Approximate Query Answering on Sensor Network Data Streams

Alfredo Cuzzocrea[1], Filippo Furfaro[2], Elio Masciari[1], Domenico Sacca[1], and Cristina Sirangelo[2]

[1] ICAR-CNR – Institute of Italian National Research Council
{cuzzocrea, masciari}@isi.cs.cnr.it
[2] DEIS-UNICAL
Via P. Bucci, 87036 Rende (CS) Italy
{furfaro, sirangelo}@si.deis.unical.it

Abstract. Sensor networks represent a non traditional source of information, as readings generated by sensors flow continuously, leading to an infinite stream of data. Traditional DBMSs, which are based on an exact and detailed representation of information, are not suitable in this context, as all the information carried by a data stream cannot be stored within a bounded storage space. Thus, compressing data (by possibly loosing less relevant information) and storing their compressed representation, rather than the original one, becomes mandatory. This approach aims to store as much information carried by the stream as possible, but makes it unfeasible to provide exact answers to queries on the stream content. However, exact answers to queries are often not necessary, as approximate ones usually suffice to get useful reports on the world monitored by the sensors. In this paper we propose a technique for providing fast approximate answers to aggregate queries on sensor data streams. Our proposal is based on a hierarchical summarization of the data stream embedded into a flexible indexing structure, which permits us to both access and update compressed data efficiently. The compressed representation of data is updated continuously, as new sensor readings arrive. When the available storage space is not enough to store new data, some space is released by compressing the "oldest" stored data progressively, so that recent information (which is usually the most relevant to retrieve) is represented with more detail than old one.

1 Introduction

Sensors are non-reactive elements which are used to monitor real life phenomena, such as live weather conditions, network traffic, etc. They are usually organized into networks where their readings are transmitted using low level protocols [11]. Sensor networks represent a non traditional source of information, as readings generated by sensors flow continuously, leading to an infinite stream of data. Traditional DBMSs, which are based on an exact and detailed representation of information, are not suitable in this context, as all the information carried by a data stream cannot be stored within a bounded storage space [1–3, 9, 10].

Moreover, traditional DBMSs are basically transaction oriented, i.e. their main goal is to guarantee data consistency, and they do not pay particular attention to query efficiency. The inefficiency of the query answering process is dramatically evident in the computation of aggregate queries (sum, mean, count, etc.), as accessing a huge amount of data is usually necessary to provide the correct answer to this kind of query.

The issue of defining new query evaluation paradigms to provide fast answers to aggregate queries is very relevant in the context of sensor networks. In fact, the amount of data produced by sensors is very large and grows continuously, and the queries need to be evaluated very quickly, in order to make it possible to perform a timely "reaction to the world" . Moreover, in order to make the information produced by sensors useful, it should be possible to retrieve an up-to-date "snapshot" of the monitored world continuously, as time passes and new readings are collected. For instance, a climate disaster prevention system would benefit from the availability of continuous information on atmospheric conditions in the last hour. Similarly, a network congestion detection system would be able to prevent network failures exploiting the knowledge of network traffic during the last minutes. If the answer to these queries, called *continuous queries*, is not fast enough, we could observe an increasing delay between the query answer and the arrival of new data, and thus a not timely reaction to the world.

In this paper we propose a technique for providing fast approximate answers to aggregate queries on sensor data streams. Our proposal is based on a hierarchical summarization of the data stream embedded into a flexible indexing structure, which permits us to both access and update compressed data efficiently. The compressed representation of data is updated continuously, as new sensor readings arrive. When the available storage space is not enough to store new data, some space is released by compressing the "oldest" stored data progressively, so that recent information (which is usually the most relevant to retrieve) is represented with more detail than old one.

Plan of the paper The paper is organized as follows. In Section 2, a model based on the use of two-dimensional arrays for the representation of sensor readings is defined, and the problem of evaluating range queries on this representation of sensor data streams is introduced. In Section 3, the notion of *quad-tree window* (i.e. a hierarchical representation of the readings produced by sensors within a given time interval) is presented. In particular, in Section 3.3 a compact physical representation of quad-tree windows is introduced. In Section 3.4 the representation of a sensor data stream by means of a list of quad-tree windows is presented, by showing how quad-tree windows can be created and populated as time passes and new sensor readings arrive. In Section 4, the *Multi-Resolution Data Stream Summary* (MRDS) is defined as a list of indexed clusters of quad-tree windows. In particular in Section 4.1 the *4-ary tree index* (an index which permits us to locate quad-tree windows efficiently within a given cluster) is defined. In Section 4.2, a compact physical representation of 4-ary tree indices is presented. In Section 4.3 and Section 4.4 the construction of the overall Multi-Resolution Data Stream Summary is described, by showing how 4-ary tree indices are dynamically constructed on the underlying quad-tree windows, and linked together, as new data arrive. In Section 5 the progressive compression of a Multi-Resolution Data Stream Summary is described, and in Section 6 the problem of answering range queries and continuous queries on a sensor data stream by accessing only the summary is investigated.

2 Problem Statement

Consider an ordered set of n sources (i.e. sensors) denoted by $\{s_1, \ldots, s_n\}$ producing n independent streams of data, representing sensor readings. Each data stream can be viewed as a sequence of triplets $\langle id_s, v, ts\rangle$, where:

1. $id_s \in \{1, .., n\}$ is the source identifier;
2. v is a non negative integer value representing the measure produced by the source identified by id_s;
3. ts is a *timestamp*, i.e. a value that indicates the time when the reading v was produced by the source id_s.

The data streams produced by the sources are caught by a *Sensor Data Stream Management System* (SDSMS), which combines the sensor readings into a unique data stream, and supports data analysis.

An important issue in managing sensor data streams is aggregating the values produced by a subset of sources within a time interval. More formally, this means answering a *range query* on the overall stream of data generated by $s_1, \ldots, s_n$. A range query is a pair $Q = \langle s_i..s_j, [t_{start}..t_{end}]\rangle$ whose answer is the evaluation of an aggregate operator (such as *sum*, *count*, *avg*, etc.) on the values produced by the sources $s_i, s_{i+1}, \ldots, s_j$ within the time interval $[t_{start}..t_{end}]$.

The sensor data stream can be represented by means of a two-dimensional array, where the first dimension corresponds to the set of sources, and the other one corresponds to time. In particular, the time is divided into intervals Δt_j of the same size. Each element $\langle s_i, \Delta t_j\rangle$ of the array is the sum of all the values generated by the source s_i whose timestamp is within the time interval Δt_j. Obviously the use of a time granularity generates a loss of information, as readings of a sensor belonging to the same time interval are aggregated. Indeed, if a time granularity which is appropriate for the particular context monitored by sensors is chosen, the loss of information will be negligible.

Using this representation, an estimate of the answer to a sum range query over $\langle s_i..s_j, [t_{start}..t_{end}]\rangle$ can be obtained by summing two contributions. The first one is given by the sum of those elements which are completely contained inside the range of the query (i.e. the elements $\langle s_k, \Delta t_l\rangle$ such that $i \leq k \leq j$ and Δt_l is completely contained into $[t_{start}..t_{end}]$). The second one is given by those elements which partially overlap the range of the query (i.e. the elements $\langle s_k, \Delta t_l\rangle$ such that $i \leq k \leq j$ and $t_{start} \in \Delta t_l$ or $t_{end} \in \Delta t_l$). The first of these two contributions does not introduce any approximation, whereas the second one is generally approximate, as the use of the time granularity makes it unfeasible to retrieve the exact distribution of values generated by each sensor within the same interval Δt_l. The latter contribution can be evaluated by performing linear interpolation, i.e. assuming that the data distribution inside each interval Δt_i is uniform (*Continuous Values Assumption - CVA*). For instance, the contribution of the element $\langle s_2, \Delta t_3 \rangle$ to the sum query represented in Fig. 1 is given by $\frac{6-5}{2} \cdot 4 = 2$.

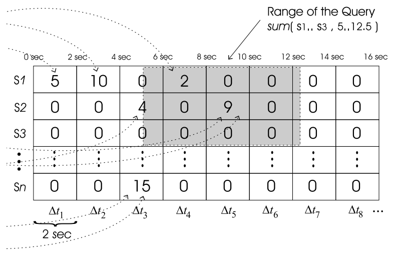


Fig. 1. Two-dimensional representation of sensor data streams

As the stream of readings produced by every source is potentially "infinite", detailed information on the stream (i.e. the exact sequence of values generated by every sensor) cannot be stored, so that exact answers to every possible range query cannot be provided.

However, exact answers to aggregate queries are often not necessary, as approximate answers usually suffice to get useful reports on the content of data streams, and to provide a meaningful description of the world monitored by sensors.

A solution for providing approximate answers to aggregate queries is to store a compressed representation of the overall data stream, and then to run queries on the compressed data. The use of a time granularity introduces a form of compression, but it does not suffice to represent the whole stream of data, as the stream length is possibly infinite. An effective structure for storing the information carried by the data stream should have the following characteristics:

1. it should be efficient to update, in order to catch the continuous stream of data coming from the sources;
2. it should provide an up-to-date representation of the sensor readings, where recent information is possibly represented more accurately than old one;
3. it should permit us to answer range queries efficiently.

Our proposal. In this paper we propose a technique for providing (fast) approximate answers to aggregate queries on sensor data streams, focusing our attention on *sum* range queries. Our proposal consists of a compressed representation of the sensor data stream where the information is summarized in a hierarchical fashion. In particular, a flexible indexing structure is embedded into the compressed data, so that information can be both accessed and updated efficiently.

In more detail, our compression technique is based on the following scheme:

– the sensor data stream is divided into "time windows" of the same size: each window consists of a finite number of contiguous unitary time intervals Δt_i (the size of each Δt_i corresponds to the granularity);
– time windows are indexed, so that windows involved in a range query can be accessed efficiently;
– as new data arrive, if the available storage space is not enough for their representation, "old" windows are compressed (or possibly removed) to release the storage space needed to represent new readings, and the index is updated to take into account the new data.

The technique used for compressing time windows is *lossy*, so that "recent" data are generally represented more accurately than "old" ones. In Fig. 2, the partitioning scheme of a stream into time windows is represented, as well as the overlying index referring to all the time windows.

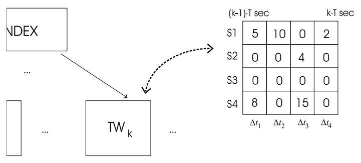


Fig. 2. A sequence of indexed time windows

3 Representing Time Windows

3.1 Preliminary Definitions

Consider given a two-dimensional $n_1 \times n_2$ array A. Without loss of generality, array indices are assumed to range respectively in $1..n_1$ and $1..n_2$. A *block* r (of the array) is a two dimensional interval $[l_1..u_1, l_2..u_2]$ such that $1 \leq l_1 \leq u_1 \leq n_1$ and $1 \leq l_2 \leq u_2 \leq n_2$. Informally, a block represents a "rectangular" region of the array. We denote by $size(r)$ the size of the block r, i.e. the value $(u1 - l_1 + 1) \cdot (u_2 - l_2 + 1)$. Given a pair $\langle v_1, v_2 \rangle$ we say that $\langle v_1, v_2 \rangle$ is inside r if $v_1 \in [l_1..u_1]$ and $v_2 \in [l_2..u_2]$. We denote by $sum(r)$ the sum of the array elements occurring in r, i.e. $sum(r) = \sum_{\langle i,j \rangle inside\, r} A[i,j]$. If r is a block corresponding to the whole array (i.e. $r = [1..n_1, 1..n_2]$), $sum(r)$ is also denoted by $sum(A)$. A block r such that $sum(r) = 0$ is called a *null block*. Given a block $r = [l_1..u_1, l_2..u_2]$ in A, we denote by r_i the $i-$th quadrant of r, i.e. $r_1 = [l_1..m_1, l_2..m_2]$, $r_2 = [m_1+1..u_1, l_2..m_2]$, $r_3 = [l_1..m_1, m_2+1..u_2]$, and $r_4 = [m_1+1..u_1, m_2+1..u_2]$. where $m_1 = \lfloor (l_1+u_1)/2 \rfloor$ and $m_2 = \lfloor (l_2+u_2)/2 \rfloor$.

Given a a time interval $t = [t_{start}..t_{end}]$ we denote by $size(t)$ the size of the time interval t, i.e. $size(t) = t_{end} - t_{start}$. Furthermore we denote by $t_{i/4}$ the i-th quarter of t. That is $t_{i/4} = [t_{is}..t_{ie}]$ with $t_{is} = t_{start} + (i-1) \cdot size(t)/4$ and $t_{ie} = t_{start} + i \cdot size(t)/4$, $i = 1,..,4$.

Given a 4−ary tree T, we denote by $Root(T)$ the root node of T and, if p is a non leaf node, we denote the i−th child node of p by $Child(p, i)$.

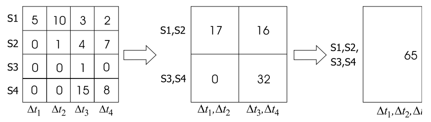
Given a triplet $x = \langle id_s, v, ts\rangle$, representing a value generated by a source, id_s is denoted by $id_s(x)$, v by $value(x)$ and ts by $ts(x)$.

3.2 The Quad-Tree Window

In order to represent data occurring in a time window, we do not store directly the corresponding two-dimensional array, indeed we choose a hierarchical data structure, called *quad-tree window*, which offers some advantages:

- it makes answering (portions of) range queries internal to the time window more efficient to perform (w.r.t. a "flat" array representation),
- it stores data in a straight compressible format. That is, data is organized according to a scheme that can be directly exploited to perform compression.

This hierarchical data organization consists of storing multiple aggregations performed over the time window array according to a quad-tree partition. This means that we store the sum of the values contained in the whole array, as well as the sum of the values contained in each quarter of the array, in each eighth of the array and so on, until the single elements of the array are stored. Fig. 3 shows an example of quad-tree partition, where each node of the quad-tree is associated to the sum of the values contained in the corresponding portion of the array.



S1
S2
S3
S4
5 10 3 2
0 1 4 7
0 0 1 0
0 0 15 8
Δt_1 Δt_2 Δt_3 Δt_4
S1,S2
S3,S4
17 16
0 32
$\Delta t_1,\Delta t_2$ $\Delta t_3,\Delta t_4$
S1,S2,
S3,S4
65
$\Delta t_1,\Delta t_2,\Delta$

Fig. 3. A Time Window and the corresponding quad-tree partition

The quad-tree structure is very effective for answering (sum) range queries inside a time window efficiently, as we can generally use the pre-aggregated sum values in the quad-tree nodes for evaluating the answer (see Section 6.1 for more details). Moreover, the space needed for storing the quad-tree representation of a time window is about the same as the space needed for a flat representation, as we will explain later.

Furthermore, the quad-tree structure is particularly prone to progressive compressions. In fact, the information represented in each node is summarized in its ancestor nodes. For instance, the node Q of the quad-tree in Fig. 3 contains the sum of its children $Q.1$, $Q.2$, $Q.3$, $Q.4$; analogously, $Q.1$ is associated to the sum of $Q1.1$, $Q1.2$, $Q1.3$, $Q1.4$, and so on. Therefore, if we prune some nodes from the quad-tree, we do not loose every information about the corresponding portions of the time window array, but we represent them with less accuracy. For instance, if we removed the nodes $Q1.1, Q1.2, Q1.3, Q1.4$, then the detailed values of the readings produced by the sensors S_1 and S_2 during the time intervals Δt_1 and Δt_2 would be lost, but it would be kept summarized in the node $Q.1$. The compression paradigm that we use for quad-tree windows will be better explained in Section 5.

We will next describe the quad-tree based data representation of a time window formally. Denoting by u the time granularity (i.e. the width of each interval Δt_j), let $T = n \cdot u$ be the time window width (where n is the number of sources). We refer to a *Time Window* starting at time t as a two-dimensional array W of size $n \times n$ such that $W[i, j]$ represents the sum of the values generated by a source s_i within the $j-$th unitary time interval of W. That is $W[i, j] = \sum_{x:id_s(x)=i \wedge ts(x) \in \Delta t_j} value(x)$, where Δt_j is the time interval $[t+(j-1) \cdot u..t+j \cdot u]$.

The whole data stream consists of an infinite sequence $W_1, W_2, \ldots$ of time windows such that the $i-$th one starts at $t_i = (i-1) \cdot T$ and ends at $t_{i+1} = i \cdot T$.

In the following, for the sake of presentation, we assume that the number of sources is a power of 2 (i.e. $n = 2^k$, where $k > 1$).

A *Quad-Tree Window* on the time window W, called $QTW(W)$, is a full $4-$ary tree whose nodes are pairs $\langle r, sum(r) \rangle$ (where r is a block of W) such that:

1. $Root(QTW(W)) = \langle [1..n, 1..n], sum([1..n, 1..n]) \rangle$;
2. each non leaf node $q = \langle r, sum(r) \rangle$ of $QTW(W)$ has four children representing the four quadrants of r; That is, $Child(q, i) = \langle r_i, sum(r_i) \rangle$ for $i = 1, \ldots, 4$.
3. the depth of $QTW(W)$ is $log_2 n + 1$.

Property 3 implies that each leaf node of $QTW(W)$ corresponds to a single element of the time window array W.

Given a node $q = \langle r, sum(r) \rangle$ of $QTW(W)$, r is referred to as *q.range* and $sum(r)$ as *q.sum*.

3.3 Compact Physical Representation of Quad-Tree Windows

The space needed for storing all the nodes of a quad-tree window $QTW(W)$ is larger than the one needed for a flat representation of W. In fact, it can be easily shown that the number of nodes of $QTW(W)$ is $\frac{4 \cdot n^2 - 1}{3}$, whereas the number of elements in W is n^2. Indeed, $QTW(W)$ can be represented compactly, as it is not necessary to store the sum values of all the nodes of the quad-tree. That is, if we have the sum values associated to a node and to three of its children, we can easily compute the sum value of its fourth child. This value can be obtained by subtracting the sum of the three children from the sum of the parent node. We say that the fourth child is a *derivable* node.

For instance, the node $Q4$ of the quad-tree window in Fig. 3 is derivable, as its sum is given by $Q.sum - (Q1.sum + Q2.sum + Q3.sum)$. Derivable nodes of the quad-tree window in Fig. 3 are all colored in white.

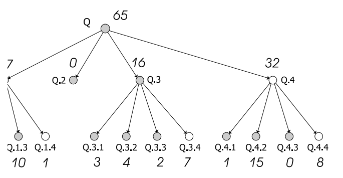
Using this storing strategy, the number of nodes that are not derivable (i.e. nodes whose sum must be necessarily stored) is n^2, that is the same as the size of W.

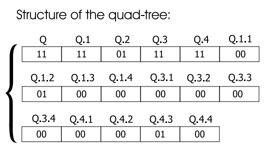
This compact representation of $QTW(W)$ can be further refined to manage occurrences of null values efficiently. If a node of the quad-tree is null, all of its descendants will be null. Therefore, we can avoid to store the sum associated to every descendant of a null node, as its value is implied. For instance, the sums of the nodes $Q2.1$, $Q2.2$, $Q2.3$, $Q2.4$ need not be stored: their value (i.e. the value 0) can be retrieved by accessing their parent.

We point out that the physically represented quad-tree describing a time window is generally not full. Indeed, null nodes having a non null parent are treated as leaves, as none of their children is physically stored. We will next focus our attention on the physical compact representation of a quad-tree window.

A quad-tree window can be stored representing separately the tree structure and the content of the nodes. The tree structure can be represented by a string of bits: two bits per node of the tree indicate whether the node is a leaf or not, and whether it is associated with a null block or not. Obviously, in this physical representation, an internal node cannot be null.

In more detail, the encoding pairs are: (1) $\langle 0, 0 \rangle$ meaning non null leaf node, (2) $\langle 0, 1 \rangle$ meaning null leaf node, (3) $\langle 1, 1 \rangle$ meaning non leaf node. It remains one available configuration (i.e., $\langle 1, 0 \rangle$) which will be used when compressing quad-tree windows, as it will be shown in Section 5. The mapping between the stored pair of bits and the corresponding nodes of the quad-tree is obtained storing the string of bits according to a predetermined linear ordering of the quad-tree nodes. In Fig. 4, the physically represented QTW corresponding to the QTW of Fig. 3 is shown. The children of $Q2$ are not explicitly stored, as they are inferable. The string of bits describing the structure of the QTW corresponds to a breadth-first visit of the quad-tree.




Structure of the quad-tree:

Q	Q.1	Q.2	Q.3	Q.4	Q.1.1
11	11	01	11	11	00
Q.1.2	Q.1.3	Q.1.4	Q.3.1	Q.3.2	Q.3.3
01	00	00	00	00	00
Q.3.4	Q.4.1	Q.4.2	Q.4.3	Q.4.4	
00	00	00	01	00	

Fig. 4. A quad-tree window and its physical representation

Note that, since the blocks in the quad-tree nodes are obtained by consecutive splits into four equally sized quadrants, the above string of bits stores enough information to reconstruct the boundaries of each of these blocks. This means that the boundaries of the blocks corresponding to the nodes do not need to be represented explicitly, as they can be retrieved by visiting the quad-tree structure. It follows that the content of the quad-tree can be represented by an array containing just the sums occurring in the nodes. Some storage space can be further saved observing that:

- we can avoid to store the sums of the null blocks, since the structure bits give enough information to identify them;
- we can avoid to store the sums contained in the *derivable* nodes of the quad-tree window, i.e. the nodes p such that $p = Child(q, 4)$, for some other node q. As explained above, the sum of p can be derived as $p.sum = q.sum - \sum_{i=1..3} Child(q, i).sum$.

Altogether, the quad-tree window content can be represented by an array storing the set $\{p.sum | p$ is a non-derivable quad-tree node and $p.sum > 0\}$.

The above sums are stored according to the same ordering criterion used for storing the structure, in order to associate the sum values to the nodes consistently. For instance, the string of sums reported on the right-hand side of Fig. 4 corresponds to the breadth-first visit which has been performed to generate the string of bits on the centre of the same figure. The sums of the nodes $Q.2$, $Q1.2$ and $Q4.3$ are not represented in the string of sums as they are null, whereas the sums of the nodes $Q.4$, $Q1.4$, $Q3.4$ and $Q4.4$ are not stored, as these nodes are derivable.

It can be shown that, if we use 32 bits for representing a sum, the largest storage space needed for a quad-tree window is $S^{max}_{QTW} = (32 + 8/3)n^2 - 2/3$ bits (assuming that the window does not contain any null value).

3.4 Populating Quad-Tree Windows

In this section we describe how a quad-tree window is populated as new data arrive. Let W_k be the time window associated to a given time interval $[(k-1) \cdot T..k \cdot T]$, and $QTW(W_k)$ the corresponding quad-tree window. Let $x = \langle id_s, v, ts \rangle$ be a new sensor reading such that ts is in $[(k-1) \cdot T..k \cdot T]$. We next describe how $QTW(W_k)$ is updated on the fly, to represent the change of the content of W_k.

Let $QTW(W_k)_{old}$ be the quad-tree windows representing the content of W_k before the arrival of x. If x is the first received reading whose timestamp belongs to the time interval of W_k, $QTW(W_k)_{old}$ consists of a unique null node (the root). The following algorithm takes as arguments x and $QTW(W_k)_{old}$, and returns the up-to-date quad-tree window Q_{new} on W_k.

Algorithm 1

```
***Function*** InsertIntoWindow
*INPUT: a sensor reading* $x = \langle id_s, v, ts \rangle$;
            *a quad-tree window* $QTW(W_k)_{old}$, *where* $ts \in [(k-1) \cdot T..k \cdot T]$.
*OUTPUT: a quad-tree window* $Q_{new}$
***begin***
```

```
    $Q_{new} := QTW(W_k)_{old}$;
    $j := \lceil (ts - (k-1)T)/u \rceil$; // $ts$ is contained into the $j$-th time interval inside $W_k$
    $Point := \langle id_s, j \rangle$;
    $CurrNode := Root(Q_{new})$;
    $UpdateSum(CurrNode, CurrNode.sum + v)$;
    **while**($size(CurrNode.range) > 1$)

        **if** ($CurrNode$ is a leaf) $Q_{new} := Split(Q_{new}, CurrNode)$;
        Let $i \in 1..4$ be such that $Point$ is inside $Child(CurrNode, i).range$;
        $CurrNode := Child(CurrNode, i)$;
        $UpdateSum(CurrNode, CurrNode.sum + v)$;

    **end while**;

**return** $Q_{new}$;
**end.**
```

wherein: 1) the function $Split(QTW, m)$ adds four children (corresponding to four null quadrants) to the leaf node m of the quad-tree window QTW, and 2) the function $UpdateSum(m, v)$ assigns the value v to the sum associated to the node m.

Algorithm 1 works as follows. First, the old quad-tree window $QTW(W_k)_{old}$ is assigned to Q_{new}. Then, the algorithm determines the coordinates $\langle id_s, j \rangle$ of the element of W_k which must be updated according to the arrival of x, and visits Q_{new} starting from its root. At each step of the visit, the algorithm processes a node of Q_{new} corresponding to a block of W_k which contains $\langle id_s, j \rangle$. The sum associated to the node is updated by adding $value(x)$ to it (see Fig. 5). If the visited node was null (before the updating), it is split into four new null children. After updating the current node (and possibly splitting it), the visit goes on processing the child of the current node which contains $\langle id_s, j \rangle$. Algorithm ends after updating the node of Q_{new} corresponding to the single element $\langle id_s, j \rangle$.

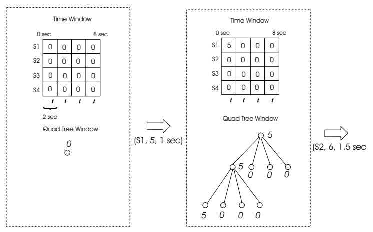


Fig. 5. Populating a quad-tree window

4 The Multi-Resolution Data Stream Summary

A quad-tree window represents the readings generated within a time interval of size T. The whole sensor data stream can be represented by a sequence of quad-tree windows $QTW(W_1), QTW(W_2), \ldots$. When a new sensor reading x arrives, it is inserted in the corresponding quad-tree window $QTW(W_k)$, where $ts(x) \in [(k-1) \cdot T..k \cdot T]$. A quad-tree window $QTW(W_k)$ is physically created when the first reading belonging to $[(k-1) \cdot T..k \cdot T]$ arrives.

In this section we define a structure that both indexes the quad-tree windows and summarizes the values carried by the stream. This structure is called *Multi-Resolution Data Stream Summary* and pursues two aims:

1. making range queries involving more than one time window efficient to evaluate;
2. making the stored data easy to compress.

We propose the following scheme for indexing quad-tree windows:

1. time windows are clustered into groups $C_1, C_2, \ldots$; each cluster consists of K contiguous time windows, thus describing a time interval of size $K \cdot T$;
2. quad-tree windows inside each cluster C_l are indexed by means of a 4-ary tree denoted by $4TI(C_l)$;
3. the whole index consists of a list linking $4TI(C_1), 4TI(C_2), \ldots$.

We next focus our attention on describing the structure of a single index $4TI(C_l)$. Then, we show how the whole index overlying the quad-tree windows is built.

4.1 Indexing a Cluster of Quad-Tree Windows

Consider the l-th cluster C_l of the sequence representing the whole sensor data stream. C_l corresponds to the time interval $[(l-1) \cdot K \cdot T..l \cdot K \cdot T]$. The time interval corresponding to C_l will be denoted by $\Delta T(C_l)$. We fix the value of K to a power of 4.

A *4-ary Tree Index* on C_l, is denoted by $4TI(C_l)$ and is a full $4-$ary tree whose nodes are pairs $\langle t, s \rangle$, with t a time interval and s a sum, such that:

1. $Root(4TI(C_l)) = \langle \Delta T(C_l), sum(\Delta T(C_l)) \rangle$
   where $sum(\Delta T(C_l))$ is the sum of the values generated within $\Delta T(C_l)$ by all the sources, i.e. $sum(\Delta T(C_l)) = \sum_{(l-1) \cdot K < i \leq l \cdot K} sum(W_i)$
2. each non leaf node $q = \langle t, s \rangle$ of $4TI(C_l)$, with $t = [j_1 T..j_2 T]$, has four child nodes corresponding to the four quarters of t, that is $Child(q, i) = \langle t_{i/4}, s_{i/4} \rangle$, $i = 1..4$, where $t_{i/4}$ is the $i-$th quarter of t, and $s_{i/4}$ is the sum of all the readings generated within $t_{i/4}$ by all the sources.
3. the depth of $4TI(C_l)$ is $log_4 K$, that is each leaf node of $4TI(C_l)$ corresponds to a time interval of size $4T$.
4. each leaf node $q = \langle t, s \rangle$ of $4TI(C_l)$, with $t = [j_1 T..j_2 T]$ $(j_2 - j_1 = 4)$, refers to the four quad-tree windows in t (i.e. $QTW(W_i)$, $j_1 < i \leq j_2$).

Given a node $q = \langle t, s \rangle$ of $4TI(C_l)$, t and s are referred to as $q.interval$ and $q.sum$, respectively. Moreover $q.range$ denotes the two-dimensional range $\langle s_1..s_n, t \rangle$.

4.2 Compact Physical Representation of 4-ary Tree Indices

In Section 3.3 we have described how quad-tree windows can be stored efficiently, saving the space needed for representing both null and derivable nodes. Analogously, 4-ary tree indices can be stored in a compact fashion: the sums of both derivable and null nodes are not explicitly stored, as they can be efficiently retrieved by accessing the stored information concerning the structure of the tree and the content of the other nodes.

The resulting physical representation is the same as the one described in Section 3.3: a string of bits is used to encode the tree structure, and an array of sums is used to represent the content of the nodes. The encoding pairs occurring in the string of bits are the same as described in Section 3.3.

The largest space consumption of a 4-ary tree index (embedding its referred QTWs) can be shown to be $S_{4TI}^{max} = (32 + 8/3) \cdot K \cdot n^2 + 8 \cdot K - 2/3$ bits.

4.3 Constructing 4-ary Tree Indices

In the same way as quad-tree windows, 4-ary tree indices can be constructed dynamically, as new data arrive and new quad-tree windows are created.

An algorithm for constructing a 4-ary tree index follows the same strategy as Algorithm 1, and, in particular, uses Algorithm 1 for populating the indexed quad-tree windows. The resulting algorithm is shown below. It consists of a function which takes as arguments a "new" reading x and the 4-ary tree index $4TI(C_l)$ where x is in $\Delta T(C_l)$, and updates both the index and the underlying quad-tree windows.

Algorithm 2

```
***Function*** Insert
*INPUT: a sensor reading* $x = \langle id_s, v, ts \rangle$*;*
        *a 4-ary tree index* $4TI(C_l)_{old}$*, where* $ts \in \Delta T(C_l)$*.*
*OUTPUT: a 4-ary tree index* $4TI(C_l)_{new}$
***begin***

    $Ind := 4TI(C_l)_{old}$;
    $CurrNode := Root(Ind)$;
    ***while***$(size(CurrNode.interval) > 4 \cdot T)$
        $UpdateSum(CurrNode, CurrNode.sum + v)$;
        ***if*** $(CurrNode$ *is a leaf*$)$
            $Ind := Split(Ind, CurrNode)$;
        *Let* $i \in 1..4$ *be such that* $ts(x)$ *belongs to* $Child(CurrNode, i).interval$;
        $CurrNode := Child(CurrNode, i)$;
    **end while**
    **if** $(CurrNode.sum = 0)$
        *Create four empty quad-tree windows corresponding*
        *to the 4 quarters of $CurrNode.interval$;*
    **end if**
    $UpdateSum(CurrNode, CurrNode.sum + v)$;
    *Let $QTW_j$ be the quad-tree window referred by $CurrNode$ whose time*
    *window contains $ts(x)$;*
    $InsertIntoWindow(QTW_j, x)$;

**return** *Ind;*
**end.**
```

wherein: 1) the function $Split(4TI, m)$ adds four children (corresponding to four null quarters) to the leaf node m of the 4-ary tree $4TI$, and 2) the function $UpdateSum(m, v)$ assigns the value v to the sum associated to the node m of $4TI$.

4.4 Linking 4-ary Tree Indices

The overall index on the sensor data stream is obtained by linking together $4TI(C_1)$, $4TI(C_2), \ldots$, i.e. the 4-ary tree indices corresponding to consecutive clusters. In particular, when a new sensor reading x arrives, it is inserted (according to Algorithm 2) into the 4-ary tree index $4TI(C_l)$ such that $ts(x)$ is in $\Delta T(C_l)$. If this $4TI$ does not exist (i.e. x is the first arrival in this cluster), then:

- a new 4-tree index $4TI(C_l)$ containing a unique null node (the root) is created;
- the function $Insert(4TI(C_l), x)$ is called;
- the updated $4TI$ returned by Algorithm 2 is added to the existing list of consecutive 4-ary tree indices.

The list of $4TI$s with the underlying list of quad-tree windows is referred to as *Multi-Resolution Data Stream Summary* - MRDS. As the sensor data stream is infinite, the length of the list of 4-ary tree indices is not bounded, so that a MRDS cannot be physically stored. In the following section we propose a compression technique which allows us to store the most relevant information carried by the (infinite) sensor data stream by keeping a finite list of (compressed) 4-ary tree indices.

5 Compression of the Multi-Resolution Data Stream Summary

Due to the bounded storage space which is available to store the information carried by the sensor data stream, the Multi-Resolution Data Stream Summary (which consists of a list of indexed clusters of quad-tree windows) cannot be physically represented, as the stream is potentially infinite.

As new sensor readings arrive, the available storage space decreases till no other reading can be stored. Indeed, we can assume that recent information is more relevant than older one for answering user queries, which usually investigate the recent evolution of the monitored world. Therefore, older information can be reasonably represented with less detail than recent data. This suggests us the following approach: as new readings arrive, if there is not enough storage space to represent them, the needed storage space is obtained by discarding some detailed information about "old" data.

We next describe our approach in detail. Let x be the new sensor reading to be inserted, and let $4TI(C_1)$, $4TI(C_2)$, ..., $4TI(C_k)$ be the list of 4-ary tree indices representing all the sensor readings preceding x. This means that x must be inserted into $4TI(C_k)$. The insertion of x is done by performing the following steps:

1. the storage space $Space(x)$ needed to represent x into $4TI(C_k)$ is computed by evaluating how the insertion of x modifies the structure and the content of $4TI(C_k)$. $Space(x)$ can be easily computed using the same visiting strategy as Algorithm 2;
2. if $Space(x)$ is larger than the left amount $Space_a$ of available storage space, then the storage space $Space(x) - Space_a$ is obtained by compressing (using a lossy technique) the oldest 4-ary tree indices, starting from $4TI(C_1)$ towards $4TI(C_k)$, till enough space is released.
3. x is inserted into $4TI(C_k)$.

We next describe in detail how the needed storage space is released from the list $4TI(C_1)$, $4TI(C_2)$, ..., $4TI(C_k)$. First, the oldest 4-ary tree index is compressed (using a technique that will be described later) trying to release the needed storage space. If the released amount of storage space is not enough, then the oldest 4-ary tree index is removed from the list, and the same compression step is executed on the new list $4TI(C_2)$, $4TI(C_3)$, ..., $4TI(C_k)$. The compression process ends when enough storage space has been released from the list of 4-ary tree indices. This process is implemented in Algorithm 3.

```
Algorithm 3
Function ReleaseStorageSpace
INPUT: a list L of 4-ary Tree Indices $4TI(C_1)$, $4TI(C_2)$, ..., $4TI(C_k)$;
          the amount $S_{req}$ of storage space to be released.
OUTPUT: a new list $L'$ of 4-ary Tree Indices.
begin

    $L' := L$;
    $S_{rel} := 0$; the storage space which has been actually released
    while($S_{rel} < S_{req}$)
        $S_{rel} := S_{rel} + Compress4TI(Root(Oldest4TI(L')), S_{req} - S_{rel})$;
        if ($S_{rel} < S_{req}$)
            $S_{rel} := S_{rel} + Space(Oldest4TI(L'))$;
            Remove $Oldest4TI(L')$ from $L'$;
        end if
    end while;
    return $L'$;
end.
```

wherein: 1) $Space(Y)$ returns the amount of storage space occupied by the 4-ary tree index Y; 2) $Oldest4TI(L)$ returns the first (i.e. the oldest) 4-ary tree index of the list L; 3) $Compress4TI(N, S)$ compresses the 4-ary tree index whose root is N till either S has been released, or the 4-ary tree index is no longer compressible. After compressing the $4TI$, the function $Compress4TI(N, S)$ returns the amount of storage space which has been actually released.

The compression strategy adopted by $Compress4TI$ exploits the hierarchical structure of the 4-ary tree indices: each internal node of a $4TI$ contains the sum of its child nodes, and the leaf nodes contain the sum of all the reading values contained in the referred quad-tree windows. This means that the information stored in a node of a $4TI$ is replicated with a coarser "resolution" in its ancestor nodes. Therefore, if we delete four sibling nodes from a 4-ary tree index, we do not loose every information carried by these nodes: the sum of their values is kept in their ancestor nodes. Analogously, if we delete a quad-tree window QTW_k, we do not loose every information about the values of the readings belonging to the time interval $[(k-1) \cdot T..k \cdot T]$, as their sum is kept in a leaf node of the $4TI$.

The compression strategy of $Compress4TI$ is based on the above reasoning. As it will be described later, it compresses the oldest $4TI$ by either compressing the referred QTWs (using an ad hoc technique for compressing quad-trees) or pruning some of its nodes. This means that the compression process modifies the structure of a $4TI$:

- a Compressed 4TI is not, in general, a full 4-ary tree, as it is obtained from a full tree (i.e. the original $4TI$) by deleting some of its nodes;
- not every leaf node refers to four QTWs, as a leaf node of the compressed $4TI$ can be obtained in three ways: 1) it corresponds to a leaf node of the original $4TI$; 2) it corresponds to a leaf node of the original $4TI$ whose referred QTWs have been deleted; 3) it corresponds to an internal node of the original $4TI$ whose child nodes have been deleted.

The compact physical representation of a $4TI$ described in Section 4.2 must be modified in order to represent a compressed $4TI$. In particular the pairs of bits encoding the tree structure need to be redefined in order to distinguish between the several kinds of leaf nodes of a compressed $4TI$. The pairs of bits which encode the tree structure of a compressed $4TI$ are: (1) $\langle 0, 0 \rangle$ meaning non null leaf node with no quad-tree windows, (2) $\langle 0, 1 \rangle$ meaning null leaf node, (3) $\langle 1, 0 \rangle$ meaning non null leaf node with quad-tree windows, (4) $\langle 1, 1 \rangle$ meaning non leaf node.

We next describe in detail the compression process of a $4TI$ performed by the function $Compress4TI$ invoked in Algorithm3. The $4TI$ to be compressed is visited in order to reach the left-most node N (i.e. the oldest node) having one of the following properties:

1. N is a leaf node of the $4TI$ which refers to 4 QTWs;
2. the node N has 4 child leaf nodes, and all the 4 children do not refer to any QTW.

In the first of the two cases, $Compress4TI$ calls an ad hoc procedure for compressing the quad-tree windows $QTW_1, \dots, QTW_4$ referred by N. The 4 QTWs are compressed till either the needed storage space is released, or they cannot be further compressed. If all the 4 QTWs are no longer compressible, then they are deleted definitively.

In the second of the two cases, the children of N are deleted. The information contained in these nodes is kept summarized in N.

In Fig. 6, several steps of the compression process on a 4-ary tree index of depth 2 (i.e. a $4TI$ indexing 16 QTWs) is shown. The QTWs underlying the $4TI$

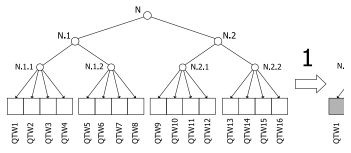


Fig. 6. Compressing a $4TI$

are represented by squares. In particular, uncompressed QTWs are white, partially compressed are grey, whereas QTWs which cannot be further compressed are crossed. We next describe the compression process reported in Fig. 6. At step 1, the oldest QTWs is partially compressed. At step 2, the needed storage space is released by continuing the compression of QTW_1 till it cannot be further compressed. As the released storage space is not enough, QTW_2 is partially compressed. After step 3, all the QTWs referred $Q.4$ are maximally compressed, and they are removed during step 4. The compression process ends after step 10: the $4TI$ consists of a unique node (the root) which will be definitively removed as further storage space is requested.

$Compress4TI(N, S)$ can be implemented using a recursive scheme, which works as follows. If N is a leaf of the 4-ary tree index referring to 4 QTWs, then it tries to release the amount S of storage space by compressing the quad-tree windows referred by N (the technique adopted to compress a quad-tree window will be explained later). In particular, the oldest of the referred quad-tree windows is compressed first. If the released storage space does not suffice, then the second QTW is compressed, and so on. If the released amount of storage space is not enough after all of the four quad-tree windows referred by N have been compressed, then they are deleted definitively.

If N is an internal node of the 4-ary tree index, then the function $Compress4TI$ is recursively called on its child nodes, starting from the left-most one (i.e. the "oldest" one) till either the needed storage space is released, or all the four child nodes have been maximally compressed (i.e. they have become leaves). In this case, the child nodes are deleted.

The implementation of this recursive scheme is given in Algorithm 4.

```
Algorithm 4
Function Compress4TI
INPUT: A node N of a 4-ary tree index;
        the amount $S_{req}$ of storage space to be released.
OUTPUT: the amount of storage space actually released.
begin

  $S_{rel} := 0$; // the storage space which has been actually released
  if (N is not a leaf)
    i:=1;
    while ($S_{rel} < S_{req}$ and $i \leq 4$)
      $S_{rel} := S_{rel} + Compress4TI(Child(N, i), S_{req} - S_{rel})$;
      $i := i + 1$;
    end while
    if ($S_{rel} < S_{req}$)
      for each $i = 1, .., 4$
        $S_{rel} := S_{rel} + Space(Child(N, i))$;
        delete the node $Child(N, i)$;
  end if
  else
  if N refers to 4 quad-tree windows $QTW_1, \ldots, QTW_4$
    i:=1;
    while ($S_{rel} < S_{req}$ and $i \leq 4$)
      $S_{rel} := S_{rel} + CompressQTW(QTW_i, S_{req} - S_{rel})$;
      $i := i + 1$;
    end while
    if ($S_{rel} < S_{req}$)
      for each $i = 1, .., 4$
        $S_{rel} := S_{rel} + Space(QTW_i)$;
        delete $QTW_i$;
  end if
  return $S_{rel}$;
```

end.

wherein: 1) $CompressQTW(QTW_i, S)$ compresses the quad-tree window QTW_i till either the amount S of storage space has been released, or QTW_i is no longer compressible. The function returns the amount of storage space actually released; 2) $Space(N)$ returns the space consumed by a 4-ary-tree node N in the $4TI$ representation described in Section 4.2; 3) $Space(QTW_i)$ returns the space occupied by the quad-tree window QTW_i.

The compression of a $4TI$ consists of removing its nodes progressively, so that the detailed information carried by the removed nodes is kept summarized in their ancestors. This summarized data will be exploited (as described in Section 6) to estimate the original information represented in the removed QTWs underlying the $4TI$. The depth of a $4TI$ (or, equivalently, the number of QTWs in the corresponding cluster) determines the maximum degree of aggregation which is reached in the MRDS. This parameter depends on the application context. That is, the particular dynamics of the monitored world determines the average size of the time intervals which need to be investigated in order to retrieve useful information. Data summarizing time intervals which are too large w.r.t. this average size are ineffective to exploit in order to estimate relevant information. For instance, the root of a $4TI$ whose depth is 100 contains the sum of the readings produced within 4^{99} consecutive time windows. Therefore, the value associated to the root cannot be profitably used to estimate the sum of the readings in a single time window effectively (unless additional information about the particular data distribution carried by the stream is available). This issue will be clearer as the estimation process on a compressed Multi-Resolution Data Stream Summary will be explained (see Section 6).

5.1 Compressing Quad-Tree Windows

The strategy used for compressing 4-ary tree indices could be adapted for compressing quad-tree windows, as quad-trees can be also viewed as 4-ary trees. For instance, we could compress a quad-tree window incrementally (i.e. as new data arrive) by searching for the left-most node N having 4 child leaf nodes, and then deleting these children.

Indeed, we refine this compression strategy in order to delay the loss of detailed information inside a QTW. Instead of simply deleting a group of nodes, we try to release the needed storage space by replacing their representation with a less accurate one, obtained by using a lower numeric resolution for storing the values of the sums.

To this end, we use a compact structure (called *n Level Tree index - nLT*) for representing approximately a portion of the QTW. *nLT* indices have been first proposed in [4, 6], where they are shown to be very effective for the compression of two-dimensional data. A *nLT* index occupies 64 bits and describes approximately both the structure and the content of a sub-tree with depth at most n of the QTW.

An example of *nLT* index (called "*3 Level Tree index*" - 3LT) is shown in Fig. 7. The left-most sub-tree $SQTW$ of the quad-tree of this figure consists of 21 nodes, which occupy $2 \cdot 21 + 32 \cdot 16 = 554$ bits ($2 \cdot 21$ bits are used to represent their structure, whereas $32 \cdot 16$ bits to represent the sums of all non derivable nodes). The 64 bits of the *nLT* index used for $SQTW$ are organized as follows: the first 17 bits are used to represent the second level of $SQTW$, the second 44 bits for the third level, and the remainder 3 bits for some structural information about the index. That is, the four nodes in the second level of $SQTW$ occupy $3 \cdot 32 + 4 \cdot 2 = 104$ bits in the exact representation, whereas they consume only 17 bits in the index. Analogously, the 16 nodes of the third level of $SQTW$ occupy $4 \cdot (3 \cdot 32 + 4 \cdot 2) = 416$ bits, and only 44 bits in the index. In Fig. 7 the first 17 bits of the 3LT index are described in more detail.

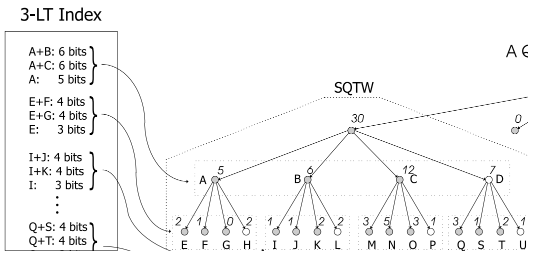


Fig. 7. A 3LT index associated to a portion of a quad-tree window

Two strings of 6 bits are used for storing $A.sum + B.sum$ and $A.sum + C.sum$, respectively, and further 5 bits are used to store $A.sum$. These string of bits do not represent the exact value of the corresponding sums, but they represent the sums as fractions of the sum of the parent node. For instance, if $R.sum$ is 100 and $A.sum = 25$, $B.sum = 30$, the 6 bit string representing $A.sum + B.sum$ stores the value: $L_{A+B} = round\left(\frac{A.sum+B.sum}{R.sum} \cdot (2^6 - 1)\right) = 35$, whereas the 5 bit string representing $A.sum$ stores the value: $L_A = round\left(\frac{A.sum}{A.sum+B.sum} \cdot (2^5 - 1)\right) = 14$. An estimate of the sums of A, B, C, D can be evaluated from the stored string of bits. For instance, an estimate of $A.sum + B.sum$ is given by: $\overline{A.sum + B.sum} = \frac{L_{A+B}}{2^6-1} \cdot R.sum = 55.6$, whereas an estimate of $B.sum$ is computed by subtracting the estimate of $A.sum$ (obtained by using L_A) from the latter value.

The 44 bits representing the third level of $SQTW$ are organized in a similar way. For instance, two strings of 4 bits are used to represent $E.sum + F.sum$ and $E.sum + G.sum$, respectively, and a string of 3 bits is used for $E.sum$. The other nodes at the third level are represented analogously.

We point out that saving one bit for storing the sum of A w.r.t. $A + B$ can be justified by considering that, on average, the value of the sum of the elements inside A is an half of the sum corresponding to $A + B$, since the size of A is an half of the size of $A + B$. Thus, on the average, the accuracy of representing $A + B$ using 6 bits is the same as the accuracy of representing A using 5 bits.

The family of nLT indices includes several types of index other than the 3LT one. Each of these indices reflects a different quad-tree structure: 3LT describes a balanced quad-tree with 3 levels, 4LT (*4 Level Tree*) an unbalanced quad-tree with at most 4 levels, and so on.

However, the exact description of nLT indices is beyond the aim of this paper. The detailed description of these indices can be found in [6].

The same portion of a quad-tree window could be represented approximately by any of the proposed nLT indices. In [6] a metric for choosing the most "suitable" nLT index to approximate a portion of a quad-tree is provided: that is, the index which permits us to re-construct the original data distribution most accurately. As it will be clear next, this metric is adopted in our compression technique: the oldest "portions" of the quad-tree window are not deleted, but they are replaced with the most suitable nLT index.

The algorithm which uses indices to compress a QTW is analogous to Algorithm 4. That is the QTW to be compressed is visited in order to reach the left-most node N (i.e. the oldest node) having one of the following properties:

1. N is an internal node of the QTW such that $size(N.range) = 16$;
2. the node N has 4 child leaf nodes, and each child is either null or equipped with an index.

Once the node with one of these properties is found, it is equipped with the most suitable nLT index, and all its descending nodes are deleted. In particular, in case 1 (i.e. N is at the last but two level of the uncompressed QTW) N is equipped with a $3LT$ index. In case 2 the following steps are performed:

1. all the children of N which are equipped with an index are "expanded": that is, the quad-trees represented by the indices are approximately reconstructed;
2. the most suitable nLT index I for the quad-tree rooted in N is chosen, using the above cited metric [6];
3. N is equipped with I and all the nodes descending from N are deleted.

The compressed QTW obtained as described above is not, in general, a full 4-ary tree, as nodes can be deleted during the compression process. Furthermore leaf nodes can be possibly equipped with an nLT index. Thus, the compact physical representation of a QTW, presented in Section 3.3, has to be modified in order to represent a compressed QTW. In particular:

- the pairs of bits which encode the tree structure are redefined as follows: (1) $\langle 0,0 \rangle$ means non null leaf node equipped with nLT index, (2) $\langle 0,1 \rangle$ means null leaf node, (3) $\langle 1,0 \rangle$ mean non null leaf node not equipped with nLT index, (4) $\langle 1,1 \rangle$ mean non leaf node.
- the array of sums representing the content of the tree is augmented with the nLT indices associated to the leaves of the compressed QTW.

5.2 The compression technique in short

The aim of our compression technique is to store as much information as possible in a given bounded storage space, satisfying the following constraints:

- compressed data must be efficient to update;
- answering to range queries on compressed data must be efficient to perform.

The compression strategy is based on the idea of releasing some storage space from "old" information stored in the Multi-Resolution Data Stream Summary as new data arrive. To this end the whole sensor data stream is divided into equally sized time windows. Each time window is represented by means of a quad-tree, called *quad-tree window* - QTW, whose hierarchical structure is particularly suitable to be compressed. A QTW is obtained by choosing a time unit and then partitioning the time window into unitary time intervals $\Delta t_1, \ldots, \Delta t_n$. Each leaf node of the QTW contains the sum of the readings produced by a source s_i in a unitary time interval Δt_j. Internal nodes of a QTW have four children and contain the sum of their values. The root of a QTW contains the sum of all the readings produced by all the sources $s_1, \ldots, s_n$ in the corresponding time window.

Time windows are then grouped into clusters. An index is associated to each cluster in order to access time windows efficiently. The index, called $4TI$, consists of a 4-ary tree having the following structure: i) the root contains the sum of all the readings represented in the cluster; ii) each leaf node refers to four consecutive quad-tree windows and contains the sum of their readings; iii) each internal node has four children and contains the sum of their values. The overall sensor data stream is represented by a list of consecutive $4TI$s.

Therefore, the data stream is stored into a hierarchical structure where information is represented using a "multi-resolution" scheme: the highest detail level is obtained in the leaf nodes of the QTWs (where the readings of the single sources are stored), and the resolution of the stored information decreases as we get closer to the roots of the $4TI$s of the list (where summarized information on the readings produced by all the sources in a wide time interval is represented).

As new data arrive, if the available storage space is not enough, the "oldest" $4TI$ of the list is compressed to release the space needed for representing the new arrivals. The oldest $4TI$ of the list is compressed incrementally till the needed space is released. The compression process sacrifices the oldest detailed information stored in the $4TI$: first, the oldest QTWs are compressed using an ad hoc technique for quad-trees. The compression of a QTW is done by replacing progressively the exact representation of some of its portions with an approximate one. The approximation is obtained by either using a lower numeric resolution (i.e. less than 32 bits) for representing the value of the sums, or deleting some of its nodes definitively. Therefore, the resolution of the information stored in a QTW decreases as the compression process goes on, till the QTW cannot be further compressed. When all the QTWs referred by a leaf of the $4TI$ cannot be further compressed, these QTWs are deleted: their sums are kept summarized in the nodes of the overlying $4TI$. As QTWs are deleted, the compression process spreads into the overlying $4TI$. The $4TI$ is compressed using an analogous strategy: the needed storage space is released by deleting four leaf nodes at a time, collapsing them in their parent node. The sacrificed leaf nodes are the oldest nodes containing no reference to any QTW: they are ancestors of the QTWs which have been deleted during some previous compression step.

The compression of a $4TI$ goes on by pruning its nodes, till the $4TI$ consists of anything but the root. When further storage space is needed, the $4TI$ is removed from the list definitively: every information about the readings generated in the corresponding time interval will be lost.

6 Estimating Range Queries on a Multi-Resolution Data Stream Summary

A sum range query $Q = \langle s_i..s_j, [t_{start}..t_{end}]\rangle$ can be computed by summing the contributions of every QTW corresponding to a time window overlapping $[t_{start}..t_{end}]$. The QTWs underlying the list of $4TI$s are represented by means of a linked list in time ascending order. Therefore the sub-list of QTWs giving some contribution to the query result can be extracted by locating the first (i.e. the oldest) and the last (i.e. the most recent) QTW involved in the query (denoted, respectively, as QTW_{start} and QTW_{end}). This can be done efficiently by accessing the list of $4TI$s indexing the QTWs, and locating the first and the last $4TI$ involved in the query. That is, the 4-ary tree indices $4TI_{start}$ and $4TI_{end}$ which contain a reference to QTW_{start} and QTW_{end}, respectively. $4TI_{start}$ and $4TI_{end}$ can be located efficiently, by performing a binary search on the list of $4TI$s. Then, QTW_{start} and QTW_{end} are identified by visiting $4TI_{start}$ and $4TI_{end}$. The answer to the query consists of the sum of the contributions of every QTW between QTW_{start} and QTW_{end}. The evaluation of each of these contributions is explained in detail in the next section.

Indeed, as the Sensor Data Stream Summary is progressively compressed, it can happen that QTW_{start} has been removed, and the information it contained is only represented in the overlying $4TI$ with less detail. Therefore, the query can be evaluated as follows:

1. the contribution of all the removed QTWs is estimated by accessing the content of the nodes of the $4TI$s where these QTWs are summarized;
2. the contribution of the QTWs which have not been removed is evaluated after locating the oldest QTW involved in the query which is still stored. This QTW will be denoted as QTW'_{start}.

Indeed, it can happen that QTW_{end} has been removed either. This means that all the QTWs involved in the query have been removed by the compression process to release some space, as the QTWs are removed in time ascending order. In this case, the query is evaluated by estimating the contribution of each involved QTW by accessing only the nodes of the overlying $4TI$s.

For instance, consider the MRDS consisting of two $4TI$s shown in Fig. 8. The QTWs whose perimeter is dashed (i.e. $QTW_1, QTW_2, \ldots, QTW_8$) have been removed by the compression process. The query represented with a grey box is evaluated by summing the contributions of the $4TI_1$ nodes $N1.1$ and $N1.2$ with the contribution of each QTW belonging to the sequence $QTW_9, QTW_{10}, \ldots, QTW_{29}$.

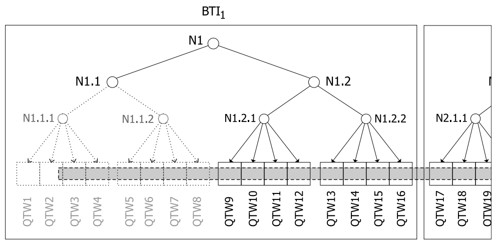


Fig. 8. A range query on a MRDS

The query estimation strategy is implemented in Algorithm 5. This algorithm uses a function
4TIBinarySearch which takes as arguments a Multi-Resolution Data Stream Summary and the time boundaries of the range query, and returns the first and the last $4TI$ of the summary involved in the query. Moreover, it uses the function *EstimateAndLocate* implemented in Algorithm 6. This function is first invoked on $4TI_{start}$ and performs two tasks: 1) it evaluates the contribution of the $4TI$ nodes involved in the query where the information of the removed QTWs is summarized, and 2) it locates (if possible) QTW'_{start}, i.e. the first QTW involved in the query which has not been removed. If QTW'_{start} is not referred by $4TI_{start}$, *EstimateAndLocate* is iteratively invoked on the subsequent $4TI$s, till either QTW'_{start} is found or all the $4TI$s involved in the query have been visited. The contribution of the $4TI$ leaf nodes to the query estimate is evaluated by performing linear interpolation. This is a simple estimation technique which is widely used on summarized data, such as *histograms*, in the context of selectivity estimation [13], and compressed datacubes, in the context of OLAP applications [12, 14]. The use of linear interpolation on a leaf node N of a $4TI$ is based on the assumption that data are uniformly distributed inside the two-dimensional range $N.range$ (CVA - *Continuous Value Assumption*). If we denote the two dimensional range corresponding to the intersection between $N.range$ and the range of the query Q as $N \cap Q$, and the size of the whole two dimensional range delimited by the node N as $size(N)$, the contribution of N to the query estimate is given by: $\frac{size(N \cap Q)}{size(N)} \cdot N.sum$.

Algorithm 5

```
Function EstimateSumQuery
INPUT: a compressed Multi-Resolution Data Stream Summary MRDS;
          a sum range query $Q = \langle s_i..s_j, [t_{start}..t_{end}]\rangle$.
OUTPUT: an estimate of the query answer.
begin
```

```
    $\langle 4TI_{start}, 4TI_{end}\rangle = 4TIBinarySearch(MRDS, [t_{start}, t_{end}])$;
    $4TI_{curr} := 4TI_{start}$;
    $Sum := 0$;
    $QTW'_{start} := null$;
    while ($QTW'_{start} = null$ and $4TI_{curr}$ precedes or coincides with $4TI_{end}$
    )
        $\langle \Delta S, QTW'_{start}\rangle := EstimateAndLocate(Q, Root(4TI_{curr}))$;
        $Sum := Sum + \Delta S$;
        Assign the $4TI$ following the current one to $4TI_{curr}$;
    end while
    if ($QTW'_{start} \neq null$ )
        $QTW_{end} := Search(MRDS, t_{end})$
        for each quad-tree window $QTW_i$ from $QTW_{start}$ to $QTW_{end}$
            $Sum := Sum + Estimate(Q, QTW_i)$;
    end if
    return $Sum$;

end
```

where :

– $Estimate(Q, QTW_i)$ evaluates the contribution of the quad-tree window QTW_i to the query Q, and will be described in more detail in Section 6.1;
– $Search(MRDS, t_{end})$ searches the most recent QTW stored in MRDS whose time window starts before t_{end}

Algorithm 6
Function EstimateAndLocate
INPUT: A query $Q = \langle s_i..s_j, [t_{start}..t_{end}]\rangle$;
A $4TI$ node N;
OUTPUT: a pair $\langle \Delta S, QTW'\rangle$, where QTW' is the first QTW which has not been removed and is referred by the sub-tree rooted in N, and ΔS is the contribution to the query result of the descending nodes of N which are older than QTW'.
begin

```
    $QTW' := null$;
    if ($N.interval$ is external to $[t_{start}..t_{end}]$)
        return $\langle 0, null\rangle$;
    if ($N.interval$ is completely contained into $[t_{start}..t_{end}]$ and $s_i..s_j =$
    $s_1..s_n$)
        return $\langle N.sum, null\rangle$;
    // otherwise, $N.interval$ overlaps $[t_{start}..t_{end}]$ partially, or $s_i..s_j$ is strictly
    contained into $s_1..s_n$
    if ($N$ is a leaf node not equipped with any QTW)
        return $\langle linear\_interp(N, Q), null\rangle$;
    if ($N$ is a leaf node referring to 4 QTWs)
```

```
        Assign the first of the referred QTWs whose time window over-
        laps $[t_{start}..t_{end}]$ to $QTW'$;
        return $\langle 0, QTW' \rangle$;
    // otherwise, $N$ is an internal node
    $i := 1$;
    $S := 0$;
    while ($i \leq 4$ and $QTW' = null$)
        $\langle \Delta S, QTW' \rangle := EstimateAndLocate(Q, Child(N, i))$;
        $S := S + \Delta S$;
        $i := i + 1$;
    end while
    return $\langle S, QTW' \rangle$;
end
```

where *linear_interp*(N, Q) evaluates the contribution of N to the estimate of Q by performing linear interpolation, as explained above.

6.1 Estimating a sum range query inside a QTW

The contribution of a QTW to a query Q is evaluated as follows. The quad-tree underlying the QTW is visited starting from its root (which corresponds to the whole time window). When a node N is being visited, three cases may occur:

1. *the range corresponding to the node is external to the range of* Q: the node gives no contribution to the estimate;
2. *the range corresponding to the node is entirely contained into the range of* Q: the contribution of the node is given by the value of its sum;
3. *the range corresponding to the node partially overlaps the range of* Q: if N is a leaf and is not equipped with any index, linear interpolation is performed for evaluating which portion of the sum associated to the node lies onto the range of the query. If N has an index, the index is "expanded" (i.e. an approximate quad-tree rooted in N is re-constructed using the information contained in the index). Then the new quad-tree is visited with the same strategy as the QTW to evaluate the contribution of its nodes (see [6] for more details). Finally, if the node N is internal, the contribution of the node is the sum of the contributions of its children, which are recursively evaluated.

The pre-aggregations stored in the nodes of quad-tree windows make the estimation inside a QTW very efficient. In fact, if a QTW node whose range is completely contained in the query range is visited during the estimation process, its sum contributes to the query result exactly, so that none of its descending nodes must be visited. This means that, generally, not all the leaf nodes involved in the query need to be accessed when evaluating the query estimate. The overall estimation process turns out to be efficient thanks to the hierarchical organization of data in the QTWs, as well as the use of the overlying $4TI$s which permits us to locate the quad-tree windows efficiently. We point out that the $4TI$s involved in the query can be located efficiently too, i.e. by performing a binary search on the ordered list of $4TI$s stored in the MRDS. The cost of this operation is logarithmic with respect to the list length, which is, in turn, proportional to the number of readings represented in the MRDS.

6.2 Answering Continuous (Range) Queries

The range query evaluation paradigm on the data summary can be easily extended to deal with *continuous range queries*. A *continuous query* is a triplet $Q = \langle s_i..s_j, \Delta T_{start}, \Delta T_{end} \rangle$ (where $\Delta T_{start} > \Delta T_{end}$) whose answer, at the current time t, is the evaluation of an aggregate operator (such as *sum*, *count*, *avg*, etc.) on the values produced by the sources $s_i, s_{i+1}, \ldots, s_j$ within the time interval $[t - \Delta T_{start}..t - \Delta T_{end}]$. In other words, a continuous query can be viewed as range query whose time interval "moves" continuously, as time goes on. The output of a continuous query is a stream of (simple) range query answers which are evaluated with a given frequency. That is, the answer to a continuous query $Q = \langle s_i..s_j, \Delta T_{start}, \Delta T_{end} \rangle$ issued at time t_0 with frequency Δt is the stream consisting of the answers of the queries $Q_0 = \langle s_i..s_j, t_0 - \Delta T_{start}, t_0 - \Delta T_{end} \rangle, Q_1 = \langle s_i..s_j, t_0 - \Delta T_{start} + \Delta t, t_0 - \Delta T_{end} + \Delta t \rangle, Q_2 = \langle s_i..s_j, t_0 - \Delta T_{start} + 2 \cdot \Delta t, t_0 - \Delta T_{end} + 2 \cdot \Delta t \rangle, \ldots$. The i-th term of this stream can be evaluated efficiently if we exploit the knowledge of the $(i-1)$-th value of the stream, provided that $\Delta t \ll \Delta T_{start} - \Delta T_{end}$. In this case the ranges of two consecutive queries Q_{i-1} and Q_i are overlapping, and Q_i can be evaluated by answering two range queries whose size is much less than the size of Q_i. These two range queries are $Q' = \langle s_i..s_j, t_0 - \Delta T_{start} + (i-1) \cdot \Delta t, t_0 - \Delta T_{start} + i \cdot \Delta t \rangle$, and $Q'' = \langle s_i..s_j, t_0 - \Delta T_{end} + (i-1) \cdot \Delta t, t_0 - \Delta T_{end} + i \cdot \Delta t \rangle$. Thus we have: $Q_i = Q_{i-1} - Q' + Q''$.

References

1. M. Datar R. Motwani J. Widom B. Babcock, S. Babu. Models and Issues in Data Stream Systems. In *Proc. PODS Symp.*, 2002.
2. D. Pregibon A. Rogers F. Smith C. Cortes, K. Fisher. Hancock: A language for extracting signatures from data streams. In *Proc. KDD Conf.*, 2000.
3. V. J. Tsotras B. Seeger D. Zhang, D. Gunopulos. Temporal aggregation over data streams using multiple granularities. In *Proc. EDBT Conf.*, 2002.
4. D. Rosaci D. Saccà F. Buccafurri, L. Pontieri. Improving range query estimation on histograms. In *Proc. ICDE*, 2002.
5. D. Saccà F. Buccafurri, F. Furfaro. Estimating Range Queries using Aggregate Data with Integrity Constraints: a Probabilistic Approach. In *Proc. ICDT*, 2001.
6. D. Saccà C. Sirangelo F. Buccafurri, F. Furfaro. A quad-tree based multiresolution approach for two-dimensional summary data. In *Proc. SSDBM Conf. (to appear)*, 2003.
7. G. Lax D. Saccà F. Buccafurri, F. Furfaro. Binary-tree histograms with tree indices. In *Proc. DEXA Conf.*, 2002.

8. F. Tian Y. Wang J. Chen, D. J. DeWitt. A scalable continuous query system for internet databases. In *Proc. ACM SIGMOD Conf.*, 2000.
9. S. Rajagopalan M. R. Henzinger, P. Raghavan. Computing on data streams. Technical Report 1998-011, Digital Systems Research Center, 1998. Available at `http://www.research.digital.com/SRC/`.
10. J. M. Hellerstein R. Avnur. Eddies: Continuously adaptive query processing. In *Proc. ACM SIGMOD Conf.*, 2000.
11. M. J. Franklin S. Madden. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. ICDE*, 2002.
12. V. Ganti V. Poosala. Fast approximate answers to aggregate queries on a datacube. In *Proc. SSDBM Conf.*, 1999.
13. Y. E. Ioannidis V. Poosala. Selectivity estimation without the attribute value independence assumption. In *Proc. VLDB Conf., Athens, Greece*, 1997.
14. Y. E. Ioannidis V. Poosala, V. Ganti. Approximate query answering using histograms. In *IEEE Data Engineering Bulletin, Vol. 22*, 1999.