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-studied problem. Most of the current techniques work well for queries of a prespecified length, but not for variable length queries. We propose a new indexing technique that works well for variable length queries. The central idea is to store index structures at different resolutions for a given dataset. The resolutions are based on wavelets. For a given query, a number of subqueries at different resolutions are generated. 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 companies which have similar profit/loss patterns, finding similar motions 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 can lead to extensive result sets, 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 [1, 5, 7, 15, 20, 21]. 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 and Wong [6] defined the *Shift Eliminated Transformation*, which maps the database and query sequences onto the *Shift Eliminated Plane*. These transformations are then used to compute the minimum distance (after all possible scaling and shiftings) between two time sequences. The authors' idea is applicable to any Euclidean distance based search algorithm, in particular to the one presented in this paper.

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, and Zhang [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 between two time sequences as a tuple of values, one representing the time and the other the amplitude. This notion of distance can be made invariant with respect to 6 transformations: shifting, uniform amplitude scaling, uniform time scaling, uniform bi-scaling, time warping and non uniform amplitude scaling. In a separate work, Park, Wesley, Chu, and Hsu [13] use the idea 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 [23]. Another distance metric D_{norm} is defined by Lee, Chun, Kim, Lee, and Chung [12] 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* [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, Faloutsos, and Swami [1] 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 considering 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. They found that Haar wavelet transform performs better than DFT. However, the performance of DFT can be improved using the symmetry of Fourier Transforms [16]. 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: "Find companies which had a similar profit pattern 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, Ranganathan, and Manolopoulos [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 present 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 and Pazzani [11] 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 are 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 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. The 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 Scan. 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 EMBR (Extended MBR) that tightly covers these MBRs. The first of these methods splits the set of all MBRs into non-overlapping subsets of equal size and replaces each of these subsets with an EMBR. The second method combines MBRs so that the volume of the resulting EMBR is minimized. The third method combines the previous techniques and balances the distribution of the subsequences into EMBRs. Our experimental results based on the third method 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 and the existing techniques are discussed in Section 2. Our index structure and the search algorithm are presented in Section 3. Experimental 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 based on whether they handle fixed length queries or 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. Only the first few frequency coefficients are maintained after this transformation. An MBR that covers the first subsequence in the frequency domain is created. This MBR is extended to cover the next subsequence if the marginal cost does not increase. Otherwise a new MBR that covers the new subsequence is created. The resulting MBRs are stored using an R-tree [3, 9].

2.2 Variable length queries: prefix search technique

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* technique with window size w. Given a query q of an arbitrary length, it searches the database using a length w prefix of q. 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 w-length prefixes of query q and a database sequence s is $\sqrt{k}$ times less than that of $d(q, s)$. This means that the Prefix Search technique actually searches the space with an implicit radius which is $\sqrt{k}$ times more than the original radius. If the method uses dim dimensions for 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 decreases the precision and increases the number of disk reads. For example, let the length of the query sequence be 16 times the window size. In this case, the implicit search radius will be 4 times the actual search radius. If the index stores 6 dimensions, then the implicit search volume will be more than 4000 times the actual search volume.

2.3 Variable length queries: multiple search technique

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 technique.

- The cost of a query increases linearly with its size. 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 a 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. The number of subqueries (k) increases linearly with the size of the original query. As a result, the total search cost of the Multiple Search technique 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.

3 Proposed method

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

3.1 Storing information at multiple resolutions

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 1. Our index structure stores a grid of trees $T_{i,j}$, where i ranges from a to b, and j ranges

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

from 1 to n. Tree $T_{i,j}$ is the set of MBRs for the j^{th} sequence corresponding to window size 2^i. In order to obtain $T_{i,j}$, we transform each sequence of length 2^j in sequence s_i using DFT or wavelets, and choose a few of the coefficients from the transformation. The transformed sequences are stored in MBRs as in the *I-adaptive* index structure. We begin with an initial MBR. It is extended to cover the next 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. 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 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.

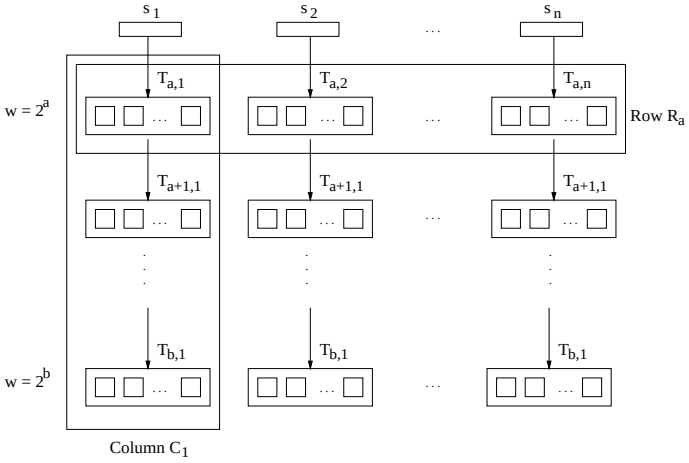

Figure 1. Layout of the index structure

3.2 Search algorithm

Our search method 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 MBRs, not the distance to the actual subsequences contained in the MBRs.

Given any query q of length $k2^a$ and a range ϵ, there is a unique partitioning[2], $q = q_1q_2...q_t$, 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.

[2]Similar to the Multiple Search technique, 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 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 the *MR (Multi Resolution) 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 $d(q,x) \leq \epsilon$ 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.

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

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

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'. 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 MBRs is then processed to find the actual subsequences using disk I/Os.

Figure 3 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 $2d$).

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

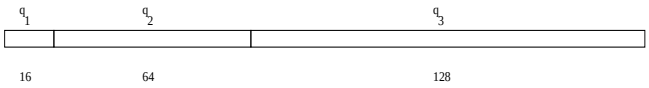

Figure 2. Partitioning for query q, $|q| = 208$

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 $j := 1$ 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 MBRs whose distances 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 disk pages corresponding to $Res_{c_t,j}$
   (d) Perform post processing to eliminate false retrievals.

Figure 3. Search algorithm.

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

We note the following about the search 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 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 (i.e. from top to bottom in Figure 1) 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 $\epsilon \times \sqrt{|q|/w}$, where w is the length of the last subquery. Let w_i and w_j be the lengths of the longest and the shortest subqueries. Let ϵ_i be the final search radius when subqueries are performed in an increasing length order, and ϵ_j be the final search radius when subqueries are performed in a decreasing length order. Then, the ratio of the search radii in the two orders is $(\epsilon_j/\epsilon_i) \times \sqrt{w_i/w_j}$. We observed experimentally that the above quantity is at least 1, implying that 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 current radius for 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 index structure 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.

Multiple Search can have as much as $|q_{max}|/w$ subqueries, where q_{max} is the longest possible query. On the other hand, the upper bound on the number of subqueries 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 number of subqueries. The other advantage of our technique is that disk reads are performed only after the last (longest) subquery, as opposed to the Multiple Search technique which performs disk reads after each subquery.

Each subquery of the MR index structure 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 our technique is much lower than the Prefix Search technique.

However, there is a drawback of the MR index structure in that it uses more space than both Prefix Search and Multiple Search. This is because it stores index structures at various resolutions. Since all the preprocessing steps must be

done in memory, the index structure must fit into memory. In Section 5, we will present several methods that shrink the size of the MR index structure without much of a drop in performance.

4 Experimental results

We carried out several experiments to compare different search techniques, and to test the impact of using different parameters on the quality of our method. 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 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 assumed that the index structure fit into memory and that the page size was 8K for these experiments.

In order to obtain data at various resolutions, we had 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, 17, 22] 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. The experimental results for both real and synthetic datasets were very similar. Therefore, we present the results only for the real dataset.

4.1 Precision and disk reads

Our first set of experiments compared precision and I/O for three different indexing techniques: Linear Scan, Prefix Search and MR search. The results are presented in Figure 4. The results for Multiple Search are not presented as it was not competitive with other methods: the total number of page accesses was even more than the total number of pages on disk.

We plot the precision and I/O (number of disk reads) as a function of the number of dimensions (coefficients) of the transformed data set. Note that for Linear Scan, the candidate set is the entire database. As a result, its precision is

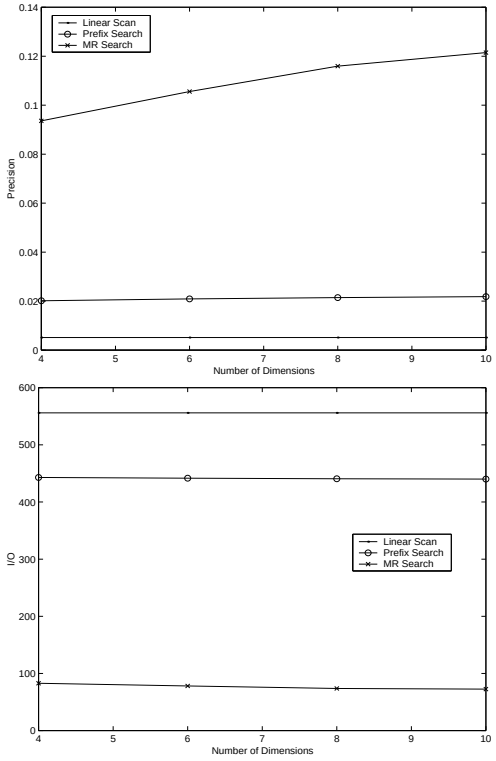

Figure 4. Precision and number of disk reads for stock market data for different dimensionalities.

the ratio of the size of the query's answer set to the size of the database. This means that in Figure 4, the answer set of the queries is about .5% of the database sequences. The performance of Linear Scan is not affected by the number of dimensions.

According to our results, the MR index structure performs better than both Linear Scan and Prefix Search in all dimensionalities in terms of both precision and the number of page accesses. For 4 dimensions, the precision of the MR index structure is more than 5 times better than the Prefix Search technique and more than 20 times better than the Linear Search technique. For the same number of dimensions, the number of page reads for the MR index structure is less than one-sixth of both Prefix Search and Linear Scan. As the number of dimensions increases, the increase in the performance of the MR index structure 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 index structure decreases by 13%, while that for Prefix Search decreases by less than 1%. Similarly, precision for the MR index structure increases by 30%, while that for 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 for ϵ between 0.1 and 0.2, the Prefix Search technique examines almost the entire search space.

The subqueries in the MR index structure 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 index structure is $(log(1024) - log(16))/2 = 3$. On the other hand, for Multiple Search, the expected number of subqueries is $512/16 = 32$, more than 10 times higher.

4.2 Estimating the total cost

Although Figure 4 gives us information about CPU cost (precision determines the candidate set size, which in turn determines the number of arithmetic operations), and the I/O cost (number of disk page reads), determining the overall cost requires an estimation of the relative cost of arithmetic operations to disk page accesses. The cost of an arithmetic operation is usually much less than the cost of a disk page read. However, the cost of disk page read itself can vary based on whether the read is sequential or random. In case of random I/O, there is a high seek and rotational latency cost for each page access. However, for sequential reads, disk head is at the start of the next page to be read, thus avoiding the seek cost. As a result, the cost of reading pages in a random order is much higher than the cost of reading them sequentially. We assume this ratio to be 12 [4, 19]. (There are also several other factors like buffering and prefetching that affect the page access cost. However, we do not consider these effects in our analysis.)

We normalize all costs to the number of sequential disk pages read. The total cost of a query is then computed as follows:

Total Cost = CPU Cost + I/O Cost, where
CPU Cost = Candidate Set Size $\times$ *c*, and
I/O Cost = k $\times$ *Number of page reads*

The constant c converts the cost of computing the distance between two sequences to the cost of a sequential disk page read. It was experimentally estimated to be between 1/300 and 1/600 depending on the hardware architecture.

The constant k depends on the search algorithm: if the pages are accessed sequentially then it is 1, if the pages are accessed in random then it is 12 [4, 19]. The Prefix Search technique performs random page reads; as a result the corresponding constant is 12. In the MR index structure, a candidate set of database sequences is determined by accessing the MBRs corresponding to the last subquery. Disk accesses are then carried out to read this set of candidate sequences. Since the candidate sequences are accessed in the disk placement order, the average cost of a page read is much less than the cost of a random page read [4, 19]. We used the cost model proposed in [19] to find the cost of reading a subset of pages in sequential order. Typically, the value of k for the MR index structure lies between 2.0 and 3.0. We also validated this experimentally by reading subsets of pages in sequential order from a real disk.

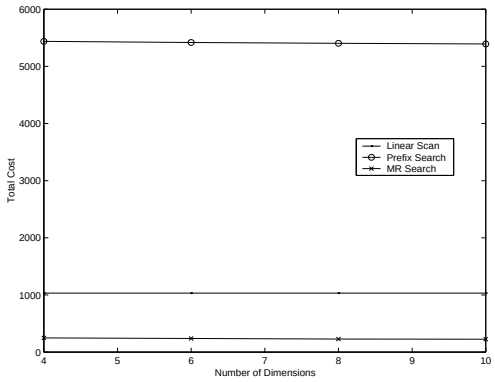

Figure 5. Total cost for queries.

The total cost comparison corresponding to Figure 4 is presented in Figure 5. It can be seen that the MR index structure performs more than 4 times better than Linear Scan, and more than 20 times better than Prefix Search. As the number of dimensions increases from 4 to 10, the performance of the MR index structure improves by 13%, while that of the Prefix Search technique improves by less than 1%. Another point we would like to observe is that the Prefix Search technique always has the highest cost among the three techniques. This is because Prefix Search reads almost every disk page in a random order. As a result, the cost of disk reads dominates CPU time. 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 variable.

5 Index compression

The MR index structure 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 the *I-adaptive* index that maintain information at a single granularity level. In order to perform the preprocessing steps of the search in memory, the index structure must fit into memory. If the index structure does not fit into mem-

ory, then the method will incur disk page reads for accessing the index. Therefore, the performance of the MR index structure could degrade if enough memory is not available. In this section, we consider index compression techniques with which 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 an extended MBR (EMBR) of the same dimensionality that tightly covers and replaces the constituent MBRs. The MR index assumes that an index node contains subsequences that start at consecutive offsets. In order to preserve this property, only consecutive MBRs can be merged into an EMBR. 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 the size of the candidate sets. Therefore, we would like to merge as many MBRs as possible into an EMBR, while minimizing the increase in volume. 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 consecutive subsequences are similar. As a result, this method merges the first r MBRs into an EMBR, second r MBRs into another EMBR, and so on. This is a very fast method. However, if consecutive MBRs are not similar, the performance drops quickly.

- *Greedy Compression* tries to minimize the increase in the total volume occupied by the MBRs at each merge operation. The algorithm chooses the two consecutive index nodes that lead to a minimal volume increase at each step until the number of index nodes is decreased by the given ratio r. This method adapts to the layout of the index nodes at each merge operation. However, since the two MBRs to be merged are determined independently at each step, some of the EMBRs can contain many more subsequences than others, leading to an unbalanced partitioning. This has the drawback that, a hit on one of the heavily populated index nodes will incur the overhead of reading large number of candidate subsequences. This decreases the precision of the index.

- The greedy compression method can be made balanced by incorporating a penalty for merging large number of subsequences. This is called *Balanced Greedy Compression*. Here, the cost of a merge operation is defined as the increase in volume multiplied by the number of subsequences in the new index node. Two index nodes that minimize this cost are selected to merge at each step. This method overcomes the problem of unbalanced partitioning.

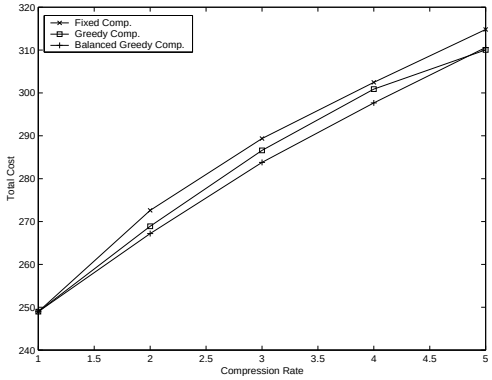

Figure 6. Total cost for different index compression techniques

5.2 Experimental results

We ran several experiments to test the performance of the compression heuristics. In these experiments, we varied the compression rate from 1 (implying no compression) to 5 (maximum compression). The results are shown in Figure 6. As evident from the figure, the Balanced Greedy heuristic has the lowest cost and Fixed Compression has the highest cost. Figure 7 compares the total cost of Linear Scan, Prefix Search, and MR Search with Balanced Greedy Compression. When the compression rate is 5, the total cost of the MR index structure is 3 times better than the cost of Linear Scan, and more than 15 times better than that of the Prefix Search technique.

Compression has another advantage. It increases the number of rows of the MR index structure, and as a result makes the index structure efficiently applicable to longer query sequences. For example, compressing the MR index to one-fifth, the number of rows of the MR index can be increased 5 times while keeping the storage cost invariant.

6 Discussion

In this paper, we considered the problem of variable length queries for subsequence matching for time sequences. We proposed a new indexing method called MR index that stores information about the data at different resolutions. An arbitrary length query is split into several subqueries corresponding to the resolutions of the index struc-

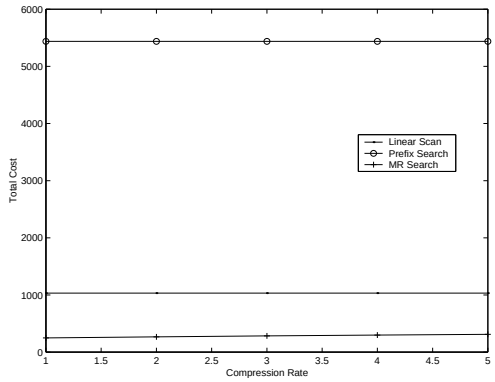

Figure 7. Total cost for different search techniques.

ture. The ranges of the subqueries are successively refined, and only the last subquery performs disk reads. As a result of these optimizations, the MR index performed 4 to 20 times better than the other methods including 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 index node. Out of these three methods, Balanced Greedy Compression has the best results since it minimizes the information loss (increase in volume) while distributing the subsequences almost evenly. The MR index performed 3 to 15 times better than other techniques 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 algorithm in Section 3 carries out a single radius refinement over all the MBRs of the database sequence. The performance of the MR index structure algorithm can be further improved by performing the range refinement decisions for subqueries locally. This can be done by computing the new range for each MBR separately. Different MBRs belonging to the same sequence would have independent refinements. 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 for B_3 at the next lower row included positions 1-300 of the sequence, then B_3 would only use the information form B_1 and B_2 in its radius refinement. In particular, radii information from boxes other than B_1 and B_2 at row R_1 would not be used. 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.

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 would be needed among the processors, except to merge the results at the end.

This paper has concentreated on range queries. It is not difficult to extend the technique to nearest neighbor queries. The minimum and the maximum distance of a given query to a database sequence can be estimated with the *MINMAX* technique [10, 18] to prune the candidate set. Finally, the necessary sequences are accessed from disk to filter and return the correct answer set.

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 *ACM 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] K.-P. Chan and A. W.-C. 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] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *ACM SIGMOD*, pages 419–429, Minneapolis MN, May 1994.

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

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *ACM 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] E. J. Keogh and M. J. Pazzani. An indexing scheme for similarity search in large time series databases. In *SSDBM*, Cleveland, Ohio, 1999.

[12] S.-L. Lee, S.-J. Chun, D.-H. Kim, J.-H. Lee, and C.-W. Chung. Similarity search for multidimensional data sequences. In *ICDE*, San Diego, CA, 2000.

[13] S. Park, J. Y. Wesley W. Chu, and C. Hsu. Fast retrieval of similar subsequences of different lengths in sequence databases. In *ICDE*, San Diego, CA, February 2000.

[14] C.-S. 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, February 2000.

[15] D. Rafiei and A. O. Mendelzon. Similarity-based queries for time series data. In *ACM 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] R. M. Rao and A. S. Bopardikar. *Wavelet Transforms Introduction to Theory and Applications.* Addison Wesley, 1998.

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

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

[20] 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.

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

[22] M. Vetterli and J. Kovacevic. *Wavelets and Subband Coding.* Prentice Hall, 1995.

[23] B.-K. Yi, H. Jagadish, and C. Faloutsos. Efficient retrieval of similar time sequences under time warping. In *ICDE 98*, pages 23 – 27, Orlando, Florida, February 1998.