A Quad-Tree Based Multiresolution Approach for Compressing Datacubes

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 Dept. - University of Calabria,
ICAR - CNR
via P. Bucci, 87030 Rende, Italy
sacca@unical.it

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

Abstract

Computing aggregate queries is a performance bottleneck for many OLAP application as datacubes can be extremely large. As quick answers are often necessary, a promising solution is to collapse a datacube in a number of blocks, storing aggregate data for each of them, and to inquire such data rather than the original ones. When the task of reconstructing the answer from aggregate data is performed, a certain estimation error cannot be avoided. In this paper we propose to store a 64 bit index for each block in order to have a compressed description of data distribution inside the block so that interpolation can be more accurate. Various types of index are described depending on data skew, based on a quad-tree scheme and their approximation errors are analyzed. Although the index is orthogonal to any technique for partitioning a datacube, the index can be also exploited to devise a new greedy technique, based on classical quad-trees, which evaluates the contribution of possible indices while deciding which block is to be partitioned at each step. The experimental results show that the new technique is very effective and gives approximation errors much smaller than other techniques such as wavelets and multidimensional histograms.

1 Introduction

On-Line Analytical Processing (OLAP) is a querying paradigm which deals with particular data structures for summary data, called datacubes. A *datacube* consists of a number of *functional attributes* (also called *dimensions*) and one or more *measure attributes* — thus a datacube is the result of applying an aggregate function to an underlying database relation. Typical operations on a datacube are range queries, that are further aggregations of data over a multi-dimensional range — e.g., given a 2-dimensional datacube C, a range is a rectangle and a range query returns the sum of the measures of all points inside the rectangle.

Computing aggregate queries is a performance bottleneck for many OLAP applications as a datacube can be extremely large. As quick answers are often necessary, a promising solution is to collapse a datacube in a number of blocks (*compressed datacube*), storing aggregate measure values for each of them, and to inquire such data rather than the original ones – *approximate query answering*. The answer to a query on a given range is computed by summing the values of all blocks included in the range and by performing suitable estimations of the pertinence values for the blocks which partially overlap the range. Therefore, a crucial issue for providing good estimations for range queries is to organize the compressed datacube in such a way that estimation errors inside each block are minimized. Two approaches are possible:

1. defining the boundaries of blocks in such a way that a straightforward estimation (in particular, linear interpolation) becomes more effective (e.g., by avoiding that large frequency differences arise inside a bucket);

2. adding further information such as quantitative data on the measure value distribution inside each bucket.

The two approaches are in general competing: possible space for quantitative data could be instead used to obtain finer-grain blocks. Current solutions follow the lines of the ones which have been pro-

posed for histograms (mono-dimensional datacubes): they privileges the exploitation of techniques for an optimal partitioning of a datacube into blocks whereas estimation inside a block receives little attention.

Recently, Buccafurri et al. in [4] have shown that providing a support to intra-block interpolation for histograms has surprising potentialities. In particular, they have proposed to use 32 bits to store the approximated representations of the partial frequency sums at 7 fixed intervals inside a block, organized in a 4-level tree (*4LT*) index. The usage of this index is combined with well-known techniques for constructing histograms, thus obtaining high improvements in the frequency estimation w.r.t. the original methods.

In Section 2 of this paper we present an index of 64 bits, called *2/n-LT index*, for the 2-dimensional case with three different formats, based on a quad-tree structure — we stress that the 4-LT index cannot be immediately extended because of explosion of intervals.

The basic index, called 2/3-LT, divides a block into 4 sub-blocks, which in turn are further divided into other 4 sub-blocks. The index stores the approximate values for the sums of these sub-blocks using a sophisticated storage scheme and it is well-suited for distributions with no strong asymmetry among sub-blocks.

A second type of index, called 2/4LT-index, is oriented to biased distributions for which two of the four sub-block at the first level of the block concentrate more sum than the other ones.

The third type, called 2/p-index (p stands for peak), is designed for capturing distribution having a few high peaks inside one of the sub-blocks.

In Section 3 we use the indices into a new method, called *2/n-LT compression*, for constructing optimal compressions of datacubes which makes a trade-off between further dividing a block into other blocks or just adding an index. As for the indices, also division of blocks is organized as a quad-tree: the root contains the sum of all values in the datacube; then the datacube is divided into 4 blocks and their values are stored in 4 children; later on, some child is (i) further divided into 4 blocks and so on, or (ii) is equipped with an index (the most appropriate among the three types available) or (iii) it is left with no additional details. A quad-tree structure for representing datacubes was first proposed in [15].

Experiments on range queries over a number of syntectic datacubes, reported in Section 4, show that the accuracy of the compression method is very high. Indeed performances are much better than recent estimation methods based on wavelets [16, 23].

In conclusion our work disconfirms the folk credence that is better to use available space for achieving finer-grain blocks rather than adding quantitative data for improving intra-block interpolation: indeed the quad-tree compression with no indices has inferior performances. Further work will be devoted to extend the approach to k-dimensional datacubes.

2 Indexing Datacube Blocks

Throughout the paper we consider given a 2−dimensional *datacube* D with functional attributes X_1 and X_2, also said *dimensions*, and a *measure attribute* Y. We assume that the domain of X_i is $1..d_i$, for each $1 \leq i \leq 2$. A *range* σ_i on the attribute X_i 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 X_i, we denote by $lh(\sigma_i)$ (*left half*) the range $lb(\sigma_i)..\lfloor (lb(\sigma_i) + ub(\sigma_i))/2 \rfloor$ on X_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 X_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. We denote by $vrt(r)$ the set of vertex of r. Informally, a block represents a "rectangular" region of D. A block r of D containing 0 elements is a *null block*.

Given a block r we denote by $sum(r)$ ($avg(r)$, resp.) the sum (the average, resp.) of the measure attribute values (i.e., the elements of the datacube) occurring in the block r.

A simple way for compressing a datacube is dividing it into blocks and substituting each block with a few summary information (typically the sum of the elements occurring in it). Partition may be done according to different criteria (consider for instance the case of semantic-driven aggregation); moreover, the required compression rate may be high. Therefore, in general, we are not guaranteed that blocks contain data distribution well-approximable by linear interpolation (corresponding to CVA, *Continuous Value Assumption*, of histograms). As a consequence, when the compressed datacube is used for evaluating approximate range queries, the estimation error risks to be intolerable. A way for facing the above problem is keeping, beside the overall sum of the element occurring in each block, further information helping us in reconstructing range queries inside the blocks.

Experience acquired in [4, 3] for the mono-dimensional case of histograms inspired us in representing storing approximated sums of internal sub-blocks of a given block b in an hierarchical fashion, by means of a tree index whose root correspond to the whole block, the children to its sub-blocks, the node at the subsequent level to further sub-blocks, and so on. Such a technique aims to produces smaller errors than

a flat partitioning into a number of sub-blocks. Indeed, in this case, the sum of a single sub-block has to be represented as a fraction of the entire sum of the block, whereas, using the index-tree approach, the sum of a sub-block, corresponding to a node of the tree at a certain level, is represented as fraction of just the sum corresponding to the parent node in the tree, that is, in general, a smaller value. Numeric approximation errors deriving from the use of a few bits for representing sums, are therefore in general reduced [4].

We have 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 balanced, and is suitable for distribution with no strong asymmetry, (2) *2/4LT-index*, which is oriented to biased distribution, such that two of the four quadrants of the block concentrate more variance than the other, (3) *2/p(eak)LT-index* which is designed for capturing distribution having a few high density peaks. The three types of indices use the same amount of storage space, 32 bits and are next described in details.

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 1 where nodes correspond to sub-blocks of the block Q of the figure.

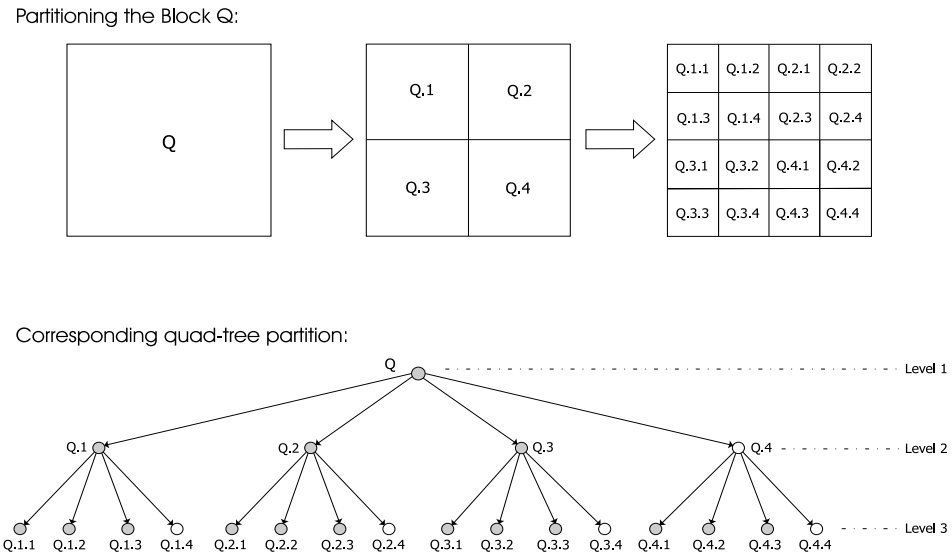

Figure 1: 2/3LT-index

The node at level 1 (i.e., 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 2.

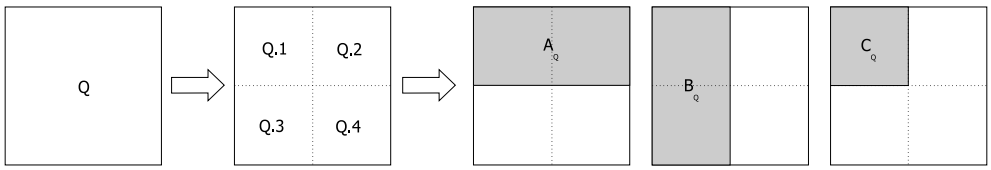

Figure 2: A_Q, B_Q, C_Q regions inside a block

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 3.

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 repre-

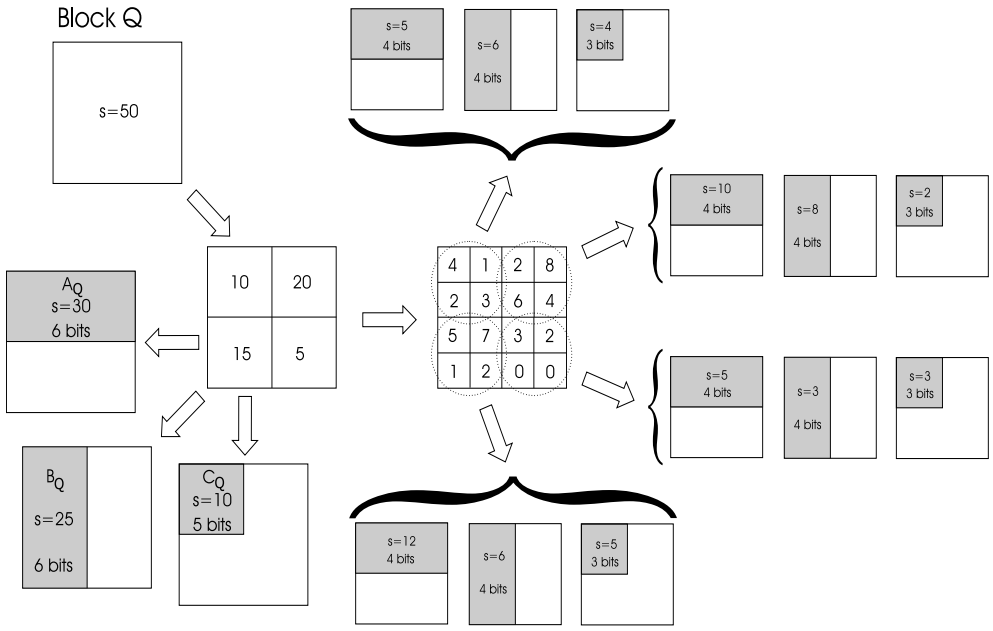

Figure 3: Building a 2/3LT-index

sents 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, for the two regions have the same size and we thus expect, in 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 $L_{sum(C_Q)} = round\left(\frac{sum(C_Q)}{\overline{sum}(AB_Q)} \cdot (2^5 - 1)\right)$, 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 3.

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. In particular, it determines a maximum resolution (additional 2 levels underneath) for one of the four quadrants of the block, a medium resolution (an additional level underneath) for a second quadrant whereas ther other two are not further explored. The structure of the index is reported in Figure 4. The index tries to capture "heterogeneous" data distributions (for instance, the index in Fig. 4 describes a block where the region Q_1 contains a very skewed data distribution, the region Q_2 is less skewed than Q_1, whereas the regions Q_2 and Q_4 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 for assigning resolution 4 and 3, respectively). Thus we need 4 bits for identifying 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 for identify 2/4LT-index among the other ones (i.e., 2/3LT-index and 2/pLT-index).

2/pLT-index. This index is designed for capturing 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,

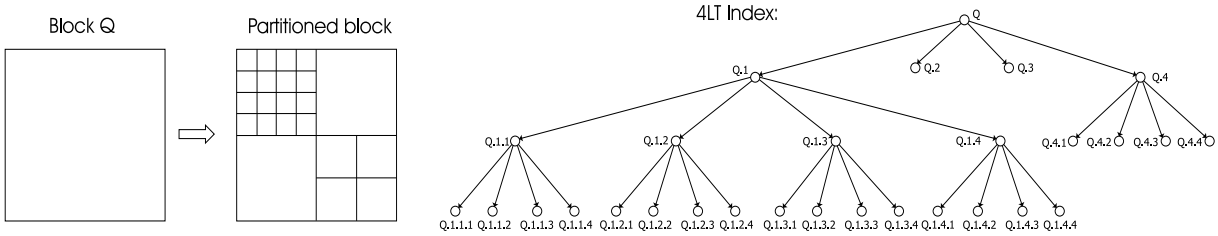

Figure 4: A 2/4LT-index

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, for identifying the 2/pLT-index among the other kinds, and for identifying for which quadrant its internal description is provided.

Summary on the representation of the 2/nLT-indices. The 64 bits od the indices are organized as 2-word frame F — 2/3LT-index requires 61 bits, 2/4LT-index 59 bits and 2/pLT-index requires 62 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 for encoding 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 for identifying 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 by using a 2/nLT-index. Suppose we have a block Q and a range query $sum(r)$, where r is a range inside Q. Again, suppose a 2/nLT-index T is associated to the block Q. The problem is how the range query can be estimated by using T. The technique we use is the following. We divide the block Q into its quadrants and we compute, separately, the portions of the query $sum(r)$ overlapping the four quadrants. Consider the portion of the query $sum(r_1)$ (for the other portions the techniques is identical), where r_1 is obtained as intersection between r and the quadrant Q_1 of Q. We directly estimate $sum(r_1)$ in two cases: if either (1) the index T does not provide, for the quadrant Q_1, higher resolution (i.e., T indices Q_1 by means of a leaf node), or (2) the portion of the query exactly coincides with the block indexed by T at that level (i.e., Q_1). In case (1) we use the the index for estimating the value of $sum(Q_1)$ and estimate $sum(r_1)$ by using linear interpolation. In case (2) the estimation of $sum(Q_1)$ directly provided by the index gives the estimation of $sum(r_1)$. If neither (1) nor (2) occur, we iterate the above process, by dividing Q_1 into its quadrants and, consequently the query $sum(r_1)$, and so on.

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:

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

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 with minimum ϵ_q^{nLT}. Indeed, instead of computing ϵ_q^{nLT} 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 if a cumulative version of the block (i.e., an array containing all possible sums of sub-blocks having a vertex coinciding with a vertex of the block) is available. Also computing the variances of the quarters of the block can be done in constant time.

3 Partitioning the Datacube: the Quad-Tree Technique

The index-based technique of the previous section can be composed with any partition technique for improving estimation inside blocks. But its effectiveness may be dramatically increased if partition technique is able to carry out blocks with SSE sufficiently low. Anyway, the main drawback limiting the effectiveness of any approach producing a general partition, is the storage space required for keeping memory of the partition itself: Advantages deriving from good partitioning risk to be deleted by the extra storage space required for representing the structure of the compressed datacube.

A way for solving the above problem could be finding some type of partition whose representation can be done in a compact fashion. A naive solution could be using the simplest partition (that we call *equi-range partition*), consisting of dividing each dimension into ranges of equal size (possibly different among distinct dimensions). In this way, no additional information has to be stored for representing the partition itself. Unfortunately, studies performed in the mono-dimensional case [4] show that for strongly skewed data such a naive technique does not give satisfactory results. Indeed, blocks so produced do not fit at all any requirement about the variance of contained values, since the partition technique is done "blindly". Moreover, this partition techniques does not allows us to save the storage space associated to blocks containing only null values (in fact, we have to spend anyway 32 bits per block). This is a serious limitation especially in case of datacubes which are typically very sparse structures.

Thus, we have to think to more sophisticated partition techniques, covering the above limitations but, at the same time, requiring a few storage space (w.r.t the space required for storing non structural information).

Our approach consists of applying the indexing-based approach used inside blocks, with no approximation (i.e., using 32 bits for each sum) also to the entire datacube, for generating blocks. In other words, the partition is obtained only by recursively splitting (some) blocks into 4 sub-blocks, but depth of the application of the splitting process, and, thus, the resolution of the aggregated representation, depends on the variance of data inside the datacube: high resolution for non uniform regions, low resolution for uniform ones. Observe that the structural information in the compressed datacube is a little portion: we only have to keep the bit-vector representation associated to the splitting $4-$ary tree (called *quad-tree*), requiring just one bit per node (we will explain in more detail below in the section). The second advantage we obtain by following this approach is that null blocks obtained during the splitting process will not split anymore, since their SSE is null. Thus, for null region, the minimum resolution is adopted. Moreover, the addition of a minimum storage space (1 bit per node) to structural information allows also to save 32 bit used for the sum of the blocks in case the block is null. The third advantage of the quad-tree approach is that we obtain a hierarchical representation of sums at different aggregation levels, and, thus, we allow to access to low levels of the tree only for the (hopefully little) portion of the query not evaluable with the higher levels of the tree.

We next describe in details all the above mentioned issue. In particular: (1) in Section 3.1 we define the *quad-tree* of a datacube (2) in Section 3.2 we deal with the problem of searching a quad-tree minimizing the approximation error and (3) in Section 3.3 we refine the technique proposed in the previous section by including the usage of 2/nLT-indices in the construction of the optimal quad-tree. Concerning the latter point, we observe that we do not limit to add to each *leaf* block of the quad-tree representation a 2/nLT-index, but we propose a step-wise refinement technique allowing us to select blocks where the application of the 2/nLT-index gives effective benefits, re-investing the saved space for increasing the resolution of the quad-tree in some region of the datacube forcing further splits.

3.1 Quad-Tree Datacube

Given a 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.

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 $Root(QTP(D)) = \langle 1..d_1, 1..d_2 \rangle$ and for each $q \in Nodes\ (QTP(D)) \setminus Leaves(QTP(D))$ it holds that the tuple of children of q coincides with its quad-split partition $Q(q)$.

Given a quad-tree partition P, we denote by $Store(P)$ the set $Nodes(P) \setminus Der(P)$.

A *quad-tree datacube* $QTD(D)$ 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 $p \in Store(P)$. For a node $p \in Store(P)$, we also denote by $S(p)$ the

value $sum(p)$.

Note that, for each node $q \in Der(P)$, $sum(q)$ can be derived by using the set S. Indeed, $sum(q) = sum(p) - \sum_{s \in Children(p) \setminus \{q\}} S(s)$, where p is the parent node of q and $Children(p)$ represents the set of children nodes of p. Moreover, the value $sum(p)$ coincides with $S(p)$ in case $p \in Store(P)$.

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

Example 1 *In Figure 5 a graphical representation of a quad-tree on a datacube 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 into blocks of D induced by P.*

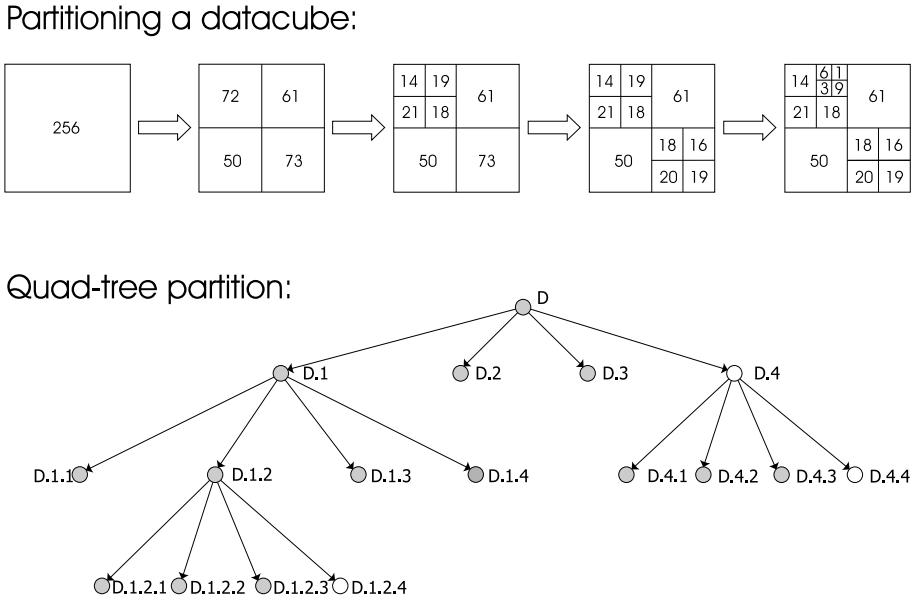

Figure 5: A quad-tree based partition of a datacube

The storage space for a quad-tree datacube $QTD = \langle P, S \rangle$ is the space occupied by the representations of P and S. P can be represented by a string of bits: each bit 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). Indeed, due to how the quad-split partition of non terminal blocks is done, it is not necessary to store any further information about sides of quad-split blocks. For reasons which will be clear in Section 3.3, the string describing the structure of the quad-tree is done by assigning 2 bits for each node of P. In particular, $\langle 0, 0 \rangle$ means leaf node whereas $\langle 1, 1 \rangle$ means split node.

If the number of splits of P is t, then the string $Str(QTD)$ representing P contains $4 \cdot 2 \cdot t + 2$ bits.

The storage space for S is the space occupied by the set $\{s_i | \exists p_i \in Store(P) \wedge \langle p_i, s_i \rangle \in S\}$. Indeed, the information contained in S can be efficiently stored by means of an array $Agg(QTD)$ of size at most $3 \cdot t + 1$ whose elements are the sums calculated inside each block in $Store(P)$. The order in which the sums are stored in such an array expresses their connection to the blocks in $Store(P)$.

Example 2 *Figure 6 reports the string representing the stored sums and the string describing the structure of the quad-tree of Example 1*

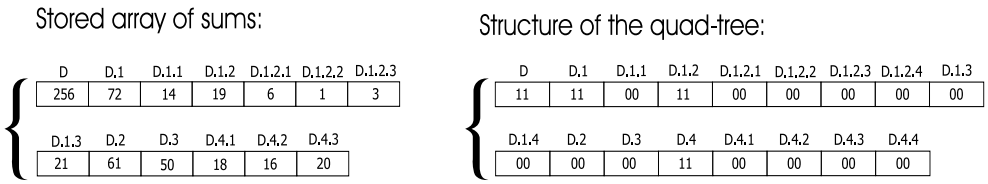

Figure 6: The bit string encoding the structure of a quad-tree

Thus, the overall storage space upper bound for a quad-tree datacube of D with t splits is

$$size(QTD) = 8 \cdot t + 2 + (3 \cdot t + 1) \cdot W \tag{2}$$

where W is the number of bits which are used for storing a sum. As it can be easy realized, suitably limiting the number t of splits, a quad-tree $QTD(D)$ of D may be dramatically more efficient in terms of storage space than the original datacube D. Often, throughout the paper, we refer to $QTD(D)$ also as *the compressed representation* of the datacube D. The crucial issue is how to build $QTD(D)$ in order to maintain satisfactory accuracy in (range) query estimation. This is the matter of the next section.

Concerning the estimation of a query $sum(r)$, where r is a block of D, we just observe that it can be done exactly as the estimation of a query inside a block by using a 2/nLT-index as described in Section 2 (the only difference here is that sums contained in the nodes of the tree are not approximated).

3.2 Constructing an Optimal Quad-Tree Datacube

Suppose the storage space for the quad-tree datacube of D is given, say K. K limits the maximum number t of splits of the datacube, which can be easily derived using results of Section 3.1.

The value of t defines the set of all quad-tree datacubes of D with the same number of splits. Among this set we could choose the best partitioned datacube w.r.t. some metrics. The metrics certainly has to be related to the approximation error, but a number of possible ways for measuring the error of a compressed representation of a datacube can be adopted. Following a well-accepted approach in literature, we measure the "goodness" of the compressed representation of a datacube by using its SSE.

Formally, given a quad-tree datacube $QTD(D)$ of D of p terminal blocks q_1, q_2, ..., q_p,

$$SSE(QTD(D)) = \sum_{i=1}^{p} SSE(q_i) \tag{3}$$

and given a terminal block q_i, such that $1 \leq i \leq p$,

$$SSE(q_i) = \sum_{j \in q_i} (D[j] - avg(q_i))^2. \tag{4}$$

where by $\sum_{j \in q_i}$ we denote that the summation is extended to all the elements of the original datacube D belonging to the block q_i.

Clearly, the lower is $SSE(QTD(D))$ the better is the representation $QTD(D)$, in terms of accuracy.

In order to reach the goal of minimizing the SSE, in favor of simplicity and speed, we chose to use a greedy approach, accepting the possibility of obtaining a sub-optimal solution. A number of possible greedy criteria may be considered, such as choosing 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. Thus, the general algorithm for constructing the *(greedy) quad-tree datacube* with t splits on the datacube D, could be parametric on the adopted greedy criterion G. However, after having compared by experiments all the above mentioned greedy criterion, we have chosen to use the greedy criterion of the maximum SSE. The algorithm is designed in such a way that sparsity of the original datacube is taken into account: every time a new split is produced, 4 new born nodes are added. However, it may happen that some of such nodes corresponds to a null block of the datacube, so we could save the 32 bits used for representing 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. Thus, the two bits (per node) describing the structure of the quad-tree datacube (see Section 3.1) can be used for encoding the different types of nodes. 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 3.3. Clearly, in case (1), the sum of the block is not kept, saving thus 32 bit.

The resulting algorithm is the following:

Greedy Algorithm

Let K be (initially) the total storage space (in bits) of the compressed datacube.

begin

$Q := \langle\ P_0, \{\langle \langle 1..d_1, 1..d_2 \rangle, sum(\langle 1..d_1, 1..d_2 \rangle)\rangle\}\ \rangle$

$K := K - 32 - 2$;

// 32 bits are spent for representing the sum of the entire datacube;

// 2 bits are spent for recording the structure of the root of the quad-tree datacube;

$t := 0$;

// t counts the number of splits;

while $(K > 0)$

Select a node in $Leaves(Part(Q))$, say it p, such that $SSE(p) = max_{q \in Leaves(Part(Q))}\{SSE(q)\}$;

Let $Q^+(p)$ be the set of non null children nodes belonging to the quad-split partition of p different from the right-most node.
$K := K - |Q^+(p)| \cdot 32 - 4 \cdot 2$;
if $(K \geq 0)$
$Q := \langle\ Split(Part(Q), p)\ ,\ Cont(Q)\ \cup \bigcup_{r \in Q^+(p)} \{\langle r, sum(r)\rangle\ \}\ \rangle$;
$t := t + 1$;
end if
end while
return Q;
return t;
end

Therein: (i) P_0 is the partition tree containing only one node (corresponding to the whole datacube), 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.

Informally, the algorithm described above takes t steps for building a quad-tree datacube with t splits. It starts from the quad-tree datacube whose partition tree has a unique node (corresponding to the whole datacube D) and, at each step, selects the best (terminal) block (according to G) and applies the quad-split partition to it.

From the computational complexity point of view the above algorithm is very efficient. Indeed, it can be easily proven that the Greedy Algorithm for the construction of an optimal quad-tree datacube is $O(t \cdot logt)$, where t is the number of splits. Moreover, $t = O(d_1 \cdot d_2)$.

3.3 Enhancing Quad-Tree Datacubes by 2/nLT-indices

In this section we illustrate how we modify the previous greedy algorithm for the construction of an optimal quad-tree datacube in order to improve the estimation accuracy. This issue is addressed by using the already described 2/nLT-indices, giving a more accurate (w.r.t. CVA) approximation of data distributions inside terminal blocks.

Before describing the quad-tree datacube construction, 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 first error we evaluate:

$$\epsilon_q^{nLT} = min_{I \in Best(q)} \epsilon_q(I)$$

where $\epsilon_q(I)$ is defined by (1) in Section 2 and $Best(q)$ is defined in Section 2 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 $sum(q)$.

Consider now an intermediate step of the greedy algorithm and suppose we have fixed the total amount of storage space, say K, required for the compressed representation. Let denote by $Q = \langle P, S\rangle$ the quad-tree datacube we are building. Suppose we break the algorithm at a certain point such that the remainder storage space (from K), is just sufficient to apply, to each node in $Leaves(P)$, the most suitable 2/nLT-index. We do not halt, in general, the construction of the quad-tree datacube here. Indeed, we check if the application of the 2/nLT-index gives a real benefit in all the terminal blocks. There might be nodes such that the application of the 2/nLT-index fails. For detecting such nodes, we evaluate, for each node q, the difference:

$$\epsilon_q^{nLT} - \epsilon_q^{CVA}$$

We expect, in the most cases, a negative value as result. But for some blocks, it might happen that CVA works well than the indexing technique, and thus, we would have a positive value for the above difference. How to manage such a situation? Simply we do not apply the 2/nLT-index to such nodes and we re-invest the associated storage space (if enough) for generating new splits and going in depth, where the greedy criterion drives us, increasing the maximum resolution of the quad-tree partition. Thus, the two bits (per node) describing the structure of the quad-tree datacube (see Section 3.1) can be used for encoding all

possible types of nodes. In particular: (1) $\langle 0,0\rangle$ means non null terminal node with no 2/nLT-index (i.e., CVA), (2) $\langle 0,1\rangle$ means null terminal node, (3) $\langle 1,0\rangle$ means non null terminal node with 2/nLT-index (4) $\langle 1,1\rangle$ means split node (i.e., non terminal node). Recall that, in case (1), the sum of the block is not kept, saving thus 32 bit. The quad-tree datacube construction algorithm is thus designed on the basis of the above reasoning. At each step of the construction of the quad-tree, we apply the 2/nLT-indices only to the terminal nodes of the quad-tree for which we obtain benefits; then, at the next step, first we select the node to be split according to the criterion of maximum SSE, then we split it; however, if the split node was equipped with a 2/nLT-index, the index is removed and the associated storage space is re-invested. The process halts when the remainder available storage space is not enough for generating a new split. The resulting algorithm is formalized as follows:

Indexed Quad-Tree Datacube Construction

Let $good(C)$ be a function receiving a set of blocks C and returning the subset of blocks q of C such that $\epsilon_q^{nLT} - \epsilon_q^{CVA} < 0$ (i.e., the application of a 2/nLT-index is fruitful).

Let K be (initially) the total storage space (in bits) of the compressed datacube.

begin

$Q := \langle\ P_0, \{\langle\langle 1..d_1, 1..d_2\rangle, sum(\langle 1..d_1, 1..d_2\rangle)\rangle\}\ \rangle$

$nLT(Q) := good(\{\langle 1..d_1, 1..d_2\rangle\}) \cap \{\langle 1..d_1, 1..d_2\rangle\}$

// $nLT(Q)$ represents the set of blocks of the current nLT-approximable terminal blocks.

$K := K - 32 - |good(\{\langle 1..d_1, 1..d_2\rangle\})| \cdot 64 - 2;$

// 32 bits are spent for representing the sum of the entire datacube;

// $|good(\{\langle 1..d_1, 1..d_2\rangle\})| \cdot 64$ counts bits spent for applying the 2/nLT-index to the entire datacube;

// 2 bits are spent for recording the structure of the root of the quad-tree datacube;

$t := 0;$

// t counts the number of splits;

while $(K \geq 0)$

Select a node in $Leaves(Part(Q))$, say it p, such that $SSE(p) = max_{q \in Leaves(Part(Q))}\{SSE(q)\}$;

Let $Q^+(p)$ be the set of non null children nodes belonging

to the quad-split partition of p different from the right-most node.

$K := K - |Q^+(p)| \cdot 32 - |good(Q(p))| \cdot 64 + |(nLT(Q) \cap \{p\})| \cdot 64 - 4 \cdot 2;$

if $(K \geq 0)$

$Q := \langle\ Split(Part(Q), p)\ ,\ Cont(Q)\ \cup \bigcup_{r \in Q^+(p)}\{\langle r, sum(r)\rangle\ \}\ \rangle;$

$t := t + 1;$

$nLT(Q) := nLT(Q) \cup good(Q(p)) \setminus (nLT(Q) \cap \{p\});$

end if

end while

apply the (suitable) 2/nLT-index to each block in $nLT(Q)$;

return Q;

return t;

end

---

where (i) P_0 is the partition tree containing only one node (corresponding to the whole datacube), 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. As a final remark, we observe that the computational complexity of the above greedy algorithm is the same as the greedy algorithm described in Section 3.2.

4 Experimental Results

In this section we present some experimental results about the accuracy of estimating range sum queries on quad-trre datacubes, comparing our method with the state-of-the-art techniques in the context of compressed datacubes. In particular, we compare our technique with the histogram-based technique MHIST proposed in [20], and with the wavelet-based techniques proposed respectively in [23] and [24]. In order to prove that the usage of 2/nLT-indices improves the accuracy of quad-tree datacubes, we have tested our method with and without 2/nLT-indices. In the following we denote by QT^- the quad-tree based method not using 2/nLT-indices (that is, the method defined in Section 3.2), and by QT the quad-tree based method using such indices (that is, the method defined in Section 3.3).

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 bucket which is the most in need of partitioning (as explained below), and partitions it along one of its dimensions. The bucket B to be split is the one which is characterized by an attribute X_i whose individual data distribution (called *marginal distribution*) contains two adjacent values e_j, e_{j+1} with the largest difference in source values. B is split along the dimension of X_i by putting a boundary between e_j and e_{j+1}. For each bucket, 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 S, the number of buckets which can be stored is given by: $\lfloor S/3 \rfloor$.

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 [17, 22]), and recently have been applied to selectivity estimation [16] and to the approximation of OLAP range queries over data cubes [23, 24]. 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 S, the number of buckets which can be stored is given by: $\lfloor S/2 \rfloor$.

The compression technique described in [23] does not apply the wavelet transform directly to the source data cube. First, the partial sum data cube is generated, and each of its cell values is replaced with its natural logarithm (it has been shown that the combination of the logarithm transformation with the approximation technique generally reduces the relative error of the approximation). Next, the above described compression process is applied to such an obtained cube.

In [24] a sophisticated wavelet based technique which mainly aims to improve the I/O efficiency of the compact data cube construction is proposed. A difference with the approach described above is that it is applied directly on the source data cube.

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

4.1 Measuring approximation error

Let's 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 [24], 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:

$QS_1 = \{Sum(r) |\ r = \langle \sigma_1, \sigma_2 \rangle$ is a range such that: $vrt(r) \cap vrt(D) \neq \emptyset\}$;
$QS_1^+ = \{Sum(r) |\ r = \langle \sigma_1, \sigma_2 \rangle$ is a range such that: $Sum(r) \in QS_1$ and $Sum(r) > 0\}$;
$QS_1^0 = QS_1 \setminus QS_1^+$;
$QS_2(\Delta_1, \Delta_2) = \{Sum(r) |\ r = \langle \sigma_1, \sigma_2 \rangle$ is such that: $ub(\sigma_1) = lb(\sigma_1) + \Delta_1$ and $ub(\sigma_2) = lb(\sigma_2) + \Delta_2\}$;
$QS_2^+(\Delta_1, \Delta_2) = \{Sum(r) |\ r = \langle \sigma_1, \sigma_2 \rangle$ is a range such that: $Sum(r) \in QS_2(\Delta_1, \Delta_2)$ and $Sum(r) > 0\}$;
$QS_2^0(\Delta_1, \Delta_2) = QS_2(\Delta_1, \Delta_2) \setminus QS_2^+(\Delta_1, \Delta_2)$.

That is, QS_1 contains sum queries defined over ranges such that one of their corners coincides with a corner of the data cube. QS_1^+ is the subset of QS_1 containing queries whose answer is not null. $QS_2(\Delta_1, \Delta_2)$ contains sum queries defined over all ranges of size $\Delta_1 \cdot \Delta_2$. $QS_2^+(\Delta_1, \Delta_2)$ is the subset of QS_2 containing queries whose answer is not null. Note that the value of the relative error for a query belonging to QS_1^+ or $QS_2^+(\Delta_1, \Delta_2)$ fulfils the classical definition of relative error.

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 $\| e^{abs}(QS) \|$.

4.2 Synthetic Data Sets

The synthetic data sets used in our experiments are similar to those of [24]. The synthetic data generator populates r rectangular regions of a bi-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 bi-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 is 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.

4.3 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 datacubes. The storage space is expressed as the number of 32 bits integers which are available for the compressed representation of the datacube.

Storage space We considered several sparse datacubes of size $2000 \cdot 2000$ generated setting $l_{min} = 25$, $l_{max} = 70$, $z_{min} = 0.5$, $z_{max} = 1.5$, containing on the average about 23000 non zero cells, and dense datacubes of size $500 \cdot 500$ with $l_{min} = 90$, $l_{max} = 130$, $z_{min} = 0.5$, $z_{max} = 1.5$, containing on the average about 97000 non zero cells. The accuracy of the estimation w.r.t. the storage space is depicted in Fig.7 (sparse datacube) and Fig.8(dense datacube), where $\| e^{rel}(QS_1^+) \|$, $\| e^{abs}(QS_1^0) \|$ and $\| e^{rel}(QS_1) \|$ for the different techniques are compared. 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 right-hand picture of Fig.7 and Fig.8 only QT and QT^- are compared, since the errors produced by the other methods are out of scale.

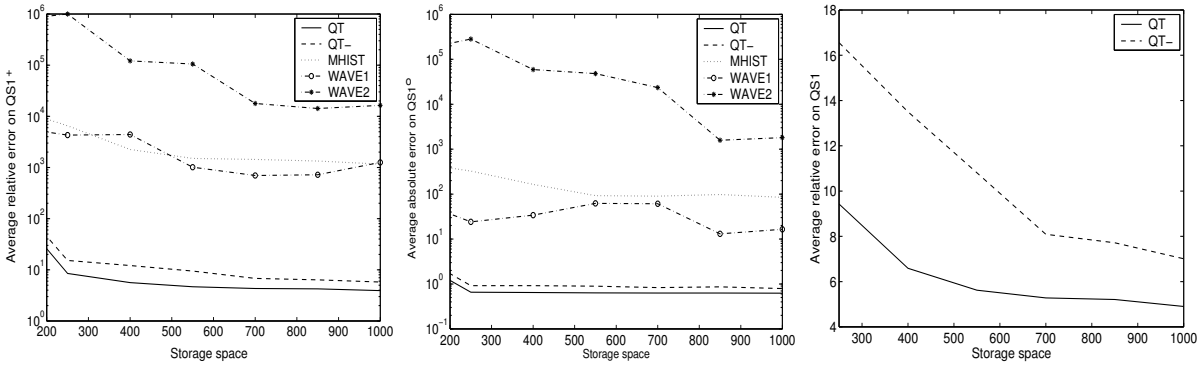

Figure 7: $\| e^{rel}(QS_1^+) \|$, $\| e^{abs}(QS_1^0) \|$ and $\| e^{rel}(QS_1) \|$ w.r.t. the storage space for a sparse datacube

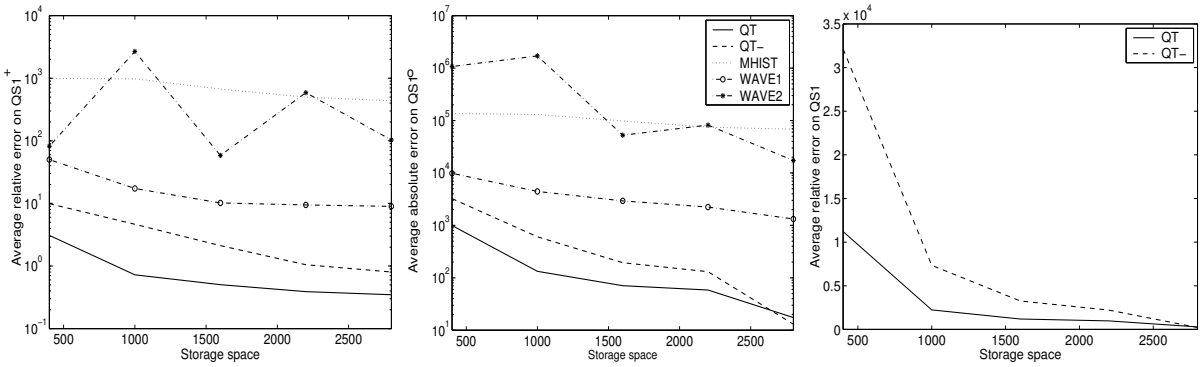

Figure 8: $\| e^{rel}(QS_1^+) \|$, $\| e^{abs}(QS_1^0) \|$ and $\| e^{rel}(QS_1) \|$ w.r.t. the storage space for a dense datacube

Skew inside regions We considered sparse datacubes 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 picture in the left-hand side of Fig.9. Interestingly, every 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 QT group these values into the same buckets causing small errors, and the wavelet decomposition applied in these regions generates a lot of with value zero. Analogously, when the skew is small, the frequencies corresponding to different values are nearly the same and so 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 datacubes 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 pictures in the centre and in the right-hand side of Fig.9, $\| e^{rel}(QS_2^+(\Delta, \Delta)) \|$ and $\| e^{rel}(QS_2(\Delta, \Delta)) \|$, for different values of Δ, are shown. In Fig.10, analogous results for dense datacubes are reported.

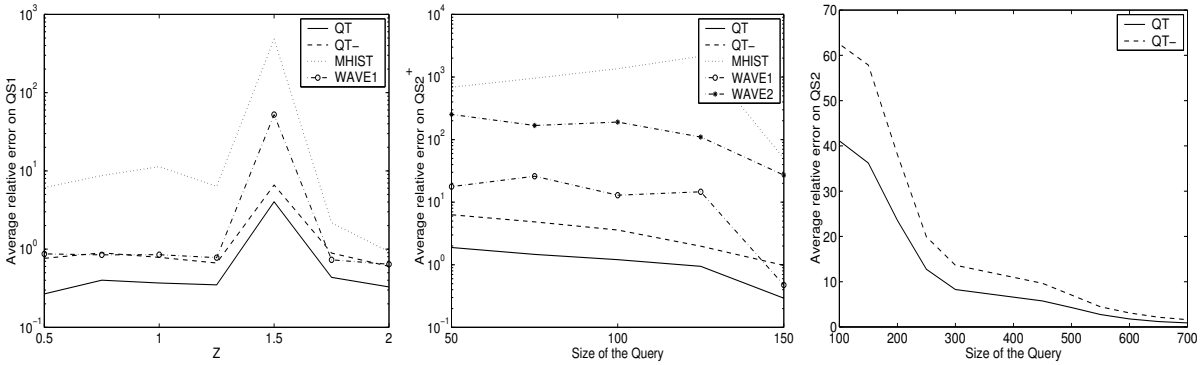

Figure 9: Results for sparse datacubes

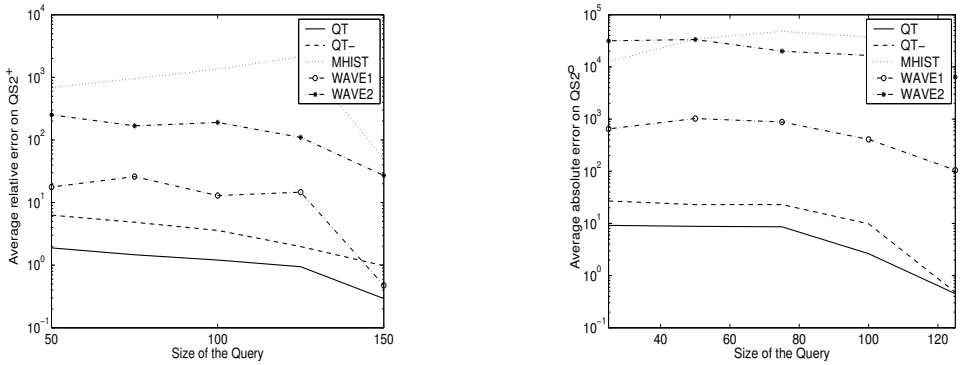

Figure 10: Results for dense datacubes on query set QS_2

References

[1] Buccafurri, F., Rosaci, D., Sacca', D., Compressed datacubes for fast OLAP applications, *DaWaK 1999,*, Florence, 65-77.

[2] Buccafurri, F., Furfaro, F., Sacca', D., Estimating Range Queries using Aggregate Data with Integrity Constraints: a Probabilistic Approach, Proc. of the 8th International Conference on Database Theory, London, UK, 4-6 January, 2001.

[3] Buccafurri, F., Furfaro, F., Lax, G., Sacca', D., Binary-tree histograms with tree indices, *Proc. of DEXA*, 2002.

[4] Buccafurri, F., Pontieri, L., Rosaci, D., Sacca', D., Improving Range Query Estimation on Histograms, *Proc. of ICDE*, 2002.

[5] Blohsfeld ,B., Korus, D., Seeger, B., A Comparison of Selectivity Estimators for Range Queries on Metric Attributes, *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, Philadelphia, USA, 1999.

[6] Chaudhuri, S., Dayal, U., An Overview of Data Warehousing and OLAP Technology, *ACM SIGMOD Record 26(1)*, March 1997.

[7] Donjerkovic, D., Ioannidis, Y.E., Ramakrishnan, R., Dynamic Histograms: Capturing Evolving Data Sets, *Proc. of the 16th International Conference on Data Engineering, ICDE 2000*, San Diego, California, 2000.

[8] Y. Ioannidis. Universality of Serial Histograms. In *Proceedings of 1993 International Conference of 1993 VLDB Very Large Data Bases Conference*, pages 256-267, 1993

[9] Y. Ioannidis, V. Poosala. Balancing histogram optimality and practicality for query result size estimation. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 233-244, 1995

[10] Y. Ioannidis, V. Poosala. Histogram-based Approximation of Set-Valued Query Answers. In *Proceedings of the 25th International Conference on Very large Data Bases, VLDB 1999* 1999.

[11] H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, T. Suel. Optimal histograms for quality guarantees. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 275-286, 1998

[12] H. V. Jagadish, Hui Jin, Beng Chin Ooi, Kian-Lee Tan. Global optimization of histograms. In *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, pages 223-234, 2001

[13] Koudas, N., Muthukrishnan, S., Srivastava, D., Optimal Histograms for Hierarchical Range Queries, *Proc. of Symposium on Principles of Database Systems - PODS* pp. 196-204, Dallas, Texas, 2000.

[14] Konig, A. C., Weikum, G., Combining Histograms and Parametric Curve Fitting for Feedback-Driven Query Result-size Estimation, *Proc. of Very Large Data Bases (VLDB) Conference* 1999.

[15] Lazaridis, I., Mehrotra, S., Progressive Approximate Aggregate Queries with a MultiResolution Tree Structure, *Proc. of Symposium on Principles of Database Systems - PODS*, 2001

[16] Y. Matias, J. S. Vitter, M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of the 1998 ACM SIGMOD Conference on Management of Data*, Seattle, Washington, June 1998

[17] Natsev, A., Rastogi, R., Shim, K., WALRUS: A Similarity Retrieval Algorithm for Image Databases, *In Proceedings of the 1999 ACM SIGMOD Conference on Management of Data*, 1999.

[18] V. Poosala. *Histogram-based Estimation Techniques in Database Systems*. PhD dissertation, University of Wisconsin-Madison, 1997

[19] Poosala, V. Ganti, V., Fast Approximate Answers to Aggregate Queries on a Datacube. In *Proc. of the 1999 International Conference on Scientific and Statistical Databases Managemets.*

[20] V. Poosala, Y. E. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proceedings of the VLDB*, 1997

[21] Poosala, V., Ganti, V., Ioannidis, Y.E., Approximate Query Answering using Histograms, *IEEE Data Engineering Bulletin* Vol. 22, March 1999.

[22] E. J. Stollnitz, T. D. Derose, and D. H. Salesin. *Wavelets for Computer Graphics*. Morgann Kauffmann, 1996

[23] J. S. Vitter, M. Wang, B. Iyer. Data Cube Approximation and Histograms via Wavelets. In *Proceedings of the 1998 CIKM International Conference on Information and Knowledge Management*, Washington, 1998

[24] J. S. Vitter, M. Wang, Approximate Computation of Multidimansional Aggrgates of Sparse Data using Wavelets, In *Proceedings of the 1999 ACM SIGMOD International Conference on Managemnet of Data*, 1999

[25] G. K. Zipf. *Human behaviour and the principle of least effort*. Addison-Wesley, Reading, MA, 1949