Exploiting cluster analysis for constructing multi-dimensional histograms on both static and evolving data ⋆

Filippo Furfaro, Giuseppe M. Mazzeo, and Cristina Sirangelo

DEIS, University of Calabria, 87030 Rende, Italy,
{furfaro, mazzeo, sirangelo}@si.deis.unical.it

Abstract. Density-based clusterization techniques are investigated as a basis for constructing histograms in multi-dimensional scenarios, where traditional techniques fail in providing effective data synopses. The main idea is that locating dense and sparse regions can be exploited to partition the data into homogeneous buckets, preventing dense and sparse regions from being summarized into the same aggregate data. The use of clustering techniques to support the histogram construction is investigated in the context of either static and dynamic data, where the use of incremental clustering strategies is mandatory due to the inefficiency of performing the clusterization task from scratch at each data update.

1 Introduction

The need to compress data into synopses of summarized information often arises in many scenarios, where the aim is to retrieve aggregate data efficiently, possibly trading off the computational efficiency with the accuracy of query answers. Selectivity estimation for query optimization in RDBMSs [4, 16], range query answering in OLAP services [17], statistical and scientific data analysis [14], window query answering in spatial databases [1, 15], are examples of application contexts where efficiently aggregating data within specified ranges of the domain is such a crucial issue, that high accuracy in query answers becomes a secondary requirement.

For instance, query optimizers in RDBMSs can build an effective query evaluation plan by estimating the selectivity of intermediate query results: this can be accomplished by retrieving aggregate information on the frequencies of attribute values. In particular, given a relation $R(A_1, \ldots, A_d)$, the selectivity of a query of the form $q = v'_1 < R.A_1 < v''_1 \ \wedge \ldots \wedge \ v'_d < R.A_d < v''_d$ (representing the intermediate result of more complex queries) is evaluated by accessing the *joint frequency distribution* [16] associated to R. The latter can be viewed as a d-dimensional array $\mathcal{F}$ whose dimensions represent the attribute domains, and whose cell with coordinates $< v_1, \ldots, v_d >$ stores the number of tuples of R where $A_1 = v_1, \ldots, A_d = v_d$. The selectivity of the query q defined above is the answer of the range-sum query

⋆ This work was supported by a grant from the Italian Research Project FIRB "Grid.it – Enabling ICT Platforms for Distributed High-Performance Computational Grids", funded by MIUR and coordinated by the National Research Council (CNR).

Q=*sum*($\langle[v'_1..v''_1],\ldots,[v'_d..v''_d]\rangle$) posed on $\mathcal{F}$, which returns the sum of the frequencies contained in the multi-dimensional range $\langle[v'_1..v''_1],\ldots,[v'_d..v''_d]\rangle$ of $\mathcal{F}$. As the size of $\mathcal{F}$ is generally very large, evaluating the exact selectivity of q (i.e. the exact answer of Q) can be inefficient.

A widely accepted approach to the problem of providing fast estimates of query selectivities consists in compressing $\mathcal{F}$ into a lossy synopsis $\widetilde{\mathcal{F}}$, and then evaluating the selectivity of queries by accessing $\widetilde{\mathcal{F}}$ rather than $\mathcal{F}$. Histograms [16] are a well-known approach for compressing the joint frequency distribution. A histogram over $\mathcal{F}$ is built by partitioning $\mathcal{F}$ into a number of blocks (called *buckets*), and then storing for each bucket b the number of tuples in R whose attributes have values belonging to the range of b. The selectivity of q is estimated on the histogram by summing the values stored in the buckets whose boundaries are completely contained inside the range-sum query Q corresponding to q, and then by estimating the "contributions" of the buckets which partially overlap the range of Q. These contributions are evaluated by performing linear interpolation, under the assumption that the data distribution inside each bucket is "homogeneous" (that is, the joint distribution of attribute values underlying b is uniform).

As expected, on the one hand, querying the histogram rather than $\mathcal{F}$ reduces the cost of evaluating selectivities (as the histogram size is much less than the original data size); on the other hand, the loss of information due to summarization introduces some approximation. Therefore, a crucial issue when dealing with histograms is finding the partition which provides the "best" accuracy in reconstructing query selectivities.

Existing approaches, such as *MHIST* [16], *MinSkew* [1], *STHoles* [3] and *GENHIST* [12], provide reasonable error rates at low-dimensionality scenarios, but worsen dramatically for higher-dimensionality data. On the one hand, this is somewhat inevitable, since, as dimensionality increases, the size of the data domain grows much more than the number of data points. That is, high-dimensionality data are likely to be much sparser than low-dimensionality ones. This implies that the number of buckets which should be used to effectively approximate data tends to explode as dimensionality increases. For instance, consider two data distributions D^2 (of size n^2) and D^{10} (of size n^{10}), where the same number of data points are distributed, respectively, on a two-dimensional and ten-dimensional domain. If we use the same number of buckets to partition D^2 and D^{10}, buckets of D^{10} are likely to be much larger in volume than those of D^2. Therefore, the aggregate information associated to buckets of D^{10} is less localized than buckets of D^2 (as the aggregate value associated to each bucket is spread onto a larger volume), thus providing a poorer description of the actual data distribution.

On the other hand, the low accuracy in query estimates provided by traditional histograms is also due to the ineffectiveness of the adopted heuristics guiding the histogram construction. That is, traditional techniques for constructing histograms often result in partitions where dense and sparse regions are put together in the same bucket which yields poor accuracy in describing data. For instance consider the bucket shown in Fig. 1, where a dense cluster is put together with a sparse region. As the bucket is summarized by the sum of its values, estimating either Q_1 and Q_2 by performing linear interpolation yields a high error rate, since the total sum is assumed to be homogeneously distributed inside b. In fact this assumption is far from being true: most of the sum of b is concentrated in the dense cluster on the right-hand side of b.

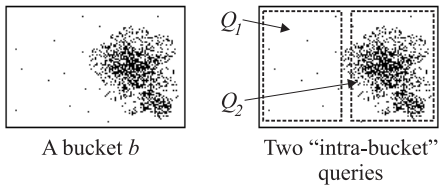

Fig. 1. Queries posed into a non-homogeneous bucket

Therefore, it is our belief that improving the ability of distinguishing dense regions can result in more accurate partitions, as this prevents buckets like that of Fig. 1 from being constructed. The problem of searching homogeneous regions is very close to the *data clustering* problem, i.e. the problem of grouping database objects into a set of meaningful classes. This issue has been widely studied in the data mining context, and several algorithms accomplishing data clustering have been proposed. For the sake of brevity we do not provide a classification of existing clustering techniques. The interested reader can find a detailed survey in [13].

This paper stems from our preliminary work [8], where we studied how histogram construction could be supported by density-based cluster analysis. In this work we propose an extension of our clustering-based compression technique to the case that data to be summarized is dynamic. In this context, the issue of maintaining data synopses has received growing attention from the research community in the last few years [5, 10, 11]. In this scenario, re-executing the clustering step at each data update is not feasible, due to the inefficiency of this task. Thus we introduce a strategy for exploiting an incremental clustering approach (where the clusterization is updated at each bulk of updates without re-processing the whole data) to efficiently propagate data updates to the histogram.

2 CHist: Clustering-based Histogram

In this section, we recall our clustering-based technique (namely, *CHist*) for constructing histograms on multi-dimensional static data. Its extension to the case of dynamic data (which is the main contribution of this paper) will be introduced in Section 3.

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

The three phases of our approach are described in detail in the following sections. The description of the algorithm is provided by assuming a d-dimensional data distribution D. D will be treated as a multi-dimensional array of integers of size n^d (without loss of generality the edges of D are assumed to be of the same size). That is, values of data points of the input distribution are represented into cells of D. The cells of D which do not correspond to any data point contain the value 0. A query Q on D is specified by a multi-dimensional range of the domain of D and its answer is the sum of the

values of the cells of D inside this range.
Any sub-array of D will be referred to as a *bucket*. The volume of a bucket b (i.e. the number of cells of the sub-array) will be denoted as $vol(b)$, the sum of data point values inside b as $sum(b)$. In order to measure the homogeneity of the data inside a bucket we adopt the SSE (namely *Sum Square Error*), defined as follows:

$$SSE(b) = \sum_{\mathbf{i} \in b} (b[\mathbf{i}] - avg(b))^2,$$

where: $avg(b)$ is the average of cell values inside b; the expression $\mathbf{i} \in b$ means that $\mathbf{i}$ denotes the coordinates of a cell inside b; $b[\mathbf{i}]$ denotes the value of the cell of b with coordinates $\mathbf{i}$. The amount of available storage space for the representation of the histogram will be denoted as B.

2.1 Step I: Clustering data

In our prototype, we have embedded the clustering algorithm DBSCAN [6] in order to group input data into dense clusters. Indeed, our approach can be viewed as orthogonal to any clustering technique: we have chosen DBSCAN as it is representative of density-based clustering algorithms.

The idea underlying DBSCAN is that points belonging to a dense cluster (except those points lying on the border of the cluster) have a dense neighborhood. A point p is said to have a dense neighborhood if there are at least *MinPts* distinct points whose distance from p is less than *Eps* (both *Eps* and *MinPts* are parameters crucial for the definition of clusters). Points with a dense neighborhood are said to be *core points*. DBSCAN scans input data searching for core points. Once a core point p is found, a new cluster C is created, and both p and all of its neighbors are grouped into C. Then C is recursively expanded by including the neighbors of all core points put in C at the last step. When C cannot be further expanded, DBSCAN searches for other core points to start new clusters, until no more core points can be found. At the end of the clustering, points which do not belong to any cluster are classified as *outliers*.

2.2 Step II: summarizing data into buckets

At this step the input data distribution is viewed as a superposition of layers. Each layer is either a cluster or the set of outliers. In the following we will denote the layer consisting of outliers as L_0, and the layers corresponding to dense clusters as $L_1, \ldots, L_c$. L_0 will be said to be the *outlier layer*, whereas $L_1, \ldots, L_c$ will be said to be *cluster layers*. Each layer is represented by means of its MBR (*Minimum Bounding Rectangle*, i.e. the minimum hyper-rectangle containing all non-null points of the layer).

The different layers are summarized separately by partitioning their MBRs into buckets. This aims at preventing the construction of buckets where dense and sparse regions are put together, which, as explained before (see Fig. 1), can yield poor accuracy. In more detail, our approach works as follows.

Layers are summarized independently of each other, and the summary of the whole data distribution will be the superimposition of the summaries of all layers. The summarization of layers is accomplished by a multi-step algorithm which, at each step, summarizes a single layer by partitioning it according to a grid and storing, for each bucket defined by this grid, both its MBR and the sum of its values (obviously, the cells

of this grid which do not contain any data point result in an empty MBR which is not stored). The MBRs of buckets obtained from the summarization of cluster layers will be said to be *c-buckets*, whereas the MBRs of the buckets constructed by partitioning L_0 will be said to be *o-buckets*.

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

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

We now show how $B(L_i)$ is exploited to store a partition of L_i into buckets. The idea is to partition L_i according to a grid and store, for each cell of the grid containing at least one point, the coordinates of its MBR and the sum of the values occurring in it. The grid on a layer L_i is constructed as follows.
If we denote as W the amount of storage space needed to store a bucket[2], the number of buckets produced by the grid on L_i can be no more than $nb = \lfloor \frac{B(L_i)}{W} \rfloor$. Thus, if t_j is the number of divisions of the grid along the j-th dimension of L_i, it should hold that $\prod_{j=1}^{d} t_j = nb$.

We partition each edge of the MBR of the layer to be summarized into a number of portions which is proportional to the length of the edge itself. See [8] for further details on the technique used for defining such a "uniform" grid for each layer partition. The cells of the grid which correspond to null regions of the data domain are not stored explicitly. In the following, nb' will denote the number of buckets generated by the grid partitioning which are stored explicitly (i.e. the number of buckets containing at least one non-null point). Therefore, after a layer L_i is summarized, the residual amount of storage space which will be available at step $i+1$ is given by $B_{i+1} = B_i - nb' \cdot W$ (that is, if some space which was assigned to the summarization of L_i has not been consumed, it is re-invested at the following steps).

Fig. 2 shows the execution of Step I and Step II on a two-dimensional data distribution.

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

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

Remark. Observe that adopting the grid-based scheme allows us to partition a layer L in linear time (each data point inside L is accessed once and summarized in the cell of the grid where it lies into): this feature will be particularly well-suited for the incremental approach where an efficient partitioning strategy is needed to propagate data changes to the histogram (see Section 3).

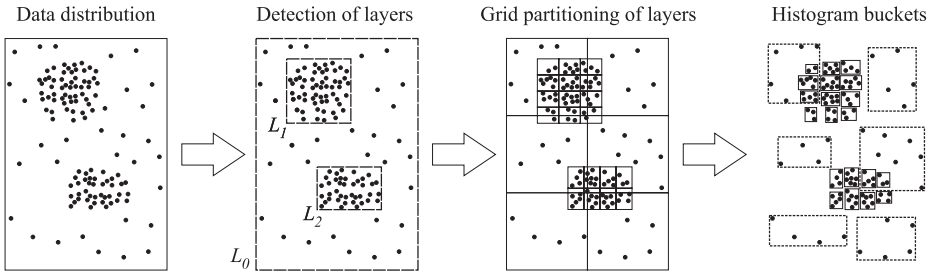

Fig. 2. Detection of layers, data partitioning, and bucket definition

2.3 Step III: representation of the histogram

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

Consider the scenario depicted in Fig. 3(a), where the query Q_1 intersects one half of the range associated to the bucket b. Adopting linear interpolation to estimate Q_1 returns: $\widetilde{Q}_1 = \frac{vol(Q_1 \cap b)}{vol(b)} \cdot sum(b)$, where $Q_1 \cap b$ refers to the intersection between the query range and the range of b. In fact points belonging to the ranges of $b_1, \ldots, b_9$ give no contribution to the value of $sum(b)$. Therefore, a more precise estimate for Q_1 is: $\widetilde{Q}_1 = \frac{vol(Q_1 \cap b)}{vol(b) - vol(b_1, \ldots, b_9)} \cdot sum(b)$, where $vol(b_1, \ldots, b_9)$ denotes the volume of the range underlying the buckets $b_1, \ldots, b_9$. Likewise, the bucket b should give no contribution to the estimate of the query Q_2 in Fig. 3(b), which lies completely on the range underlying the buckets $b_1, \ldots, b_9$.

In the following the number of cells of an o-bucket b which are not contained in any hole of b will be said to be the *actual volume* of b. In the case depicted in Fig. 3(a) evaluating the actual volume of b can be accomplished efficiently, as $b_1, \ldots, b_9$ do not overlap. Indeed also c-buckets inside an o-bucket b can intersect one another [3]. For instance, in Fig. 3(c) the three buckets b_1, b_2, b_3 inside b overlap. In this case computing

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

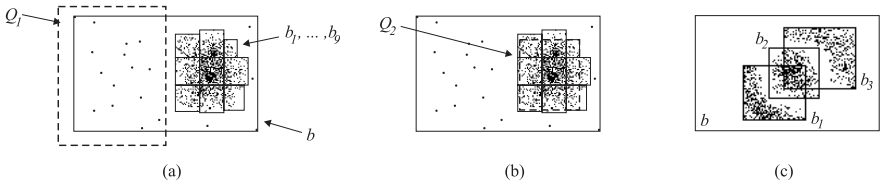

Fig. 3. O-buckets with holes

the actual volume of b requires $vol(b_1)$, $vol(b_2)$, $vol(b_3)$, $vol(b_1 \cap b_2)$, $vol(b_2 \cap b_3)$ and $vol(b_1 \cap b_2 \cap b_3)$ to be computed. This computation becomes more and more complex when more buckets intersect in the same region: we need to compute the volumes of all the intersections between 2 holes, 3 holes, and so on. Obviously, this slows down query estimations. Due to this reason, we prefer to estimate the actual volume of an o-bucket b involved in a query instead of evaluating its exact value: To this end we consider only a maximal subset of *Holes*(b) (denoted as *NOHoles*(b)) consisting of non-overlapping c-buckets, thus avoiding intersections between holes to be computed. For instance, in the case depicted in Fig. 3(c) we can choose *NOHoles*$(b) = \{b_3\}$, thus we can estimate the actual volume of b as $vol(b) - vol(b_3)$. However we point out that from our experiments on real-life data it turned out that intersections between c-buckets are unlikely to occur.

The adopted representation model partitions buckets into two levels. The buckets at the second level are those belonging to *NOHoles*(b) for some b. The first level consists of all the other buckets. In [8] we present an efficient physical representation scheme, that is based on the possibility to linearize the two bucket levels and allows range query answers to be estimated by accessing each bucket at most once.

3 Incremental maintenance of CHIST on evolving data-sets

The computational complexity of the histogram construction is dominated by the cost of executing DBSCAN. DBSCAN runs in $O(N \cdot \log N)$ if a multi-dimensional indexing technique is adopted to support the efficient location of neighbors. Indeed its complexity degrades to $O(N^2)$ on high-dimensional data sets, where no indexing technique is known to be efficient in searching the neighbors of data points. This is likely to limit the applicability of CHIST to static data sets, such as non-evolving historical data, where the construction of the histogram is performed only once. Otherwise, in the case of evolving data sets, any change of the data would require the re-execution of the algorithm from scratch. In order to reduce the overhead due to this task, the re-computation of the histogram could be scheduled to be run periodically (e.g. every night) or when the system managing data is unloaded. But this could make the histogram out-of-date, thus compromising the estimation accuracy, especially in the case that data change much more frequently w.r.t. histogram re-computation. Observe that the adoption of a clustering technique more efficient than DBSCAN does not suffice to solve this problem, as no technique is known to accomplish the (from-scratch) clusterization fast enough.

A possible solution to this problem is to adopt an incremental clustering technique to propagate efficiently data changes to the clusterization. An incremental clustering algo-

rithm computes the clusterization of the updated data starting from the pre-existing clusterization and modifying it according to the data updates, aiming at reducing as much as possible the amount of data to be accessed. However, replacing the non-incremental clustering step with an incremental one at Step I may not suffice to make the whole technique well-suited for reacting to frequent updates. In fact, Step II requires a linear scanning of data to compute the bucketization of all layers. In order to exploit the advantage of incremental clustering, Step II needs to be changed too, so that layers which are not affected by the data updates are not re-partitioned, thus exploiting as much as possible the pre-existing bucketization.

Motivated by these observations, in this section we propose an incremental algorithm for maintaining the histogram up-to-date w.r.t. data changes. In more detail, our strategy works in three steps, which will be described in the following sections:

I Incremental clustering;
II Storage space distribution among layers and partitioning;
III Re-arrangement of buckets.

Throughout the following sections we assume that each point p of the data distribution is marked with two labels *Flag*(p) and *Layer*(p) [4]. The former has a boolean value, specifying whether p is an outlier or belongs to a cluster. *Layer*(p) is the identifier of the layer where p is summarized: thus, if p is an outlier summarized in a o-bucket then *Layer*$(p) = 0$, else if p is a point summarized in a c-bucket obtained by partitioning the layer L_i then *Layer*(p) is the identifier of L_i. Basically, the values of *Flag*(p) and *Layer*(p) describe the current composition of layers before executing a bulk of updates, and are changed accordingly to the data updates during steps I,II. In particular, during the execution of these steps, *Layer*(p) can be also assigned -1, meaning that p has not been assigned to any layer yet.

3.1 Step I: Incremental clustering

The task performed at this step consists in propagating data updates to the clusterization. There are several techniques in literature which accomplish this task in an incremental fashion, that is they compute the clusterization of updated data without re-executing the clustering algorithm from scratch on all the data. In our prototype we adopted *Incremental DBScan* [7]. According to this technique, data updates may have different effects on the clusterization, and thus on the corresponding layers. When a new point p is added to the data distribution, one of the following cases may occur:

I1- *no new cluster is created, and no old cluster is affected*: this happens if p is an outlier; in this case, the layer of outliers must be augmented, whereas the other layers need no change; *Flag*(p) is assigned 0 (meaning that p is classified as an outlier) and *Layer*(p) is assigned -1 (meaning that p is an outlier which has not been summarized in any bucket yet);
I2- *a new cluster including p is created, and no old cluster is affected*: in this case, a new layer is created (corresponding to the new cluster), and the layer of outliers

[4] Apart from further labels possibly associated to the points by the adopted clustering algorithm

may need to be reduced (in the case that some pre-existing outliers are absorbed into the new cluster). Layers corresponding to pre-existing clusters need no change. In this case, for each point p' included in the new cluster, *Flag*(p') is assigned 1 and *Layer*(p') is assigned the id of the new cluster.

I3- *no new cluster is created, and some old clusters are affected*: this can arise from one of the following cases:

- p *is adsorbed by exactly one of the pre-existing clusters*: in this case, the layer of the involved cluster must be augmented; *Flag*(p) is assigned 1 and *Layer*(p) is assigned the id of the cluster adsorbing p;
- p *is adsorbed by two or more clusters, and these clusters are merged in a single one*: in this case, the layers of the merged clusters must be deleted, and a new layer corresponding to the new cluster must be created. For each point p' adsorbed by the new cluster *Flag*(p') is assigned 1 and *Layer*(p') is assigned the id of the new cluster.

Moreover, in both cases the layer of outliers must be reduced if some pre-existing outliers are absorbed into a cluster together with p. For each of these points p', *Flag*(p') changes from 0 to 1 and *Layer*(p') is assigned the id of the adsorbing cluster.

Analogously, when a point p is deleted from the data distribution it can be one of the following cases:

D1- *no old cluster is affected*: this happens if p was an outlier; in this case, the layer of outliers must be reduced and no other layer need updates;

D2- *exactly one old cluster is affected*: this happens if p belonged to a cluster C. In particular, one of the following cases can occur:

a. C *is reduced*: this happens when after the removal of p some points of C become outliers; in this case the layer of C must be reduced and the outlier layer must be augmented. In particular, for each p' which is no more a cluster point, *Flag*(p') is assigned 0 and *Layer*(p') is assigned -1;

b. C *is deleted*: this happens when the removal of p results in making no point of C have a dense neighborhood, thus all points of C become outliers; in this case the layer corresponding to C is deleted and the layer of outliers must be augmented. For each point p' which belonged to C *Flag*(p') is assigned 0 and *Layer*(p') are assigned -1;

c. C *is split into two or more clusters*: this happens when, after the removal of p, two or more core points are no more density-reachable from one another; thus they define distinct clusters. In this case the layer corresponding to C is split into two layers (i.e. the layer of C is deleted and two new layers are created). Moreover the layer of outliers may need to be augmented (in the case that some points belonging to C become outliers). For each point p' involved in the split, the values of *Flag*(p') and *Layer*(p') are changed consistently.

The above-reported list summarizes the operations performed on layers for a single update. Indeed an incremental clustering step consists of processing a bulk of updates, which is processed as a sequence of single updates. See [7] for further details and graphical examples on how inserting/deleting points can change clusterization.

The histogram maintenance is supported by an auxiliary (main-memory resident) data structure consisting of two sets $\mathcal{L}_{new}$ and $\mathcal{L}_{old}$, whose items are of the form $< L, MBR(L), sum(L), sum^2(L), count(L), B(L) >$, where L is a layer identifier and $B(L)$ denotes the amount of storage space which was invested to partition L during the construction of the old histogram (obviously $B(L) = 0$ if L is a newly detected layer). The aggregate data $sum(L)$, $sum^2(L)$ and $count(L)$, as well as $MBR(L)$, will be used at Step 2 to evaluate the SSE of L, whereas $B(L)$ will be used to decide whether old layers need to be re-partitioned or not. Basically, at the end of the incremental clustering step, $\mathcal{L}_{new}$ and $\mathcal{L}_{old}$ contain the up-to-date clusterization (w.r.t. the processed bulk of insertions and deletions). In particular, $\mathcal{L}_{old}$ contains the list of the layers which existed before the bulk of updates and which have not been affected by the updates; on the contrary, $\mathcal{L}_{new}$ consists of the layers which were not in the pre-existing clusterization. Neither $\mathcal{L}_{old}$ nor $\mathcal{L}_{new}$ contain any tuple corresponding to the layer of outliers: aggregate data of L_0 are stored separately from these lists.

At the beginning of the incremental clustering step, $\mathcal{L}_{new}$ is empty while $\mathcal{L}_{old}$ contains the list of the pre-existing layer identifiers and their aggregate data (except from L_0). During the execution of the incremental clustering step, both $\mathcal{L}_{new}$ and $\mathcal{L}_{old}$ are maintained up-to-date as follows. Consider an update operation u (i.e. insertion or deletion) in the processed bulk of updates. Let *Affected*(u) be the set of layers affected by u and *Created*(u) the set of layers created after performing u. Basically, *Affected*(u) contains layers in $\mathcal{L}_{old} \cup \mathcal{L}_{new}$ which need either augmentation or reduction or deletion, whereas *Created*(u) contains layers which need to be created (i.e. layers in *Created*(u) can result from either splitting clusters, merging clusters, or creating new clusters). For each layer L in *Created*(u) the tuple $< L, MBR(L), sum(L), sum^2(L), count(L), 0 >$ is inserted into $\mathcal{L}_{new}$. For each layer L in *Affected*(u) the following operations are performed. If L has to be deleted, then the corresponding tuple is removed from the list it belongs to (either $\mathcal{L}_{new}$ or $\mathcal{L}_{old}$). Otherwise, if L needs either augmentation or reduction, the attributes $MBR(L)$, $sum(L)$, $sum^2(L)$, $count(L)$ in the corresponding tuple are updated. Moreover, if L was in $\mathcal{L}_{old}$ the corresponding tuple is moved to $\mathcal{L}_{new}$ (after assigning 0 to $B(L)$). Finally, for each outlier p which had been summarized into the buckets of some layer in *Affected*(u) the value of *Layer*(p) is changed to -1.

Therefore, at the end of the incremental clustering, every point p classified as a cluster point is assigned *Flag*$(p) = 1$ and *Layer*$(p) = id(L)$, where L is the layer corresponding to the cluster containing p. For each outlier p, *Flag*(p) is assigned 0; as regards *Layer*(p) one of the following cases can occur:

- *Layer*$(p) = i \geq 0$: this means that p is currently summarized into a bucket associated to the layer L_i;
- *Layer*$(p) = -1$: this means that p is not currently summarized into any bucket (this can be due to two reasons: either p is a newly created outlier, or p was an outlier summarized into a layer affected by the data update).

As we will show in the following section, every outlier whose *Layer* value is -1 will be assigned to exactly one layer and summarized into one of its buckets. That is, if the outlier p happens to be summarized into an o-bucket, then *Layer*(p) will be assigned 0, else if p happens to be adsorbed by a c-bucket b, then *Layer*(p) will be assigned the id of the layer which b refers to.

Lists $\mathcal{L}_{new}$ and $\mathcal{L}_{old}$ will be used at Step II to detect layers which need to be partitioned.

3.2 Step II: Storage space distribution among layers and partitioning

The incremental clustering step results in a new clusterization, where new layers may be added and some pre-existing layers may be either deleted or modified w.r.t. the previous clusterization. The overall amount of storage space B must be now re-distributed among the layers in $\mathcal{L} = \mathcal{L}_{new} \cup \mathcal{L}_{old} \cup \{L_0\}$. Adopting the same criterion as the non-incremental approach (see Section 2.2) is likely to result in changing the amount of storage space assigned to layers in $\mathcal{L}_{old}$, thus requiring also all layers non-affected by data updates to be re-partitioned. This should be avoided, as it would imply to re-scan all data points. Thus, in the incremental approach, we adopt a different strategy to distribute the available storage space B among layers. This strategy aims at being fair and restricting as much as possible the set of pre-existing layers to be re-partitioned.

Layers in $\mathcal{L}_{old}$ and $\mathcal{L}_{new}$ and the layer of outliers L_0 will be considered into three distinct phases, to be executed in the following order.

Partitioning layers in $\mathcal{L}_{old}$. We denote as $\widehat{B}(\mathcal{L}_{old})$ the portion of B which we want to assign on the whole to layers in $\mathcal{L}_{old}$. According to a fair distribution of the available storage space B between $\mathcal{L}_{old}$ and $\mathcal{L}_{new}$, we choose:

$$\widehat{B}(\mathcal{L}_{old}) = \frac{SSE(\mathcal{L}_{old})}{SSE(\mathcal{L})} \cdot B,$$

where: $SSE(\mathcal{L}_{old}) = \sum_{L \in \mathcal{L}_{old}} SSE(L)$ is an estimate of the overall inhomogeneity of layers in $\mathcal{L}_{old}$, and: $SSE(\mathcal{L}) = \sum_{L \in \mathcal{L}} SSE(L)$ measures the overall inhomogeneity of all the layers resulting from the clusterization. Notice that the SSE of each layer L is computed by accessing the aggregate data $sum(L)$, $sum^2(L)$, $count(L)$, $MBR(L)$, stored in the tuple in $\mathcal{L}$ corresponding to L: $SSE(L) = sum^2(L) - \frac{(sum(L))^2}{Vol(L)}$.

Let $B(\mathcal{L}_{old}) = \sum_{L \in \mathcal{L}_{old}} B(L)$ be the amount of storage space consumed by the summarization of all the layers in $\mathcal{L}_{old}$. First $\widehat{B}(\mathcal{L}_{old})$ is compared to $B(\mathcal{L}_{old})$. The idea is that if $B(\mathcal{L}_{old})$ is pretty "close" to $\widehat{B}(\mathcal{L}_{old})$ we do not re-partition layers in $\mathcal{L}_{old}$. In particular in order to decide whether $B(\mathcal{L}_{old})$ is close to $\widehat{B}(\mathcal{L}_{old})$, we introduce a threshold parameter t. Thus if $|\widehat{B}(\mathcal{L}_{old}) - B(\mathcal{L}_{old})| < t \cdot \widehat{B}(\mathcal{L}_{old})$, the pre-existing partition of layers in $\mathcal{L}_{old}$ will not be changed.

Otherwise layers in $\mathcal{L}_{old}$ are re-partitioned depending on which of the following cases occurs:

- $B(\mathcal{L}_{old}) > (1 + t) \cdot \widehat{B}(\mathcal{L}_{old})$: this means that the amount of storage space currently invested to summarize layers in $\mathcal{L}_{old}$ is on the whole too large (according to the adopted fair-distribution criterion); thus we re-partition some layers in $\mathcal{L}_{old}$ by means of a coarser-grain grid in order to release some storage space;
- $B(\mathcal{L}_{old}) < (1 - t) \cdot \widehat{B}(\mathcal{L}_{old})$: in this case we augment the storage space currently invested to summarize $\mathcal{L}_{old}$, by re-partitioning by means of a finer-grain grid the layers in $\mathcal{L}_{old}$ which are the most in need of a finer partition.

In order to choose the layers to be re-partitioned, for each layer L in $\mathcal{L}_{old}$ we evaluate $\widehat{B}(L) = \frac{SSE(L)}{SSE(\mathcal{L})} \cdot B$. The value of $\widehat{B}(L)$ is a fair portion of the available storage space to be assigned to L.

A layer L in $\mathcal{L}_{old}$ such that $B(L) > \widehat{B}(L)$ is said to be *indebted*, in the sense that it is assigned an amount of storage space larger than the amount it would be assigned in a fair space distribution based on its relative inhomogeneity. So it is "in debt" of some storage space to other layers. On the contrary, a layer L such that $B(L) < \widehat{B}(L)$ is said to be *creditor*, in the sense that it is assigned an amount of storage space smaller than the one it would need according to its SSE. That is, it is creditor of some storage space.

Consider the case that $B(\mathcal{L}_{old}) > (1+t) \cdot \widehat{B}(\mathcal{L}_{old})$ holds. Then, it is straightforward to see that there is at least one indebted layer in $\mathcal{L}_{old}$ such that $B(L) > (1+t) \cdot \widehat{B}(L)$. Let L^* be the most indebted layer in $\mathcal{L}_{old}$. The idea is to deprive L^* of some storage space in order to make the overall space consumed by layers in $\mathcal{L}_{old}$ closer to $\widehat{B}(\mathcal{L}_{old})$. In particular, we steal from $B(L^*)$ a portion of storage space which makes L^* creditor of $\frac{t}{2} \cdot \widehat{B}(L^*)$. Therefore, the amount of storage space stolen from L^* is:

$$B^-(L^*) = B(L^*) - \left(1 - \frac{t}{2}\right) \cdot \widehat{B}(L^*).$$

Then we re-partition L^* by investing the amount of storage space $B(L^*) - B^-(L^*)$. If at the end of this step $B(\mathcal{L}_{old}) > \widehat{B}(\mathcal{L}_{old})$ still holds, then we choose the layer in $\mathcal{L}_{old}$ which is the most in debt and deprive it of some storage space, using the same strategy as above. This process goes on until $B(\mathcal{L}_{old}) \leq \widehat{B}(\mathcal{L}_{old})$. That is, we take layers which are "very much indebted" and make them "pretty" in credit (we use the threshold value $t/2$ to estimate that a layer is creditor in a small extent): this strategy aims at reaching rapidly the condition $B(\mathcal{L}_{old}) \leq \widehat{B}(\mathcal{L}_{old})$, by reducing the number of layers to be re-partitioned, which is mandatory for the efficiency requirements of the incremental approach.

In the case that $B(\mathcal{L}_{old}) < (1-t) \cdot \widehat{B}(\mathcal{L}_{old})$ an analogous approach is adopted: we choose the layer L^* which is creditor of the largest amount of storage space, and we augment its storage space by adding to it:

$$B^+(L^*) = \left(1 + \frac{t}{2}\right) \cdot \widehat{B}(L^*) - B(L^*),$$

which means making L^* indebted of at most $\frac{t}{2} \cdot B(L^*)$. Then we re-partition L^*, and re-iterate this procedure on the other creditors in $\mathcal{L}_{old}$ until $\widehat{B}(\mathcal{L}_{old}) \geq B(\mathcal{L}_{old})$.

By means of experiments, we found that the threshold value $t = 20\%$ preserves the accuracy of the updated histogram and it effectively limits the number of layers to be re-partitioned.

If a layer $L \in \mathcal{L}_{old}$ is chosen to be re-partitioned (as it is creditor or indebted in too large extent), the *Layer* value of the outliers which were summarized in the buckets of L at some previous step is assigned the value -1. These outliers will be considered for summarization into some bucket at the following step.

In the following, outliers whose *Layer* value is -1 will be said to be *new outliers*, whereas outliers whose *Layer* value is greater than or equal to 0 will be said to be *old outliers*.

Partitioning layers in $\mathcal{L}_{new}$. Layers $L_1, \ldots, L_\alpha$ in $\mathcal{L}_{new}$ are partitioned sequentially according to the same scheme adopted in the non-incremental approach. The amount of storage space invested to partition layers in $\mathcal{L}_{new}$ is $B(\mathcal{L}_{new}) = \frac{SSE(\mathcal{L}_{new})}{SSE(\mathcal{L})} \cdot B$. Then for each $i \in [1..\alpha]$, the layer L_i is summarized according to the grid-partitioning scheme described in Section 2.2 by investing the amount of storage space:
$B(L_i) = B_i \cdot \frac{SSE(L_i)}{SSE(L_0)+\sum_{j=i}^{\alpha} SSE(L_j)}$,
where $B_1 = B(\mathcal{L}_{new})$ and B_i is the portion of $B(\mathcal{L}_{new})$ which is left from the summarization of $L_1, \ldots, L_{i-1}$.
Partitioning L_0. Let $B' = B - B(\mathcal{L}_{new}) - B(\mathcal{L}_{old})$ be the amount of storage space which can be invested to summarize L_0, i.e. the portion of B which is left from summarizing layers in $\mathcal{L}_{new}$ and $\mathcal{L}_{old}$. L_0 is re-partitioned if one of the following cases occurs:

1. $B(L_0) \geq B'$: this means that the current bucketization of L_0 makes the overall storage space consumption of the histogram exceed B, thus L_0 must be re-partitioned using a coarser-grain grid;
2. $B(L_0) \leq (1-t) \cdot B'$: this means that the space currently invested to partition L_0 is too small (according to the threshold t), thus L_0 must be re-partitioned using a finer-grain grid.

If either case 1 or case 2 occurs, L_0 must be re-partitioned, thus a new grid is defined on L_0 (by investing the amount of storage space $B - B(\mathcal{L}_{\text{new}}) - B(\mathcal{L}_{old})$). In this case both new and old outliers are scanned, and each outlier is summarized either into a c-bucket or into an o-bucket, depending on whether it lies into the range of some c-bucket or not.

Otherwise, if neither case 1 nor case 2 occurs, the existing grid-partitioning of L_0 is kept and the current summarization is updated as follows. First, the new outliers are scanned and summarized into either a c-bucket or an o-bucket, as for the previous case. Then, the buckets of L_0 are deprived of the outliers which lie into the range of some newly created c-bucket.

Details on how these tasks are accomplished in the implementation are given in Section 4.

3.3 Step III: Re-arrangement of buckets

The task accomplished at this step consists of applying the same physical representation scheme as the non-incremental approach to the set of buckets resulting from Step II. The up-to-date histogram consists of buckets of four types: 1) c-buckets resulting from partitioning layers in $\mathcal{L}_{new}$, 2) c-buckets resulting from re-partitioning selected layers in $\mathcal{L}_{old}$, 3) c-buckets inherited from the previous histogram which refer to layers in $\mathcal{L}_{old}$ which have not been re-partitioned, 4) o-buckets partitioning L_0 (these buckets can result either from updating the o-buckets of the previous histogram or from re-partitioning L_0). The pre-existing arrangement of buckets is not exploited to re-arrange new buckets, as this does not result in a relevant overhead. In fact all the operations needed to accomplish this task are performed in main memory (where the new bucketization is stored), without accessing disk-resident data.

The algorithm implementing steps I to III is shown on the next page.

Algorithm Incremental CHIST

INPUT: D: a multi-dimensional data distribution; B: 32bit words used to store the histogram;
H: the histogram currently built on D; u: the bulk of updates to be propagated to H;
OUTPUT: H': an up-to-date histogram on D within B;

Step I

$< \mathcal{L}_{old}, \mathcal{L}_{new}, L_0 >$:= Incremental_DBSCAN$(H, D, u)$; // the new layerization reflecting updates;

Step II

Partitioned=$\emptyset$; *NewBuckets*=$\emptyset$; $\widehat{B}(\mathcal{L}_{old}) = \frac{SSE(\mathcal{L}_{old})}{SSE(\mathcal{L})} \cdot B$; $B(\mathcal{L}_{old}) = \sum_{L \in \mathcal{L}_{old}} B(L)$;

if $B(\mathcal{L}_{old}) \geq (1+t) \cdot \widehat{B}(\mathcal{L}_{old})$ **then**

while $B(\mathcal{L}_{old}) \geq (1+t) \cdot \widehat{B}(\mathcal{L}_{old})$ **do**

$L =$ *SelectMostIndebted*$(\mathcal{L}_{old})$; *Partitioned*=*Partitioned* $\cup L$; $B(L) = (1-\frac{t}{2}) \cdot \frac{SSE(L)}{SSE(\mathcal{L})} \cdot B$;

NewBuckets= *NewBuckets* $\cup$ *GridPartition*$(L,\ B(L))$; // L is re-partitioned and all the outliers
// which were summarized in it are
// assigned to L_0;

end_while

elsif $B(\mathcal{L}_{old}) \leq (1-t) \cdot \widehat{B}(\mathcal{L}_{old})$ **then**

while $B(\mathcal{L}_{old}) \leq (1-t) \cdot \widehat{B}(\mathcal{L}_{old})$ **do**

$L =$ *SelectMostCreditor*$(\mathcal{L}_{old})$; *Partitioned*=*Partitioned* $\cup L$; $B(L) = (1+\frac{t}{2}) \frac{SSE(L)}{SSE(\mathcal{L})} B$;

NewBuckets= *NewBuckets* $\cup$ *GridPartition*$(L,\ B(L))$;

end_while

end_if;

$\mathcal{L}_{old} = \mathcal{L}_{old} \setminus$ *Partitioned*; // $\mathcal{L}_{old}$ contains now all the non re-partitioned pre-existing layers;

for each L **in** $\mathcal{L}_{new}$ **do**

$B(L) = \frac{L.SSE}{L_0.SSE + SSE(\mathcal{L}_{new})} \cdot (B - size(Partitioned) - size(\mathcal{L}_{old}))$; // the amount of memory invested
// to summarize L;

NewBuckets= *NewBuckets* $\cup$ *GridPartition*$(L, B(L))$; // L is partitioned and resulting buckets
// are added to *NewBuckets*;

Partitioned=*Partitioned* $\cup L$;

end_for;

$B' = B -$ *size*(*Partitioned*) $-$ *size*$(\mathcal{L}_{old})$; // function *size* returns the amount of memory
// needed to store the buckets taken as input;

if $B(L_0) \leq B'$ and $B(L_0) \geq (1-t) \cdot B'$ **then**

O-Buckets= H.*O-Buckets*; // H' inherits the set of o-buckets of H;

DistributeNewOutliers(*NewBuckets*, *O-Buckets*); // New outliers are distributed among
// new c-buckets and old o-buckets;

MoveOutliers(*NewBuckets*, *O-Buckets*); // O-buckets are deprived of old outliers
// lying into some new c-bucket;

else

O-Buckets= *PartitionAndDistribute*$(L_0, B - size(Partitioned) - size(\mathcal{L}_{old}), NewBuckets)$;

end;

Step III

H' =*Assemble*(*NewBuckets*, *UnchangedBuckets*$(H, \mathcal{L}_{old})$, *O-Buckets*);

return H';

4 Costs of the non-incremental and incremental approaches

The difference in the number of disk accesses between the two approaches is due to two reasons. First, the adoption of the incremental clustering, which is likely to result in much fewer disk accesses w.r.t. the non-incremental one. Secondly, the strategy adopted at Step II, which aims at limiting the number of layers to be re-partitioned, avoiding re-scanning the whole data. As regards the former aspect, the extent of the benefit introduced by the use of an incremental clustering approach strictly depends on the particular clustering algorithm invoked. In the case of DBSCAN, no simple formula is known to provide the speedup factor corresponding to the use of its incremental version, thus the speedup must be determined experimentally.

As regards the second aspect, we can compare the number of disk accesses as follows. In the non incremental approach, after accomplishing the clusterization, a region query must be posed corresponding to the MBR of each detected dense cluster to partition it according to the grid; then, the list of outliers must be scanned to distribute them among c-buckets and o-buckets. Thus, denoting the number of dense clusters as c, the number of data points as N and the number of pages containing outliers as *Out*, the number of disk accesses is $O(c \cdot \log N + \mathit{Out})$ (we assume that a multi-dimensional index enabling region queries to be answered with $\log N$ accesses is maintained, as well as an inverted index of the pages containing outliers). As regards the incremental approach, let c' be the number of clusters which need to be partitioned, *OldOut* the number of pages containing the old outliers and *NewOut* the number of pages containing the new outliers. We must pose c' region queries to partition the dense clusters and we have to scan *NewOut* pages to distribute the new outliers among c-buckets and o-buckets. Moreover, if it is the case that L_0 must be re-partitioned, we must also scan *OldOut* pages to repartition it and possibly adsorb old outliers into new c-buckets. Otherwise, if L_0 does not need re-partitioning, we must only pose b_{new} region queries on the set of old outliers to possibly adsorb some of them into new c-buckets (b_{new} denotes the number of the new c-buckets). Therefore, the overall number of disk accesses is $O(c' \cdot \log N + \mathit{NewOut} + X)$, where X is either $b_{new} \cdot \log \mathit{OldOut}$ (if L_0 is not re-partitioned) or *OldOut* (if L_0 must be re-partitioned). Observe that in order to support the incremental approach we also maintain an inverted index on the newly detected outliers (which allows us to scan all the new outliers by means of *NewOut* accesses) as well as a multi-dimensional index to answer region queries on old outliers with $\log$ *Out* accesses. Notice that in the worst case $\mathit{NewOut} + X = \mathit{Out}$ (in the non-incremental approach there is no distinction between new and old outliers, thus $\mathit{Out} = \mathit{NewOut} + \mathit{OldOut}$), but in the case that the outlier layer is not repartitioned $\mathit{NewOut} + X$ can be reasonably assumed much smaller than Out. Moreover we can assume c' much smaller than c. Therefore if the number of outliers is "small" w.r.t. the whole data size, then the adoption of the incremental strategy always results in a relevant benefit, otherwise the extent of this benefit depends on the probability that L_0 must be repartitioned.

The latter issue cannot be investigated but experimentally, as well as the speedup due to the adoption of the incremental clustering strategy. Therefore in the following section we will provide an experimental analysis of the overall benefit of the incremen-

tal approach, which both the incremental clusterization and the partitioning strategy contribute to.

5 Experimental Results

Experimental results showing the higher accuracy provided by CHIST w.r.t state-of-the-art techniques on static data have been shown in [8]. Here we present experiments testing the effectiveness of the incremental approach, comparing it with the from-scratch execution of CHIST. Synthetic data sets were generated according to a multi-dimensional zipf distribution law. Basically a data set consists of a number of dense regions randomly distributed in the data domain. Randomly generated points are added to simulate noise (for further details see [8]).

To generate a bulk of insertions on a distribution D, we first generated a distribution D' on the same domain and using the same data generator as D. Then a bulk of insertions on D is created by randomly extracting some points from D'. The idea of extracting points from D' to be inserted into D is that this allows us to simulate both the creation of new dense clusters and new outliers in D. Deletions on D consist of randomly selected points of D. Thus a bulk of updates is a set of insertions and deletions, generated as explained above. For a bulk of updates u, we will denote as p_u the percentage of insertions in u.

Diagrams in Fig. 4(a,b,c) refer to a 4D data distribution where n (the edge size of the domain) is 1000, while diagrams in Fig. 4 (d,e,f) refer to an 8D data distribution with $n = 1000$.

Fig. 4 (a,d) show the speedup due to the use of Incremental CHIST versus the size of updates (expressed as percentage of the data size) on a 4D and a 8D synthetic data distribution, respectively. For a bulk of updates u, the speedup is the ratio $\frac{\textit{N. of pages accessed by CHIST}}{\textit{N. of pages accessed by Incremental CHIST}}$. Fig. 4 (a,d) show that the benefit of using the incremental approach is very relevant (in both cases we obtained a speedup value of about 200 for update size of 1%). As expected, the speedup decreases as the size of the updates gets larger. Observe that the speedup depends also on the type of updates: the larger the percentage of insertions, the higher the speedup. This is in accordance with [7], where it was observed that deletions on the average result in more complex changes of the clusterization, as they involve a larger number of pre-existing clusters than insertions.

Diagrams in Fig. 4 (b,e) study the accuracy of the incremental approach, compared with that of the non-incremental one. They depict the ratio $e = \frac{e_{ni}}{e_i}$ between the relative errors provided by the non-incremental approach (i.e. e_{ni}) and the incremental one (i.e. e_i) versus the size of updates. Experiments were conducted investing 2000 buckets to represent the histogram. Two query workloads (one for the 4D case and the other one for the 8D case) were used to evaluate estimation accuracy. Each of them consists of 50000 queries whose selectivity is between 0.4% and 0.5% . Fig. 4 (b,e) show that the ratio $\frac{e_{ni}}{e_i}$ is close to 1 and is almost unaffected by the size of updates. This means that the adoption of the incremental approach does not result in degrading accuracy w.r.t. the non-incremental one.

Observe that the diagrams in Fig. 4 (b,e) do not say that the accuracy provided by the histograms computed after each bulk of updates is constant as data changes. In fact,

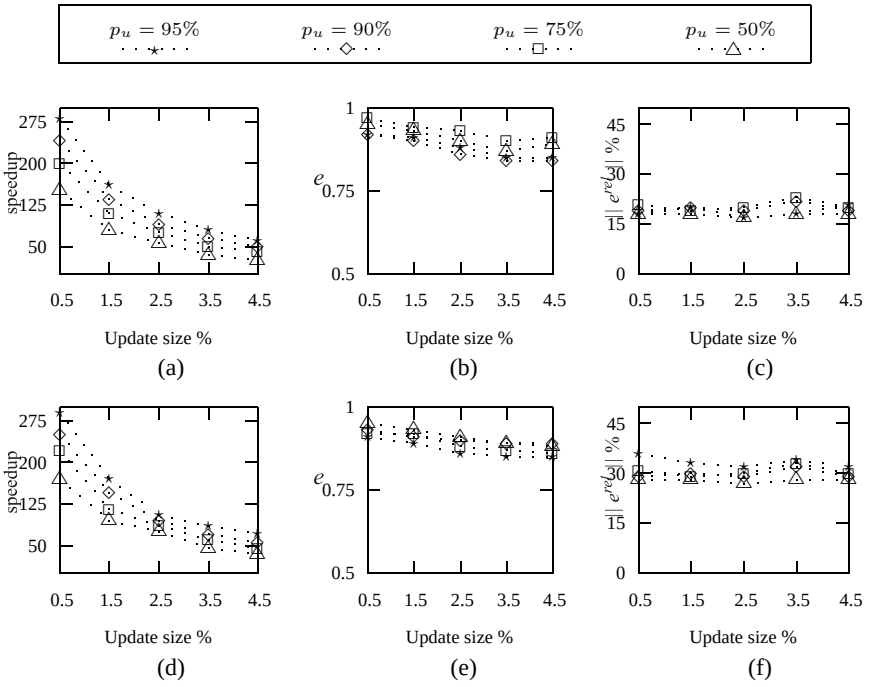

Fig. 4. Speedup synthetic data (a,d); ratio $e = \frac{e_{ni}}{e_i}$ for fixed histogram size (b,e); relative error for fixed compression ratio (c,f)

if the bulk of updates consists mainly of insertions the size of the whole data distribution increases, thus if the size of the histogram is kept constant the accuracy obviously decreases as new data are inserted. Analogously, if the bulk of updates consists mainly of deletions the size of the whole data distribution decreases, a new histogram within the same storage space bound provides higher accuracy. Thus we performed other experiments keeping the compression ratio constant (i.e. the ratio between the size of the data and that of the histogram): that is, instead of keeping B constant, at each invocation of the incremental algorithm, we re-computed B according to the size of the updated data. Fig. 4(c,f) depict how error rates change w.r.t. update size (compression ratio was kept equal to 50). In these experiments the same query workloads of Fig. 4(b,e) were used. They show that error rates are almost unaffected by changes in update size.

Conclusions

We have proposed a new technique for constructing multi-dimensional histograms providing high accuracy for selectivity estimation. Our technique invokes a density-based clustering algorithm to partition data into dense and sparse regions which are further partitioned according to a grid-based scheme. We have extended this approach to the case of dynamic data. In this context we have designed a technique exploiting an in-

cremental cluster analysis strategy to propagate data updates to the histogram which aims at limiting the amount of data to be re-processed. We have tested the effectiveness of the incremental approach showing that it yields a relevant speedup factor (w.r.t. the non-incremental one) and that its adoption preserves accuracy as data changes.

Future work will aim at considering different clustering techniques to be embedded into our approach, in order to study how they can be exploited to improve the histogram construction cost while preserving its accuracy. Moreover the effectiveness of combining our approach with techniques for reducing data dimensionality (such as SPARTAN [2]) will be investigated.

Acknowledgements. The authors are grateful to Giuseppe Manco for fruitful discussions and valuable comments on several issues related to cluster analysis.

References

1. Acharya, S., Poosala, V., Ramaswamy, S., Selectivity estimation in spatial databases, *Proc. ACM SIGMOD Conf. 1999*.
2. Babu, S., Garofalakis, M. N., Rastogi, R., SPARTAN: A Model-Based Semantic Compression System for Massive Data Tables, *Proc. ACM SIGMOD Conf. 2001*.
3. Bruno, N., Chaudhuri, S., Gravano, L., STHoles: a multi-dimensional workload aware histogram, *Proc. ACM SIGMOD Conf. 2001*
4. Chaudhuri, S., An Overview of Query Optimization in Relational Systems, *Proc. PODS 1998*.
5. Donjerkovic, D., Ioannidis, Y. E., Ramakrishnan, R., Dynamic Histograms: Capturing Evolving Data Sets, *Proc. ICDE 2000*.
6. Ester, M., Kriegel, H. P., Sander, J., Xu, X., A density-based algorithm for discorvering clusters in large spatial databases with noise, *Proc. KDD 1996*.
7. Ester, M., Kriegel, H. P., Wimmer, M., Xu, X., Incremental clustering for mining in a data warehousing environment, *Proc. VLDB 1998*.
8. Furfaro, F., Mazzeo, G., M., Sirangelo, C., Clustering-Based Histograms for Multi-dimensional Data, *Proc. DaWaK 2005*.
9. Garofalakis, M., Gibbons, P.B., Wavelet Synopses with Error Guarantees, *Proc. ACM SIGMOD Conf. 2002*.
10. Gibbons, P. B., Matias, Y., Poosala, V., Fast Incremental Maintenance of Approximate Histograms, *Proc. VLDB 1997*.
11. Guha, S., Indyk, P., Muthukrishnan, M., Strauss, M., Histogramming Data Streams with Fast Per-Item Processing, *Proc. ICALP 2002*.
12. Gunopulos, D., Kollios, G., Tsotras, V. J., Domeniconi, C., Selectivity estimators for multi-dimensional range queries over real attributes, *The VLDB Journal*, Vol. 14(2), April 2005.
13. Kaufman, L., Rousseeuw, P. J., *Finding Groups in Data: An Introduction to Cluster Analysis*, Wiley, 2005.
14. Korn, F., Johnson, T., Jagadish, H. V., Range Selectivity Estimation for Continuous Attributes, *Proc. SSDBM 1999*.
15. Mamoulis, N., Papadias, D., Selectivity Estimation Of Complex Spatial Queries, *Proc. SSTD 2001*.
16. Poosala, V., Ioannidis, Y. E., Selectivity estimation without the attribute value independence assumption, *Proc. VLDB 1997*.
17. Shanmugasundaram, J., Fayyad, U., Bradley, P. S., Compressed data cubes for OLAP aggregate query approximation on continuous dimensions, *Proc. KDD 1999*.