} = |S - \tilde{S}|$, where $S$ denotes the actual answer of $q$, and $\tilde{S}$ its estimate. The *relative error* is defined as: $e^{rel} = \frac{|S-\tilde{S}|}{S}$. The accuracy of the various techniques was evaluated by measuring the absolute error and the relative error of the estimates of range-sum queries computed by accessing the histogram. The impact of a number of parameters on the accuracy was considered: the amount of storage space used to represent the histogram, the selectivity of queries, as well as several characteristics of the input data (such as dimensionality and domain size). The sensitivity to each of these parameters was

analyzed by varying it and fixing the other ones. In particular, in order to generate groups of queries with the same selectivity on a data distribution $D$, we used the following strategy. First, a number of distinct points were randomly selected inside $D$. Then, for each of these points $p$, a set of queries was generated starting from the query whose range coincided with $p$ and by progressively enlarging the query volume. This resulted in queries centered in $p$ with increasing selectivity. Finally, such obtained queries were grouped by their selectivity.

### 4.4.2 Synthetic Data

Our synthetic data generator is similar to those of [31, 81]. It takes as argument the following parameters: $d$, $n_1, \ldots, n_d$, $T$, $m$, $l_{min}$, $l_{max}$, $z_{min}$, $z_{max}$. Data are generated by creating $m$ dense regions inside a $d$-dimensional array $D$ with volume $n_1 \times \ldots \times n_d$. These dense regions will be denoted as $r_1, \ldots, r_m$. Each $r_i$ is characterized by its center $c_i$, its width $l_i$ and the *skew parameter* $z_i$ (as it will be clear later, $z_i$ determines the data distribution inside $r_i$). The coordinates of the centers $c_1, \ldots, c_m$ are generated according to a uniform distribution on the domain of $D$; the widths $l_1, \ldots, l_m$ are randomly chosen between $l_{min}$ and $l_{max}$, and $z_1, \ldots, z_m$ are randomly chosen between $z_{min}$ and $z_{max}$ (in our experiments, where it is not differently stated, we used $z_{min} = 0.5$ and $z_{max} = 2.5$). Initially $D$ is empty (i.e. it contains only null points), and at the end of the generation process it will contain a number of points whose sum is equal to $T$. In particular, $T$ is divided into $T_{\text{noise}}$ and $T_{\text{reg}}$. $T_{\text{reg}}$ is distributed among the $m$ regions according to a uniform distribution. The sum of points inside $r_i$ will be denoted as $T_i$. Region $r_i$ is populated in two steps:
1. a number $T_i$ of points inside the range of $r_i$ (namely, $p_1, \ldots, p_{T_i}$) are generated. Each of these points $p_j$ is obtained by first generating its distance $\delta_j$ from the center $c_i$ and then randomly choosing $p_j$ among the points having this distance from $c_i$. The value of $\delta_j$ is chosen according to a Zipf distribution on $[0..l_i]$ with parameter $z_i$.
2. for each $p_j$ (with $j \in [1..T_i]$) the value of $D[p_j]$ is increased by one. Thus, each cell $p$ in $r_i$ will contain the number of occurrences of $p$ in the sequence $p_1, \ldots, p_{T_i}$.

As regards $T_{\text{noise}}$, it is used to simulate noise in the data distribution: it is distributed among randomly generated points inside $D$ (in our experiments we used $T_{\text{noise}} = 0.05 \cdot T$).

This generation paradigm results in a data distribution where, for each $r_i$, the higher $z_i$, the more "concentrated" around $c_i$ the data points: as $z_i$ increases, points having large distances from $c_i$ are less probable to be generated.

### 4.4.3 Real life Data

We considered two real-life data sets. The first will be referred to as *Census* and was obtained from the *U.S. Census Bureau* using their *DataFerret* application for retrieving data. The data source is the *Current Population Survey* (CPS), from which the *March Questionnaire Supplement (1994)* file, containing 150943 tuples, was extracted. 8 attributes have been chosen: *Age*, *Parent's line number*, *Major Occupation*, *Marital Status*, *Race*, *Family Type*, *Public Assistance Type*, *School Enrollment*. The corresponding 8-dimensional array has about $4.6 \cdot 10^7$ cells, and contains 19129 non-null values (the density is about $4.2 \cdot 10^{-4}$).

The other data set will be referred to as *Forest Cover*. It was obtained from the U.S. Forest Service and is available at the UCI KDD archive site. It consists of 581012 tuples having 54 attributes. Among these, 10 attributes are numerical. As in [44], we considered the tuples projected on these numerical attributes, thus obtaining a 10-dimensional data distribution. The corresponding 10-dimensional array has about $4.4 \cdot 10^{28}$ cells (the density is about $1.3 \cdot 10^{-23}$).

### 4.4.4 Comparing *FBH* and *HBH* under Different Greedy Criteria

As explained in Sect. 4.2.3, the tree-based representation models of *HBH* and *GHBH* are an alternative to the MBR-based one of *FBH*. On the one hand, the tree-based representation model yields a larger number of buckets w.r.t. the MBR-based one (within the same storage space bound); on the other hand, buckets represented by means of their MBRs are likely to provide a more accurate description of the underlying data distribution. Therefore it is worth investigating which of these alternatives yields the best accuracy [2]. Hence, we studied how the accuracy provided by the different representation models of *FBH*, *HBH*, *GHBH* depends on the particular greedy criterion adopted to guide the histogram construction.

In order to establish how using different greedy criteria and employing different representation models affect the histogram effectiveness, we studied the accuracy of histograms obtained using different combinations *greedy criterion / representation model*. In particular, in this section we provide experimental results studying how error rates change when the same greedy criterion is used with either the flat representation model of *FBH* and the tree-based representation model of *HBH*. The impact of the use of the grid-constrained partitioning of *GHBH* will be considered in more detail in the following section.

[2] Obviously, we will consider only *FBH* adopting MBRs to represent the leaves of the partition tree: otherwise, the comparison of *FBH* with *HBH* and *GHBH* (in terms of accuracy) is rather expected.

Observe that the combinations *Max-Red*$^{marg}$*/FBH* and *MaxDiff/FBH* coincide with the techniques *MinSkew* and *MHIST*, respectively. However, the aim of this section is not to provide a complete comparison between our approach and these well-known techniques: further experiments comparing our approach with *MinSkew* and *MHIST* will be reported in Sect. 4.4.6.

Diagrams in Fig. 4.11 (a,b) were obtained on a synthetic 4-dimensional data distribution with volume $200 \times 200 \times 200 \times 200$ having a density of 0.02%, whereas diagrams in Fig. 4.11(c,d) were obtained on the 10-dimensional Forest Cover data set. The accuracy of the various criteria is evaluated w.r.t. the *storage space* available for the summarized representation.

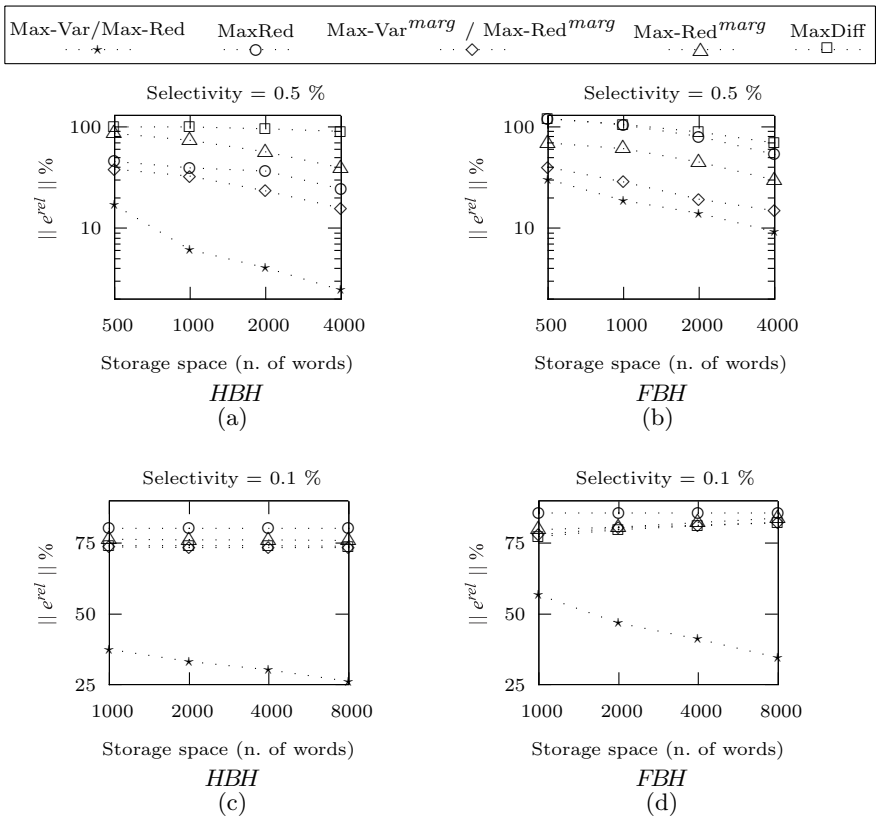


**Fig. 4.11.** Comparing combinations *greedy criterion / representation model* on 4D-synthetic data (a,b) and Forest Cover (c,d)

The main results which turn out from diagrams in Fig. 4.11 are the following:

1. for both *FBH* and *HBH*, *Max-Var/Max-Red* provides lower error rates than all other criteria; moreover, this criterion exploits the amount of storage space more effectively: as the available storage space increases, error rates decrease more rapidly w.r.t. the other criteria;
2. the accuracy of histograms built by employing any criterion other than *Max-Var/Max-Red* and *Max-Red* is nearly the same when either the *FBH* or *HBH* representation model is adopted. On the contrary, histograms constructed adopting either *Max-Var/Max-Red* or, to a lesser extent, *Max-Red* benefit from the use of the *HBH* representation model.

In order to explain why employing different greedy criteria results in different error rates, we considered a two-dimensional data distribution and studied how data is partitioned depending on the adopted criterion for both *HBH* and *FBH*. Partitions resulting from different combinations *greedy criterion / representation model* are depicted in Fig. 4.12.

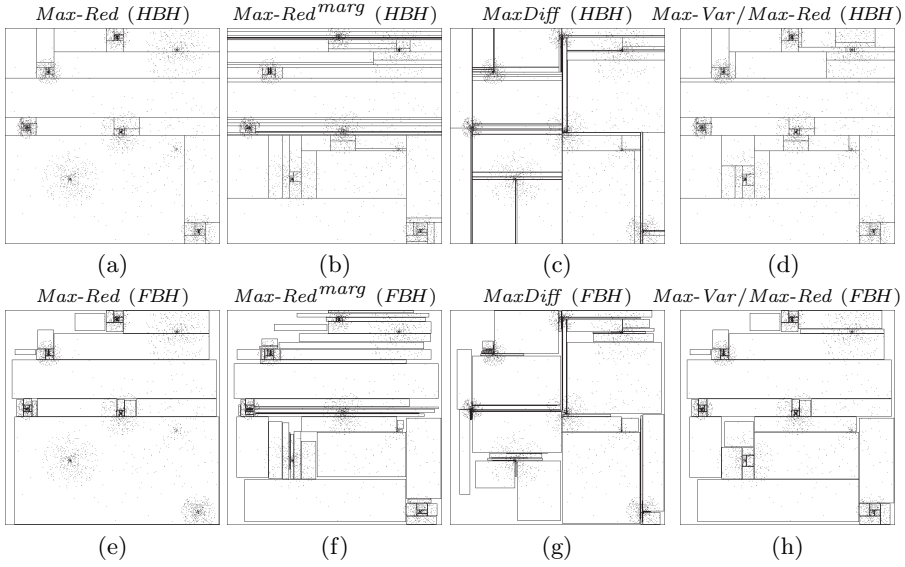


**Fig. 4.12.** Different combinations *greedy criterion / representation model*

By analyzing diagrams in Fig. 4.12, the following considerations can be drawn:
- *Max-Red* fails in building effective partitions, as it tends to progressively split "small" dense regions. This is made clear in Fig. 4.12(a) and (e): some clusters of data are rather disregarded, even if the amount of storage space would suffice to perform enough splits to isolate them; indeed, many splits are "wasted" to partition the core of other dense regions. This behavior can be explained by analyzing (4.1) which expresses the reduction of SSE of a block $b$ due to the binary split $< b^l, b^h >$. In fact, from this formula, it turns

out that splitting a small dense block $b_1$ can result in a larger reduction of SSE w.r.t. splitting a larger block $b_2$, even if $SSE(b_2) \gg SSE(b_1)$. [3]

- the behavior of *Max-Red*$^{marg}$ can be explained as for *Max-Red*. In this case, the criterion tends to choose blocks having small size along one of their dimensions, and split them along this dimension, as this yields the maximum reduction of (marginal) SSE. This explains the shape of the partitions generated by *Max-Red*$^{marg}$ shown in Fig. 4.12(b) and (f), where rectangular blocks are split along their smallest dimension, and the obtained blocks are recursively split along the same dimension.
- adopting *MaxDiff* results in partitions which poorly describe the underlying data, as this criterion is unable to separate dense regions from null ones. In fact, there is no reason to assume that largest differences in marginal values arise correspondingly to the boundaries of dense regions.
- *Max-Var*/*Max-Red* succeeds in locating dense regions where a finer-grain partition of data is needed: as shown in Fig. 4.12(d) and (h) this criterion is fairer in choosing the region to be split w.r.t. *Max-Red*, *Max-Red*$^{marg}$, and *MaxDiff*. This explains why it outperforms all the other criteria for both *HBH* and *FBH*.

In Fig. 4.12 the partition obtained with *Max-Var*$^{marg}$/*Max-Red*$^{marg}$ is not shown, as it was very similar to that of *Max-Var*/*Max-Red*. Indeed, differences in error rates between these two criteria arise more significantly in higher dimensionality settings. As dimensionality increases, marginal distributions contain less and less information on the internal content of blocks: in fact, the total size of marginal distributions of a block (i.e. the sum of the lengths of all marginal distributions of the block) grows linearly with $d$ (it is $O(d \cdot n)$), whereas the volume of blocks grows exponentially with $d$ (it is $O(n^d)$). Therefore, investigating the content of marginal distributions to decide whether a block needs to be partitioned is likely to provide less reliable information as dimensionality increases. As a matter of fact, isolated dense regions of the multi-dimensional space can collapse into a single mono-dimensional dense region, when projected on each dimension (for instance, consider two adjacent circular dense regions located at the ends of a diagonal in the 2D space). Therefore, it is unlikely to succeed in isolating the dense multi-dimensional regions by only looking at their projections (i.e. the marginal distributions) on space dimensions.

We now focus our attention on explaining why only *Max-Var*/*Max-Red* and *Max-Red* benefits from the adoption of *HBH* representation model. As regards *Max-Var*/*Max-Red*, the reason of this is that MBRs do not help this criterion

[3] As a hint, consider two buckets $b_1$, $b_2$ with $vol(b_1) \ll vol(b_2)$ containing the same data values distributed differently. Then $SSE(b_2) \gg SSE(b_1)$ holds, but the reduction of SSE due to any binary split $< b_1^l, b_1^h >$ of $b_1$ is likely to be larger w.r.t. any binary split $< b_2^l, b_2^h >$ of $b_2$.

in isolating dense regions from null ones: therefore, *Max-Var/Max-Red* can exploit significantly the increase in number of buckets due to the *HBH* representation model, investing a larger number of buckets to approximate dense regions in more detail (in fact, error rates for *Max-Var/Max-Red* are very sensitive to an increase in storage space, as shown in diagrams in Fig. 4.11). As regards *Max-Red*, on the one hand MBRs do not prevent this criterion from generating several small buckets in a few dense regions. On the other hand, the use of MBR reduces the number of splits, so that a larger number of dense regions are likely to be disregarded by this criterion (for instance, the dense region at the bottom right-hand part of the data distribution in Fig. 4.12(e) is not partitioned when MBRs are used).

On the contrary, all other criteria are not effective in assigning distinct dense regions to distinct buckets: in this case, MBRs do provide a more accurate description of the data underlying buckets, but this positive benefit of *FBH* representation model turns out to be counterbalanced by the smaller number of buckets w.r.t. the *HBH* representation model.

Due to the higher level of accuracy provided by *Max-Var/Max-Red* under the tree-based representation model, and since this criterion can be evaluated as efficiently as the other ones (as stated in Theorem 4.9), in the following only *HBH* using this criterion will be considered.

### 4.4.5 Comparing *HBH* with *GHBH*

In this section we study how the introduction of a grid constraining block splits affects the accuracy of histograms. We first compare *GHBH* and *HBH* under the same greedy criterion (thus we adopt *Max-Var/Max-Red* under which *HBH* yields the best accuracy), then we briefly discuss the impact of adopting different greedy criteria to guide the construction of a *GHBH*.

#### *HBH* vs *GHBH* under *Max-Var/Max-Red*

*GHBH*s of different degrees have been tested. The term *GHBH*$(x)$ will be used to denote a *GHBH* which employs $x$ bits to store the splitting position. Diagrams in Fig. 4.13 (a,b) were obtained on Census data set by performing samples of queries whose selectivity is respectively 0.5% and 3%, whereas diagrams in 4.13 (c,d) were obtained on 4-dimensional synthetic data with volume $400 \times 400 \times 400 \times 400$ containing 30 dense regions.

From diagrams in Fig. 4.13 it emerges that *GHBH* yields higher accuracy than *HBH*. Although the *HBH* algorithm is able to perform more effective splits at each step (as splits are not constrained by the grid), the number of buckets generated by *GHBH* algorithms is larger.

In order to investigate in more detail how the optimal grid degree depends on the characteristics of data, we also performed experiments studying how

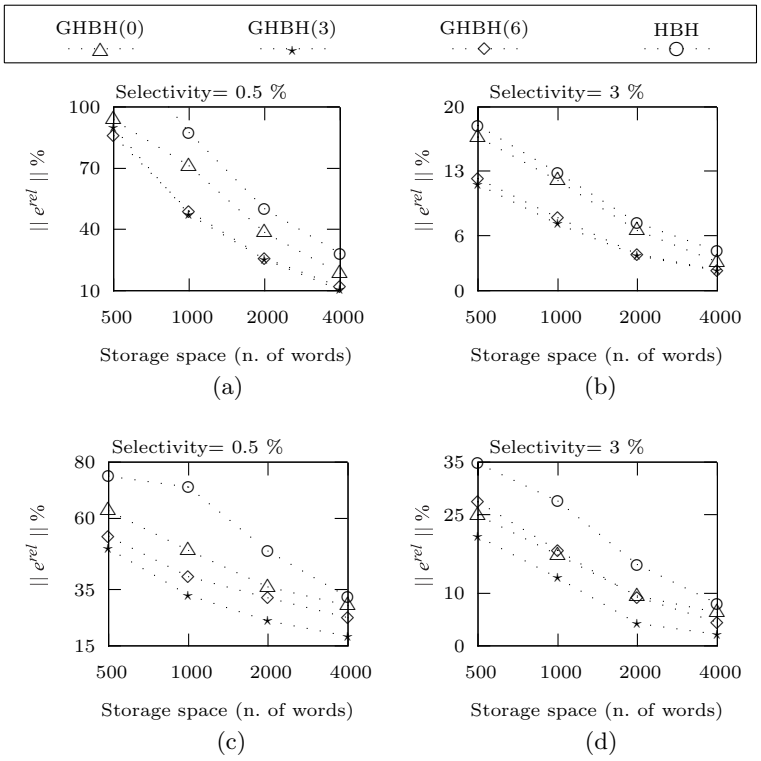


**Fig. 4.13.** *HBH* vs *GHBH* on real-life data (a,b) and synthetic data (c,d)

the accuracy of *GHBH*s adopting grids with different degrees is affected by either the size of the domain and the data skewness. As regards the former issue, intuition would suggest that, as the volume of data increases, in order to keep the same accuracy in partitioning data, a higher-degree grid should be adopted. To analyze this aspect, we performed some experiments investigating how changing the grid degree affects the effectiveness of isolating dense regions distributed on larger and larger domains.

To this aim, we tested *GHBH*s with different grid-degrees on data distributions having the same dimensionality, increasing volume, and containing the same dense regions differently distributed in the data domain. Diagram 4.14(a) depicts error rates on 3-dimensional cubic data sets with increasing edges, from 200 to 1600. These data sets will be denoted as $D^n$, where $n$ is the edge size. $D^{200}, \ldots, D^{1600}$ were generated by first creating 30 dense regions, and then distributing randomly the centers of these regions in the different data domains. For instance, each dense region $r_i$ (with $i \in [1..30]$) inside $D^{400}$ contains the same distribution of values as the region $r'_i$ inside $D^{200}$, but the center $c_i$ of $r_i$ has different coordinates w.r.t. the center $c'_i$ of $r'_i$ ($c_i$ and $c'_i$ being randomly selected points inside $D^{400}$ and $D^{200}$, respectively.). For each

data set $D^n$ a query set $QS^n$ is generated as follows. $QS^{200}$ contains, for each dense region $r_i$ of $D^{200}$, 1000 hypercubic queries intersecting $r_i$. The centers of these queries are characterized by their relative coordinates to $c_i$. $QS^{400}$ contains, for each $q \in QS^{200}$ involving $r_i$, a query $q'$ involving $r'_i$ with the same volume as $q$, and whose center has the same relative coordinates to $c'_i$ as $q$ does to $c_i$. Query sets $QS^{800}$, $QS^{1600}$ have been constructed analogously. Evaluating error rates w.r.t. these query sets allows us to establish whether the optimal grid degree for a *GHBH* depends on the domain size. Diagram 4.14(b) was obtained analogously, but for 6-dimensional data.

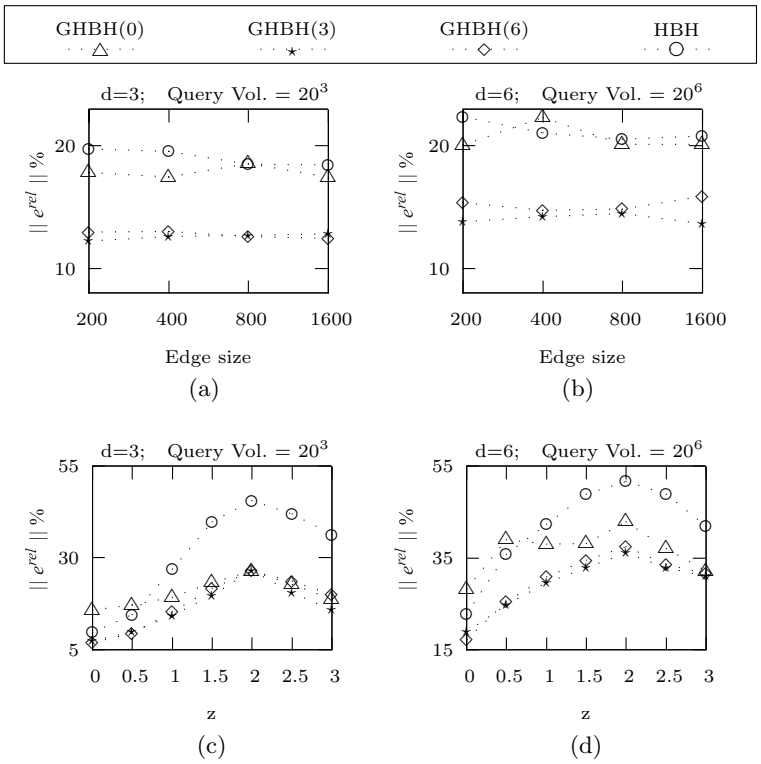


**Fig. 4.14.** Error rates of *HBH* and *GHBH* versus edge size (a,b) and data skewness (c,d)

Diagrams in Fig. 4.14(a,b) show that the effectiveness of adopting a particular grid degree is almost unaffected by the size of the domain.

Diagrams in Fig. 4.14(c,d) show how error rates depend on data skewness. Diagram in Fig. 4.14(c) depicts error rates for 3-dimensional data distributions with volume $400 \times 400 \times 400$ having 30 dense regions (having the same value

$z$ of the skew parameter), whereas diagram in Fig. 4.14(d) was obtained on 6-dimensional data distributions with volume $400^6$ with 30 dense regions. As for diagrams 4.14(a,b) we considered a query set consisting of queries with the same volume overlapping dense regions. Observe that data density decreases as skewness increases: as $z$ gets larger, dense regions become sparser, since tuples are generated with a decreasing probability of being far from the center of the region. That is, when $z = 0$ the distances of generated tuples from the center are uniformly distributed, while as $z$ becomes larger, large distances from the center become less and less probable. Going to the limit, dense regions collapse to a unique cell.

From diagrams in Fig. 4.14(c,d) it turns out that "intermediate" values of $z$ yield the largest error rates. Indeed, if skew is either very low or very high, dense regions will be rather uniform, or collapse, respectively. In both cases, isolating the dense region into a few buckets suffices to have good accuracy, whereas for intermediate values of $z$ dense regions need more and more splits to be accurately described by the partition.

From these results, we can draw the conclusion that the use of grids provides an effective trade-off between the accuracy of splits and the number of splits which can be generated within a given storage space bound. The effectiveness of this trade-off depends on the degree of the allowed binary splits. In fact, when a high degree is adopted, a single split can be very "effective" in partitioning a block, in the sense that it can produce a pair of blocks which are more homogeneous w.r.t. the case that the splitting position is constrained to be laid onto a coarser grid. On the other hand, the higher the degree of splits, the larger the amount of storage space needed to represent each split. From our results, it emerges that *GHBH*(3) (using binary splits of degree 8) generally gives the best performances in terms of accuracy, and as the number of bits used to define the grid increases, the accuracy decreases. However, we point out that *GHBH*s with small degree values do not exhibit large differences in error rates. Therefore, even if a value for the grid degree yielding the best accuracy in any setting cannot be found, this is not a limit of our approach, as any low-degree grid provides error rates which are close to those of the "optimal" degree. In the rest of the chapter, all results on *GHBH* will be presented by using 3 bits for storing splitting positions.

### *HBH* vs *GHBH* under different greedy criteria

It is worth noting that the improvement of *HBH* accuracy obtained by introducing the grid constraint is not simply orthogonal to any greedy criterion of the table in Fig. 4.7. Although we have discussed the benefits of using grids only when *Max-Var*/*Max-Red* is used, from several experiments it turned out that, in the case that another greedy criterion is adopted, the improvement in accuracy when moving from *HBH* to *GHBH* is not so relevant. We do not report related diagrams here, however, this result is rather expected, as it is

worth noting that *Max-Var/Max-Red* is the only criterion which improves significantly as the available storage space becomes larger (see Fig. 4.11), whereas the other criteria are less sensitive on this parameter. Therefore, it is unlikely that the other criteria exploit the increase of the number of available splits due to the introduction of grids as *Max-Var/Max-Red* does. Therefore in the following we will consider only *GHBH* adopting *Max-Var/Max-Red*.

### 4.4.6 *GHBH* versus Other Techniques

We compared the effectiveness of *GHBH* with the state-of-the-art techniques for compressing multi-dimensional data. Besides considering MHIST and MinSkew [4], we compared the accuracy of *GHBH* with GENHIST (see Sect. 2.2.4) and the wavelet-based techniques proposed respectively in [81] (that will be referred as WAVE1) and in [80] (WAVE2) (see Sect. 2.3; we recall that the former applies the wavelet transform directly on the source data, whereas the latter performs a pre-computation step on original data). The experiments were conducted at the same storage space.

Diagrams of Fig. 4.15 were obtained on four-dimensional synthetic data with volume $8 \times 16 \times 256 \times 1024$ with density 0.1%. In particular, diagrams (a,b) show how the accuracy of the techniques changes as the storage space increases, whereas diagrams (c,d) depict error rates w.r.t. the selectivity of queries.

*GHBH* exploits the increase of storage space better than the other techniques. Relative error rates of all techniques increase as query selectivity decreases: this can be easily explained by considering that as selectivity decreases (i.e. query answers become smaller in value) even a small difference between the actual query answer and the estimated one can lead to a large relative error.

Diagrams of Fig. 4.16 were obtained on the 10-dimensional Forest Cover data set. In particular, diagram 4.16(c) refers to very low selectivities and reports absolute errors. In this case relative errors have not been considered as, for small selectivities, they are likely be high even if a reasonable approximation is obtained: for instance, while a 300% error rate on a query with selectivity 1 corresponds to a good accuracy of the estimate, the same error rate on a query whose selectivity is high (w.r.t. $N$) makes the inaccuracy of the query estimate intolerable. Thus, relative error may not be indicative of the actual accuracy. In diagram 4.16(d) relative error rates at higher selectivities are reported. On this data set wavelet techniques were not considered as our prototype does not support data sets with so large volumes.

Diagrams of Fig. 4.17 were obtained similarly to diagrams in Fig. 4.16, but on the 8-dimensional Census data set. In particular, in this case we considered

[4] Our implementation of MinSkew is a straightforward extension of MinSkew to the multidimensional case, where each bucket stores its sum and the coordinates of its MBR.

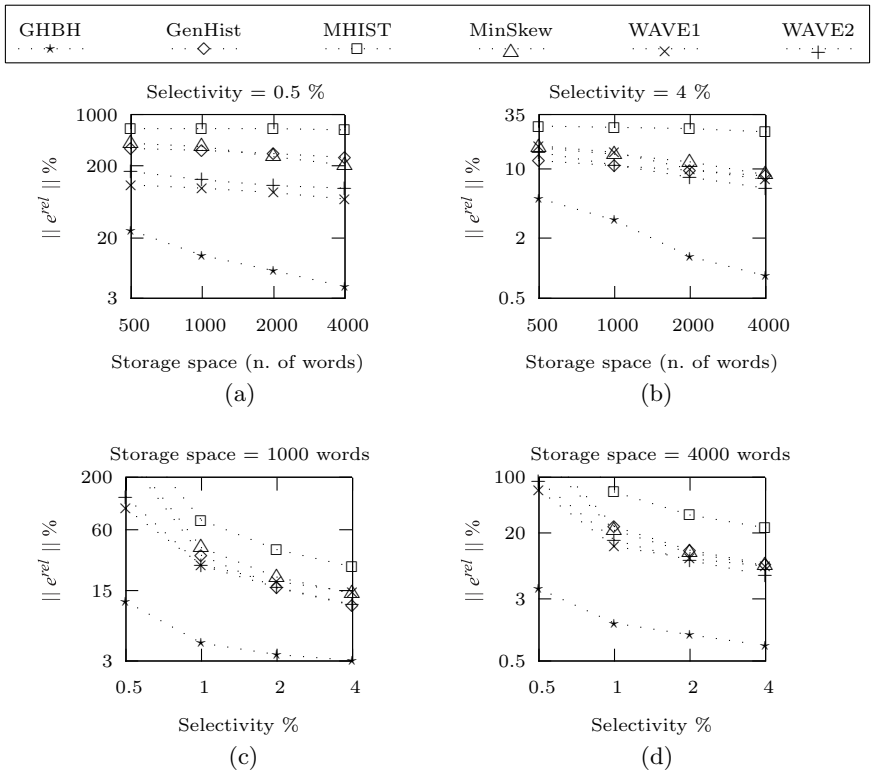


**Fig. 4.15.** Comparing techniques on synthetic data

a broader range of selectivities, as this data set is less sparse than Forest Cover and thus queries with very low selectivities are unlikely to occur.

All the diagrams in Figs. 4.15, 4.16 and 4.17 show that *GHBH* adopting *Max-Var/Max-Red* outperforms the other techniques on all the examined data sets.

We also investigated in detail how the accuracy of the various techniques is affected by an increase in dimensionality of the input data. In particular, diagrams 4.18(a) and (b) refer to synthetic data. These diagrams were obtained by starting from a 10-dimensional data distribution (called $D^{10}$) containing $10^{20}$ cells (the size of each dimension is equal to 100), where about 53000 non null values (density $\simeq 5.3 \cdot 10^{-16}$) are distributed among 1000 dense regions. The data distributions with lower dimensionality (called $D^i$, with $i \in 4..9$) were generated by projecting the values of $D^{10}$ on the first $i$ of its dimensions. By means of this strategy, we created a sequence of multi-dimensional data distributions, with increasing dimensionality (from 4 to 10) and with decreasing density (from $3.9 \cdot 10^{-4}$ to $5.3 \cdot 10^{-16}$). Diagrams 4.18(a) and (b) were

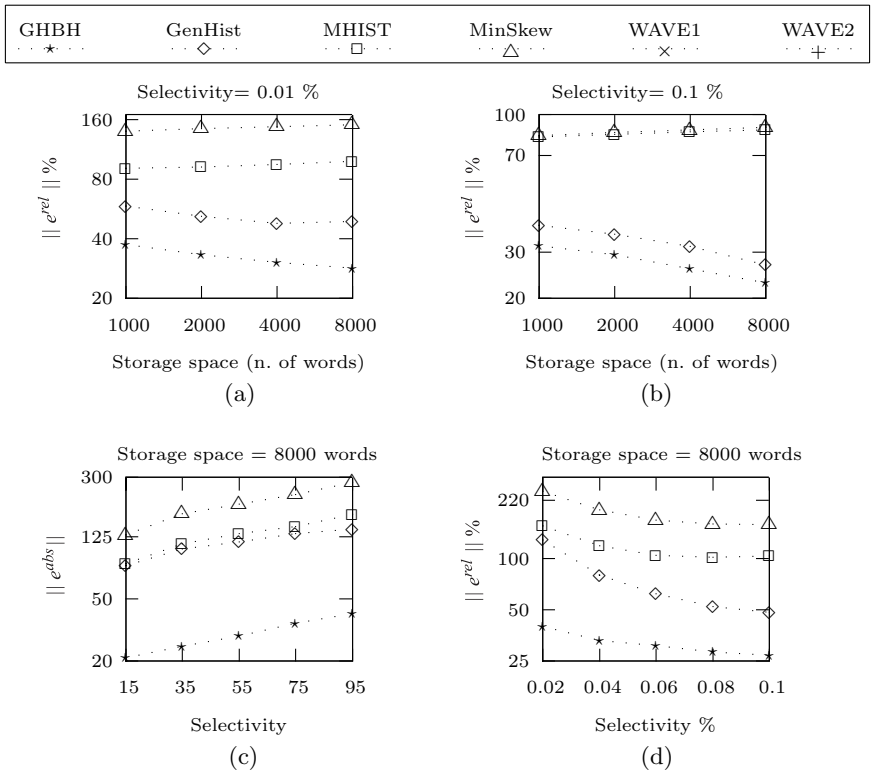


**Fig. 4.16.** Comparing techniques on Forest Cover

obtained by considering, for each $D^i$, a sample of range queries whose selectivity is respectively 0.1% and 1%. Both these diagrams have been obtained using a storage space of 2000 words. The same kind of experiments were performed on Census data set. In this case we projected the 8-dimensional data set described in 4.4.3 on the first $i$ of its dimensions ($i = 5..7$). In Fig. 4.18(c) and (d) results obtained on samples of queries having selectivity 0.1% and 1% (respectively) are depicted.

Both WAVE1 and WAVE2 were not considered on synthetic data, as our prototype does not work on so large data sets. Error rates for WAVE1 are not reported in the diagram in Fig. 4.18(c) as they were out of scale. Diagrams in Fig. 4.18 show that, for both synthetic and real-life data, error rates of every technique tend to increase as dimensionality increases, but *GHBH* accuracy gets worse very slightly and outperforms all the other techniques at all considered dimensionalities.

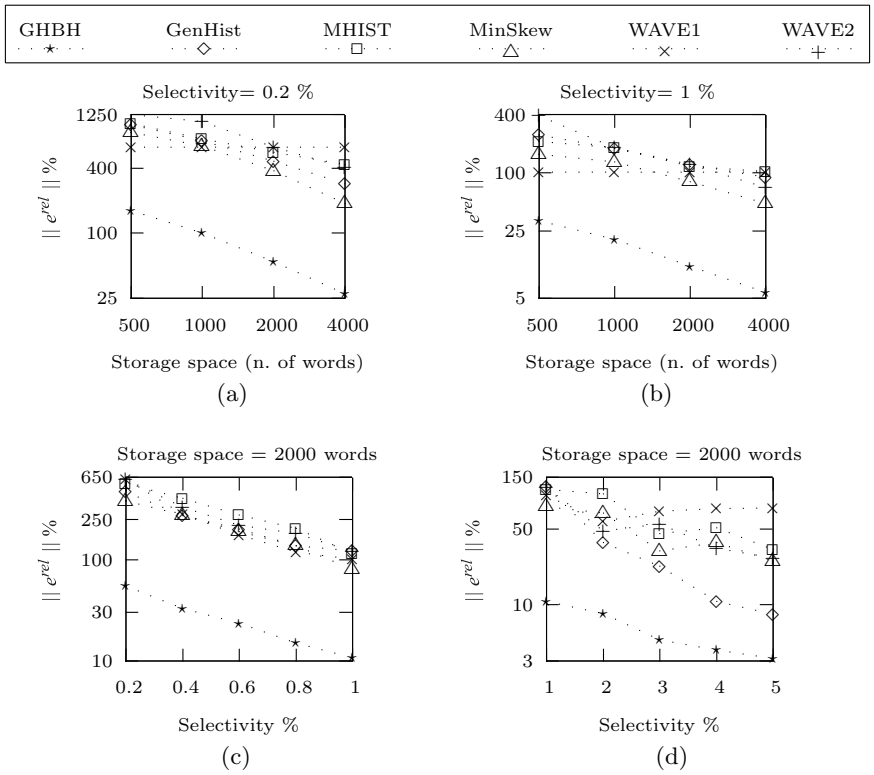


**Fig. 4.17.** Comparing techniques on Census

### 4.4.7 Execution Time of Greedy Algorithm

In this section we present some experimental results studying how the execution times of Greedy Algorithms constructing a *GHBH* depend on several parameters, such as the storage space (i.e. number of buckets), the density, the volume, the dimensionality of $D$, and the grid degree. In particular, we have compared the execution times when either the sparse data model, or the non-sparse one, or pre-computation is adopted.
Diagrams in Fig. 4.19 have been obtained for greedy algorithms adopting the *Max-Var/Max-Red* constructing an 8-*GHBH*.
Experimental results reported in Fig. 4.19 are basically consistent with the complexity bounds of Fig. 4.10, and can be summarized as follows:

- when the sparse model is used, the execution time of Greedy Algorithm is sensitive only on the number of non-null values in $D$ (it grows linearly with $N$ - see Fig. 4.19(a)), but is almost independent on either the data domain volume - Fig. 4.19(b) - and the dimensionality - Fig. 4.19(d);
- otherwise (if either the non-sparse data model is adopted or pre-computation is performed) the execution time of Greedy Algorithm is unaffected by an

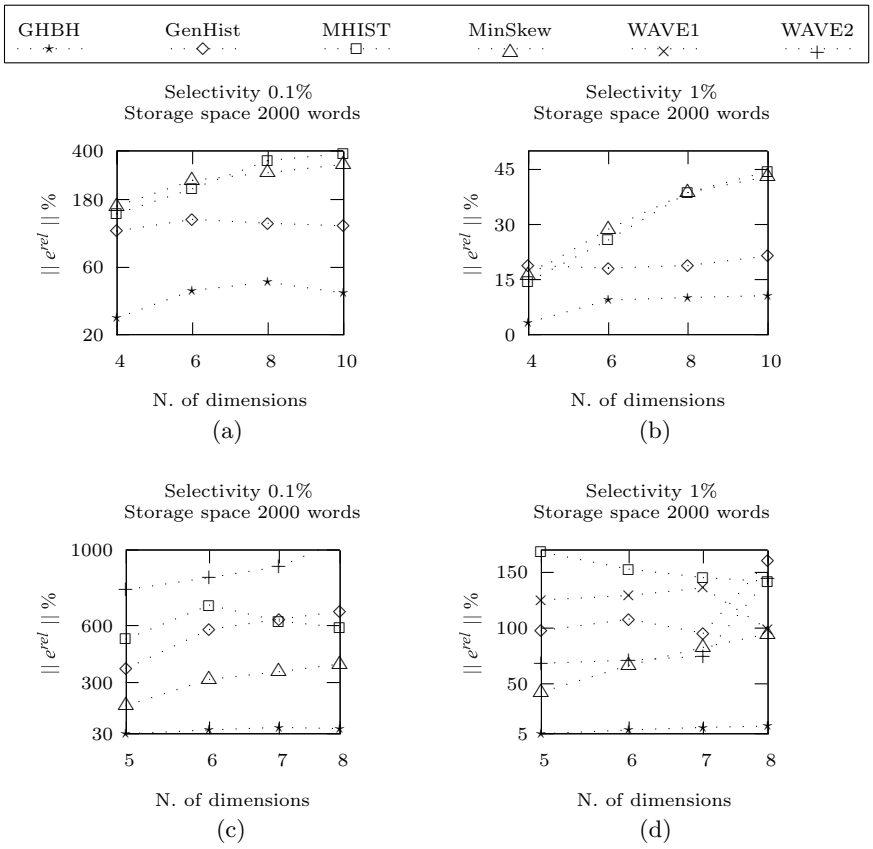


**Fig. 4.18.** Sensitivity to dimensionality on synthetic data (a,b) and Census data (c,d)

increase in $N$, but it worsens dramatically as either $d$ or $n^d$ increases; in particular, the algorithm using pre-computation is faster than the one adopting the non-sparse model without pre-computation;

- when the sparse data model is used, if data density is smaller than a threshold, Greedy Algorithm is faster w.r.t. the case of non-sparse model or pre-computation. For instance, in the case depicted in Fig. 4.19(a), if data density is smaller that $\rho^* = 3\%$ the adoption of the sparse data model provides better performances than the use of pre-computation. Indeed, the exact value of the data density $\rho^*$ where the execution times of the different approaches (based on either the sparse data model or the use of pre-computation) are about the same depends on a lot of parameters. In particular, it is worth noting that as the dimensionality increases, algorithms performing pre-computation or adopting the non-sparse data model slow down dramatically (as shown in

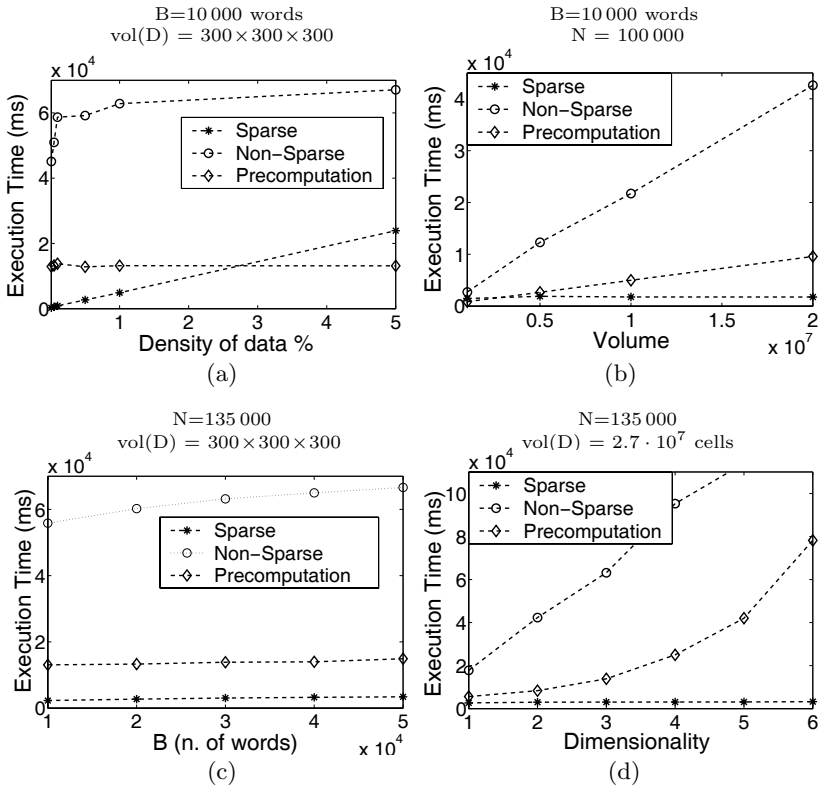


**Fig. 4.19.** Comparing efficiency of algorithms

Fig. 4.19(d)) so that the data density threshold gets a much larger value. However, in practical scenarios, especially for high-dimensionality data distributions, the data density is so small that algorithms based on the sparse data model do perform much better than the others.

Figure 4.19(c) shows that execution times of greedy algorithms is not very sensitive w.r.t. the size of the available storage space. This can be explained as follows:

- in the case that pre-computation is performed, the cost of the pre-computation step dominates the construction of the histogram. As explained in Sect. 4.3.2, the pre-computation step makes *Evaluate* more efficient to be performed, so that even if the number of buckets to be built increases, the computational overhead needed to compute them is almost negligible (i.e. the number of invocations of *Evaluate* increases, but each invocation is fast to be accomplished);
- in the other cases, the largest portion of execution times is devoted to the computation of the "first" steps, which involve very large buckets. For instance, at the first step of the greedy algorithms, the function *Evaluate* has

to scan all values of $D$, that is either $n^d$ values (non-sparse data model) or $N$ values (sparse data model) must be accessed. As the construction process goes on, the buckets to be processed become smaller and smaller, so that the cost of performing further splits is almost negligible.

Experimental results analyzing how execution times depend on the grid degree are shown in Fig. 4.20. They have been obtained on a 3-dimensional data distribution with volume $300 \times 300 \times 300$ containing 135000 non null values, using a storage space of 10000 words.

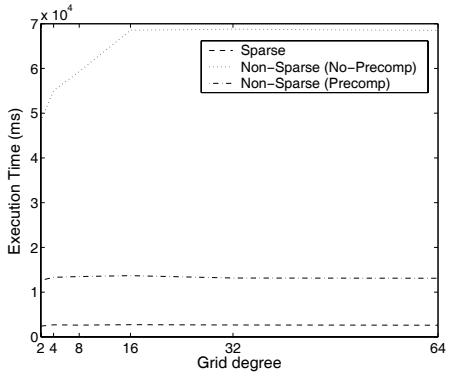


**Fig. 4.20.** Comparing efficiency of algorithms

From the diagram above it emerges that execution times slightly depend on the grid degree. This could be due to the fact that, as the grid degree increases, the number of splits to be evaluated at each step of the construction increases, but the number of steps of the algorithm decreases (as the number of buckets which can be stored within the given storage space decreases). We do not report experimental results on execution times of greedy algorithms constructing an *HBH*. However it is easy to see that an *HBH* can be viewed as a *GHBH* with high degree.

# 5

# Clustering-based Multi-dimensional Histograms

*In this chapter we propose a new technique for constructing multi-dimensional histograms based on data clustering. This technique first invokes a density-based clustering algorithm to locate dense and sparse regions of the input data. Then the data distribution inside each of these regions is summarized by means of a grid-based partitioning. The proposed approach is compared with state-of-the-art histograms on both synthetic and real-life data and is shown to be more effective.*

## 5.1 Introduction

As shown by the experiments presented in Chap. 4, existing approaches for summarizing multi-dimensional data often adopt ineffective strategies for guiding the histogram construction, thus yielding low accuracy in estimating range queries. Especially when data dimensionality increases, the adopted partitioning heuristics produce less and less accurate partitions, consisting of buckets containing inhomogeneous regions. In particular, traditional techniques for constructing histograms often result in partitions where dense and sparse regions are put together in the same bucket, which yield poor accuracy in describing data.

For instance, consider the bucket shown in Fig. 5.1, where a dense cluster is put together with a sparse region. As the bucket is summarized by the sum of its values, estimating the sum of the values in either $Q_1$ and $Q_2$ by performing linear interpolation gives a high error rate, since the total sum is assumed to be homogeneously distributed inside $b$. In fact, this assumption is far from being true: most of the sum of $b$ is concentrated in the dense cluster on the right-hand side of $b$.

Therefore, it's our belief that improving the ability of distinguishing dense regions can result in more accurate partitions, as this prevents buckets like that of Fig. 5.1 from being constructed. This is also proved by the experiments conducted in Chap. 4 (see diagrams in Sect. 4.4.4) where we have shown that

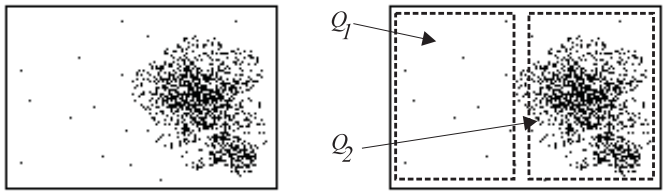


**Fig. 5.1.** Queries posed into a non-homogeneous bucket

the best performing partition criteria are those which are effective in isolating dense regions. Thus, we expect that enhanced estimation accuracy can be achieved by adopting a new approach: rather than searching for new criteria which are more effective in guiding the top-down data partitioning (so as to reach and isolate dense regions), we adopt a bottom-up approach which first locates dense regions and then further partition them.

The problem of searching homogeneous regions is very close to the *data clustering* problem, i.e. the problem of grouping database objects into a set of meaningful classes. This issue has been widely studied in the data mining context, and several algorithms accomplishing data clustering have been proposed. They can be divided into partitional, hierarchical, and locality-based algorithms. In *partitional* algorithms the partition of data points into clusters is chosen which optimizes an objective criterion, such as distance. Each cluster is represented by either the mean of its records (k-means [59]) or by one of its records chosen as representative (k-medoid [54]). CLARANS [68] is an extension of traditional k-medoid algorithms yielding higher accuracy in locating clusters (even if it could converge to a local optimum), but is not well-suited for large databases as it may require multiple scans of the data points.

*Hierarchical* algorithms construct a hierarchy of clusters by adopting either a top-down strategy (divisive hierarchical techniques) or a bottom-up one (agglomerative strategies). BIRCH [83] and CURE [41] are examples of hierarchical algorithms. The former first populates a special data structure (namely, *CF Tree*) where summary information of sub-clusters of objects is stored, and then runs an agglomerative algorithm on the previously generated sub-clusters. It is known to be unsuitable for distributions consisting of arbitrary shaped clusters or clusters having different sizes. On the contrary, CURE succeeds in identifying clusters having complex shapes with different sizes, and outperforms BIRCH on large databases. It uses a combination of random sampling and partitioning, where clusters are characterized by a set of representative points, instead of a single centroid (as BIRCH does).

DBSCAN [27] is a *density-based* algorithm aiming at separating dense regions from sparse ones. A cluster is built progressively by starting from

a core-point (i.e. a point having a dense neighborhood), inserting all of its neighbors into the cluster, and then expanding the cluster from all core-points included into the clusters at some previous step. OPTICS [6] is an extension of DBSCAN producing an augmented ordering of records representing its density-based clustering structure. A detailed survey on clustering techniques can be found in [54].

The approach proposed in this chapter aims at enhancing the histogram construction by exploiting the capability of clustering techniques to locate dense regions. We define a new technique for constructing multi-dimensional histograms which first invokes a density-based clustering algorithm for partitioning the data into dense and sparse regions, and then further refines this partitioning by adopting a grid-based paradigm. We study, by means of experiments, the accuracy of the proposed histogram in answering sum-range queries over either synthetic and real life data sets.

## 5.2 CHist: Clustering-based Histogram

Our technique works in three steps. At the first step clusters of data and outliers (i.e. points which do not belong to any cluster) are located. At the second step, these clusters and the set of outliers are treated as distinct layers, and each layer is summarized by partitioning it according to a grid-based paradigm. At the last step the histogram is constructed by "assembling" all the buckets obtained at the previous step.

The three phases of our approach are described in detail in the following sections. The description of the algorithm is provided by assuming a $d$-dimensional data distribution $D$. $D$ will be treated as a multi-dimensional array of integers of size $n^d$ (see Sect. 1.4). The amount of available storage space for the representation of the histogram will be denoted as $B$.

### 5.2.1 Step I: Clustering Data

In our prototype, we have embedded the clustering algorithm DBSCAN [27] in order to group input data into dense clusters. Indeed, our approach can be viewed as orthogonal to any clustering technique: we have chosen DBSCAN as it is representative of density-based clustering algorithms.

The idea underlying DBSCAN is that points belonging to a dense cluster (except those points lying on the border of the cluster) have a dense neighborhood. A point $p$ is said to have a dense neighborhood if there are at least *MinPts* distinct points whose distance from $p$ is less than *Eps* (both *Eps* and *MinPts* are parameters crucial for the definition of clusters). Points with a dense neighborhood are said to be *core points*. DBSCAN scans input data searching for core points. Once a core point $p$ is found, a new cluster $C$ is created, and both $p$ and all of its neighbors are grouped into $C$. Then $C$ is recursively expanded by including the neighbors of all core points put in $C$

at the last step. When $C$ cannot be further expanded, DBSCAN searches for other core points to start new clusters, until no more core points can be found. At the end of the clustering, points which do not belong to any clusters are classified as *outliers*. Figure 5.2 shows an example of clustering obtained by DBSCAN.

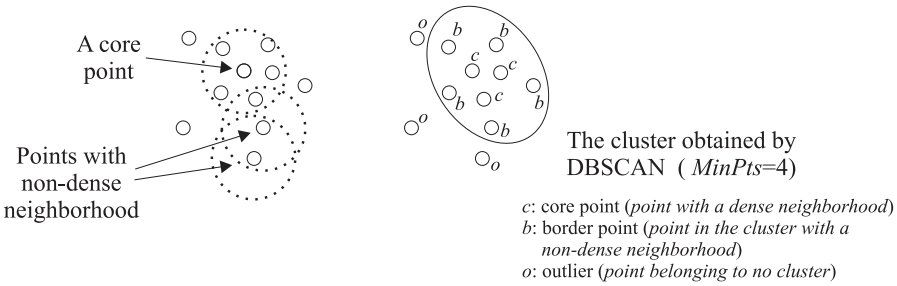


**Fig. 5.2.** Running DBSCAN on a set of points

### 5.2.2 Step II: Summarizing Data into Buckets

At this step the input data distribution is viewed as a superposition of layers. Each layer is either a cluster or the set of outliers. In the following we will denote the layer consisting of outliers as $L_0$, and the layers corresponding to dense clusters as $L_1, \ldots, L_c$. $L_0$ will be said to be the *outlier layer*, whereas $L_1, \ldots, L_c$ will be said to be *cluster layers*. Each layer is represented by means of its MBR.

The different layers are summarized separately by partitioning their MBRs into buckets. This aims at preventing the construction of buckets where dense and sparse regions are put together, which, as explained before (see Fig. 5.1), can yield poor accuracy. In more detail, our approach works as follows.

**(Step II.a)** Before summarizing the layers into buckets, possible peaks are located among the set of outliers. The rationale of this step can be explained as follows. Due to the sparsity of the outlier layer, any summarization of $L_0$ is likely to consist of buckets larger in volume than buckets constructed on cluster-layers. Obviously, the larger in volume a bucket, the less accurate its description of the underlying data distribution: aggregating two or more points into a bucket means spreading the total value of these points onto the range of the bucket, thus the larger this volume, the less localized the information associated to these points is kept. Therefore, if some peaks occur in the set of outliers, aggregating them with small-value outliers can result in a relevant loss of accuracy.

In order to detect peaks we use a threshold parameter (namely, $t$) to decide whether an outlier is a peak or not: if the value of an outlier $o$ is greater

than $t$ times the average value of input points, then $o$ is a peak, and will be removed from $L_0$ and stored in detail (this can be viewed as creating buckets containing single points). In our experiments we used the value $t = 3$.

**(Step II.b)** The amount of storage space left from the representation of peaks is invested to summarize the clusters and outliers not previously selected. Layers are summarized independently of each other, and the summary of the whole data distribution will be the superimposition of the summaries of all layers.

The summarization of layers is accomplished by a multi-step algorithm which, at each step, summarizes a single layer by partitioning it according to a grid and storing, for each bucket defined by this grid, both its MBR and the sum of its values (obviously, the cells of this grid which do not contain any data point result in an empty MBR which is not stored). The MBRs of buckets obtained from the summarization of cluster layers will be said to be *c-buckets*, whereas the MBRs of the buckets constructed by partitioning $L_0$ will be said to be *o-buckets*.

Indeed, layer $L_0$ is processed after the summarization of all the cluster layers. In particular, before summarizing the outlier layer, we scan all outliers to locate those lying onto the range of some c-bucket. Each outlier $o$ which lies onto some c-bucket is removed from $L_0$ and "added" to one c-bucket whose range contains the coordinates of $o$ [1]. This allows us to view c-buckets as "holes" of $L_0$, in the sense that, after performing this task, there are no points lying onto the range of some c-bucket which belong to $L_0$. As it will be clear in the following, this will be exploited in the physical representation of the histogram to improve its accuracy.

We now describe how the available storage space is used to summarize layers. Let $B_i$ be the amount of memory which is left from the $i-1$ previous summarization steps (at the first step, $B_1$ is the residual of the initial amount of storage space which is left from the representation of peaks). The portion of $B_i$ which is invested to summarize $L_i$ is denoted as $B(L_i)$ and is computed by comparing the need of being partitioned of $L_i$ with all remaining layers $L_{i+1}, \ldots, L_c, L_0$. The need of being partitioned of a layer $L$ is estimated by computing its SSE (denoted as $SSE(L)$), thus $B(L_i) = B_i \cdot \frac{SSE(L_i)}{SSE(L_0)+\sum_{j=i+1}^{c} SSE(L_j)}$.

We now show how $B(L_i)$ is exploited to store a partition of $L_i$ into buckets. The idea is to partition $L_i$ according to a grid and store, for each cell of the grid containing at least one point, the coordinates of its MBR and the sum of the values occurring in it. The grid on a layer $L_i$ is constructed as follows.

[1] If more than one c-bucket contains $o$, one of these c-buckets is randomly selected to incorporate $o$. Adding an outlier $o$ to a c-bucket $b$ means removing $o$ from $L_0$ and adding the value of $o$ to $sum(b)$.

If we denote as $W$ the amount of storage space needed to store a bucket[2], the number of buckets produced by the grid on $L_i$ can be no more than $nb = \lfloor \frac{B(L_i)}{W} \rfloor$. Thus, if $t_j$ is the number of divisions of the grid along the $j$-th dimension of $L_i$, it should be $\prod_{j=1}^{d} t_j = nb$.

A grid could be easily constructed by means of an equi-partitioning strategy, i.e. by partitioning all the dimensions in the same number of portions. That is, we could choose $t_1 = t_2 = \ldots = t_d = \sqrt[d]{nb}$. Indeed, this choice can result in a grid whose cells are hyper-rectangles having edges with large differences in length, unless the MBR of $L_i$ has edges with about the same size. In fact, a partition where, for each bucket, there are large differences in edge sizes is likely to provide poor accuracy, as it is less effective in preserving the locality of information. This is due to the fact that summarizing the points inside the bucket into a single value means spreading each point onto the whole range of the domain delimited by the bucket itself. Therefore each point can give a contribution to cells of the domain whose distance is bounded by the maximum diagonal $d_{max}$ of the bucket. The value of $d_{max}$ is of the same order of magnitude of the longest edge of the bucket, thus buckets whose edges are close in length have a smaller value of $d_{max}$ w.r.t. buckets (with the same volume) having edges with large differences in length.

For instance, consider the two buckets $b'$ of size $10 \times 10 \times 10$, and $b''$ of size $100 \times 5 \times 2$. $b'$ and $b''$ have the same volume. The value of the maximal diagonal for $b'$ is $d'_{max} = \sqrt{10^2 + 10^2 + 10^2} \cong 17.1$, whereas the maximal diagonal for $b''$ is $d''_{max} = \sqrt{100^2 + 5^2 + 2^2} \cong 100.1$. Therefore, after performing summarization, the information of a point at a vertex of the range of $b'$ is kept more localized w.r.t. a point at a vertex of the range of $b''$.

In order to prevent large differences in value between bucket edges, we partition each edge of the MBR of the layer to be summarized into a number of portions which is proportional to the length of the edge itself.
Let $w_j$ be the length of the edge along the $j$-th dimension, and $t_j$ be the number of divisions performed along the same dimension. Choosing $t_j = w_j \cdot \sqrt[d]{\frac{nb}{vol(L_i)}}$ (where $vol(L_i)$ is the volume of the MBR of $L_i$), guarantees both that $\prod t_j = nb$ and that the grid degree along each dimension is chosen by weighting the corresponding edge size. Indeed, this formula can result in non-integer value coefficients $t_1, \ldots, t_d$. Therefore $t_1, \ldots, t_d$ must be rounded, with the constraint that their product cannot be larger than $nb$. This can be accomplished by rounding each $t_j$ to $\lfloor t_j \rfloor$, but this may result in a grid with much fewer cells than $nb$. For instance, assume that $nb = 120$, and $t_1 = 2.66$, $t_2 = 7.5$, $t_3 = 2.4$, $t_4 = 2.5$. Observe that $\prod_{j=1}^{4} t_j = 120$, but $\prod_{j=1}^{4} \lfloor t_j \rfloor = 16$ .

Therefore we use the following strategy to construct the grid. The degrees of the grid along each dimension are computed progressively, starting from $t_1$ to $t_d$, according to the following scheme:

[2] We use $2 \cdot d$ 32-bit words for storing bucket boundaries, and one 32-bit word for storing the sum-aggregate

$t'_1 = max\{\lfloor t_1 \rfloor, 1\}; \quad t'_2 = max\left\{\left\lfloor \frac{t_1 \cdot t_2}{t'_1} \right\rfloor, 1\right\}; \quad \ldots \quad t'_d = max\left\{\left\lfloor \frac{\prod_{j=1}^{d} t_j}{\prod_{j=1}^{d-1} t'_j} \right\rfloor, 1\right\}.$

That is, the value of each $t_j$ is approximated to $t'_j$ by taking into account the approximations already performed at the $j-1$ previous steps. For instance, consider a 4-dimensional layer $L_i$ whose MBR has size $30 \times 60 \times 120 \times 240$. Let $Vol = 30 \cdot 60 \cdot 120 \cdot 240$ be the volume of the MBR of $L_i$, $nb = 100$ be the number of buckets which can be constructed on $L_i$, and $K = \sqrt[d]{\frac{nb}{Vol}}$. We have that:

$t_1 = 30 \cdot K = 1.118; \quad t_2 = 60 \cdot K = 2.236; \quad t_3 = 120 \cdot K = 4.472; \quad t_4 = 240 \cdot K = 8.944$

and:

$t'_1 = \lfloor t_1 \rfloor = 1; \quad t'_2 = \left\lfloor \frac{1.118 \cdot 2.236}{1} \right\rfloor = 2;$

$t'_3 = \left\lfloor \frac{1.118 \cdot 2.236 \cdot 4.472}{1 \cdot 2} \right\rfloor = 5; \quad t'_4 = \left\lfloor \frac{1.118 \cdot 2.236 \cdot 4.472 \cdot 8.944}{1 \cdot 2 \cdot 5} \right\rfloor = 10.$

Although in the case shown above $t'_1 \cdot t'_2 \cdot t'_3 \cdot t'_4 = nb$, it can happen that constructing a grid using this strategy results in $nb'$ buckets, with $nb'$ strictly less than $nb$. This can be due either to numerical approximation (the value of $\prod t'_j$ can be less than $nb$), or to the fact that some cells of the grid can correspond to null regions of the data domain, so that they are not stored explicitly. Therefore, after a layer $L_i$ is summarized, the residual amount of storage space which will be available at step $i+1$ is given by $B_{i+1} = B_i - nb' \cdot W$ (that is, if some space which was assigned to the summarization of $L_i$ has not been consumed, it is re-invested at the following steps).

### 5.2.3 Step III: Representation of the Histogram

The strategy adopted to summarize layers can yield overlapping buckets. In particular, buckets aggregating points of $L_0$ (the layer consisting of outliers) are likely to be larger than buckets describing clusters. Therefore, several c-buckets $b_1, \ldots, b_k$ can lie onto the range of an o-bucket $b$. In this scenario $b_1, \ldots, b_k$ can be viewed as "holes" of $b$, as the aggregate information associated to $b$ does not refer to points contained inside $b_1, \ldots, b_k$. We now show how this observation can be exploited to make query estimation more accurate. In the following, given an o-bucket $b$, the set of c-buckets completely contained into $b$ will be denoted as $Holes(b)$.

Consider the scenario depicted in Fig. 5.3(a), where the query $Q_1$ intersects one half of the range associated to the bucket $b$. Adopting linear interpolation to estimate $Q_1$ returns: $\widetilde{Q}_1 = \frac{vol(Q_1 \cap b)}{vol(b)} \cdot sum(b)$, where $Q_1 \cap b$ refers to the intersection between the query range and the range of $b$. In fact, points belonging to the ranges of $b_1, \ldots, b_9$ give no contribution to the value of $sum(b)$. Therefore, a more precise estimate for $Q_1$ is: $\widetilde{Q}_1 = \frac{vol(Q_1 \cap b)}{vol(b) - vol(b_1, \ldots, b_9)} \cdot sum(b)$, where $vol(b_1, \ldots, b_9)$ denotes the volume of the range underlying the buckets $b_1, \ldots, b_9$. Likewise, the bucket $b$ should give no contribution to the estimate of the query $Q_2$ in Fig. 5.3(b), which lies completely on the range underlying the buckets $b_1, \ldots, b_9$.

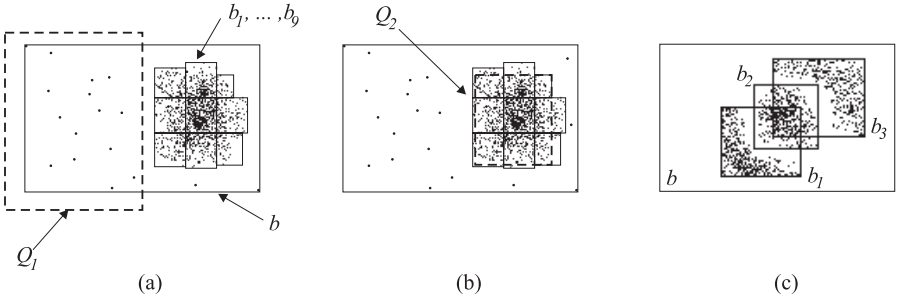


**Fig. 5.3.** O-buckets with holes

In the following the number of cells of an o-bucket $b$ which are not contained in any hole of $b$ will be said to be the *actual volume* of $b$. In the case depicted in Fig. 5.3(a) evaluating the actual volume of $b$ can be accomplished efficiently, as $b_1, \ldots, b_9$ do not overlap. Indeed, also c-buckets inside an o-bucket $b$ can intersect one another [3]. For instance, in Fig. 5.3(c) the three buckets $b_1, b_2, b_3$ inside $b$ overlap. In this case computing the actual volume of $b$ requires $vol(b_1)$, $vol(b_2)$, $vol(b_3)$, $vol(b_1 \cap b_2)$, $vol(b_2 \cap b_3)$ and $vol(b_1 \cap b_2 \cap b_3)$ to be computed. This computation becomes more and more complex when more buckets intersect in the same region: we need to compute the volumes of all the intersections between 2 holes, 3 holes, and so on. Obviously, this slows down query estimations. Due to this reason, we prefer to estimate the actual volume of an o-bucket $b$ involved in a query instead of evaluating its exact value: To this end we consider only a maximal subset of $Holes(b)$ (denoted as $NOHoles(b)$) consisting of non-overlapping c-buckets, thus avoiding intersections between holes to be computed. For instance, in the case depicted in Fig. 5.4(a) we can estimate the actual volume of $b$ as $vol(b) - vol(b_1) - vol(b_3)$. However we point out that from our experiments on real-life data it turned out that intersections between c-buckets are unlikely to occur.

The adopted representation model partitions buckets into two levels. The buckets at the second level are those belonging to $NOHoles(b)$ for some $b$. The first level consists of all the other buckets.

The physical representation model can be exploited to evaluate query answers efficiently. The answer to a given sum range query $Q$ is computed as follows:
1) the first-level buckets whose range overlap the query range are located;
2) for each of these buckets $b$, all of its hole-buckets $b_1, \ldots, b_k$ are accessed and those involved in the query are located. Then, the contribution of both $b$ and its holes to the query estimate are evaluated. In particular, for each hole $b_i$

[3] Although no pair of clusters $C_1$, $C_2$ can overlap (otherwise $C_1$, $C_2$ would be a unique cluster), MBRs of clusters can overlap (see Fig. 5.3(c)). Thus, partitioning overlapping MBRs can result in overlapping c-buckets.

the contribution to the estimate of $Q$ is given by: $\frac{vol(b_i \cap Q)}{vol(b_i)} \cdot sum(b_i)$, whereas the contribution of $b$ is $\frac{vol(b \cap Q) - \sum_{i=1}^{k} vol(b_i \cap Q)}{vol(b) - \sum_{i=1}^{k} vol(b_i)} \cdot sum(b)$.

For instance, the estimate of the query $Q$ shown in Fig. 5.4(a), according to the representation shown in Fig. 5.4(b), is given by: $\widetilde{Q} = \frac{vol(b \cap Q) - vol(b_1 \cap Q)}{vol(b) - vol(b_1) - vol(b_3)} \cdot sum(b) + sum(b_1) + \frac{vol(b_2 \cap Q)}{vol(b_2)} \cdot sum(b_2)$. Therefore the adopted representation scheme enables range query answers to be estimated by accessing each bucket at most once.

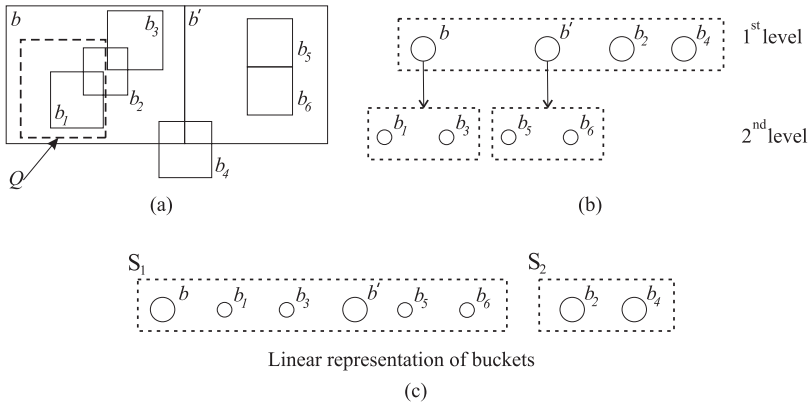


**Fig. 5.4.** Nested representation of buckets

Observe that representing some c-buckets as holes of o-buckets introduces no spatial overhead on the representation of o-buckets. That is, the two-levels organization of buckets can be linearized by representing buckets into two distinct sequences $S_1$, $S_2$. In particular, $S_1$ contains all o-buckets and their non-overlapping holes: each o-bucket $b$ is followed by the representation of c-buckets in $NOHoles(b)$ (see Fig. 5.4(c)). Thus, locating non-overlapping holes of an o-bucket $b$ at position $i$ in this sequence can be accomplished by scanning the positions of the sequence following $i$, till either the end of the sequence or an o-bucket having an empty intersection with $b$ is reached (for instance, the holes $b_1$, $b_3$ of $b$ occur in the sequence between $b$ and $b'$). Sequence $S_2$ contains all c-buckets which do not belong to any $NOHoles(b)$ for any o-bucket $b$.

This explains why we do not consider c-buckets which partially overlap o-buckets as holes. For instance, in the case of Fig. 5.4(a), bucket $b_4$ is not taken into account to estimate the actual volumes of $b$ and $b'$. Otherwise we should insert into both the representations of $b$ and $b'$ a reference to $b_4$ (which cannot be accomplished by a sequential physical representation of the histogram), and moreover bucket $b_4$ should be accessed more than once to estimate queries involving $b$ and $b'$.

The idea of representing some buckets as holes of other buckets was introduced with STHoles histograms in [15] (see Sect. 2.2.6) . We recall that in this

technique, holes of buckets are determined by query results feedback; if two holes overlap, one of them is shrunk; the aggregate information of the portion of the hole which has been cut off is estimated by linear interpolation and spread onto the overlying bucket. This approach is not suitable in our context as our holes are dense buckets, so that spreading their aggregate information onto an o-bucket (which is often much larger and sparser) may result in a severe loss of accuracy.

The algorithm implementing steps I, II, III is as follows.

**Algorithm** CHIST
*INPUT*: $D$: a multi-dimensional data distribution; $B$: number of words available to store the histogram;
*OUTPUT*: $H$: a histogram on $D$ within $B$;

*Step I*
L := DBSCAN(D); // L is the array of layers resulting from the execution of DBSCAN;
*Step II.a*
$PBuckets = Peaks(L[0])$; // Peaks are detected among outliers and removed from $L[0]$;
// *PBuckets* is the set of buckets associated to these peaks;
$B = B - size(PBuckets)$;
$H = PBuckets$; // the initial histogram consists of the buckets representing peaks;
*Step II.b*
$SSE_{tot} = \sum_{i=0}^{L.size-1} SSE(L[i])$;
*C-Buckets*= $\emptyset$; // this set will contain all the buckets constructed on
// layers containing clusters;
FOR i:=1 TO L.size-1 DO
$B_{L_i} = \frac{SSE(L[i])}{SSE_{tot}} \cdot (B - size(C\text{-}Buckets))$; // $B_{L_i}$ is the amount of memory which
// will be used to summarize the layer $L_i$;
*C-Buckets*=*C-Buckets*$\cup$ $GridPartition(L[i], B_{L_i})$; // $L[i]$ is partitioned by a
// grid, and buckets summarizing $L[i]$ are added to *C-Buckets*;
$SSE_{tot} = SSE_{tot} - SSE(L[i])$; // The total SSE of non-summarized layers is
// updated
END FOR;
$Distribute(L[0], C\text{-}Buckets)$;
*O-Buckets*=$GridPartition(L[0], (B - size(C\text{-}Buckets)))$;
*Step III*
FOR EACH $b$ IN *O-Buckets* DO
$NOH = NOHoles(b)$;
*C-Buckets*= *C-Buckets* - *NOH*;
$H = H \oplus b \oplus NOH$; //$H$ is augmented with $b$ and its non-overlapping holes (according
// to the adopted physical representation model);
END FOR;
$H = H \oplus$ *C-Buckets*; //$H$ is augmented with c-buckets which have not been selected as
// holes;
RETURN $H$;

Therein:
1) *size* takes as argument a set of buckets and returns their storage space consumption.
2) *GridPartition* takes as argument the MBR of a layer and a storage space bound; it creates a grid on the MBR of layer such that the number of cells can be stored within the specified storage space bound; then it returns all the non-null buckets defined by the grid.
3) *Distribute* takes as argument the layer of outliers $L[0]$ and the set of c-buckets. For each outlier $o$ lying onto the range of some c-bucket, this function removes $o$ from $L[0]$ and adds $o$ to a c-bucket $b$ whose range contains $o$ (that is, the value of $o$ is added to the aggregate $sum(b)$). If $o$ lies onto the range of more than one c-bucket, then one of these c-buckets $b$ is randomly chosen and $o$ is added to it.
4) *NOHoles* takes as argument an o-bucket $b$ and returns a maximal subset of non-overlapping c-buckets whose range is completely contained inside that of $b$.

**Remark.** Notice that the idea of possibly representing a c-bucket as a hole of an o-bucket cannot be extended to the case of pairs of c-buckets $b'$, $b''$ such that $b'$ is completely contained into $b''$. Figure 5.5 shows an example of two clusters $C_1$, $C_2$ whose MBRs overlap. Observe that after partitioning $C_1$ into $b_1$, $b_2$ and $C_2$ into $b_3$, $b_4$, the range of the bucket $b_4$ is completely contained into that of $b_2$, but $b_4$ cannot be considered as a hole of $b_2$, as there are points of both $C_1$ and $C_2$ laid into the range of $b_4$.


**Fig. 5.5.** Two clusters whose MBRs overlap

## 5.3 Experimental Results

In this section we present some experimental results comparing the accuracy of estimating sum range query answers by means of CHIST with state-of-the-art techniques. In the experiments we will concentrate on sum range queries over joint frequency distributions, for the estimation of query selectivities. The accuracy of a histogram $\widetilde{\mathcal{F}}$ built on the joint frequency distribution $\mathcal{F}$ of an input relation $R$ is measured by evaluating the average relative error of the estimates obtained by accessing the histogram. Given a query $q$ on $R$, we denote as $Q$ the sum range query on $\mathcal{F}$ computing the selectivity of $q$;

we denote as $\widetilde{Q}$ the estimate of $Q$ evaluated on $\widetilde{\mathcal{F}}$. The *relative error* of the estimate $\widetilde{Q}$ is defined as: $e^{rel} = \frac{|Q-\widetilde{Q}|}{\max\{1,Q\}}$. We performed several experiments both on synthetic and real-life data.

**Synthetic data.** Our synthetic data are similar to those of [31]. They are generated by creating an empty $d$-dimensional array $D$ of size $n^d$, and then by populating $r$ regions of $D$ by distributing into each of them a portion of the total sum value $T$. The size of the dimensions of each region is randomly chosen between $l_{min}$ and $l_{max}$, and the regions are uniformly distributed in the multi-dimensional array. The total sum $T$ is partitioned across the $r$ regions according to a Zipf distribution with parameter $z$. To populate each region, we first generate a Zipf distribution whose parameter is randomly chosen between $z_{min}$ and $z_{max}$. Next, we associate these values to the cells in such a way that the closer a cell to the centre of the region, the larger its value. Outside the dense regions, some isolated non-zero values are randomly assigned to the array cells. As explained in [31], data-sets generated by using this strategy well represent many classes of real-life distributions.

**Real life data.** We considered a real-life data set which will be referred to as *Forest Cover*. It was obtained from the U.S. Forest Service and is available at the UCI KDD archive site. It consists of 581012 tuples having 54 attributes. Among these, 10 attributes are numerical. As in [44], we considered the tuples projected on these numerical attributes, thus obtaining a 10-dimensional data distribution which will be denoted as $FC_{10}$. We projected $FC_{10}$ on five attributes, thus obtaining a 5-dimensional data distribution which will be denoted as $FC_5$.

In our experiments we compared CHIST with MHIST, MinSkew and GENHIST (see Sect. 2.2 for related work). The comparison has been accomplished by considering histograms (with the same number of buckets) obtained by the four techniques. We have investigated how the accuracy depends on the number of buckets and on the exact selectivity of the queries. Diagrams (a, b) in Fig. 5.6 refer to 4-dimensional synthetic data ($d$=4; $n$=1000; $T$=100000; $r$=100; $z_{min}$=0.5; $z_{max}$=2.5; $l_{min}$= 30; $l_{max}$= 200), whereas diagrams (c, d) refer to 8-dimensional synthetic data ($d$ = 8; $n$ = 1000; $T$ = 200000; $r$ = 200; $z_{min}$= 0.5; $z_{max}$= 2.5; $l_{min}$= 30; $l_{max}$= 200). In both cases we set *MinPts*= 4 and *Eps*= 4. Diagrams (a, b) in Fig. 5.7 were obtained on $FC_5$, whereas diagrams (c, d) refer to $FC_{10}$ (in these cases we set *MinPts*= 4 and *Eps*= 25).

The query workload was constructed by first randomly generating 10000 query centers in the data domain; then, for each of these centers, queries with increasing selectivity were generated by progressively enlarging the query volume, till a selectivity threshold is reached (this threshold is 6.4% for synthetic data, and 5% for real-life data). Finally, the results on the accuracy of the answers were grouped by the query selectivity.

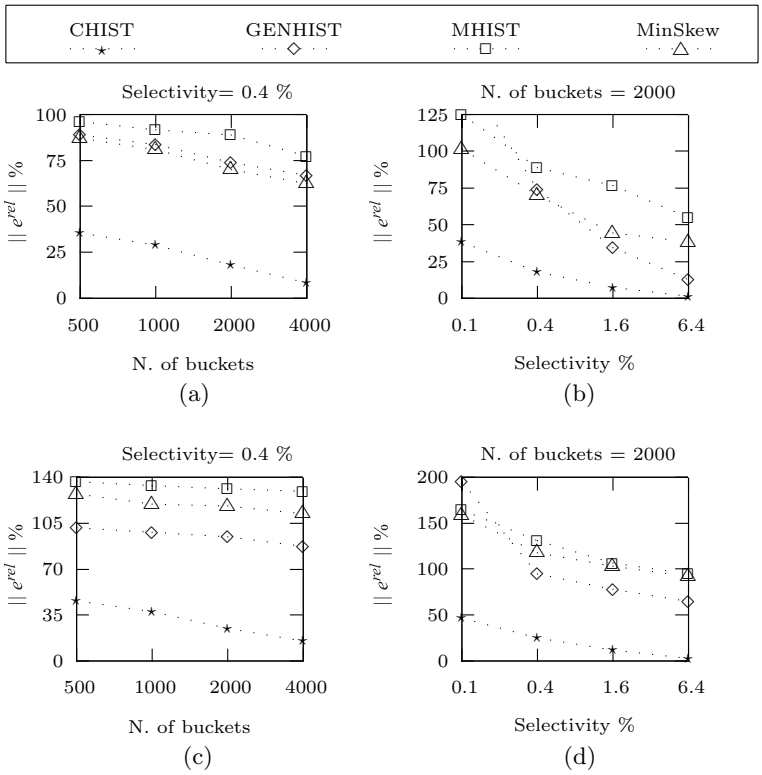


**Fig. 5.6.** Accuracy of techniques on 4D synthetic data (a, b) and 8D synthetic data (c, d)

From the analysis of the diagrams in Figs. 5.6 and 5.7 it turns out that, for all the techniques, the accuracy of estimates improves as the number of buckets increases. Likewise, error rates decrease as selectivity increases. This is mainly due to the fact that queries having higher selectivity are likely to have larger volumes: the larger the volume, the more the buckets of the histogram which are completely contained in the query range (such buckets give an exact contribution to the query evaluation).

Diagrams in Figs. 5.6 and 5.7 show that CHIST outperforms all the other techniques on both synthetic and real-life data.

The table in Fig. 5.8 reports construction times of the examined techniques on the 8-dimensional synthetic data set considered above. The columns refer to different number of buckets (from 500 to 4000). Execution times for CHIST are expressed as a sum of two contributions: the first one is the time needed to execute DBSCAN, the second one refers to Step II and Step III of our algorithm. MHIST and MinSkew construction costs are less than GENHIST and CHIST, but their accuracy is very poor. On the one hand, the fact that CHIST is slower than the other techniques is not a crucial drawback: the sum-

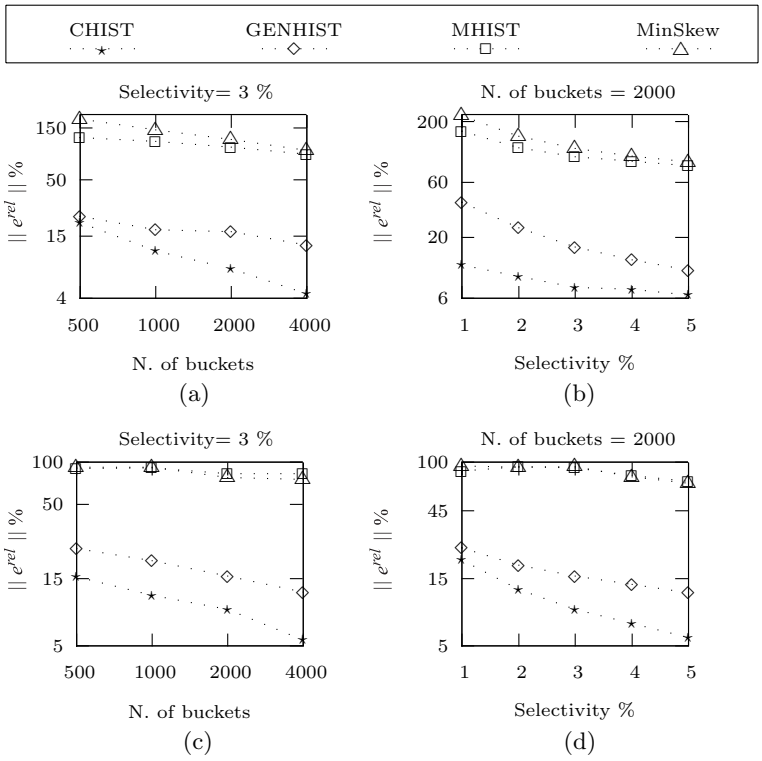


**Fig. 5.7.** Accuracy of techniques on $FC_5$ (a, b) and $FC_{10}$ (c, d)

marization of data is an off-line task which is usually performed on historical data, so that in practice it is executed only once. On the other hand, it is worth noting that the execution time of CHIST is dominated by the clustering step. Thus, our technique is likely to benefit from the adoption of more efficient clustering algorithms. In particular, future work on CHIST will be devoted to investigate how other clustering techniques can be embedded into the scheme of our algorithm, and how their adoption affects both the accuracy and the construction time of histograms.

**Number of buckets**

| | 500 | 1000 | 2000 | 4000 |
|---|---|---|---|---|
| CHIST | 361 + 6.1 | 361 + 11 | 361 + 26 | 361 + 54 |
| GENHIST | 18 | 30 | 71 | 149 |
| MHIST | 1.5 | 1.6 | 2.0 | 2.2 |
| MinSkew | 1.8 | 2.3 | 2.5 | 3.0 |

**Fig. 5.8.** Histogram construction times (seconds)

# 6

# Summarization of Sensor Data Streams

*In this chapter we address the problem of summarizing streaming data in order to provide fast approximate answers to aggregate queries over time. In particular, we focus on the processing of streams of readings generated by sensor networks to monitor real life phenomena. We propose a hierarchical summarization of the sensor data stream which is incrementally maintained as new readings are received, by progressively compressing older stored data.*

## 6.1 Introduction

Sensors are special devices used to monitor real life phenomena, such as live weather conditions, network traffic, etc. They are usually organized into networks where their readings are transmitted using low level protocols [60]. Sensor networks represent a non-traditional source of information, as readings generated by sensors flow continuously, leading to an infinite stream of data. The readings produced by the sensors are caught by a *Sensor Data Stream Management System* (SDSMS), which combines them into a unique data stream, and supports data analysis. The stream processor is provided with a bounded amount of storage space, typically very small relative to the stream size (which is possibly unbounded). Traditional DBMSs, which are based on an exact and detailed representation of information, are not suitable in this context, as all the information carried by a data stream cannot be stored within a bounded storage space [8, 22, 82, 48, 7]. Thus, processed data items must be either discarded or only partially archived; in any case it's not feasible to compute exact answers to most common queries on the data stream.

A possible solution to this issue consists in summarizing received data into a compact structure which fits the available storage space (by possibly losing less relevant information) and posing queries on summary data. This approach aims at allowing approximate answers on the data stream, by storing as much information carried by the stream as possible.

Obviously, this strategy shares many similarities with the summarization of static data sets, described in the previous chapters, but there are some differences.
First of all, summary structures on streaming data have to be constructed and maintained dynamically, as data arrive, while static data are usually historical and very unfrequent to be refreshed. Thus, the updating efficiency is crucial for data stream summaries, while its not so addressed in static contexts, where summarization is mainly an off-line task. Moreover, the efficiency requirement in answering queries on sensor data streams is quite strict. In fact, the amount of data produced by sensors is very large, grows continuously, and usually measures the conditions of a monitored world: queries need to be evaluated very quickly, in order to allow the stream processor to react timely to possible critical events.
Moreover, in order to make the information produced by sensors useful, it should be possible to retrieve an up-to-date "snapshot" of the monitored world continuously, as time passes and new readings are collected. For instance, a climate disaster prevention system would benefit from the availability of continuous information on atmospheric conditions in the last hour. Similarly, a network congestion detection system would be able to prevent network failures by exploiting the knowledge of network traffic during the last minutes. If the answer to these queries, called *continuous queries*, is not fast enough, we could observe an increasing delay between the query answer and the arrival of new data, and thus a not timely reaction to the world.

In this chapter we propose a technique for providing fast approximate answers to aggregate queries on sensor data streams. Our proposal adopts a hierarchical summarization of the data stream embedded into a flexible indexing structure, which enables both efficient access and incremental maintenance of the summary structure. The summarized representation of data is updated continuously, as new sensor readings arrive. When the available storage space is not enough to store new data, some space is released by compressing the "oldest" stored data progressively, so that recent information (which is usually the most relevant to retrieve) is represented with more detail than old one.

As we describe in Sect. 6.2, we model the overall stream as a two-dimensional data set where the first dimension corresponds to the set of sources, and the other one (potentially infinite) corresponds to time. We design a summarization technique for this two-dimensional data set, by taking into account the peculiar nature of time dimension.
In particular, the sensor data stream is divided into "time windows" of the same size; the readings produced by sensors within each time window are represented by a quad-tree structure, called *quad-tree window*, which is defined in Sect. 6.3. In Sect. 6.3.4 we represent the overall sensor data stream by means of a list of quad-tree windows, by showing how quad-tree windows can be created and populated dynamically as data arrive.

In order to allow efficient access to the data stream synopsis, quad-tree windows are divided into clusters and indexed: in Sect. 6.4 we define the

overall structure summarizing the data stream, called *Multi-Resolution Data Stream Summary* (MRDS), as a list of indexed clusters of quad-tree windows. In particular, in Sect. 6.4.1 and 6.4.2 we introduce the *binary tree index*, which permits the efficient location of quad-tree windows within a given cluster; then in Sect. 6.4.3 we describe the construction of the overall Multi-Resolution Data Stream Summary, by showing how binary tree indices are dynamically constructed on the underlying quad-tree windows, and linked together, as new data arrive. Moreover, in Sect. 6.5 we describe the progressive compression of a Multi-Resolution Data Stream Summary, as new data is received, in order to release the storage space needed to represent new readings. Finally, in Sect. 6.6, we show how range queries and continuous queries are answered on a sensor data stream by accessing only the MRDS.

## 6.2 Problem Statement

Consider an ordered set of $n$ sources (i.e. sensors) denoted by $\{s_1, \ldots, s_n\}$ producing $n$ independent streams of data, representing sensor readings. Each data stream can be viewed as a sequence of triplets $\langle id_s, v, ts \rangle$, where: 1) $id_s \in \{1, .., n\}$ is the source identifier; 2) $v$ is a non negative integer value representing the measure produced by the source identified by $id_s$; 3) $ts$ is a *timestamp*, i.e. a value that indicates the time when the reading $v$ was produced by the source $id_s$.

The data streams produced by the sources are caught by a *Sensor Data Stream Management System* (SDSMS), which combines the sensor readings into a unique data stream, and supports data analysis.

An important issue in managing sensor data streams is aggregating the values produced by a subset of sources within a time interval. More formally, this means answering a *range query* on the overall stream of data generated by $s_1, \ldots, s_n$. A range query is a pair $Q = \langle s_i..s_j, [t_{start}..t_{end}] \rangle$ whose answer is the evaluation of an aggregate operator (such as *sum*, *count*, *avg*, etc.) on the values produced by the sources $s_i, s_{i+1}, \ldots, s_j$ within the time interval $[t_{start}..t_{end}]$. In particular, the work presented in this chapter focuses on sum range queries.

We point out that considering the set of sources as an ordered set implies the assumption that the sensors in the network can be organized according to a linear ordering. Whenever any implicit linear order among sources cannot be found (for instance, consider the case that sources are identified by a geographical location), a mapping should be defined between the set of sources and a one-dimensional ordering. This mapping should be closeness-preserving, that is sensors which are "close" in the network should be close in the linear ordering. Obviously, it is not always possible to define a liner ordering such that no information about the "relative" location of every source w.r.t. each other is lost. It can happen that two sources which can be considered as contiguous in the network are not located in contiguous positions according

to the linear ordering criterion. In this case, a range query involving a set of contiguous sensors in the network is possibly translated into more than one range query on the linear paradigm used to represent the whole set of sources.

The sensor data stream can be represented by means of a two-dimensional array, where the first dimension corresponds to the set of sources, and the other one corresponds to time. In particular, the time is divided into intervals $\Delta t_j$ of the same size. Each element $\langle s_i, \Delta t_j \rangle$ of the array is the sum of all the values generated by the source $s_i$ whose timestamp is within the time interval $\Delta t_j$. Obviously, the use of a time granularity generates a loss of information, as readings of a sensor belonging to the same time interval are aggregated. Indeed, if a time granularity which is appropriate for the particular context monitored by sensors is chosen, the loss of information will be negligible.

Using this representation, an estimate of the answer to a sum range query over $\langle s_i..s_j, [t_{start}..t_{end}] \rangle$ can be obtained by summing two contributions. The first one is given by the sum of those elements which are completely contained inside the range of the query (i.e. the elements $\langle s_k, \Delta t_l \rangle$ such that $i \leq k \leq j$ and $\Delta t_l$ is completely contained into $[t_{start}..t_{end}]$). The second one is given by those elements which partially overlap the range of the query (i.e. the elements $\langle s_k, \Delta t_l \rangle$ such that $i \leq k \leq j$ and $t_{start} \in \Delta t_l$ or $t_{end} \in \Delta t_l$). The first of these two contributions does not introduce any approximation, whereas the second one is generally approximate, as the use of the time granularity makes it unfeasible to retrieve the exact distribution of values generated by each sensor within the same interval $\Delta t_l$. The latter contribution can be evaluated by performing linear interpolation, i.e. assuming that the data distribution inside each interval $\Delta t_i$ is uniform (*Continuous Values Assumption - CVA*). For instance, the contribution of the element $\langle s_2, \Delta t_3 \rangle$ to the sum query represented in Fig. 6.1 is given by $\frac{6-5}{2} \cdot 4 = 2$.

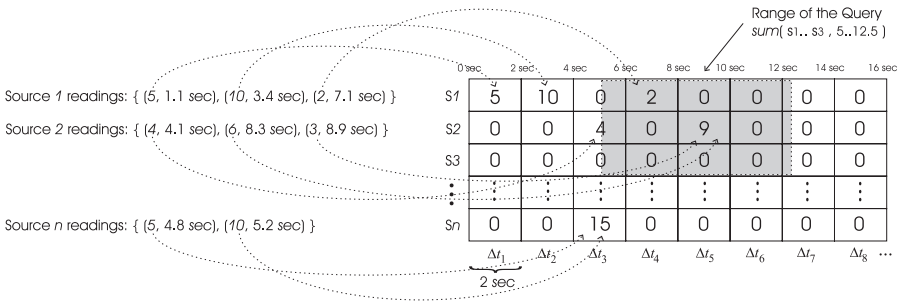


**Fig. 6.1.** Two-dimensional representation of sensor data streams

As the stream of readings produced by every source is potentially "infinite", detailed information on the stream (i.e. the exact sequence of values generated by every sensor) cannot be stored, so that exact answers to every possible range query cannot be provided.

However, exact answers to aggregate queries are often not necessary, as approximate answers usually suffice to get useful reports on the content of data streams, and to provide a meaningful description of the world monitored by sensors.

A solution for providing approximate answers to aggregate queries is to store a summarized representation of the overall data stream, and then to run queries on the summary data. The use of a time granularity introduces a form of summarization, but it does not suffice to represent the whole stream of data, as the stream length is possibly infinite. An effective structure for storing the information carried by the data stream should have the following characteristics: i) it should be efficient to update, in order to catch the continuous stream of data coming from the sources; ii) it should provide an up-to-date representation of the sensor readings, where recent information is possibly represented more accurately than old one; iii) it should permit us to answer range queries efficiently.

**Our proposal.** In this chapter we propose a technique for providing (fast) approximate answers to aggregate queries on sensor data streams, focusing our attention on *sum* range queries. Our proposal consists in a compact representation of the sensor data stream where the information is summarized in a hierarchical fashion. In particular, a flexible indexing structure is embedded into the summary data representation, so that information can be both accessed and updated efficiently.

In more detail, our summarization technique is based on the following scheme:

- the sensor data stream is divided into "time windows" of the same size: each window consists of a finite number of contiguous unitary time intervals $\Delta t_i$ (the size of each $\Delta t_i$ corresponds to the granularity);
- time windows are indexed, so that windows involved in a range query can be accessed efficiently;
- as new data arrive, if the available storage space is not enough for their representation, "old" windows are compressed (or possibly removed) to release the storage space needed to represent new readings, and the index is updated to take into account the new data.

The technique used for compressing time windows is *lossy*, so that "recent" data are generally represented more accurately than "old" ones. In Fig. 6.2, the partitioning scheme of a stream into time windows is represented, as well as the overlying index referring to all the time windows.

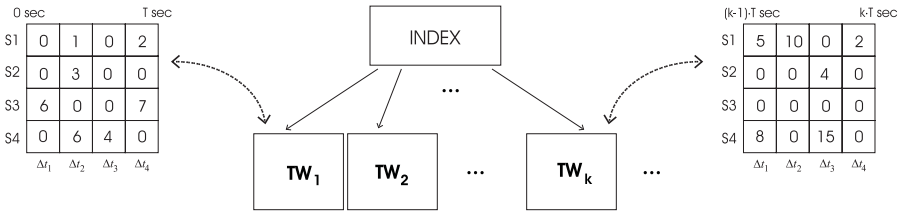


**Fig. 6.2.** A sequence of indexed time windows

## 6.3 Representing Time Windows

### 6.3.1 Preliminary Definitions

To deal with multi-dimensional arrays we will refer to the basic notations introduced in Sect. 1.4. In particular, we are interested in two-dimensional data distributions. Given a block $r = [l_1..u_1, l_2..u_2]$ of a two-dimensional array, we denote by $r_i$ the $i-$th quadrant of $r$, i.e. $r_1 = [l_1..m_1, l_2..m_2]$, $r_2 = [m_1 + 1..u_1, l_2..m_2]$, $r_3 = [l_1..m_1, m_2 + 1..u_2]$, and $r_4 = [m_1 + 1..u_1, m_2 + 1..u_2]$. where $m_1 = \lfloor (l_1 + u_1)/2 \rfloor$ and $m_2 = \lfloor (l_2 + u_2)/2 \rfloor$. Given a a time interval $t = [t_{start}..t_{end}]$ we denote by $size(t)$ the size of the time interval $t$, i.e. $size(t) = t_{end} - t_{start}$. Furthermore, we denote by $t_{i/2}$ the $i$-th half of $t$. That is $t_{1/2} = [t_{start}..(t_{start} + t_{end})/2]$ and $t_{2/2} = [(t_{start} + t_{end})/2..t_{end}]$. Given a tree $T$, we denote by $Root(T)$ the root node of $T$ and, if $p$ is a non leaf node, we denote the $i-$th child node of $p$ by $Child(p, i)$. Given a triplet $x = \langle id_s, v, ts \rangle$, representing a value generated by a source, $id_s$ is denoted by $id_s(x)$, $v$ by $value(x)$ and $ts$ by $ts(x)$.

### 6.3.2 The Quad-Tree Window

In order to represent data occurring in a time window, we do not store directly the corresponding two-dimensional array, indeed we choose a hierarchical data structure, called *quad-tree window*, which offers some advantages: it makes answering (portions of) range queries internal to the time window more efficient to perform (w.r.t. a "flat" array representation), and it stores data in a straight compressible format. That is, data is organized according to a scheme that can be directly exploited to perform compression.

This hierarchical data organization consists in storing multiple aggregations performed over the time window array according to a quad-tree partition. The quad-tree partition of an array has been introduced in Chap. 3; here we adopt it to represent all the elements contained in a time window. This means that we store the sum of the values contained in the whole array, as well as the sum of the values contained in each quarter of the array, in each eighth of the array and so on, until the single elements of the array are stored. Figure 6.3 shows an example of "complete" quad-tree partition, where each

node of the quad-tree is associated to the sum of the values contained in the corresponding portion of the array.

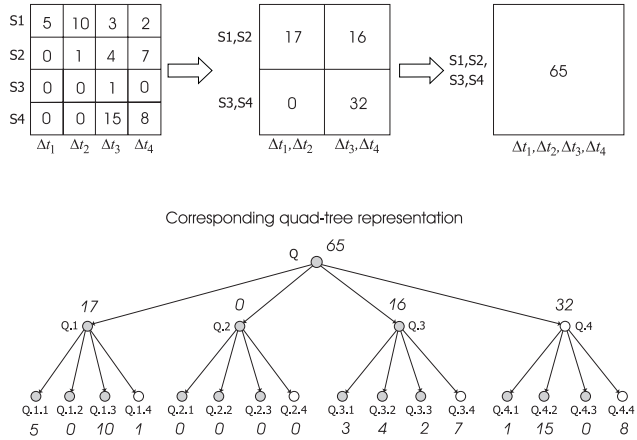


**Fig. 6.3.** A Time Window and the corresponding quad-tree partition

The quad-tree structure is very effective for answering (sum) range queries inside a time window efficiently, as we can generally use the pre-aggregated sum values in the quad-tree nodes for evaluating the answer (see Sect. 6.6.1 for more details). Moreover, the space needed for storing the quad-tree representation of a time window is about the same as the space needed for a flat representation, as we will explain later. Furthermore, the quad-tree structure is particularly prone to progressive compressions. In fact, the information represented in each node is summarized in its ancestor nodes. For instance, the node $Q$ of the quad-tree in Fig. 6.3 contains the sum of its children $Q.1$, $Q.2$, $Q.3$, $Q.4$; analogously, $Q.1$ is associated to the sum of $Q1.1$, $Q1.2$, $Q1.3$, $Q1.4$, and so on. Therefore, if we prune some nodes from the quad-tree, we do not loose every information about the corresponding portions of the time window array, but we represent them with less accuracy. For instance, if we removed the nodes $Q1.1, Q1.2, Q1.3, Q1.4$, then the detailed values of the readings produced by the sensors $S_1$ and $S_2$ during the time intervals $\Delta t_1$ and $\Delta t_2$ would be lost, but it would be kept summarized in the node $Q.1$. The compression paradigm that we use for quad-tree windows will be better explained in Sect. 6.5.

We will next describe the quad-tree based data representation of a time window formally. Denoting by $u$ the time granularity (i.e. the width of each interval $\Delta t_j$), let $T = n \cdot u$ be the time window width (where $n$ is the number of sources). We refer to a *Time Window* starting at time $t$ as a two-dimensional array $W$ of size $n \times n$ such that $W[i, j]$ represents the sum of the values generated by a source $s_i$ within the $j-$th unitary time interval of $W$. That is $W[i, j] = \sum_{x:id_s(x)=i \wedge ts(x) \in \Delta t_j} value(x)$, where $\Delta t_j$ is the time interval

$[t+(j-1)\cdot u..t+j\cdot u]$. The whole data stream consists of an infinite sequence $W_1, W_2, \ldots$ of time windows such that the $i-$th one starts at $t_i = (i-1)\cdot T$ and ends at $t_{i+1} = i \cdot T$.

In the following, for the sake of presentation, we assume that the number of sources is a power of 2 (i.e. $n = 2^k$, where $k > 1$).

A *Quad-Tree Window* on the time window $W$, called $QTW(W)$, is a full $4-$ary tree whose nodes are pairs $\langle r, sum(r)\rangle$ (where $r$ is a block of $W$) such that:

1. $Root(QTW(W)) = \langle [1..n, 1..n], sum([1..n, 1..n])\rangle$;
2. each non leaf node $q = \langle r, sum(r)\rangle$ of $QTW(W)$ has four children representing the four quadrants of $r$; That is, $Child(q, i) = \langle r_i, sum(r_i)\rangle$ for $i = 1, \ldots, 4$.
3. the depth of $QTW(W)$ is $log_2 n + 1$.

Property 3 implies that each leaf node of $QTW(W)$ corresponds to a single element of the time window array $W$. Given a node $q = \langle r, sum(r)\rangle$ of $QTW(W)$, $r$ is referred to as $q.range$ and $sum(r)$ as $q.sum$.

### 6.3.3 Compact Physical Representation of Quad-Tree Windows

The space needed for storing all the nodes of a quad-tree window $QTW(W)$ is larger than the one needed for a flat representation of $W$. In fact, it can be easily shown that the number of nodes of $QTW(W)$ is $\frac{4\cdot n^2-1}{3}$, whereas the number of elements in $W$ is $n^2$. Indeed, $QTW(W)$ can be represented compactly, as it is not necessary to store the sum values of all the nodes of the quad-tree (in a similar way as explained in Chap. 3). That is, if we have the sum values associated to a node and to three of its children, we can easily compute the sum value of its fourth child. This value can be obtained by subtracting the sum of the three children from the sum of the parent node. We say that the fourth child is a *derivable* node.

For instance, the node $Q4$ of the quad-tree window in Fig. 6.3 is derivable, as its sum is given by $Q.sum - (Q1.sum + Q2.sum + Q3.sum)$. Derivable nodes of the quad-tree window in Fig. 6.3 are all colored in white. Using this storing strategy, the number of nodes that are not derivable (i.e. nodes whose sum must be necessarily stored) is $n^2$, that is the same as the size of $W$.

This compact representation of $QTW(W)$ can be further refined to manage occurrences of null values efficiently. If a node of the quad-tree is null, all of its descendants will be null. Therefore, we can avoid to store the sum associated to every descendant of a null node, as its value is implied. For instance, the sums of the nodes $Q2.1$, $Q2.2$, $Q2.3$, $Q2.4$ need not be stored: their value (i.e. the value 0) can be retrieved by accessing their parent.

We point out that the physically represented quad-tree describing a time window is generally not full. Indeed, null nodes having a non null parent are treated as leaves, as none of their children is physically stored. We will next

focus our attention on the physical compact representation of a quad-tree window.

In a similar way as done for QTSs in Chap. 3, a quad-tree window can be stored by representing separately the tree structure and the content of the nodes. The tree structure can be represented by a string of bits: two bits per node of the tree indicate whether the node is a leaf or not, and whether it is associated with a null block or not. Obviously, in this physical representation, an internal node cannot be null.

In more detail, the encoding pairs are: (1) $\langle 0,0 \rangle$ meaning non null leaf node, (2) $\langle 0,1 \rangle$ meaning null leaf node, (3) $\langle 1,1 \rangle$ meaning non leaf node. It remains one available configuration (i.e., $\langle 1,0 \rangle$) which will be used when compressing quad-tree windows, as it will be shown in Sect. 6.5. The mapping between the stored pair of bits and the corresponding nodes of the quad-tree is obtained storing the string of bits according to a predetermined linear ordering of the quad-tree nodes. In Fig. 6.4, the physically represented QTW corresponding to the QTW of Fig. 6.3 is shown. The children of $Q2$ are not explicitly stored, as they are inferable. The string of bits describing the structure of the QTW corresponds to a breadth-first visit of the quad-tree. Note that, since the blocks in the quad-tree nodes are obtained by consec-

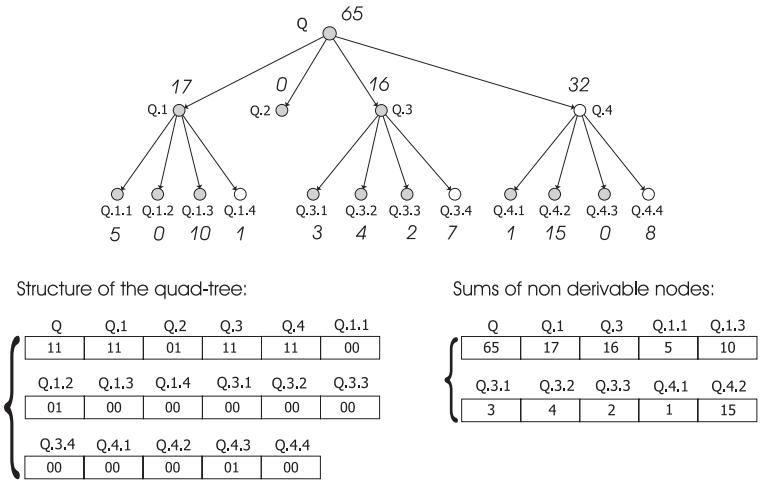


**Fig. 6.4.** A quad-tree window and its physical representation

utive splits into four equally sized quadrants, the above string of bits stores enough information to reconstruct the boundaries of each of these blocks. This means that the boundaries of the blocks corresponding to the nodes do not need to be represented explicitly, as they can be retrieved by visiting the quad-tree structure. It follows that the content of the quad-tree can be represented by an array containing just the sums occurring in the nodes. Some

storage space can be further saved observing that: 1) we can avoid to store the sums of the null blocks, since the structure bits give enough information to identify them; 2) we can avoid to store the sums contained in the *derivable* nodes of the quad-tree window, i.e. the nodes $p$ such that $p = Child(q, 4)$, for some other node $q$. As explained above, the sum of $p$ can be derived as $p.sum = q.sum - \sum_{i=1..3} Child(q, i).sum$.

Altogether, the quad-tree window content can be represented by an array storing the set $\{p.sum | p$ is a non-derivable quad-tree node and $p.sum > 0\}$. The above sums are stored according to the same ordering criterion used for storing the structure, in order to associate the sum values to the nodes consistently. For instance, the string of sums reported on the right-hand side of Fig. 6.4 corresponds to the breadth-first visit which has been performed to generate the string of bits on the center of the same figure. The sums of the nodes $Q.2$, $Q1.2$ and $Q4.3$ are not represented in the string of sums as they are null, whereas the sums of the nodes $Q.4$, $Q1.4$, $Q3.4$ and $Q4.4$ are not stored, as these nodes are derivable.

It can be shown that, if we use 32 bits for representing a sum, the largest storage space needed for a quad-tree window is $S^{max}_{QTW} = (32 + 8/3)n^2 - 2/3$ bits (assuming that the window does not contain any null value).

### 6.3.4 Populating Quad-Tree Windows

In this section we describe how a quad-tree window is populated as new data arrive. Let $W_k$ be the time window associated to a given time interval $[(k - 1) \cdot T..k \cdot T]$, and $QTW(W_k)$ the corresponding quad-tree window. Let $x = \langle id_s, v, ts \rangle$ be a new sensor reading such that $ts$ is in $[(k - 1) \cdot T..k \cdot T]$. We next describe how $QTW(W_k)$ is updated on the fly, to represent the change of the content of $W_k$.

Let $QTW(W_k)_{old}$ be the quad-tree window representing the content of $W_k$ before the arrival of $x$. If $x$ is the first received reading whose timestamp belongs to the time interval of $W_k$, $QTW(W_k)_{old}$ consists of a unique null node (the root). Algorithm 1 shown in Appendix B.1 takes as arguments $x$ and $QTW(W_k)_{old}$, and returns the up-to-date quad-tree window $Q_{new}$ on $W_k$. Algorithm 1 works as follows. First, the old quad-tree window $QTW(W_k)_{old}$ is assigned to $Q_{new}$. Then, the algorithm determines the coordinates $\langle id_s, j \rangle$ of the element of $W_k$ which must be updated according to the arrival of $x$, and visits $Q_{new}$ starting from its root. At each step of the visit, the algorithm processes a node of $Q_{new}$ corresponding to a block of $W_k$ which contains $\langle id_s, j \rangle$. The sum associated to the node is updated by adding $value(x)$ to it (see Fig. 6.5). If the visited node was null (before the updating), it is split into four new null children. After updating the current node (and possibly splitting it), the visit goes on processing the child of the current node which contains $\langle id_s, j \rangle$. The algorithm ends after updating the node of $Q_{new}$ corresponding to the single element $\langle id_s, j \rangle$.

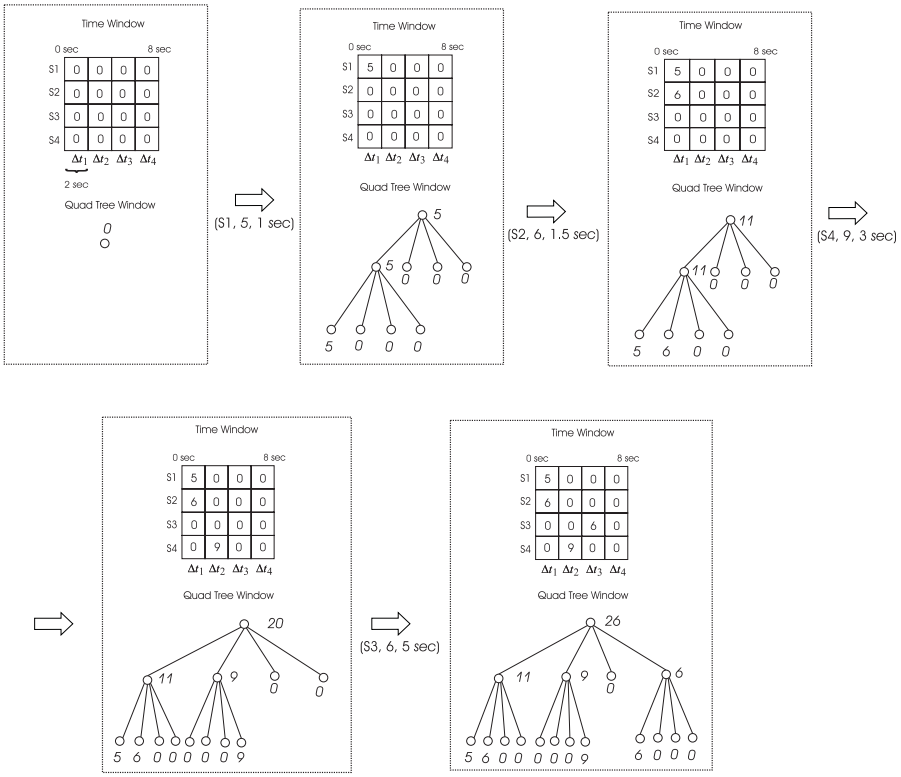


**Fig. 6.5.** Populating a quad-tree window

## 6.4 The Multi-Resolution Data Stream Summary

A quad-tree window represents the readings generated within a time interval of size $T$. The whole sensor data stream can be represented by a sequence of quad-tree windows $QTW(W_1), QTW(W_2), \ldots$. When a new sensor reading $x$ arrives, it is inserted in the corresponding quad-tree window $QTW(W_k)$, where $ts(x) \in [(k-1) \cdot T..k \cdot T]$. A quad-tree window $QTW(W_k)$ is physically created when the first reading belonging to $[(k-1) \cdot T..k \cdot T]$ arrives.

In this section we define a structure that both indexes the quad-tree windows and summarizes the values carried by the stream. This structure is called *Multi-Resolution Data Stream Summary* and pursues two aims: 1) making range queries involving more than one time window efficient to evaluate; 2) making the stored data easy to compress.

We propose the following scheme for indexing quad-tree windows:

1. time windows are clustered into groups $C_1, C_2, \ldots$; each cluster consists of $K$ contiguous time windows, thus describing a time interval of size $K \cdot T$;

2. quad-tree windows inside each cluster $C_l$ are indexed by means of a binary tree denoted by $BTI(C_l)$;
3. the whole index consists of a list linking $BTI(C_1), BTI(C_2), \ldots$.

We next focus our attention on describing the structure of a single index $BTI(C_l)$. Then, we show how the whole index overlying the quad-tree windows is built.

### 6.4.1 Indexing a Cluster of Quad-Tree Windows

Consider the $l$-th cluster $C_l$ of the sequence representing the whole sensor data stream. $C_l$ corresponds to the time interval $[(l-1) \cdot K \cdot T..l \cdot K \cdot T]$. The time interval corresponding to $C_l$ will be denoted by $\Delta T(C_l)$. We fix the value of $K$ to a power of 2.

A *Binary Tree Index* on $C_l$, is denoted by $BTI(C_l)$ and is a full binary tree whose nodes are pairs $\langle t, s \rangle$, with $t$ a time interval and $s$ a sum, such that:

1. $Root(BTI(C_l)) = \langle \Delta T(C_l), sum(\Delta T(C_l)) \rangle$ where $sum(\Delta T(C_l))$ is the sum of the values generated within $\Delta T(C_l)$ by all the sources, i.e. $sum(\Delta T(C_l)) = \sum_{(l-1) \cdot K < i \leq l \cdot K} sum(W_i)$
2. each non leaf node $q = \langle t, s \rangle$ of $BTI(C_l)$, with $t = [j_1 T..j_2 T]$, has two child nodes corresponding to the two halves of $t$, that is $Child(q, i) = \langle t_{i/2}, s_{i/2} \rangle$, $i = 1, 2$, where $t_{i/2}$ is the $i-$th half of $t$, and $s_{i/2}$ is the sum of all the readings generated within $t_{i/2}$ by all the sources.
3. the depth of $BTI(C_l)$ is $log_2 K$, that is each leaf node of $BTI(C_l)$ corresponds to a time interval of size $2T$.
4. each leaf node $q = \langle t, s \rangle$ of $BTI(C_l)$, with $t = [j_1 T..j_2 T]$ $(j_2 - j_1 = 2)$, refers to the two quad-tree windows in $t$ (i.e. $QTW(W_i)$, $j_1 < i \leq j_2$).

Given a node $q = \langle t, s \rangle$ of $BTI(C_l)$, $t$ and $s$ are referred to as $q.interval$ and $q.sum$, respectively. Moreover, $q.range$ denotes the two-dimensional range $\langle s_1..s_n, t \rangle$.

### 6.4.2 Compact Physical Representation of Binary Tree Indices

In Sect. 6.3.3 we have described how quad-tree windows can be stored efficiently, saving the space needed for representing both null and derivable nodes. Analogously, binary tree indices can be stored in a compact fashion: in a binary tree index a node is derivable if it is the second child of another node. The sum of a derivable node can be computed by subtracting the sum of the sibling node from the sum of the parent node. The sums of both derivable and null nodes are not explicitly stored, as they can be efficiently retrieved by accessing the stored information concerning the structure of the tree and the content of the other nodes. The resulting physical representation is the same as the one described in Sect. 6.3.3: a string of bits is used to encode the tree

structure, and an array of sums is used to represent the content of the nodes. The encoding pairs occurring in the string of bits are the same as described in Sect. 6.3.3. The largest space consumption of a binary tree index (embedding its referred QTWs) can be shown to be $S^{max}_{BTI} = (32+8/3)\cdot K\cdot n^2+(52/3)\cdot K-2$ bits.

### 6.4.3 Constructing and Linking Binary Tree Indices

In the same way as quad-tree windows, binary tree indices can be constructed dynamically, as new data arrive and new quad-tree windows are created. An algorithm for constructing a binary tree index follows the same strategy as Algorithm 1, and, in particular, uses Algorithm 1 for populating the indexed quad-tree windows. The resulting algorithm is reported in Appendix B.2. It consists of a function which takes as arguments a "new" reading $x$ and the binary tree index $BTI(C_l)$ where $x$ is in $\Delta T(C_l)$, and updates both the index and the underlying quad-tree windows.

The overall index on the sensor data stream is obtained by linking together $BTI(C_1)$, $BTI(C_2), \ldots$, i.e. the binary tree indices corresponding to consecutive clusters. In particular, when a new sensor reading $x$ arrives, it is inserted (according to Algorithm 2) into the binary tree index $BTI(C_l)$ such that $ts(x)$ is in $\Delta T(C_l)$. If this $BTI$ does not exist (i.e. $x$ is the first arrival in this cluster), first of all a new binary tree index $BTI(C_l)$ containing a unique null node (the root) is created. Then the function $Insert(BTI(C_l), x)$ is called, and the updated $BTI$ returned by Algorithm 2 is added to the existing list of consecutive binary tree indices. The list of $BTI$s with the underlying list of quad-tree windows is referred to as *Multi-Resolution Data Stream Summary* - MRDS. As the sensor data stream is infinite, the length of the list of binary tree indices is not bounded, so that a MRDS cannot be physically stored. In the following section we propose a summarization technique which allows us to store the most relevant information carried by the (infinite) sensor data stream by keeping a finite list of (compressed) binary tree indices.

## 6.5 Compression of the Multi-Resolution Data Stream Summary

Due to the bounded storage space which is available to store the information carried by the sensor data stream, the Multi-Resolution Data Stream Summary (which consists of a list of indexed clusters of quad-tree windows) cannot be physically represented, as the stream is potentially infinite.

As new sensor readings arrive, the available storage space decreases till no other reading can be stored. Indeed, we can assume that recent information is more relevant than older one for answering user queries, which usually investigate the recent evolution of the monitored world. Therefore, older information can be reasonably represented with less detail than recent data.

This suggests us the following approach: as new readings arrive, if there is not enough storage space to represent them, the needed storage space is obtained by discarding some detailed information about "old" data.

We next describe our approach in detail. Let $x$ be the new sensor reading to be inserted, and let $BTI(C_1)$, $BTI(C_2)$, ..., $BTI(C_k)$ be the list of binary tree indices representing all the sensor readings preceding $x$. This means that $x$ must be inserted into $BTI(C_k)$. The insertion of $x$ is done by performing the following steps:

1. the storage space $Space(x)$ needed to represent $x$ into $BTI(C_k)$ is computed by evaluating how the insertion of $x$ modifies the structure and the content of $BTI(C_k)$. $Space(x)$ can be easily computed using the same visiting strategy as Algorithm 2;
2. if $Space(x)$ is larger than the left amount $Space_a$ of available storage space, then the storage space $Space(x) - Space_a$ is obtained by compressing (using a lossy technique) the oldest binary tree indices, starting from $BTI(C_1)$ towards $BTI(C_k)$, till enough space is released.
3. $x$ is inserted into $BTI(C_k)$.

We next describe in detail how the needed storage space is released from the list $BTI(C_1)$, $BTI(C_2)$, ..., $BTI(C_k)$. First, the oldest binary tree index is compressed (using a technique that will be described later) trying to release the needed storage space. If the released amount of storage space is not enough, then the oldest binary tree index is removed from the list, and the same compression step is executed on the new list $BTI(C_2)$, $BTI(C_3)$, ..., $BTI(C_k)$. The compression process ends when enough storage space has been released from the list of binary tree indices. This process is implemented in Algorithm 3 shown in Appendix B.3.

The compression strategy adopted by the function $CompressBTI$ in Algorithm 3 exploits the hierarchical structure of the binary tree indices: each internal node of a $BTI$ contains the sum of its child nodes, and the leaf nodes contain the sum of all the reading values contained in the referred quad-tree windows. This means that the information stored in a node of a $BTI$ is replicated with a coarser "resolution" in its ancestor nodes. Therefore, if we delete two sibling nodes from a binary tree index, we do not loose every information carried by these nodes: the sum of their values is kept in their ancestor nodes. Analogously, if we delete a quad-tree window $QTW_k$, we do not loose every information about the values of the readings belonging to the time interval $[(k-1) \cdot T..k \cdot T]$, as their sum is kept in a leaf node of the $BTI$.

The compression strategy of $CompressBTI$ is based on the above reasoning. As it will be described later, it compresses the oldest $BTI$ by either compressing the referred QTWs (using an ad hoc technique for compressing quad-trees) or pruning some of its nodes. This means that the compression process modifies the structure of a $BTI$:

- a compressed $BTI$ is not, in general, a full binary tree, as it is obtained from a full tree (i.e. the original $BTI$) by deleting some of its nodes;

- not every leaf node refers to two QTWs, as a leaf node of the compressed $BTI$ can be obtained in three ways: 1) it corresponds to a leaf node of the original $BTI$; 2) it corresponds to a leaf node of the original $BTI$ whose referred QTWs have been deleted; 3) it corresponds to an internal node of the original $BTI$ whose child nodes have been deleted.

The compact physical representation of a $BTI$ described in Sect. 6.4.2 must be modified in order to represent a compressed $BTI$. In particular, the pairs of bits encoding the tree structure need to be redefined in order to distinguish between the several kinds of leaf nodes of a compressed $BTI$. The pairs of bits which encode the tree structure of a compressed $BTI$ are: (1) $\langle 0,0\rangle$ meaning non null leaf node with no quad-tree windows, (2) $\langle 0,1\rangle$ meaning null leaf node, (3) $\langle 1,0\rangle$ meaning non null leaf node with quad-tree windows, (4) $\langle 1,1\rangle$ meaning non leaf node.

We next describe in detail the compression process of a $BTI$ performed by the function $CompressBTI$ invoked in Algorithm 3. The $BTI$ to be compressed is visited in order to reach the left-most node $N$ (i.e. the oldest node) having one of the following properties:

1. $N$ is a leaf node of the $BTI$ which refers to 2 QTWs;
2. the node $N$ has 2 child leaf nodes, and all the 2 children do not refer to any QTW.

In the first of the two cases, $CompressBTI$ calls an ad hoc procedure for compressing the quad-tree windows referred by $N$. The 2 QTWs are compressed till either the needed storage space is released, or they cannot be further compressed. If both QTWs are no longer compressible, then they are deleted definitively. In the second of the two cases, the children of $N$ are deleted. The information contained in these nodes is kept summarized in $N$.

In Fig. 6.6, several steps of the compression process on a binary tree index of depth 4 (i.e. a $BTI$ indexing 16 QTWs) is shown. The QTWs underlying the $BTI$ are represented by squares. In particular, uncompressed QTWs are white, partially compressed are grey, whereas QTWs which cannot be further compressed are crossed. We next describe the compression process reported in Fig. 6.6. At step 1, the oldest QTW is partially compressed. At step 2, the needed storage space is released by continuing the compression of $QTW_1$ till it cannot be further compressed. As the released storage space is not enough, $QTW_2$ is partially compressed. After step 3, all the QTWs referred by $N$.1.1 are maximally compressed, and they are removed during step 4. Step 6 consists of removing the four QTWs referred by $N$.1.2. The node $N$.1.2 will be removed together with $N$.1.1 during step 7: as the space released by deleting $N$.1.1 and $N$.1.2 does not suffice, some QTWs referred by $N$.2.1 are compressed too during the same compression step. The compression process ends after step 10: the $BTI$ consists of a unique node (the root) which will be definitively removed as further storage space is needed.

$CompressBTI(N,S)$ can be implemented using a recursive scheme, which works as follows. If $N$ is a leaf of the binary tree index referring to 2 QTWs,

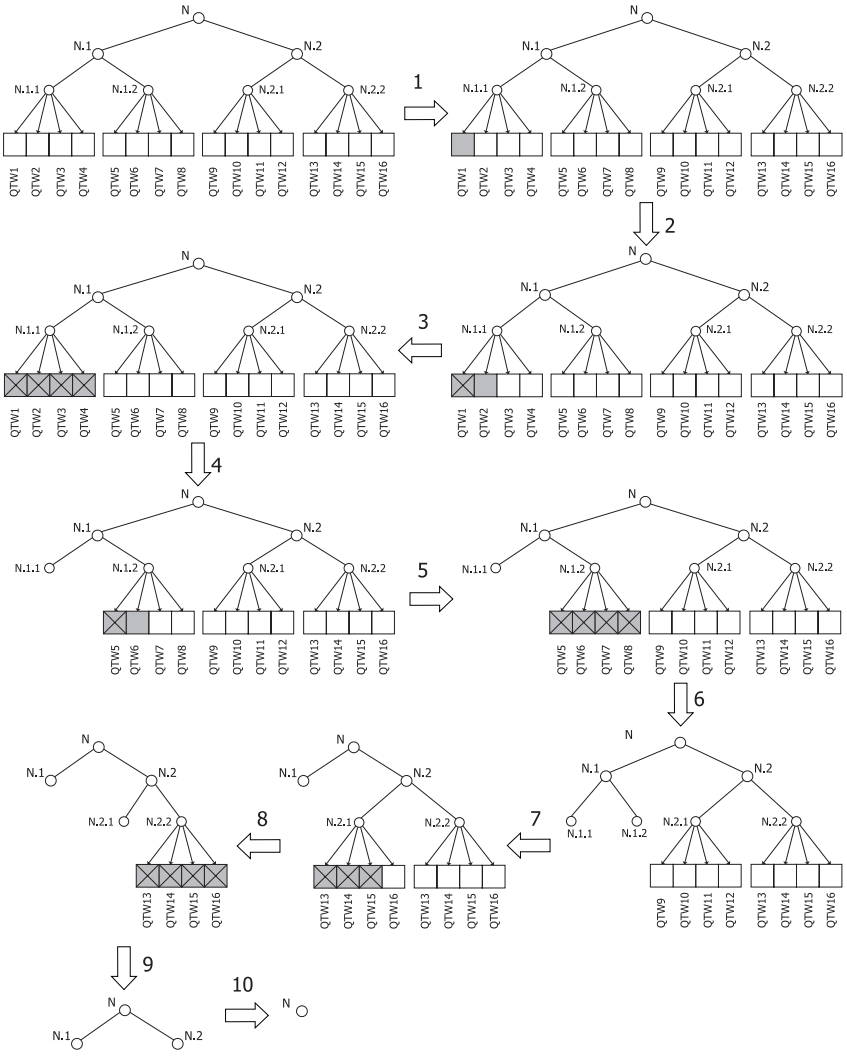


**Fig. 6.6.** Compressing a $BTI$

then it tries to release the amount $S$ of storage space by compressing the quad-tree windows referred by $N$ (the technique adopted to compress a quad-tree window will be explained later). In particular, the oldest of the referred quad-tree windows is compressed first. If the released storage space does not suffice, then the second QTW is compressed. If the released amount of storage space is not enough after all the quad-tree windows referred by $N$ have been compressed, then they are deleted definitively. If $N$ is an internal node of the binary tree index, then the function $CompressBTI$ is recursively called on its child nodes, starting from the left-most one (i.e. the "oldest" one) till

either the needed storage space is released, or all the child nodes have been maximally compressed (i.e. they have become leaves). In this case, the child nodes are deleted. The implementation of this recursive scheme is given in Algorithm 4 shown in Appendix B.4.

The compression of a $BTI$ consists in removing its nodes progressively, so that the detailed information carried by the removed nodes is kept summarized in their ancestors. This summarized data will be exploited (as described in Sect. 6.6) to estimate the original information represented in the removed QTWs underlying the $BTI$. The depth of a $BTI$ (or, equivalently, the number of QTWs in the corresponding cluster) determines the maximum degree of aggregation which is reached in the MRDS. This parameter depends on the application context. That is, the particular dynamics of the monitored world determines the average size of the time intervals which need to be investigated in order to retrieve useful information. Data summarizing time intervals which are too large w.r.t. this average size are ineffective to exploit in order to estimate relevant information. For instance, the root of a $BTI$ whose depth is 100 contains the sum of the readings produced within $2^{100}$ consecutive time windows. Therefore, the value associated to the root cannot be profitably used to estimate the sum of the readings in a single time window effectively (unless additional information about the particular data distribution carried by the stream is available). This issue will be clearer as the estimation process on a compressed Multi-Resolution Data Stream Summary will be explained (see Sect. 6.6).

### 6.5.1 Compressing Quad-Tree Windows

The strategy used for compressing binary tree indices could be adapted for compressing quad-tree windows. In fact, the compression strategy, designed for binary trees, can be easily extended to operate on 4-ary trees. For instance, we could compress a quad-tree window incrementally (i.e. as new data arrive) by searching for the left-most node $N$ having 4 child leaf nodes, and then deleting these children.

Indeed, we refine this compression strategy in order to delay the loss of detailed information inside a QTW. Instead of simply deleting a group of nodes, we try to release the needed storage space by replacing their representation with a less accurate one, obtained by using a lower numeric resolution for storing the values of the sums. To this end, we use *n-Level-Tree indices – nLT indices*, the compact structures introduced in Chap. 3 (Sect. 3.4.1), for representing approximately a portion of the QTW. In Chap. 3, $nLT$ indices have been shown to be very effective for the summarization of two-dimensional data. In the approach proposed in this chapter a 64-bit $nLT$ index is used to describe approximately both the structure and the content of a sub-tree with depth at most $n$ of the QTW. An example of $nLT$ index (in particular "*3 Level Tree index*" - 3LT) on a QTW is shown in Fig. 6.7. The left-most sub-tree $SQTW$ of the quad-tree of this figure consists of 21 nodes, which occupy

$2 \cdot 21 + 32 \cdot 16 = 554$ bits ($2 \cdot 21$ bits are used to represent their structure, whereas $32 \cdot 16$ bits to represent the sums of all non derivable nodes). The 64 bits of the $nLT$ index used for $SQTW$ are organized as follows: the first 17 bits are used to represent the second level of $SQTW$, the second 44 bits for the third level, and the remainder 3 bits for some structural information about the index. That is, the four nodes in the second level of $SQTW$ occupy $3 \cdot 32 + 4 \cdot 2 = 104$ bits in the exact representation, whereas they consume only 17 bits in the index. Analogously, the 16 nodes of the third level of $SQTW$ occupy $4 \cdot (3 \cdot 32 + 4 \cdot 2) = 416$ bits, and only 44 bits in the index. In Fig. 6.7 the first 17 bits of the 3LT index are described in more detail.

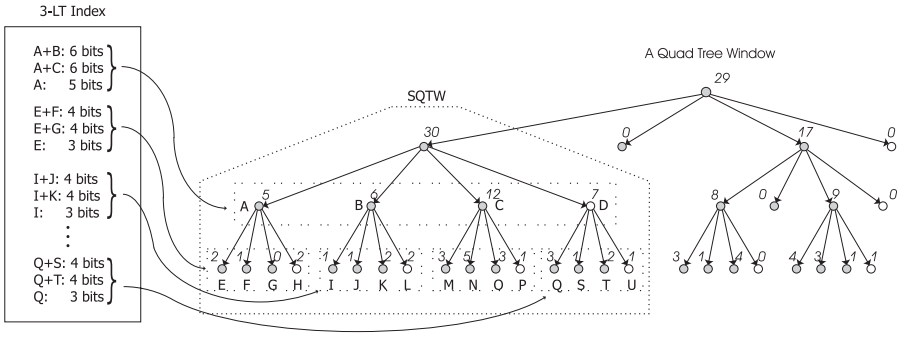


**Fig. 6.7.** A 3LT index associated to a portion of a quad-tree window

Two strings of 6 bits are used for storing $A.sum + B.sum$ and $A.sum + C.sum$, respectively, and further 5 bits are used to store $A.sum$. These string of bits do not represent the exact value of the corresponding sums, but they represent the sums as fractions of the sum of the parent node. For instance, if $R.sum$ is 100 and $A.sum = 25$, $B.sum = 30$, the 6 bit string representing $A.sum + B.sum$ stores the value: $L_{A+B} = round\left(\frac{A.sum+B.sum}{R.sum} \cdot (2^6 - 1)\right) = 35$, whereas the 5 bit string representing $A.sum$ stores the value: $L_A = round\left(\frac{A.sum}{A.sum+B.sum} \cdot (2^5 - 1)\right) = 14$. An estimate of the sums of $A, B, C, D$ can be evaluated from the stored string of bits. For instance, an estimate of $A.sum + B.sum$ is given by: $\overline{A.sum + B.sum} = \frac{L_{A+B}}{2^6-1} \cdot R.sum = 55.6$, whereas an estimate of $B.sum$ is computed by subtracting the estimate of $A.sum$ (obtained by using $L_A$) from the latter value.

The 44 bits representing the third level of $SQTW$ are organized in a similar way: two strings of 4 bits are used to represent $E.sum + F.sum$ and $E.sum + G.sum$, respectively, and a string of 3 bits is used for $E.sum$. The other nodes at the third level are represented analogously.

As described in Chap. 3, the family of nLT indices includes several types of index other than the 3LT one. Each of these indices reflects a different quad-tree structure: 3LT describes a balanced quad-tree with 3 levels, 4LT (*4 Level Tree*) an unbalanced quad-tree with at most 4 levels, and so on.

The same portion of a quad-tree window could be represented approximately by any of the proposed $nLT$ indices. In Sect. 3.4.1 a metric for choosing the most "suitable" $nLT$ index to approximate a portion of a quad-tree is provided: that is, the index which permits us to re-construct the original data distribution most accurately. As it will be clear next, this metric is adopted in our compression technique: the oldest "portions" of the quad-tree window are not deleted, but they are replaced with the most suitable $nLT$ index.

The algorithm which uses indices to compress a QTW is analogous to Algorithm 4 (suitably adapted to work with 4-ary trees). That is the $QTW$ to be compressed is visited in order to reach the left-most node $N$ (i.e. the oldest node) having one of the following properties: 1) $N$ is an internal node of the $QTW$ such that $size(N.range) = 16$; 2) the node $N$ has 4 child leaf nodes, and each child is either null or equipped with an index.

Once the node with one of these properties is found, it is equipped with the most suitable $nLT$ index, and all its descending nodes are deleted. In particular, in case 1 (i.e. $N$ is at the last but two level of the uncompressed QTW) $N$ is equipped with a $3LT$ index. In case 2 the following steps are performed:

1. all the children of $N$ which are equipped with an index are "expanded": that is, the quad-trees represented by the indices are approximately re-constructed;
2. the most suitable $nLT$ index $I$ for the quad-tree rooted in $N$ is chosen, using the above cited metric;
3. $N$ is equipped with $I$ and all the nodes descending from $N$ are deleted.

The compressed QTW obtained as described above is not, in general, a full 4-ary tree, as nodes can be deleted during the compression process. Furthermore, leaf nodes can be possibly equipped with an nLT index. Thus, the compact physical representation of a QTW, presented in Sect. 6.3.3, has to be modified in order to represent a compressed QTW. In particular, the pairs of bits which encode the tree structure are redefined as follows: (1) $\langle 0, 0\rangle$ means non null leaf node equipped with nLT index, (2) $\langle 0, 1\rangle$ means null leaf node, (3) $\langle 1, 0\rangle$ mean non null leaf node not equipped with nLT index, (4) $\langle 1, 1\rangle$ mean non leaf node; the array of sums representing the content of the tree is augmented with the nLT indices associated to the leaves of the compressed QTW.

### 6.5.2 The Summarization Technique in Short

The aim of our summarization technique is to store as much information as possible in a given bounded storage space, satisfying the following constraints:

- summary data must be efficient to update;
- answering to range queries on summary data must be efficient to perform.

The summarization strategy is based on the idea of releasing some storage space from "old" information stored in the Multi-Resolution Data Stream Summary as new data arrive. To this end the whole sensor data stream is divided into equally sized time windows. Each time window is represented by means of a quad-tree, called *quad-tree window* - QTW, whose hierarchical structure is particularly suitable to be compressed. A QTW is obtained by choosing a time unit and then partitioning the time window into unitary time intervals $\Delta t_1, \ldots, \Delta t_n$. Each leaf node of the QTW contains the sum of the readings produced by a source $s_i$ in a unitary time interval $\Delta t_j$. Internal nodes of a QTW have four children and contain the sum of their values. The root of a QTW contains the sum of all the readings produced by all the sources $s_1, \ldots, s_n$ in the corresponding time window.

Time windows are then grouped into clusters. An index is associated to each cluster in order to access time windows efficiently. The index, called $BTI$, consists of a binary tree having the following structure: i) the root contains the sum of all the readings represented in the cluster; ii) each leaf node refers to two consecutive quad-tree windows and contains the sum of their readings; iii) each internal node has two children and contains the sum of their values. The overall sensor data stream is represented by a list of consecutive $BTI$s.

Therefore, the data stream is stored into a hierarchical structure where information is represented using a "multi-resolution" scheme: the highest detail level is obtained in the leaf nodes of the QTWs (where the readings of the single sources are stored), and the resolution of the stored information decreases as we get closer to the roots of the $BTI$s of the list (where summarized information on the readings produced by all the sources in a wide time interval is represented).

As new data arrive, if the available storage space is not enough, the "oldest" $BTI$ of the list is compressed to release the space needed for representing the new arrivals. The oldest $BTI$ of the list is compressed incrementally till the needed space is released. The compression process sacrifices the oldest detailed information stored in the $BTI$: first, the oldest QTWs are compressed using an ad hoc technique for quad-trees. The compression of a QTW is done by replacing progressively the exact representation of some of its portions with an approximate one. The approximation is obtained by either using a lower numeric resolution (i.e. less than 32 bits) for representing the value of the sums, or deleting some of its nodes definitively. Therefore, the resolution of the information stored in a QTW decreases as the compression process goes on, till the QTW cannot be further compressed. When all the QTWs referred by a leaf of the $BTI$ cannot be further compressed, these QTWs are deleted: their sums are kept summarized in the nodes of the overlying $BTI$. As QTWs are deleted, the compression process spreads into the overlying $BTI$. The $BTI$ is compressed using an analogous strategy: the needed storage space is released by deleting two leaf nodes at a time, collapsing them in their parent node. The sacrificed leaf nodes are the oldest nodes containing no reference to any QTW: they are ancestors of the QTWs which have been deleted during

some previous compression step.

The compression of a $BTI$ goes on by pruning its nodes, till the $BTI$ consists of anything but the root. When further storage space is needed, the $BTI$ is removed from the list definitively: every information about the readings generated in the corresponding time interval will be lost.

## 6.6 Estimating Range Queries on a Multi-Resolution Data Stream Summary

A sum range query $Q = \langle s_i..s_j, [t_{start}..t_{end}] \rangle$ can be computed by summing the contributions of every QTW corresponding to a time window overlapping $[t_{start}..t_{end}]$. The QTWs underlying the list of $BTI$s are represented by means of a linked list in time ascending order. Therefore the sub-list of QTWs giving some contribution to the query result can be extracted by locating the first (i.e. the oldest) and the last (i.e. the most recent) QTW involved in the query (denoted, respectively, as $QTW_{start}$ and $QTW_{end}$). This can be done efficiently by accessing the list of $BTI$s indexing the QTWs, and locating the first and the last $BTI$ involved in the query. That is, the binary tree indices $BTI_{start}$ and $BTI_{end}$ which contain a reference to $QTW_{start}$ and $QTW_{end}$, respectively. $BTI_{start}$ and $BTI_{end}$ can be located efficiently, by performing a binary search on the list of $BTI$s. Then, $QTW_{start}$ and $QTW_{end}$ are identified by visiting $BTI_{start}$ and $BTI_{end}$. The answer to the query consists of the sum of the contributions of every QTW between $QTW_{start}$ and $QTW_{end}$. The evaluation of each of these contributions is explained in detail in the next section.

Indeed, as the Sensor Data Stream Summary is progressively compressed, it can happen that $QTW_{start}$ has been removed, and the information it contained is only represented in the overlying $BTI$ with less detail. Therefore, the query can be evaluated as follows:

1. the contribution of all the removed QTWs is estimated by accessing the content of the nodes of the $BTI$s where these QTWs are summarized;
2. the contribution of the QTWs which have not been removed is evaluated after locating the oldest QTW involved in the query which is still stored. This QTW will be denoted as $QTW'_{start}$.

Indeed, it can happen that $QTW_{end}$ has been removed either. This means that all the QTWs involved in the query have been removed by the compression process to release some space, as the QTWs are removed in time ascending order. In this case, the query is evaluated by estimating the contribution of each involved QTW by accessing only the nodes of the overlying $BTI$s.

For instance, consider the MRDS consisting of two $BTI$s shown in Fig. 6.8. The QTWs and the BTI nodes whose perimeter is dashed (i.e. $QTW_1$, $QTW_2, \ldots, QTW_8$, as well as the nodes $N1.1.1$ and $N1.1.2$) have been removed

by the compression process. The query represented with a grey box is evaluated by summing the contributions of the $BTI_1$ node $N1.1$ with the contribution of each QTW belonging to the sequence $QTW_9, QTW_{10}, \ldots, QTW_{29}$.

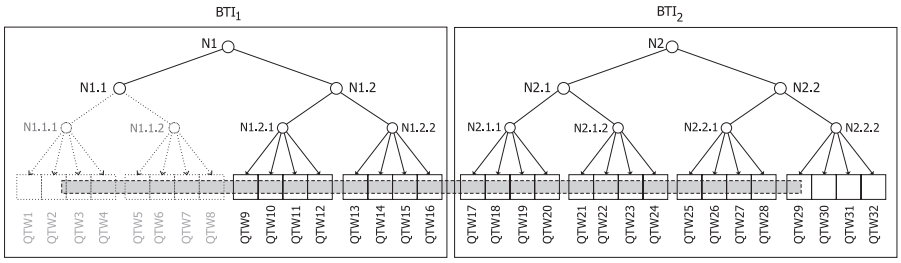


**Fig. 6.8.** A range query on a MRDS

The query estimation strategy is implemented in Algorithm 5 reported in Appendix B.5. This algorithm uses a function $BTIBinarySearch$ which takes as arguments a Multi-Resolution Data Stream Summary and the time boundaries of the range query, and returns the first and the last $BTI$ of the summary involved in the query. Moreover, it uses the function $EstimateAndLocate$ implemented in Algorithm 6 reported in Appendix B.5. This function is first invoked on $BTI_{start}$ and performs two tasks: 1) it evaluates the contribution of the $BTI$ nodes involved in the query where the information of the removed QTWs is summarized, and 2) it locates (if possible) $QTW'_{start}$, i.e. the first QTW involved in the query which has not been removed. If $QTW'_{start}$ is not referred by $BTI_{start}$, $EstimateAndLocate$ is iteratively invoked on the subsequent $BTI$s, till either $QTW'_{start}$ is found or all the $BTI$s involved in the query have been visited. The contribution of the $BTI$ leaf nodes to the query estimate is evaluated by performing linear interpolation. The use of linear interpolation on a leaf node $N$ of a $BTI$ is based on the assumption that data are uniformly distributed inside the two-dimensional range $N.range$ (Continuous Value Assumption). If we denote the two dimensional range corresponding to the intersection between $N.range$ and the range of the query $Q$ as $N \cap Q$, and the size of the whole two dimensional range delimited by the node $N$ as $size(N)$, the contribution of $N$ to the query estimate is given by: $\frac{size(N \cap Q)}{size(N)} \cdot N.sum$.

### 6.6.1 Estimating a Sum Range Query inside a QTW

The contribution of a QTW to a query $Q$ is evaluated as follows. The quadtree underlying the QTW is visited starting from its root (which corresponds to the whole time window). When a node $N$ is being visited, three cases may occur:

1. *the range corresponding to the node is external to the range of* $Q$: the node gives no contribution to the estimate;

2. *the range corresponding to the node is entirely contained into the range of* $Q$: the contribution of the node is given by the value of its sum;
3. *the range corresponding to the node partially overlaps the range of* $Q$: if $N$ is a leaf and is not equipped with any index, linear interpolation is performed for evaluating which portion of the sum associated to the node lies onto the range of the query. If $N$ has an index, the index is "expanded" (i.e. an approximate quad-tree rooted in $N$ is re-constructed using the information contained in the index). Then the new quad-tree is visited with the same strategy as the QTW to evaluate the contribution of its nodes. Finally, if the node $N$ is internal, the contribution of the node is the sum of the contributions of its children, which are recursively evaluated.

The pre-aggregations stored in the nodes of quad-tree windows make the estimation inside a QTW very efficient. In fact, if a QTW node whose range is completely contained in the query range is visited during the estimation process, its sum contributes to the query result exactly, so that none of its descending nodes must be visited. This means that, generally, not all the leaf nodes involved in the query need to be accessed when evaluating the query estimate. The overall estimation process turns out to be efficient thanks to the hierarchical organization of data in the QTWs, as well as the use of the overlying $BTI$s which permits us to locate the quad-tree windows efficiently. We point out that the $BTI$s involved in the query can be located efficiently too, i.e. by performing a binary search on the ordered list of $BTI$s stored in the MRDS. The cost of this operation is logarithmic with respect to the list length, which is, in turn, proportional to the number of readings represented in the MRDS.

### 6.6.2 Answering Continuous (Range) Queries

The range query evaluation paradigm on the data summary can be easily extended to deal with *continuous range queries*. A *continuous query* is a triplet $Q = \langle s_i..s_j, \Delta T_{start}, \Delta T_{end} \rangle$ (where $\Delta T_{start} > \Delta T_{end}$) whose answer, at the current time $t$, is the evaluation of an aggregate operator (such as *sum*, *count*, *avg*, etc.) on the values produced by the sources $s_i, s_{i+1}, \ldots, s_j$ within the time interval $[t - \Delta T_{start}..t - \Delta T_{end}]$. In other words, a continuous query can be viewed as range query whose time interval "moves" continuously, as time goes on. The output of a continuous query is a stream of (simple) range query answers which are evaluated with a given frequency. That is, the answer to a continuous query $Q = \langle s_i..s_j, \Delta T_{start}, \Delta T_{end} \rangle$ issued at time $t_0$ with frequency $\Delta t$ is the stream consisting of the answers of the queries $Q_0 = \langle s_i..s_j, t_0 - \Delta T_{start}, t_0 - \Delta T_{end} \rangle, Q_1 = \langle s_i..s_j, t_0 - \Delta T_{start} + \Delta t, t_0 - \Delta T_{end} + \Delta t \rangle, Q_2 = \langle s_i..s_j, t_0 - \Delta T_{start} + 2 \cdot \Delta t, t_0 - \Delta T_{end} + 2 \cdot \Delta t \rangle, \ldots$. The $i$-th term of this stream can be evaluated efficiently if we exploit the knowledge of the $(i-1)$-th value of the stream, provided that $\Delta t \ll \Delta T_{start} - \Delta T_{end}$. In this case the ranges of two

consecutive queries $Q_{i-1}$ and $Q_i$ are overlapping, and $Q_i$ can be evaluated by answering two range queries whose size is much less than the size of $Q_i$. These two range queries are $Q' = \langle s_i..s_j, t_0 - \Delta T_{start} + (i-1) \cdot \Delta t, t_0 - \Delta T_{start} + i \cdot \Delta t \rangle$, and $Q'' = \langle s_i..s_j, t_0 - \Delta T_{end} + (i-1) \cdot \Delta t, t_0 - \Delta T_{end} + i \cdot \Delta t \rangle$. Thus we have: $Q_i = Q_{i-1} - Q' + Q''$.

# Conclusions

In this thesis we have investigated the issue of efficiently retrieving aggregate information from multi-dimensional data. We have described the main application scenarios where data is suitably modelled according to the multi-dimensional paradigm, and we have addressed one of the most relevant tasks in these contexts: the efficient computation of aggregations over specified ranges of the domain (called *range queries*).
We have shown how the multi-dimensional model is especially well suited for the evaluation of range queries, although the cost of computing exact answers is unaffordable in most contexts, where efficiency requirements go beyond the need for accurate query results.

We have presented summarization of multi-dimensional data as a widely accepted approach for speeding up the query answering process by allowing some approximation in query responses. We have given an overview of the most popular and effective state-of-the-art summarization techniques, focusing our attention on histogram-based ones: they store aggregate information about the data distribution inside a set of non-overlapping buckets partitioning the data domain. We have reviewed the main theoretical results about the construction of the *V-Optimal* histogram within a given storage space bound, showing that it's unfeasible to afford in practice. Thus, we have described the main heuristics adopted in the literature for constructing histograms, remarking their poor performances in the multi-dimensional context.

In this scenario, we have defined new histogram-based summarization techniques for multi-dimensional data which outperform state-of-the-art ones, and we have provided the following main contributions:

1. We have shown that high accuracy in estimating query answers can be achieved by designing summary structures suited to data with specific number of dimensions. We have focused on two-dimensional data by proposing a new summary structure (the *Quad-tree Summary* – QTS) which is based on a quad-tree partition of the multi-dimensional space. In this context we have shown how the use of "rigid" partitioning schemes,

which can be stored very compactly (such as the quad-tree one), although limiting flexibility in the arrangement of buckets, can result in a better data description w.r.t. arbitrary partitions. In fact, the storage space saved thanks to the compact representation of the histogram structure can be reinvested to store (into suitable indices) additional information about intra-bucket data distribution. We have provided efficient greedy strategies that construct effective QTSs, and experimental results showing that QTSs perform better than state-of-the-art summarization techniques.

2. We have investigated how binary partitions can be used to define very effective histograms for data with high dimensionality. We have introduced two new classes of histogram: *HBH* which adopts a binary partition of the multi-dimensional space and *GHBH* which additionally introduces a grid constraint on the hierarchical partitioning. We have shown how *HBH* and *GHBH* can be stored in a redundancy-free space-saving fashion, which allows more buckets in the same storage space bound w.r.t traditional histograms. We have also shown that the increased number of buckets results in better histogram estimation performances only if combined with effective heuristics for the histogram construction.
   We have investigated several heuristics capable of effectively arranging the available buckets in the multi-dimensional space and, by means of experiments, we have analyzed the performances (in terms of accuracy) of different combinations *heuristic/representation model*. We have found out that *GHBH* with "low"-degree grids, combined with one of the proposed heuristics, yields the highest accuracy. We have provided a thorough experimental analysis comparing the best performing *GHBH* with other state-of-the-art multi-dimensional summarization techniques, proving its effectiveness also for high-dimensional data distributions.
3. We have extended state-of the art results on the complexity of constructing V-Optimal histograms, by analyzing the construction cost of V-Optimal QTSs and V-Optimal *HBH*s and *GHBH*s. We have shown that, although polynomial, the cost of computing the optimal solutions is unaffordable in practice.
4. We have combined summarization of multi-dimensional data with the use of clustering techniques. We have proposed a new multi-dimensional histogram (CHIST) which is constructed by invoking a density-based clustering algorithm to partition data into dense and sparse regions, which are further partitioned according to a grid-based scheme. We have shown how (possible nesting) buckets generated by data partitioning can be assembled, so as to enhance both efficiency and accuracy of query answering. We have provided experimental results on both synthetic and real-life data, proving how CHIST overcomes the main limitation of existing multi-dimensional histograms.
5. We have proposed a new dynamic summarization technique for evaluating approximate aggregations on sensor data streams; it consists in the incremental maintenance of a quad-tree based summary structure (the MRDS),

where more recent data is represented with more detail than older one. We have shown how the MRDS is dynamically populated and its oldest summary information is progressively compressed when new sensor readings are received, in order to release the needed storage space. We have presented an efficient algorithm for answering range queries on the MRDS, which exploits a hierarchical indexing structure (embedded in the MRDS) allowing efficient location of summarized information over time. We have also extended the proposed query estimation paradigm to perform incremental evaluation of *continuous range queries*, i.e. queries producing output streams consisting of (simple) range query answers, evaluated at a given frequency.

Further work on summarization of multi-dimensional data will deal with the following issues:

*a.* Improving the efficiency of accessing summary data in the proposed hierarchical histograms. In particular, we are interested in developing new physical representation models aimed at minimizing not only the storage consumption of the histogram, but also the overhead of de-compression (that is the overhead of evaluating queries directly on the compactly encoded structure).

*b.* Extending to a dynamic environment the summarization techniques proposed for (static) multi-dimensional data. The issue of histogram incremental maintenance has been addressed, in this thesis, only for two-dimensional data, in the context of sensor data stream management. In this context data sources have been assumed linearly ordered, thus allowing the data stream to be modelled over time as a two-dimensional data set; however, in some cases this assumption is not suitable as data sources can be distributed in a multi-dimensional environment. Thus, dynamic summarization techniques for general multi-dimensional data are needed. Furthermore, even when summarizing non-streaming data, if frequent changes to multi-dimensional data sets are allowed, re-computation of the histogram from scratch can be too inefficient, and techniques for its incremental update should be developed. Most of our summarization techniques are prone to a dynamic re-adaptation; in particular, clustering-based histograms can be maintained dynamically by exploiting incremental clustering algorithms (that is algorithms which propagate data updates to the computed clusterization).

*c.* Considering different clustering techniques to be embedded into our clustering-based histograms. In fact, in our implementation of CHIST, we adopted the well-known clustering algorithm DBSCAN, whose execution time turns out to dominate the cost of the histogram construction. DBSCAN is known to provide poor performances (in terms of computational cost) on large data sets with high-dimensionality. In future work different clustering techniques will be adopted, in order to study how they can be

exploited to improve the histogram construction cost, while preserving its accuracy.

*d.* developing metrics for deciding the best histogram technique to be adopted on a given multi-dimensional data set. In fact, CHIST histogram is likely to achieve high performances on data sets characterized by several clusters dispersed in the data domain; while data sets without clusters are likely to be described better by GHBH (in particular, clusters are unlikely to occur in data having very high dimensionality). In general, depending on data density and on the distribution of clusters on the domain, either CHIST or GHBH can be valid options. An alternative approach could be to combine CHIST and GHBH by partitioning the layers resulting from the clustering step with a GHBH histogram (rather than a grid-based strategy). This option is worth being considered on static data sets, while in the case that the histogram has to be maintained incrementally, grid-partitioning is better-suited as it has the advantage of being easy and efficient to re-compute.

*e.* Extending the issue of approximate query answering to a richer set of queries. This implies considering the computation of other kinds of aggregate operators, as well as other query paradigms which can be found in specific application domains – such as spatial window queries in GIS or general SQL operators on streaming data.

# A

# Proof of Theorems

## A.1 Proof of Proposition 4.6

*Let $D$ be a $d$-dimensional data distribution, $B$ a storage space bound, and $T$ a type of histogram (where $T$ is either FBH, HBH or k-GHBH). The number of buckets $\beta_T$ of a $B$-maximal histogram $H$ of type $T$ on $D$ are in the ranges reported in Table 1.*

| **Type** | **Number of buckets** |
|---|---|
| *FBH* | $\beta_{FBH} = \left\lfloor \frac{B}{32\cdot(2\cdot d+1)} \right\rfloor$ |
| *HBH* | $\beta_{HBH}^{min} = \left\lfloor \frac{B+\lceil log\ d\rceil+34}{67+\lceil log\ d\rceil} \right\rfloor \leq \beta_{HBH} \leq \left\lfloor \frac{B+\lceil log\ d\rceil+2}{35+\lceil log\ d\rceil} \right\rfloor = \beta_{HBH}^{max}$ |
| k-*GHBH* | $\beta_{GHBH}^{min} = \left\lfloor \frac{B+log\ k+\lceil log\ d\rceil+2}{35+log\ k+\lceil log\ d\rceil} \right\rfloor \leq \beta_{GHBH} \leq \left\lfloor \frac{B+log\ k+\lceil log\ d\rceil-30}{3+log\ k+\lceil log\ d\rceil} \right\rfloor = \beta_{GHBH}^{max}$ |

***Table 1***

**Proof.**

**1.** $T$=*FBH*. The size of an *FBH* having $\beta$ buckets is precisely $size(FBH)=(2\cdot d+1)\cdot 32\cdot\beta$ bits, so that $size(FBH) \leq B$ holds for all values of $\beta \leq \left\lfloor \frac{B}{32\cdot(2\cdot d+1)} \right\rfloor$. Therefore the latter bound on $\beta$ is the number of buckets of a $B$-maximal *FBH*.

**2.** $T$=*HBH* or $T$=k-*GHBH*. An *HBH*, as well as a k-*GHBH*, with $\beta$ buckets has a space consumption which can vary between a minimum and a maximum value (depending on the partition tree and on the data distribution). We denote by $size_T^{min}(\beta)$ and $size_T^{max}(\beta)$ respectively, the minimum and the maximum space consumption of any histogram of type $T$ having $\beta$ buckets. The upper bound on the number of buckets of a $B$-maximal histogram of

type $T$ is obtained as the largest value of $\beta$ which satisfies the inequality $size_T^{min}(\beta) \leq B$. Similarly, the lower bound on the number of buckets of a $B$-maximal histogram of type $T$ is obtained as the largest value of $\beta$ which satisfies the inequality $size_T^{max}(\beta) \leq B$.
We will next compute $size_{HBH}^{min}$, $size_{GHBH}^{min}$, $size_{HBH}^{max}$ and $size_{GHBH}^{max}$ as functions of $\beta$.

According to the physical representation of an *HBH* described in Sect. 4.2.3, the size of an *HBH* $H$ with $\beta$ buckets can be expressed as the sum of four contributions:

$$size_{HBH}(H) = (2 \cdot \beta - 1) + (\beta - 1) \cdot (\lceil log\ d \rceil + 32) + ndl(H) + 32 \cdot ndn^{+}(H)$$

where $ndl(H)$ and $ndn^{+}(H)$ stand for the number of non-derivable leaves of $H$ and, respectively, the number of non-null non-derivable nodes of $H$. Analogously, we will denote by $ndl^{+}(H)$ and $ndl^{0}(H)$ the number of non-null non-derivable leaves and, respectively, the number of null derivable leaves of $H$. As $ndl(H) = ndl^{+}(H) + ndl^{0}(H)$ and $ndn^{+}(H) = \beta - ndl^{0}(H)$, then

$$size_{HBH}(H) = (2 \cdot \beta - 1) + (\beta - 1) \cdot (\lceil log\ d \rceil + 32) + 32 \cdot \beta + ndl^{+}(H) - 31 \cdot ndl^{0}(H)$$

Similarly the size of a k-*GHBH* $H$ having $\beta$ buckets is

$$size_{GHBH}(H) = (2 \cdot \beta - 1) + (\beta - 1) \cdot (\lceil log\ d \rceil + \log\ k) + 32 \cdot \beta + ndl^{+}(H) - 31 \cdot ndl^{0}(H)$$

The expressions for $size_T(H)$ (for either $T$=*HBH* and $T$=*GHBH*) have minimum value when $ndl^{+}(H) = 0$ and $ndl^{0}(H) = \beta - 1$, which occurs for a histogram of type $T$ with $\beta$ buckets where all but one leaves are non-derivable and null. Likewise, the expressions for $size_T(H)$ have maximum value when $ndl^{+}(H) = \beta - 1$ and $ndl^{0}(H) = 0$, which occurs for a histogram of type $T$ with $\beta$ buckets where all but one leaves are non-derivable and not null. Thus, the minimum and maximum storage consumption of an *HBH* and a *GHBH* having $\beta$ buckets are, respectively:

$$size_{HBH}^{min}(\beta) = \beta \cdot (35 + \lceil log\ d \rceil) - \lceil log\ d \rceil - 2;$$

$$size_{GHBH}^{min}(\beta) = \beta \cdot (3 + \lceil log\ d \rceil + log\ k) - \lceil log\ d \rceil - log\ k + 30;$$

$$size_{HBH}^{max}(\beta) = \beta \cdot (67 + \lceil log\ d \rceil) - \lceil log\ d \rceil - 34;$$

$$size_{GHBH}^{max}(\beta) = \beta \cdot (35 + \lceil log\ d \rceil + log\ k) - \lceil log\ d \rceil - log\ k - 2.$$

As said above, $\beta_{HBH}^{max}$, $\beta_{GHBH}^{max}$, as well as $\beta_{HBH}^{min}$, $\beta_{GHBH}^{min}$, are straightforward. □

## A.2 Proof of Theorem 4.8

*Let $D$ be a $d$-dimensional data distribution, $B$ a storage space bound, and $T$ a type of histogram (where $T$ is either FBH, HBH, or k-GHBH). A V-Optimal histogram $H^\star$ of type $T$ on $D$ w.r.t. $B$ can be computed in the complexity bounds reported in Table 2.*

| Type of histogram | Complexity bound for V-Optimal histogram |
|---|---|
| *FBH* | $O(\frac{B^2}{d \cdot 2^d} \cdot n^{2d+1})$ |
| *HBH* | $O\left(d \cdot \frac{B^2}{2^d} \cdot n^{2d+1}\right)$ |
| k-*GHBH* | $O\left(d \cdot \frac{B^2}{2^d} \cdot k^{d+1} \cdot n^d\right)$ |

***Table 2***

**Proof.**

**1.** $T$=*FBH*. The problem of finding the V-Optimal *FBH* on $D$ can be solved by the following dynamic programming approach. Given a block $b$ of $D$, denoting the storage space needed to represent a single block as $\gamma = (2 \cdot d + 1) \cdot 32$, the minimum SSE of any *FBH* $H$ on $b$ with $size(H) \leq S$ can be defined recursively as follows:

1. $SSE^*(b, S) = \infty$, if $S < \gamma$;
2. $SSE^*(b, S) = SSE(b)$, if $S \geq \gamma \ \wedge \ (S < 2{\cdot}\gamma \ \vee \ vol(b){=}1)$;
3. $SSE^*(b, S) = min\{SSE^*(b^{low}, S1) + SSE^*(b^{high}, S2) \mid \ \langle b^{low}, b^{high}\rangle$ *is a binary split on* $b$, $S1{>}0$, $S2{>}0$, $S1{+}\ S2{=}S\}$, otherwise

Our optimization problem consists in evaluating $SSE^*(D, B)$. As implied by the above recursive definition, $SSE^*(D, B)$ can be computed after evaluating $SSE^*(b, S)$ for each block $b$ of $D$ and each $S$ in $[0..B]$ which is multiple of $\gamma$. At each step of the dynamic programming algorithm, $SSE^*(b, S)$ is evaluated by accessing $O(d \cdot n \cdot \frac{B}{d})$ values computed at the previous steps, as the possible binary splits of a block are $O(d{\cdot}n)$ and there are $O(\frac{B}{d})$ possible ways to divide $S$ into two halves which are multiple of $\gamma$.
The number of different $SSE^*(b, S)$ to be computed are $O(\frac{B}{d} \cdot \frac{n^{2d}}{2^d})$, as the number of sub-blocks of $D$ are $O(\frac{n^{2d}}{2^d})$, and the number of possible values of $S$ are $O(\frac{B}{d})$.
On the other hand, the SSE of all the sub-blocks of $D$ must be computed. It can be shown that the cost of accomplishing this task is dominated by $O(n^{2d})$. It follows that the overall cost of the dynamic programming algorithm is $O(\frac{B^2}{d \cdot 2^d} \cdot n^{2d+1})$.

**2.** $T$=*HBH*. The problem of finding the V-optimal *HBH* can be formalized and solved following the same approach as the one just described for *FBH*. The main difference is that when evaluating the optimal *HBH* on a block $b$, two distinct optimization problems must be addressed, corresponding to the cases that $b$ appears in $HBH^*$ as either a left-hand child or a right-hand child of some node. In fact, due to the physical representation paradigm (Sect. 4.2.3), the storage consumption of an *HBH* constructed on $b$ is different in these two cases. Intuitively enough, this leads to a recursive formulation of the V-optimal problem which is different from the one described for *FBH*. We define the minimum SSE of any *HBH* $H$ on $b$ having $size(H) \leq S$ both in the case that $b$ is considered as a left-hand child node (which we denote by $SSE^*_{left}(b,S)$) and a right-hand child node (which we denote by $SSE^*_{right}(b,S)$). Both $SSE^*_{left}(b,S)$ and $SSE^*_{right}(b,S)$ can be defined recursively in a way that is similar to the recursive definition of $SSE^*(b,S)$ for *FBH*. The main differences are that in the non-recursive cases (i.e. the cases such that no *HBH* can be constructed or no split can be performed on $b$) more complex conditions on the storage space must be expressed, as the storage space consumption of $b$ depends also on whether $b$ is null or not. Moreover, the recursive case is defined as the minimum value of $SSE^*_{left}(b^{low},S1) + SSE^*_{right}(b^{high},S2)$, for each possible binary split $< b^{low}, b^{high} >$ on $b$, and for each $S1$ and $S2$ which are consistent with the bound $S$ on the overall space consumption allowed on $b$. The dynamic programming algorithm must compute both $SSE^*_{left}(b,S)$ and $SSE^*_{right}(b,S)$ for each sub-block of $D$ and for each $S$ in $[0..B]$. This algorithm computes $O(B \cdot \frac{n^{2d}}{2^d})$ values of $SSE^*_{left}(b,S)$ and $O(B \cdot \frac{n^{2d}}{2^d})$ values of $SSE^*_{right}(b,S)$, where each one is computed in time $O(d \cdot n \cdot B)$.

**3.** $T$=k-*GHBH*. The problem of finding the V-Optimal k-*GHBH* can be formalized by means of some minor adaptation in the definition of $SSE^*_{left}(b,S)$ and $SSE^*_{right}(b,S)$ introduced for *HBH*s: 1) each constant which represents a storage space consumption is changed by replacing the 32 bits needed to represent the splitting position with $log\ k$ bits. 2) the minimum value of $SSE^*_{left}(b,S) + SSE^*_{right}(b,S)$ which define the recursive case is evaluated by considering only the binary splits of degree $k$.
The dynamic programming algorithm which computes all the values of both $SSE^*_{left}(b,S)$ and $SSE^*_{right}(b,S)$ needed to compute $SSE^*_{left}(D,B)$ exhibits a different complexity bound w.r.t. the case of *HBH* as:

1. The cost of computing a single value of $SSE^*_{left}(b,S)$ or $SSE^*_{right}(b,S)$ is reduced to $O(d \cdot k \cdot B)$, since all the possible binary splits of degree $k$ on a block are $d \cdot k$ (instead of $d \cdot n$).
2. Due to the restriction on the possible binary splits of a block, the recursive definition of $SSE^*(D,B)$ induces the computation of $SSE^*_{left}(b,S)$ or $SSE^*_{right}(b,S)$ for a proper subset of all the possible sub-blocks of $D$. It can be shown that the number of such blocks is $O(n^d \cdot \frac{k^d}{2^d})$ (instead of

$O(\frac{n^{2d}}{2^d})$). Thus, the number of values of $SSE^*_{left}(b,S)$ or $SSE^*_{right}(b,S)$ to be computed is $O(n^d \cdot \frac{k^d}{2^d})$ for each $S$ in $[0..B]$.

3. The cost of computing the SSE of all the $O(n^d \cdot \frac{k^d}{2^d})$ blocks is $O(n^d \cdot k^d)$.

All considered, the cost of the dynamic programming algorithm which computes the V-Optimal *GHBH* of degree $k$ on $D$ is $O(d \cdot \frac{B^2}{2^d} \cdot k^{d+1} \cdot n^d)$. □

## A.3 Proof of Theorem 4.9

*Given a d-dimensional data distribution D with volume $n^d$ containing exactly N non-null points, the time complexity of the greedy algorithms computing a histogram of type T (where T is either FBH, HBH or k-GHBH) on D, adopting either the sparse data model, or the non-sparse data model, or pre-computation, are reported in Fig. A.1, where $\alpha = n$ if* Max-Var$^{marg}$/Max-Red$^{marg}$ *criterion is adopted, and $\alpha = k$ for all the other greedy criteria.*

| | ***Sparse model*** | ***Non-Sparse model*** | ***Using pre-computation*** | |
|---|---|---|---|---|
| | | | *Cost of pre-computation* | *Cost of computation* |
| *FBH* | $O\left((d \cdot N + d \cdot n) \cdot \beta^{max}_{FBH}\right)$ | $O\left(n^d \cdot \beta^{max}_{FBH}\right)$ | $O\left(2^d \cdot n^d\right)$ | $O\left(2^d \cdot d \cdot n \cdot \beta^{max}_{FBH}\right)$ |
| *HBH* | $O\left((d \cdot N + d \cdot n) \cdot \beta^{max}_{HBH}\right)$ | $O\left(n^d \cdot \beta^{max}_{HBH}\right)$ | $O\left(2^d \cdot n^d\right)$ | $O\left(2^d \cdot d \cdot n \cdot \beta^{max}_{HBH}\right)$ |
| *k-GHBH* | $O\left((d \cdot N + d \cdot \alpha) \cdot \beta^{max}_{GHBH}\right)$ | $O\left(n^d \cdot \beta^{max}_{GHBH}\right)$ | $O\left(2^d \cdot n^d\right)$ | $O\left(\left(2^d \cdot d \cdot \alpha + \log \beta^{max}_{GHBH}\right) \cdot \beta^{max}_{GHBH}\right)$ |

**Fig. A.1.** Complexity bounds of Greedy Algorithm

**Proof.** Complexity bounds when pre-computation is not used were obtained by multiplying the maximum number of iterations of Greedy Algorithm (which are $O(\beta^{max}_T)$) for the cost of each iteration. The cost of each iteration of Greedy Algorithm is dominated by the cost of evaluating the greedy criterion $G$ on a bucket $b$, that is by the cost of computing $Evaluate(G,b)$ (which has been computed in Sect. 4.3.2 for *FBH* and *HBH*, and in Sect. 4.3.2 for *GHBH*).
In the case that pre-computation of $F$ and $F^2$ is performed, the cost of Greedy Algorithm is given by the sum of three contributions:

1. *PreComp*: the cost of pre-computing $F$ and $F^2$;
2. $C^U$: the cost of all the updates to the priority queue;
3. $C^E$: the cost of computing the function *Evaluate* for all the nodes to be inserted in the queue.

These contributions can be computed as follows:
1) Both $F$ and $F^2$ can be constructed "incrementally", by accessing only once

each cell of the multi-dimensional array corresponding to $D$ and accessing $2^d - 1$ cells of $F$ and $F^2$ computed at the previous steps. For instance, in the two dimensional case:
$F[< i, j >] = D[< i, j >] + F[< i-1, j >] + F[< i, j-1 >] - F[< i-1, j-1 >]$, and
$F^2[< i, j >] = (D[< i, j >])^2 + F^2[< i-1, j >] + F^2[< i, j-1 >] - F^2[< i-1, j-1 >]$.
These formulas con be easily generalized to the multi-dimensional case, so that the cost of computing $F$ and $F^2$ is given by: $PreComp = O(2^d \cdot n^d)$.
2) As to term $C^U$, at each iteration of the algorithm the first element of the priority queue is extracted and two new elements are inserted. The cost of either top-extraction and insertion is logarithmic w.r.t. the size of the queue, which is in turn bounded by the number of buckets of the output histogram. On the other hand, the number of iterations of Greedy Algorithm is equal to the number of buckets it produces. Thus, if we denote as $\beta$ the number of buckets of the histogram produced by the greedy algorithm, the overall cost $C^U$ of the updates to the priority queue is $O(\beta \cdot log(\beta))$.
3) We denote the cost of computing the function *Evaluate* on the block $b$ w.r.t. the greedy criterion $G$ as $C(Evaluate(G, b))$; moreover, we denote the binary histogram produced by Greedy Algorithm as $H$. Thus, the term $C^E$ is given by $\sum_{b \in Nodes(H)} C(Evaluate(G, b))$.
As shown in Sect. 4.3.2, the $SSE$ of a block and the reduction of $SSE$ due to a split can be evaluated accessing $2^d$ elements of $F$ and $2^d$ elements of $F^2$, instead of accessing all the elements of the block (see (4.3) and (4.1)). Clearly, also the reduction of $SSE(marg_{dim}(b))$ due to the split of $b$ along any point on $dim$ can be computed in $O(2^d)$, as it can be derived from the reduction of $SSE(b)$ due to the same split (see formula (4.2)). On the contrary, evaluating $SSE(marg_{dim}(b))$ requires the computation and scanning of the marginal distribution of $b$ along $dim$, which, using the array of partial sums $F$, can be done in $O(2^d \cdot n)$. Therefore, for all the proposed greedy criteria $G$ but *Max-Var*$^{marg}$/ *Max-Red*$^{marg}$, $C(Evaluate(G, b)) = O(2^d \cdot \eta)$, where $\eta$ is the number of reductions of SSE or marginal SSE which have to be computed. In particular, $\eta = d \cdot n$ for *FBH* and *HBH*, whereas $\eta = d \cdot k$ for k-*GHBH*.
In the case that *Max-Var*$^{marg}$/ *Max-Red*$^{marg}$ is adopted, the cost of computing the $d$ marginal SSEs of the block is $O(2^d \cdot d \cdot n)$ for either *FBH*, *HBH* and k-*GHBH*, and dominates the cost of computing the reductions of marginal SSE.
To sum up, when pre-computation is adopted, $C(Evaluate(G, b)) = O(2^d \cdot d \cdot n)$ for *FBH* and *HBH*, and $C(Evaluate(G, b)) = O(2^d \cdot d \cdot \alpha)$ for k-*GHBH* (where $\alpha = k$ for all the greedy criteria $G$ but *Max-Var*$^{marg}$/ *Max-Red*$^{marg}$, for which $\alpha = n$).
Therefore, $C^E$ is given by:

1. in the case that $T$=*FBH* or *HBH*, $C^E = \sum_{b \in Nodes(H)} C(Evaluate(G, b)) = O(\beta_T^{max} \cdot 2^d \cdot d \cdot n)$
2. in the case that $T$=k-*GHBH*, $C^E = O(\beta_{GHBH}^{max} \cdot 2^d \cdot d \cdot \alpha)$

Observe that in the case of *FBH* and *HBH* under any greedy criterion, as well as in the case of k-*GHBH* under *Max-Var*$^{marg}$/ *Max-Red*$^{marg}$, term $C^U$ is negligible w.r.t. $C^E$. In fact, the number of buckets $\beta$ is not greater than $n^d$, which implies $\log \beta \leq d \cdot \log n < 2^d \cdot d \cdot n$. In the case of k-*GHBH*, when a criterion different from *Max-Var*$^{marg}$/ *Max-Red*$^{marg}$ is adopted, it can happen that the inequalities $\log \beta < 2^d \cdot d \cdot k$ do not hold, even though, in practical cases, the number of buckets rarely exceeds the value $2^{2^d \cdot d \cdot k}$. □

# B

# Algorithms

## B.1 An Algorithm for Populating a Quad-Tree Window

**Algorithm 1**

***Function*** InsertIntoWindow

*INPUT: a sensor reading* $x = \langle id_s, v, ts \rangle$*;*
*a quad-tree window* $QTW(W_k)_{old}$*, where* $ts \in [(k-1) \cdot T..k \cdot T]$.
*OUTPUT: a quad-tree window* $Q_{new}$

***begin***

$Q_{new} := QTW(W_k)_{old}$*;*
$j := \lceil (ts - (k-1)T)/u \rceil$*;* // $ts$ is contained into the $j$-th time interval
// inside $W_k$
$Point := \langle id_s, j \rangle$*;*
$CurrNode := Root(Q_{new})$*;*
$UpdateSum(CurrNode, CurrNode.sum + v)$*;*
***while****(*$size(CurrNode.range) > 1$*)*
***if*** *(*$CurrNode$ *is a leaf)* $Q_{new} := Split(Q_{new}, CurrNode)$*;*
*Let* $i \in 1..4$ *be such that* $Point$ *is inside* $Child(CurrNode, i).range$*;*
$CurrNode := Child(CurrNode, i)$*;*
$UpdateSum(CurrNode, CurrNode.sum + v)$*;*
***end while****;*

***return*** $Q_{new}$*;*
***end.***

wherein: 1) the function $Split(QTW, m)$ adds four children (corresponding to four null quadrants) to the leaf node $m$ of the quad-tree window $QTW$, and 2) the function $UpdateSum(m, v)$ assigns the value $v$ to the sum associated to the node $m$.

## B.2 An Algorithm for Constructing Binary Tree Indices

**Algorithm 2**

***Function*** Insert

*INPUT: a sensor reading* $x = \langle id_s, v, ts \rangle$*;*
*a binary tree index* $BTI(C_l)_{old}$*, where* $ts \in \Delta T(C_l)$*.*
*OUTPUT: a binary tree index* $BTI(C_l)_{new}$

***begin***

$Ind := BTI(C_l)_{old}$*;*
$CurrNode := Root(Ind)$*;*
***while***$(size(CurrNode.interval) > 2 \cdot T)$
$UpdateSum(CurrNode, CurrNode.sum + v)$*;*
***if*** *(*$CurrNode$ *is a leaf)*
$Ind := Split(Ind, CurrNode)$*;*
*Let* $i \in 1..2$ *be such that* $ts(x)$ *belongs to* $Child(CurrNode, i).interval$*;*
$CurrNode := Child(CurrNode, i)$*;*
***end while***
***if*** $(CurrNode.sum = 0)$
*Create two empty quad-tree windows corresponding to the two halves of* $CurrNode.interval$*;*
***end if***
$UpdateSum(CurrNode, CurrNode.sum + v)$*;*
*Let* $QTW_j$ *be the quad-tree window referred by* $CurrNode$ *whose time window contains* $ts(x)$*;*
$InsertIntoWindow(QTW_j, x)$*;*

***return*** $Ind$*;*
***end.***

wherein: 1) the function $Split(BTI, m)$ adds two children (corresponding to two null halves) to the leaf node $m$ of the binary tree $BTI$, and 2) the function $UpdateSum(m, v)$ assigns the value $v$ to the sum associated to the node $m$ of $BTI$.

## B.3 An Algorithm for Compressing a Multi-Resolution Data Stream Summary

**Algorithm 3**

***Function*** ReleaseStorageSpace

*INPUT: a list L of Binary Tree Indices* $BTI(C_1)$, $BTI(C_2)$, ..., $BTI(C_k)$*;*
*the amount* $S_{req}$ *of storage space to be released.*
*OUTPUT: a new list* $L'$ *of Binary Tree Indices.*

***begin***

$L' := L$;
$S_{rel} := 0$; *the storage space which has been actually released*
***while***$(S_{rel} < S_{req})$
$S_{rel} := S_{rel} + CompressBTI(Root(OldestBTI(L')), S_{req} - S_{rel})$;
***if*** $(S_{rel} < S_{req})$
$S_{rel} := S_{rel} + Space(OldestBTI(L'))$;
*Remove* $OldestBTI(L')$ *from* $L'$;
***end if***
***end while***;
***return*** $L'$;

***end.***

wherein: 1) $Space(Y)$ returns the amount of storage space occupied by the binary tree index $Y$; 2) $OldestBTI(L)$ returns the first (i.e. the oldest) binary tree index of the list $L$; 3) $CompressBTI(N, S)$ compresses the binary tree index whose root is $N$ till either $S$ has been released, or the binary tree index is no longer compressible. After compressing the $BTI$, the function $CompressBTI(N, S)$ returns the amount of storage space which has been actually released.

## B.4 An Algorithm for Compressing Binary Tree Indices

**Algorithm 4**

***Function*** CompressBTI

*INPUT: A node $N$ of a binary tree index;*
*the amount $S_{req}$ of storage space to be released.*
*OUTPUT: the amount of storage space actually released.*

***begin***

$S_{rel} := 0$; // *the storage space which has been actually released*
***if*** *($N$ is not a leaf)*
*i:=1;*
***while*** *($S_{rel} < S_{req}$ and $i \leq 2$)*
$S_{rel} := S_{rel} + CompressBTI(Child(N, i), S_{req} - S_{rel})$;
$i := i + 1$;
***end while***
***if*** *($S_{rel} < S_{req}$)*
***for each*** $i = 1, .., 2$
$S_{rel} := S_{rel} + Space(Child(N, i))$;
*delete the node* $Child(N, i)$;
***end if***
***else***
***if*** *$N$ refers to 2 quad-tree windows* $QTW_1, QTW_2$
*i:=1;*
***while*** *($S_{rel} < S_{req}$ and $i \leq 2$)*
$S_{rel} := S_{rel} + CompressQTW(QTW_i, S_{req} - S_{rel})$;
$i := i + 1$;
***end while***
***if*** *($S_{rel} < S_{req}$)*
***for each*** $i = 1, .., 2$
$S_{rel} := S_{rel} + Space(QTW_i)$;
*delete* $QTW_i$;
***end if***
***return*** $S_{rel}$;

***end.***

wherein: 1) $CompressQTW(QTW_i, S)$ compresses the quad-tree window $QTW_i$ till either the amount $S$ of storage space has been released, or $QTW_i$ is no longer compressible. The function returns the amount of storage space actually released; 2) $Space(N)$ returns the space consumed by a binary-tree node $N$ in the $BTI$ representation described in Sect. 6.4.2; 3) $Space(QTW_i)$ returns the space occupied by the quad-tree window $QTW_i$.

## B.5 Algorithms for Estimating Range Queries on a MRDS

**Algorithm 5**

***Function*** EstimateSumQuery

*INPUT: a compressed Multi-Resolution Data Stream Summary $MRDS$;*
*a sum range query $Q = \langle s_i..s_j, [t_{start}..t_{end}]\rangle$.*
*OUTPUT: an estimate of the query answer.*

***begin***

$\langle BTI_{start}, BTI_{end}\rangle = BTIBinarySearch(MRDS, [t_{start}, t_{end}])$;
$BTI_{curr} := BTI_{start}$;
$Sum := 0$;
$QTW'_{start} := null$;
***while*** *($QTW'_{start}$ = null and $BTI_{curr}$ precedes or coincides with $BTI_{end}$ )*
$\langle \Delta S, QTW'_{start}\rangle := EstimateAndLocate(Q, Root(BTI_{curr}))$;
$Sum := Sum + \Delta S$;
*Assign the BTI following the current one to $BTI_{curr}$;*
***end while***
***if*** *($QTW'_{start} \neq null$ )*
$QTW_{end} := Search(MRDS, t_{end})$;
***for each*** *quad-tree window $QTW_i$ from $QTW_{start}$ to $QTW_{end}$*
$Sum := Sum + Estimate(Q, QTW_i)$;
***end if***
***return*** *Sum;*

***end***

where :

- $Estimate(Q, QTW_i)$ evaluates the contribution of the quad-tree window $QTW_i$ to the query $Q$, and will be described in more detail in Sect. 6.6.1;
- $Search(MRDS, t_{end})$ searches the most recent QTW stored in MRDS whose time window starts before $t_{end}$

**Algorithm 6**

***Function*** EstimateAndLocate

*INPUT: A query* $Q = \langle s_i..s_j, [t_{start}..t_{end}]\rangle$*;*
*A BTI node* $N$*;*
*OUTPUT: a pair* $\langle \Delta S, QTW'\rangle$*, where* $QTW'$ *is the first QTW which has not been removed and is referred by the sub-tree rooted in* $N$*, and* $\Delta S$ *is the contribution to the query result of the descending nodes of* $N$ *which are older than* $QTW'$*.*

***begin***

$QTW' := null$;
***if*** *(*$N.interval$ *is external to* $[t_{start}..t_{end}]$*)*
***return*** $\langle 0, null\rangle$*;*
***if*** *(*$N.interval$ *is completely contained into* $[t_{start}..t_{end}]$ *and* $s_i..s_j = s_1..s_n$*)*
***return*** $\langle N.sum, null\rangle$*;*
*// otherwise,* $N.interval$ *overlaps* $[t_{start}..t_{end}]$ *partially, or* $s_i..s_j$ *is strictly*
*// contained into* $s_1..s_n$
***if*** *(*$N$ *is a leaf node not equipped with any QTW)*
***return*** $\langle linear\_interp(N, Q), null\rangle$*;*
***if*** *(*$N$ *is a leaf node referring to 2 QTWs)*
*Assign the first of the referred QTWs whose time window overlaps* $[t_{start}..t_{end}]$ *to* $QTW'$*;*
***return*** $\langle 0, QTW'\rangle$*;*
***end if***
*// otherwise,* $N$ *is an internal node*
$i := 1$*;*
$S := 0$*;*
***while*** *(*$i \leq 2$ *and* $QTW' = null$*)*
$\langle \Delta S, QTW'\rangle := EstimateAndLocate(Q, Child(N, i))$*;*
$S := S + \Delta S$*;*
$i := i + 1$;
***end while***
***return*** $\langle S, QTW'\rangle$*;*

***end***

where $linear\_interp(N, Q)$ evaluates the contribution of $N$ to the estimate of $Q$ by performing linear interpolation, as explained above.

# References


1. Abiteboul S, Hull R, Vianu V (1994) *Foundations of Databases*, Addison-Wesley, Reading, Massachusetts
2. Aboulnaga A, Chaudhuri S (1999) Self-tuning histograms: building histograms without looking at data. *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
3. Acharia S, Gibbons P B, Poosala V, Ramaswamy S (1999) The Aqua Approximate Query Answering System. *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
4. Acharya S, Gibbons P B, Poosala V, Ramaswamy S (1999) Join Synopses for Approximate Query Answering. *Proc. 1999 ACM SIGMOD International Conference On Management Of Data*, Philadelphia, Pennsylvania
5. Acharya S, Poosala V, Ramaswamy S (1999) Selectivity estimation in spatial databases. *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
6. Ankerst M, Bruenig M M, Kriegel H P, Sander J (1999) OPTICS: Ordering Points To Identify the Clustering Structure. *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
7. Avnur R, Hellerstein J M (2000) Eddies: Continuously Adaptive Query Processing. *Proc. 2000 ACM SIGMOD Int. Conference on Management of Data*, Dallas, Texas
8. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and Issues in Data Stream Systems. *Proc. 21st Symposium on Principles of Database Systems – PODS*, Madison, Wisconsin
9. Barbara D, DuMouchel W, Faloutsos C, Haas P J, Hellerstein J M, Ioannidis Y, Jagadish H V, Johnson T, Ng R, Poosala V, Ross K A, Sevcik K C (1997) The New Jersey data reduction report. *Bulletin of the Technical Committee on Data Engineering* 20(4): 3–45
10. Buccafurri F, Furfaro F, Lax G, Saccà D (2002) Binary-tree histograms with tree indices. *Proc. 13th Int. Conference on Database and Expert Systems Applications – DEXA*, Aix en Provence, France
11. Buccafurri F, Furfaro F, Saccà D (2001) Estimating Range Queries using Aggregate Data with Integrity Constraints: a Probabilistic Approach. *Proc. 8th Int. Conference on Database Theory –ICDT*, London, UK

12. Buccafurri F, Furfaro F, Saccà D, Sirangelo C (2003) A Quad-Tree Based Multiresolution Approach for Two-dimensional Summary Data. *Proc. 15th Int. Conference on Scientific and Statistical Database Management – SSDBM*, Cambridge, Massachusetts
13. Buccafurri F, Pontieri L, Rosaci D, Saccà D, (2002) Improving Range Query Estimation on Histograms. *Proc. 18th Int. Conference on Data Engineering –ICDE*, San Josè, California
14. Buccafurri F, Rosaci D, Saccà D (1999) Compressed datacubes for fast OLAP applications. *Proc. 1st Int. Conference on Data Warehousing and Knowledge Discovery – DaWak*, Florence, Italy
15. Bruno N, Chaudhuri S, Gravano L (2001) STHoles: a multi-dimensional workload aware histogram. *Proc. 2001 ACM SIGMOD Int. Conference on Management of Data*, Santa Barbara, California
16. Chaudhuri S (1998) An Overview of Query Optimization in Relational Systems. *Proc. 17th Symposium on Principles of Database Systems – PODS*, Seattle, Washington
17. Chaudhuri S, Dayal U (1997) An Overview of Data Warehousing and OLAP Technology, *ACM SIGMOD Record* 26(1): 65–74
18. Chakrabarti K, Garofalakis M, Rastogi R, Shim K (2000) Approximate query processing using wavelets. *Proc. 26th Int. Conference on Very Large Databases – VLDB*, New York City, NY
19. Chen J, DeWitt D J, Tian F, Wang Y (2000) A scalable continuous query system for internet databases. *Proc. 2000 ACM SIGMOD Int. Conference on Management of Data*, Dallas, Texas
20. Christodoulakis S (1981) Estimating Selectivities in Databases. *PhD Thesis, CSRG Report N. 136*, University of Toronto, Canada
21. Christodoulakis S (1984) Implications of Certain Assumptions in Data Base Perfomance Evaluations. *ACM Transactions on Database Systems*
22. Cortes C, Fisher K, Pregibon D, Rogers A, Smith F (2000) Hancock: A Language for Extracting Signatures from Data Streams. *Proc. 6th Int. Conference on Knowledge Discovery and Data Mining – KDD*, Boston, Massachusetts
23. Cuzzocrea A, Furfaro F, Masciari E, Sirangelo C (2003) Approximate Query Answering on Sensor Network Data Streams. *Proc. 1st Geo Sensor Networks Workshop – GSN*, Portland, Maine
24. Deshpande A, Garofalakis M, Rastogi R (2001) Independence is good: dependency-based histogram synopses for high-dimensional data. *Proc. 2001 ACM SIGMOD Int. Conference on Management of Data*, Santa Barbara, California
25. Dobra A (2005) Histograms Revisited: When are histograms the best approximation method for aggregates over joins?. *Proc. 24th ACM Symposium on Principles of Database Systems- PODS*, Baltimore, Maryland
26. Donjerkovic D, Ioannidis Y E, Ramakrishnan R (2000) Dynamic Histograms: Capturing Evolving Data Sets. *Proc. 16th International Conference on Data Engineering – ICDE*, San Diego, California
27. Ester M, Kriegel H P, Sander J, Xu X (1996) A density-based algorithm for discorvering clusters in large spatial databases with noise. *Proc. 2nd Int. Conference on Knowledge Discovery and Data Mining – KDD*, Menlo Park, California

28. Faloutsos C, Jagadish H V, Sidiripoulos N D (1997) Recovering Information from Summary Data. *Proc. 23rd Int Conference on Very Large Data Bases – VLDB*, Athens, Greece
29. Furfaro F, Mazzeo G M, Sacc D, Sirangelo C (2005) Hierarchical Binary Histograms for Summarizing MultiDimensional Data. *Proc. 20th ACM Symposium on Applied Computing – SAC*, Santa Fe, New Mexico
30. Furfaro F, Mazzeo G M, Sirangelo C (2005) Clustering-based histograms for multi-dimensional data. *Proc. 7th International Conference on Data Warehousing and Knowledge Discovery – DaWak* , Copenhagen, Denmark
31. Garofalakis M, Gibbons P B (2002) Wavelet Synopses with Error Guarantees. *Proc. 2002 ACM SIGMOD Int. Conference on Management of Data*, Madison, Wisconsin
32. Garofalakis M, Gibbons P B (2004) Probabilistic Wavelet Synopses. *ACM Transactions on Database Systems – TODS* 29(1):43–90
33. Garofalakis M, Gibbons P B (2004) Deterministic Wavelet Thresholding for Maximum-Error Metrics. *Proc. 23rd Symposium on Principles of Database Systems – PODS*, Paris, France
34. Gibbons P B, Matias Y (1998) New sampling-based summary statistics for improving approximate query answers. *Proc. ACM SIGMOD International Conference on Management of Data*, Seattle, Washington
35. Gibbons P B, Matias Y, Poosala V (1997) Fast incremental maintenance of approximate histograms. *Proc. 23rd Int. Conference on Very Large Data Bases – VLDB*, Athens, Greece
36. Gilbert A C, Kotidis Y, Muthukrishnan S, Strauss M (2001) Optimal and Approximate Computation of Summary Statistics for Range Aggregates. *Proc. 20th Int. Symposium on Principles of Database Systems – PODS*, Santa Barbara, California
37. Gray J, Bosworth A, Layman A, Pirahesh H (1997) Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. *Journal of Data Mining and Knowledge Discovery*, 1(1):29–53
38. Grumbach S, Rafanelli M, Tininini L (1999) Querying Aggregate Data. *Proc. 18th ACM Symposium on Principles of Database Systems – PODS*, Philadephia, Pennsylvania
39. Guha S, Indyk P, Muthukrishnan S, Strauss M (2002) Histogramming Data Streams with Fast Per-Item Processing. *Proc. 29th Int. Colloquium on Automata, Languages and Programming – ICALP*, Malaga, Spain
40. Guha S, Koudas N, Shim K (2001) Data-streams and histograms. *Proc. 33rd ACM Symposium on Theory of Computing – STOC*, Crete, Greece
41. Guha S, Rastogi R, Shim K (1998) CURE: An efficient clustering algorithm for Large Databases. *Proc. 1998 ACM-SIGMOD Int. Conference on Management of Data*, Seattle, Washington
42. Guha S, Shim K, Woo J (2004) REHIST: Relative Error Histogram Construction Algorithms. *Proc. 30th Int. Conference on Very Large Databases -VLDB*, Toronto, Canada
43. Gunopulos D, Kollios G, Tsotras V J, Domeniconi C (2000) Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes, *Proc. of ACM SIGMOD Conference on Management of Data*, Dallas, Texas
44. Gunopulos D, Kollios G, Tsotras V J, Domeniconi C (2005) Selectivity estimators for multidimensional range queries over real attributes. *The VLDB Journal* 14(2):137–154

45. Haas P J (1997) Large-Sample and Deterministic Confidence Intervals for Online Aggregation. *Proc. 9th Int. Conference on Statistical and Scientific Database Management – SSDBM*, Olympia, Washington
46. Harinarayan V, Rajaraman A, Ullman J D (1996) Implementing Data Cubes Efficiently. *Proc. 1996 ACM SIGMOD International Conference on Management of Data*, Montreal, Canada
47. Hellerstein J M, Haas P J, Wang H J (1997) Online Aggregation. *Proc. 1997 ACM SIGMOD International Conference on Management of Data*, Tucson, Arizona
48. Henzinger M R, Raghavan P, Rajagopalan S (1998) Computing on data streams. *Technical report 1998-011*, Digital Systems Research Center, Palo Alto, California
49. Ioannidis Y E (1993) Universality of Serial Histograms. *Proc. 19th Int. Conference on Very Large Data Bases – VLDB*, Dublin, Ireland
50. Ioannidis Y E, Poosala V (1995) Balancing histogram optimality and practicality for query result size estimation. *Proc. 1995 ACM SIGMOD Int. Conference on Management of Data*, San Josè, California
51. Jagadish H V, Jin H, Ooi B C, Tan K-L (2001) Global optimization of histograms. *Proc. ACM SIGMOD 2001 Int. Conference on Management of Data*, Santa Barbara, California
52. Jagadish H V, Koudas N, Muthukrishnan S, Poosala V, Sevcik K C, Suel T (1998) Optimal histograms with quality guarantees. *Proc. 24th Int. Conference on Very Large Databases – VLDB*, New York City, NY
53. Jawerth B, Sweldens W (1994) An Overview of Wavelet Based Multiresolution Analyses. *SIAM Review* 36(3):377–412
54. Kaufman L, Rousseeuw P J (2005) *Finding Groups in Data: An Introduction to Cluster Analysis*. Wiley-Interscience, New York
55. Kooi R P (1980) The optimization of queries in relational databases. PhD thesis, Case Western Reserve University, Cleveland, Ohio
56. Korn F, Johnson T, Jagadish H V (1999) Range Selectivity Estimation for Continuous Attributes. *Proc. 11th Int. Conference on Scientific and Statistical Database Management – SSDBM*, Cleveland, Ohio
57. Koudas N, Muthukrishnan S, Srivastava D (2000) Optimal Histograms for Hierarchical Range Queries. *Proc. 19th Symposium on Principles of Database Systems - PODS*, Dallas, Texas
58. Lazaridis I, Mehrotra S (2001) Progressive Approximate Aggregate Queries with a MultiResolution Tree Structure. *Proc. 2001 ACM SIGMOD Int. Conference on Management of Data*, Santa Barbara, California
59. MacQueen J (1967) Some Methods for Classification and Analysis of Multivariate Observations. *Proc. 5th Berkeley Symposium on Mathematics, Statistics and Probability*
60. Madden S, Franklin M J (2002) Fjording the stream: An architecture for queries over streaming sensor data. *Proc. 18th Int. Conference on Data Engineering – ICDE*, San Jose, California
61. Malvestuto F (1993) A Universal-Scheme Approach to Statistical Databases Containing Homogeneous Summary Tables. *ACM TODS* 18(4): 678–708
62. Mamoulis N, Papadias D (2001) Selectivity Estimation Of Complex Spatial Queries. *Proc. 7th Int. Symposium on Advances in Spatial and Temporal Databases – SSTD*, Redondo Beach, California

63. Martin G R, Packwood R A, Rhee I (1996) Variable size block matching motion estimation with minimal error. *SPIE Proc. of Digital Video Compression: Algorithms and Technologies*, San Josè, California
64. Matias Y, Vitter J S, Wang M (1998) Wavelet-based histograms for selectivity estimation, *Proc. 1998 ACM SIGMOD Int. Conference on Management of Data*, Seattle, Washington
65. Muralikrishna M, DeWitt D J (1988) Equi-Depth Histograms For Estimating Selectivity Factors For Multi-Dimensional Queries. *Proc. 1988 ACM SIGMOD Int. Conference on Management of Data*, Chicago, Illinois
66. Muthukrishnan S, Poosala V, Suel T (1999) On Rectangular Partitioning in Two Dimensions: Algorithms, Complexity and Applications. *Proc. 7th Int. Conference on Database Theory –ICDT*, Jerusalem, Israel
67. Natsev A, Rastogi R, Shim K (1999) WALRUS: A Similarity Retrieval Algorithm for Image Databases, *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
68. Ng R T, Han J (1994) Efficient and effective clustering methods for spatial data mining. *Proc. 20th International Conference on Very Large Databases*, Santiago, Chile
69. Poosala V (1997) Histogram-based Estimation Techniques in Database Systems. PhD dissertation, University of Wisconsin-Madison
70. Poosala V, Ganti V (1999) Fast Approximate Answers to Aggregate Queries on a Datacube. *Proc. 11th Int. Conference on Scientific and Statistical Database Management – SSDBM*, Cleaveland, Ohio
71. Poosala V, Ganti V, Ioannidis Y E (1999) Approximate Query Answering using Histograms. *IEEE Data Engineering Bulletin, Vol. 22*
72. Poosala V, Ioannidis Y E, Haas P J, Shekita E J (1996) Improved histograms for selectivity estimation of range predicates. *Proc. 1996 ACM SIGMOD Int. Conference on Management of Data*, Montreal, Canada
73. Poosala V, Ioannidis Y E (1997) Selectivity estimation without the attribute value independence assumption. *Proc. 23rd Int. Conference on Very Large Databases – VLDB*, Athens, Greece
74. Selinger P G, Astrahan M M, Chamberlin D D, Lorie R A, Price T G (1979) Access path selection in a relational database management system. *Proc. 1979 ACM SIGMOD Int. Conference on Management of Data*, Boston, Massachusetts
75. Shanmugasundaram J, Fayyad U, Bradley P S (1999) Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. *Proc. 5th Int. Conference on Knowledge Discovery and Data Mining – KDD*, San Diego, California
76. Sitzmann I, Stuckey P J (2000) Improving Temporal Joins Using Histograms. *Proc. 11th Int. Conference on Database and Expert Systems Applications – DEXA*, London, UK
77. Stollnitz E J, DeRose T D, Salesin D H (1996) Wavelets for Computer Graphics - Theory and Applications. Morgan Kaufmann Publishers, San Francisco, California
78. Thaper N, Guha S, Indyk P, Koudas N (2002) Dynamic Multidimensional Histograms. *Proc. 2002 ACM SIGMOD Conference on Management of Data*, Madison, Wisconsin

79. Theodoridis Y, Papadias D, Stefanakis E, Sellis T (1998) Direction relations and Two-Dimensional Range Queries: Optimisation Techniques, *Data Knowledge Engineering – DKE*, 27(3):313–336
80. Vitter J S, Wang M, Iyer B (1998) Data Cube Approximation and Histograms via Wavelets. *Proc. 7th Int. Conference on Information and Knowledge Management – CIKM*, Bethesda, Maryland
81. Vitter J S, Wang M (1999) Approximate Computation of Multidimensional Aggregates of Sparse Data using Wavelets. *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
82. Zhang D, Gunopulos D, Tsotras V J, Seeger B (2002) Temporal aggregation over data streams using multiple granularities. *Proc. 8th Int. Conference on Extending Database Technology – EDBT*, Prague, Czech Republic
83. Zhang T, Ramakrishnan R, Livny M (1999) BIRCH: An Efficient Data Clustering Method for Very Large Databases, *Proc. 1999 ACM SIGMOD Int. Conference on Management of Data*, Philadelphia, Pennsylvania
84. Ziegelmann M (2001) Constrained Shortest Paths and related problems. PhD dissertation, *Universität des Saarlandes*
85. Zipf G K (1949) Human behaviour and the principle of least effort. Addison-Wesley, Reading, Massachusetts