Available online at www.sciencedirect.com
ScienceDirect
Data & Knowledge Engineering 63 (2007) 478–502
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Adaptive similarity search in streaming time series with sliding windows

Maria Kontaki, Apostolos N. Papadopoulos *, Yannis Manolopoulos

Data Engineering Research Laboratory, Department of Informatics, Aristotle University, 54124 Thessaloniki, Greece

Received 28 July 2006; received in revised form 13 November 2006; accepted 15 March 2007
Available online 28 March 2007

Abstract

The challenge in a database of evolving time series is to provide efficient algorithms and access methods for query processing, taking into consideration the fact that the database changes continuously as new data become available. Traditional access methods that continuously update the data are considered inappropriate, due to significant update costs. In this paper, we use the IDC-Index (Incremental DFT Computation – Index), an efficient technique for similarity query processing in streaming time series. The index is based on a multidimensional access method enhanced with a deferred update policy and an incremental computation of the Discrete Fourier Transform (DFT), which is used as a feature extraction method. We focus both on range and nearest-neighbor queries, since both types are frequently used in modern applications. An important characteristic of the proposed approach is its ability to adapt to the update frequency of the data streams. By using a simple heuristic approach, we manage to keep the update frequency at a specified level to guarantee efficiency. In order to investigate the efficiency of the proposed method, experiments have been performed for range queries and k-nearest-neighbor queries on real-life data sets. The proposed method manages to reduce the number of false alarms examined, achieving high answers vs. candidates ratio. Moreover, the results have shown that the new techniques exhibit consistently better performance in comparison to previously proposed approaches.

Keywords: Streaming time series, Similarity search; Sliding window; Performance evaluation

1. Introduction

Nowadays, a significant number of applications require the manipulation of data streams [6,2,7,17,8,10]. Examples of these applications are online stock analysis, computer network monitoring, network traffic management, earthquake prediction. The major common characteristic of the above applications is that they are all time-critical. Therefore, the Database Management System (DBMS) must be equipped by effective and efficient tools for data stream processing, towards acceptable performance during insert, update and query

* Corresponding author.
E-mail addresses: kontaki@delab.csd.auth.gr (M. Kontaki), apostol@delab.csd.auth.gr (A.N. Papadopoulos), manolopo@delab.csd.auth.gr (Y. Manolopoulos).

0169-023X/$ - see front matter
doi:10.1016/j.datak.2007.03.002

operations. Due to the highly dynamic nature of data streams, random access is prohibitive. Therefore, each data stream is possible to be read only once (or a limited number of times). This feature poses additional difficulties for query processing, since the data can only be accessed in arrival order, and therefore data streams are not compatible with traditional query processing approaches used in Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP) systems, and therefore additional tools are required.

An important query type that has been studied thoroughly in database literature is the similarity query. Given a query object Q the similarity query asks for all objects O_x that are similar to Q to a sufficient degree. Similarity queries have been studied for multidimensional objects, images, video, time series and other non-traditional data types. In data streams the problem is more challenging since the query object, the data or both may change over time. The similarity between two objects is expressed by means of a distance metric *dist* (e.g., Euclidean, Manhattan). Basically, there are three similarity query types that have been extensively used in the literature:

- *similarity range query*: given a query object q, a set of objects $\mathscr{A}$ and a distance e retrieve all objects $a \in \mathscr{A}$ such that $dist(q,a) \leqslant e$.
- *similarity k-nearest-neighbor query*: given a query object q, a set of objects $\mathscr{A}$ and an integer k retrieve k database objects $a_i \in \mathscr{A}(1 \leqslant i \leqslant |A|)$ such that for any other object $a_j \in \mathscr{A}(1 \leqslant j \leqslant |A|$ and $j \neq i)$ $dist(q,a_i) \leqslant dist(q,a_j)$.
- *similarity join query*: given two sets of objects $\mathscr{A}$ and $\mathscr{B}$ and a distance e retrieve pairs (a,b) with $a \in \mathscr{A}$ and $b \in \mathscr{B}$ such that $dist(a,b) \leqslant e$.

In this paper, we study similarity e-range and k-nearest-neighbor queries in streaming time series, where both the query sequence and the data sequences change over time. The length of a streaming time series can be very large, since new values are continuously appended. Therefore, the similarity of two time series is expressed by means of the last values of each stream (e.g., 128, 256, 1024 values), using a sliding window approach. Each stream can be represented as a vector in a high-dimensional space. Dimensionality reduction techniques (e.g., Discrete Fourier Transform, Karhunen–Loeve Transform) can be used in order to reduce the number of dimensions, allowing efficient multidimensional access methods to be utilized. However, each vector changes over time since new values are continuously appended. The naive approach is to delete the old feature vector by updating the access method, to re-apply the dimensionality reduction technique to the new vector, and finally, to store the resulting feature vector in the access method. This process is very costly both in Central Processing Unit (CPU) time and number of disk accesses and therefore, it is inappropriate in our case. We develop a novel method in order to process similarity queries in data streams with sliding windows. The basic characteristics of the proposed approach are summarized as follows:

- Streaming queries over streaming data are supported.
- R-tree-based [21,5] access methods are utilized, which are well-studied in the literature and they have already been incorporated in commercial database management systems.
- The feature extraction method works in an incremental manner, enabling considerable savings in CPU time during the feature extraction process.
- Index updates are either avoided or performed in a bottom-up manner, which results in improved performance. Moreover, the update ratio is controllable and can be altered according to the current system workload.
- Storage requirements are significantly less in comparison to existing techniques, and therefore in-memory maintenance of access methods is feasible.
- The total running time is significantly less than existing approaches, which is a favorable property in data stream applications.

The rest of the work is organized as follows. In the next section, we give the appropriate background, the related work and the motivation behind the proposed research. Section 3 studies in detail the proposed method for similarity query processing. Section 4 presents performance evaluation results based on real-life data sets. Finally, Section 5 concludes the work and shortly discusses future research in the area.

2. Background, related work and contribution

A streaming time-series S is a sequence of real values $s_1, s_2, \ldots$, where new values are continuously appended as time progresses. Formally a streaming time series S consists of a set of (tuple, timestamp) pairs. The ordering that tuples become available induced by the timestamps. For example, a temperature sensor which monitors the environmental temperature every five minutes, produces a streaming time-series of temperature values. As another example, consider a car equipped with a Global Positioning System (GPS) device and a communication module, which transmits its position to a server every ten minutes. A streaming time-series of two-dimensional points (the x and y coordinates of its position) is produced. Note that, in a streaming time-series data values are ordered with respect to the arrival time. New values are appended at the end of the series.

Similarity queries in streaming time series have been studied in [11] where whole-match queries are investigated by using the Euclidean distance as the similarity measure. A prediction-based approach is used for query processing. The distances between the query and each data stream are calculated using the predicted values. When the actual values of the query are available, the upper and lower bound of the prediction error are calculated and the candidate set is formed using the predicted distances. Then, false alarms are discarded. The same authors have proposed two different approaches, based on pre-fetching [12,13]. Both the aforementioned research efforts examine the case of whole-match queries, where the data set is composed of static time series and the query is dynamic (changes over time).

A class of algorithms for stream processing focuses on the recent past of data streams by applying a *sliding window* on the data stream [4,11,15,17,23]. In this way, only the last W values of each streaming time series is considered for query processing, whereas older values are considered obsolete and they are not taken into account. As it is illustrated in Fig. 1, streams that are non-similar for a window of length W (left), may be similar if the window is shifted in the time axis (right).

In [17] the authors present a method for query processing in streaming time series where both the query object and the data are dynamic. The VA-stream and VA$^+$-stream access methods have been proposed, which are variations of the vector approximation file (VA-file) [22]. These structures are able to generate a summarization of the data and enable the incremental update of the structure every time a new value arrives. This method is our competitor since both queries and data are dynamic. We give a brief description and we discuss some implementation issues of this method later in the paper.

The design of efficient index structures with respect to the number of updates has attracted the interest of the researchers. The authors in [14] have proposed a variation of the R^*-tree. A auxiliary data structure is used which provides additional access path to the R^*-tree. Moreover, the reinsertion of objects in the index structure is not applied to the root but to internal nodes. In [16] the authors have proposed a variation of the R^*-tree which is designed towards more efficient indexing in the presence of frequent updates. Two auxiliary data structures are used in addition to the R^*-tree: (i) a hash table which maps object identifiers to leaf nodes and (ii) a summary data structure which is used to obtain access to the internal tree nodes. These approaches

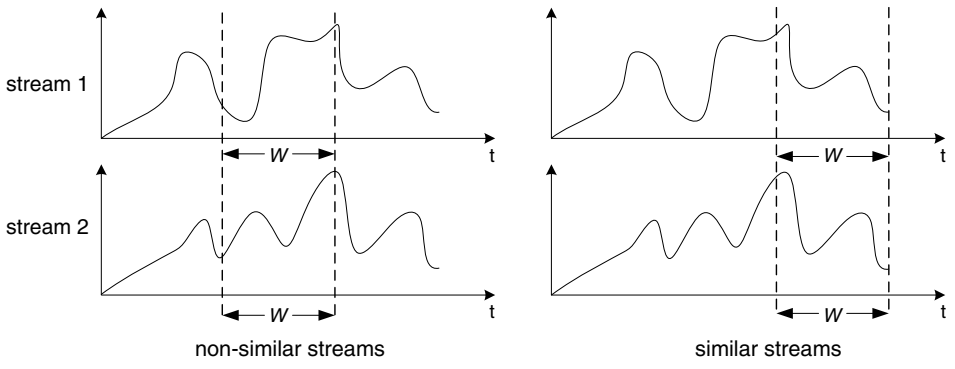


Fig. 1. Similarity using a sliding window of length W.

are not applicable in streaming data because their main focus is to accelerate the update procedure rather than to reduce the number of updates. In the streaming case, the control of the update frequency of the index is necessary to avoid system degradation.

In [15], we have proposed an indexing scheme and a query processing mechanism for similarity range query processing on streaming time series, using the sliding window approach. The proposed approach (IDC-Index) outperforms the VA$^+$-stream method during range queries, whereas the overall performance for queries and updates has been shown to be better. In this article, we extend the work studied in [15] in the following ways:

- In addition to range queries, k-NN queries are considered.
- A technique is proposed which adapts the index update rate according to application requirements.
- A bottom-up approach for index updates is adopted in order to accelerate update operations.
- Several optimizations to the original IDC-Index are proposed, towards more efficient query processing.

3. Proposed approach

A stream is denoted by the symbol S_x and a finite time series by the symbol $S_x[i:j]$, where i is the first time instance of the time series and j is the last. The number of values of a time series is therefore $j-i+1$ and corresponds to a window of length W. $S_x(i)$ is the ith value of the time series.

In our study, the Euclidean distance between two finite time series is used as the similarity measure. The distance between two streaming time series S_x and S_y is defined by the Euclidean distance between the last W values of S_x and S_y, denoted by $D_E(S_x, S_y)$. Table 1 summarizes the basic symbols and definitions used throughout the study.

Let us assume the existence of n streaming time series, each updated over time. To determine similar streaming time series, we use only the last W values of each one and update these values when a new value becomes available. Given a query streaming time series the challenge is to determine similar time series as the data evolve with time.

The naive approach is the Sequential Scan (SS). In each update, the distances between the query streaming time series and each data streaming time series are computed. Then similar streaming time series are reported. The streaming environment poses new challenges to the applications as unbounded memory requisites, high input rates and fast response times. Therefore more sophisticated approaches are necessary to speed up the similarity process.

Feature extraction and dimensionality reduction are well known and established techniques that have been proposed in order to simplify complicated problems. The similarity measure is applied on the extracted features. Moreover, an indexing method is used to prune time series and therefore to avoid distance computations.

Table 1
Basic notations used throughout the study

Symbol	Description
S, T, S_x, S_y	Streaming time series
S_q	Query streaming time series
$S[i:j]$	A finite time series between time instances i and j
$S(i)$	The ith value of the time series
$DFT(S)$, $DFT(S_x)$	The DFT of streaming time series S, S_x
$\mathrm{DFT}_i(S)$	The ith DFT coefficient of S
$D_E(S_x, S_y)$	Euclidean distance between streaming time series S_x and S_y
Δ_u	Update threshold value
Δ_q	Query expansion value
k	Number of nearest neighbors requested
d_k	The kth best distance
e	Radius of circular range query
W	Sliding window length
U	Desirable update frequency

In static environments, the overall procedure for similarity queries includes the following steps: (a) feature extraction is applied on time series, (b) the extracted features are inserted in an index structure, (c) feature extraction is applied on the query time series, (d) the index is used to retrieve candidate time series respecting a user-defined threshold and (e) the distances between query and candidates time series are computed in a post processing step to discard false alarms.

Following the above scheme, in this paper we used the DFT coefficients as features and R^*-tree based indexes. To satisfy the limitations posed by the streaming environment, (a) we introduced an incremental computation of DFT in order to avoid costly DFT computation, and (b) we introduced different deferred update policies in order to avoid the degradation of the system due to the usage of the index structure and the high number of updates.

Fig. 2 depicts the architecture of the system. The last W values of each stream are stored in the disk. The DFT coefficients of each stream are inserted into the index. At the leaf level of the index a link associates the DFT coefficients with the stream. Additionally each stream maintains a link to the leaf where the corresponding DFT coefficients are stored. When a value becomes available the window of the stream is updated, the features of the stream are extracted incrementally and the new DFT coefficients replace the old ones using the link "stream to leaf". Then the deferred update policy decides if the index will be update. If yes, a bottom-up adjustment is performed from the leaf up to the root of the tree if necessary. The query is applied to the index to retrieve candidates streaming time series using the link "leaf to stream". Then in a post processing step the actual distances are computed to discard false alarms. The system architecture can be divided into three subsections: the incremental feature extraction, the deferred update policy and the index structure. The next sections provide details on each of them.

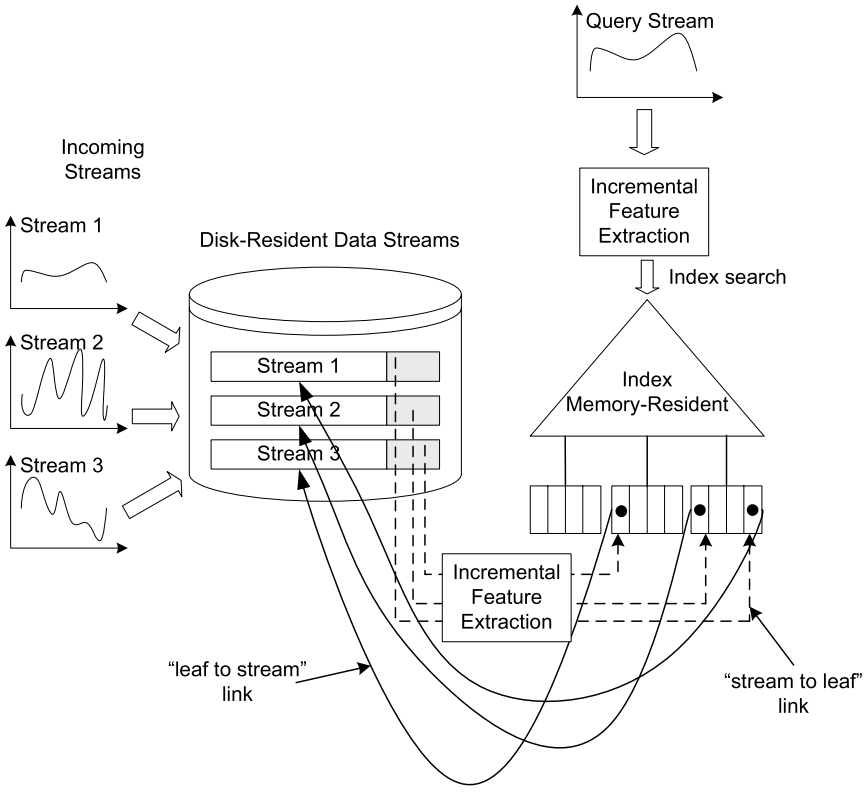


Fig. 2. System architecture.

3.1. Incremental DFT computation

The DFT is used as the feature extraction method, which preserves the Euclidean distance between two sequences. Real-life time series often concentrate the energy in the first few components of the DFT. Therefore, we need less information to capture the characteristics of the original vector. Another important feature of the DFT is that the Euclidean distance in the time domain and the Euclidean distance in the frequency domain are equal. By taking the first coefficients of the DFT vectors, the resulting distance between two vectors is reduced, and therefore, no false dismissals occur during range query processing, since the distance is lower-bounded [1,9]. There is a trade-off between the number of the DFT coefficients and the approximation of the distance in the time domain. If more DFT coefficients are used then the number of candidates is reduced during the query processing, thus the query procedure speeds up.

Normally, every time a new value for a stream arrives, the DFT vector must be recalculated by using the last W values of the stream. This may lead to high costs since the re-computation of the DFT is quite expensive. However, as the following proposition explains, the computation of the DFT can be performed incrementally, avoiding re-computation, by exploiting the last calculated DFT coefficients.

Proposition 1. *Let S be a streaming time series with values $S(0), S(1), \ldots, S(W-1)$ and length W. Moreover, let $DFT_0(S), DFT_1(S), \ldots, DFT_{W-1}(S)$ denote the DFT coefficients of S. If a new value for this stream arrives, we get the sequence $T(1), T(2), \ldots, T(W)$, where $S(i) = T(i)$ for $1 \leqslant i \leqslant W-1$ and $T(W)$ is the new value. The DFT coefficients of T can be computed by the DFT coefficients of S according to the following equation*:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot \mathrm{DFT}_n(S) - S(0) + T(W)\right) \cdot \mathrm{e}^{\mathrm{j}2\pi n/W}, \quad (0 \leqslant n \leqslant W-1) \tag{1}$$

Proof. See Appendix. □

Each DFT coefficient has a real and an imaginary part. An implementation issue must be solved is how we can compute separately each part of a DFT coefficient. The following proposition explains.

Proposition 2. *Let S be a streaming time series with values $S(0), S(1), \ldots, S(W-1)$ and length W. Moreover, let $DFT_0(S), DFT_1(S), \ldots, DFT_{W-1}(S)$ denote the DFT coefficients of S. If a new value for this stream arrives, we get the sequence $T(1), T(2), \ldots, T(W)$, where $S(i) = T(i)$ for $1 \leqslant i \leqslant W-1$ and $T(W)$ is the new value. The real $(DFT_n(T)_{\mathrm{real}})$ and the imaginary $(DFT_n(T)_{\mathrm{imag}})$ part of the DFT coefficients of T can be computed by the DFT coefficients of S according to the following equations*:

$$\mathrm{DFT}_n(T)_{\mathrm{real}} = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot \mathrm{DFT}_n(S)_{\mathrm{real}} - S(0) + T(W)\right) \cdot \cos\left(\frac{2\pi n}{W}\right) - \mathrm{DFT}_n(S)_{\mathrm{imag}} \cdot \sin\left(\frac{2\pi n}{W}\right) \tag{2}$$

and

$$\mathrm{DFT}_n(T)_{\mathrm{imag}} = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot \mathrm{DFT}_n(S)_{\mathrm{real}} - S(0) + T(W)\right) \cdot \sin\left(\frac{2\pi n}{W}\right) + \mathrm{DFT}_n(S)_{\mathrm{imag}} \cdot \cos\left(\frac{2\pi n}{W}\right) \tag{3}$$

where $(0 \leqslant n \leqslant W-1)$.

Proof. See Appendix. □

The above proposition can be used to incrementally compute the new DFT vector of a streaming time series, taking into account the previous one, and therefore, avoiding the re-computation.

Example 1. Assume the streaming time series S: 3, 2, 1, 3 with $W = 4$. The second DFT coefficient of S is $\mathrm{DFT}_1(S) = \frac{2+j}{2}$ or $\mathrm{DFT}_1(S)_{\mathrm{real}} = 1$ and $\mathrm{DFT}_1(S)_{\mathrm{imag}} = \frac{1}{2}$. Moreover, assume that a new value arrives (e.g. 4) and forms a new streaming time series T: 2, 1, 3, 4. So if for example we want to compute $\mathrm{DFT}_1(T)$, we can use $\mathrm{DFT}_1(S)$. Therefore $\mathrm{DFT}_1(T)_{\mathrm{real}} = \frac{1}{\sqrt{4}} \cdot (\sqrt{4} \cdot 1 - 3 + 4) \cdot \cos(\frac{2\pi}{4}) - \frac{1}{2} \cdot \sin(\frac{2\pi}{4}) = -\frac{1}{2}$ and $\mathrm{DFT}_1(T)_{\mathrm{imag}} = \frac{1}{\sqrt{4}} \cdot (\sqrt{4} \cdot 1 - 3 + 4) \cdot \sin\left(\frac{2\pi}{4}\right) + \frac{1}{2} \cdot \cos\left(\frac{2\pi}{4}\right) = \frac{3}{2}$.

3.2. Deferred update policy

Since the number of streams may be quite large, the use of an index structure is desirable to avoid the computation of the distance between the query and all the time series. We use the R^*-tree as an index structure for the DFT coefficients of the streaming data series. In our case the problem is that the DFT coefficients of a streaming time series must be updated when a new value arrives. If we update the index every time a new value becomes available, the overhead may be prohibitive due to additional page accesses. To avoid continuous deletions and insertions in the R^*-tree, we use a deferred update policy. A parameter Δ_u is used to control the updates. If the distance between the new and the old DFT vector exceeds the value of parameter Δ_u, then the R^*-tree is updated. Otherwise, no update is performed. This technique leads to considerable savings in CPU and I/O time. The last recorded DFT vector is stored in the last disk page of every streaming time series, to become available when a new value arrives. The price paid is that the indexing scheme may not always reflect the data distribution. The use of the parameter Δ_u raises two issues: (i) whether it leads in false dismissals and (ii) whether it affects the query processing efficiency. As we shall demonstrate later, with appropriate modifications performed to the query, no false dismissals occur and moreover, the query processing efficiency is not affected significantly.

Let S be a streaming time series. The last W values form a sequence denoted by $S_1[N-W+1:N]$, where N is the position of the last value of the time series. When a new value for S_1 is available, a new sequence $S_2[N-W+2:N+1]$ is formed. Assume further that $\mathrm{DFT}(S_1)$ is the last recorded DFT sequence corresponding to $S_1[N-W+1:N]$, and $\mathrm{DFT}(S_2)$ is the DFT sequence corresponding to $S_2[N-W+2:N+1]$, which is computed incrementally using the $\mathrm{DFT}(S_1)$. If $D_E(\mathrm{DFT}(S_1),\mathrm{DFT}(S_2)) \leqslant \Delta_u$, then $\mathrm{DFT}(S_2)$ is stored as the most recent DFT (replaces $\mathrm{DFT}(S_1)$) but it is not inserted into the R^*-tree. Assume that another value for the same stream arrives. Let $S_3[N-W+3:N+2]$ be the new time series and $\mathrm{DFT}(S_3)$ the DFT of this sequence, which is computed incrementally using $\mathrm{DFT}(S_2)$. $\mathrm{DFT}(S_3)$ replaces $\mathrm{DFT}(S_2)$ as the most recent DFT. If $D_E(\mathrm{DFT}(S_3),\mathrm{DFT}(S_1)) \leqslant \Delta_u$, no update is performed in the R^*-tree. On the other hand, if $D_E(\mathrm{DFT}(S_3),\mathrm{DFT}(S_1)) > \Delta_u$, then $\mathrm{DFT}(S_3)$ replaces $\mathrm{DFT}(S_1)$ in the tree, so an update is performed.

In summary, we need both the last recorded DFT vector and the previously calculated DFT vector. The first is used to decide whether an update will occur or not, and the second is used for the incremental computation of the new DFT vector. Fig. 3 describes the steps of the deferred update policy. Step 1 is the incremental feature extraction of Fig. 2. Step 3 uses the "stream to leaf" link to compute the distance between the DFT coefficients. Step 4 updates the internal nodes of the index.

3.3. IDC-index with global query expansion

The value of the update threshold Δ_u can either be fixed or can vary according to the application requirements. Having a fixed Δ_u requires statistical analysis of the streaming time series, to select a value that guarantees performance efficiency. However, since the characteristics of a streaming time series change over time, selecting a fixed value for Δ_u is very restrictive. In [15] a fixed value for Δ_u had been used. In the next sections, we show how we can keep Δ_u up-to-date as streams evolve with time.

When a new value arrives:
1. Compute the DFT coefficients of the new streaming time series using the coefficients of the previous streaming time series.
2. Replace the previous DFT coefficients with the new DFT coefficients.
3. Compute the distance between the new DFT coefficients and the coefficients that are stored in the index (i.e. the last recorded coefficients).
4. If the distance is more than Δ_u, update the index.

Fig. 3. Deferred update policy.

3.3.1. Selecting the update threshold Δ_u

In this section, we elaborate in this issue by adapting the update threshold according to the desirable update frequency U posed by the specific application. The update frequency denote the maximum allowed number of updates that must be performed to the index to guarantee efficiency. Taking into consideration that a huge number of updates may be required as new stream values become available, the parameter U is used to compensate the excessive update demand. This parameter can be set by the application, or may vary according to the "mobility" of the data streams. For example, when new values arrive in a very slow rate, then the system can afford a larger number of index updates. On the other hand, when new values arrive at a very high rate the index update frequency should be reduced to prevent performance degradation. In the sequel we explain in detail how to estimate the value of Δ_u dynamically, to approximate U.

The target is to maintain the value of U as accurately as possible, based on the recent past of the streaming time series. In this way, we can determine a convenient value for Δ_u for the near future. In order to achieve this, an adaptive calculation of Δ_u is applied. The last u values arrived are used in order to determine Δ_u. For example, let U be 20%, which means that every 100 update requests only 20 index updates will be performed. For the next $u = 10$ update requests we monitor the Euclidean distance between the last and the previous DFT vector for the affected streaming time series. In order to achieve 20% index updates, the value of Δ_u must be set to 3, as it is illustrated in Fig. 4. In this way, only 2 out of 10 updates affect the index and the percentage of 20% is achieved. We use this Δ_u for the next $u = 10$ values. Moreover, we monitor the next u values. Again, we select the second minimum value as Δ_u to achieve the desirable update frequency for the next period. This procedure is continuously repeated. An important issue that we note, is that the number u must be selected to have a significant number of stream values for the determination of Δ_u. To achieve this, values between 500 and 1000 have been used for the estimation of Δ_u in our experiments.

If the value of U remains constant, then no specialized data structures are required. In the previous example, if $U = 20\%$ and $u = 10$, then we need only to maintain the two smallest values. However, this is not a realistic scenario, since U can be increased when stream mobility is low, and decreased when mobility is high. To support this, a minheap data structure (priority queue) is utilized. Recall, that a minheap structure stores the minimum value at the root. This means that the minimum value is available in O(1) time. By deleting the minimum value, the heap is adjusted in O(log$n$) time (where n is the size of the heap) and the next minimum is placed at the root. Therefore, if the xth minimum value must be determined, the heap is accessed $x - 1$ times, until the xth smallest value reaches the root. In Fig. 4, we see that to guarantee 20% index updates the second smallest value must be determined. This value is 3, and reaches the root after element 2 is deleted. Therefore, we set Δ_u to 3. Using the recent past stream values to predict the near future has been proven very accurate, as it is illustrated in the performance evaluation.

Maintaining the update frequency of the IDC-Index fixed, the method avoids system degradation due to high input rates of streams. Moreover, as we have already mentioned, we shall demonstrate that the use of Δ_u does not introduce false dismissals and therefore does not affect the matching quality.

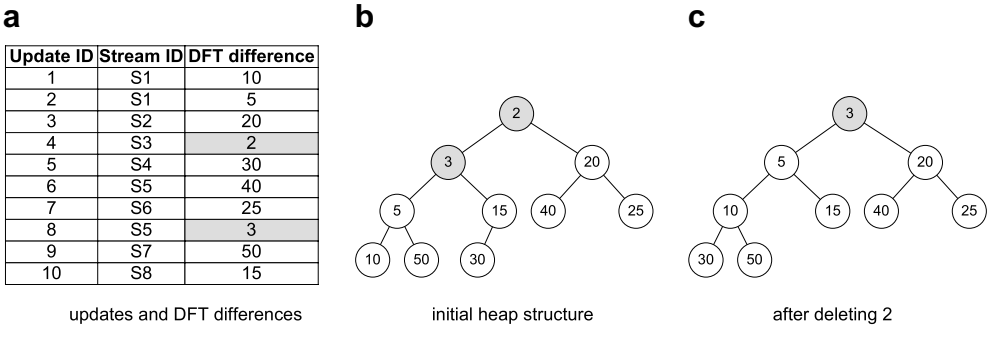


Fig. 4. Determination of Δ_u.

3.3.2. Updating the index

The traditional method to update the R^*-tree index requires to locate the relevant object in the leaf node, delete the old entry and insert the new entry using the normal insertion algorithm. However, this process is computationally expensive and may lead to performance degradation when the stream "mobility" is high.

Recall that in our case, an index update will take place if the Euclidean distance between the new DFT vector and the last recorded DFT vector for a stream is greater than Δ_u. In such a case, instead of using the traditional index update process, another method is applied which is more efficient regarding computational costs.

Each stream maintains a link to the leaf where the corresponding DFT vector is stored. Therefore, there is no need to search for the specific entry in a top-down manner. When a new value arrives, then the new DFT vector for the specific stream is stored at the leaf in addition to the previously recorded one. If the Euclidean distance between the new DFT vector and the last recorded DFT vector is less than or equal to Δ_u, then no more modifications are performed to the R^*-tree structure. Otherwise, a bottom-up adjustment of the Minimum Bounding Rectangles (MBRs) is performed, by recalculating MBRs from the leaf, up to the root of the tree if necessary. To perform the bottom-up traversal, parent links are required which enable node visits in a bottom-up manner (from a child node to its parent). By applying this technique, the use of the parameter Δ_u does not affect the query processing efficiency, since the leaves always contain up-to-date DFT vectors and therefore the candidate set is the same either Δ_u is used or not.

For example, assume that a new value arrives for the stream 1 of Fig. 2. The new DFT coefficients are computed using the incremental feature extraction, and the "stream to leaf" link is used to update the DFT coefficients of the leaf of the index. Then the deferred update policy decides if the index should be updated. In this case a bottom-up adjustment of the MBRs is performed so all the MBRs of the path from the leaf to the root to contain the new DFT coefficients. Note that the replacement of the DFT coefficients of the leaf is always performed regardless of the decision of the deferred update policy to update the index or not.

3.3.3. Query processing issues

So far we have focused on the detailed description of the feature extraction process, the way index updates are performed and how to adjust the update threshold Δ_u. We proceed by describing the way query processing is performed. Both similarity range and nearest-neighbor queries are considered, and detailed algorithms are given. Both query processing algorithms follow a filter-refinement processing paradigm, to eliminate false alarms which may be present due to the approximation performed by the use of DFT vectors. The architecture outline of the query processor is depicted in Fig. 2.

3.3.4. Range query processing

In order for similarity range queries to produce the correct results, the user-defined distance e must be expanded by a value of Δ_q, which is the maximum Δ_u value seen so far. In this way, we guarantee that no false dismissals occur.

Proposition 3. *If the query radius e is expanded by Δ_q in the internal nodes of R^*-tree, where Δ_q is the maximum Δ_u seen so far, then no false dismissals are introduced.*

Proof. Assume that we have the DFT vector of the query stream $DFT(S_q)$, the $\mathrm{MBR}_{\mathrm{LR}}$ that is formed by the last recorded DFT vectors and a stream S_x that belongs to the $\mathrm{MBR}_{\mathrm{LR}}$. Moreover, assume that we have the last recorded DFT vector $\mathrm{DFT}(S_x)_{\mathrm{LR}}$ of stream S_x and the current DFT vector $\mathrm{DFT}(S_x)$ of stream S_x.

$$
\begin{aligned}
&D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)) \leqslant e \Rightarrow \\
&D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)) + \Delta_q \leqslant e + \Delta_q \Rightarrow \\
&D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)) + D_E(\mathrm{DFT}(S_x), \mathrm{DFT}(S_x)_{\mathrm{LR}}) \leqslant e + \Delta_q \Rightarrow \\
&D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)_{\mathrm{LR}}) \leqslant e + \Delta_q \quad (\text{triangular inequality}) \Rightarrow \\
&\mathrm{MinDist}(\mathrm{DFT}(S_q), \mathrm{MBR}_{\mathrm{LR}}) \leqslant e + \Delta_q \qquad \square
\end{aligned}
$$

Algorithm RangeQuery ($S_q[N-W+1:N]$, e)
Input: $S_q[N-W+1:N]$ is the last W values of the query time series, e is the query radius.
Output: the IDs of the relevant streams are stored in $\mathcal{A}$.

1. Calculate the DFT vector $DFT(S_q)$ of $S_q[N-W+1:N]$;
2. Set r=e+Δ_q;
3. Perform a range query using the R*-tree with $DFT(S_q)$ and r in the internal nodes and e in the leaves; // filtering
4. Store candidate streams to $\mathcal{C}$;
5. **foreach** DFT vector $DFT(S_x) \in \mathcal{C}$; // refinement
6. Get the original time series S_x;
7. **if** $D_E(S_x[N-W+1:N], S_q[N-W+1:N]) \leq e$ **then** add S_x to $\mathcal{A}$;
8. **endfor**
9. Report $\mathcal{A}$;

Fig. 5. Outline of similarity range query processing algorithm.

Proposition 3 implies that if the current DFT vector of a stream overlaps the query region, then the corresponding $\mathrm{MBR_{LR}}$ (a MBR that is formed by the last recorded DFT coefficients) will overlap the extended query region. Notice that Proposition 3 is applied only in the internal nodes of the R^*-tree, since the leaves always contain up-to-date DFT vectors. The outline of the similarity range query processing algorithm is illustrated in Fig. 5.

3.3.5. Nearest-neighbor query processing

For the nearest-neighbor queries, we use the multi-step k-NN algorithm [20]. The algorithm reduce the number of candidates by decreasing the value of the kth nearest neighbor distance with ongoing exact evaluation of the candidates. To eliminate false dismissals, as in range queries, we expand the kth distance by Δ_q.

Proposition 4. *If the kth nearest neighbor distance is expanded by Δ_q when searching the internal nodes of the R^*-tree, then no false dismissals are introduced.*

Proof. Assume that we have the DFT vector of the query stream $\mathrm{DFT}(S_q)$, the $\mathrm{MBR_{LR}}$ that is formed by the last recorded DFT vectors and a stream S_x that belongs to the $\mathrm{MBR_{LR}}$. Moreover, assume that we have the last recorded DFT vector $\mathrm{DFT}(S_x)_{\mathrm{LR}}$ of stream S_x and the current DFT vector $DFT(S_x)$ of stream S_x. The kth nearest neighbor distance is d_k.

$$
\begin{aligned}
D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)) \leqslant d_k &\Rightarrow D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)) + \Delta_q \leqslant d_k + \Delta_q \\
&\Rightarrow D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)) + D_E(\mathrm{DFT}(S_x), \mathrm{DFT}(S_x)_{\mathrm{LR}}) \leqslant d_k + \Delta_q \\
&\Rightarrow D_E(\mathrm{DFT}(S_q), \mathrm{DFT}(S_x)_{\mathrm{LR}}) \leqslant d_k + \Delta_q \quad \text{(triangular inequality)} \\
&\Rightarrow \mathrm{MinDist}(\mathrm{DFT}(S_q), \mathrm{MBR_{LR}}) \leqslant d_k + \Delta_q \qquad \square
\end{aligned}
$$

Proposition 4 implies that if the current DFT vector of a stream is closer to the query point than the kth neighbor, then the corresponding $\mathrm{MBR_{LR}}$ will be inserted in the heap for further examination, if we expand the kth nearest neighbor distance by Δ_q.

The k-NN query processing algorithm can be further improved, if a more sophisticated scheme is used for the initialization of the kth NN distance d_k. For example, we can visit the first k streams and compute their real distances from the query. The maximum distance can be used for the initialization of d_k, instead of ∞. The outline of the k-NN query processing algorithm is depicted in Fig. 6.

3.4. IDC-Index with local query expansion

In the IDC-Index with global query expansion, only one value for Δ_q is maintained for all internal nodes of the R^*-tree. Moreover, Δ_q cannot be reduced, in order to preserve the correctness of the algorithm. Since the value of Δ_q is determined by taking the maximum value of Δ_u seen so far, if at some time Δ_u assumes a large

Algorithm NNQuery ($S_q[N-W+1:N]$, k)
Input: $S_q[N-W+1:N]$ is the last W values of the query time series, k is the number of NNs.
Output: $Heap_k$ is the heap of the best k answers.

```
1.  Calculate the DFT vector DFT(S_q) of S_q[N − W + 1 : N];
2.  Set d_k=∞; Set Heap_k=∅; // d_k is the distance to the k-th nearest neighbor;
3.  Set node = the root of the R*-tree;
4.  foreach entry e of node
5.      if node is internal then
6.          Check if mindist(DFT(S_q), e) ≤ d_k+Δ_q;
7.      else
8.          Check if mindist(DFT(S_q), e) ≤ d_k;
9.  endfor
10. Sort relevant entries wrt mindist values;
11. Search subtrees recursively;
12. if a leaf node is reached then
13.     Use the multi-step k-NN algorithm to update Heap_k if required
```

Fig. 6. Outline of similarity k-NN query processing algorithm.

value, the performance of queries will degrade. Moreover, if a few streams are characterized by intense fluctuations, this will have a direct impact on Δ_u (and hence to Δ_q), despite the fact that the majority of the streams do not change their values in a sharp manner. Note that, a large value of Δ_q implies a larger query radius e. This means that more index MBRs will overlap the query range, leading to increased processing cost.

For example, assume that the parameter Δ_u is 10 and the parameter Δ_q is also 10. Moreover, assume that a number of streams have intense temporary fluctuations, so the parameter Δ_u increases to 20 to keep the update frequency of the index stable. In the IDC-Index with global query expansion, Δ_q increases to 20, since it is equal to the maximum value of Δ_u seen so far. When the intense fluctuations are over, Δ_u will decrease to 10 but the Δ_q will not be reduced in order to preserve the correctness of the algorithm.

Instead of using only one Δ_q for all streams, we can maintain one Δ_q (local Δ_q) for each entry of the R^*-tree. A parent entry has its local Δ_q set to the maximum Δ_q of all the entries in its subtree. By using a local Δ_q, a query covers a minimum region, since the expansion of the query is as small as possible. This implies that fewer MBRs will overlap the query range, resulting in a more efficient processing scheme.

On the other hand, the maintenance of local Δ_q for each entry requires some additional cost. As it is demonstrated in the experimental results, the use of the local Δ_q is suggested only when the number of queries is significantly larger than the number of updates in the workload.

3.4.1. Updating the index

As in the case of IDC-Index with global query expansion, the IDC-Index with local query expansion uses the parameter Δ_u. The process of selecting and maintaining the value of the parameter in this case is the same. Recall that with global Δ_q the DFT vector of the leaf is updated every time. A full bottom-up update occurs only when the difference between the new DFT vector and the last recorded DFT vector exceeds the Δ_u threshold.

In the IDC-Index with local Δ_q, the parameter Δ_q is updated every time, regardless if an update occurs or not, in order to preserve the correctness of the query processing algorithm. The update of the local Δ_q is performed by a bottom-up procedure. Notice that the cost to update the local Δ_q which is a real number is much less than the cost to update the MBRs of the R^*-tree. Therefore, significant savings in computation may be achieved.

When an update occurs, as we update the MBRs of internal nodes, we also update the local Δ_q by choosing as local Δ_q of the parent entry the maximum local Δ_q of the child node. If a new value arrives and there is no need to update the MBRs, we update only the local Δ_q. First, we update the local Δ_q of the leaf entry. To proceed to the upper levels of the tree, the following condition must be satisfied:

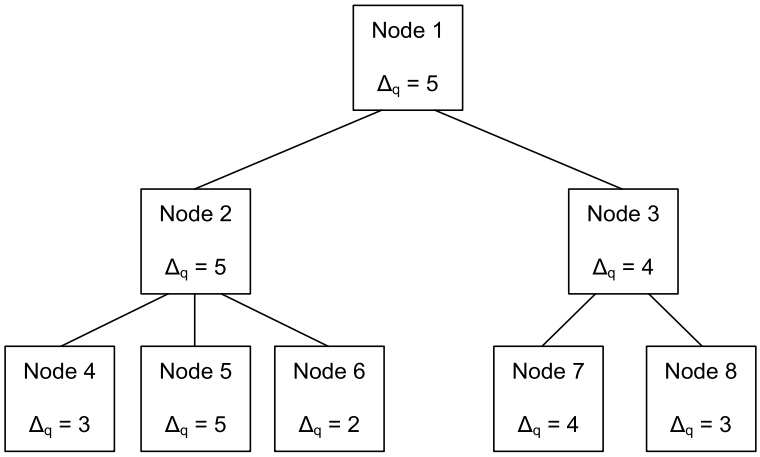


Fig. 7. IDC-Index with local Δ_q.

$$((\Delta_{q\text{NEW}} < \Delta_{q\text{OLD}}) \wedge (\Delta_{q\text{PARENT}} = \Delta_{q\text{OLD}})) \vee ((\Delta_{q\text{NEW}} > \Delta_{q\text{OLD}}) \wedge (\Delta_{q\text{PARENT}} < \Delta_{q\text{NEW}}))$$

Fig. 7 shows an example to clarify the above condition. To simplify the example, we use local Δ_q only for the nodes and not for the entries. It is obvious that the local Δ_q of a node is equal to the greater local Δ_q of its entries. We first examine the case where the new local Δ_q is smaller than the old local Δ_q of a node. If the new local Δ_q of node 5 is 4 then the local Δ_q of the father (node 2) should be reduced (because the old local Δ_q is equal to local Δ_q of the father). The modifications proceed up to the root (node 1). If the new local Δ_q of the node 4 is 2 then no modifications of the father are required since the local Δ_q of the father is greater than the old local Δ_q and therefore depends on the local Δ_q of node 5. Now we examine the case where the new local Δ_q is greater than the old local Δ_q of a node. If the new local Δ_q of node 5 is 6 then the local Δ_q of the father should be increased. If the new local Δ_q of node 4 is 4 then no modifications of the father are required since the local Δ_q of the father is greater than the new local Δ_q and therefore depends on the local Δ_q of node 5.

3.4.2. Query processing issues

The algorithms for both range and nearest neighbor queries in IDC-Index with local Δ_q are similar to Figs. 5 and 6. The difference for the range query (lines 2 and 3 of Fig. 5) is that it is not used a fixed query region r. The query region initially is $r = e$, so it is expanded for each entry by the local Δ_q of the entry. Similarly, the difference for the nearest neighbor query (line 6 of Fig. 6) is that the MinDist between the query and the entry is compared with the sum of the kth nearest neighbor distance and the local Δ_q of the entry. Propositions 2 and 3 can be modified by replacing Δ_q with local Δ_q. Therefore, the correctness of the query algorithms is proved.

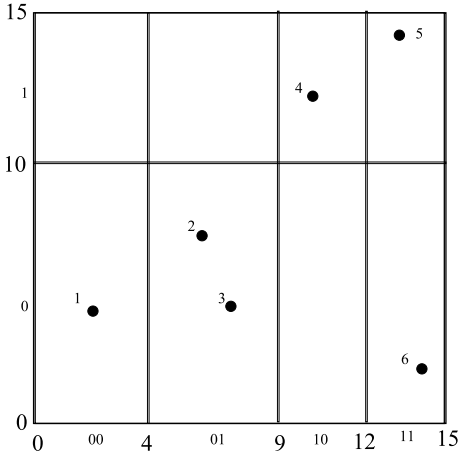
4. Performance study

4.1. The VA$^+$-stream approach

Before we present the experimental results, we briefly describe the VA$^+$-stream access method which has been proposed in [17] as a similarity search method in streaming time series. The VA$^+$-stream is based on VA$^+$-file [22], a structure that has been proposed as an index method to overcome the dimensionality curse and to support efficient similarity search for non-uniform data.

Since a streaming time sequence contains a large number of values, similarity is expressed with respect to the W last values of the streams. This is applied to the query time sequence as well. Therefore, if $W = 256$ then each sequence is represented as a point in the 256-dimensional space.

The VA$^+$-stream divides the data space into 2^b cells, where b is a user-specified parameter. The VA$^+$-stream allocates different number of bits for each dimension. The sum of these bits is equal to b. Each cell is an approximation of the data points that fall into this cell and is represented by a bit string of length b. An example of six time sequences in the 2D space ($W = 2$) with $b = 3$ is given in Fig. 8.



	Data Point	Cell Vector
1	(2,4)	00 0
2	(6,6)	01 0
3	(7,4)	01 0
4	(10,12)	10 1
5	(13,14)	11 1
6	(14,2)	11 0

Fig. 8. An example of VA$^+$-stream.

The data stream values considered are obtained by a sliding window which always contains the last W stream values. In order to adjust the structure due to the newly arrived values, a bit reallocation method is applied. Each dimension is quantized independently with its assigned bits, in order to achieve the least reproduction error. A drawback of this approach is that requires all the streams to have new values in order to adjust the structure, in contrast with IDC-Index which can manipulate streams with different data collection rate. The VA$^+$-stream access method can answer similarity range and nearest-neighbor queries.

The performance of this approach is highly dependent on the number of bits associated with each dimension. The VA$^+$-stream divides the space into 2^b cells, where b is the total number of bits. Since many of these cells are not used, to reduce the number of the cells the authors proposed a structure named CSET. This structure stores the cells in which the streams lie. The drawback of this structure is its size. To define a cell, if d dimensions are used (i.e. the window size is d), then d integers are required. Therefore, the size of the CSET structure is $n \cdot d$ integers, where n is the number of streams. We used the CSET structure described in [17].

4.2. Experimental results

In this section, we report the experimental results performed on real-life data sets. All methods are implemented in C++ and experiments have been conducted on a Pentium IV system at 3 GHz, with 1 GB of main memory running Windows XP. We have conducted a series of experiments to evaluate the performance of the IDC-Index and its variations. We have used the VA$^+$-stream and the sequential scanning methods as the competitors of IDC-Index. We use the labels SS for the sequential scanning algorithm, IDC-INDEX for the scheme with a global Δ_q and IDC-LOCAL for the scheme with the local Δ_q.

As we have already mentioned, one of the major disadvantages of the VA$^+$-stream is that it requires all the streams to have a new value in order to update the VA structure. On the other hand, the local IDC-Index (IDC-Index with local Δ_q) can be efficiently used only in cases where a fraction of the streams is updated in each time instance. Therefore, we divided the experiments into two categories. Experiments in which all the streams are updated in each time instance and experiments in which a fraction of the streams is updated in each time instance. In the first category, we compared the global IDC-Index (IDC-Index with global Δ_q), the VA$^+$-stream and the SS and in the second category we compared the global and the local IDC-Index. All the required data structures are maintained in main memory.

Both range and k-NN queries are considered. The VA$^+$-stream is proposed for k-NN queries [17], so we modified the method to be applicable for range query processing. We have studied the performance of the methods by varying several of the most important parameters such as the query distance e in range queries, the value k for k-NN queries, the desirable update ratio, the buffer size, the length of the sliding window, the number of DFT coefficients and the workload. The workload is composed of queries intermixed with updates. An update operation includes all the updates in a specific time instance. We have measured the CPU cost per

query and per update, the number of disk accesses and the number of candidates. Moreover, we have studied the behavior of the R^*-tree and the space requirements of the data structures.

The default values for the parameters (if not otherwise specified) are: the distance e has been chosen so that 1% of the streams to be in the answer. The desirable update ratio is 0.1%. Thus only 0.1% of the streams will be actually updated in each time instance. Among the DFT coefficients the first four and the last four are used (eight in total) [18]. The buffer size is set to 10% of the total number of pages on the disk. The sliding window size is set 256. The workload is another important parameter. We have chosen two different workloads: (i) a "heavy" workload comprising 20% queries and 80% updates, and (ii) an "light" workload comprising 80% queries and 20% updates. The real-life data sets used are described below shortly:

- *Stocks*: contains daily stock prices obtained from http://finance.yahoo.com. In order to have an adequate number of streams, we have generated new ones by transposing values of the real streams. The data set consists of 50,400 time sequences, each with a length of 1500.
- *Tropical Atmosphere Ocean (TAO)*: contains wind speed values of 65 sites on Pacific and Atlantic Ocean since 1974. The set is available by the Pacific Marine Environmental Laboratory (http://www.pmal.noaa.gov/tao), and consists of 12,217 series, each having a length of 1000.

4.2.1. Estimation accuracy

Recall that the update ratio U defines the number of updates that will be performed in the index, with respect to the total number of updates. For this reason the parameter Δ_u is used. The way Δ_u is calculated, has been described in a previous section. Here, we demonstrate the accuracy in preserving the required update ratio. Tables 2 and 3 illustrate the results for the STOCKS and TAO data sets, respectively. For each data set two different workloads have been used. The first workload contains 100 range queries and 400 update operations, whereas the second workload contains 400 range queries and 100 updates. Note that all streams update their values, which means that the number of update operations is multiplied by the number of streams to obtain the total number of updates.

The first column contains the desired update ratio. The column labeled "Estimated" contains the estimated number of index updates that should be performed to guarantee the desired update ratio. The column labeled "Real" shows the number of updates that actually took place. It is evident that the number of updates performed is very close to the predicted value. This means that the estimation of Δ_u manages to maintain the desired update ratio very accurately.

4.2.2. Performance of range similarity queries

In this section, we show the experimental results obtained from range queries for the first category where all streams are updated in each time instance. In the first experiment, we show the performance of the three methods with respect to the distance e. Figs. 9 and 10 illustrate the results for the STOCKS and the TAO data set, respectively. Both workloads have been used but here we present only the most representative results since the behavior of the methods is expected. In Fig. 9, the "light" workload is used while in 10 the "heavy" workload

Table 2
Estimation accuracy for STOCKS data set

Workload/Update ratio	100 queries–400 updates		400 queries–100 updates	
	Estimated	Real	Estimated	Real
0.05	12,079	12,078	4519	4514
0.1	21,159	21,160	6039	6038
1	201,695	201,708	50,498	50,526
5	1,007,999	1,007,991	252,014	252,009
10	2,015,969	2,015,949	503,999	503,993
20	4,031,924	4,031,888	1,007,984	1,007,914
50	10,079,801	10,079,624	2,519,951	2,519,790
100	20,159,600	20,159,600	5,039,900	5,039,900

Table 3
Estimation accuracy for TAO data set

Workload/update ratio	100 queries–400 updates		400 queries–100 updates	
	Estimated	Real	Estimated	Real
0.05	4442	4445	2610	2680
0.1	5885	5889	2221	2233
1	48,963	48,958	12,315	12,317
5	24,4339	24,4328	61,099	61,099
10	488,649	488,654	122,169	122,172
20	977,284	977,284	244,324	244,325
50	2,443,201	2,443,205	610,801	610,802
100	4,886,400	4,886,400	1,221,600	1,221,600

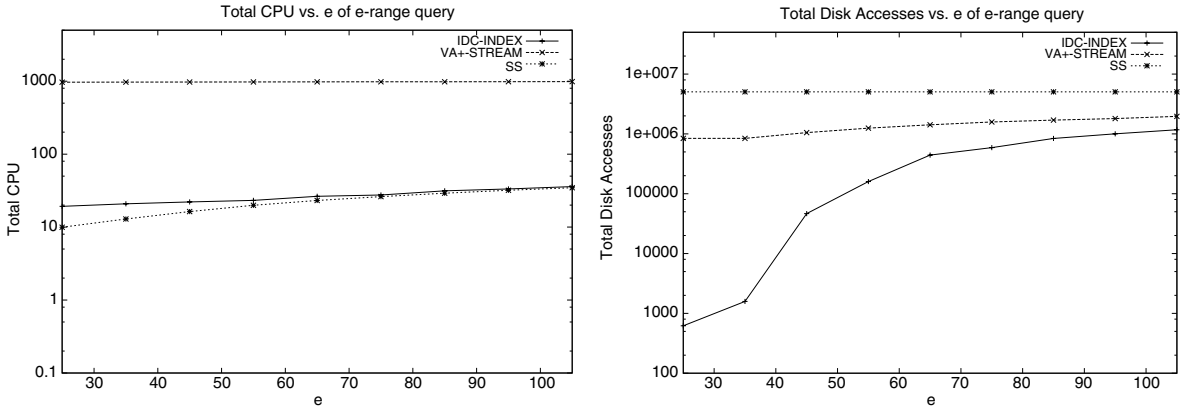


Fig. 9. CPU time (left) and disk accesses (right) vs. e for STOCKS data set (“light” workload).

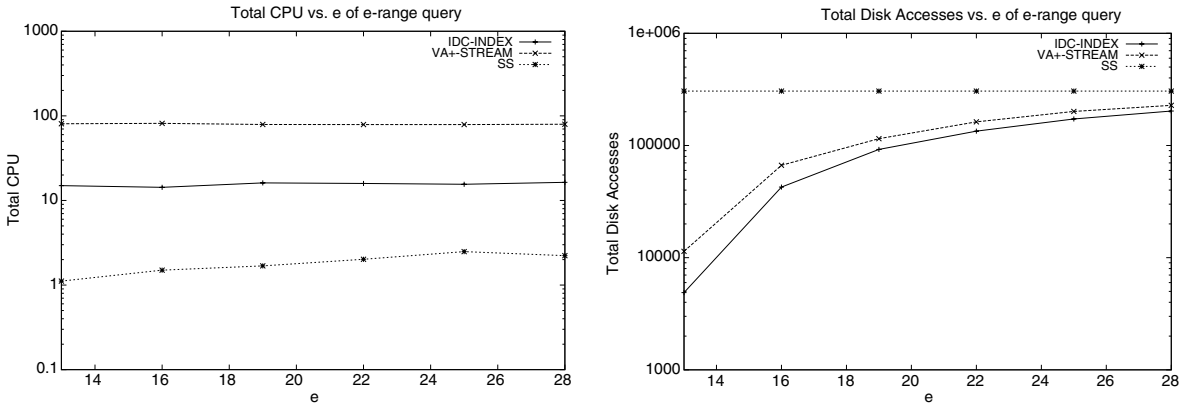


Fig. 10. CPU time (left) and disk accesses (right) vs. e for TAO data set (“heavy” workload).

is used. Both CPU cost and disk accesses are given in each experiment. The CPU cost is given in seconds. The y-axis is in logarithmic scale.

The IDC-Index outperforms the other two methods. Notice that the CPU cost of SS is less than that of the IDC-Index when the number of queries is low. This is expected since the SS does not require any index updates. We emphasize that the problem is disk-dominated so the number of disk accesses determines the total performance of a method. As e increases, the difference between the three methods is decreased

because the number of streams that are contained in the final answer increases rapidly. Moreover in the "light" workload the gap between IDC-Index and VA$^+$-stream is greater because IDC-Index processes queries faster than VA$^+$-stream since IDC-Index introduces less false dismissals as we will show later in the experiments.

Fig. 11 illustrates the performance of the methods with respect to variable workload. The IDC-Index is steadily more efficient than VA$^+$-stream and SS. Again the CPU cost of SS is less than that of the IDC-Index when the number of queries is low. As we mentioned above this is expected since the SS does not use an index structure. The gain from the disk accesses surpasses the CPU cost. IDC-Index outperforms VA$^+$-stream especially when the number of queries is high. That is because, as already mentioned in the previous experiment, IDC-Index achieves better hit ratio than VA$^+$-stream. This impacts in the number of disk accesses, thus the difference between IDC-Index and both the other two methods increases. Recall that the y-axis is in logarithmic scale.

The number of DFT coefficients has an important impact on the performance. As the number of DFT coefficients increases, distance preservation is improved and therefore, less false alarms are introduced. Fig. 12 shows the hit ratio with respect to the number of the DFT coefficients both for STOCKS and TAO data sets. For the STOCKS data set the hit ratio of the IDC-Index is much better than that of the VA$^+$-stream because the stocks values match with the DFT properties. On the contrary, to achieve a good hit ratio for the TAO data set, more DFT coefficients are required. A reasonable question is how the number of coefficients influences the CPU cost. Fig. 13 depicts the CPU cost and the disk accesses for the TAO data set. The gain from

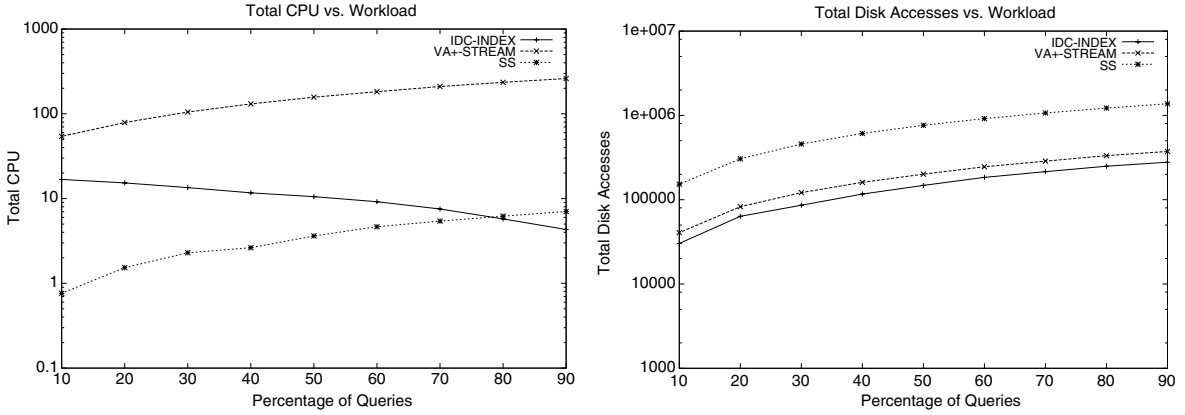


Fig. 11. CPU time (left) and disk accesses (right) vs. workload for TAO data set.

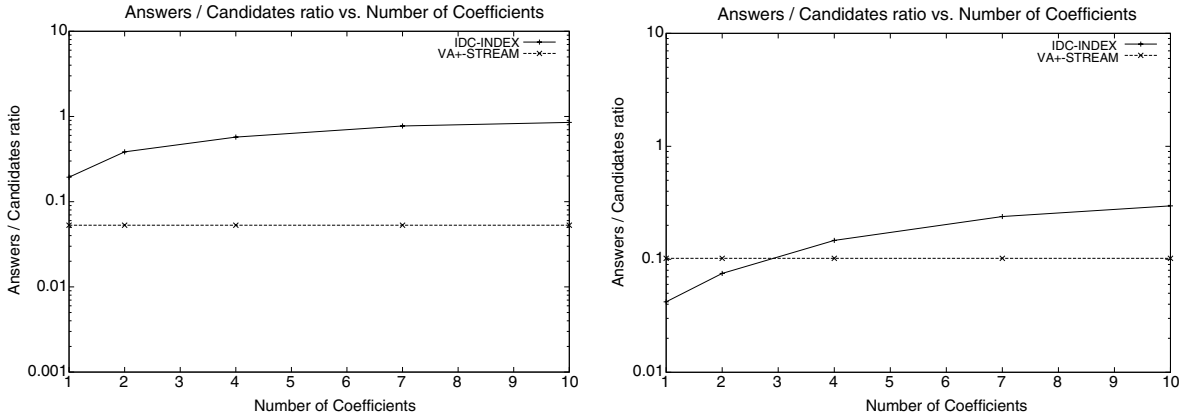


Fig. 12. Hit ratio vs. number of DFT Coefficients for STOCKS (left) and TAO (right) data sets.

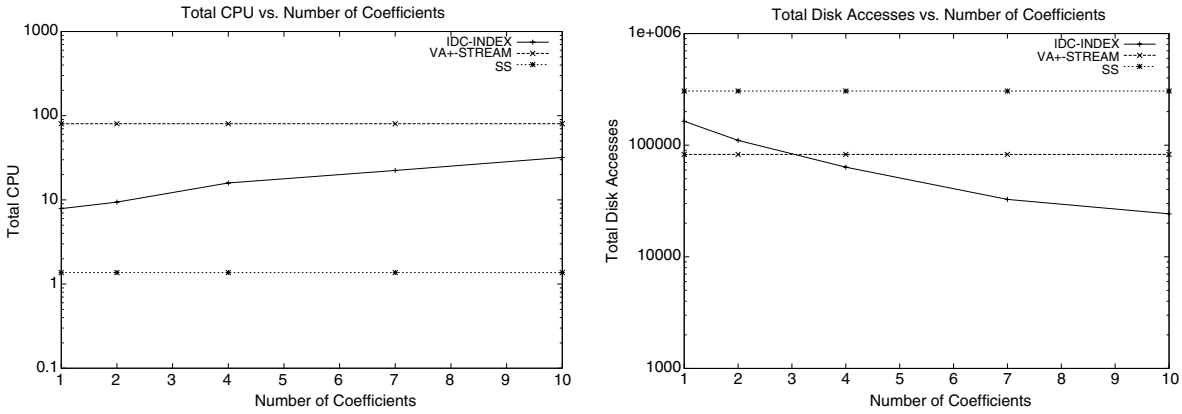


Fig. 13. CPU time (left) and disk accesses (right) vs. number of DFT coefficients for TAO data set.

the reduction of the disk accesses is more than the overhead imposed by the CPU. For example, in Fig. 13 the difference between the use of two and eight coefficients, is about 8 seconds for the CPU and 100,000 for the number of disk accesses. Thus it is better to use an adequate number of DFT coefficients sacrificing low CPU cost. Achieving a good hit ratio is important because hit ratio impacts the query efficiency and thus the overall method.

We also studied the behavior of the method with respect to specified update ratio. As the update ratio increases, the update cost is increased and the query cost is reduced. That is because the parameters Δ_u and Δ_q have small value therefore the R^*-tree is more up-to-date so the tree traversal is restricted in less nodes (recall that the query is expanded by Δ_q). We chose a very low update ratio for the experiments because, Fig. 14 shows, the gain for the query is not significant. Moreover the update ratio does not affect the number of candidates since the leaves of the index have always the current DFT coefficients so the number of disk accesses is not affected by the specified update ratio.

An advantage of the IDC-Index is that it can handle different window sizes. Fig. 15 illustrates the CPU cost and the number of disk accesses with respect to window size. The IDC-Index is again more efficient than the other two methods. The CPU cost for the IDC-Index is almost unaffected from the window size since the number of DFT coefficients is fixed. It is expected that the number of disk accesses will increase as the window size is increased since the pruning ability of R^*-tree is reduced. The other two methods are affected from the window size since they perform operations in each dimension. Therefore as the number of dimensions increases the CPU cost is increased.

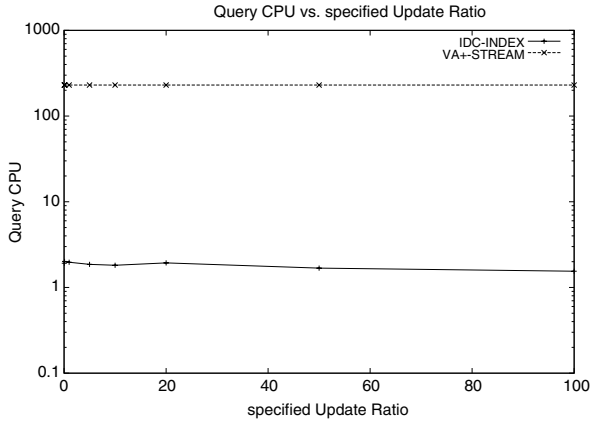


Fig. 14. CPU time vs update ratio (U) for TAO data set.

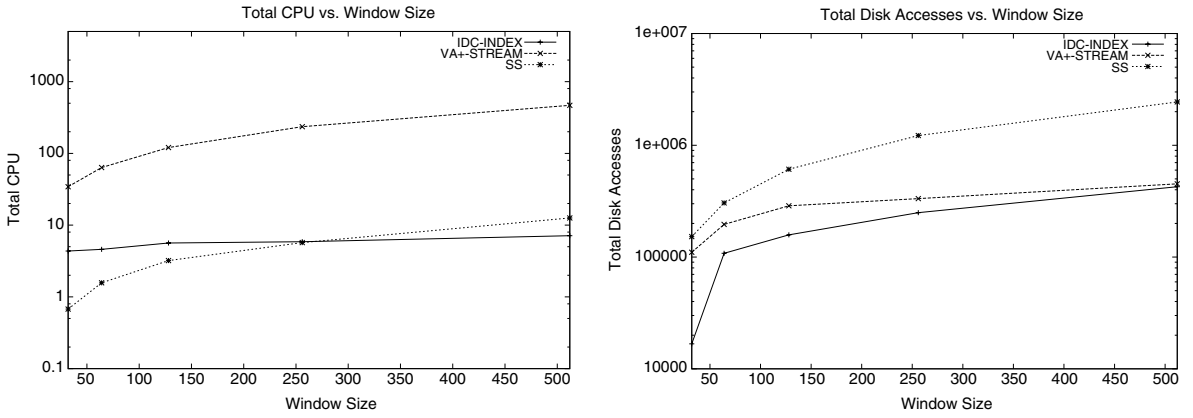


Fig. 15. CPU time (left) and disk accesses (right) vs sliding window size (W) for TAO data set.

Fig. 16 shows the space requirements for the two methods for the STOCKS and TAO data sets, with respect to the sliding window size. The space requirements of the IDC-Index remains fairly constant. On the other hand, the size of the CSET structure of the VA$^+$-stream increases linearly with respect to the sliding window size. Recall that the CSET structure uses one integer for each dimension to determine the cell of each stream.

4.2.3. Performance of k-NN similarity queries

In the sequel, we studied the performance of the three methods in k-NN query processing. The first experiment studies the performance of the methods with respect to k. Figs. 17 and 18 show the results for the STOCKS and the TAO data set, respectively. IDC-Index is consistently more efficient than SS and VA$^+$-stream. It is evident that the impact of k is not significant. The methods have the same behavior in both k-NN and range queries. Again the CPU cost of SS is less than that of the IDC-Index but the gain of disk accesses surpasses this cost. Moreover, IDC-Index is better than VA$^+$-stream since IDC-Index achieves better hit ratio.

Fig. 19 depicts the performance with respect to the workload. The CPU cost of SS is less than that of the other methods, whereas the I/O cost dominates. IDC-Index outperforms VA$^+$-stream since IDC-Index processes queries faster than VA$^+$-stream.

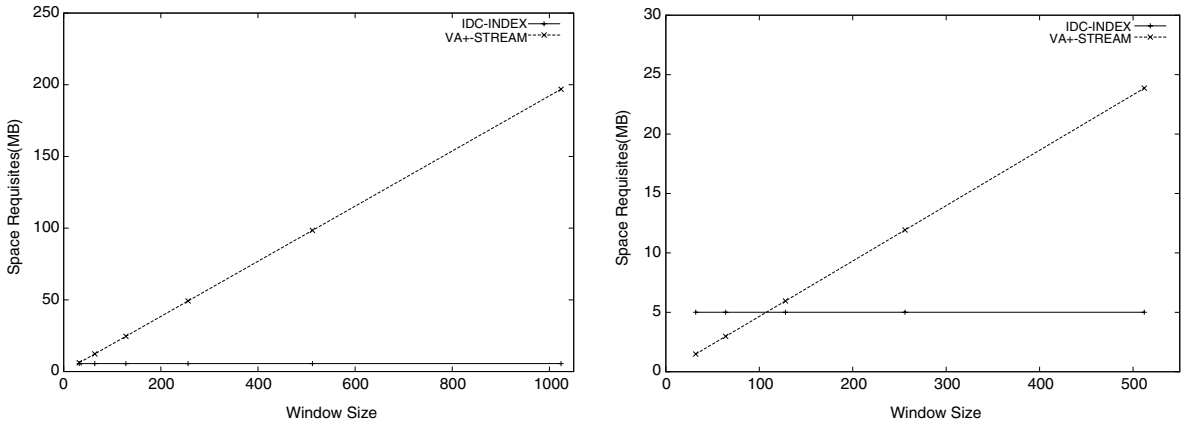


Fig. 16. Storage requirements of IDC-Index and VA$^+$-stream schemes for STOCKS (left) and TAO (right) data sets vs. sliding windows size.

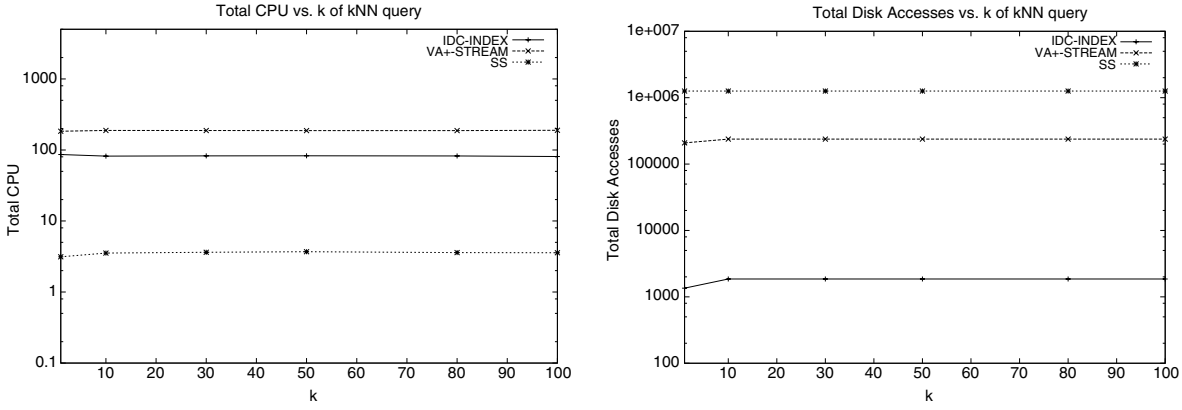


Fig. 17. CPU time (left) and disk accesses (right) vs. k for STOCKS data set.

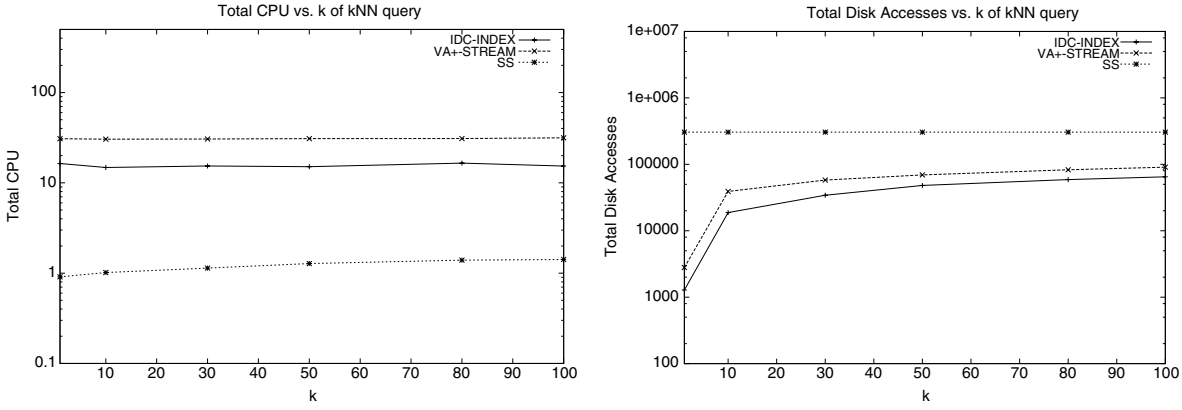


Fig. 18. CPU time (left) and disk accesses (right) vs. k for TAO data set.

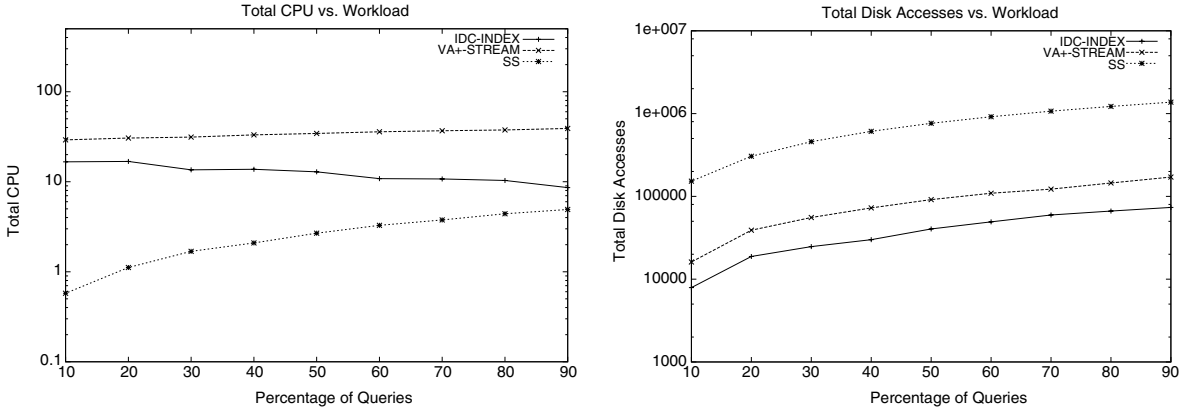


Fig. 19. CPU time (left) and disk accesses (right) vs. workload for TAO data set.

As we already mentioned, we have used a buffering for the IDC-Index and the VA$^+$-stream. For SS the use of buffer is meaningless, since for each query all the streams are accessed. The buffer has an important role in the performance. We have studied the performance with respect to the buffer size and the results are illustrated

in Fig. 20. The buffer size is expressed as the percentage of the total number of pages occupied by the data. Both methods are equally affected by the buffer size.

Next, we examine the impact of the sliding window size to the performance of the methods. It is expected that the CPU cost for the IDC-Index will not be affected significantly, since the number of coefficients remains fixed. The number of disk accesses of the IDC-Index increases due to the loss of information as the window size increases. Nevertheless, the IDC-Index method is more efficient than the VA$^+$-stream and SS, as it is depicted in Fig. 21.

4.2.4. Comparison of global and local IDC-index

In this section, we show the experimental results obtained by the comparison of global and local IDC-Index. In these experiments, only a fraction of the streams is updated in each time instance. The specified update ratio is set equal to 1%. We studied the performance of the methods both for range and k-NN queries. Notice that both methods have back pointers at the leaves and therefore have the current DFT values. Thus the number of disk accesses is the same.

The first experiment shows the CPU cost of the methods in executing k-NN queries with respect to the ratio of the streams that are updated. The ratio varies from 0.1% to 10%. We used two different workloads: (a) 20% queries and 80% updates and (b) 80% queries and 20% updates. Fig. 22 depicts the results for $k = 10$ for the TAO data set. It is observed that the local IDC-Index shows better performance, especially when the workload contains more queries than updates and the ratio of updated streams is low. Recall that

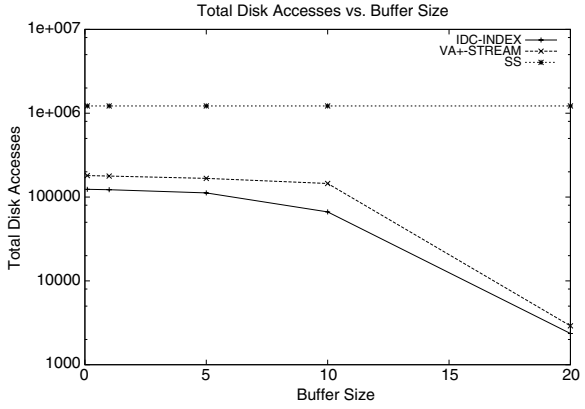


Fig. 20. Disk accesses vs. buffer size for TAO data set.

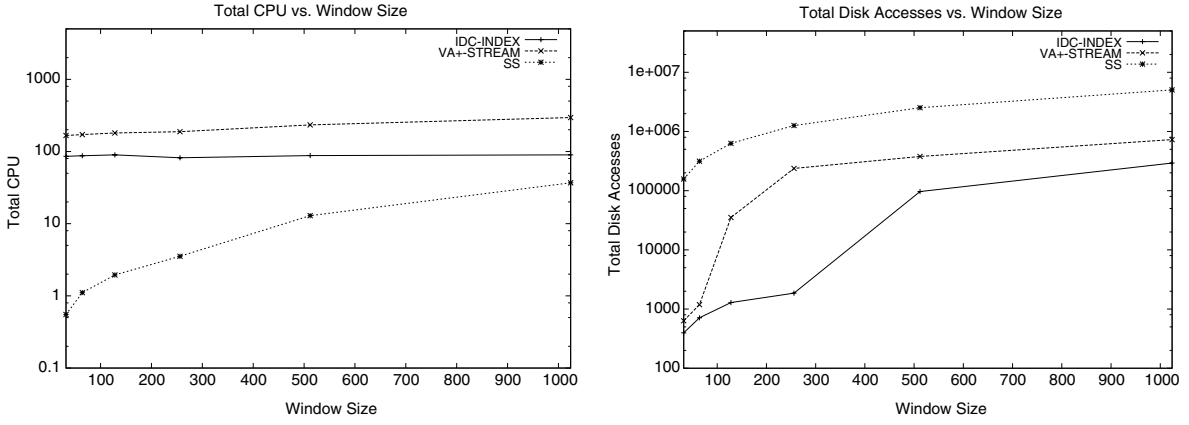


Fig. 21. CPU time (left) and disk accesses (right) vs. sliding window size for STOCKS data set.

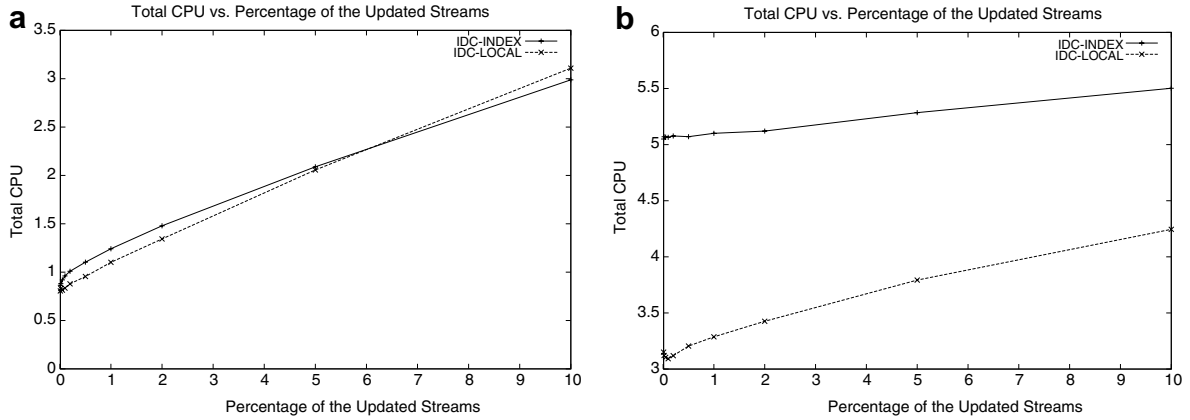


Fig. 22. CPU time vs. stream update ratio (TAO data set, $k = 10$). (a) 20% queries, 80% updates. (b) 80% queries, 20% updates.

the IDC-Index with local Δ_q updates the local Δ_q of the leaf when a new value arrives. This can cause a bottom-up update to the upper levels. If the number of updates is much more than the number of queries or the ratio of streams that are updated is high, then the IDC-Index with local Δ_q cannot perform well. Fig. 23 shows the same results for $k = 1000$ for the TAO data set. The results are similar to those of the previous case.

An important parameter in the IDC-Index is the specified update ratio. If the update ratio is low, then the update operation is fast, whereas query processing efficiency may be affected. Since the update operation is implemented in a bottom-up manner, the question is how this modification affects the performance of the R^*-tree and therefore the overall query execution time. The last experiment compares the performance of our modified R^*-tree to that of a regular R^*-tree, which is reconstructed in each time instance in order to guarantee the quality of the structure. We used both the global and the local IDC-Index. Fig. 24 illustrates the query CPU cost and total CPU cost with respect to k, for the "light" workload. As expected, the reconstructed R^*-tree has slightly better performance with respect to the query CPU time, since the quality of the structure is higher, but not enough to surpass the total CPU overhead. Notice that the gap between the reconstructed R^*-tree and the modifications is of the order of 4 seconds at most for the query CPU time. In contrast, the gap for the total CPU time is of the order of up to 1500 seconds, since the reconstruction of the structure is a very costly operation. Moreover the IDC-Index with local Δ_q is better than the IDC-Index with global Δ_q. This was expected, since the use of the local Δ_q restricts the expansion of the query and therefore less tree nodes are accessed.

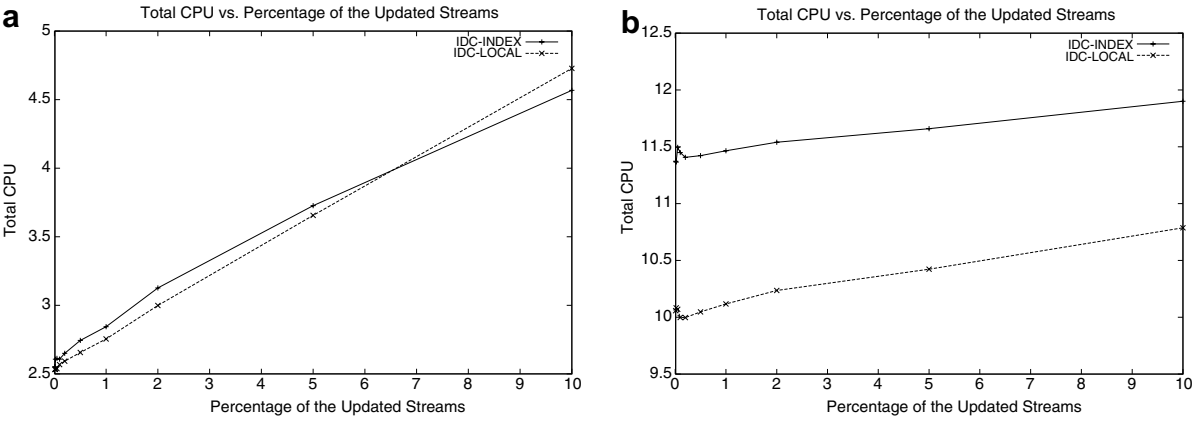


Fig. 23. CPU time vs. stream update ratio (TAO data set, $k = 1000$). (a) 20% queries, 80% updates. (b) 80% queries, 20% updates.

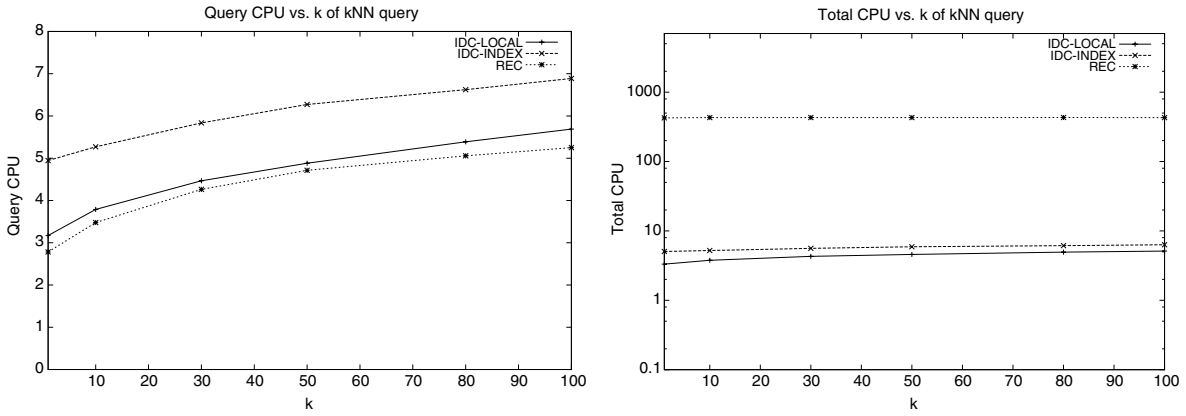


Fig. 24. Query and total CPU time vs. k ("light" workload).

5. Concluding remarks

Data stream processing is an active area of research aiming at the design of efficient methods for handling time evolving data with frequent updates. Streaming time series compose a special category of data streams that appear in many applications such as network monitoring, sensor networks, financial applications, telecommunications data management. A streaming time series is a sequence of data values, where new values are continuously appended as time progresses. In many cases we are interested only in the recent values of a streaming time series. Therefore, a sliding window of length W is defined to capture the last W values of each streaming time series.

An important operation in streaming time series is to determine similar time series with respect to a query series. Similarity is expressed by means of the last W values of the streams. In this paper, we have studied similarity range queries and similarity nearest-neighbor queries in such an environment. More specifically, we addressed the issues of: (1) incremental feature extraction, (2) efficient in-memory indexing by means of R^*-tree-based access methods, (3) algorithms for range and nearest-neighbor query processing, (4) adapting the indexing scheme to approximate the update frequency. Performance evaluation results have shown that significant improvement is achieved in comparison to a recent proposal based on the Vector Approximation File (VA-File), both in storage requirements and query processing efficiency. The current research can be extended in a number of different directions:

- the support of multiple continuous queries,
- the consideration of other similarity measures such as Dynamic Time Warping (DTW) [3] and Discrete Wavelet Transform (DWT) [19],
- the selection of the number of coefficients with respect to the properties of the data set, and
- the efficient processing of similarity join queries in a streaming environment.

Acknowledgement

The research supported by the PENED 2003 program, funded by the General Secretariat for Research and Technology, Ministry of Development, Greece.

Appendix

Proposition 1. *Let S be a streaming time series with values $S(0), S(1), \ldots, S(W-1)$ and length W. Moreover, let $DFT_0(S), DFT_1(S), \ldots, DFT_{W-1}(S)$ denote the DFT coefficients of S. If a new value for this stream arrives, we get*

the sequence $T(1), T(2), \ldots, T(W)$, *where* $S(i) = T(i)$ *for* $1 \leqslant i \leqslant W-1$ *and* $T(W)$ *is the new value. The DFT coefficients of T can be computed by the DFT coefficients of S according to the following equation*:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot \mathrm{DFT}_n(S) - S(0) + T(W)\right) \cdot \mathrm{e}^{\mathrm{j}2\pi n/W}, \quad (0 \leqslant n \leqslant W-1) \tag{4}$$

Proof. Note that $S(i) = T(i)$ for $1 \leqslant i \leqslant W-1$. The nth DFT coefficient of a streaming time series S is given by:

$$\mathrm{DFT}_n(S) = \frac{1}{\sqrt{W}} \sum_{k=0}^{W-1} S(k) \cdot \mathrm{e}^{-\mathrm{j}2\pi kn/W} \tag{5}$$

Similarly, the nth DFT coefficient of a streaming time series T is given by:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}} \sum_{k=0}^{W-1} T(k+1) \cdot \mathrm{e}^{-\mathrm{j}2\pi kn/W} \tag{6}$$

We begin with Eq. (6) and substitute the values of $\mathrm{DFT}(S_n)$ as follows:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}}(S(0) + S(1)\mathrm{e}^{-\mathrm{j}2\pi n/W} + \cdots + S(W-1)\mathrm{e}^{-\mathrm{j}2\pi(W-1)n/W} - S(0) + T(W))\mathrm{e}^{\mathrm{j}2\pi n/W} \tag{7}$$

By algebraic manipulations in the above equation and taking into consideration that $S(i) = T(i)$ for $1 \leqslant i \leqslant W-1$, and that $\mathrm{e}^{\mathrm{j}2\pi n/W} = \mathrm{e}^{-\mathrm{j}2\pi(W-1)n/W}$ we get:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}}(T(1) + T(2)\mathrm{e}^{-\mathrm{j}2\pi n/W} + \cdots + T(W-1)\mathrm{e}^{-\mathrm{j}2\pi(W-2)n/W} + T(W)\mathrm{e}^{-\mathrm{j}2\pi(W-1)n/W}) \tag{8}$$

which is exactly Eq. (4). □

Proposition 2. *Let S be a streaming time series with values* $S(0), S(1), \ldots, S(W-1)$ *and length W. Moreover, let* $DFT_0(S), DFT_1(S), \ldots, DFT_{W-1}(S)$ *denote the DFT coefficients of S. If a new value for this stream arrives, we get the sequence* $T(1), T(2), \ldots, T(W)$, *where* $S(i) = T(i)$ *for* $1 \leqslant i \leqslant W-1$ *and* $T(W)$ *is the new value. The real* $(DFT_n(T)_{\mathrm{real}})$ *and the imaginary* $(DFT_n(T)_{\mathrm{imag}})$ *part of the DFT coefficients of T can be computed by the DFT coefficients of S according to the following equations*:

$$\mathrm{DFT}_n(T)_{\mathrm{real}} = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot \mathrm{DFT}_n(S)_{\mathrm{real}} - S(0) + T(W)\right) \cdot \cos\left(\frac{2\pi n}{W}\right) - \mathrm{DFT}_n(S)_{\mathrm{imag}} \cdot \sin\left(\frac{2\pi n}{W}\right) \tag{9}$$

and

$$\mathrm{DFT}_n(T)_{\mathrm{imag}} = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot \mathrm{DFT}_n(S)_{\mathrm{real}} - S(0) + T(W)\right) \cdot \sin\left(\frac{2\pi n}{W}\right) + \mathrm{DFT}_n(S)_{\mathrm{imag}} \cdot \cos\left(\frac{2\pi n}{W}\right) \tag{10}$$

where $(0 \leqslant n \leqslant W-1)$.

Proof. Using Eq. (4), we substitute the $\mathrm{DFT}_n(S)$ with the real and the imaginary part:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot (\mathrm{DFT}_n(S)_{\mathrm{real}} + j \cdot \mathrm{DFT}_n(S)_{\mathrm{imag}}) - S(0) + T(W)\right) \cdot \mathrm{e}^{\mathrm{j}2\pi n/W}$$

Using the Euler's formula, we get:

$$\mathrm{DFT}_n(T) = \frac{1}{\sqrt{W}} \cdot \left(\sqrt{W} \cdot (\mathrm{DFT}_n(S)_{\mathrm{real}} + \mathrm{j} \cdot \mathrm{DFT}_n(S)_{\mathrm{imag}}) - S(0) + T(W)\right) \cdot \left(\cos\left(\frac{2\pi n}{W}\right) + \mathrm{j} \cdot \sin\left(\frac{2\pi n}{W}\right)\right)$$

By algebraic manipulations in the above equation we get Eqs. (7) and (8). □

References

[1] R. Agrawal, C. Faloutsos, A. Swami, Efficient similarity search in sequence databases, in: Proceedings of the 4th International Conference on Foundations of Data Organization and Algorithms (FODO 1993), Evanston, IL, USA, 1993, pp. 69–84.

[2] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom, Models and issues in data stream systems, in: Proceedings of the 21st ACM SIGACT-SIGMOD-SIGART on Symposium Principles of Database Systems (PODS 2002), Madison, WI, 2002, pp. 1–16.

[3] D. Berndt, J. Clifford, Using dynamic time warping to find patterns in time series, in: Proceedings of the Workshop on Knowledge Discovery in Databases, 1994, pp. 359–370.

[4] B. Babcock, M. Datar, R. Motwani, L. O'Callaghan, Maintaining variance and k-medians over data stream windows, in: Proceedings of the Symposium on Principles of Database Systems (PODS'03), 2003, pp. 234–243.

[5] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the ACM SIGMOD, Atlantic City, NJ, 1990, pp. 322–331.

[6] S. Babu, J. Widom, Continuous queries over data streams, ACM SIGMOD Record 30 (3) (2001) 109–120.

[7] S. Chandrasekaran, M.J. Franklin, Streaming queries over streaming data, in: Proceedings of the 28th International Conference on Very Large Databases (VLDB 2002), Hong Kong, China, 2002.

[8] G. Cormode, M. Datar, P. Indyk, S. Muthukrishnan, Comparing data streams using hamming norms (how to zero in), IEEE Transactions on Knowledge and Data Engineering 15 (3) (2003) 529–540.

[9] C. Faloutsos, M. Ranganathan, Y. Manolopoulos, Fast subsequence matching in time-series databases, in: Proceedings of the ACM SIGMOD Conference, Minneapolis, MN, USA, 1994, pp. 419–429.

[10] S. Guha, A. Meyerson, N. Mishra, R. Motwani, L. OCallaghan, Clustering data streams: theory and practice, IEEE Transactions on Knowledge and Data Engineering 15 (3) (2003) 515–528.

[11] L. Gao, X.S, Wang, Continually evaluating similarity-based pattern queries on a streaming time series, in: Proceedings of the, Madison, WI, 2002.

[12] L. Gao, Z. Yao, X.S. Wang, Evaluating continuous nearest neighbor queries for streaming time series via pre-fetching, in: Proceedings of the 28th International Conference on Very Large Data Bases (VLDB 2002), Hong Kong, China, 2002.

[13] L. Gao, X.S. Wang, Improving the performance of continuous queries on fast data streams: time series case, in: Proceedings of the ACM SIGMOD DMKD Workshop, Madison, WI, 2002.

[14] KLL02 D. Kwon, S. Lee, S. Lee, Indexing the current positions of moving objects using the lazy update R-tree, in: Proceedings of the Third International Conference on Mobile Data Management, Washington, DC, USA, 2002, pp.113–120.

[15] M. Kontaki, A.N. Papadopoulos, Efficient similarity search in streaming time sequences, in: Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM 2004), Santorini, Greece, 2004.

[16] M. Lee, W. Hsu, C.S. Jensen, B. Cui, K.L. Teo, Supporting frequent updates in R-trees: a bottom-up approach, in: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB 2003), Berlin, Germany, 2003, pp. 608–619.

[17] X. Liu, H. Ferhatosmanoglu, Efficient *k*-NN search on streaming data series, in: Proceedings of the 8th International Symposium on Spatial and Temporal Databases (SSTD 2003), Santorini, Greece, 2003.

[18] A.V. Oppenheim, R.W. Schafer, Digital Signal Processing, Prentice-Hall, Englewood Cliffs, NJ, 1975.

[19] C. Sidney Burrus, R.A. Gopinath, H. Guo, Introduction to Wavelets and Wavelet Transforms, Prentice-Hall, 1997.

[20] T. Seidl, H.-P. Kriegel, Optimal multi-step k-nearest neighbor search, in: Proceedings of the ACM SIGMOD Conference, Seatle, WA, USA, 1998, pp.154–165.

[21] T. Sellis, N. Roussopoulos, C. Faloutsos, The R+ tree: a dynamic index for multidimensional objects, In: Proceedings of the 13th International Conference on VLDB (VLDB 1987), England, 1987, pp. 507–518.

[22] R. Weber, H.-J. Schek, S. Blott, A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces, in: Proceedings of the 24th International Conference on Very Large Databases (VLDB 1998), New York City, NY, 1998, pp.194–205.

[23] H. Wu, B. Salzberg, D. Zhang, Online event-driven subsequence matching over financial data streams, in: Proceedings of the ACM SIGMOD Conference, Paris, France, 2004.

Maria Kontaki received her B.S. degree in Computer Science from the Aristotle University of Thessaloniki, and she is currently a Ph.D. student in the Department of Informatics of Aristotle University of Thessaloniki. Her research interests include data streams processing, data mining and data management issues in sensor networks. Further information can be found at http://delab.csd.auth.gr/kontaki.

Apostolos N. Papadopoulos was born in Eleftheroupolis, Greece in 971. He received his 5-year Diploma degree in Computer Engineering and Informatics from the University of Patras and his Ph.D. degree from Aristotle University of Thessaloniki in 1994 and 2000 respectively. He has published several research papers in journals and proceedings of international conferences. From March 1998 to August 1998 he was a visitor researcher at INRIA Research Center in Rocquencourt, France, to perform research in spatial databases. His research interests include spatial and spatiotemporal databases, data stream processing, data mining and information retrieval. His research work has over 230 citations in scientific journals and conference proceedings. He has served as a track co-chair of ACM SAC DTTA (Database Technologies Techniques and Applications) Track 2005, 2006 and 2007. He is a member of the Technical Chamber of Greece. Currently, he is a Lecturer in the Department of Informatics of Aristotle University of Thessaloniki. Further information can be found at http://delab.csd.auth.gr/apostol.

Yannis Manolopoulos was born in Thessaloniki, Greece in 1957. He received a B.Eng (1981) in Electrical Eng. and a PhD (1986) in Computer Eng., both from the Aristotle Univ. of Thessaloniki. Currently, he is Professor at the Department of Informatics of the latter university. He has been with the Department of Computer Science of the Univ. of Toronto, the Department of Computer Science of the Univ. of Maryland at College Park and the Department of Computer Science of the University of Cyprus. He has published about 200 papers in refereed scientific journals and conference proceedings. He is co-author of the following books: "Advanced Database Indexing" and "Advanced Signature Indexing for Multimedia and Web Applications" by Kluwer, as well as "Nearest Neighbor Search: a Database Perspective" and "R-trees: Theory and Applications" by Springer. His published work has received over 1700 citations from over 450 institutional groups. He served/serves as General/PC Chair/Cochair of the 8th National Computer Conference (2001), the 6th ADBIS Conference (2002) the 5th WDAS Workshop (2003), the 8th SSTD Symposium (2003), the 1st Balkan Conference in Informatics (2003), the 16th SSDBM Conference (2004) and the 8th ICEIS Conference (2006), the 10th ADBIS Conference (2006). His research interests include Databases, Data mining, Web and Geographical Information Systems, Bibliometrics/Webometrics, Performance evaluation of storage subsystems. Further information can be found at http://delab.csd.auth.gr/manolopo.