$\mathcal{LCS}$-Hist: Taming Massive High-Dimensional Data Cube Compression

Alfredo Cuzzocrea
ICAR-CNR and University of Calabria, Italy
cuzzocrea@si.deis.unical.it

Paolo Serafino
ICAR-CNR and University of Calabria, Italy
serafino@si.deis.unical.it

ABSTRACT

The problem of efficiently compressing massive high-dimensional data cubes still waits for efficient solutions capable of overcoming well-recognized scalability limitations of state-of-the-art histogram-based techniques, which perform well on small-in-size low-dimensional data cubes, whereas their performance in both representing the input data domain and efficiently supporting approximate query answering against the generated compressed data structure decreases dramatically when data cubes grow in dimension number and size. To overcome this relevant research challenge, in this paper we propose *$\mathcal{LCS}$-Hist*, an innovative multidimensional histogram devising a complex methodology that combines intelligent data modeling and processing techniques in order to tame the annoying problem of compressing massive high-dimensional data cubes. With respect to similar histogram-based proposals, our technique introduces (*i*) a surprising consumption of the storage space available to house the compressed representation of the input data cube, and (*ii*) a superior scalability on high-dimensional data cubes. Finally, several experimental results performed against various classes of data cubes confirm the advantages of *$\mathcal{LCS}$-Hist*, even in comparison with those given by state-of-the-art similar techniques.

1. INTRODUCTION

During the last two decades, research communities have devoted a great deal of attention to the annoying problem of compressing data cubes for various purposes, among which providing *approximate answers* to resource-intensive OLAP queries against such multidimensional data structures is the most relevant one. Briefly, benefits coming from the data cube compression proposal can be synthesized in a *faster evaluation* of queries that is balanced by some *approximation* in the final answers, which, however, is perfectly tolerable with respect to OLAP analysis goals [6]. In turn, performance improvement gained from mitigating effects of resource-intensive OLAP query evaluation makes consolidated methodologies for extracting *integrated, summarized, OLAP-shaped knowledge* from large amounts of data stored in massive corporate data cubes, like *On-Line Analytical Mining* (OLAM), feasible-in-practice and efficient.

According to these considerations, a plethora of *approximate query answering techniques* have been proposed in literature, each of them aiming at minimizing the occupancy of compressed data cubes, while, at the same time, minimizing the query error due to *approximate answers* to OLAP queries against such condensed representations. Among all the alternatives, *histograms*, *wavelets* [18], and *random sampling* (e.g., [7]) are the most popular solutions, which, in several instances, have been implemented within the core layer of commercial OLAP server platforms. Specifically, histograms, which have a long history, are the most successful proposal, and offer the best performance in both compressing data cubes and efficiently supporting approximate query answering.

Nevertheless, just like other approximate query answering techniques, performance of histograms decreases when data cubes grow in dimension number and size. In more detail, histograms suffer of *scalability issues* (e.g., see [6]), i.e. they perform well on small-in-size low-dimensional data cubes whereas they do not scale satisfactorily on massive high-dimensional data cubes. For this reason, when the latter kind of data cubes are considered, we generally observe a significant performance degradation in both representing the input data domain and introducing low (query) errors in the retrieved approximate answers. It should be noted that, from the application side perspective, the latter is the most problematic issue to be faced-off, being OLAP queries the baseline operations on top of which advanced *Knowledge Discovery in Multidimensional Databases* (KDMD) processes (e.g., OLAM and *Decision Support* (DS) processes) are implemented. Unfortunately, real-life data cubes that one can find in *Data Warehousing* (DW) systems, *Business Intelligence* (BI) systems, *Sensor Network Data Analysis* tools, and, without any loss of generality, in all those data-intensive application scenarios where distributed massive data must be analyzed on the basis of a multidimensional and multi-resolution vision, are characterized by a dimension number that, due to the same processes according to which knowledge is produced, processed and mined, is very large, and can easily reach several tens of dimensions in most cases!

To adequately fulfill this gap between the nature of real-life data cubes and resulting compression/approximate-query-answering issues deriving-from and emphasized-by massive sizes and high dimension numbers, in this paper we propose an innovative multidimensional histogram, called *$\mathcal{LCS}$-Hist*, whose main goal is *to tame the annoying problem of compressing massive high-dimensional data cubes*. Similarly to histogram-based data cube compression techniques, *$\mathcal{LCS}$-Hist* makes use of a *partitioned representation* of the input data cube in terms of *buckets*, data blocks storing some aggregate information on the items they contain. To this end, the methodology underlying *$\mathcal{LCS}$-Hist* defines an innovative data cube compression technique that

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EDBT'09, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003…$5.00.

combines: (*i*) *£inear programming*, (*ii*) *Constrained partitions of multidimensional data domains*, and (*iii*) *Similarity metrics on one-dimensional histograms*. The main motivation underlying the vision carried out by *£CS-Hist* is the following. Since traditional histogram-based data cube compression techniques introduce problematic limitations when applied to massive high-dimensional data cubes, *combine* advanced data modeling and processing techniques in order to obtain a final compressed data structure (i.e., *£CS-Hist*) that, although paying something in terms of computational overheads, allows us to achieve excellent performance in both representing the input data cube and efficiently supporting approximate query answering to resource-intensive OLAP queries against the compressed data cube. Moreover, contrary to state-of-the-art data cube compression techniques, *£CS-Hist* ensures high scalability and efficiency on massive high-dimensional data cubes, as we experimentally prove in our comprehensive experimental analysis.

Being *£CS-Hist* computed in an off-line manner, likewise to what happens with other common maintenance operations of commercial OLAP server platforms, the cost needed for computing *£CS-Hist* is *transparent* to the target OLAP client application, whereas, at query time, the latter takes advantages from obtaining fast, approximate answers to resource-intensive OLAP queries for knowledge discovery and decision support purposes.

Another peculiar characteristic of *£CS-Hist* is the idea of combining a *collection* of intelligent data modeling and processing techniques in a *systematic methodology* that, overall, allows us to effectively tame massive high-dimensional data cube compression. This approach has similar, well-known initiatives in the context of histogram-based data cube compression techniques. Just like *£CS-Hist*, *Min-Skew* [1], *GenHist* [10], and *STHoles* [3] histograms propose computing the final compressed representation of the input data cube by means of a rather complex methodology, which makes such class of proposals a breaking point with respect to traditional approaches exposing simpler build algorithms (e.g., *Equi-Depth* [14] and *MHist* [16] histograms). In virtue of this, in our work we consider *Min-Skew* [1], *GenHist* [10], and *STHoles* [3] histograms as similar, related techniques, whereas the comparison of *£CS-Hist* with different-in-nature data cube compression techniques appeared in literature recently, such as *condensed* [19] and *Dwarf* [17] cubes, or *distributed and parallel data cube compression methodologies*, such as [8], is outside the scope of this paper and postponed as future work. On the other hand, latest proposals in the context of approximate query answering have confirmed the trend of encompassing complex methodologies to solve this so-exciting research challenge. Among these proposals, we mention the *optimized stratified sampling technique* by Chaudhuri *et al.* [4], and the *DBO OLAP engine* by Jermaine *et al.* [12]. Despite this analogy, both [4] and [12] are not comparable with our work as [4] makes strong assumptions on the target query-workload in order to improve the quality of stratified sampling, and [12] investigates the specific problem of supporting *scalable online aggregation*, which can be both reasonably considered as application scenarios very different from ours.

An admissible limitation of complex histogram-based data cube compression techniques like ours is represented by the fact that, due to the computational overheads introduced by the underlying (complex) build/restructure procedure, such techniques are not suitable to be adopted within *dynamic* software components extending OLAP server platforms and capable of efficiently accomplishing updates that can happen in the relational data sources alimenting the target data cube. This amenity could be achieved by means of methodologies allowing the *actual* configuration of the compressed representation to be rapidly *re-tuned* accordingly, like happens in other proposals (e.g., [3]) inspired by widely-recognized principles of *self-tuning databases*. However, these research aspects are outside the scope of this paper, and the usage of *£CS-Hist* as off-line component within the core layer of OLAP server platforms can be reasonably assumed as the *most appropriate* one to be considered.

The remaining part of this paper is organized as follows. In Section 2, we report on state-of-the-art complex histogram-based data cube compression techniques, namely *Min-Skew* [1], *GenHist* [10], and *STHoles* [3] histograms. Section 3 is devoted to fundamentals and basic definitions of our research. In Section 4, due to the complexity of the methodology underlying *£CS-Hist*, we provide a useful overview of such methodology throughout a sketch on the *£CS-Hist* build algorithm. Section 5 describes how the data cube partition underlying *£CS-Hist* is obtained by means of an innovative *dynamic programming procedure*. In Section 6, we show how the partition above is further optimized by means of imposing *a set of partitioning constraints*. In Section 7, we introduce our innovative *similarity metrics on one-dimensional histograms*, which allows us to reduce the overall size of the partition and, as a consequence, the final occupancy of *£CS-Hist*, while ensuring a high degree of accuracy of the retrieved approximate answers. As a noticeable side effect, this strategy allows us to obtain high scalability and efficiency of *£CS-Hist* in both representing the input data cube and efficiently supporting approximate query answering in the presence of massive sizes and high dimension numbers. In Section 8, we describe how the final constrained partition of the input data cube is obtained. This partition is the one on top of which the final histogram is computed. Section 9 is devoted to our comprehensive experimental evaluation that demonstrates the advantages coming from *£CS-Hist*, even in comparison with state-of-the-art similar techniques. Finally, in Section 10, we complete our overall contribution by discussing conclusions and putting the basis for future work in this research field.

2. RELATED WORK

As highlighted in Section 1, within the context delineated by the wide literature on histogram-based data cube techniques, three reference proposals are close to our work: *Min-Skew* [1], *GenHist* [10], and *STHoles* [3] histograms. Similarly to ours, these techniques make use of complex, systematic methodologies to deal with the annoying problem of compressing massive high-dimensional data cubes. For this reason, in this Section we provide a brief overview on these proposals.

Min-Skew histogram was originally designed by Acharya *et al.* [1] to tackle the problem of selectivity estimation of *spatial data* in *Geographical Information Systems* (GIS). Spatial data are referred to *spatial* (or *geographical*) *entities* such as points, lines, poly-lines, polygons and surfaces, and are very often treated by means

of minimal rectangles containing them, namely *Minimum Bounding Rectangles* (MBR). The main idea behind a *Min-Skew* histogram is to follow the criterion of minimizing the *spatial skew* of the histogram by performing a *Binary Space Partitioning* (BSP) via recursively dividing the data space along one of the dimensions each time. More formally, each point in the space of a given GIS instance is associated to a *spatial density*, defined as the number of MBR that contain such a point. When performing the partition, the *spatial skew* $S(B_i)$ is assigned to each bucket B_i. $S(B_i)$ is defined as follows: $S(B_i) = \sum_{k=0}^{N(B_i)-1} (f(k) - \bar{f}_i)^2 / N(B_i)$, where: (*i*) $N(B_i)$ is the number of data items contained within B_i; (*ii*) $f(k)$ is the *spatial frequency* of the *k*-th item within B_i; (*iii*) $\bar{f}_i$ represents the *average frequency* of all the items within B_i. The *total skew* $S(H)$ of the histogram H is defined as follows: $S(H) = \sum_{k=0}^{N_B-1} N(B_k) \cdot S(B_k)$, where N_B is the number of buckets of the histogram H, and $N(B_k)$ and $S(B_k)$ are defined as introduced above. The build algorithm of *MinSkew* tries, at each step, to minimize the overall spatial skew of the histogram by selecting (*i*) a bucket to be split, (*ii*) a dimension of the multidimensional space along which splitting, and (*iii*) a splitting point in that dimension, such that the overall spatial skew computed after the split is smaller than the one computed at the previous step. Finally, noticing that the spatial skew captures the variance of the spatial density of MBR within each bucket, we can say that *Min-Skew* follows, in some sense, the spirit of the well-know *V-Optimal* histogram.

Gunopulos *et al.* propose *GenHist* histogram [10], a multidimensional histogram that is different from the previous ones with respect to the particular nature of the build algorithm. The key idea of *GenHist* is the following. Given a histogram H with N_B buckets on the input data cube A, a *GenHist* histogram is built by finding n_b overlapping buckets on H, such that n_b is an input parameter. To this end, thanks to a greedy algorithm that considers *increasingly-coarser grids*, the technique individuates the number of distinct regions that is much larger than the original number of buckets N_B. At each step, this algorithm selects the set of cells J of highest density, and moves enough randomly-selected points from J into a bucket to make J and its neighboring cells "close-to-uniform". Therefore, the novelty of this proposal consists in defining a truly multidimensional splitting policy, based on the concept of *tuple density*. A drawback of the *GenHist* proposal is represented by the difficulty of choosing the right values for setting the input parameters, which are quite numerous.

Bruno *et al.* [3] propose the *workload-aware histogram STHoles*, a multidimensional histogram based on the *analysis of the query-workload* on the target data cube. Rather than an arbitrary overlap, a *STHoles* histogram allows bucket nesting, thus achieving the definition of the so-called *bucket tree*. The query-workload is handled as follows. Query result stream Q_R to be analyzed is intercepted and, for each query Q_k belonging to Q_R and for each bucket B_i belonging to the current bucket tree T_B, the number $|Q_k \cap B_i|$ is counted. Then, "holes" in B_i for regions of different tuple density are "drilled" and "pulled out" as children of B_i. Finally, buckets of similar densities are merged in such a way as to keep the number of buckets constant. *STHoles* makes use of a tree-like in-memory-data-structure, since the parent-child relationship in a tree well represents the nesting relationship, and the sibling-sibling relationship is well represented by buckets nested within the same bucket. Thereby, *STHoles* build algorithm does not take into account the original data set; indeed, the needed information is instead gathered by inspecting the target query-workload and query feedback. This amenity makes *STHoles self tunable*, i.e. adaptable to updates and modifications of the query-workload's characteristics. On the basis of this strategy, a relevant amount of the total storage space available for housing the histogram is invested in representing "heavy-queried regions", thus providing a better approximation for such regions, whereas a fewer storage space is reserved to "lowly-queried regions", thus admitting some inaccuracy for such regions. In addition to this, in [3] authors also show that, on the DBMS Microsoft SQL Server, query-workload analysis overheads introduced by *STHoles* are very low, less than 10 % of the overall DBMS throughput.

3. FUNDAMENTALS AND BASIC DEFINITIONS

For the sake of explanation, in this Section we introduce fundamentals and definitions used throughout the paper. Let A be an n-dimensional OLAP data cube having $D = \{d_0, d_1, \ldots, d_{n-1}\}$ as dimension set. A *data cell* of A is denoted as $C[k_0, k_1, \ldots, k_{n-1}]$, such that $\{k_0, k_1, \ldots, k_{n-1}\}$ represents the set of OLAP *metadata* that univocally identify C within the n-dimensional space defined by A, each k_j being a *member* of the dimension d_j. When there is not the need of explicitly referring the OLAP metadata of a generic data cell C of A (i.e., $C[k_0, k_1, \ldots, k_{n-1}]$), we will refer to such cell as C simply.

A bucket over A is a (n-dimensional) sub-set of the n-dimensional space of A, denoted as $B[l_0{:}u_0, l_1{:}u_1, \ldots, l_{n-1}{:}u_{n-1}]$, where each pair $[l_j{:}u_j]$ defines the bounds of B along the dimension d_j. Likewise, we will denote a generic bucket of A as B simply, whenever there is not the need of explicitly referring its bounds along dimensions of A.

Given an n-dimensional bucket $B[l_0{:}u_0, l_1{:}u_1, \ldots, l_{n-1}{:}u_{n-1}]$, we say that a data cell $C[k_0, k_1, \ldots, k_{n-1}]$ is a *vertex* of B if either $k_j = l_j$ or $k_j = u_j$, for each j in $\{0, 1, \ldots, n-1\}$.

It should be noted that each OLAP dimension constitutes a *totally ordered set*. Given a dimension d_j of a data cube A, if members of d_j are numerical, then the order relation of d_j is naturally defined. Otherwise, if members of d_j are categorical, then the order relation of d_j is the one exposed by the target OLAP server (this depending on the way multidimensional data are organized, represented and processed). Therefore, it makes sense to introduce the notion of *interval* of members over an OLAP dimension. To this end, instead of explicitly referring a specific member k_j of d_j, we will denote such member *by means of an index on its position in the ordering of* d_j. This notation also suggests that, given data cell C of A, apart from the logical representation, C could alternatively be regarded as a *vector* in the *vectorial space* defined by A, denoted as $\mathcal{Z}(A)$.

Given a data cell $C[k_0, k_1, \ldots, k_{n-1}]$ and a bucket $B[l_0{:}u_0, l_1{:}u_1, \ldots, l_{n-1}{:}u_{n-1}]$, we say that C is *contained* within B, i.e. $C \in B$, if $l_j \leq k_j \leq u_j$, for each j in $\{0, 1, \ldots, n-1\}$. Furthermore, we define *Cells*(B) as the set of all data cells in A that are contained within the bucket B.

Finally, some statistical properties of buckets will also be used throughout the paper. Here, we introduce these properties. Given

a bucket $B[l_0:u_0, l_1:u_1, \dots, l_{n-1}:u_{n-1}]$, we define the *statistical mean* of B, denoted as $\mu(B)$, as follows:

$$\mu(B) = \frac{\sum_{k_0=l_0}^{u_0} \sum_{k_1=l_1}^{u_1} \cdots \sum_{k_{n-1}=l_{n-1}}^{u_{n-1}} C[k_0, k_1, \dots, k_{n-1}]}{|d_0| \times |d_1| \times \cdots \times |d_{n-1}|} \quad \textbf{(1)}$$

and the *statistical variance* of B, denoted as $\sigma^2(B)$, as follows:

$$\sigma^2(B) = \frac{\sum_{k_0=l_0}^{u_0} \sum_{k_1=l_1}^{u_1} \cdots \sum_{k_{n-1}=l_{n-1}}^{u_{n-1}} \left(C[k_0, k_1, \dots, k_{n-1}] - \mu(B)\right)^2}{|d_0| \times |d_1| \times \cdots \times |d_{n-1}|} \quad \textbf{(2)}$$

4. *ℒCS-Hist* OVERVIEW

In this Section, we provide a sketch on *ℒCS-Hist* throughout its build algorithm. This allows us to give a first intuition of the overall data cube compression methodology underlying *ℒCS-Hist* we propose. Then, in the following Sections, we detail each main task of our proposed methodology.

ℒCS-Hist build algorithm is a four-step algorithm such that each step codifies a certain intelligent data modeling and processing technique, according to the guidelines given in Section 1.

The first step consists in computing an initial, *raw* partition Φ^R of the input data cube A by means of an innovative dynamic programming procedure that follows the widely-accepted criterion of *minimizing the variance* σ^2 of data contained within buckets of the partition. This criterion allows us to generate *close-to-uniform* buckets, which involve in important benefits concerning to the increase of accuracy of retrieved approximate answers (e.g., see [1]). From active literature, recall that among all possible partitions of a given data cube, the *optimal* one is that minimizing the query error due to the data cube compression process. However, *finding the optimal partition is an NP-Hard problem* [15], so that very often *greedy approaches* are adopted (e.g., [16]). In our solution, the adoption of a dynamic programming procedure instead of most-used-in-literature greedy approaches gives us the opportunity of obtaining a *closer-to-optimal* (raw) partition at the cost of a little performance degradation. This because, due to their *local-optimum-focused search policies*, greedy algorithms usually compute sub-optimal solutions only which could not perfectly realize the final goal of obtaining *fair* data cube partitions as meant by any histogram-based data cube compression technique (e.g., see [6]). In our data cube compression framework, the problem of computing the raw partition Φ^R is rigorously formalized and solved in terms of an *Integer Linear Programming* (ILP) *optimization problem*, whose *feasible region* is shaped by taking into account geometrical properties of buckets. Finally, it is worth noticing that the so-computed partition Φ^R is *isomorphic* to the input data cube A, i.e. each bucket B of Φ^R has the same *dimensionality* of A.

Subsequently, in the second step of the algorithm, the initial partition Φ^R is further refined via imposing a set of partitioning constraints, which make *constrained* the final partition we obtain, denoted by Φ^B. Constraints above are meant to significantly reduce the search space of the final problem of finding an optimal partition of the input data cube by progressively refining the actual partition via subsequent steps, thus significantly lowering the overall computational cost needed to compute *ℒCS-Hist*.

In the third step, *ℒCS-Hist* build algorithm tries to further reduce the cardinality of Φ^B by removing *redundant buckets*, namely buckets that store data having distribution similar to the one of other buckets, thus achieving the partition Φ^T. Similarly to previous considerations, reducing the amount of redundant information stored within the histogram (and, as a consequence, the final size of the histogram) is the underlying goal of such an approach. In more detail, *ℒCS-Hist* build algorithm partitions buckets in Φ^B into sets of buckets with *close-to-similar* distributions, and, for each such set, *retains just one bucket* while discarding the remaining ones. The intuition in this case is that retaining just one bucket for each (similar) bucket set, although introducing some approximation, leads, *in most cases*, to a surprising storage space saving. It should be clear from here that, since finding an optimal partition with respect to the *bucket similarity* is a key issue to be considered in order to reduce the introduced overall approximation error, the definition of a proper *similarity metrics* for detecting similar buckets (to be grouped within the same set) plays a critical role. In fact, intuitively enough, the more is the *quality* of the similarity metrics, the less is the introduced overall approximation error, thus the more is the efficiency of the final compressed data structure computed on top of such a partitioned representation. With these ideas in mind, we exploit research results provided by Kamarainen *et al.* in [13], which introduce a *mathematical transformation* for histograms, called *Neighbor-Bank Transform* (NBT). Briefly, NBT allows the performance of traditional similarity metrics, such as *Euclidean*, *Manhattan* and *Minkowsky* metrics, in detecting similar *one-dimensional* histograms to be sensitively improved. As we demonstrate in Section 7, in our work we meaningfully exploit this nice capability of NBT in order to effectively detect similar buckets, thus obtaining the above-highlighted data reduction goals, and deriving benefits.

In the fourth step of the algorithm, the partition Φ^T is finally obtained from the input partition Φ^B by addressing two goals: (*i*) Φ^T must effectively approximate OLAP data stored in the given data cube A, and (*ii*) Φ^T must not contain redundant buckets, i.e. buckets having a certain degree of similarity with other buckets in the partition. Similarly to step two of the algorithm, this defines an ILP problem, which we rigorously formalize and solve as well. In this case, the feasible region is shaped by taking into account (*i*) the so-called *similarity threshold* ϑ, i.e. the maximum value of the similarity metrics according to which two NBT histograms (buckets, respectively) are considered similar, and (*ii*) the maximum size allowed for the in-memory-representation of *ℒCS-Hist*, denoted as G. Therefore, Φ^T is obtained as the *sub-optimal partition* solving the ILP problem above. The immediate benefit we obtain from such an approach is a relevant reduction of the *cardinality* (i.e., the number of buckets) of the actual partition (i.e., $|\Phi^T| << |\Phi^B|$) and, as a consequence, a relevant reduction of the final size of *ℒCS-Hist*, while ensuring a high degree of accuracy of retrieved approximate answers.

5. PARTITIONING DATA CUBES BY DYNAMIC PROGRAMMING

In the first step of the *ℒCS-Hist* build algorithm, we are given a multidimensional data cube A and we aim at obtaining a raw partition Φ^R of A by meaningfully exploiting the statistical

variance of buckets to be generated and stored within Φ^R. The goal is finally obtaining Φ^R in such a way as to minimize the variance of its buckets. To this end, we make use of an innovative dynamic programming procedure, whose main goal is to overcome limitations of traditional histogram-based data cube compression greedy techniques, on the basis of motivations given in Section 4. According to the strategy we propose, based on the well-known dynamic programming paradigm, (*i*) the whole problem (i.e., partitioning the data cube) is divided into a set of sub-problems (characterized next), (*ii*) each sub-problem is solved individually, and (*iii*) the optimal solution of the original (whole) problem is obtained by meaningfully exploiting optimal solutions of sub-problems. In our framework, we characterize a generic sub-problem as the basic problem of *finding the bucket B_ℓ with the lowest variance among a given set of possible ones*. The procedure that computes Φ^R is a three-step procedure, outlined in the following: (s.0) for each data cell C in A, compute the set of all possible buckets of A having C as *vertex*, denoted as $BS(A, C)$; (s.1) determine the *lowest variance bucket set* of A, denoted as $LVBS(A)$, among buckets in the set $BS(A) = \bigcup_{k=0}^{|A|-1} BS(A, C_k)$; ($s$.2) compute $\Phi^R \subseteq LVBS(A)$ via extracting from $LVBS(A)$ the *lowest variance covering bucket set*, namely via extracting from $LVBS(A)$ a set of buckets (i.e., Φ^R) that satisfies the so-called *covering property*. This property imposes that, for each data cell C in A, there exists a bucket B in Φ^R such that B contains C. Among all sub-sets of $LVBS(A)$ that satisfy the covering property, Φ^R is obtained as the one having the *lowest overall statistical variance* $\sigma^2(\Phi^R)$, defined as follows:

$$\sigma^2(\Phi^R) = \sum_{k=0}^{|\Phi^R|-1} \sigma^2(B_k) \tag{3}$$

Let us now focus on each sub-step of our dynamic programming strategy in greater detail. In order to properly treat step (s.0), we need to introduce a simple-yet-necessary data model useful to represent vertex data cells. Given an n-dimensional bucket $B[l_0{:}u_0, l_1{:}u_1, \ldots, l_{n-1}{:}u_{n-1}]$, we associate to each vertex data cell C of B a *binary tuple* called *vertex identifier*, denoted as $V(C) = \langle v_0, v_1, \ldots, v_{n-1}\rangle$, such that: (*i*) $v_i \in \{0, 1\}$, for each i in $\{0, 1, \ldots, n-1\}$; (*ii*) $v_i = 0$ means that $k_i = l_i$, for each i in $\{0, 1, \ldots, n-1\}$; (*iii*) $v_i = 1$ means that $k_i = u_i$, for each i in $\{0, 1, \ldots, n-1\}$. Roughly speaking, given an n-dimensional bucket B, a vertex identifier on B identifies a vertex of B among all possible 2^n vertices.

The task of step (s.0), i.e. computing the set $BS(A, C)$ for a given data cell C of A, is implemented by a procedure that takes as input (*i*) a data cube A, (*ii*) the set D of its dimensions, (*iii*) a data cell C of A, and computes the set $BS(A, C)$ by means of the following steps: (*i*) generate each possible vertex identifier $V(C)$ of C throughout very efficient base-2 arithmetic operations; (*ii*) for each $V(C)$, build the bucket with the *minimum admissible volume*, denoted as $B_{\min}^{C,V(C)}$ (note that this step depends on the kind of vertex identifier $V(C)$); (*iii*) enlarge the bounds of $B_{\min}^{C,V(C)}$ on the basis of the nature of vertex C (in turn, this is again determined by the kind of vertex identifier $V(C)$), until further enlargements are possible, i.e. there exists a dimension d_j of A such that the bounds $[l_j : u_j]$ of $B_{\min}^{C,V(C)}$ along d_j do not satisfy neither $l_j = 0$ nor $u_j = |d_j| - 1$.

Step (s.1) involves the trivial tasks of (*i*) finding, for each data cell C in A, the bucket in $BS(A, C)$ with the lowest statistical variance, and, thereafter, (*ii*) determining the final set $LVBS(A)$ by merging the previous intermediate results. Both tasks are very simple and do not deserve further details.

For what concerns step (s.2), we formalize and solve the problem underlying it as an ILP problem. Let us (*i*) assign an index i to each bucket B in $LVBS(A)$, (*ii*) denote as $I(A)$ the set of all such indices on buckets, (*iii*) assign an index j to each data cell C in A, (*iv*) denote as $J(A)$ the set of all such indices on cells. In addition to this, let us introduce the function $D_{j,i}$, which models whenever a given cell C_j is contained by a given bucket B_i (i.e., $C_j \in B_i$). $D_{j,i}$ is formally defined as follows:

$$D_{j,i} = \begin{cases} 1 & \text{if } C_j \in B_i \\ 0 & \text{otherwise} \end{cases} \tag{4}$$

Furthermore, we introduce $|LVBS(A)|$ *decision variables* x_i, one for each bucket B_i in $LVBS(A)$, defined as follows:

$$x_i = \begin{cases} 1 & \text{if } B_i \text{ in } \Phi^R \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Finally, the ILP formulation of step (1.3) is the following:

$$\begin{cases} \min \sum_{\forall i \in I(A)} \sigma^2(B_i) \cdot x_i & (\ell.0) \\ \sum_{\forall i \in I(A)} D_{j,i} \cdot x_i \geq 1 \quad \forall C_j \in J(A) & (\ell.1) \\ x_i \in \{0,1\} \quad \forall B_i \in I(A) & (\ell.2) \end{cases} \tag{6}$$

wherein: (*i*) function (ℓ.*0*) is the *objective function* to be minimized (the sum of the variances of buckets retained in Φ^R); (*ii*) constraint (ℓ.*1*) imposes that, for each data cell C_j of $J(A)$, there exists at least one bucket B_i in Φ^R such that B_i contains C_j; finally, (*iii*) constraint (ℓ.*2*) imposes that decision variables x_i must be binary-valued (i.e., either 0 or 1).

6. REFINING THE PARTITIONED REPRESENTATION OF DATA CUBES BY PARTITIONING CONSTRAINTS

This Section deals with the second step of the *ℒCS-Hist* build algorithm. Given the so-far-computed raw partitioned representation of A, Φ^R, second step produces in output the refined partition Φ^B by means of imposing the following partitioning constraints: (c.0) no overlapping buckets allowed, (c.1) no nested buckets allowed, (c.2) whenever bucket splitting is needed, use a *splitting policy* still aimed at variance reduction.

Another obvious requirement due to geometrical issues is that buckets in Φ^B must be *convex*. Therefore, before explaining how the refined partition Φ^B is obtained from the raw partition Φ^R, we need to introduce the concept of *bucket convexity*. As explained in Section 3, a data cube A can be thought of as a multidimensional vectorial space $\mathcal{Z}(A)$, each data cell C being a vector $W(C)$ in $\mathcal{Z}(A)$. Under this vision, we define the bucket convexity concept by borrowing the concept of convexity from the *convex set*

theory. Given a multidimensional bucket B, we say that B is convex if the following condition holds:

$$\forall C_i, C_j \in Cells(B) \quad \lambda \cdot C_i \oplus (1-\lambda) \cdot C_j = C_k \in Cells(B), \quad \lambda \in [0,1] \quad (7)$$

where the symbol $\oplus$ denotes the (vectorial) sum among the coordinates of C_i and C_j in their vectorial representation.

Bucket convexity is of interest to the goals of our framework due to the fact that while applying constraints (c.0), (c.1) and (c.2) some buckets in the current partition (i.e., Φ^R) could *temporally* lose the convexity property, and hence they *must be made convex* since buckets in Φ^B must be convex by construction. *Bucket convexification* is therefore obtained by splitting each non-convex bucket B into a set of convex buckets, denoted as $Conv(B)$ and defined as follows:

$$Conv(B) = \{B_i \mid Cells(B_i) \subset B \wedge \bigcup_i B_i = B \wedge \bigcap_i B_i = \varnothing \wedge B_i \text{ is convex}\} \quad (8)$$

With respect to the bucket convexification process, a significant problem appears when, given a non-convex bucket B, *there could exist more than one possible convexification of* B. Let $\mathcal{C}(B)$ denote the set of convexifications of B, and $Conv_j(B)$ denote a generic item of $\mathcal{C}(B)$ (i.e., a generic convexification of B). Due to the constraint (c.2) (i.e., splits must be done based on a variance-reduction-driven policy), in order to make B convex we replace B in Φ^B with that convexification (of B) leading to the *minimum overall variance*, denoted as $MVConv(B)$, and defined as follows:

$$MVConv(B) = \arg\min_{j \in \mathcal{C}(B)} \{\sigma^2(Conv_j(B))\} \quad (9)$$

such that $\sigma^2(Conv_j(B))$ is defined as follows:

$$\sigma^2(Conv_j(B)) = \sum_{B_i \in Conv_j(B)} \sigma^2(B_i) \quad (10)$$

Let us now examine partitioning constraints (c.0) and (c.1), and the way they are applied to Φ^R in order to obtain Φ^B. Consider two *non-disjoint* buckets B_i and B_j, as this case is more interesting than the trivial case in which B_i and B_j are disjointed. In this condition (i.e., non-disjoint buckets), B_i and B_j can violate partitioning constraints (c.0) and (c.1) in (at most) three ways: (a) B_i and B_j are *nested* one within the other, (b) B_i and B_j *coincide*, (c) B_i and B_j *overlap*. We next carefully investigate all these alternatives, and provide solutions to them. Basically, these solutions consist in removing the buckets of the current partition Φ^R that cause constraint violations.

Case (*a*): Nested Buckets. Given two buckets B_i and B_j, B_i is nested within B_j if $Cells(B_i) \subset Cells(B_j)$. Obviously, in the most general case, we could have a singleton bucket B_j containing a *set of nested bucket* $NB(B_j) = \{B_i \mid Cells(B_i) \subset Cells(B_j)\}$. Partitioning constraint (c.0) is imposed by removing from Φ^R buckets in $NB(B_j)$ and retaining just the bucket B_j. It should be noted that this simple method used to remove nested buckets has two major benefits. First, it reduces the overall cardinality of Φ^R and, in turn, the one of Φ^B (note that this involves the amenity of reducing the final space occupancy of *ℒCS-Hist*). Second, it follows the well-accepted-in-OLAP criterion of *aggregating data* as it allows the final partition to store more aggregated data (i.e., the *singleton* bucket B_j) instead of less aggregated ones (i.e., buckets in $NB(B_j)$). Figure 1 shows an example on how nested buckets are removed in order to satisfy the partitioning constraint (c.0). Specifically, in Figure 1 nested buckets B_1, B_2, B_3, and B_4 are removed from Φ^R.

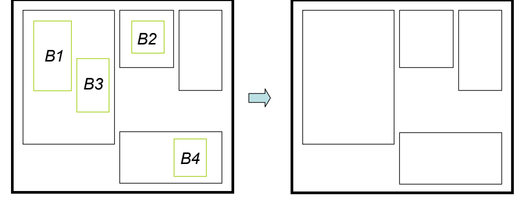

Figure 1. Removing nested buckets.

Case (*b*): Identical Buckets. Two buckets B_i and B_j *coincide* if the following two conditions hold: (i) $Cells(B_i) \subseteq Cells(B_j)$, and ($ii$) $Cells(B_j) \subseteq Cells(B_i)$. In this case, partitioning constraint (c.0) is imposed by simply removing from Φ^R one bucket between B_i and B_j. It does not matter which bucket is removed since B_i and B_j cover the same area exactly, and have the same statistical properties exactly.

Case (*c*): Overlapping Buckets. Two buckets B_i and B_j *overlap* if the following conditions hold:

$$\begin{array}{c} (i)\ Cells(B_i) \cap Cells(B_j) \neq Cells(B_i) \\ (ii)\ Cells(B_i) \cap Cells(B_j) \neq Cells(B_j) \\ (iii)\ Cells(B_i) \cap Cells(B_j) \neq \varnothing \end{array} \quad (11)$$

Obviously, it could be the case that there exist in Φ^R more than just two overlapping buckets, but this case can be meaningfully treated by decomposition, thus considering only a pair of overlapping buckets at time. Therefore, in the following we will restrict our analysis to the case of having two overlapping buckets B_i and B_j, and we will denote as $B_k = B_i \cap B_j$ the intersection area, or, equally, the intersection bucket.

With respect to the overall variance reduction criterion, by comparing statistical variances of the involved buckets (i.e., B_i, B_j, and B_k), three cases can occur at this point. We carefully investigate these cases in the following.

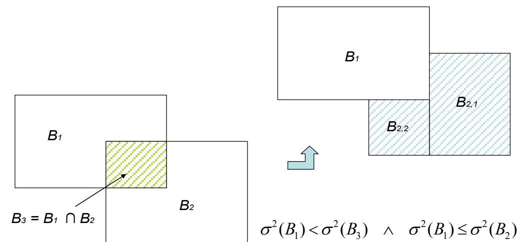

Figure 2. Removing overlapping buckets – case (*c*.1).

Case (*c*.1): $\sigma^2(B_i) < \sigma^2(B_k) \wedge \sigma^2(B_i) \leq \sigma^2(B_j)$. In this case, in order to obtain an overall variance reduction within Φ^B (as imposed by constraint (c.2)), we maintain B_i and cut from B_j the area

corresponding to B_k, thus removing the overlapping area. Since the area $B_j - B_i$ may not be convex, we apply the *minimum variance convexification* by replacing B_j with $MVConv(B_j - B_i)$. As an example, Figure 2 shows the procedure above on buckets B_1, B_2, and B_3, such that $MVConv(B_2 - B_1) = \{B_{2,1}, B_{2,2}\}$.

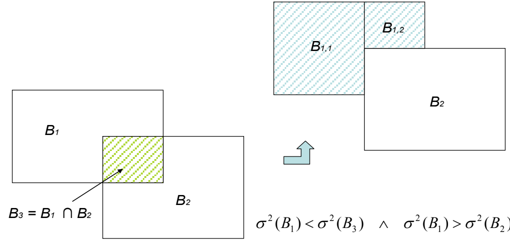

Figure 3. Removing overlapping buckets – case (*c*.2).

Case (*c*.2): $\sigma^2(B_i) < \sigma^2(B_k) \wedge \sigma^2(B_i) > \sigma^2(B_j)$ (i.e., $\sigma^2(B_j) < \sigma^2(B_i) < \sigma^2(B_k)$). In this case, in order to obtain an overall variance reduction within Φ^B, we (*i*) maintain B_j (since it covers the area with the lowest variance), (*ii*) cut the area corresponding to B_k from bucket B_i, and (*iii*) replace B_i with $MVConv(B_i - B_k)$. Figure 3 shows the procedure above on buckets B_1, B_2, and B_3, such that $MVConv(B_1 - B_3) = \{B_{1,1}, B_{1,2}\}$.

Case (*c*.3): $\sigma^2(B_k) < \sigma^2(B_i) \wedge \sigma^2(B_k) > \sigma^2(B_j)$. In this case, in order to achieve an overall variance reduction within Φ^B, we originate a *new* bucket from B_k and cut from both B_i and B_j the area corresponding to B_k, thus eliminating the overlapping area. Buckets B_i and B_j are then replaced by $MVConv(B_i)$ and $MVConv(B_j)$, respectively. To give an example, consider Figure 4, where, given three buckets B_1, B_2, and B_3, it is assumed that $MVConv(B_1 - B_3) = \{B_{1,1}, B_{1,2}\}$ and $MVConv(B_2 - B_3) = \{B_{2,1}, B_{2,2}\}$. Applying the strategy above, we finally obtain five new buckets belonging to the partition Φ^B (see Figure 4).

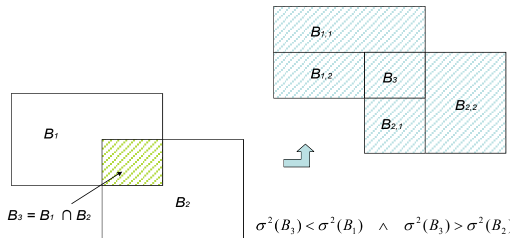

Figure 4. Removing overlapping buckets – case (*c*.3).

From the analysis above, it clearly follows that, due to geometrical issues, there could be settings for which new buckets are originated and, as a consequence, the final cardinality of Φ^B is greater than the one of Φ^R (i.e., $|\Phi^B| > |\Phi^R|$), which would lead to an increase of the final size of *£CS-Hist*. Nevertheless, this effect is effectively contrasted by the third step of the methodology we propose, where redundant buckets are discarded (see Section 4).

7. EXPLOITING SIMILARITY METRICS ON ONE-DIMENSIONAL HISTOGRAMS TO DISCARD REDUNDANT BUCKETS

As we discussed in Section 4, in step three of the *£CS-Hist* build algorithm our approach aims at minimizing the number of buckets storing similar information. To this end, a critical issue is represented by the problem of choosing an effective similarity metrics capable of properly detecting similar buckets. In this Section, we will present the similarity metrics we introduce in our framework. This formally corresponds to define a procedure for computing the *similarity function* $\theta : \Im \times \Im \longrightarrow \Re$, such that (*i*) $\Im$ denotes a set of multidimensional buckets, and (*ii*) $\Re$ denotes the set of real numbers.

As highlighted in Section 4, in order to achieve this goal we efficiently exploit the NBT proposed by Kamarainen *et al.* [13]. [13] experimentally demonstrates that performance of traditional similarity metrics on one-dimensional histograms is poor, due to the "difficult nature" of such data structures. To overcome this limitation, [13] proposes applying NBT to the input histograms before using any (traditional) similarity metrics over them. Given two histograms H_i and H_j, and a one-dimensional histogram similarity metrics m, the NBT-transformed histograms H_i^T and H_j^T, respectively, allow us to achieve a *finer evaluation* of m over H_i and H_j, denoted as $m(H_i^T, H_j^T)$, rather than the one provided by the *same* metrics applied to the original histograms, denoted as $m(H_i, H_j)$. Thanks to this nice property, based on NBT we can detect similar histograms better than using traditional metrics directly.

Nevertheless, since NBT is suitable for one-dimensional histograms only, whereas buckets in Φ^B are multidimensional in nature, in order to effectively exploit benefits coming from NBT, we perform the following tasks: (*s*.0) for each bucket B_i in Φ^B, build the so-called *linearized set* $\Psi(B_i)$ (i.e., a one-dimensional representation of B_i) by sorting data cells in $Cells(B_i)$ using an *innovative locality-preserving linearization technique*; (*s*.1) for each so-generated set $\Psi(B_i)$, build the one-dimensional histogram $H(\Psi(B_i))$ associated to B_i; (*s*.2) for each so-generated histogram $H(\Psi(B_i))$, apply NBT to $H(\Psi(B_i))$, thus obtaining the transformed histogram $H^T(\Psi(B_i))$; finally, (*s*.3) for each pair of NBT-transformed histograms $H^T(\Psi(B_i))$ and $H^T(\Psi(B_j))$ built from buckets B_i and B_j, respectively, apply the *Mahalanobis similarity metrics* [2], denoted as $m(H^T(\Psi(B_i)), H^T(\Psi(B_j)))$. Note that the above-listed tasks finally implement the procedure for computing the similarity function θ.

According to the procedure above, in order to detect if a pair of buckets B_i and B_j belonging to Φ^B are similar, so that one can be discarded in favor of the other one, the Mahalanobis metrics on the corresponding pair of NBT histograms $H^T(\Psi(B_i))$ and $H^T(\Psi(B_j))$ is considered. Among all the similarity metrics available in literature, we choose to adopt the Mahalanobis metrics due to the following motivations: (*i*) it is *statistical in nature* and specialized for *correlated data*, thus suitable for OLAP data [5]; (*ii*) it can be easily implemented within a dedicated software component, being based on conventional

matrix algebra operations; (*iii*) according to [13], compared with traditional metrics, it also gives the *best* performance on correlated data, turning to be perfect for OLAP.

Let us now carefully investigate each step of the procedure for computing θ (i.e., (s.0), (s.1), (s.2), and (s.3)) in a greater detail. Step (s.0) aims at computing the linearized set $\Psi(B_i)$ from a given multidimensional bucket B_i by means of a locality-preserving linearization technique, which is inspired by the work of Hamilton and Rau-Chaplin [11]. This technique, which plays a leading role in our framework, is composed by the following tasks: (*i*) perform a *coordinate transformation* in order to express the set of coordinates of each data cell C within B_i in a set of coordinates that is *attached to* B_i (i.e., a set of coordinates of C defined with respect to the B_i's coordinate space rather than the coordinate space of the entire data cube A); (*ii*) for each data cell C, compute its *Compact Hilbert Index* (CHI) [11] (described next), denoted as $\Omega(C)$; (*iii*) sort data cells in $Cells(B_i)$ by their CHIs.

In more detail, the main idea behind the construction of $\Psi(B_i)$ is to exploit both *Hilbert curves* and their relevant improvements proposed in [11]. Briefly, Hilbert curves are *continuous self-similar functions* that provide a mapping between a multidimensional set and a one-dimensional interval (thus providing a *linear ordering* of points in the multidimensional set), and having the amenity of preserving *data locality*. This property states that data that are "near" to each other in the original multidimensional set are "close" one to another also in the resulting linear ordering given by the Hilbert-curves-based mapping. On top of this data locality property, [11] makes two interesting contributions. First one consists in an extension to standard Hilbert curves able to effectively treat the case in which dimensions of the original multidimensional space have *different cardinalities*. Is should be noted that this specific feature is very useful in several application scenarios, but particularly in OLAP, where this case occurs very often. To give an example, consider a data cube storing sales data. Here, we can have a dimension, say *ProductID*, with thousands of members, and another dimension, say *Gender*, with just two members. The second contribution consists in an efficient algorithm, called `compHilbertIndex`, which, given a point P in a multidimensional space, based on simple base-2 arithmetic operations, computes the CHI $\Omega(P)$, i.e. an integer-valued binary label indicating the position of P in the linear ordering imposed by Hilbert curves. `compHilbertIndex` takes as input (*i*) the number of dimensions n, (*ii*) the cardinality of each dimension $|d_j|$, with j in $\{0, 1, \ldots, n-1\}$, and (*iii*) the set of coordinates identifying P with respect to the coordinate system of the input n-dimensional space. In our approach, in order to meaningfully exploit `compHilbertIndex`, for each bucket B_i in Φ^B, we apply the following *coordinate transformation*, denoted as $\mathcal{T}(\bullet)$, to each data cell $C[k_0, k_1, \ldots, k_{n-1}]$ in $Cells(B_i)$:

$$k'_j = k_j - l_j,\ for\ each\ j\ in\ \{0,1,\ldots,n-1\} \tag{12}$$

This allows us to obtain the transformed set of coordinates $\{k'_0, k'_1, \ldots, k'_{n-1}\}$ of C with respect to the B_i's coordinate space. After this transformation, we can safely apply `compHilbertIndex` to C, thus generating the CHI $\Omega(C)$. By exploiting this task as baseline operation, the linearized set $\Psi(B_i)$ of B_i is finally computed by sorting *each* data cell C in $Cells(B_i)$ by its CHI $\Omega(C)$. This gives raise to the following formal definition:

$$\begin{aligned}\Psi(B_i) = \{\ \mathcal{I} \mid \mathcal{I} = {} & \Omega(C[k'_0, k'_1, \ldots, k'_{n-1}]) = \\ & \texttt{compHilbertIndex}(\ n, |d_0|, |d_1|, \ldots, |d_{n-1}|, \\ & \{k'_0, k'_1, \ldots, k'_{n-1}\}) \wedge \\ & C[k_0, k_1, \ldots, k_{n-1}] \in Cells(B_i) \wedge \\ & \{k'_0, k'_1, \ldots, k'_{n-1}\} = \mathcal{T}(\{k_0, k_1, \ldots, k_{n-1}\}) \wedge \\ & C[k'_0, k'_1, \ldots, k'_{n-1}] \in B \wedge \\ & \Omega(C_l) \prec \Omega(C_p) \text{ if } C_l \prec C_p \text{ with } l < p\ \}\end{aligned} \tag{13}$$

such that $\prec$ denotes the ordering relation between two items (i.e., $C_l \prec C_p$ means that C_l *precedes* C_p in the target ordering). As a clarifying example, Figure 5 shows the locality-preserving linearization technique applied to the two-dimensional bucket B[3:6; 18:21].

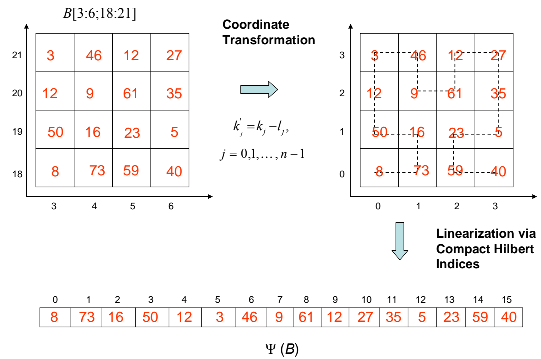

Figure 5. Locality-preserving linearization of a 2D bucket.

Step (s.1) builds the one-dimensional histogram $H(\Psi(B_i))$ by grouping adjacent items in $\Psi(B_i)$. Determining the number of buckets to be allocated for each $H(\Psi(B_i))$ is a non-trivial engagement, so that it is not possible to give valid-for-all guidelines on which range this parameter should assume. This because it depends on several factors such as the particular data distribution and the cardinality of $\Psi(B_i)$. Regardless of this, even due to query efficiency purposes (e.g., [14]), the sole property to be necessarily satisfied *in a global fashion* is that all the histograms $H(\Psi(B_i))$ *have the same cardinality* N (i.e., the same number of buckets). Despite of the different sizes of buckets in Φ^B, this requirement can be easily fulfilled by fixing N to an appropriate value, and setting the *class interval* of each histogram $H(\Psi(B_i))$ accordingly. Recall that, given a histogram H on a data domain D, the class interval of H is defined as the number of adjacent items of D to be grouped within the *same* bucket B in H. This way, the desired number of buckets N can be finally obtained.

Step (s.2) consists in applying NBT to each histogram $H(\Psi(B_i))$, thus obtaining the NBT-transformed histogram $H^T(\Psi(B_i))$. Specifically, NBT is obtained by projecting $H(\Psi(B_i))$ onto a *fewer* cardinality set of *neighbor-banks* by means of a *linear*

transformation, in order to enhance the performance of traditional similarity metrics applied to *ordered histograms* (i.e., histograms such that adjacent buckets represents *highly correlated* data). It should be noted how requirements of NBT are perfectly met by our histograms $H(\Psi(B_i))$, because of two main reasons. First, the ordered histogram assumption holds for histograms $H(\Psi(B_i))$ since: (*i*) they are built on top of OLAP data cubes (by definition, adjacent cells in OLAP data cubes store strongly correlated data [5]); (*ii*) they are obtained via the locality-preserving linearization technique (step (s.1)) that has the amenity of preserving the correlation of data *even* in the linearized set $\Psi(B_i)$; (*iii*) they are built by grouping together adjacent (correlated) items of $\Psi(B_i)$; (*vi*) more generally, partitions Φ^R and Φ^B are computed by procedures aiming at reducing the overall statistical variance, thus overall inducting a greater correlation into data. The second reason of the suitability of our framework to the NBT's requirements is that, as argued in [13], NBT is particularly suitable for *sparse data sets*, and OLAP data, beyond highly correlated are also, typically, sparse [5].

Projection performed by NBT is done by means of a set of cos^2 functions while ensuring that (*i*) the sum of all banks is equal to 1, and (*ii*) each bank is *equi-weighted* [13]. More formally, let (*i*) N_b be an input parameter modeling the number of banks, and (*ii*) i be an index ranging from 0 to $N_b - 1$, the i-th bank b_i, for each i in $\{0, 1, \ldots, N_b - 1\}$, is defined as follows:

$$b_i(x) = \begin{cases} \cos^2\left(\pi \cdot \left(\frac{x}{N}\frac{N_b - 1}{2} + \frac{i}{2}\right)\right) & x \in S_i \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(14)}$$

such that (*i*) N is the desired cardinality of histograms, and (*ii*) S_i is defined as follows:

$$S_i = \left\{ x \mid (i-1)\cdot\frac{N}{N_b - 1} \le x \le (i+1)\cdot\frac{N}{N_b - 1} \right\} \quad \textbf{(15)}$$

Finally, in step (s.3), in order to obtain a numeric value that measures the degree of similarity between two buckets B_i and B_j, we apply the Mahalanobis metrics to their NBT-transformed histogram-based representations, $H^T(\Psi(B_i))$ and $H^T(\Psi(B_j))$, respectively (denoted hereinafter as $\tilde{H}_i$ and $\tilde{H}_j$, respectively). Given $\tilde{H}_i$ and $\tilde{H}_j$, the Mahalanobis metrics $m(\tilde{H}_i, \tilde{H}_j)$ is defined as follows [2]:

$$m\left(\tilde{H}_i, \tilde{H}_j\right) = \left(\tilde{H}_i - \tilde{H}_j\right)^{\Theta} \cdot \Lambda^{-1}_{\tilde{H}_i,\tilde{H}_j} \cdot \left(\tilde{H}_i - \tilde{H}_j\right) \quad \textbf{(16)}$$

where: (*i*) Θ denotes the *matrix transposition operator*, and (*ii*) $\Lambda_{\tilde{H}_i,\tilde{H}_j}$ denotes the *covariance matrix* between $\tilde{H}_i$ and $\tilde{H}_j$, which, in turn, is defined as follows:

$$\Lambda_{\tilde{H}_i,\tilde{H}_j} = \begin{bmatrix} E\left[\left(\tilde{H}_i[1]-\mu(B_i)\right)\cdot\left(\tilde{H}_j[1]-\mu(B_j)\right)\right] & \cdots & E\left[\left(\tilde{H}_i[1]-\mu(B_i)\right)\cdot\left(\tilde{H}_j[N]-\mu(B_j)\right)\right] \\ \vdots & \ddots & \vdots \\ E\left[\left(\tilde{H}_i[N]-\mu(B_i)\right)\cdot\left(\tilde{H}_j[1]-\mu(B_j)\right)\right] & \cdots & E\left[\left(\tilde{H}_i[N]-\mu(B_i)\right)\cdot\left(\tilde{H}_j[N]-\mu(B_j)\right)\right] \end{bmatrix} \quad \textbf{(17)}$$

such that: (*i*) $E(f(x))$ is the *expected value* of the function $f(x)$, defined as: $E(f(x)) = \sum_x f(x)\cdot P(x)$, where $P(x)$ is a given *probability density function*; (*ii*) $\tilde{H}_i[l]$ denotes the value associated with the l-th bucket of $\tilde{H}_i$; (*iii*) $\mu(B_i)$ denotes the statistical mean of B_i (see Section 3).

8. COMPUTING THE FINAL PARTITIONED REPRESENTATION OF DATA CUBES

In the fourth step, the *£CS-Hist* build algorithm computes the final partition Φ^T by extracting from Φ^B a sub-set of buckets having the following properties: (*i*) buckets in Φ^B storing redundant (i.e., *too similar*) information are not retained in Φ^T; (*ii*) if a bucket B_i in Φ^B is not retained in Φ^T, there must exist a bucket B_j in Φ^T such that $\theta(B_i, B_j) \le \vartheta$, being θ the similarity function (see Section 7) and ϑ the similarity threshold (see Section 4); (*iii*) the size of Φ^T must not exceed the given space budget G, i.e. the total amount of storage space available for housing *£CS-Hist*.

As highlighted in Section 4, in our framework the problem underlying step four is formulated and solved in the vest of an ILP problem, similarly to what done with the problem of computing the partition Φ^R of the input data cube A addressed in the step one of our algorithm (see Section 5).

Let us (*i*) assign an index i to each bucket B_i in Φ^B, and (*ii*) denote as $I(\Phi^B)$ the set of all such indices on buckets. Furthermore, we introduce $|\Phi^B|$ *decision variable* x_i, one variable for each bucket B_i in Φ^B, defined as follows:

$$x_i = \begin{cases} 1 & \text{if } B_i \text{ in } \Phi^T \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(18)}$$

and $|\Phi^B| \times |\Phi^B|$ coefficients $\Theta_{i,j}$, defined as follows:

$$\Theta_{i,j} = \begin{cases} 1 & \text{if } \theta(B_i, B_j) \le \vartheta \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(19)}$$

B_i and B_j being the i-th and j-th buckets of Φ^B, respectively. Finally, let S_i denote the storage space needed for housing the bucket B_i.

Given these theoretical tools, the problem underlying step four of the *£CS-Hist* build algorithm can be formulated and solved as follows:

$$\begin{cases} \min \sum_{i \in I(\Phi^B)} \sigma^2(B_i)\cdot x_i & (\ell.0) \\ \sum_{i \in I(\Phi^B)} S_i \cdot x_i \le G & (\ell.1) \\ \sum_{i \in I(\Phi^B)} \Theta_{i,j} \cdot x_i \ge 1 \quad \forall j \in I(\Phi^B) & (\ell.2) \\ x_i \in \{0,1\} \quad \forall i \in I(\Phi^B) & (\ell.3) \end{cases} \quad \textbf{(20)}$$

wherein: (*i*) function (ℓ.*0*) is the *objective function* to be minimized (the sum of the variances of buckets retained in Φ^T); (*ii*) constraint (ℓ.*1*) imposes that Φ^T does not exceed the given space budget G; (*iii*) constraint (ℓ.*2*) imposes that, among all buckets that are similar one to another, at least one bucket is

retained in Φ^T; finally, (*iv*) constraint ($\ell.3$) imposes that decision variables x_i must be binary-valued (i.e., either 0 or 1).

It should be noted that, in order to have a *non-empty* feasible region for the above ILP problem, the value of the parameter G set as input to instances of such problem must belong to a proper *validity range*. Without any loss of generality, we highlight that there exists a correlation between this value and the value of the similarity threshold ϑ. In particular, if we lower ϑ with the aim of refining the similarity detection process and consider two buckets to be similar if they have a *high* degree of similarity, then the value of G must belong to a range whose bounds have "big" values. On the contrary, if we relax this assumption and consider a more *soft* similarity condition, i.e. we higher ϑ with the aim of considering two buckets similar if they have a *low* degree of similarity, then the value of G must belong to a range whose bounds have "small" values.

Finally, it is clear that computing the histogram *ℒCS-Hist* is a resource-intensive task. The counterpart to this aspect is represented by a higher capability of effectively and efficiently compressing massive high-dimensional data cubes with respect to traditional histogram-based data cube compression approaches, at a provable degree of accuracy of retrieved approximate answers. However, as highlighted throughout the paper, *ℒCS-Hist* is built in an off-line manner, similarly to other data cube maintenance (and update) operations occurring in conventional OLAP server platforms, thus determining a transparent-for-the-application process. The complexity analysis of the *ℒCS-Hist* build algorithm would merit a specialized research effort, which is outside the scope of this paper. However, to give insights, the most resource-intensive part of the algorithm is represented by the first step, where the initial raw partition of the input data cube is computed by means of a dynamic programming procedure.

9. EXPERIMENTAL RESULTS

In order to test the performance of our proposed compression technique, we conducted an extensive series of experiments on several classes of data cubes. Ranging the input parameters (such as dimension number, *sparseness coefficient* etc) is the major benefit coming from using synthetic data cubes instead of real-life ones. On the other hand, experiments on real-life data sets are useful to complete the overall assessment of any data-intensive technique like ours. In our experience, experiments on real-life data cubes have exposed observations similar to those of synthetic ones, so that in this Section we present experimental results on synthetic data cubes. Beyond space limitation, this choice is, above all, motivated by the fact that, as highlighted in Section 1, in our research we consider the class of complex histogram-based data cube compression techniques, so that, being more "customizable", synthetic data cubes are more suitable to stress the performance of (complex) techniques depending on several building parameters rather than real-life ones that, on the contrary, cannot be customized easily.

Specifically, we engineered two kinds of synthetic data cubes: *CVA* and *SKEW*. In the first kind of data cubes, data cells are generated according to a Uniform distribution defined on a given range $[L_{min}, L_{max}]$, with $L_{min} < L_{max}$. In other words, for such data cubes the *Continuous Value Assumption* (CVA) [5], which assumes that data cells are uniformly distributed over the target domain, holds. On the contrary, in the second kind of data cubes, data cells are generated according to a Zipf distribution defined on a given parameter z, with z in [0, 1]. Furthermore, in both kinds of data cubes, the dimension number n is an input parameter that allows us to obtain different data cube "instances" having different values of dimensionality and size. Finally, to obtain close-to-real-life data cubes, we imposed a sparseness coefficient s equal to around 0.001, which is a widely-accepted setting for similar experimental experiences in approximate query answering against compressed data cubes (e.g., see [7]).

As comparison, according to motivations given in Section 1, we chosen the following well-know-in-literature approximate query answering techniques: *Min-Skew* [1], *GenHist* [10] and *STHoles* [3]. To compare performance of *ℒCS-Hist* against those of comparison methods on a common basis, we imposed that all the techniques have the *same* space budget G for generating their own compressed representations of the target data cube. This aspect of our experimental analysis has been modeled by the *compression ratio* r, which is defined as follows:

$$r = \frac{size(H)}{size(A)} \quad \textbf{(21)}$$

such that: (*i*) $size(H)$ is the size of the compressed representation H, and (*ii*) $size(A)$ is the size of the target data cube A. For instance, r equal to 10 %, is widely recognized as a "reasonable" value for such kind of experiments (e.g., see [7]). Therefore, in our experimental analysis we set r to such a reference threshold. Furthermore, for each comparison technique, we tried our best to set the configuration of input parameters that respective authors consider the best in their papers, thus ensuring a *fair* experimental analysis.

In our first kind of experiments, in order to study the *accuracy* of comparison techniques against *average-sized* data cubes, we considered the following data cubes: (*i*) CVA_{15}, which is a fifteen-dimensional CVA data cube defined on the range [40, 60], and occupying around 1.8 GB disk space; (*ii*) $SKEW_{15}$, which is a fifteen-dimensional SKEW data cube defined on the parameter z = 0.5, and occupying around 1.9 GB disk space; (*iii*) CVA_{20}, which is a twenty-dimensional CVA data cube defined on the range [70, 90], and occupying around 2.2 GB disk space; (*iv*) $SKEW_{20}$, which is a twenty-dimensional SKEW data cube defined on the parameter z = 0.9, and occupying around 2.4 GB disk space.

As regards the input of our experimental framework, we engineered the *query population* $Q_S(A, v)$ which is composed by *all* the multidimensional range-SUM queries Q that can be defined on A by varying the query dimensional ranges and having a *query selectivity* $\|Q\|$ equal to the v % of the entire volume of A, being v an integer parameter ranging on the interval [10, 100]. Formally, $Q_S(A, v)$ is defined as follows:

$$Q_S(A, v) = \{Q \mid Q \subseteq A \wedge \|Q\| = \frac{v}{100} \cdot \|A\| \wedge v \in [10, 100]\} \quad \textbf{(22)}$$

As regards metrics, we considered the *accuracy metrics* defined by the *Average Relative Error* ε_{rel}, whose definition is the following:

$$\varepsilon_{rel} = \frac{1}{|Q_S(A,v)|} \cdot \sum_{k=0}^{|Q_S(A,v)|-1} \frac{|A(Q_k) - \widetilde{A}(Q_k)|}{\max\{1, \widetilde{A}(Q_k)\}} \quad \textbf{(23)}$$

such that: (*i*) $|Q_S(A, v)|$ denotes the cardinality of $Q_S(v)$; (*ii*) Q_k denotes the generic query belonging to $Q_S(v)$; (*iii*) $A(Q_k)$ denotes the exact answer to Q_k (i.e., the answer to Q_k evaluated against A); (*iv*) $\widetilde{A}(Q_k)$ denotes the approximate answer to Q_k (i.e., the answer to Q_k evaluated against the compressed representation H).

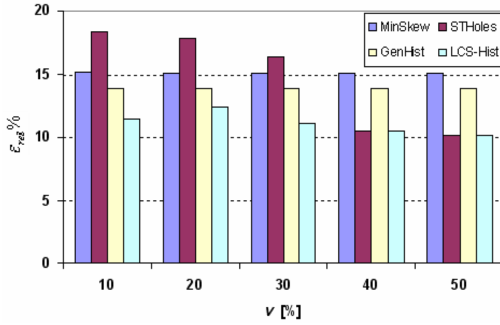

Figure 6. Percentage variation of ε_{rel} w.r.t. ||Q|| on data cube CVA_{15}.

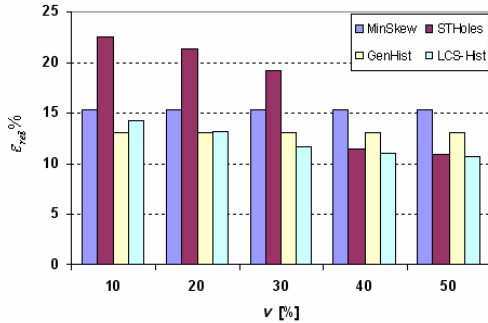

Figure 7. Percentage variation of ε_{rel} w.r.t. ||Q|| on data cube $SKEW_{15}$.

Experimental results presented in Figures 6-9 show that, with respect to the accuracy metrics, our proposed technique has generally performance comparable to those provided by the other comparison techniques, and better in some cases.

In order to study the *scalability* and the *space efficiency* of our proposed technique, which are both two critical factors when dealing with massive high-dimensional data cubes, we carried out a second series of experiments on both types of synthetic data cubes (i.e., CVA and SKEW), still keeping *r* fixed to 10%, but ranging now the dimension number *n* of data cubes on the interval [15, 35]. This leads to the definition of the data cube classes CVA_n and $SKEW_n$, which differ from the previous classes (i.e., CVA and SKEW) for the fact that now the number of dimensions is an input parameter. On top of such classes, we obtained several data cube instances having a dimensionality *higher* than the one considered in the previous series of experiments. As regards the input query set, in order to take into account scalability and efficiency issues we considered a slightly modified version of the query population (22) in such a way as to generate a sub-set of queries against *A* *larger* than the one considered previously. This gives raise to the query population $Q_S^*(A, \rho, \varphi)$, which is defined as follows:

$$Q_S^*(A, \rho, \varphi) = \bigcup_{\ell=0}^{\rho} Q_s(A, \varphi \cdot \ell + \varphi) \quad \textbf{(24)}$$

such that ρ and φ are again input parameters.

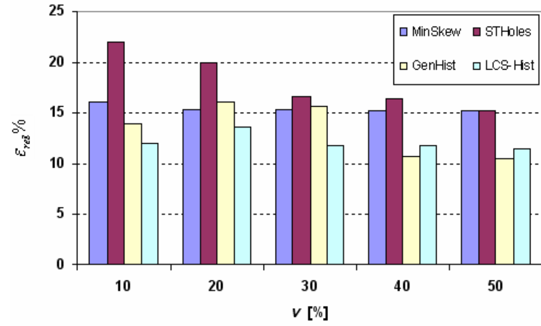

Figure 8. Percentage variation of ε_{rel} w.r.t. ||Q|| on data cube CVA_{20}.

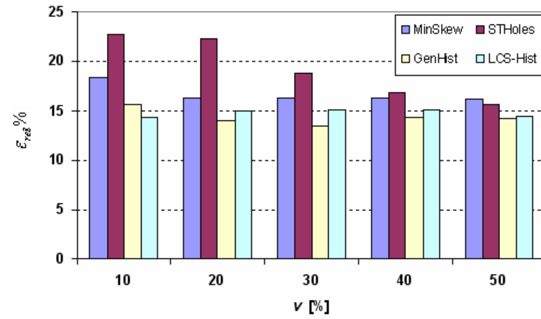

Figure 9. Percentage variation of ε_{rel} w.r.t. ||Q|| on data cube $SKEW_{20}$.

In the second series of experiments, we fixed this query population, and varied the dimension number *n* of data cubes. From (24), the accuracy metrics (23) is slightly modified accordingly, as follows:

$$\varepsilon_{rel}^* = \frac{1}{|Q_S^*(A,\rho,\varphi)|} \cdot \sum_{k=0}^{|Q_S^*(A,\rho,\varphi)|-1} \frac{|A(Q_k) - \widetilde{A}(Q_k)|}{\max\{1, \widetilde{A}(Q_k)\}} \quad \textbf{(25)}$$

Experiments belonging to the second series are intended to compare the growth-in-space-complexity of our technique against the one of comparison ones, and in dependence on the increase of the number of dimensions of the target date cube. Indeed, from Figures 10-11 we observe that as the dimension number increases, the average approximation error of our technique becomes significantly smaller than the one of comparison techniques. Basically, this is due to the *effectiveness of our bucket similarity detection technique* that finally produces a *substantial space saving* by discarding similar buckets from the final partitioned representation of the input data cube. The saved space can

consequently be used to *better represent* (i.e., using a *greater level of detail*) *worse-approximated regions* of the data cube, thus obtaining a better "global" degree of approximation with respect to traditional approaches. This amenity ensured by *ℒCS-Hist* finally involves in a better scalability and a better efficiency on increasing-in-size-and-dimensionality data cubes, which are very popular in next-generation DW and BI systems.

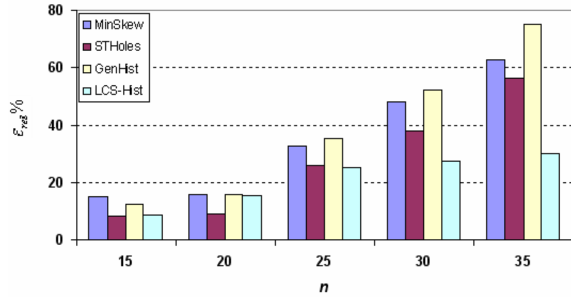

Figure 10. Percentage variation of ε^*_{rel} w.r.t. ||*Q*|| on data cube *CVA*$_n$ with ρ = 5 and φ = 10.

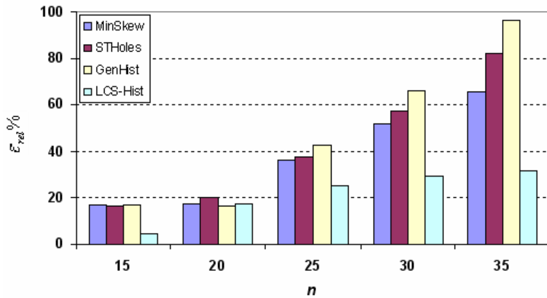

Figure 11. Percentage variation of ε^*_{rel} w.r.t. ||*Q*|| on data cube *SKEW*$_n$ with ρ = 5 and φ = 10.

10. CONCLUSIONS AND FUTURE WORK

Starting from well-known limitations of traditional histogram-based data cube compression techniques on massive high-dimensional data cubes, in this paper we have presented a complex methodology that combines intelligent data modeling and processing techniques in order to tame the compression of such kind of data cubes, which are very popular in next-generation DW and BI systems.

We have formally presented all the tasks of the proposed methodology, along with theoretical formalizations of main problems arising in this approach. The result is represented by a complete framework that encompasses several points of innovation in the OLAP data cube compression research field, and the novel multidimensional histogram *ℒCS-Hist*. Another contribution of our work has been represented by a comprehensive experimental evaluation of accuracy, scalability and space efficiency of our technique in comparison with similar approaches. This evaluation has clearly demonstrated the benefits coming from applying *ℒCS-Hist* to DW and BI contexts characterized by massive sizes and high dimension numbers.

Future work is mainly focused on multiple directions: (*i*) devising and testing different partitioning constraints, e.g. what happens if nested buckets are allowed? (*ii*) investigating the problem of efficiently handling data updates that can occur in the relational data sources alimenting the target data cube; (*iii*) embedding within the proposed framework *probabilistic guarantees* over the degree of approximation of the retrieved answers, following approximate query answering paradigms appeared in literature recently (e.g., [9]).

11. REFERENCES

[1] Acharya, S., Poosala, V., and Ramaswamy, S. 1999. Selectivity Estimation in Spatial Databases. In *ACM SIGMOD*, 13-24.

[2] Anderson, T.W. 1958. *Introduction to Multivariate Statistical Analysis*, Wiley.

[3] Bruno, N., Chaudhuri, S., and Gravano, L. 2001. STHoles: A Multidimensional Workload-Aware Histogram. In *ACM SIGMOD*, 211-222.

[4] Chaudhuri, S., Das, G., and Narasayya, V.R. 2007. Optimized Stratified Sampling for Approximate Query Processing. *ACM Transactions on Database Systems 32(2)*, 9.

[5] Colliat, G. 1996. OLAP, Relational, and Multidimensional Database Systems. *SIGMOD Record 25(3)*, 64-69.

[6] Cuzzocrea, A. 2005. Overcoming Limitations of Approximate Query Answering in OLAP. In *IEEE IDEAS*, 200-209.

[7] Cuzzocrea, A. and Wang, W. 2007. Approximate Range-Sum Query Answering with Probabilistic Guarantees. *Journal of Intelligent Information Systems 28(2)*, 161-197.

[8] Dehne, F., Eavis, T., and Rau-Chaplin, A. 2004. The cgmCUBE Project: Optimizing Parallel Data Cube Generation for ROLAP. *Distributed and Parallel Databases 19(1)*, 29-62.

[9] Garofalakis, M.N. and Kumar, A. 2004. Deterministic Wavelet Thresholding for Maximum-Error Metrics. In *ACM PODS*, 166-176.

[10] Gunopulos, D., Kollios, G., Tsotras, V.J., and Domeniconi, C. 2000. Approximating Multi-Dimensional Aggregate Range Queries over Real Attributes. In *ACM SIGMOD*, 463-474.

[11] Hamilton, C.H. and Rau-Chaplin, A. 2007. Compact Hilbert Indices for Multidimensional Data. In *IEEE CISIS*, 139-146.

[12] Jermaine, C.M., Arumugam, S., Pol, A., and Dobra, A. 2007. Scalable Approximate Query Processing with the DBO Engine. In *ACM SIGMOD*, 725-736.

[13] Kamarainen, J.-K., Kyrki, V., Ilonen, J., and Kälviäinen, H. 2003. Improving Similarity Measures of Histograms Using Smoothing Projections. *Pattern Recognition Letters 24(12)*, 2009-2019.

[14] Muralikrishna, M. and DeWitt, D.J. 1988. Equi-Depth Histograms for Estimating Selectivity Factors for Multi-Dimensional Queries. In *ACM SIGMOD*, 28-36.

[15] Muthukrishnan, S., Poosala, V., and Suel, T. 1999. On Rectangular Partitioning in Two Dimensions: Algorithms, Complexity, and Applications. In *ICDT*, 236-256.

[16] Poosala, V. and Ioannidis, Y.E. 1997. Selectivity Estimation without the Attribute Value Independence Assumption. In *VLDB*, 486-495.

[17] Sismanis, Y., Deligiannakis, A., Roussopoulos, N., and Kotidis, Y. 2002. Dwarf: Shrinking the PetaCube. In *ACM SIGMOD*, 464-475.

[18] Vitter, J.S., Wang, M., and Iyer, B. 1998. Data Cube Approximation and Histograms via Wavelets. In *ACM CIKM*, 96-104.

[19] Wang, W., Lu, H., Feng, J., and Xu Yu, J. 2002. Condensed Cube: An Efficient Approach to Reducing Data Cube Size. In *IEEE ICDE*, 155-165.