Variable Length Queries for Time Series Data

Tamer Kahveci Ambuj Singh
Department of Computer Science
University of California
Santa Barbara, CA 93106
{tamer,ambuj}@cs.ucsb.edu

Abstract

Finding similar patterns in a time sequence is a well-known problem that has been addressed by many authors. Most of the current techniques work well for queries of a prespecified length, but fail for variable length queries. We propose a new indexing technique that works well for variable length queries. Our idea is to store index structures at different resolutions for a given dataset. The resolutions are based on wavelets. A number of subqueries at different resolutions are generated for each variable length query. The ranges of the subqueries are progressively refined based on results from previous subqueries. Our experiments show that the total cost for our method is 4 to 20 times less than the current techniques including Linear Scan. Because of the need to store information at multiple resolution levels, the storage requirement of our method could potentially be large. In the second part of the paper, we show how the index information can be compressed with minimal information loss. According to our experimental results, even after compressing the size of the index to one fifth, the total cost of our method is 3 to 15 times less than the current techniques.

1 Introduction

Time series or sequence data arises naturally in many real world applications like stock market, weather forecasts, video databases and medical data. 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, or finding similar patterns in medical sensor data. The presence of a large number of long time sequences in the database leads to a larger number of potential result subsequences making this problem computationally and I/O intensive.

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 [2, 5, 7, 14, 18, 19]. However, Euclidean distance by itself may be insufficient to describe the similarity. For example, if one time series is a constant multiple of the other, or is its shifted form, they can still be similar depending on the application. Chu et al. [6] defined the *Shift Eliminated Transformation*. This transformation maps the sequences onto the *Shift Eliminated Plane*, such that the Euclidean distance is computed as the minimum value after all possible scalings and shiftings of the two sequences. Such shift and scale elimination can be easily applied to any subsequence searching algorithm based on Euclidean distance.

Non-Euclidean metrics have also been used to compute the similarity for time sequences. Agrawal et al. [1] use L_∞ as the distance metric. Landmark model [13] chooses only a subset of values from a time sequence, which are peak points and uses them to represent the corresponding sequence. They define distance as a tuple of values, one representing the time and the other the amplitude. These two values together determine the dissimilarity. In this model, the distance can be made invariant with respect to 6 transformations. These transformations are shifting, uniform amplitude scaling, uniform time scaling, uniform bi-scaling, time warping and non uniform amplitude scaling. In a separate work, Park et al. [12] use the notion of time warping distance. This distance metric compares sequences of different lengths by stretching them. However, this metric does not satisfy triangle properties and can cause false drops [22]. Another distance metric D_{norm} is defined by Lee et al [11] for multidimensional sequences. Although this metric has a high recall, it again allows false dismissals, when determining candidate solution intervals.

Range searches and nearest neighbor searches in *whole matching* and *subsequence matching* [2] 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. [2] developed one of the first solutions to this problem. They 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 values that have most of the energy contained in the sequence. Chan et al. [5] used Haar wavelet transform to reduce the number of dimensions and compared this method to DFT. They found that Haar wavelet transform performs better than DFT. However, the performance of DFT could be improved using the symmetry of Fourier Transforms [15]. In this case, both methods give similar results.

Subsequence matching is a more difficult problem. Here, the query sequence is potentially shorter than the sequences in the database. The query asks for subsequences in the database that have the same length as the query and a similar pattern. For example one can ask a query like: "Find companies which had a similar profit pattern in the last 10 years as company A's profit pattern in January 2000". A brute force solution is to check all possible subsequences of the given length. However, this is not feasible because of the large number of long sequences in the database.

Faloutsos et al. [7] proposed *I-adaptive* index to solve the matching problem for queries of prespecified length. They store the trails of the prespecified length in MBRs (Minimum Bounding Rectangles). In the same paper, they also talk about two methods *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 subsequences of prespecified length and performs queries for each of these subsequences.

A different method called *STB-indexing* has been presented by Keogh et al. [10] for subsequence matching. In this method, the database sequences are divided into non-overlapping parts of a prespecified window size. If the values of the time sequence within a segment is mostly increasing, then this segment is represented by 1. If these values are mostly decreasing, then the segment is represented by 0. The net result is that the number of dimensions in the time sequences is decreased by a factor of window size. After this transformation, the transformed sequences are stored in bins. Each bin also stores a matrix whose entries are the distance between all pairs of sequences within that bin. For a given query, the authors use the query sequence for bin-pruning and the distance matrix for inter-bin-pruning.

There are several ways to test the quality of an index structure. The size of an index structure must be small enough to fit into the memory. Furthermore, the running time of the search algorithm must be small. Two parameters are crucial to the running time: precision and the number of disk page accesses. Precision is defined as the ratio of the number of actual solutions to the number of candidate solutions generated by the search algorithm. A good indexing method and a search algorithm should have a high precision while reading few disk pages.

In this paper, we investigate the problem of range searching for queries of variable lengths. We propose an index structure and a search method to solve this problem efficiently. Our index structure stores MBRs corresponding to database sequences at different resolutions. The search algorithm splits a given query into several non-overlapping subqueries at various resolutions. For each subquery, the method performs a search in the index structure corresponding to the resolution of the subquery. These results of a subquery are used to refine the radius of the next subquery. Since the search volume is proportional to ϵ^d, where d is the number of dimensions and ϵ is the radius of the query, the search volume decreases exponentially when the radius of the query decreases. This dramatically reduces redundant computations and disk page reads. Our experimental results show that, our method is 4 to 20 times faster than the current techniques including Linear Search. To obtain the MBRs of appropriate dimensionality at different resolutions, we consider different compression techniques, namely DFT, Haar and DB2.

Storing information at different resolutions increases the amount of storage needed by our index structure. However, a good index structure must not occupy too much space. So, in the second part of the paper we propose three methods to compress the index information. These methods decrease the number of MBRs by replacing a set of MBRs with a larger MBR that tightly covers these MBRs. The first of these methods splits the set of all MBRs into equal non-overlapping subsets of equal size and replaces each of these sets with an MBR. The second method minimizes the increase in the total volume due to merges of MBRs. The third method combines the previous techniques and balances the distribution of the subsequences into MBRs. Our experimental results show that our index structure is 3 to 15 times faster than the current techniques even after compressing the index to one-fifth.

The rest of the paper is as follows. The problem of range queries for subsequence search is defined in Section 2. Previous work is discussed in the same section. Our index structure and the search algorithm are presented in Section 3. Experiment results are given in Section 4. Index compression methods are presented in Section 5. We end with a brief discussion in Section 6.

2 Current Search Techniques

Current subsequence search techniques can be classified in two groups as fixed length queries and variable length queries. Section 2.1 discusses the case when the length of the queries is prespecified. Sections 2.2 and 2.3 discuss two solutions for variable length queries.

2.1 Fixed Length Queries

A simpler version of the subsequence matching problem is when the length of a query sequence equals a prespecified window size w. An indexing method called *I-adaptive* was proposed by Faloutsos et al. [7] for this problem. This method first transforms all possible subsequences of length w in the database using DFT. They initially create an MBR that covers the first subsequence in the frequency domain. The method extends the MBR to cover the next subsequence only if the marginal cost of the MBR does not increase. If the marginal cost increases, the method does not extend the MBR, but it creates a new MBR that covers the new subsequence. After constructing all the MBRs, it uses an R-tree [9, 3, 20] to index them.

2.2 Prefix Search Method for Variable Length Queries

For variable length queries, Faloutsos et al. propose two different solutions using *I-adaptive* index, namely *Prefix Search* and *Multiple Search* [7].

Prefix Search assumes that there is a prespecified lower bound w on the length of the query. The method constructs the index structure using *I-adaptive* method with window size w. For a given query q of any length, it searches the database using a prefix of q of length w. If the distance between the query q and a subsequence s of length $|q|$ is $d(q, s)$, where $d(q, s)$ represents the Euclidean distance between sequences q and s, then the distance between any of their prefixes can not be greater than $d(q, s)$. Therefore, the method does not cause any false drops.

However, Prefix Search does not use all the information available in the query sequence. If $|q| = kw$ for some k, then the expected distance between the prefixes of query q and a database sequence s of the same length is $\sqrt{k}$ times less than that of $d(q, s)$. This means that the Prefix Search method actually searches the space with an implicit radius which is $\sqrt{k}$ times more than the original radius. If the method uses *dim* dimensions of the transformed sequences, the search volume increases by $(\sqrt{k})^{dim}$. If $|q|$ is much larger than the window size, the method incurs many false retrievals which decrease the precision and increase the number of disk reads and post processing cost. For example, let the length of the query sequence be 16 times greater than the window size. In this case, the implicit search radius is 4 times more than actual search radius. If the index stores 6 dimensions, then the implicit search volume is more then 4000 times larger than actual search volume.

2.3 Multiple Search Method for Variable Length Queries

Multiple Search assumes that the length of the query sequence is an integer multiple of the window size[1], i.e. $|q| = kw$ for some integer k. Given a query sequence q and a radius ϵ, the method divides the query sequence into k non-overlapping subsequences of length w. Later, the method runs k subqueries, i.e. one for each subsequence, with radius $\epsilon/\sqrt{k}$, and combines the results of these subqueries. The method does not incur any false drops, because if a sequence is within ϵ distance of the query, then at least one of the corresponding subsequences must be within the distance $\epsilon/\sqrt{k}$.

However, there are several problems with the Multiple Search method.

- The number of subqueries increases linearly with the size of the query. For example assume that the length of a query sequence is 2^9 and $w = 16$, then the number of subqueries for this query is 32. At the same time, the expected cost of each subquery is the same as the cost of the initial query. To explain this, consider a query of length kw. The radius for each subquery is $\epsilon/\sqrt{k}$. The likelihood of finding a subsequence of length w within a distance of $\epsilon/\sqrt{k}$ from the subquery is the same as that of finding a sequence of length kw within a distance of ϵ from the original query. This implies that the expected cost of each subquery is the same as that of the original query. On account of the linear number of subqueries, this implies that the total search cost of the Multiple Search method increases linearly with the length of the query.

- Multiple Search has a post-processing step, in which the results from different subqueries are combined and filtered. This is an extra CPU overhead that is avoided in Prefix Search.

3 Our Proposed Method

The main problem with the two solutions for variable length queries is that the index structure cannot use all the information available in the query sequence due to the static structure of the index and unpredictable length of the query. Our solution basically alleviates this problem by storing information at different resolutions about the time sequences in the database.

The rest of this section is as follows. The index structure is explained in Section 3.1. The search algorithm on this index structure is given in Section 3.2. Our method is compared with the Prefix Search and Multiple Search methods in Section 3.3.

3.1 Index Structure

Our index structure is created as follows. 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. 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 MBRs for the j^{th} sequence corresponding to window size 2^i. Similar to the *I-adaptive* index, each MBR in $T_{i,j}$ is extended to tightly cover the next transformed subsequence of length w until the marginal cost of the MBR increases. When the marginal cost of an MBR increases, a new MBR is created and used for later subsequences.

Our index structure is presented in Figure 1. The index structure stores separate trees for the MBRs of different time sequences instead of storing them using a single R-tree. The i^{th} row of our index structure is represented by R_i, where $R_i = \{T_{i,1}, ..., T_{i,n}\}$ corresponds to the set of all sub-structures 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 sub-structures for the j^{th} time sequence in the database.

[1]If the query is not an exact multiple of the window size, then the longest prefix that is a multiple of the window size can be used.

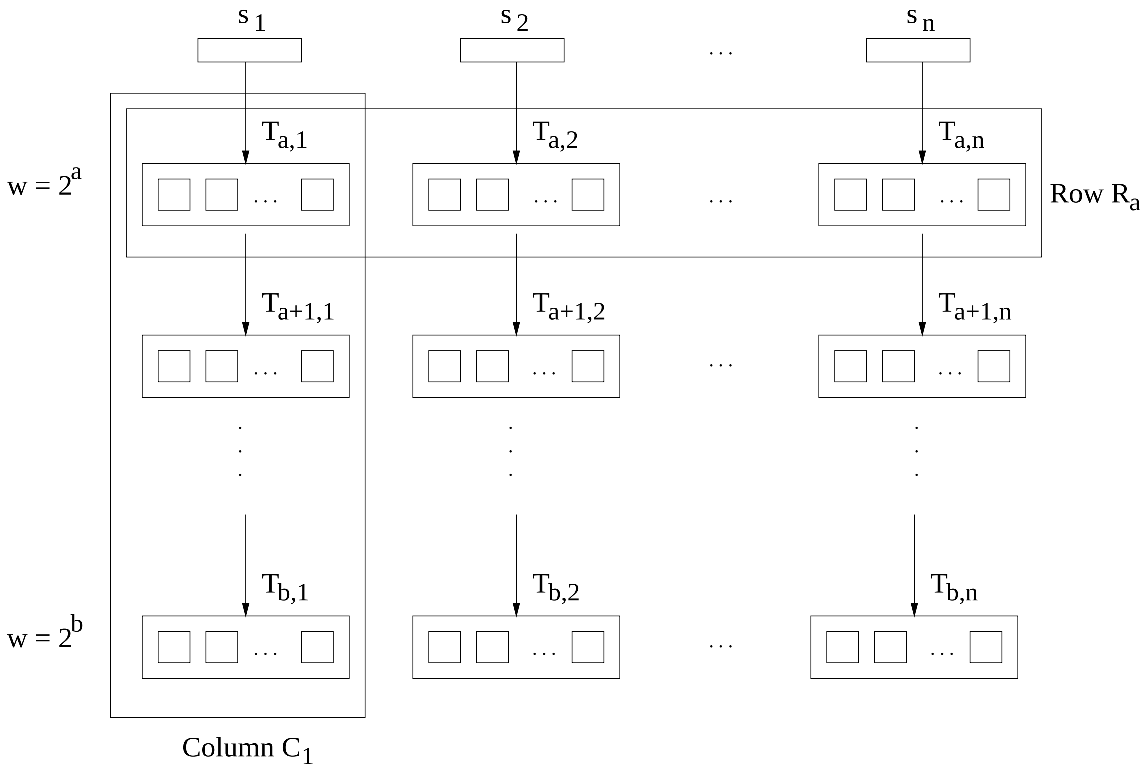


Figure 1: An illustration of our index structure

3.2 Search Algorithm

Our search method partitions a query sequence of any 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 our index structure. This is called a partial range query, because our method computes distance of the query subsequence to the MBR's, but it does not compute the distance of the query subsequence to the actual subsequences contained in these MBRs.

Given any query q of length $k2^a$, there is a unique partitioning[2] for it as $q = q_1 q_2 ... q_t$, with $|q_i| = 2^{c_i}$ and $a \leq c_1 < ... < c_i < c_{i+1} < ... c_t \leq b$. This partitioning corresponds to the 1's in the binary representation of k. The method first performs 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 MBRs that lie within a distance of ϵ from q_1. Using the distances to these MBRs, 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}. Since each subquery is performed at different resolution, we call this method *Multi Resolution*, i.e. *MR method*. Similarly, we call our index structure the *MR index structure*. The idea of the above radius refinement is captured in the following lemma.

Lemma 1 *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 $d(q, x) < \epsilon$ then*

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

where B is an MBR that covers x_1 and $\mathcal{B}$ is any arbitrary set of MBRs that contains B.

[2]Similar to the Multiple Search, 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 in the query.

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

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

Since $d(q_2,x_2) \geq d(q_2,B)$,

$$d(q_2,x_2) \leq \sqrt{\epsilon^2 - d(q_1,B)^2})$$

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

An example partitioning of a query sequence is given in Figure 2. 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'. 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 MBRs is then processed to find the actual subsequences using disk I/Os.

Figure 3 presents the complete 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 is carried out (Step $2b(i)$), and a range refinement is carried out (Step $2b(ii)$). When all rows have been searched, the disk pages corresponding to the sequence are read (Step $2c$). Finally post-processing is carried out to eliminate false retrievals (Step $2d$).

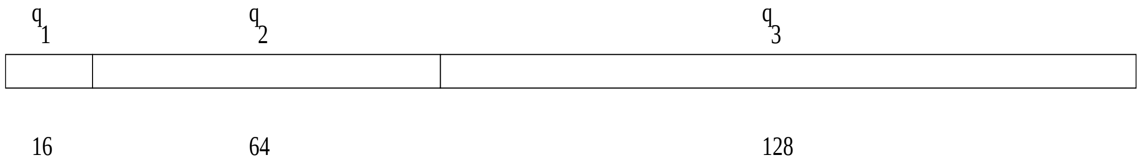


Figure 2: Query partitioning example for $|q| = 208$

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 $j = 1 : n$
   (a) Define $\epsilon_{c_1,j} = \epsilon$
   (b) For $i = 1 : t$
      i. Perform range query on $T_{c_i,j}$ using $\epsilon_{c_i,j}$. Let $Res_{c_i,j}$ be the resulting set of MBRs whose distance to q_i are less than $\epsilon_{c_i,j}$.
      ii. $\epsilon_{c_{i+1},j} = max_{B \in Res_{c_i,j}}\{\sqrt{\epsilon^2_{c_i,j} - d(q_i,B)^2}\}$
   (c) Read the disk pages corresponding to $Res_{c_t,j}$
   (d) Perform post processing to eliminate the false retrievals.

Figure 3: MR algorithm for variable length range queries

As a consequence of Lemma 1 we have the following theorem.

Theorem 1 *The MR algorithm does not incur any false drops.*

We should note the following about the above algorithm.

- Every column of the index structure (or each sequence in the database) is searched independently. For each column, the refinement of radius is carried out independently, and proceeds from top to bottom.

- For each sequence, no disk reads are done until the termination of the for loop in Step $2b$. Furthermore, the target pages are read in the same order as their location on the disk. As a result, the average cost of a page access is much less than the cost of a random page access.

- Performing subqueries in increasing length order improves the performance. This can be explained as follows: Disk reads occurs in the last subquery. Therefore the last subquery dominates the cost of the algorithm. The implicit search radius for the last subquery is $\epsilon \times \sqrt{|q|/w}$, if w is the length of the last subquery. Let w_i and w_j be the lengths of the longest and the shortest subqueries. The index typically stores at most 6 rows, as a result, the expected value for the ratio w_i/w_j is at most 16. Let ϵ_i be the search radius when subqueries are performed in increasing lengths and ϵ_j be the search radius when subqueries are preformed in decreasing lengths. Then, the expected ratio of the final implicit search radius for increasing length order to decreasing length order is $(\epsilon_j/\epsilon_i) \times \sqrt{w_i/w_j}$. Our experimental results show that the average value for the final radius is at least 30% of ϵ and the average value for ϵ_j is between $40 - 50\%$ of ϵ. Therefore, the expected value for the implicit radius is minimal if the index is searched from top to bottom.

- A special condition occurs when one of the subqueries q_i turns out to be in one of the MBRs (say B) of tree $T_{i,j}$. In this case $d(q_i, B) = 0$, $\epsilon_{c_{i+1},j} = \epsilon_{c_i,j}$, and no radius refinement is possible at this search step. However, if we use the actual subsequences corresponding to B (by accessing disk pages), a refinement may still be possible.

- One may be tempted to simplify the algorithm by iterating row by row, i.e., by finding MBRs within the radius corresponding to all sequences in a row, and then refining the search radius simultaneously across all sequences for the next row. Such a row based refinement impacts the performance of the algorithm. This is because the amount of radius refinement is reduced due to non-local considerations. With respect to Lemma 1, this means that a row-based approach increases the size of $\mathcal{B}$, and consequently reduces the amount of refinement.

3.3 Comparison of the Methods

The MR method alleviates a number of problems of Prefix Search and Multiple Search that arise from their fixed structure. These problems are mainly the large amount of redundant computation and redundant disk page reads. Therefore, the run time of the MR method is much lower compared to these two methods.

Multiple Search can have as much as $|q_{max}|/w$ subqueries, where q_{max} is the longest possible query. On the other hand, since there is a binary decomposition for the MR method, the upper bound on the number of subqueries for a given query is $log(|q_{max}|) - log(w)$ where q_{max} is the longest possible query and w is the minimum window length. This means a dramatic reduction in the preprocessing cost. Furthermore, disk reads are performed for each subquery of the Multiple Search. On the other hand, for the MR method, disk reads are performed only after the last subquery.

Each subquery of the MR method refines the radius of the query. Since the search volume is proportional to ϵ^{dim}, even small refinements in the radius can prevent large number of disk reads. For example, if 10 dimensions are used in the index, then a 5% decrease in the radius will decrease the search volume by 40%. Therefore, the total search volume, hence the amount of computation and disk reads, for the MR method will be much lower than the Prefix Search method.

However, there is a drawback of the MR index structure. The index structure uses more space than both Prefix Search and Multiple Search. This is because it stores index structures at various resolutions. Since all the preprocessing step must be done in the memory, the index structure must fit into memory. Defining an upper bound on the window size, i.e. the number of different resolutions,

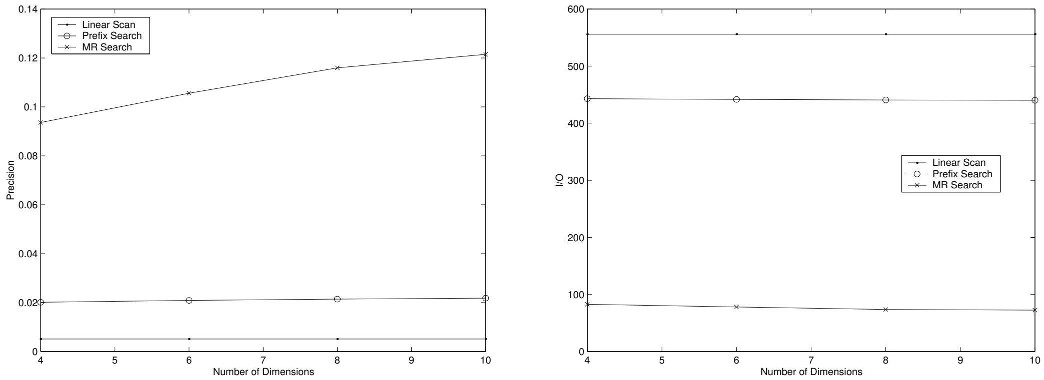


Figure 4: Precision and number of disk reads for stock market data for different dimensionalities.

can bound the size of the MR index. Later, we will present several methods to shrink the MR index structure to fit into memory without much performance drop.

4 Experimental Results

We have done several experiments to compare the techniques discussed and to test the impact of using different parameters on the quality of these methods. We used two datasets in our experiments. The first dataset is composed of stock market data taken from *chart.yahoo.com*. This dataset contains information about 556 companies from various fields for 1024 days. The second dataset is formed synthetically by adding four sine signals of random frequencies and some amount of random noise. This dataset is composed of 500 time series data each of length 1024. We considered variable length queries of lengths between 16 and 1024. We assumed that the lengths of the queries are uniformly distributed. The radii for these queries are selected uniformly between 0.10 and 0.20. We assume that the index structure fits into the memory for these experiments.

In order to obtain data at various resolutions, we need to transform the data to compact the time sequences and obtain MBRs of appropriate dimensionality. We experimented with three transformations, namely DFT, Haar wavelet [8, 16, 21] and DB2 wavelet. We also run some of the experiments using other wavelet bases such as DB3, DB5, SYM2, SYM3, SYM5 and several Coeiflets. The results for SYM wavelets were very close to DB wavelets, and we did not get any improvements with these basis functions. The results for Coeiflets were also not promising. Although there is no significant difference, Haar wavelet performed slightly better than DFT. DB2 performed poor for lower dimensions, but its performance was comparable to Haar for higher dimensions. We report all our experiments using Haar transformation.

4.1 Precision and Disk Reads for Variable Length Queries

Our first set of experiments compared precision and I/O for the two dataset for three different indexing methods: Linear Search, Prefix Search and MR search. The results are presented in Figure 4 and Figure 5. Figure 4 reports the results for stock market data. Figure 5 reports the results for synthetic data. The results for Multiple Search are not presented, because it was not competitive with other methods. The total number of page accesses for this technique was even more than the total number of pages on the disk. This is because real data is generally clustered, and each of the subqueries access a high percentage of the pages.

According to our results, the MR method performs better than both Linear Scan and Prefix Search in all dimensionalities in terms of both precision and the number of page accesses. For stock market data using 10 dimensions, the precision of the MR method is more than 5 times better than

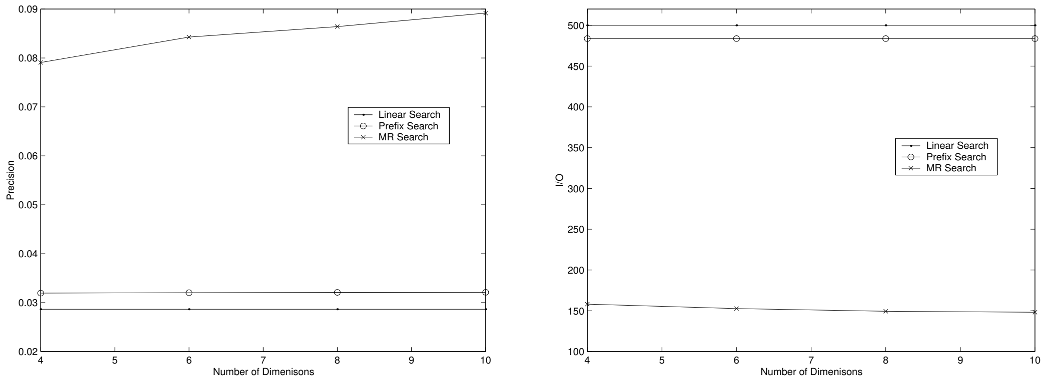


Figure 5: Precision and number of disk reads for synthetic data for different dimensionalities.

the Prefix Search method and more than 20 times better than the Linear Search method. For the same parameters, the number of page reads for the MR method is less than one-sixth of both Prefix Search and Linear Search. Precision increases as the number of dimensions increases, because more information can be stored by using more dimensions. The amount of I/O decreases as the number of dimensions increases because of the same reason. As the number of dimensions increases, the increase in the performance of the MR method is much higher than that of Prefix Search. For example, by using 10 dimensions instead of 4 dimensions, the number of page accesses for the MR method decreases by 13% while that of the Prefix Search decreases by less than 1%. Similarly, precision for the MR method increases by 30% while precision for the Prefix Search increases by 8%.

Another important observation is that Prefix Search reads almost all the pages. The expected length of a query is 512. Since the minimum query length is 16, the expected implicit radius for a query is $\sqrt{512/16}$ (approximately 5.6) times larger than the actual range. This means that the expected value of ϵ is 0.88. For this value of ϵ, any given query will probably cover the entire search space.

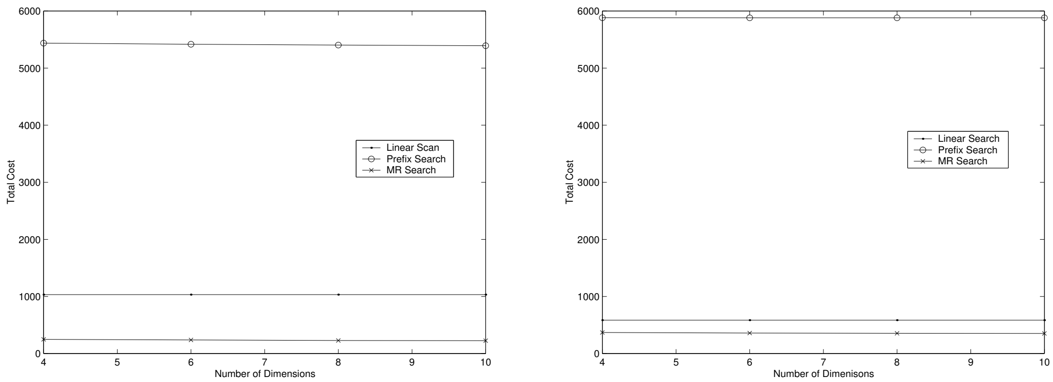
The subqueries in the MR method correspond to binary representation of the ratio of the length of the query sequence to the minimum window size. Therefore, the expected number of subqueries for the MR method is $(log(1024) - log(16))/2 = 3$. In our experiments, the average number of subqueries for the MR method is 3.08. On the other hand, for Multiple Search, the expected number of subqueries is $512/16 = 32$. In our experiments, the average number of subqueries for the Multiple Search method is 31.85, more than 10 times higher than the MR method.

4.2 Total Cost for Variable Length Queries

Although Figure 4 and Figure 5 give us information about the precision and the number of disk page reads, this is not sufficient to determine the relative total cost. In order to find the total cost of the methods, we have to determine the contribution of these parameters to total cost.

The cost of reading a page is much higher than the cost of making arithmetic operations if the operands are in the memory. Furthermore, the cost of reading pages in a random order is much higher than the cost of reading them sequentially. In case of random I/O, there is a high seek and rotational latency cost for each page access. However, for sequential scan, when one page is read, the disk head is at the start of the next page to be read. There are also several other factors that effect page access cost like buffering and prefetching. The ratio of the cost of random read to the cost of sequential read depends on the architecture. Typical values for this ratio are between 5 and 15.

Both Prefix Search and Multiple Search performs random page reads. Because the MBRs are stored in R-tree, and an MBR in higher levels can contain MBRs from different time sequences, there is no internal order. On the other hand, Linear Search reads pages sequentially. For the MR index



(a) Total cost for stock dataset (b) Total cost for synthetic dataset

Figure 6: Total cost in terms of number of page transfers

structure, since the sub-structures $T_{i,j}$ corresponding to time sequences in the MR index structure are traversed in the same order as the corresponding pages on the disk, target pages are read in sequential order. Therefore, the average cost of a page read is much less than the cost of a random page read [4, 17]. We use the cost model proposed in [17] to find the cost of reading a set of pages efficiently.

In order to combine the experimental results for precision and disk read, we found the ratio of the average time elapsed for a single page read and a single Euclidean distance computation experimentally on several machines. Later, we used a disk simulator to check these results. Experimental results and simulator results were very close. Our page size was $8K$ which corresponds to one time sequence of length 2^{10}, where each entry is a double. As discussed before, the expected query length is half of the maximum possible query length, i.e. 2^9. Therefore, the expected cost of a candidate subsequence is equal to the cost of finding the Euclidean distance between two double sequences of length 2^9. According to our experimental results, the ratio of the cost of a page read to the cost of a distance computation varies between 300 to 600 on different machines. We used the upper bound, i.e. 600 in our cost computation. This is the worst case for the MR method, because lower values of this ratio increases the contribution of precision to total cost, and our method has much higher precision than other techniques. Total cost of a query is computed as follows:

$$Total\ Cost = CPU\ Cost + IO\ Cost, \text{ where}$$
$$CPU\ Cost = Database\ Size \times Precision/600 \text{ and}$$
$$IO\ Cost = k \times Number\ of\ page\ reads$$

The *conversion coefficient* k in the computation of the IO Cost is used to convert the average cost of page read in a method to a single page transfer cost. For Prefix Search, this coefficient is 12, because the cost of a random page read is 12 times more than that of a sequential page read [17]. For Linear Search, conversion coefficient is simply 1. For the MR method, this value depends on the disk used, database size and the number of pages read. We used the table provided in [17] to determine the proper coefficients. Typically, this value is between 2.0 and 3.0. We also validated this experimentally by reading subsets of pages in order from a disk.

The total cost comparison corresponding to the stock market dataset in Figure 4 is presented in Figure 6a and the total cost comparison corresponding to the synthetic dataset in Figure 5 is presented in Figure 6b It can be seen that for stock market data, the MR method over-performs Prefix Search and Linear Search in total cost. The MR Search method performs more than 4 times better than Linear Search and more than 20 times better than Prefix Search. Another observation is Prefix Search always has the highest cost. Prefix Search reads almost every page on the disk in a random order. Therefore, the cost of disk reads dominates CPU time and it performs even worse than Linear Search.

Although, Prefix Search is a good method for subsequence search when the query size is predefined, we conclude that it is not suitable when the query size is not fixed. As the number of dimensions increase from 4 to 10, the total cost of the MR method decreases by 13%, while that of the Prefix Search method decreases by less than 1%.

5 Index Compression

The MR method performs better than all the other techniques discussed in this paper in terms of precision, number of page accesses, and total cost. However, the method keeps information about the data at different resolutions in the index structure. Therefore, the MR index structure uses more space than some other index structure like *I-adaptive* that maintain information at a single granularity level. In order to perform the preprocessing steps of the search in the memory, the index structure must fit into the memory. If the index structure does not fit into the memory, then the method will incur early page accesses for the index. Therefore, the performance of the MR method could degrade if enough memory is not available. In this section, we consider index compression techniques with which the memory requirements of the MR index structure can be made similar to the other index structures.

5.1 Compression methods

We experimented with three index compression methods for the MR index to shrink the size of the index: *Fixed Compression*, *Greedy Compression* and *Balanced Greedy Compression*. These methods merge a set of MBRs to create another MBR of the same dimensionality that tightly covers and replaces the constituent MBRs. The MR index assumes that an MBR contains subsequences that start at consecutive offsets. In order to preserve this property, only consecutive MBRs are merged. The next question is which MBRs should be merged. Each merge operation increases the total volume covered by the index structure. As the total volume increases, the probability that a range query accesses one or more of these MBRs increases. This increases both the number of disk reads and the size of the candidate sets. We consider three strategies for merging MBRs. Let r be the desired compression rate.

- Fixed Compression is based on the observation that in real applications closer subsequences are similar. This method merges the first r MBRs into an MBR, second r MBRs into another MBR and so on. For example, Assume that a time sequence has 11 MBRs, represented by $< 1 >$, $< 2 >$, $< 3 >$, $< 4 >$, $< 5 >$, $< 6 >$, $< 7 >$, $< 8 >$, $< 9 >$,$< 10 >$ and $< 11 >$. If the compression rate is 3, after compression it looks like $< 1,2,3 >$, $< 4,5,6 >$, $< 7,8,9 >$, $< 10,11 >$. This is a very fast method. However, if consecutive MBRs are not close, the performance drops quickly since it does not adapt to the MBRs in the index structure.

- Greedy Compression tries to minimize the increase in the total volume occupied by the MBRs at each merge operation. The algorithm simply chooses the two consecutive MBRs with minimal volume increase until the number of MBRs is decreased by the given ratio r. This method does not guarantee to find the best way of shrinking, but it is expected to perform better than fixed method, because it tries to adapt to the layout of the MBRs at each merge operation. However, there is a problem with this method. The two MBRs to be merged are determined at each step independently. Therefore, in the compressed index, some of the MBRs can contain many more subsequences than others. This will make the partitioning unbalanced. Consequently, a hit on one of the boxes will incur the overhead of reading large number of candidate subsequences. This decreases the precision of the index.

- The compression method can be made balanced by incorporating a penalty on the merge operation corresponding to the number of subsequences contained in the resulting MBR. This is called *Balanced Greedy Compression*. In this method, the cost of a merge operation is defined as the volume increase multiplied by the number of subsequences in the new MBR. Similar to the Greedy Compression Method, two MBRs that minimize this cost are selected to merge at each

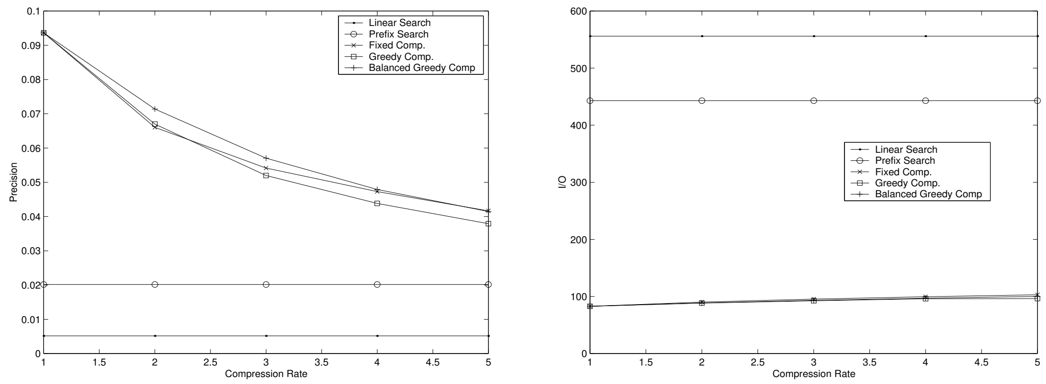


Figure 7: Precision and number of disk reads for stock market data for different compression rates.

step. This method overcomes the problem of unbalanced partitions discussed for the Greedy Compression technique.

5.2 Experimental Results

The experimental results for these index compression techniques applied to the MR Search method are presented in Figure 7. The experiments are run on stock market dataset using Haar wavelet transform. The experiment results for the synthetic dataset are similar. The compression rate varies from 1 to 5. If the compression rate is 1, it means that the index is not compressed. If the compression rate is 5, it means that the index is compressed to one-fifth. For comparison, these figures also include the results of Prefix Search method and the Linear Search method. The index is not compressed for Prefix Search. According to these results, the precision for the Balanced Compression technique is always highest and Fixed Compression is better than the Greedy Compression method. Even after compressing the MR index to one-fifth, the precision of the MR method is more than 2 times better than the precision of Prefix Search for all three compression techniques. The number of disk reads of the MR method does not increase much with the compression. This is because, although there is an increase in the size of the candidate set, these candidates come from the same time sequence. Therefore, there is only one page access corresponding to that hit. Greedy Compression is slightly better than the other two techniques in terms of disk page reads.

The total cost for the MR method using these index compression methods is given in Figure 8. The parameters for the cost model are obtained from Figure 7. Balanced Greedy heuristic always has the lowest cost and Fixed Compression has the highest cost. Figure 9 compares the total cost of Linear Search, Prefix Search and MR Search with Balanced Greedy Compression for compression rates up to 5 using the same parameters. In this figure, we compare the performance of the compressed MR index with the uncompressed versions of the other methods. When the compression rate is 5, the total cost of the MR method is 3 times better than the cost of Linear Search and more than 15 times better than that of the Prefix Search method.

By compressing the MR index to one-fifth, the number of rows the MR index can be increased 5 times, because the space requirement of the index decreases to one-fifth. If the original index structure contains n_r rows, increasing the number of rows 5 times increases the length of the longest query that can be handled effectively by our method 2^{5n_r} times. If the MR index structure contains a single row like the I-adaptive index, then by compressing the MR index to one-fifth and increasing the number of rows to 5, the length of the longest query that can be handled effectively turns out to be 2^5 times more than that for Prefix Search.

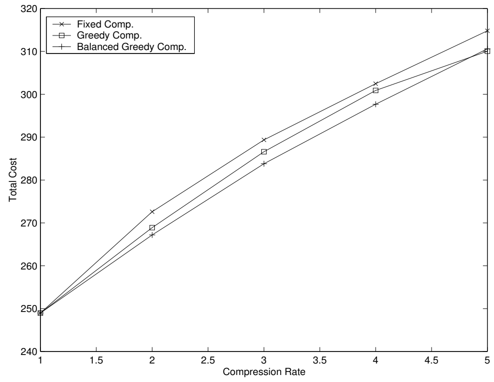


Figure 8: Total cost for different index compression techniques

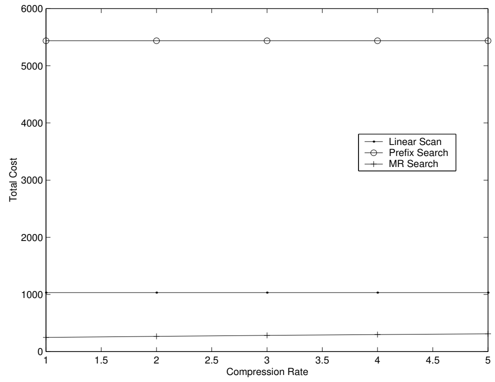


Figure 9: Total cost for different search techniques.

6 Discussion

In this paper, we considered the problem of variable length queries for subsequence matching for time sequences. We discussed two existing solutions: Prefix Search and Multiple Search. These methods are adaptations from a similar problem when the query size is prespecified. We discussed the reasons that make these two solutions unsuitable for variable length queries. The main problem with these methods was that they did not use all the information available in the query sequence due to the static nature of the index. Later, we proposed a new indexing method called MR index to overcome this problem. Our index stores information about the data at different resolutions. Given a query of any length, our method splits this query into several subqueries at these resolutions. The ranges of the subqueries are successively refined, and only the last subquery performs disk reads. In our experiments, the value of ϵ decreased by $50-60\%$ in the refinement steps. A 50% decrease in the radius of the query means 94% decrease in the query volume if 4 dimensions are used in the index. For stock market dataset, the MR method performed 4 to 20 times better than the other methods including Linear Search. For the synthetic dataset, the number of page reads for the MR method was very close to the minimum possible number of page accesses. For example, for Haar wavelets and 6 dimensions, the number of page reads for the MR method is only 60% more than the minimum number of pages required. On the other hand Prefix Search accessed almost all the pages.

Since the MR index stores resolutions at powers of two, there is a logarithmic upper bound on the number of subqueries. This is a big improvement over Multiple Search in which the number of subqueries is bounded linearly. In our experiments, the average number of subqueries is 3.08, which is very close to the theoretical value for the expected number of subqueries (3) for our query model. On the other hand, average number of subqueries for the Multiple Search was 32, which is 10 times more than that of our method.

Our search technique traverses the time sequences in the same order as their locations on the disk. Therefore, target pages are read in sequential order. This decreases the page access cost dramatically. This is one of the main problems with both Prefix Search and Multiple Search. Since they perform random page reads, the cost of a single page read is much higher than that of Linear Scan.

Since the MR index stores information at multiple resolutions, we proposed three methods to further compress the index structure without losing much information. These methods are Fixed Compression, Greedy Compression and Balanced Greedy Compression. These methods compress the index structure by merging some of the MBRs to create a larger MBR. Out of these three methods, Balanced Greedy Compression has the best results since it minimizes the information loss while distributing the subsequences almost evenly to the MBRs. As a result, the MR method performed 3 to 15 times better than other methods even after compressing the index to one-fifth.

Our notion of similarity between two time sequences is based on Euclidean distance. Such a metric is not invariant with respect to scaling and shifting, a possible shortcoming for comparison of time-based data. However, as shown by Chu et al. [6], a transformation onto the shift eliminated plane achieves this invariance with respect to scaling and shifting. This transformation could be applied to all the subsequences prior to the construction of the index. In this case, each query subsequence must be mapped onto the shift eliminated plane before compressing it with DFT or DWT.

The performance of the MR search algorithm can be further improved by performing the refinement decisions more locally. This can be done by computing ϵ for each MBR separately. Different MBRs belonging to the same sequence would have independent refinements. The propagation of radius between rows would be based on the subsequences included within the MBRs. For example, if MBRs B_1 and B_2 in row R_1 included positions 1-150 and 151-300 of a database sequence, and if MBR B_3 at the next row included positions 1-300 of the sequence, then B_3 would use the radii form B_1 and B_2 in its radius refinement. If the time sequences in the database are very long leading to a large number of MBRs corresponding to a time sequence, then this change in the search technique could be useful. However, index compression decreases the benefits of this improvement since it decreases the number of MBR's.

The simple grid structure of our index provides a good opportunity for parallel implementation. The index structure can be partitioned into subsets of columns and each subset can be mapped to a separate processor with its own disk. No communication is needed among the processors, except to merge the results at the end.

References

[1] 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.

[2] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. Efficient similarity search in sequence databases. In *FODO*, Evanston, Illinois, October 1993.

[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 *ACM SIGMOD*, pages 322–331, Atlantic City, NJ, 1990.

[4] Seeger Bernhard, Larson Per-ke, and McFayden 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] Kelvin Kam Wing Chu and Man Hon Wong. Fast time-series searching with scaling and shifting. In *PODS*, Philedelphia, PA, 1999.

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

[8] Amara Graps. An introduction to wavelets. In *IEEE CS&E*, 1995.

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

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

[11] 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.

[12] Sanghyun Park, Jeehee Yoon Wesley W. Chu, and Chihcheng Hsu. Fast retrieval of similar subsequences of different lengths in sequence databases. In *ICDE*, San Diego, CA, February 2000.

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

[14] Davood Rafiei and Alberto O. Mendelzon. Similarity-based queries for time series data. In *ACM SIGMOD*, pages 13–25, Tucson, AZ, 1997.

[15] Davood Rafiei and Alberto O. Mendelzon. Efficient retrieval of similar time sequences using dft. In *FODO*, Kobe, Japan, 1998.

[16] R. M. Rao and A. S. Bopardikar. *Wavelet Transforms Introduction to Theory and Applications.* Addison Wesley, 1998.

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

[18] Cyrus Shahabi, Xiaoming Tian, and Wugang Zhao. TSA-tree: A wavelet-based approach to improve the efficieny of multi-level surprise and trend queries. In *SSDBM*, 2000.

[19] A. Singh, D. Agrawal, and K. V. Ravi Kanth. Dimensionality-reduction for similarity searching in dynamic databases. In *ACM SIGMOD*, Seattle, WA, June 1998.

[20] V.S. Subrahmanian. *Principles of Multimedia Database Systems.* The Morgan Kaufmann Series, 1998.

[21] Martin Vetterli and Jelena Kovacevic. *Wavelets and Subband Coding.* Prentice Hall, 1995.

[22] Byoung-Kee Yi, H.V. Jagadish, and Christos Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE 98*, pages 23 – 27, Orlando, Florida, February 1998.