A Quad-Tree Based Multiresolution Approach for Two-dimensional Summary Data *

Francesco Buccafurri
DIMET Dept.,
University of Reggio Calabria
Feo di Vito, 89060 Reggio Cal., Italy
bucca@ing.unirc.it

Filippo Furfaro
DEIS Dept.,
University of Calabria
via P. Bucci, 87030 Rende, Italy
furfaro@si.deis.unical.it

Domenico Saccà
DEIS - University of Calabria
ICAR - CNR
via P. Bucci, 87030 Rende, Italy
sacca@icar.cnr.it

Cristina Sirangelo
DEIS Dept.,
University of Calabria
via P. Bucci, 87030 Rende, Italy
sirangelo@si.deis.unical.it

Abstract

In many application contexts, like statistical databases, scientific databases, query optimizers, OLAP, and so on, data are often summarized into synopses of aggregate values. Summarization has the great advantage of saving space, but querying aggregate data rather than the original ones introduces estimation errors which cannot be in general avoided, as summarization is a lossy compression. A central problem in designing summarization techniques is to retain a certain degree of accuracy in reconstructing query answers. In this paper we restrict our attention to two-dimensional data, which are relevant for a number of applications, and propose a hierarchical summarization technique which is combined with the use of indices, *i.e. compact structures providing an approximate description of portions of the original data. Experimental results show that the technique gives approximation errors much smaller than other "general purpose" techniques, such as wavelets and various types of multi-dimensional histogram.*

1 Introduction

There are several application scenarios where the main goal is to extract summary information from available data, rather than inquiring single data. For instance, transaction recording systems, OLAP applications, data mining activities, intrusion detection systems, scientific databases, usually operate on a huge amount of data, but do not return detailed pieces of information: they are mainly interested in aggregating data within a specified range of the domain. These kinds of aggregate query are called *range queries*.

In the above contexts, a widely accepted solution to the problem of efficiently extracting useful knowledge from the available data is to summarize information into a compressed structure, and issue range queries on the summarized data (rather than the original one), in order to get fast (but, in general, approximate) answers. The most significant example of such an approach is represented by histograms [6, 13]. Histograms, initially designed in the context of query optimization for query size estimation, can be also effectively used to estimate range queries in on-line analytical processing [12]. A histogram is obtained by partitioning the frequency distribution (which is generally represented as a multidimensional array) into a set of blocks (called *buckets*), and storing, for each block, the sum of the frequencies inside it. The answer to a sum range query evaluated on the histogram is computed by summing the contributions of each bucket, i.e. by estimating which portion of the sum associated to each bucket lies onto the range of the query. This estimate is evaluated by performing linear interpolation, i.e. by assuming that data distribution inside each bucket is uniform (*Continuous Values Assumption - CVA*), and thus the contribution of the buckets which partially overlap the range of the query is generally approximate (unless the original distribution of frequencies inside these buckets is actually uniform).

Histograms, first proposed in the context of monodimensional data, can be extended to the multi-dimensional case, but their performances, in terms of accuracy, are rather

*This work was partially supported by the National Research Council project "SP1: Reti Internet, efficienza, integrazione e sicurezza"

poor. Better results in the multi-dimensional case are given by other approaches, such as wavelet based ones [4, 15, 16]. Yet, also in the latter approaches, accuracy is far from being satisfactory.

Rather searching for a general method which scales up for any dimension of data, we expect that, in specific application domains, higher accuracy can be achieved by designing *ad-hoc* solutions. In this paper we follow this direction and consider specifically two-dimensional data. This is not a severe restriction, as the need to estimate range queries over these distributions arises in a number of applications:

1. *selectivity estimation on spatial databases* [1, 8]: this problem consists of 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* [14]: 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 time-series databases*: data generated by multiple 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 it consists of continuous and "infinite" readings.

Our approach is closely related to histograms which are suitably extended to the two-dimensional case. In the same way as for mono-dimensional histograms, the data distribution is partitioned into blocks, but, differently, the adopted partition schema is hierarchical, i.e. it consists of progressively splitting blocks produced by previous splits.

Main Contributions

1. *Formal definition of the problem of summarizing two-dimensional data arrays.* We present a quad-tree based partition schema and introduce the notion of *Quad-Tree Summary* (QTS) (the summary structure obtained applying our partition schema on a given data distribution). We define a 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 (called *V-Optimal Quad-Tree Summary*) w.r.t. this metric.
2. *Analysis of the optimal solution and proposal of a greedy algorithm.* We present a polynomial time solution for finding an optimal partition. As the resulting cost function is $O(B \cdot n^2 \cdot log\, n)$, and n is in general very high, we cannot effort such a cost. Therefore we present a greedy algorithm with cost $O(B \cdot logB)$ (where B is the available storage space for the summary structure), so that it can be effectively computed also for very large two-dimensional data (B is much smaller than n^2).
3. *Enhancing the estimation accuracy of the greedy algorithm by introducing indices.* 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 inside each block by replacing linear interpolation with a a more accurate technique based on a compact structure (called *index*) designed specifically for two-dimensional data and containing an approximate description of the original data distribution inside the block. The experiments we have carried out over a large number of syntectic two-dimensional data arrays show that the greedy algorithm with the indices have much better performances than state of the art "general purpose" approaches.

2 Summarizing two-dimensional data: the problem

In this section we present our quad-tree based partition schema for summarizing two-dimensional data arrays, and introduce the notion of *Quad-Tree Summary* (QTS) (the summary structure obtained applying our partition schema on a given data distribution). We define a 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 (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 compressed 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 of finding partitions whose representation can be done compactly. A naive solution consists of 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. Unfortu-

nately, 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.

2.1 Quad-Tree Partition

We are given a two-dimensional data distribution D which can be also viewed as a two-dimensional array of size $d_1 \times d_2$. A *range* σ_i on the i-th dimension of D is an interval $l..u$, such that $1 \leq l \leq u \leq d_i$. Boundaries l and u of r_i are denoted by $lb(\sigma_i)$ (*lower bound*) and $ub(\sigma_i)$ (*upper bound*), respectively.

Given a range σ_i on the dimension i, we denote by $lh(\sigma_i)$ (*left half*) the range $lb(\sigma_i)..\lfloor(lb(\sigma_i) + ub(\sigma_i))/2\rfloor$ on i, and by $rh(\sigma_i)$ (*right half*) the range $\lfloor(lb(\sigma_i) + ub(\sigma_i))/2\rfloor + 1..ub(\sigma_i)$.

A *block* r (*of* D) is a pair $\langle\sigma_1, \sigma_2\rangle$ where σ_i is a range on the dimension i, for each $1 \leq i \leq 2$. σ_1 and σ_2 are said *sides of* r. A pair $\langle v_1, v_2\rangle$ such that v_1 is either $lb(\sigma_1)$ or $ub(\sigma_1)$ and v_2 is either $lb(\sigma_2)$ or $ub(\sigma_2)$ is said a *vertex* of r. Informally, a block represents a "rectangular" region of D. A block r of D containing no non-zero elements is called a *null block*.

Given a block r we denote by $sum(r)$ ($avg(r)$, resp.) the sum (the average, resp.) of the array elements occurring in the block r.

Given two ranges σ_1, σ_2 defining the block $r = \langle\sigma_1, \sigma_2\rangle$, a *quad-split block* of r is any block $\langle\rho_1, \rho_2\rangle$ such that ρ_i is either $lh(\sigma_i)$ or $rh(\sigma_i)$. Observe that, for a given block r of D, there are 4 different quad-split blocks; each of these correspond to one of quadrants of r.

Given a block $r = \langle\sigma_1, \sigma_2\rangle$ of D, we denote by $Q(r)$ the 4-tuple $\langle r_1, r_2, r_3, r_4\rangle$ such that $r_1 = \langle lh(\sigma_1), rh(\sigma_2)\rangle$, $r_2 = \langle rh(\sigma_1), rh(\sigma_2)\rangle$, $r_3 = \langle lh(\sigma_1), lh(\sigma_2)\rangle$, and $r_4 = \langle rh(\sigma_1), lh(\sigma_2)\rangle$. $Q(r)$ is said the *quad-split partition of* r. Often, with a little abuse of notation we refer to $Q(r)$ as a set. Informally, the quad-split partition of r contains the four quadrants of r.

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) $Root(QTP(D)) = \langle 1..d_1, 1..d_2\rangle$, 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 by $Null(P)$ the set $\{p \in Leaves(P) \mid sum(p) = 0\}$. 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 $Nodes(P) \setminus \{Der(P) \bigcup Null(P)\}$.

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.

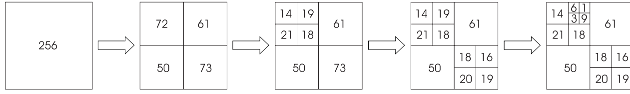
Given a quad-tree summary $QTS = \langle P, S\rangle$ of D, P is said the *partition-tree* of QTS, and we denote it by $Part(QTS)$; S is said the *content set* of QTS and we denote it by $Cont(QTS)$. A node r of P is said a *terminal block* if $r \in Leaves(P)$, a *non-terminal block* otherwise.

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

In Figure 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.

The storage space for a quad-tree summary $QTS = \langle P, S\rangle$ is the space occupied 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 Section 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.

Partitioning a datacube:



Quad-tree partition:

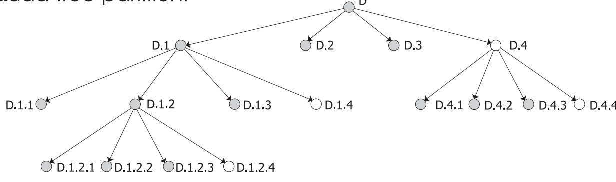


Figure 1. A quad-tree based partition

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 2 reports the strings representing the sums and the structure of the quad-tree of Fig. 1.

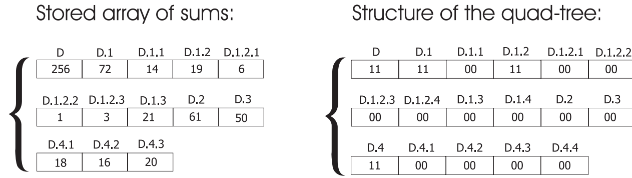


Figure 2. Quad-tree structure encodement

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 paper, we refer to $QTS(D)$ also as *the compressed representation* of the array D.

2.3 Estimating range queries on a QTS

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 starting 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.

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 compressed representation of a data array can be adopted. Following a well-accepted approach in literature, we measure the "goodness" of the compressed representation of a data array by using its SSE. 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: $SSE(q_i) = \sum_{j \in q_i} (D[j] - avg(q_i))^2$, where by $\sum_{j \in q_i}$ we denote that the summation is extended to all the elements of the original array D belonging to the block q_i. Clearly, the smaller $SSE(QTS(D))$, the "better" the representation provided by $QTS(D)$ is, in terms of accuracy.

Definition 1 *Given a two-dimensional data distribution D, we call* V-Optimal Quad-Tree Summary *on D (for a bounded storage space 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 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 $d_1 \times d_2$ 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 [5] for more details).

Theorem 1 *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 $O(B \cdot n^2 \cdot log\, n)$.

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 compressed 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 1 we have that a V-Optimal Quad-Tree Summary can be computed in polynomial time.

Remark. We point out that finding an arbitrary partition (i.e. with no constraints on its structure) minimizing SSE is a NP-Hard problem, as shown in [11]. 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 admissible 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 [9], 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 [11], 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
  $Q := \langle\, P_0, \{\langle\langle 1..d_1, 1..d_2\rangle, sum(\langle 1..d_1, 1..d_2\rangle)\rangle\}\ \rangle$;
  $B := B - 32 - 2$;
  // 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 2 *Given a two-dimensional data array 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 $O(B \cdot logB)$.*

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.

4.1 Indexing two-dimensional data blocks

Experience acquired in [2, 3] 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 Figure 3 where nodes correspond to sub-blocks of the block Q of the figure. The node at level 1

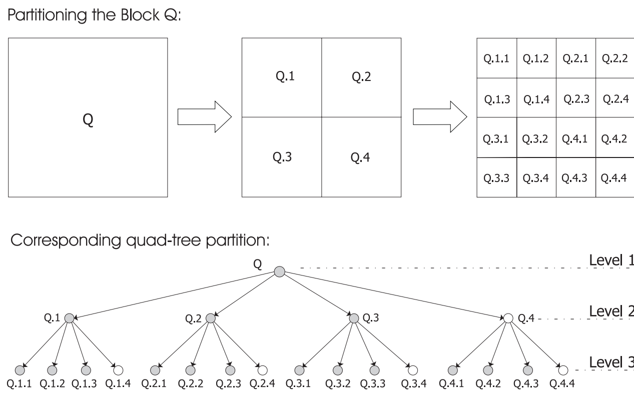


Figure 3. 2/3LT-index

(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 Figure 4.

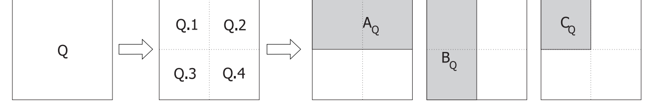


Figure 4. A_Q, B_Q, C_Q regions inside a block

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:

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

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 Figure 5.

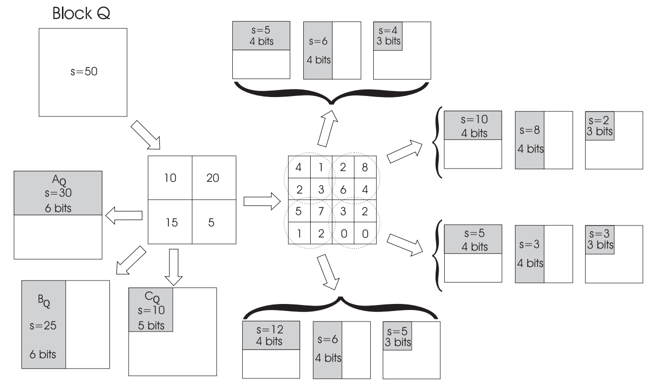


Figure 5. Building a 2/3LT-index

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 repre-

senting 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 Figure 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. 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).

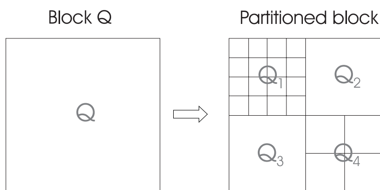


Figure 6. The structure of a 2/4LT-index

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 Section 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[3]. 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:

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

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 Section 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 (1) in Section 4 and $Best(q)$ is defined in Section 4, just after (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 Section 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 := ⟨ P_0, {⟨⟨1..d_1, 1..d_2⟩, sum(⟨1..d_1, 1..d_2⟩)⟩} ⟩;
    B := B − 32 − |good({⟨1..d_1, 1..d_2⟩})| · 64 − 2;
    // 32 bits are spent for the sum of the entire array;
    // |good({⟨1..d_1, 1..d_2⟩})| · 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 ≥ 0)
      Select a node p in Leaves(Q) such that:
        SSE(p) = max_{q∈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)| · 32 − |good(Q(p))| · 64+
            +|good({p})| · 64 − 4 · 2;
      if (B ≥ 0)
        Q := ⟨ Split(Part(Q), p) ,
              Cont(Q) ∪ ⋃_{r∈Q^+(p)}{⟨r, sum(r)⟩} ⟩;
      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 *Given a two-dimensional data array 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*.

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 compressed data. In particular, we compare our technique with the histogram-based technique MHIST proposed in [13], and with the wavelet-based techniques proposed respectively in [15] and [16]. 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.

First, we briefly describe such three techniques; next, we present the test bed used in our experiments.

MHIST (*Multi-dimensional Histogram*). An MHIST histogram is built by a multi-step algorithm which, at each step, chooses the block which is the most in need of partitioning (as explained below), and partitions it along one of its dimensions. The block to be partitioned is chosen as follows. First, the *marginal distributions* along every dimension are computed for each block. In the 2-D case, the marginal distribution of a block $b = \langle x_1..x_m, y_1..y_n \rangle$ along the first dimension is obtained by computing, for each x_i ($i = 1..m$), the value $sum(\langle x_i..x_i, y_1..y_n \rangle)$. The block b to be split is the one which is characterized by a marginal distribution (along any dimension X_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 X_i by putting a boundary between e_j and e_{j+1}. For each block, three values are stored: the sum of its elements and the positions of its front corner (w.r.t. the linear order of the cells) and its far corner. Denoting the amount of available storage space as B, the number of blocks which can be stored is given by: $\lfloor B/3 \rfloor$.

Wavelet-based Compression Techniques. Wavelets are mathematical transformations implementing hierarchical decomposition of functions. They have been originally used in different research and application contexts (like image and signal processing), and recently have been applied to selectivity estimation [10] and to the approximation of OLAP range queries over data cubes [15, 16]. The compressed representation of a data distribution is obtained in two steps. First, a wavelet transformation is applied to the data distribution, and N wavelet coefficients are generated (the value of N depends both on the size of the data and on the particular type of wavelet transform which has been used). Next, among such N coefficients, the $m < N$ most significant ones (i.e. the largest coefficients) are selected. 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 compression technique described in [15] does not apply the wavelet transform directly to the source array of data. First, the partial sum data array is generated, and each of its cell values is replaced with its natural logarithm (it has been shown that the combination of the logarithm transfor-

mation with the approximation technique generally reduces the relative error of the approximation). Then, the above described compression process is applied to such an obtained array.

In [16] a sophisticated wavelet based technique which mainly aims to improve the I/O efficiency of the compact data construction is proposed. The main difference with the approach described above is that it is applied directly on the source data.

In the following, the two wavelet based techniques will be denoted respectively as WAVE1 (working on the partial sum data array) and WAVE2.

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 [16], 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 $\| \; e^{abs}(QS) \; \|$.

5.2 Synthetic Data Sets

The synthetic data sets used in our experiments are similar to those of [16]. 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.

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 compressed representation, the *skew* inside each region, the size of the queries (using query set QS_2), and we consider both dense and sparse data arrays. The storage space is expressed as the number of 32 bits integers which are available for the compressed 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 compressed data array) is depicted in Fig.7 (sparse data) and Fig.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.8, only QTS and $IQTS$ are compared, as the errors produced by the other methods are out of scale.

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.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 val-

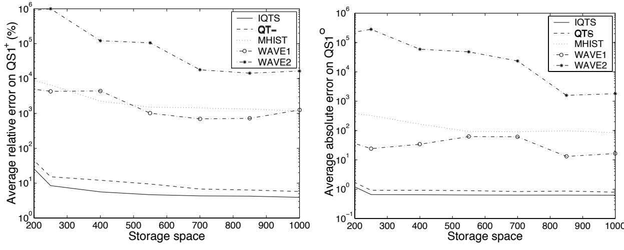


Figure 7. Errors of estimates for sparse data

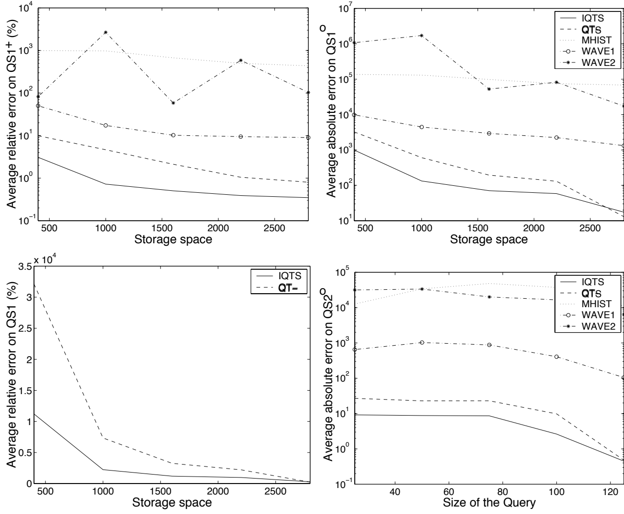


Figure 8. Results for dense data

ues. 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 compressed representations obtained using 1600 4-byte integers. In the picture on the right-hand side of Fig. 9, the value of $\parallel e^{rel}(QS_2^+(\Delta, \Delta)) \parallel$ obtained on sparse data for different values of the query size (i.e. Δ) is reported. In the picture on the bottom-right corner of Fig.8, values of $\parallel e^{rel}(QS_2^+(\Delta, \Delta)) \parallel$ obtained for dense data are shown.

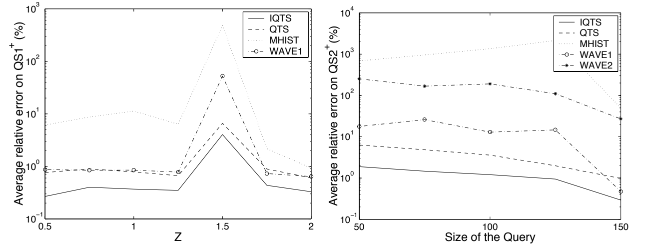


Figure 9. Results for sparse data

References

[1] Acharya, S., Poosala, V., Ramaswamy, S., Selectivity estimation in spatial databases, *Proc. ACM SIGMOD Conf. 1999*, Philadelphia, PA, USA.

[2] Buccafurri, F., Furfaro, F., Lax, G., Saccà, D., Binary-tree histograms with tree indices, *Proc. DEXA 2002*, Aix en Provence, France.

[3] Buccafurri, F., Pontieri, L., Rosaci, D., Saccà, D., Improving Range Query Estimation on Histograms, *Proc. ICDE 2002*, San Josè, CA, USA.

[4] Garofalakis, M., Gibbons, P. B., Wavelet Synopses with Error Guarantees, *Proc. ACM SIGMOD 2002*, Madison, WI, USA

[5] Jagadish, H. V., Koudas, N., Muthukrishnan, S., Poosala, V., Sevcik, K., Suel, T., Optimal histograms with quality guarantees, *Proc. VLDB Conf. 1998*, New York City, USA.

[6] Jagadish, H. V., Jin, H., Ooi, B. C., Tan, K.-L., Global optimization of histograms, *Proc. ACM SIGMOD Conf. 2001*, Santa Barbara, CA, USA.

[7] Lazaridis, I., Mehrotra, S., Progressive Approximate Aggregate Queries with a MultiResolution Tree Structure, *Proc. ACM SIGMOD Conf. 2001*, Santa Barbara, CA, USA.

[8] Mamoulis, N., Papadias, D., Selectivity Estimation Of Complex Spatial Queries, *Proc. SSTD 2001*, Redondo Beach, CA, USA.

[9] Martin, G. R., Packwood, R. A., Rhee, I., Variable size block matching motion estimation with minimal error, *Proc. DVCAT Conf. 1996*, San Josè, CA, USA.

[10] Matias, Y., Vitter, J. S., Wang, M., Wavelet-based histograms for selectivity estimation, *Proc. ACM SIGMOD Conf. 1998*, Seattle, Washington, USA.

[11] Muthukrishnan, S., Poosala, V., Suel, T., On Rectangular Partitioning in Two Dimensions: Algorithms, Complexity and Applications, *Proc. ICDT 1999*, Jerusalem, Israel.

[12] Poosala, V., Ganti, V., Fast Approximate Answers to Aggregate Queries on a Datacube, *Proc. SSDBM Conf. 1999*, Cleaveland, OH, USA.

[13] Poosala, V., Ioannidis, Y. E., Selectivity estimation without the attribute value independence assumption, *Proc. VLDB Conf. 1997*, Athens, Greece.

[14] Theodoridis, Y., Papadias, D., Stefanakis, E., Sellis, T., Direction relations and Two-Dimensional Range Queries: Optimisation Techniques, *DKE*, Vol. 27(3), 1998.

[15] Vitter, J. S., Wang, M., Iyer, B., Data Cube Approximation and Histograms via Wavelets, *Proc. CIKM 1998*, Washington, USA.

[16] Vitter, J. S., Wang, M., Approximate Computation of Multidimensional Aggregates of Sparse Data using Wavelets, *Proc. ACM SIGMOD Conf. 1999*, Philadelphia, PA, USA.