Shift and Scale Invariant Search of Multi-attribute Time Sequences

Tamer Kahveci
Department of Computer Science

Ambuj Singh
Department of Computer Science

Aliekber Gurel
Department of Mathematics

University of California
Santa Barbara, CA 93106

{tamer,ambuj}@cs.ucsb.edu, aliekber@math.ucsb.edu

Abstract

We investigate the problem of searching similar multi-attribute time sequences in databases. Such sequences arise naturally in a number of medical, financial, video, weather forecast, and stock market databases where more than one attribute is of interest at a time instant. We formulate a new symmetric scale and shift invariant notion of distance for such sequences. We also propose a new index structure that transforms the data sequences and clusters them according to their shiftings and scalings. This clustering improves the efficiency considerably. According to our experiments with real and synthetic datasets, the index structure's performance is 5 to 60 times better than competing techniques, the exact speedup based on other optimizations such as caching and replication. Finally, we also consider the subsequence search problem.

1 Introduction

Time series or sequence data sets arise naturally in many real world applications like stock market, weather forecasts, video databases, sensor-based controls, and medicine. There is a frequent need to understand the information content of this data in order to respond better to common trends, to provide corrective emergency steps, or to predict the future evolution based on past records. Some examples of queries on such datasets include finding the companies which have similar profit/loss patterns, finding similar motion patterns in a video database, finding similar patterns in medical sensor data in order to respond to patient health problems, or to predict infrastructure usage by comparison with past trends.

Time series data is said to have d attributes if d values are stored for each time point. Stock market data, which is formed by storing the closing prices of a company is an example of an 1-attribute sequence. If we include the P/E ratio and the number of shares sold, this becomes a 3-attribute sequence. The trajectory of an object moving on a plane is a 2-attribute sequence, because two values (i.e. X and Y coordinates) are stored for each discrete time point. Medical data is usually multi-attribute since a single sensor is seldom sufficient to record the health of a patient: such a sequence can be obtained by using the blood pressure values, the heart beat rate, the amount of calcium, and other parameters recorded periodically from a patient. A database composed of such time sequences can be used to make predictions for illnesses and take corrective actions based on physician input.

There are many ways to compare the similarity of two time sequences. One approach is to define the distance between two sequences to be the Euclidean distance, by viewing a sequence as a point in an appropriate multi-dimensional space [1, 5, 8, 11, 15, 19]. Non-Euclidean metrics have also been used to compute the similarity for time sequences. Agrawal, Lin, Sawhney, and Shim [2] use L_∞ as the distance metric. The Landmark model by Perng, Wang, Zhang and Parker [14] chooses only a subset of values from a time sequence, which are peak points and uses them to represent the corresponding sequence. The authors define distance as a tuple of values, one representing the time and the other the amplitude. These two values together determine the dissimilarity. Another distance metric D_{norm} is defined by Lee, Chun, Kim, Lee, and Chung [13] for multi-attribute time sequences. Although this metric has a high recall, it allows false dismissals.

Range searches and nearest neighbor searches for *whole matching* and *subsequence matching* [1] have been the principal queries of interest for time series data. *Whole matching* corresponds to the case when the query sequence and the sequences in the database have the same length. Agrawal et al. [1] developed one of the first solutions to this problem. The authors transformed the time sequence to the frequency domain by using DFT. Later, they reduced the number of dimensions to a feasible size by storing the first few frequency coefficients. Chan and Fu [5] used Haar wavelet transform to reduce the number of dimensions and compared this method to DFT. The authors found that Haar wavelet transform performs better than DFT. However, the performance of DFT could be improved using the symmetry of Fourier Transforms [16]. In this case, both methods gave similar results.

Faloutsos, Ranganathan, and Manolopoulos [8] proposed *I-adaptive* index to solve the *subsequence matching* problem for queries of prespecified length. The authors store the trails of the prespecified length in MBRs (Minimum Bounding Rectangles). The same paper also presents the techniques of *Prefix Search* and *Multiple Search* to relax the restriction of prespecified query lengths. Prefix Search performs a database search using a prefix (of a prespecified length) of the query sequence. Multiple Search splits the query sequence to non-overlapping subqueries (of prespecified length) and performs database search for each of these subqueries. However, these two methods incur high cost if the query length is not prespecified.

Kahveci and Singh [11] proposed an efficient method for subsequence matching problem when the query length is not prespecified. The authors propose a new index structure, called *MR-index*. This index has a grid structure: each row stores information at a different resolution. A given query of arbitrary length is split into subqueries of resolutions available in the *MR-index*, and successive subqueries are performed on the index structure to refine the range of the query.

1.1 Need for scaling and shifting

Defining the distance as some norm of the difference between two time sequences may be insufficient if the sequences can be made closer by linear transformations. The most important transformations are scaling and shifting. Scaling is needed because of the need to compare time sequences recorded on devices with different calibrations or different units. For example, a medical sensor in North America may record body temperature in $^\circ F$ while a similar sensor in Europe may use $^\circ C$. A sensor may also lose exact calibration over time (e.g. weather sensors on a satellite) leading to a need to compare measurements after different scalings. A similar case can be made for comparing time series data after eliminating shifts. Usually this is because the zero value in a time sequence differs in different measurements, and the results of a query need to be invariant under all possible shiftings.

In a previous paper, Chu and Wong [6] defined the *Shift Eliminated Transformation*. This transformation maps data sequences onto the *Shift Eliminated Plane*, making the transformed data *shift-invariant*. The authors propose to store the transformed data in an R*-tree [9, 3]. The distance between a data sequence and a query is defined as the minimum distance between the shift eliminated data sequence and all possible scalings of the query sequence. However, as we will explain later in this paper, this distance function is not symmetric and may lead to counter-intuitive results.

In the Landmark model [14], the authors propose to store the turning points of the time sequences. The distance between time sequences here is invariant with respect to 6 transformations: shifting, uniform amplitude scaling, uniform time scaling, uniform bi-scaling, time warping, and non uniform amplitude scaling. However, storing the peak points of a time sequence partially eliminates the notion of time. This is because the peak points may be located at arbitrary places in the time sequences. Therefore, the duration between consecutive peak points may be different, and this information is unavailable to queries.

Dagtas, Khatip, Ghafoor and Kashyap [7] consider finding similar trails of moving objects in a video sequence. They consider time-invariant motions as well time-variant motions. In the case of time-invariant motion, time sequences are simply projected onto the X-Y plane, and a Mellin Transform is used to find a similar trail under shift and scale invariance. For the time-variant case, the focus of our paper, the authors do not consider scale invariant searching.

1.2 Our contribution

In this paper, we point out the problems with current methods for scale and shift invariant distance computations. We define a correct distance function for multi-attribute time sequences under all possible scalings and shiftings. This correction makes the definition of distance symmetric: the distance between two time sequences is defined to be the smallest Euclidean distance after scaling and shifting either one of the sequences to be as close to the other. We define two models for comparing multi-attribute time sequences: in the first model, the scalings and shiftings of the component sequences are dependent, and in the second model they are independent.

We propose a new index structure called *CS-Index* for shift and scale invariant comparison of time sequences. As a part of this technique, the sequences in the database are first mapped to the shift eliminated plane [6]. The transformed points are then clustered in hierarchical cone slices. These slices are stored on disk in a depth-first-search order, and a pointer to each slice along with angle and spatial extent information is maintained in memory. Given any query, it is first mapped onto shift eliminated plane. The shift and scale invariant distance between the query and the slices are computed in memory to obtain a set of candidate slices. The hierarchical construction of the index structure allows early pruning in this step. Finally, the candidate slices are read from disk in a single seek, and false hits are eliminated. The storage of the slices in a depth-first-search order on disk reduces the number of pages that need to be read.

Experimental results show that our method performs 5 to 10 times faster than other techniques. The efficiency of the index structure can be further improved by selectively replicating or caching parts of the index structure. If the pages at the first level of the index structure are replicated on disk, then the performance of our method improves further by a factor of two. Replicating one more level improves the performance slightly, but further replication has an adverse effect. Caching of the index structure (instead of replication) can also be used to improve the performance of the index structure. In our experiments, caching only one level of the index structure, led to a performance improvement of 15 to 40 times over other techniques.

The final part of the paper concentrates on *subsequence matching*. We propose another index structure called *Multi-Resolution CS index* (MRCS) that uses the idea of multiple resolutions [11]. This index structure stores a grid of cone slices. Each row of this grid corresponds to a different resolution, and is constructed by sliding a window of a prespecified length on the database sequences. A given query sequence is partitioned into subqueries corresponding to the resolutions available in the index structure. For a range query, the technique successively reduces the query range by performing an in-memory search of the subqueries. This index structure is dynamic and allows queries of arbitrary lengths.

The rest of the paper is organized as follows. We define the problem of shift and scale invariant search of multi-attribute time sequences, and explain the problem with the current method in Section 2. We also present a new symmetric distance function for shift and scale invariance. In Section 3 we propose the *CS-index* structure for range queries and nearest neighbor queries. We present experimental results on a number of synthetic and real datasets in Section 4. The MRCS structure for subsequence matching index structure is presented in Section 5. We end with a brief discussion in Section 6.

2 Definition of scale and shift invariance

A *d-attribute* time sequence is formed by storing d values at each time point. If v is a d-attribute time sequence of length l, then it can be represented as a vector $v = (v_1, v_2, ..., v_l)$, where $v_i = (v_{i,1}, v_{i,2}, ..., v_{i,d})$ for $1 \leq i \leq l$.

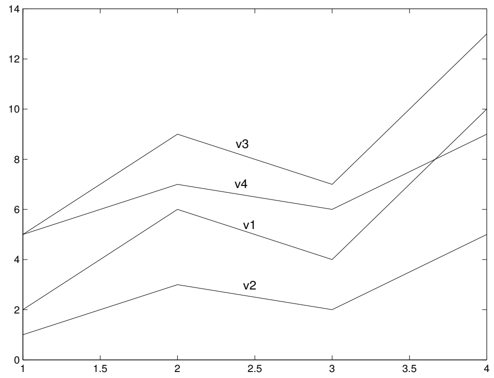


Figure 1. Four $1 - attribute$ **time sequences,** $v_1 = (2, 6, 4, 10)$**,** $v_2 = (1, 3, 2, 5)$**,** $v_3 = (5, 9, 7, 13)$**, and** $v_4 = (5, 7, 6, 9)$**.**

The Euclidean distance, D_2, between d-attribute time sequences u and v of length l is defined as

$$D_2(u, v) = \sqrt{\sum_{1 \le i \le l} \sum_{1 \le j \le d} (u_{i,j} - v_{i,j})^2}.$$

We first discuss the notion of shift and scale invariant distance between 1-attribute time sequences. Later, we extend the idea to the multi-attribute case.

2.1 1-attribute time sequences

Consider the 1-attribute time sequences v_1, v_2, v_3 and v_4 given in Figure 1. Although the Euclidean distance between any two of these sequences is large, these sequences can be made identical by scaling and shifting. For example $v_1 = 2 \times v_2$, $v_1 = v_3 - 3 \times N$, and $v_1 = 2 \times v_4 - 8 \times N$, where $N = (1, 1, ..., 1)$ is the normal vector.

Chu and Wong [6] addressed shift and scale invariant time series similarity for 1-attribute time sequences. If v is a 1-attribute time sequence of length l, the set of all possible scalings of v is defined as

$$SC(v) = \{c \times v | c \in \mathcal{R}\}.$$

The set of all scalings of v forms a line in l-dimensional space that passes through origin and v. The set of all possible shiftings of v is defined as

$$SH(v) = \{v + c \times N | c \in \mathcal{R}\}.$$

The set of all shiftings forms a line in l-dimensional space which passes through v and which is parallel to the vector N.

Chu and Wong [6] defined a combined transformation function $F_{a,b}(v) = av + bN$ for scaling and shifting, where $a, b \in \mathcal{R}$. They defined the distance between two 1-attribute time sequences u and v as

$$d(u,v) = min_{a,b \in \mathcal{R}}\{D_2(F_{a,b}(u), v)\}.$$

In other words, the distance $d(u,v)$ between u and v is defined as the minimum Euclidean distance between v and all possible shiftings and scalings of u. This is equivalent to the distance between $SC(u)$ and $SH(v)$. They defined the plane which passes through the origin and is perpendicular to the normal vector N as the *shift eliminated plane* (SE-plane). The projection of a sequence v onto the SE-plane is a point is defined as $TSE(v)$. The shifting line of any sequence is perpendicular to the shift eliminated plane. Therefore the projection of the shifting line of any sequence onto the SE-plane is a point. On the other hand, the projection of the scaling line of a sequence v onto the SE-plane is a line that passes through the origin and $TSE(v)$. The distance between $SC(u)$ and $SH(v)$ is equal to the distance between the projections of $SC(u)$ and $SH(v)$ onto the SE-plane. Using this idea, the authors propose to project the shifting lines of all the data sequences onto the SE-plane, and store these using an R*-tree index structure. Given any query sequence q, their method constructs the projection of $SC(q)$ on the SE-plane, and uses the transformed data points to obtain the necessary result set of a nearest neighbor or a range query. The following lemma is easily established.

Lemma 1 *Given two 1-attribute time sequences u and v of the same length, $d(u,v) =$*

$$||TSE(v)||_2 \times \sqrt{1 - (\frac{TSE(v).TSE(u)}{||TSE(v)||_2 \times ||TSE(u)||_2})^2},$$

where $u.v$ is the dot product of the vectors u and v, and $||v||_2$ is the second norm of v.

Proof:
The distance $d(u,v)$ is defined as the distance between the point TSE(v) and the line SC(TSE(u)). Since the line SC(TSE(v)) passes through the origin, the distance between TSE(v) and SC(TSE(u)) is equal to the distance of TSE(v) to the origin multiplied by the sine of the angle between the vectors TSE(v) and TSE(u). □

Though efficiently computable, the above distance formulation by Chu and Wong [6] is not symmetric. That $d(u,v) \neq d(v,u)$ for some u, v can be seen from the following counter examples.

- $u = (1,0)$ and $v = (1,1)$. In this case,

$$d(u,v) = D_2(F_{0,1}(u), v) = 0,$$
$$d(v,u) = D_2(F_{0.5,0}(v), u) = 0.5.$$

- $u = (0,0,1,1)$ and $v = (3,2,1,0)$. In this case,

$$d(u,v) = D_2(F_{-2,2.5}(u), v) = 1,$$
$$d(v,u) = D_2(F_{-0.5,1.25}(v), u) = 0.25.$$

The absence of symmetry in the definition of $d(u, v)$ can lead to counter-intuitive results. For example, consider two sequences u and v from the database. It is possible that u is in the result set when we perform a range query using v as the query sequence, but not vice-versa. It follows from Lemma 1 that the distance function $d(u, v)$ is symmetric, i.e. $d(u, v) = d(v, u)$, if and only if $||TSE(u)||_2 = ||TSE(v)||_2$. Since this condition is quite restrictive, we modify the definition of distance in order to make it symmetric.

Definition 1 *Given two 1-attribute time sequences u and v of the same length, the distance between these sequences is defined as*

$$dist(u, v) = min\{d(u, v), d(v, u)\}.$$

We could have use other functions such as max, or $average$ in the above definition, but it is the min function that captures the notion of distance more adequately. Note that like other shift and scale invariant distance functions in the literature [6, 13], this distance function is also not a metric. This is because it does not satisfy the triangle inequality. For example, consider the degenerate case when all the values of a time sequence x are 0. In this case both $dist(u, x)$ and $dist(x, v)$ are equal to zero ($u \times c = v \times c = x$ for $c = 0$), for all u and v. On the other hand, $dist(u, v)$ may be greater than zero, violating the triangle inequality. The only way to ensure triangle inequality seems to be to charge a cost for each transformation [15], but we do not do so in this paper. The absence of triangle inequality does not pose any problem for our search algorithms since we only need to compute the distance between a query point and a data region.

2.2 Multi-attribute time sequences

The attributes of a multi-attribute time sequence can be independent, i.e., allowing independent scalings and shiftings, or dependent. We term a time sequence in which the attributes are dependent a *type-1* sequence. If the attributes are independent, then we call this a *type-2* sequence. Examples of type-1 and type-2 time sequences are given in Figure 2. In this figure, m_1 represents the trajectory of an object moving in a circle on a plane. If this is a type-1 time sequence, then a scaling and shifting of this time sequence could result in a trajectory like m_2, which is a circular motion of a different radius, and shifted in X and Y coordinates by the same amount. One can imagine that the motions m_1 and m_2 are the same motions recorded at different zooms. However, if m_1 is a type-2 time sequence, then a motion such as m_3 may be a scaled and shifted version of m_1 as well. Notice that the two attributes of the sequence m_1 are scaled and shifted independently in m_3.

Next, we define the shifting and scaling transformations for type-1 and type-2 time sequences of k attributes. Let $N_{k,l}$ be the $k \times l$ matrix which is composed of all 1's. Define I_k to be the identity matrix of k dimensions. Let v be a k-attribute time sequence of length l.

Definition 2 *If v is a k-attribute type-1 time sequence of length l, then the set of all possible scalings of v is defined as*

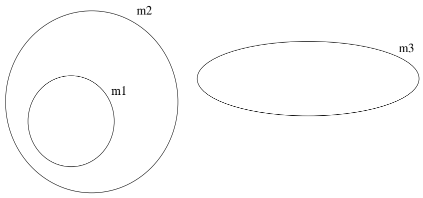


Figure 2. The trail of a moving object m_1 on a plane. If m_1 is type-1, then m_2 (but not m_3) is a transformed version of m_1. If m_1 is type-2, then both m_2 and m_3 are transformed versions of m_1.

$$SC(v) = \{c \times v | c \in \mathcal{R}\},$$

and the set of all possible shiftings of v is defined as

$$SH(v) = \{v + c \times N_{k,l} | c \in \mathcal{R}\}.$$

Definition 3 *If v is a k-attribute type-2 time sequence of length l, then the set of all possible scalings of v is defined as*

$$SC(v) = \{C_k I_k v | C_k \in \mathcal{R}^k\},$$

and the set of all possible shiftings of v is defined as

$$SH(v) = \{v + C_k I_k N_{k,l} | C_k \in \mathcal{R}^k\}.$$

Given the above definitions of scalings and shiftings of multi-attribute time sequences, the distance function given in Definition 1 can be used for both type-1 and type-2 multi-attribute time sequences. In our indexing scheme, we consider both type-1 and type-2 sequences. It is up to the user to choose the appropriate type of semantics for multi-attribute time sequences.

3 The CS-index structure

In this section, we propose a new index structure which clusters the data sequences according to both their scaling and shifting lines. We call this index structure *CS-index* (Cone Slice) for it resembles slices of hierarchically ordered cones. The idea is to project both the scaling lines and the shifting lines of the data sequences on to the SE-plane and cluster the sequences whose projected shifting lines are close and for whom the angles between the projected scaling lines are small. Consideration of both the shifting line and the scaling line of data sequences is prompted by Definition 1. If we consider only the shifting lines of the data sequences, and use an R*-tree [9, 3] index structure as proposed by Chu and Wong [6], then there are several disadvantages. 1)The angular distance between scaling lines of the data sequences clustered within a disk page may be larger than the angular distance between scaling lines of the data sequences in different pages. This may result in a large number of false hits. 2) Disk I/O's for reading the candidate sequences involve random seeks, resulting in a high overhead.

We will first describe the construction of the *CS-index* structure for the 1-attribute time sequences. Later we will extend the idea to the multi-attribute case.

3.1 CS-index for 1-attribute time sequences

Let q be a query sequence and let v be a time sequence in the database. Let $\theta_{q,u}$ be the angle between the vectors $TSE(q)$ and $TSE(u)$. Using Lemma 1 and Definition 1, we conclude that $dist(q,u) =$

$$min\{||TSE(q)||_2, ||TSE(u)||_2\} \times sin\theta_{q,u}.$$

This means that the distance between a query and a database sequence is based on the lengths of the projections of the two sequences on the SE-plane and the angle between the projections. Therefore, a good index structure must cluster radially, i.e., based on the distance from the origin as well as the angular distance.

The CS-index structure consists of a set of data pages stored on disk, and summary information stored in memory about the data pages. The data pages are organized in a tree structure defined by two parameters: page capacity, denoted p, and fanout, denoted f. The tree structure is virtual in the sense that it is used only for clustering of data; there are no physical index pages. The rationale for choosing the appropriate fanout will be explained later.

Algorithm CS-INDEX-BULKLOAD
Let S be the set of time sequences in the database. Let f be the fanout and p be the page capacity.
For all $v \in S$
    $v := TSE(v)$
SPLIT(S, f, p)

Figure 3. CS-index bulk-loading algorithm

The data sequences are bulk-loaded into the index structure as shown in Figures 3 and 4. First the shifting lines of all the sequences are projected onto the SE-plane (in procedure *CS-INDEX-BULKLOAD*). After that, procedure *SPLIT* is invoked with a specified value of fanout and page capacity. This procedure carries out a clustering based on angles and distances from origin. In angle-based clustering, one of the points on the SE-plane is chosen randomly as a pivot (Step 2). Using the angular distances from this pivot, the set of sequences is partitioned into f subsets $S_1, S_2, \ldots, S_f$ (Step 3 and 4). In the second splitting phase (Step 5), a piece of size p (Steps 5.a, 5.b, and 5.c). is chopped from each cone by intersecting it with a sphere centered at the origin. Each of these pieces is called a *slice*. The indexing method recursively splits the rest of the points (Step 5.d). This process goes on until all the data points are assigned to a page.

Algorithm SPLIT(S, f, p)

- if $|S| > p$
  1. if $|S| < p \times f$ then $f := \frac{|S|}{p}$
  2. v := a random sequence in S
  3. Sort all sequences in S in ascending order of angular distance from v
  4. Partition the sorted set into sets $S_1, S_2, ..., S_f$ of size $|S|/f$
  5. for i:= 1 to f
     (a) $P := p$ sequences closest to the origin in S_i
     (b) store P as the next page
     (c) $S_i := S_i - P$
     (d) $SPLIT(S_i, f, p)$
  6. end for
- end if

Figure 4. Split Algorithm. This algorithm recursively splits the data points first according to the angles and then according to their distances from the origin.

Each slice s can be viewed as the intersection of a cone with a ring that is determined by two radii, r_s and R_s, where r_s is the minimum Euclidean distance between the origin and a point on the slice and R_s is the maximum Euclidean distance between the origin and a point on the slice. An illustration of the CS-index structure when the SE-plane has 2 dimensions and $f = 3$ is shown in Figure 5. The index structure has three levels; the root level contains three slices 1, 14, and 27. There is a total of 39 slices, each containing at most p points. A slice s_i is said to be the *parent* of a slice s_j (and similarly, slice s_j is said to be a *child* of s_i) if the cone corresponding to s_j is contained entirely in the cone corresponding to s_i. For example, in Figure 5, slice 1 is a parent of slices $2 \ldots 13$, and slice 2 is a parent of slices 3, 4, and 5.

The pages resulting from the *SPLIT* procedure in Figure 4 are placed on the disk linearly in the order of depth-first-search. For example, the numbers on the pages in Figure 5 represent the ordering of the pages for this index structure. As will be seen later, this placement minimizes the number of non-candidate pages that are read.

The distance between a query sequence q and a slice is defined as the minimum scale and shift invariant distance between q and any point on the slice. The formal definition is as follows:

Definition 4 *Let s be a slice in the CS-index and q be a query sequence on the SE-plane. Let $\theta_{q,s}$ be the angle between q and a point in s for which $sin\theta_{q,s}$ is the minimum. The distance between q and s is defined as:*

$$dist(q, s) = min\{||q||_2, r_s\} \times sin\theta_{q,s}.$$

Using this definition, we have the following theorems:

Theorem 1 *Let s be a slice in the CS-index, v be a data sequence contained in s, and q be a query sequence on the SE-plane, then*

$$dist(q, s) \leq dist(q, v).$$

Proof:

Since v is contained in s, $r_s \leq ||v||_2$. Similarly, $sin\theta_{q,s} \leq sin\theta_{q,v}$. Therefore, $dist(q, s) \leq dist(q, v)$. □

Theorem 2 *Let s_1 and s_2 be two slices in the CS-index such that s_1 is the parent of s_2. Let q be a query sequence on the SE-plane, then*

$$dist(q, s_1) \leq dist(q, s_2).$$

Proof:

Since s_1 is a parent of s_2, $r_{s_1} < r_{s_2}$. Similarly, $sin\theta_{q,s_1} \leq sin\theta_{q,s_2}$. Therefore, $dist(q, s_1) \leq dist(q, s_2)$. □

Some important observations about the CS-index are as follows. 1) If the distance between a slice and a query sequence is greater than ϵ (Theorem 1), then the distance between the query sequence and any data sequence contained in that slice is greater than ϵ. In other words, given a range query, a slice may contain candidate sequences only if the distance between the query sequence and that slice is less than the search range. 2) If the distance between a slice and a query sequence is greater than ϵ, then the distance between that query sequence and all children of that slice is greater than ϵ (Theorem 2). 3) If the distance between a slice and a query sequence is less than ϵ, then the distance between that query sequence and all the parents of that slice is less than ϵ (Theorem 2).

The range query algorithm on the CS-index is presented in Figure 6. A range query is performed in two phases: an in-memory candidate generating phase followed by a disk-based post-processing step. In the in-memory phase, the search starts from the root pages and proceeds downwards. If the distance between the query sequence and a slice is less than the query range (Step 4.b), then the method marks this slice as a candidate and expands the query to all the children of that slice. If the distance between the query sequence and a slice is greater than the search range, then the algorithm prunes that slice and all of its children. Once the candidate sets are determined, the disk-based processing step begins. Using the disk placement information, a sequential scan is used to read all the candidate pages. For example if the candidate set in Figure 5 is {1, 2, 5, 6, 7}, then we read pages 1, 2, 3, 4, 5, 6, and 7 using sequential scan. Note that pages 3 and 4 are non-candidate pages that are being read. Once the actual data points from the candidate pages have been read, points not in the range are pruned to ensure no false

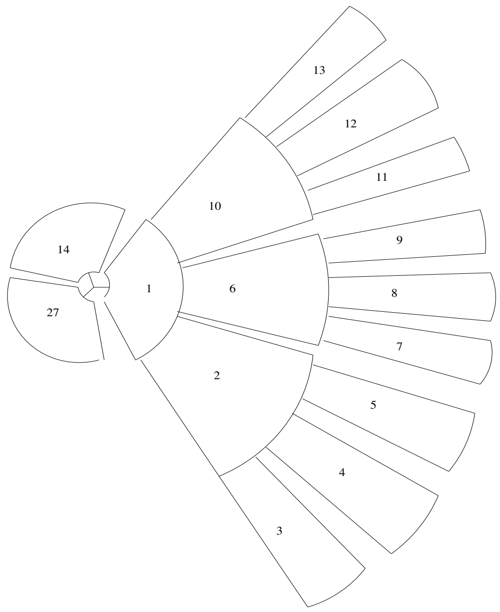


Figure 5. An illustration of the CS-index for 2 dimensions when f = 3.

retrievals. Since the set of pages read from the disk during the post-processing step is a subset of all the pages, our method performs no worse than sequential scan. In fact, it performs much better since clustering reduces the range of the sequential scan considerably. As a consequence of Theorem 1 and Theorem 2, we have the following corollary:

Corollary 1 *The range query algorithm in Figure 6 does not incur any false drops.*

Figure 7 presents the NN-search for the CS-index structure. A nearest neighbor query follows the standard *MINDIST* and *MINMAXDIST* based pruning [10, 17]. There is an in-memory candidate generation phase, and a disk-based post-processing phase as in range query. In the in-memory phase, the method chooses one of the root slices as the candidate and computes the *MINDIST* and the *MINMAXDIST* between the query sequence and the slice. For all other slices:

- if the *MINDIST* between that slice and the query sequence is greater than the current *MINMAXDIST*, then the method prunes that slice and all of its children (Step 6.b).
- if the *MINMAXDIST* between that slice and the query sequence is less than the current *MINDIST*, then the method removes all the slices from the candidate set and inserts the new slice to the candidate set (Step 6.c).
- else, the method inserts the new slice to the candidate set and pushes the children of the slice on to the stack (Step 6.d).

After this candidate generation phase, the disk-based post-processing phase starts. As in range queries, the disk placement information is used to read all the candidate pages, and find the nearest neighbor.

Algorithm RANGE-QUERY(q, ϵ)

1. $C := \emptyset$ /* the set of candidate pages*/
2. $S := \emptyset$ /* stack for navigation*/
3. Push all the root slices s_1, s_2, ..., s_f on to S.
4. While $S \neq \emptyset$
   (a) $s := Pop(S)$
   (b) if $dist(q, s) \leq \epsilon$
      i. $C := C \cup \{s\}$
      ii. Push all the children of s on to S
5. Read the candidate pages in set C and perform postprocessing to eliminate false retrievals.

Figure 6. Range search algorithm

3.2 Multi-attribute CS-index

In Section 3.1, we explained the CS-index structure, and range queries and nearest neighbor queries for 1-attribute time sequences. In this section, we will explain how these ideas can be extended to handle k-attribute time sequences.

If the multi-attribute time sequences in the database are k-attribute type-1 time sequences of length l (the k attributes are dependent), then the problem reduces to the 1-attribute case by simply considering the $k \times l$ time sequences as $1 \times kl$ time sequences. In other words, the query and the database sequences of length l are transformed to 1-attribute sequences of length $k \times l$. This is justified because all the entries are scaled and shifted by the same amount.

If the database consists of k-attribute type-2 time sequences of length l, then all the attributes must be considered separately. This is because different attributes may be scaled or shifted by different amounts. We first split all the time sequences into k 1-attribute time sequences of length l. This is like splitting a $k \times l$ dimensional space into k different l-dimensional spaces. As a result, each k-attribute time sequence corresponds to k points in non-overlapping l-dimensional spaces. We determine the SE-planes of these l-dimensional sub-spaces. We project the 1-attribute time sequences onto their corresponding SE-planes, and concatenate the vectors corresponding to these k projections to construct a 1-attribute $k \times (l-1)$ dimensional point. We construct the CS-index on these points as described in Section 3.1. Every slice of the constructed index can be projected into k different spaces; these projections are called *subslices*. (Note that another choice would be to maintain k separate CS-index structures for

Algorithm NEAREST-NEIGHBOR-QUERY(q)

1. $CURRENT\_MINDIST := CURRENT\_MINMAXDIST := \infty$
2. $C := S := \emptyset$ /* the set of candidate pages and stack for navigation*/
3. Push all the root slices s_1, s_2, ..., s_f on to S.
4. While $S \neq \emptyset$
   (a) $s := Pop(S)$
   (b) if $MINDIST(q, s) > CURRENT\_MINMAXDIST$
      i. Continue /*prune s and all its children*/
   (c) else if $MINMAXDIST(q, s) < CURRENT\_MINDIST$
      i. $C := \{s\}$ /*s is the only candidate slice*/
      ii. Update $CURRENT\_MINDIST$ and $CURRENT\_MINMAXDIST$
   (d) else
      i. $C := C \cup \{s\}$
      ii. push all children of s on to S; update $CURRENT\_MINMAXDIST$ and $CURRENT\_MINMAXDIST$.
5. Read the candidate pages in set C and perform postprocessing to eliminate false retrievals.

Figure 7. Nearest Neighbor search algorithm

1-attribute $(l-1)$ dimensional points. But, this would require additional post-processing.)

For a given range query or a nearest neighbor query of k attributes, we split the query into k 1-attribute subqueries, one for each attribute. For each subquery, we obtain its distance to a slice by considering the subslice corresponding to that attribute. The distance between a query and a slice is defined to be the square root of the sum of squares of the k different subquery subslice distances. Once these distances are obtained, pruning and post-processing proceeds as in the single-attribute case.

3.3 Improving post-processing performance

The performance of our method can be improved by decreasing the amount of redundant disk reads. This can be achieved by three techniques: carrying out a more effective pruning in the in-memory phase, optimizing the placement of pages on disk, and caching/replication of disk pages. We discuss each of these three techniques next.

We noted earlier that the fanout f is an independent parameter of the bulk-loading algorithm that is not affected by the size of disk pages. For a given dataset, a large fanout leads to thick slices that span a smaller angle, whereas a small fanout leads to thin slices that span a larger angle. The success of the pruning procedure depends on both the thickness and the angular span of the slices: a thick or a wide slice is less likely to be pruned. The right choice of fanout ensures that slices are not too thick and not too wide. This can be determined either experimentally or theoretically if the data distribution is known.

The second parameter that improves the post-processing performance is the placement of pages on disk. Note that if the distance between a query sequence q and a slice s is less than the given search range ϵ, then the distance between q and the parent of s is also less than ϵ. Therefore, if a slice s is in the candidate set, then all its ancestors are also in the candidate set. For example, if slice 4 of the CS-index in Figure 5 is a candidate, then the slices 2 and 1 are also guaranteed to be in the candidate set. In this case, we read all the pages from 1 to 4. Note that, slice 3 may not be in the candidate set and may be read redundantly. On the other hand, by reading some redundant pages, the technique prevents from multiple seek costs in the disk read phase. If the candidate pages are located closely on the disk, then one sequential read has much lower cost than multiple random disk reads. The placement order that preserves the parent-child relationship is the depth-first-search order. Accordingly, we use it to place the pages on disk.

The final parameter that improves the post-processing performance is the degree of caching and replication of disk pages. Both caching and replication can reduce the number of redundant pages that are read. Replicating a page means that we maintain a copy of the page with all of its subtrees on the disk. Replicating k levels of the index structure means that we replicate the pages at the first k levels of the CS-index. For example, if one level of the CS-index at Figure 5 is replicated, then the pages 1, 14, and 27 are replicated with their subtrees. This results in the following page order on disk: 1, 2, 3, 4, 5, 1, 6, 7, 8, 9, 1, 10, 11, 12, 13, $\ldots$. If 2 levels are replicated, then the page order on disk is 1, 2, 3, 1, 2, 4, 1, 2, 5, 1, 6, 7, 1, 6, 8, 1, 6, 9, $\ldots$. If pages 1, 10, and 13 are in the candidate set for a given query, then without replication we will read all pages from 1 to 13, leading to 10 redundant reads. With one level of replication, the number of redundant pages read is reduced to 2 (pages 11 and 12), and with two levels of replication, there are no redundant pages read.

Though replication can reduce the gap between a parent and a child, it increases the gap between pages at the same level. For example, if pages 3, 4, and 5 are in the candidate set then we do not access any redundant pages without replication, but we access pages 1 and 2 twice with two levels of replication. In general, a very high degree of replication may decrease the performance. This will be borne out by our experimental results.

Caching c levels of the index structure means to keep the pages at the first c levels of the CS-index in the memory buffer. In this case, the pages starting from the $(c+1)^{th}$ level are stored on the disk in depth-first-search order. For example, if one level of the CS-index in Figure 5 is cached, then pages 1, 14, and 27 are kept in the buffer. Unlike replication, caching can *only* improve the performance of our index structure.

4 Experimental results

We carried out several experiments to test the performance of our transformation technique and the CS-index structure. We used four different datasets in our experiments: the first dataset consists of a single attribute, the second and the third datasets consist of 2 attributes, and the fourth dataset consists of 4 separate datasets of 1, 2, 4, and 8 attributes.

- The first dataset is composed of stock market data taken from *chart.yahoo.com*. This dataset contains the closing prices of companies for 32 consecutive days. This is a 1-attribute dataset and contains 18000 time sequences.
- The second dataset is also a stock market dataset, obtained again from *chart.yahoo.com*. The time sequences in this dataset consist of 2 attributes. The first attribute is the closing price, and the second attribute is the volume. There are 20000 time sequences of length 32 in this dataset.
- The third dataset is obtained synthetically by considering four different kinds of object trails in a 2-attribute sequence. The motions that we consider are: bouncing ball, circular motion, billiard ball moving within the confines of a rectangular table with perfect carom and elasticity, and a periodic motion along a sine curve. This dataset contains 2^{15} time sequences, distributed evenly among the four different motion types. The length of the time sequences is 32.
- The fourth dataset is actually 4 different datasets each of them corresponds to a different number of attributes (1, 2, 4, and 8 attributes). Each of these datasets are obtained synthetically by adding four sine signals of random frequencies and amplitudes and some amount of random noise. Each dataset is composed of 2^{15} time sequences of length 32. A similar dataset has been used in [12] for 1-attribute time sequences.

We compressed the time sequences in the four datasets to 2 and 4 dimensions using DFT, and then built the CS-index structure on the compressed data. Since both DFT and TSE are distance preserving transformations, there are no false drops in the resulting index structure. The multi-attribute time sequences in our database are considered to be of type-2 (i.e. independent attributes). Since type-1 multi-attribute time sequences are equivalent to 1-attribute time sequences we do not present separate experiment results for them. In our query model, we considered range queries with 20 different values of ϵ in the range $(0.001, 0.01)$. For every value of ϵ, 1000 sequences in the database were chosen at random for querying. We assume that the page size is 4K in our experiments.

The first experiment considers the 1-attribute stock market dataset, and compares the performance of the CS-index to sequential scan and the R-Tree index as proposed by Chu and Wong [6]. Since the CS-index structure uses bulk loading, we used the VAM-Split implementation of R-Trees [20] that also uses bulk loading. We compressed the time sequences in the dataset to 4 dimensions using DFT for both. Figure 8 shows the number of disk reads for sequential scan, and the two index structures. For the CS-index structure, we also present results for replicating

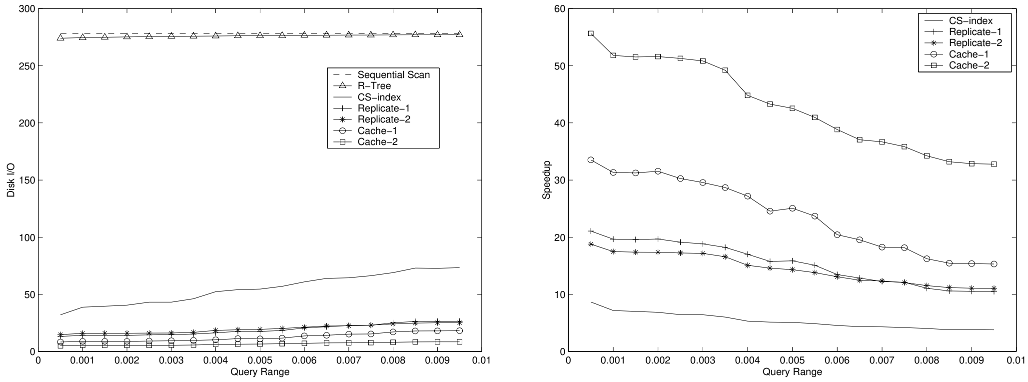


Figure 8. Number of disk reads and speedup of our method versus R-Tree and sequential scan for the 1-attribute stock market dataset compressed to 4 dimensions.

one level, replicating two levels, caching one level, and caching two levels. The results show that the R-Tree index structure accesses almost all the pages. This can be explained as follows. The angular distance between two points may be large even if the Euclidean distance between them is small. Since R-Tree index structure clusters based on Euclidean distance, the resulting boxes may span a large angular distance. This may lead to a large number of candidate MBRs. Though the R-tree index and sequential scan access the same number of pages, the R-tree index accesses the pages randomly. Since a random read can cost up to 10 times a sequential read [4, 18, 11], the performance of the R-tree index is about 10 times worse than sequential scan. Therefore, we conclude that the R-Tree index structure is not appropriate for scale and shift invariant searches. On the other hand, the CS-index structure accesses less than 20% of all the pages. Since, the CS-index structure performs a single sequential disk read, it incurs seek cost only once (like sequential scan). For this dataset, the CS-index is 5 to 10 times faster than sequential scan. Replicating only one one level of the CS-index structure doubles the speedup of our method. Replicating more than two levels degrades the performance. If only one level of the CS-index is cached, then the speedup of our method is 15 to 35 for 4 dimensions. The speedup of our method more than doubles if one more level is cached.

In remaining experiments, we compare the performance of our method to sequential scan. R-Tree results are not presented, because its performance was much worse than sequential scan. Figures 9 and 10 show the experimental results for the 2-attribute datasets: the stock market dataset and the motion dataset. We obtained speedups up to 8 for CS-index with no replication and caching. The speedups doubled when one level is replicated. The speedups increased up to 55 when 2 levels of our index structure are cached.

The next experiment reports the scalability of the CS-index structure for an increasing number of attributes. In this experiment, we generate sine curve datasets of 1, 2, 4, and 8 attributes. Each of these datasets contains 2^{15} sequences of length 32. We run 20000 random range queries as in previous experiments on each of these datasets.

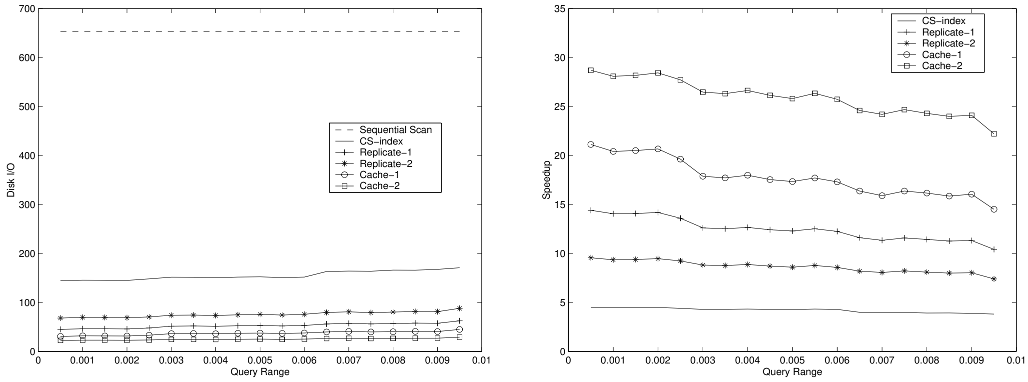


Figure 9. Number of disk reads and speedup of our method versus sequential scan for the 2-attribute stock market dataset compressed to 2 dimensions.

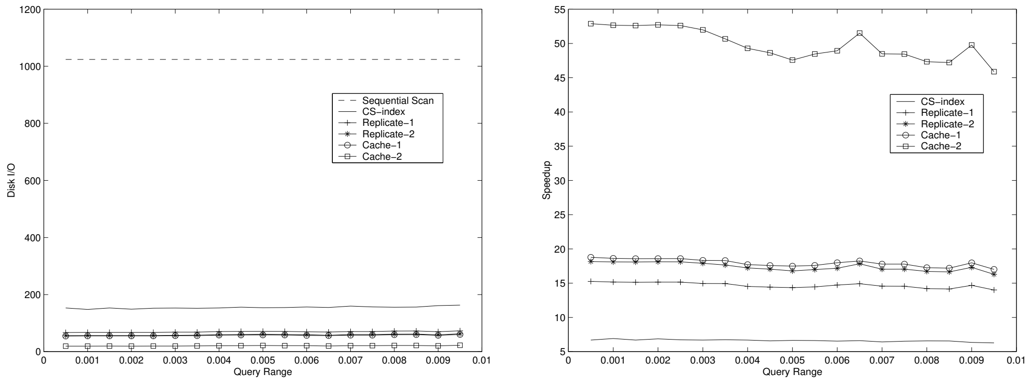


Figure 10. Number of disk reads and speedup of our method versus sequential scan for the 2-attribute motion dataset compressed to 2 dimensions.

Figure 11 plots the number of disk reads for sequential scan and CS-index structure for no replication and caching, replicating 1 and 2 levels, and caching 1 and 2 levels. As the number of attributes increases, the size of the dataset increases linearly. This is reflected in the number of disk reads for sequential scan. The experimental results show that the number of disk reads for the CS-index structure increases linearly with the number of attributes. Therefore, the speedup of the CS-index structure remains invariant under an increasing number of attributes.

Figure 12(a) plots the number of disk read for the CS-index structure (without replication and caching) for different values of fanout for the 2-attribute stock market data set compressed to 2 dimensions. The CS-index performs the best when the fanout is 7 for this data set. We obtained similar *u-curved* graph for the other datasets too. The optimal fanout for other datasets vary between 5 and 7. The number of disk reads increases for low and high values of fanout. This can be explained as follows. For small values of fanout, the slices span a large

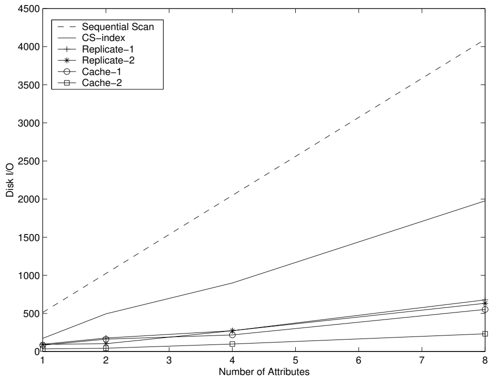
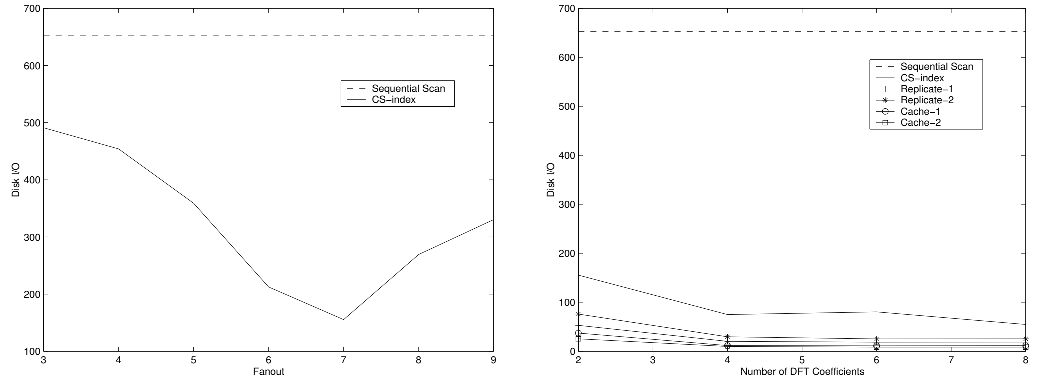


Figure 11. Number of disk reads of our method versus sequential scan for the sine curve dataset of various number of attributes compressed to 2 dimensions.

angle. Therefore, the expected angular distance between a query sequence and the slices decreases. Hence, more slices intersects with the scaling line of the query sequence, and the number of candidate slices increases. For large values of fanout, the Euclidean distance span of a slice increases. Since the distance between a slice and a query sequence is computed using the minimum Euclidean distance between the slice and the origin (as well as the angular distance), the number of candidate slices increases for higher values of fanout.

The last experiment reports the number of disk reads of the CS-index structure for different number of DFT coefficients. In this experiment we compressed the 2-attribute stock market data set to 2, 4, 6, and 8 dimensions and run 1000 random range queries for 20 different values ϵ in (0.001, 0.01), and found the average number of disk reads for these experiments. The results are plotted in Figure 12(b). As it can be seen from this figure, the performance of the CS-index structure increases as the number of DFT coefficients stored increases. However, the memory requirements increase as more coefficients are used. For the case of 2-4 coefficients used in most of our experiments, the size of the index structure is about $2-4\%$ of the size of the dataset.

We can summarize the important aspects of the experimental results as follows. 1) In all the measurements, our method performs better than sequential scan and R-Tree, even when no replication and caching is used. 2) R-Tree is not a suitable index structure for shift and scale invariant search applications. 3) The experimental results of the CS-index structure for real datasets are similar to that for synthetic datasets. This means that, the CS-index structure performs equally well for clustered and unclustered data. 4) The number of disk reads for the CS-index structure increases linearly with the number of attributes. As a result, the speedup over sequential scan is invariant. 5) Replicating the first few levels of the CS-index structure increases the performance, but further replication degrades the performance. 6) Caching one more level more than doubles the performance.



(a) (b)

Figure 12. Number of disk reads of our method versus sequential scan for the 2-attribute sine curve dataset for (a) various fanout values compressed to 2 dimensions (b) various number of DFT Coefficients.

5 Subsequence matching

The CS-index structure is an efficient index structure for shift and scale invariant search of multi-attribute time sequences for *whole matching*, i.e., when the length of the query sequence and the data sequences are the same. However, for some applications the query sequence is is shorter than data sequences, and the data sequences may have different lengths. In this case, a range or a nearest neighbor query searches all the *subsequences* of all the data sequences in the database. This kind of search is called *subsequence search*. Furthermore, based on the application, the database may be dynamic in two ways. First, the length of the sequences may change in time. Second, new time sequences may be inserted to the database. In this section, we use ideas of the CS-index structure and MR-index structure [11] to construct a dynamic index structure that can answer queries of arbitrary lengths.

5.1 Multi-Resolution CS-index structure

Let s be the longest time sequence in the database, where $2^b \leq |s| < 2^{b+1}$ for some integer b. Similarly, let the minimum possible length for a query be 2^a, for some integer a where $a \leq b$. Let s_1, s_2, ..., s_n be the time sequences in the database as shown in Figure 13. Our index structure stores a grid of trees $T_{i,j}$, where i ranges from a to b, and j ranges from 1 to n. Tree $T_{i,j}$ is the set of slices for the j^{th} sequence corresponding to window size 2^i. Figure 13 shows a layout of this index structure.

In order to obtain $T_{i,j}$, we slide a window of length 2^i on sequence s_j, starting from the leftmost point of s_j. We transform each sequence of length 2^i using DFT or wavelets, and choose a few coefficients from the transformation. Later, we map the transformed point onto the shift eliminated plane using TSE transformation. Note that, each transformed sequence corresponds to a point on the TSE plane. We begin with the first point

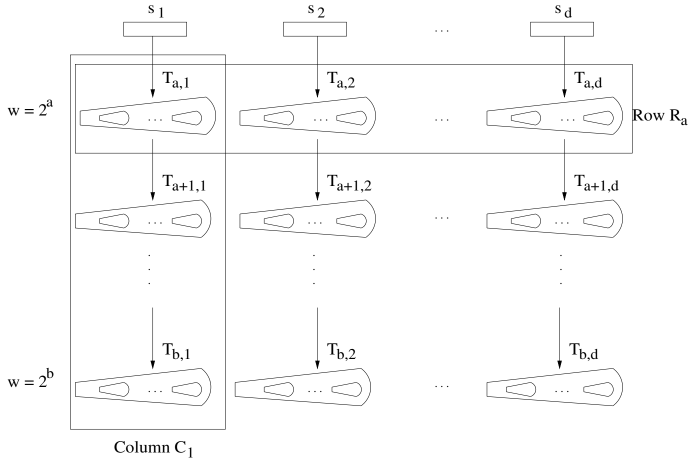


Figure 13. Layout of the MRCS index structure

and find the minimum slice, that covers this point. This slice is later extended to cover the first c subsequences, where c is the page capacity. After the first c subsequences are transformed, a new slice is created to cover the next c subsequences. This process continues until all subsequences of length 2^i are transformed. Since the index structure stores frequencies at different resolutions, we call this index structure the *Multi Resolution CS-Index Structure* (MRCS). Note that, since the MRCS-index has a grid structure, we do not have a fanout parameter for it.

The i^{th} row of the MRCS-index structure is represented by R_i, where $R_i = \{T_{i,1}, ..., T_{i,n}\}$ corresponds to the set of all trees at resolution 2^i. Similarly, the j^{th} column of our index structure is represented by C_j, where $C_j = \{T_{a,j}, ..., T_{b,j}\}$ corresponds to the set of all trees for the j^{th} time sequence in the database.

The MRCS index structure is dynamic because a row (a data sequence) can be inserted independently. Columns can also be inserted independently; this corresponds to an increase in the size of sequence. Moreover, different rows of the index structure can have different lengths.

5.2 Search algorithm

The search technique for the MRCS-index structure partitions a given query sequence of arbitrary length into a number of subqueries at various resolutions available in our index structure. Later, it performs a *partial range query* for each of these subqueries on the corresponding row of the index structure. This is called a partial range query, because it only computes distance of the query subsequence to the slices, not the distance to the actual subsequences contained in the slices.

Given any query q of length $k2^a$ and a range ϵ, there is a unique partitioning[1], $q = q_1 q_2 ... q_t$, with $|q_i| = 2^{c_i}$ and $a \leq c_1 < ... < c_i \leq c_{i+1} \leq ... c_t \leq b$. This partitioning corresponds to the 1's in the binary representation of k.

[1] If the length of the query sequence is not a multiple of the minimum window size, then the longest prefix of the query whose length is a multiple of the minimum window size can be used.

We first perform a search using q_1 on row R_{c_1} of the index structure. As a result of this search, we obtain a set of slices that lie within a distance of ϵ from q_1. Using the distances to these slices, we refine the value of ϵ and make a second query using q_2 on row R_{c_2} and the new value of ϵ. This process continues for the remaining rows R_{c_3} ... R_{c_t}. The idea of the above radius refinement is captured in the following lemma.

Lemma 2 *Let q be a given query sequence and x be an arbitrary subsequence of the same length in the database. Let q_1 be a prefix of q and q_2 be the rest of x. Similarly, let x_1 be a prefix of x and x_2 be the rest of x, such that $|q_1| = |x_1|$. If $d(q,x) \leq \epsilon$ then*

$$dist(q_2, x_2) \leq max_{S \in \mathcal{S}}(\sqrt{\epsilon^2 - dist(q_1, S)^2}),$$

where S is a slice that covers x_1 and $\mathcal{S}$ is any arbitrary set of slices that contains S.

Proof:
Since $dist(q, x) = \sqrt{dist(q_1, x_1)^2 + dist(q_2, x_2)^2} \leq \epsilon$, we have

$$dist(q_2, x_2) \leq \sqrt{\epsilon^2 - dist(q_1, x_1)^2}.$$

Since $dist(q_2, x_2) \geq dist(q_2, S)$,

$$dist(q_2, x_2) \leq \sqrt{\epsilon^2 - dist(q_1, S)^2}).$$

From this, it follows that
$dist(q_2, x_2) \leq max_{S \in \mathcal{S}}(\sqrt{\epsilon^2 - dist(q_1, S)^2}).$ □

An example partitioning of a query sequence is given in Figure 14. In this example, the length of the query sequence is 208 and the minimal query length is 16. The query is split into 3 pieces of length 16, 64 and 128. Three subqueries are performed, one for each partition. The first subquery q_1 is performed on row R_4 with ϵ as the radius. As a result of this search, ϵ is refined to ϵ'. The next subquery q_2 is performed on row R_6 with the smaller radius of ϵ'. As a result of this subquery, ϵ' is refined further to ϵ''. Finally, subquery q_3 is performed on row R_7 with radius ϵ''. The resulting set of slices is then processed to find the actual subsequences using disk I/Os.

Figure 15 presents the complete search algorithm. Step 1 partitions the query q into separate pieces corresponding to a subset of the rows R_{c_1}, R_{c_2}, ..., R_{c_t} of the index structure. In Step 2, these rows are searched from top to bottom independently for each sequence (column) in the database. At every row, a partial range query (Step $2b(i)$), and then a range refinement (Step $2b(ii)$) are carried out. When all rows have been searched, the disk pages corresponding to the last result set are read (Step $2c$). Finally post-processing is carried out to eliminate false retrievals (Step $2d$).

We ran an experiment to compare the performance of the MRCS-index structure to sequential scan. In this experiment, we used a 1-attribute stock market data set obtained from *chart.yahoo.com*. This data set contains

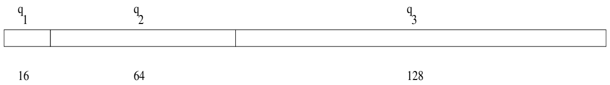


Figure 14. Partitioning for query q, $|q| = 208$

Algorithm RANGE-QUERY(q, ϵ)

1. Partition the query q into t parts as q_1, q_2, ..., q_t such that $|q_i| = 2^{c_i}$ and $a \leq c_1 < c_2 < ... < c_t \leq b$.

2. For $j := 1$ to n

   (a) $\epsilon_{c_1,j} := \epsilon$

   (b) For $i := 1$ to t

      i. Perform range query on $T_{c_i,j}$ using $\epsilon_{c_i,j}$. Let $Res_{c_i,j}$ be the resulting set of slices whose distances to q_i are less than $\epsilon_{c_i,j}$.

      ii. $\epsilon_{c_{i+1},j} :=$
      $max_{S \in Res_{c_i,j}}\{\sqrt{\epsilon^2_{c_i,j} - dist(q_i, S)^2}\}$

   (c) Read disk pages corresponding to $Res_{c_t,j}$

   (d) Perform post processing to eliminate false retrievals.

Figure 15. Subsequence search algorithm.

456 sequences of length 2^{10}. We constructed resolutions 16, 32, 64, 128, 256, and 512. We compressed the subsequences of the data sequences to 2 dimensions usinf DFT. We performed 1000 arbitrary length random range queries for 10 different values of ϵ in (0.001, 0.01). Figure 16 plots the average number of disk reads for this experiment. The number of disk reads for the MRCS-index structure is about one third of that for sequential scan.

6 Discussion

In this paper we considered the problem of shift and scale invariant search for multi-attribute time sequences. We discussed the existing solution to this problem for 1-attribute time sequences. We showed that the existing distance function is not symmetric, and can lead to counter-intuitive results. We defined a new symmetric distance function that allows shift and scale transformations for both the time sequences being compared. We extended this definition to the multi-attribute case.

We proposed a new index structure called CS-index that clusters the time sequences according to their scalings and shiftings. This index structure recursively splits the search space into hierarchical cones and selects a slice of each cone as a disk page. This index structure allows early pruning in the search phase. Later, we considered techniques to improve the performance of the index structure: depth-first-search order based placement on disk,

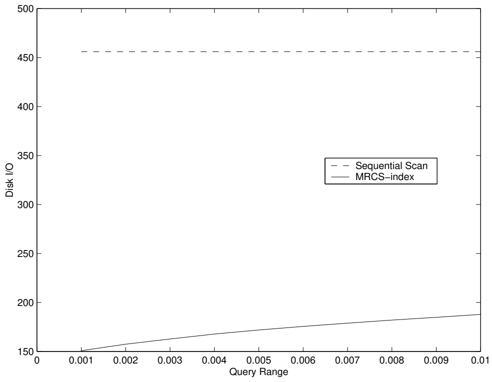


Figure 16. Number of disk reads of the multi resolution index structure versus sequential scan for subsequence matching problem for the 1-attribute stock market dataset compressed to 4 dimensions.

choice of right fanout, and caching/replication. Finally, we extended the method to handle multi-attribute time sequences.

According to experimental results with both real and synthetic datasets, our method works 2 to 10 times faster than sequential scan. We also evaluated the effects of replicating higher levels of the index structure. Replicating only the root level of the CS-index almost doubled the performance of our method. Further replication eventually degraded the performance. We also experimented with caching. According to our experiments, if only the pages at the root level are cached, our method works 5 to 40 times faster than sequential scan. We obtained speedups up to 60 when we cached one more level.

Based on the idea of using multiple resolution levels in [11], we proposed another index structure called MRCS-index that clusters the subsequences for different resolutions in a grid of cone slices. The MRCS-index structure is a dynamic index structure and it allows subsequence search for queries of arbitrary length. According to our experiments, the MRCS-index structure works 3 times faster than sequential scan.

Multi-attribute time sequences are an important emerging class of applications. They arise ubiquitously and naturally: in medical applications, in control applications, in video and event sequences, and in history-based applications such as the stock market. The ability to query such data without consideration for scaling and shifting is a neccesity for understanding and analyzing the characteristics of such datasets. The algorithms presented in this paper are an important first step toward this, and should be widely applicable.

In the future we would like to experiment with video datasets in which the motion of an object can be tracked through the use of motion vectors. It will also be interesting to consider further classification of multi-attribute time sequences, rather than just type-1 and type-2 sequences. Supposing different search algorithms in the context of a real database is also planned for future.

References

[1] R. Agrawal, C. Faloutsos, and A. Swami. Efficient similarity search in sequence databases. In *FODO*, Evanston, Illinois, October 1993.

[2] R. Agrawal, K. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *VLDB*, Zurich, Switzerland, September 1995.

[3] N. Beckmann, H. P.Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD*, pages 322–331, Atlantic City, NJ, 1990.

[4] S. Bernhard, L. Per-ke, and M. Ron. Reading a set of disk pages. In *VLDB*, pages 592–603, 1993.

[5] Kin-Pong Chan and Ada Wai-Chee Fu. Efficient time series matching by wavelets. In *ICDE*, 1999.

[6] K. K. W. Chu and M. H. Wong. Fast time-series searching with scaling and shifting. In *PODS*, Philedelphia, PA, 1999.

[7] S. Dagtas, W. Al-Khatip, A. Ghafoor, and R. L. Kashyap. Models for motion-based video indexing and retrieval. *IEEE Transactions on Image Processing*, 9:88–101, January 2000.

[8] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, pages 419–429, Minneapolis MN, May 1994.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[10] G. R. Hjaltason and H. Samet. Ranking in spatial databases. In *Proceedings of the* 4^{th} *Symposium on Spatial Databases*, pages 83–95, Portland, Maine, August 1995.

[11] T. Kahveci and A. Singh. Variable length queries for time series data. In *ICDE*, Heidelberg, Germany, 2001.

[12] E. J. Keogh and M. J. Pazzani. An indexing scheme for similarity search in large time series databases. In *SSDBM*, Cleveland, Ohio, 1999.

[13] Seok-Lyong Lee, Seok-Ju Chun, Deok-Hwan Kim, Ju-Hong Lee, and Chin-Wan Chung. Similarity search for multidimensional data sequences. In *ICDE*, San Diego, CA, 2000.

[14] Chang-Shing Perng, H. Wang, S. R. Zhang, and D. S. Parker. Landmarks: a new model for similarity-based pattern querying in time series databases. In *ICDE*, San Diego, USA, 2000.

[15] D. Rafiei and A. O. Mendelzon. Similarity-based queries for time series data. In *SIGMOD*, pages 13–25, Tucson, AZ, 1997.

[16] D. Rafiei and A. O. Mendelzon. Efficient retrieval of similar time sequences using dft. In *FODO*, Kobe, Japan, 1998.

[17] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *ACM SIGMOD*, San Jose, CA, 1995.

[18] B. Seeger. An analysis of schedules for performing multi-page requests. *Information Systems*, 21(5):387–407, 1996.

[19] C. Shahabi, X. Tian, and W. Zhao. TSA-tree: A wavelet-based approach to improve the efficieny of multi-level surprise and trend queries. In *SSDBM*, 2000.

[20] D. White and R. Jain. Similarity indexing: Algorithms and performance. In *SPIE Storage and Retrieval for Image and Video Databases*, 1996.