Reducing Data Stream Sliding Windows by Cyclic Tree-Like Histograms*

Francesco Buccafurri and Gianluca Lax

DIMET, Università degli Studi Mediterranea di Reggio Calabria
Via Graziella, Località Feo di Vito, 89060 Reggio Calabria, Italy
bucca@unirc.it,lax@unirc.it

Abstract. Data reduction is a basic step in a KDD process useful for delivering to successive stages more concise and meaningful data. When mining is applied to data streams, that are continuous data flows, the issue of suitably reducing them is highly interesting, in order to arrange effective approaches requiring multiple scans on data, that, in such a way, may be performed over one or more reduced sliding windows. A class of queries, whose importance in the context of KDD is widely accepted, corresponds to *sum range queries*. In this paper we propose a histogram-based technique for reducing sliding windows supporting approximate arbitrary (i.e., non biased) sum range queries. The histogram, based on a hierarchical structure (opposed to the flat structure of traditional ones), results suitable for directly supporting hierarchical queries, and, thus, drill-down and roll-up operations. In addition, the structure well supports sliding window shifting and quick query answering (both these operations are logarithmic in the sliding window size). Experimental analysis shows the superiority of our method in terms of accuracy w.r.t. the state-of-the-art approaches in the context of histogram-based sliding window reduction techniques.

1 Introduction

It is well known that data pre-processing techniques (data cleaning and data reduction), when applied prior to mining, may significantly improve the overall data mining results. This is particularly true in the context of data stream mining, where data comes continuously and mining may be done on the basis of sliding windows including only the most recent data [1]. Indeed, in order to give significance to the sliding window itself, the size, that is the number of most recent data we keep in each instant, should be as large as possible. As a consequence any technique capable of reducing (i.e., compressing) sliding windows by maintaining a good approximate representation of data distribution inside it,

* This work was partially funded by the Italian National Council Research under the "Reti Internet: efficienza, integrazione e sicurezza" project and by the European Union under the "SESTANTE - Strumenti Telematici per la Sicurezza e l'Efficienza Documentale della Catena Logistica di Porti e Interporti" Interreg III-B Mediterranee Occidentale project

J.-F. Boulicaut et al. (Eds.): PKDD 2004, LNAI 3202, pp. 75–86, 2004.

and, at the same time, by smoothing possible outliers, is certainly relevant in the field of data stream mining. Observe that, reducing sliding windows allows us also to keep simultaneously more than just one approximate sliding window, in order to implement similarity queries and other analysis, like *change mining queries* [12], useful for trend analysis and, in general, for understanding the dynamics of the data stream. In sum, since in a typical streaming environment, only limited memory resources are available [14], reduction is a key factor allowing us query processing also requiring multiple scans on data. But, which properties a sliding window reduction technique has to satisfy? Necessarily, the reduced sliding window should maintain in a certain measure the *semantic nature* of original data, in such a way that meaningful queries for mining activities can be submitted to reduced data in place of original ones. Then, for a given kind of query, accuracy of the reduced structure should be enough independent of the position where the query is applied. Indeed, mining needs the possibility of freely querying data. In addition, the reduction technique should not to limit too much the capability of drilling-down and rolling-up data.

In this paper we propose a histogram-based technique for reducing sliding windows supporting approximate arbitrary range-sum queries satisfying all the above properties. Observe that range-sum queries represents a class of queries very frequent in the field of data stream mining. Our histogram, called *c-tree*, differently from traditional ones, is based on a hierarchical structure. Its nodes contain, hierarchically, pre-computed range-sum queries, stored by approximate (via bit saving) encoding. For this reason, the structure directly supports the estimation of arbitrary range-sum queries (indeed, range-sum queries are either embedded in the histogram or derivable by linear interpolation by the latter ones). Reduction derives both from aggregation implemented by leaves of the tree (discretization), and from the saving of bits obtained by representing range queries with less than 32 bits (assumed enough for an exact representation). The number of bits used for representing range queries decreases as the level of the tree increases. The structure is designed as dynamic, in the sense that each update, for maintaining the c-tree on the sliding window, can be applied in logarithmic time in worst case (w.r.t. the window size). Moreover, answering to a range query requires at most logarithmic time too. Observe that hierarchical structure directly supports querying at different abstraction levels, thus allowing drill-down and roll-up operations. Finally, bucket summarization smoothes each data value by consulting the "neighborhood" or values around it. This works to remove the noise from data. But the main feature we have to remark for our histogram concerns its accuracy. Indeed, in order the reduction technique to have significance, error should be either guaranteed or heuristically shown to be low (and this is our case), compared with that of the state-of-the-art techniques. There is no large literature about the important issue of evaluating approximate arbitrary range queries on sliding windows. Most of the recent approaches are based on histograms [17, 16] and Wavelet [15, 21]. Other approaches use sampling [2, 19, 8] and sketches [7, 13]. Histograms are a lossy compression technique widely applied in various application contexts, like query optimization, statistical

and temporal databases, and OLAP applications. [11] deals with the problem of reducing sliding windows by error guaranteed histograms (called *exponential histograms*), by solving the problem only in case of biased range queries (i.e., queries involving the last q data of the sliding window) that are significant queries in the context of data stream processing. For arbitrary queries, error may increase dramatically, especially if queries are distant from the most recent data of the sliding window. The proposal of [6] presents the same characteristics. Unfortunately, limiting range queries to a particular case is not acceptable in the KDD context, as observed earlier. Thus, having a technique not focused on analytic error guarantee, but experimentally shown to be uniformly accurate w.r.t. arbitrary range queries is strongly preferable to biased (even error guaranteed) methods.

Validation of our method is conducted experimentally by comparing the c-tree with the V-optimal histogram [18]. We have chosen such a histogram since its superiority (in terms of accuracy) w.r.t. the state-of-the-art proposals is proven in a recent paper [16]. Actually, [16] concerns a more efficient version of V-optimal (called ϵ-*approximate V-optimal histogram*) defined in order to have an effective method (since V-optimal updating is polylinear). But, in the same paper, it is shown that the ϵ-approximate V-optimal histogram is (slightly) less accurate than the classical V-optimal defined in [18]. This guarantees the significance of our comparison. In addition, we observe that while our method requires $O(\log\ w)$ time for answering to a range query (where w is the size of the sliding window), V-Optimal requires $O(w)$ time (by assuming that the number of bucket is linear on the sliding window size). [16] shows that the ϵ-approximate approach is also definitely superior w.r.t. Wavelet approach [21]. Anyway, we perform comparisons also with Wavelet histograms [20]. Observe that, in order to increase significance of our experiments, we have used exact V-optimal [18] and Wavelet [20] histograms, that are more accurate than histograms defined in [16] and [21], respectively. Indeed, the latter two approaches were introduced since both [18] and [20] are not effective in the context of data streams due to their high maintenance computational cost.

The plan of the paper is the following. The c-tree histogram is introduced in Section 2 where we describe also the dynamics of the c-tree, that is how it is updated while the sliding window is moved. Section 3 shows how the c-tree histogram is represented; moreover some considerations about the proposed approach are remarked. In Section 4 we show how to use the c-tree for evaluating range queries. Section 5 analyzes experimental results validating the method. In Section 6 we draw our conclusions.

2 The c-Tree Histogram

In this section we describe the core of our proposal, consisting of a tree-like histogram, named *c-tree*, used for managing data streams under sliding windows. A *sliding window* of size w[1] (on a data stream D) is the tuple $\langle x_w, x_{w-1}, \ldots, x_1 \rangle$

[1] For simplicity, we assume that $w = 2^z$ for a given positive integer $z > 0$

containing the w-most recent values of D in arrival ordering (observe that x_w represents the oldest value whereas x_1 is the most recent one). Clearly, at each new time instant the sliding window is updated by inserting from the right hand the new value of D and deleting the left-most one (corresponding to the oldest value of the sliding window).

The histogram is built on top of the sliding window, by hierarchically summarizing the values occurring in it. In order to describe the c-tree we chose a constructive fashion. In particular we define the initial configuration (at the time instant 0 – coinciding with the origin of the data flow) and we show, at a generic instant coinciding with the arrival of a new data, how the c-tree is updated.

Initial Configuration. The c-tree histogram consists of:

1. A full binary tree T with n levels, where n is a parameter set according to the required data reduction (this issue will be treated in Section 5). Each leaf node N of T is associated with a range $\langle l(N), u(N)\rangle$ of size $d = \frac{w}{2^{n-1}}$ and the set of such ranges produces an equi-width partition of the array $\langle 1, w\rangle$. In addition, we require that adjacent leaves correspond to adjacent ranges of $\langle 1, w\rangle$ and the left-most leaf corresponds to the range $\langle 1, d\rangle$. We denote by $Val(N)$ the value of a node N. In the initial state, all nodes of T contains the value 0.

2. A buffer (of size 2) $B = \langle e, s\rangle$, where $0 \leq e < d$ and $s \geq 0$. s represents the sum of the e most recent elements of the sliding window. Initially, $e = s = 0$.

3. An index P, with $1 \leq P \leq 2^{n-1}$, identifying a leaf node of T. P is initially set to 1, and thus it identifies the left-most leaf of T.

We denote by H the above data structure. Now we describe how H is updated when new data arrive.

State Transition. Let x_t be the data coming at the instant $t > 0$. Then, $e := (e+1)$ **mod** d and $s := s + x_t$. Now, if $e \neq 0$ (i.e., the buffer B is not full), then the updating of H halts. Otherwise (i.e., $e = 0$), the value s (which summarizes the last d data) has to be stored in T and, then, the buffer has to be emptied. We explain now how the insertion of s in T is implemented. Let $\delta = s - val(N_P)$, where N_P is the leaf of T identified by P. Then, $val(N_P) := val(N_P) + s$ and δ is also added to all nodes belonging to the path from N_P to the root of H. Finally, e and s of B are reset (i.e., they assume value 0) and $P := P$ **mod** $2^{n-1} + 1$ (this way, leaf nodes of the tree are managed as a cyclic array). Observe that P points to the leaf node containing the less recent data, and such data are replaced by new data incoming. Each update operation requires $O(\log w)$ time, where w is the size of the sliding window. Now we show an example of 3-levels c-tree building and updating.

Example 1. Let $\langle 35, 51, 40, 118, 132, 21, 15, 16, 18, 29, ...\rangle$ be the data stream order by arrival time increasing and let the sliding window size be 8; moreover, let $d = w/2^{n-1} = 2$. Initially, $e = 0$, $s = 0$, $P = 1$ and the value of all nodes of T is 0. The first data coming from the stream is 35, thus $e = 1$ and $s = 35$. Since $e \neq 0$ no other updating operation has to be done. Then, the data coming from the stream is 51, thus $e = 0$ and $s = 35 + 51 = 86$. Since $e = 0$, the first leaf node of T is set to the value s, and all nodes belonging to the path between

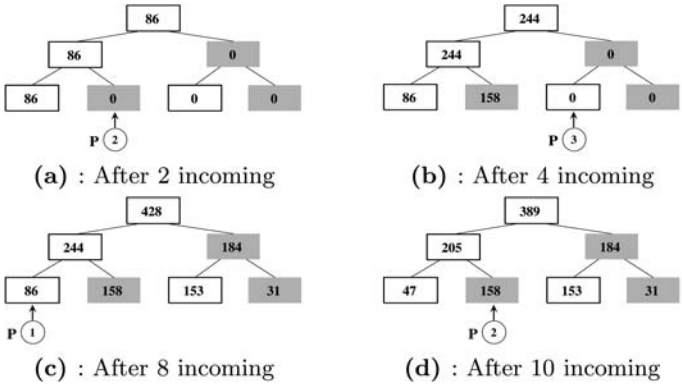


Fig. 1. The c-tree of Example 1

such leaf and the root are increased by $\delta = 86$. Finally, $P = 2$, and e and s are reset. In Figure 1.(a) the resulting c-tree is reported. Therein (as well as in the other figures of the example), we have omitted buffer values since they are null. Moreover, right-hand child nodes are represented with the color grey. This is because, as we will explain in Section 3, these nodes have not to be saved since they can be computed from white nodes. For the first 8 data arrivals, updates proceeds as before. In Figure 1.(b) and 1.(c) we report just the snapshot after 4 and 8 updates, respectively. The pointer P is now changed assuming the value 1. Now, the data 18 arrives. Thus, $e = 1$ and $s = 18$. At the next time instant, $e = 0$ and $s = 47$. Since $e = 0$, $\delta = 47 - 86 = -39$ is added to the leaf node pointed by P (that is the first leaf) determining its new value 47. Moreover, nodes belonging to the path between such leaf and the root are increased by δ. At this point, P assumes the value 2. The final c-tree is shown in Figure 1.(d).

3 c-Tree Representation

In this section we describe how the c-tree histogram is represented. Beside storing just necessary nodes, we use a bit-saving based encoding in order to reduce storing space. As already sketched in Example 1, each right-hand child node can be derived as a difference between the value of the parent node and the value of the sibling node. As a consequence, right-hand child node values have not to be stored. In addition, we encode node values through length-variable representations. In particular: (1) The root is encoded by 32 bits (we assume that anyway the overall sum of the sliding window data can be represented by 32 bits with no scaling error). (2) The root left-child is represented by k bits (where k is a parameter suitably set – we will discuss about such an issue in Section 5). (3) All nodes which belong to the same level are represented by the same number of bits. (4) Nodes belonging to a level, say l, with $2 \leq l \leq n - 1$, are represented by a bit less than nodes belonging to the level $l - 1$.

Substantially, the approach is based on the assumption that, in the average, the sum of occurrences of a given interval of the frequency vector, is twice than the sum of the occurrences of each half of such an interval. This assumption is chosen as a heuristic criterion for designing c-tree, and this explains the choice of reducing by 1 per level the number of bits used for representing numbers. Clearly, the sum contained in a given node is represented as a fraction of the sum contained in the parent node. Observe that, in principle, it could be used also a representation allowing possibly different number of bits for nodes belonging to the same level, depending on the actual value contained into nodes. However, we should deal with the spatial *overhead* due to these variable codes. The reduction of 1 bit per level appears as a reasonable compromise. Our approach is validated by previous results shown in [3, 4] for histograms on persistent data and in [5] for improving estimation inside histogram buckets.

Remark. We remark that this bit-saving approach is not applicable to non-indexed histograms (by a tree). Indeed, for a "flat" histogram, the scaling size used for representing numbers would be related to the overall sliding window sum value, that is, bucket values would be represented as a fraction of this overall sum, with a considerable increasing of the scaling error. One could argue that also data-distribution-driven histograms, like V-Optimal [18], whose accuracy has been widely proven in the literature, could be improved by building a tree index on top, and by reducing the storage space by trivially applying our bit-saving approach. However, such indexed histograms, induce a *non* equi-width partition, and, as a consequence, the reduction of 1 bit per level in the index of our approach would be not well founded.

Encoding a given node N with a certain number of bits, say i, is done in a standard fashion. Let denote by P the parent node of N. The value $val(N)$ of the node N will be recovered not exactly, in general. It will be affected by a certain scaling approximation. We denote by $\widetilde{val^i}(N)$ the encoding of $val(N)$ done with i bits and by $\overline{val^i}(N)$ the approximation of $val(N)$ obtained by $\widetilde{val^i}(N)$.
We have that: $\widetilde{val^i}(N) = Round(\frac{val(N)}{val(P)} \cdot (2^i - 1))$. Clearly, $0 \leq \widetilde{val^i}(N) \leq 2^i - 1$.
Concerning the approximation of $val(N)$ it results: $\overline{val^i}(N) = (\frac{\widetilde{val^i}(N)}{2^i - 1} \cdot val(P))$
The *absolute error* due to the i-bit encoding of the node N, with parent node P, is: $\epsilon_a(val(N), val(P), i) = |val(N) - \overline{val^i}(N)|$. It can be easily verified that: $0 \leq \epsilon_a(val(N), val(P), i) \leq \frac{val(P)}{2^{i+1}}$.

We conclude this section by analyzing both overall scaling error and storage space required by the c-tree, once the two input parameters are fixed, that is: n, i.e., the number of levels n and k, i.e., the number of bits uses for encoding the left-hand child of the root. Concerning scaling error we have to understand how it is propagated over the path from the root to the leaves of the tree. Indeed, the error for a stand-alone node is analyzed above. We may determine an upper bound of the worst-case error by considering the sum of the maximum scaling error at each stage. Assume that R is the maximum value appearing in the data stream and w is the sliding window size. According to considerations above, since at the first level we use k bits for encoding numbers, the maximum absolute error

at this level is $\frac{R \cdot w}{2^{k+1}}$. Going down to the second level cannot increase the maximum error. Indeed, we double the scale granularity (since coding is reduced by 1 bit) but the maximum allowed value is halved. More precisely, the maximum absolute error at the second level is $\frac{R \cdot \frac{w}{2}}{2^k}$. Clearly, the same reasoning can be applied to lower levels, so that the above claim is easily verified. In sum, the maximum absolute scaling error of the c-tree is $\frac{R \cdot w}{2^{k+1}}$; interestingly, observe that the error is independent of the tree depth n.

Concerning the storage space (in bits) required by the c-tree, we have:

$$(n-1) + \lceil \log(R \cdot d) \rceil + \lceil \log(d) \rceil + 32 + \sum_{h=0}^{n-2} (k-h) \cdot 2^h \tag{1}$$

where $d = \frac{w}{2^{n-1}}$ and the first three components of the sum takes account of P, s and e, respectively, while $32 + \sum_{h=0}^{n-2} (k-h) \cdot 2^h$ is the space required for saved nodes of T (recall that only left child nodes are stored). In Section 5 we will discuss about the setting of the parameters n and k.

4 Evaluation of a Range-Sum Query

In this section we describe the algorithm used for evaluating the answer to a range query $Q(t_1, t_2)$, where $0 \leq t_1 < t_2 < w$, that computes the sum of data arrived between time instants $t - t_1$ and $t - t_2$, respectively, where t denotes the current time instant. For example, if $t_1 = 0$ and $t_2 = 5$ it represents the sum of the 5 most recent data. C-tree allows us to reduce the storage space required for storing data of the sliding windows and, at the same time, to give fast yet approximate answers to range queries. As usual in this context, this approximation is the price we have to pay for having a small data structure to manage and for obtaining fast query answering. We now introduce some definitions that will be use in the algorithm.

Notations: Given a range query $Q(t_1, t_2)$, with $t_1 > e$: (1) Let η be the set of leaf nodes containing at least one data involved in the range query. Let $l_i = (P - \mathbf{ceil}\langle \frac{t_i - e}{d} \rangle)/2^{n-1}$, with $i = 1, 2$ be the indexes of the two leaf nodes L_1 and L_2. η consists of all leaf nodes succeeding L_2 and preceding L_1 (including both L_1 and L_2) in the ordering obtained by considering leaf nodes as a cyclic array. (2) Given a non leaf node of N, let *L(N)* be the set of leaf nodes descending from N. (3) Given a leaf node N, we define $I(N, Q) = \frac{i}{d} \cdot val(N)$, where i is the number of data stored by N involved in Q. $I(N, Q)$ computes the contribution of a leaf node to a range query (by linear interpolation). (4) Let $\overline{Q}$ be the estimation of the range query computed by means the c-tree.

First, suppose that $t_1 > e$ (recall e is the number of data in the buffer B) in such a way that the range query doesn't involve data in the buffer B. The algorithm for the range query evaluation is performed by calling the function *contribution* on the root of the c-tree.

The function *contribution* is shown below:

```
function contribution (N)
    if (N is a leaf)
        Q = Q + I(N, Q)
        return Q //the function halts.
    endif else
        for each N_x child of N
            if (L(N_x) ⊆ η)
                Q = Q + val(N_x) endif
            if (L(N_x) ∩ η ≠ ∅)
                contribution (N_x) endif
endfunction
```

The first test checks if N is a leaf node, and in such a case the function, before halting, computes the contribution of N to Q by linear interpolation. In case N is not a leaf node, it is tested if all nodes descending from N_x (denoting a child of N) are involved in the query. If this is the case, their contribution to the range query coincides with the value of N_x. In case not all nodes descending from N_x are involved in the query, but only some of them, their contribution is obtained by recursively calling the function on N_x. The algorithm performs, in the worst case, two descents from the root to two leaves. Thus, asymptotic computational cost of answering a range query is $O(\log w)$ where, w is the window size. Note that the exact cost is upper bounded by n, where $n = \lceil \log \frac{w}{d} \rceil + 1$ is the number of levels of the c-tree and d is the size of leaf nodes.
In case $t_1 < t_2 \leq e$, the range query involves only data in the buffer B and $Q(t_1, t_2) = (t_2 - t_1) \cdot \frac{s}{e}$ (recall that s represents the sum of data buffered in B). Finally, in case $t_1 \leq e < t_2$, we have that $Q(t_1, t_2) = Q(t_1, e) + Q(e, t_2)$ that can be computed by exploiting the two above cases.

5 Experiments

We start this section by describing the test bed used for our experiments.

Available Storage: For experiments conducted we have used 22 four-byte numbers for all techniques. According to (1) (given in Section 3), the above constraint has to be taken into account when the two basic parameters n and k of the c-tree are set. We have chosen to fix these parameters to values $n = 7$ and $k = 14$ (we will motivate such a choice next in this section).

Techniques: We compare our technique with (the motivations of such a choice are given in the Introduction): (1) *V-Optimal* (VO) [18], which produces 11 bucket; for each bucket both upper bound and value are stored; (2) *Wavelet* (WA) [20], which are constructed using the *bi-orthogonal* 2.2 decomposition of the MATLAB 5.3 with 11 four-byte Wavelet coefficients plus another 11 four-byte numbers for storing coefficient positions.

Synthetic Data Streams: Synthetic data streams are obtained by randomly generating 10000 data values belonging to the range $[0, 100]$.

Real-Life Data Streams: Real-life data have been retrieved from [10] and represent the daily maximum air temperature stored by the station STBARBRA.A in the County of Santa Barbara from 1994 to 2001. Its size is 2922 and the range is from 10.6 to 38.3 degree Celsius.

Query Set and Error Metric: In our experiments, we use two different query sets for evaluating the effectiveness of the various methods: (1) QS_1 consists of all range queries from 1 to q with $1 \leq q \leq w$ and (2) QS_2 is the set of all range queries having size $round(\frac{w}{10})$, where, we recall, w is the size of the sliding window. At each time we measure the error $E(t)$ produced by techniques on the above query set by using the average of the relative error $\frac{1}{Q}\sum_{i=1}^{Q} e_i^{rel}$, where Q is the cardinality of the query set, and e_i^{rel} is the *relative error* , i.e., $e_i^{rel} = \frac{|S_i - \widetilde{S}_i|}{S_i}$, where S_i and $\widetilde{S}_i$ are, respectively, the actual answer and the estimated answer of the query i-th of the considered query set. Then we compute the average of the error $E(t)$ over the entire data stream duration. After a suitable initial delay sufficient to fill the sliding window, queries are applied at each new arrival.

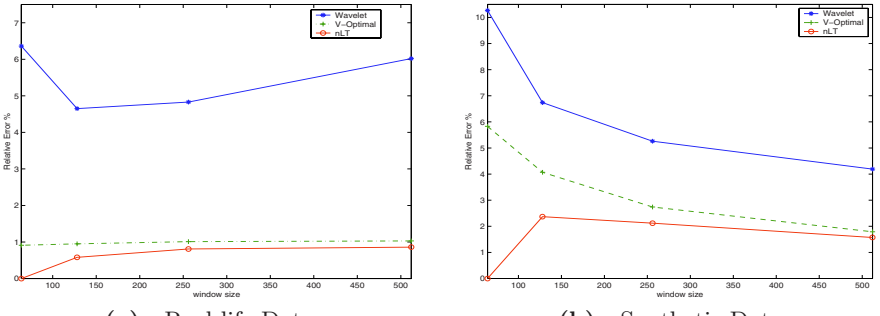
Sliding Window Size: In our experiments, we use sliding windows of size 64, 128, 256, 512, 1024, that are dimensions frequently used for experiments in this context (e.g., see [6, 9, 16]).

Now we consider the problem of the choice of a suitable value for n and k, that are, we recall, number of levels of the c-tree and number of bits used for encoding the left child node of the root (for the successive levels, as already mentioned, we drop 1 bit per level) respectively. Observe that, according to the result about the error given in Section 3, setting the parameter k means fixing also the error due to scaling approximation. We have performed some experiments on synthetic data in order to test the error dependence on the parameters n and k for different window sizes by using the average relative error on query set 1. Experiments produce similar curves (not reported here for space limitations) showing that the error decreases as n increases and decreases as k increases until $k = 11$ and then it remains near constant. Indeed, the error consists of two components: (1) the error due to the interpolation inside the leaves nodes partially involved in the query, and (2) the scaling approximation. For $k > 11$, the last component is negligible, and the error keeps a quasi-constant behavior since the first component depends only on n. Therefore, in order to reduce the error, we should set k to a value as large as possible allowing us to represent leaves with a sufficient number of bits (not to much lower than the threshold heuristically determined above). However, for a fixed compression ratio, this may limit the depth of the tree and, thus, the resolution determined by the leaves. As a consequence, the error arising from linear interpolation done inside leaf nodes increases. In sum, the choice of k plays the role of solving the above trade-off. These criteria are employed in experiments in order to choose the value of n and k, respectively, on the basis of the storage space amount.

Now we present results obtained by experiments. For each data set we have calculated the average relative error on both query set 1 and query set 2.

In Figures 2.(a) and 2.(b) we have reported results obtained on real and synthetic data sets, respectively varying the sliding window size (we have considered sizes: 64, 128, 256, 512) and using query set 1. C-tree shows the best accuracy, with significant gaps especially respect to Wavelet. Note that c-tree in case of sliding window of size 64 does not produce error since there is no discretization and, furthermore, a leaf node is encoded by 9 bits which are sufficient to represent exactly a single data value.

In Figures 3.(a) and 3.(b) we have replicated the previous experiment considering the behavior of techniques on query set 2. Observe that accuracy of techniques becames worse on query set 2 since the range query size is very small (indeed, the range query involves only the 10% of sliding window data). Also this comparison shows the superiority of the c-tree over other histogram methods. In Figure 4.(a) and 4.(b) we have studied the accuracy of c-tree versus the number levels, fixing $k = 14$, with sliding windows of size 256, 512 and 1024. In this experiment we have used the query set 1. Finally, we observe that, thanks to experiments conducted with query set 2, we have verified that the behaviour of the c-tree is "macroscopically" independent of the position of the range query in the window. Macroscopically here means that even though some queries can be privileged (for instance those involving only entire buckets), it happens that both average and variability of the query answer error is not biased. This basically reflects the equi-width nature of the c-tree histogram.

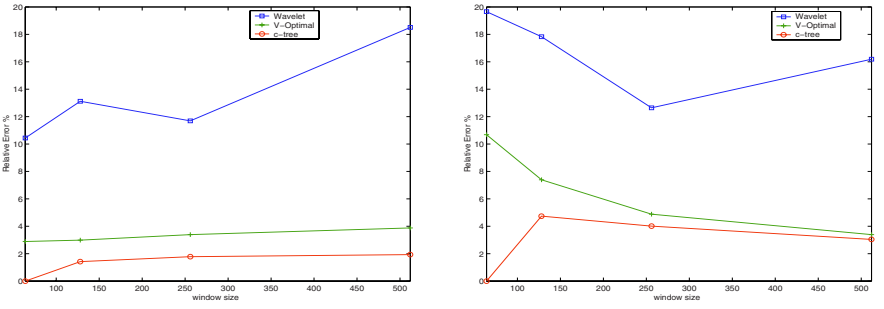


(a) : Real-life Data **(b)** : Synthetic Data

Fig. 2. Error for query set 1

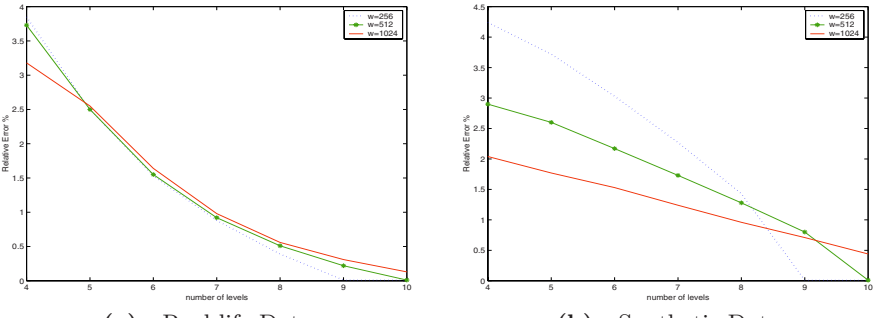
6 Conclusions and Future Work

In this paper we have presented a tree-like histogram used for reducing sliding windows and supporting fast approximate answers to arbitrary range-sum queries on them. Through a large set of experiments, the method is successfully compared with the most relevant proposals among the related histogram-based approaches. The histogram is designed for implementing data stream preprocessing in a KDD process which exploits arbitrary hierarchical range-sum



(a) : Real-life Data **(b)** : Synthetic Data

Fig. 3. Error for query set 2



(a) : Real-life Data **(b)** : Synthetic Data

Fig. 4. Error for query set 1

queries. Our feeling is that the c-tree histogram, as it encodes a concise multi-layer description of the sliding window, can be adapted for supporting further kinds of queries, also useful in the context of data stream mining. The study of this issue is left as a future research.

References

1. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream system. In *PODS*, pages 1–16, 2002.
2. B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. of the thirteenth annual ACM-SIAM Symposium on Discrete algorithms*, pages 633–634, 2002.
3. F. Buccafurri and G. Lax. Fast range query estimation by n-level tree histograms. *Data & Knowledge Engineering Journal.* To appear.
4. F. Buccafurri and G. Lax. Pre-computing approximate hierarchical range queries in a tree-like histogram. In *Proc. of the Int. Conf. on Data Warehousing and Knowledge Discovery, DaWak*, pages 350–359, 2003.

5. F. Buccafurri, L. Pontieri, D. Rosaci, and D. Saccà. Improving range query estimation on histograms. In *Proc. of the Int. Conf. on Data Engineering, ICDE*, pages 628–638, 2002.
6. A. Bulut and A. K. Singh. SWAT: Hierarchical stream summarization in large networks. In *ICDE*, pages 303–314, 2003.
7. M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. In *Proc. of the 29th Int. Colloquium on Automata, Languages and Programming*, pages 693–703. Springer-Verlag, 2002.
8. S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD Int. Conf. on Management of data*, pages 263–274. ACM Press, 1999.
9. A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *Proc. of the ACM SIGMOD Int. Conf. on Management of data*, pages 40–51. ACM Press, 2003.
10. California Weather Databases. http://www.ipm.ucdavis.edu/calludt.cgi/wxstationdata?map=santabarbara.html&stn=stbarbra.a.
11. M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows: (extended abstract). In *Proc. of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 635–644, 2002.
12. G. Dong, J. Han, L. V. S. Lakshmanan, J. Pei, H. Wang, and P. S. Yu. Online mining of changes from data streams: Research problems and preliminary results. In *Proc. of the 2003 ACM SIGMOD Workshop on Management and Processing of Data Streams*, 2003.
13. J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. In *Proc. of the 40th Annual Symposium on Foundations of Computer Science*, page 501. IEEE Computer Society, 1999.
14. M.N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look (tutorial). In *Proc. of the Int. Conf. on Management of Data ACM SIGMOD*, page 635, 2002.
15. A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *The VLDB Journal*, pages 79–88, 2001.
16. S. Guha and N. Koudas. Approximating a data streams for querying and estimation: Algorithms and performance evaluation. In *ICDE*, pages 567–576, 2002.
17. S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the thirty-third annual ACM symposium on Theory of computing*, pages 471–475. ACM Press, 2001.
18. H. V. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. C. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. 24th Int. Conf. Very Large Data Bases, VLDB*, pages 275–286, 24–27 1998.
19. G. S. Manku and R. Motwani. Approximate frequency counts over data streams. In *Proc. Int. Conf. on Very Large Data Bases*, pages 346–357, 2002.
20. Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of the 1998 ACM SIGMOD Int. Conf. on Management of data*, pages 448–459. ACM Press, 1998.
21. Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *The VLDB Journal*, pages 101–110, 2000.