

# Continually Evaluating Similarity-Based Pattern Queries on a Streaming Time Series\*

Like Gao, X. Sean Wang  
Department of Information and Software Engineering  
George Mason University, Fairfax, VA 22030, USA  
{lgao, xywang}@gmu.edu

## ABSTRACT

In many applications, local or remote sensors send in streams of data, and the system needs to monitor the streams to discover relevant events/patterns and deliver instant reaction correspondingly. An important scenario is that the incoming stream is a continually appended time series, and the patterns are time series in a database. At each time when a new value arrives (called a time position), the system needs to find, from the database, the nearest or near neighbors of the incoming time series up to the time position. This paper attacks the problem by using Fast Fourier Transform (FFT) to efficiently find the cross correlations of time series, which yields, in a batch mode, the nearest and near neighbors of the incoming time series at many time positions. To take advantage of this batch processing in achieving fast response time, this paper uses prediction methods to predict future values. FFT is used to compute the cross correlations of the predicted series (with the values that have already arrived) and the database patterns, and to obtain predicted distances between the incoming time series at many future time positions and the database patterns. When the actual data value arrives, the prediction error together with the predicted distances is used to filter out patterns that are not possible to be the nearest or near neighbors, which provides fast responses. Experiments show that with reasonable prediction errors, the performance gain is significant.

## 1. INTRODUCTION

In many applications, data streams from various sensors arrive at a system, and the system must monitor the streams to discover relevant events or patterns, and to react correspondingly. The reaction often needs to be fast each time a new value arrives. Example applications include computer network monitoring, automated reconnaissance flight control, and automated security trading. These applications call for a type of “continuous query” processing, emphasizing

\*This work was partially supported by the NSF career award 9875114.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA  
Copyright 2002 ACM 1-58113-497-5/02/06 ...\$5.00.

on fast responses. It has been realized that traditional data processing systems are not enough for continuous queries, and new techniques need to be developed [29, 7, 23, 3].

In this paper, we deal with an important scenario where the incoming stream takes the form of a continually appended time series (the streaming series), and the patterns are represented as a fixed set of time series in the database. The pattern series may be of various lengths. At each time when a new value arrives (called a time position), the system needs to find quickly, from the database, the nearest or near neighbors for the streaming series up to that time position. For example, in an automated stock exchange/monitoring application, the system may be asked to monitor the stock prices and to respond in a particular manner at any moment when the price trend shows a strong similarity (i.e., the distance is within a certain threshold) to one of the preselected price (pattern) series.

In the above stock exchange/monitoring example, if the number of preselected price series is small and the new values arrive at a slow rate, fast response may be achieved by a straightforward algorithm that scans all the preselected price series to match against the streaming series every time a new value arrives. However, when the number of preselected series is large and/or new values arrive very frequently, more efficient algorithms are needed. Existing indexing strategies do not work well because pattern series may have various lengths (see Section 5 for more details). The challenge is to develop a new strategy so that the system can respond quickly once new values reach the system.

We attack the problem by taking advantage of the fact that Fast Fourier Transform (FFT) can be used to efficiently find the cross correlations of the streaming series with the database patterns. These cross correlations are closely related to the similarity measure we choose, namely the *weighted Euclidean distance*. Using the cross correlations, we can directly derive the nearest and near neighbors of the streaming time series at *many* time positions contained in the streaming series, in a batch processing mode.

The above batch processing, however, may not deliver fast response time. Indeed, batch processing only saves overall processing time, and does so by waiting for a number of values to arrive before launching the batch processing. Since the system does not immediately process the values as they arrive, response time suffers.

In order to obtain fast response time and yet take advantage of batch process, we use prediction methods to predict future values. We use FFT to compute the cross correlations of the predicted series (*with* the already arrived values) and the database patterns, and obtain predicted distances between the streaming time series at many future time positions and the database patterns. When the actual data value arrives, the prediction error, together with the predicted distances, is used to filter out patterns that are not possible to be the nearest or near neighbors, and thus expedite the process at each time position. In some cases, we can even find out some or all answers directly using the prediction.

The above strategy is similar to the blocked access idea in cache management where a block of data values is brought into the cache when any one value in the block is accessed. This is advantageous in cache management because accessing a block costs much less than accessing all the values in the block individually and the other values in the block are likely to be useful in speeding up subsequent accesses. In our method, the “block” consists of predicted values to take advantage of batch processing and the predicted values in the “block” are useful for fast processing when new values arrive.

The effectiveness of the above strategy obviously depends on the accuracy of the prediction model. In order to evaluate its effectiveness, we perform experiments under various prediction accuracy assumptions, instead of using some fixed prediction models. Specific applications may use their particular prediction models to achieve accurate predictions [12, 16, 24, 11, 30]. Our experiments show that when the error between the predicted series and actual series is reasonably small, the performance gain of this strategy is significant.

The contribution of the paper is threefold. Firstly, we introduce a new general effective strategy, based on prediction and batch processing, to achieve better response time for continuous queries. Secondly, we show in detail how the above strategy is used on streaming time series to find similarity-based patterns, i.e., the nearest neighbor or near neighbors. And lastly, we demonstrate through experiments the effectiveness of the method under various conditions.

The remainder of the paper is organized as follows. In Section 2, we formally define our problem and show how FFT can be used to provide us with the batch processing capability. We discuss in Section 3 the details of continuous query processing using the batch processing and prediction. In Section 4, we report our experiments and the effectiveness of our method. We discuss related work in Section 5 and conclude the paper in Section 6 with some summary remarks and future research directions.

## 2. PROBLEM FORMULATION AND BATCH PROCESSING USING FFT

In this section, we start with introducing some basic notions, and defining precisely the continuous queries we are dealing with. We then present the batch processing technique based on FFT.

We assume that all (one-dimensional) time series are sampled at an *equal* time interval. Without loss of general-

Symbol	Meaning
$x, y, \dots$	time series
$x[i, j]$	subseries of $x$ between positions $i$ and $j$ , inclusively
$\mathcal{IS}$	streaming time series (query series)
$\mathcal{PS}$	predicted time series
$F_i$	the $i^{\text{th}}$ pattern or feature series. $l_i + 1$ is its length.

Table 1: Some frequently used symbols.

ity, we take the interval as the unit and thus all time series are represented as a sequence of real numbers, with the position of the sequence corresponding to the sampling time. We further assume that the first sample is always taken at time 0. Hence, a time series  $x$  takes the form of  $\langle x[0], x[1], \dots, x[l], \dots \rangle$ . A time series  $x$  is *finite* if it ends at certain  $l \geq 0$ , and we then say that the series has a *length*  $l + 1$ . A time series is infinite if no such  $l$  exists. If  $x$  is a time series, we use  $x[i, j]$  to denote the finite time (sub)series  $\langle x[i], x[i + 1], \dots, x[j] \rangle$ , where  $0 \leq i \leq j$  are integers with  $j$  less than the length of  $x$  if  $x$  is finite.

There are situations where the values in time series are not sampled at a fixed interval. When this is the case, interpolation may be necessary to even define the notion of distance of a pair of time series. How the interpolation is done and other related issues are beyond the scope of this paper.

Given two series of length  $l + 1$ , we use a weighted Euclidean distance to measure the distance between them and the weight is the square root of the length. More specifically, given two finite series  $x$  and  $y$  of length  $l + 1$ , the distance between  $x$  and  $y$  is defined as:

$$D(x, y) = \sqrt{\sum_{s=0}^l (x[s] - y[s])^2 / (l + 1)}.$$

In our scenario, we assume that we have a (fixed) set of finite time series, called pattern series,  $F_i$ . The series  $F_i$  is of length  $l_i + 1$ . These are the pattern or feature series that the system needs to watch out for. In the stock exchange/monitoring example mentioned in the introduction, the preselected historical price trends are these pattern series. We also assume there is an infinite time series, which is our streaming series, denoted  $\mathcal{IS}$ ; and at time position  $p \geq 0$ , the value  $\mathcal{IS}[p]$  comes into the system. Hence, at time position  $p \geq 0$ , the streaming series is a finite series of length  $p + 1$ . In the stock exchange/monitoring example, the stock prices of the market is this infinite streaming series. At time  $p$ , the stock price at the time arrives at the system.

The continuous query we are dealing with is that at each time  $p$ , find the nearest neighbor or near neighbors of the “current”  $\mathcal{IS}$  among the pattern series  $F_i$ . To precisely define these neighbor concepts, we first have:

**Definition** Let  $F_i$  be a pattern series in the database with length  $l_i + 1$ . For a given  $p \geq l_i$ , the *distance between  $\mathcal{IS}$  and*

pattern  $F_i$  at position  $p$ , is defined as  $D(\mathcal{IS}[p - l_i, p], F_i)$ . If  $p < l_i$ , then this distance is defined as positive infinite.

Intuitively, the distance of  $\mathcal{IS}$  and  $F_i$  at position  $p$  is from comparing  $F_i$  with the values of  $\mathcal{IS}$  at position  $p$  and looking backward for  $l_i$  steps. With the above distance definition, we define neighbors of the streaming series at position  $p$ .

**Definition** Let  $p \geq 0$  be an integer. Given a real number  $h \geq 0$ , a pattern  $F_i$  is the *nearest neighbor of  $\mathcal{IS}$  at position  $p$*  if for all other patterns  $F_j (j \neq i)$ ,  $D(\mathcal{IS}[p - l_i, p], F_i) < D(\mathcal{IS}[p - l_j, p], F_j)$ ; and  $F_i$  is an  *$h$ -near neighbor of  $\mathcal{IS}$  at position  $p$*  if  $D(\mathcal{IS}[p - l_i, p], F_i) \leq h$ .

In order to simplify the presentation, in the remainder of the paper, we assume that *no two finite distances (two reals) will ever be exactly the same*. This assumption can be lifted without much problem, and is not pursued in this paper.

**Definition** A *continuous query* on a streaming time series is one of the following standing requests: (1) *For each position  $p$ , find the nearest neighbor of  $\mathcal{IS}$  at position  $p$* ; or (2) *For each position  $p$ , find the  $h$ -near neighbors of  $\mathcal{IS}$  at position  $p$ , where  $h \geq 0$  is a real number, called *threshold*.*

We assume we are interested only in position  $p$  such that the distance between  $\mathcal{IS}$  and  $F_i$  at position  $p$  is defined for each  $F_i$ . In other words, we are only interested in time position  $p$  such that  $p \geq \max\{l_i | F_i\}$ , where  $l_i + 1$  is the length of  $F_i$ . This is not a severe restriction at all since the streaming time series at most of the time is much longer than any of the pattern series. Lifting this restriction is not difficult, either, but is not pursued in this paper.

It is possible that at position  $p$ , the answer of the  $h$ -near neighbor query returns an empty set. This is when the distance of  $\mathcal{IS}$  and  $F_i$  is greater than  $h$  for each  $F_i$ .

The naive method of processing the query is to find the distance of  $\mathcal{IS}$  and  $F_i$  at each time position  $p$  for each  $F_i$ . As mentioned earlier, in many situations, this naive method may not be enough to derive sufficiently fast response. In the remainder of the section, we derive a batch processing method to calculate distance of  $\mathcal{IS}$  and  $F_i$  at multiple time positions.

Consider the definition of  $D(\mathcal{IS}[p - l_i, p], F_i)$ . We have ( $l_i + 1$  is the length of  $F_i$ ):

$$\begin{aligned} & (l_i + 1) * D^2(\mathcal{IS}[p - l_i, p], F_i) \\ &= \sum_{s=0}^{l_i} \mathcal{IS}[(p - l_i) + s]^2 + \sum_{s=0}^{l_i} F_i[s]^2 \\ & \quad - 2 \sum_{s=0}^{l_i} \mathcal{IS}[(p - l_i) + s] * F_i[s] \end{aligned} \quad (1)$$

Once the values are known for all three terms on the right hand side of the equation,  $D(\mathcal{IS}[p - l_i, p], F_i)$  is easily obtained. The first term on the right hand side can be computed incrementally as  $p$  moves forward to the next position.

The second does not change and can be pre-computed. As for the third, we can see that each multiplicand  $\mathcal{IS}[(p - l_i) + s]$  will shift forward to the next value as  $p$  becomes  $p + 1$ , while the multiplier  $F_i[s]$  keeps the same. The sum of products of current computation has little relationship with the last one; and is the most time-consuming item. Fortunately, as is shown below, this term is exactly double of the value of a *cross correlation* of  $\mathcal{IS}$  and  $F_i$ , and Fast Fourier Transform (FFT) can serve to compute multiple cross correlations efficiently in a batch mode.

Given an infinite time series  $x$  and a finite series  $y$  of length  $l + 1$ , the *cross correlation function* of  $x$  and  $y$  is defined as

$$\text{CCorr}_{x,y}[d] = \sum_{s=0}^l x[d + s] * y[s], \quad d = 0, 1, 2, \dots \quad (2)$$

In the above,  $d$  is the number of shifts between the  $x$  and  $y$ , and is called the *lag* parameter. It follows from the above definition that only the values  $x[d], \dots, x[d + l]$  are used in calculating the cross correlation of  $x$  and  $y$  of lag  $d$ . Consider equation (1), the third term is exactly  $2 * \text{CCorr}_{\mathcal{IS}, F_i}[p - l_i]$ .

There are a number of fast methods to calculate cross correlations via convolutions [31, 5]. Here, we use the *Circular Correlation Theorem* [25, 20], a property of the Discrete Fourier Transform (DFT), to perform the calculation. Before stating the theorem, we need some auxiliary notation.

**Definition** (1) Let  $x$  and  $X$  be two series of length  $N + 1$ . Then  $x \xleftrightarrow[N+1]{\text{DFT}} X$  denotes the fact that  $X$  is the  $(N + 1)$ -point DFT of  $x$ . Clearly, if this is the case,  $x$  is the  $(N + 1)$ -point *inverse DFT* of  $X$ . (2) Given two finite series  $x$  and  $y$  of length  $N + 1$ , then  $\text{CirCCorr}_{x,y}$  denotes the *unnormalized circular cross correlation* sequence, defined as

$$\begin{aligned} \text{CirCCorr}_{x,y}[d] &= \sum_{s=0}^N x[(d + s) \bmod (N + 1)] * y[s], \\ & \quad d = 0, 1, 2, \dots, N \end{aligned}$$

We now give the Circular Correlation Theorem [25, 20].

**Theorem** Let  $x, X, y$  and  $Y$  be finite series of length  $N + 1$ . Assume  $x \xleftrightarrow[N+1]{\text{DFT}} X$  and  $y \xleftrightarrow[N+1]{\text{DFT}} Y$ . Then

$$\text{CirCCorr}_{x,y} \xleftrightarrow[N+1]{\text{DFT}} \langle X[0] * \overline{Y[0]}, \dots, X[N] * \overline{Y[N]} \rangle,$$

where  $\overline{Y[s]}$  is the complex-conjugate of  $Y[s]$ .

If  $N = 2^k - 1$  for some positive integer  $k$ , then we may use the Fast Fourier Transform (FFT) algorithm to calculate DFT and inverse DFT, and hence the circular cross correlations. More specifically, given time series  $x$  and  $y$  of length  $N + 1$ , we first use FFT to calculate  $X$  and  $Y$ . We then generate the sequence  $\langle X[0] * \overline{Y[0]}, \dots, X[N] * \overline{Y[N]} \rangle$  and use inverse FFT to obtain the circular cross correlations.

By using the circular cross correlation function, we can now calculate the cross correlations of the streaming series  $\mathcal{IS}$

and a pattern series  $F_i$ . Indeed, assume the length of  $F_i$  is  $l_i + 1$ , and let  $N \geq l_i$ . We define  $x = \mathcal{IS}[p_s - l_i, p_s - l_i + N]$ , and  $y$  as the time series  $\langle F_i[0], \dots, F_i[l_i], 0, \dots, 0 \rangle$ , where 0 appears  $N - l_i$  times. Since  $y[l_i + 1] = \dots = y[N] = 0$ , it follows from the definitions that  $\mathbf{CCorr}_{x,y}[d] = \mathbf{CirCCorr}_{x,y}[d]$  for each  $d$  with  $0 \leq d \leq N - l_i$ . This is because that the “last” circular cross correlation that is still the same as the cross correlation is when the last value of  $x$  “aligns” with  $y[l_i]$ , i.e., when the lag is up to  $N - l_i$ ; and hence, we obtain the cross correlations from  $\mathbf{CCorr}_{x,y}[0]$  up to  $\mathbf{CCorr}_{x,y}[N - l_i]$  from the circular cross correlations. By the definition of  $x$  and  $y$ , we see that we have the cross correlations from  $\mathbf{CCorr}_{\mathcal{IS}, F_i}[p_s - l_i]$  up to  $\mathbf{CCorr}_{\mathcal{IS}, F_i}[p_s - l_i + N - l_i]$ . By the comment after Equation (2), we can obtain the distances of  $\mathcal{IS}$  and  $F_i$  at positions from  $p_s$  to  $p_s + N - l_i$  in a batch mode.

For a set of pattern series  $F_i$ , we will use a single time series  $x$  for all the pattern series in the batch process described in the previous paragraph. In order to do so, we take  $l_{max} = \max\{l_i | l_i \text{ is the length of pattern } F_i\}$ , and  $N = 2^k - 1 \geq l_{max}$  for some integer  $k$ . For each pattern series  $F_i$ , we define  $y_i$  as  $F_i$  padded with  $N - l_i$  zeros, and we let  $x = \mathcal{IS}[p_s - l_{max}, p_s - l_{max} + N]$ . With the same argument as in the paragraph above, we see that the batch processing, applied to  $x$  and  $y_i$  for each  $i$ , can give us the distances between  $\mathcal{IS}$  and  $F_i$  for each  $i$  at positions  $p_s, \dots, p_s + N - l_{max}$ .

In summary, the batch processing has the following inputs and outputs:

#### Batch Process

**Input:**  $\mathcal{IS}[p_s - l_{max}, p_s - l_{max} + N]$  and all pattern series  $F_i$ , where  $N = 2^k - 1 \geq l_{max}$  for some  $k$ , and  $l_{max} + 1$  is the maximum length of all the pattern series.

**Output:** Distances between  $\mathcal{IS}$  and  $F_i$  for each  $i$  at positions  $p_s, \dots, p_s + N - l_{max}$ .

Experiments show that such a batch processing is much faster than a naive algorithm that calculates each single distance separately, and furthermore, the greater the value  $N$  is, the more savings this batch method provides.

### 3. CONTINUOUS QUERY WITH PREDICTION

The batch processing given in the previous section saves overall computation time, but does not directly give fast response time. In this section, we show how this batch processing strategy can be used to deliver fast responses.

The problem of the batch processing is that the response time suffers since it waits for  $N - l_{max}$  values to come before launching the batch process. If  $N$  is chosen so that  $N - l_{max}$  is small, the batch processing does not save too much time, and may actually be slower than the naive method due to its overhead. On the other hand, if  $N - l_{max}$  is large, then the response time will be poor.

We will use the prediction to solve the above slow response time problem. In practical applications, most time series have some trends or patterns that can be used to successfully predict future values in a streaming time series at most of the time. Much research has been dedicated to this subject and

provided many prediction models and algorithms for specific applications [12, 16, 24, 11, 30]. An  $n$ -step ahead prediction model predicts the values for the next  $n$  time positions.

By using predicted values instead of actual values from the streaming series, we may use the FFT batch method to calculate the *predicted distances* between the streaming series and the pattern series at many future positions. (The number of the future positions to be used largely depends on the accuracy of the prediction model because the farther into the future, the less accurate the predictions.) When the actual value arrives, the prediction error will be known, and will be used together with the predicted distances to obtain the neighbors of the streaming series. We call such a method *Continuous Querying with Prediction* or **CQP** for short.

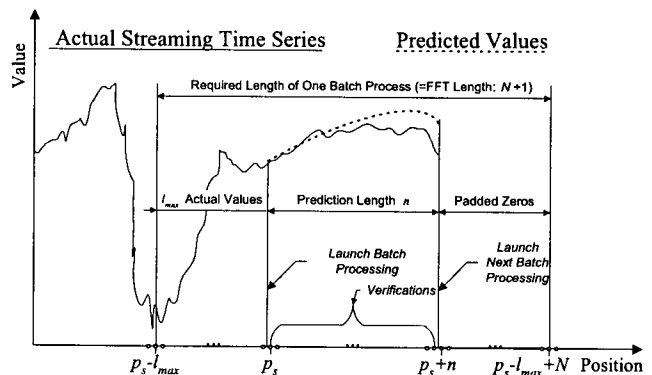


Figure 1: Prediction and batch processing.

Figure 1 illustrates the **CQP** process. Assume currently we are considering time position  $p_s$ , i.e., the continuous query has been answered for up to and including time position  $p_s - 1$ , and the value at position  $p_s$  has not arrived yet. At this moment, we obtain  $n$ -step ahead prediction, and depict the situation in the figure. We use *prediction length* to denote the maximum prediction step achieved. At time before  $p_s$ , the prediction model provides  $n$  looking forward predicted values for time positions  $p_s, p_s + 1, \dots, p_s + n - 1$  (the dotted curve). We use  $\mathcal{PS}$  to denote the time series formed by the values of  $\mathcal{IS}$  up to  $p_s - 1$ , and take the  $n$  predicted values, and pad the series with infinite number zeros towards the end. We call  $\mathcal{PS}$  the *predicted series*. More precisely,

$$\mathcal{PS} = \langle \mathcal{IS}[0], \dots, \mathcal{IS}[p_s - 1], P_0, \dots, P_{n-1}, 0, \dots, 0, \dots \rangle,$$

where  $P_i, i = 0, \dots, n - 1$ , are the predicted values.

We can now use the batch processing described at the end of Section 2 to calculate the predicted distances. In the batch process, instead of using  $\mathcal{IS}$ , we use  $\mathcal{PS}$  as the input. Also, we choose a value  $N$  that is no less than  $n + l_{max}$ , as illustrated in Figure 1. We use  $\mathcal{PS}[p_s - l_{max}, p_s - l_{max} + N]$  as the input (along with the pattern series) to the batch process, and we will obtain the distances between  $\mathcal{PS}$  and  $F_i$  for positions  $p_s, \dots, p_s + N - l_{max}$ . Since values after  $p_s + n - 1$  are not really predicted values (they are padded zeros), we actually obtain the (predicted) distances between  $\mathcal{PS}$  and  $F_i$  for positions  $p_s, \dots, p_s + n - 1$ .

Step	Action
1.	From the next position $p_s$ , generate $n$ predicted values, and form the <i>predicted series</i> $\mathcal{PS}$ .
2.	Use the <b>batch process</b> on $\mathcal{PS}$ with all pattern series $F_i$ to generate <i>predicted distances</i> for positions $p_s, \dots, p_s + n - 1$ .
3.	For each time position $p$ , within the range from $p_s$ to $p_s + n - 1$ , when the actual value arrives, do: <ul style="list-style-type: none"> <li>3.1. Use the <i>prediction error</i>, i.e., the distance between the predicted values and actual values, and the predicted distances to partition the patterns <math>F_i</math> into three categories: <ul style="list-style-type: none"> <li>Category (1): those that satisfy the query,</li> <li>Category (2): those that cannot satisfy the query, and</li> <li>Category (3): those that are in neither category (1) nor category (2).</li> </ul>           We call the patterns in category (3) as <i>candidate patterns</i>.            (See two subsections below for details on these two types of queries.)         </li> <li>3.2. Verify among the candidate patterns to find (further) answers to the query            (This is done directly using the formula in Equation (1).)</li> </ul>
4.	Change $p_s$ to be $p_s + n$ , and perform steps 1-4 repeatedly.

Figure 2: Continuous querying with prediction (CQP).

Once the actual value at a time position from  $p_s$  to  $p_s + n - 1$  arrives for the streaming time series, we can use the prediction error to find the neighbors of the streaming time series. This can be done very efficiently if the prediction errors are small. This step of the process depends on the type of query we are considering, namely whether it is the nearest neighbor or the  $h$ -near neighbor query; and we will illustrate the details in the following two subsections. Once we use up the predicted values, we launch another batch process.

We summarize the **CQP** algorithm with  $n$ -step prediction in Figure 2.

The verification procedure, Step 3.2 in Figure 2, can be done with a direct application of the distance definition. The cost of this step is in proportion to the number of the candidates. If the candidate list has many patterns, then this step will be costly. On the other hand, if the candidate list is small, verification can be fast, and sometime can even be skipped. (Indeed, if the query is asking for the nearest neighbor and the candidate list only consists of one pattern, then this pattern has to be the nearest neighbor and no verification is necessary.) Experiments show that the verification step is the main cost of **CQP** algorithm.

### 3.1 CQP for the nearest neighbor

In this subsection, we develop the details for Step 3.1 in Figure 2 for the case of finding the nearest neighbor at each time position  $p$ . The basic task is to find the “candidate” pattern or feature series that must be considered. In other words, we want to filter out all the pattern series that cannot be the nearest neighbor based on the predicted distances and the prediction errors.

Consider a time position  $p$ , which is between  $p_s$  and  $p_s + n - 1$  (see Figure 1). Through the batch processing of Step 2, the predicted distance  $D(\mathcal{PS}[p - l_i, p], F_i)$  is already known for each  $F_i$ . Also, it is easy to incrementally calculate the prediction error between the predicted series and the actual streaming time series  $\mathcal{IS}$  at position  $p$  with the length of

$l_i + 1$ , i.e.,  $D(\mathcal{PS}[p - l_i, p], \mathcal{IS}[p - l_i, p])$  is known once the data values of  $\mathcal{IS}$  up to position  $p$  have arrived.

With the above predicted distances and prediction errors, we may derive upper and lower bounds for the distance of the actual streaming series with each pattern series at position  $p$ . Indeed, because we use a variation of the Euclidean distance, the following triangular relationship holds:

$$|D(x, F_i) - D(x, y)| \leq D(y, F_i) \leq |D(x, F_i) + D(x, y)| \quad (3)$$

where  $x = \mathcal{PS}[p - l_i, p]$  and  $y = \mathcal{IS}[p - l_i, p]$ . This triangular relationship is depicted in Figure 4.

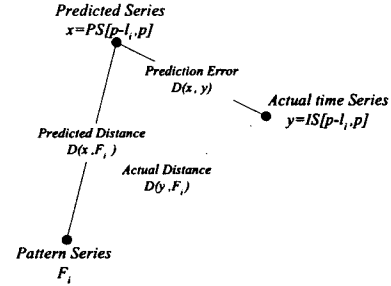


Figure 4: Triangular relationship among distances.

Inequality (3) holds for each pattern  $F_i$  at position  $p$ . To simplify our algorithm, we take the maximum prediction error to calculate the above bounds, i.e., let  $\max D(\mathcal{PS}_p, \mathcal{IS}_p) = \max\{D(\mathcal{PS}[p - l_i, p], \mathcal{IS}[p - l_i, p])\}$  for all  $l_i$ . Given pattern  $F_i$ ,  $D(\mathcal{PS}[p - l_i, p], F_i) + \max D(\mathcal{PS}_p, \mathcal{IS}_p)$  is the *derived upper bound*, and  $D(\mathcal{PS}[p - l_i, p], F_i) - \max D(\mathcal{PS}_p, \mathcal{IS}_p)$  is the *derived lower bound*. Note that we can use this derived lower bound partly because  $a - b \leq |a - b|$  for all numbers  $a$  and  $b$ .

Figure 5 shows the derived *upper bound* and the *lower bound* at time position  $p$  in the increasing order of the predicted distances  $D(\mathcal{PS}[p - l_i, p], F_i)$ ,  $i = 0, \dots, m$ . To simplify the

INPUT	$D(\mathcal{PS}[p - l_i, p], F_i)$ for all pattern $F_i$
OUTPUT	Candidate pattern list.
METHOD:	<p>Step 1 <b>QuickSort</b> the list of <math>F_i</math> based on the values <math>D(\mathcal{PS}[p - l_i, p], F_i)</math>, <math>i = 0, 1, 2, \dots, m</math>. Assume the list obtained is <math>F_{s_0}, \dots, F_{s_m}</math>.</p> <p>Step 2 Let <math>\max D(\mathcal{PS}_p, \mathcal{IS}_p) = \max\{D(\mathcal{PS}[p - l_i, p], \mathcal{IS}[p - l_i, p])\}</math> for all <math>l_i</math>, and let <math>\min Up = \max D(\mathcal{PS}_p, \mathcal{IS}_p) + D(\mathcal{PS}[p - l_{s_0}, p], F_{s_0})</math>.</p> <p>Step 3 Find the last pattern <math>F_{s_i}</math> in the list obtained in Step 1 such that <math>D(\mathcal{PS}[p - l_{s_i}, p], F_{s_i}) - \max D(\mathcal{PS}_p, \mathcal{IS}_p)</math> is less than or equal to <math>\min Up</math>. Call this pattern <math>F_{s_L}</math> and <b>Return</b> <math>F_{s_0}, \dots, F_{s_L}</math>.</p>

Figure 3: Find the candidate patterns for the nearest neighbor at position  $p$ .

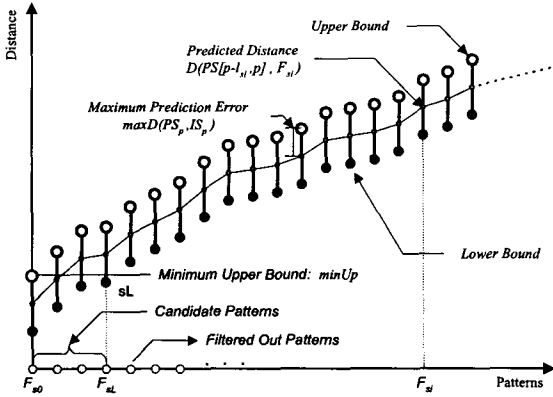


Figure 5: Candidates for the nearest neighbor.

illustration, we reassign the subscripts of pattern series in the order of these distances with  $\{s_0, s_1, \dots, s_m\}$ . Hence,  $D(\mathcal{PS}[p - l_{s_i}, p], F_{s_i}) < D(\mathcal{PS}[p - l_{s_j}, p], F_{s_j})$  for all  $s_i < s_j$ . (Again note we assume that no two distances can be exactly the same.) The actual distances of  $\mathcal{IS}$  and  $F_i$  at position  $p$  must be between the corresponding lower and upper bounds (inclusively).

Our goal is to find the nearest neighbor at position  $p$ . Since  $F_{s_0}$  has the smallest predicted distance, it has the smallest derived upper bound among all patterns since the derived upper bounds are the predicted distances for all  $F_i$  plus the same number  $\max D(\mathcal{PS}_p, \mathcal{IS}_p)$ . Note that if the derived lower bound of the distance from  $\mathcal{IS}$  and a feature  $F_{s_i}$  is greater than this smallest upper bound, then  $F_{s_i}$  cannot be the nearest neighbor since definitely  $F_{s_0}$  must be closer to  $\mathcal{IS}$  than  $F_{s_i}$ . On the other hand, if the derived lower bound of the distance from  $F_{s_i}$  is not greater than this smallest upper bound, then we cannot be sure which of the two,  $F_{s_0}$  or  $F_{s_i}$ , is closer to  $\mathcal{IS}$ . In this case,  $F_{s_0}$  and  $F_{s_i}$  both need to be considered further, and we call them *candidate patterns*. Figure 5 illustrates the candidate list for the nearest neighbor at position  $p$ .

The procedure to find the candidate list of the nearest neighbor is given in Figure 3. Consider the three categories in Figure 2. If  $s_L = 0$ , then we know that  $F_{s_0}$  is the only answer, thus in category (1). Otherwise, category (1) is empty,

and  $F_{s_{(L+1)}, \dots, F_{s_m}}$  are in category (2) and  $F_{s_0}, \dots, F_{s_L}$  are the candidate patterns in category (3).

Note that the **QuickSort** step does not involve the actual time series value  $\mathcal{IS}[p]$  and can be performed before  $\mathcal{IS}[p]$  comes. The computation of the upper bound and lower bound will involve the error distance from actual time series to the predicted series at each position. It's easy to see that this computation can be implemented incrementally and can be done very efficiently. So this procedure of finding the candidate patterns needs very little CPU time.

**Proposition** Algorithm **CQP** in Figure 2 with Step 3.1 as implemented in Figure 3 correctly processes the continuous query for the nearest neighbor.

### 3.2 CQP for the $h$ -near neighbors

The batch process on evaluating this query is exactly as what we discussed earlier and summarized in Figure 2. In this subsection, we develop Step 3.1 of Figure 2 for the  $h$ -near neighbor query. Unlike the query of finding the nearest neighbor that has exactly one pattern series as the output, the  $h$ -near neighbor query may have 0 or any number of patterns as the output.

We observe that the triangular relationship in Inequation (3), as well as shown in Figure 4, among the three distances still holds for each position  $p$ . We will use the same derived *upper bound* and *lower bound* as those in Subsection 3.1. We sort the patterns  $F_i$  based on their distances to the predicted series  $\mathcal{PS}$ , and illustrate the derived upper and lower bounds derived from the triangular relationship in Figure 6.

Unlike the nearest neighbor query where the derived upper bound can be used to filter out patterns that cannot be the answer, the threshold  $h$  for the  $h$ -near neighbor query can be any real number that is no less than 0.

In order to find the  $h$ -near neighbors, we classify pattern series into three categories: a pattern series  $F_i$  is in *category (1)* if the threshold  $h$  is greater than or equal to its derived upper bound,  $F_i$  is in *category (2)* if the threshold  $h$  is less than its derived lower bound, and  $F_i$  is in *category (3)* if the threshold is between its derived lower bound (inclusive) and derived upper bound (exclusive).

INPUT	$D(\mathcal{PS}[p - l_i, p], F_i)$ for all pattern $F_i$
OUTPUT	Candidate pattern list and some $h$ -near neighbors.
METHOD:	<p>Step 1 <b>QuickSort</b> the list of <math>F_i</math> based on the values <math>D(\mathcal{PS}[p - l_i, p], F_i)</math>, <math>i = 0, 1, 2, \dots, m</math>. Assume the list obtained is <math>F_{s_0}, \dots, F_{s_m}</math>.</p> <p>Step 2 Let <math>\max D(\mathcal{PS}_p, \mathcal{IS}_p) = \max\{D(\mathcal{PS}[p - l_i, p], \mathcal{IS}[p - l_i, p])\}</math> for all <math>l_i</math>, and let <math>\max U_p = \max D(\mathcal{PS}_p, \mathcal{IS}_p) + D(\mathcal{PS}[p - l_{s_m}, p], F_{s_m})</math>, <math>\min Low = D(\mathcal{PS}[p - l_{s_m}, p], F_{s_m}) - \max D(\mathcal{PS}_p, \mathcal{IS}_p)</math>.</p> <p>Step 3 Find the first pattern <math>F_{s_i}</math> in the list obtained in Step 1 such that <math>D(\mathcal{PS}[p - l_{s_i}, p], F_{s_i}) + \max D(\mathcal{PS}_p, \mathcal{IS}_p) &gt; h</math>. Call this pattern <math>F_{s_U}</math>. If <math>F_{s_U}</math> does not exist, then <b>Return</b> (all patterns as near neighbors).</p> <p>Step 4 Find the last pattern <math>F_{s_i}</math> in the list obtained in Step 1 such that <math>D(\mathcal{PS}[p - l_{s_i}, p], F_{s_i}) - \max D(\mathcal{PS}_p, \mathcal{IS}_p) \leq h</math>. Call this pattern <math>F_{s_L}</math>. If <math>F_{s_L}</math> does not exist, then <b>Return</b> (no <math>h</math>-near neighbors).</p> <p>Step 5 <b>Return</b> candidate list: <math>F_{s_U}, \dots, F_{s_L}</math>, and <math>h</math>-near neighbors: <math>F_{s_0}, \dots, F_{s_{(U-1)}}</math> if <math>s_U \neq s_0</math>.</p>

Figure 7: Find the  $h$ -near neighbors and candidate patterns for the near neighbors at position  $p$ .

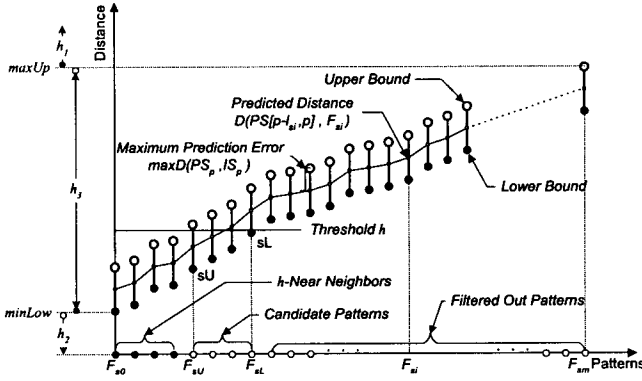


Figure 6: The  $h$ -near neighbors and candidates.

Clearly, patterns in category (1) are those that are definitely  $h$ -near neighbors since the actual distance cannot be greater than  $h$ . Likewise, patterns in category (2) are those that are definitely not  $h$ -near neighbors (the “filtered out” patterns) since the actual distance must be greater than  $h$ . Patterns in category (3) are those we are not sure; these are what we call *candidate patterns*, and need to be verified by computing their actual distance to find out which of the candidates are indeed  $h$ -near neighbors. These categories correspond exactly to the categories in Figure 2.

It is not difficult to find the above three categories as shown in Figure 7. We first sort the pattern series in the increasing order by their distances to the predicted series. We then try to find the first pattern series (call it  $F_{s_U}$ ) that has the derived upper bound greater than  $h$ . If no such pattern exists, then all pattern series are in category (1), i.e., all patterns are  $h$ -near neighbors. It is easily seen that this is the case when  $h$  is in the range  $h_1$  in Figure 6, i.e.,  $h$  is greater than or equal to the maximum upper bound  $\max U_p = \max D(\mathcal{PS}_p, \mathcal{IS}_p) + D(\mathcal{PS}[p - l_{s_m}, p], F_{s_m})$ .

We then try to find the last pattern series (call it  $F_{s_L}$ ) that has the derived lower bound less than or equal to  $h$ . If no such pattern exists, then all pattern series are in category (2), i.e., no pattern series are  $h$ -near neighbors. It is easily seen that this is the case when  $h$  is in the range  $h_2$  in Figure 6, i.e.,  $h$  is less than  $\min Low$ , defined as  $D(\mathcal{PS}[p - l_{s_0}, p], F_{s_0}) - \max D(\mathcal{PS}_p, \mathcal{IS}_p)$ .

If  $F_{s_U}$  and  $F_{s_L}$  both exist, then we have the case  $\min Low \leq h < \max U_p$ . This corresponds to the case when  $h$  is in the range  $h_3$  in Figure 6. In this case, the pattern series  $F_{s_0}, \dots, F_{s_{(U-1)}}$  are in category (1), i.e., they are definitely  $h$ -near neighbors, the patterns  $F_{s_{(L+1)}}, \dots, F_{s_m}$  are in category (2), i.e., the filtered out patterns, and patterns  $F_{s_U}, \dots, F_{s_L}$  are the *candidate* patterns. Whether a candidate pattern is an  $h$ -near neighbor or not needs to be verified. The verification uses the distance formula directly in our algorithm.

Figure 7 summarizes all the above steps in locating the patterns in the three categories.

Note that in the case of  $h$ -near neighbor query, the number of candidates to be verified is unrelated to the number of answers. Indeed, in the above, the numbers of patterns in category (1) and category (3) are not related.

**Proposition** Algorithm CQP in Figure 2 with Step 3.1 as implemented in Figure 7 correctly processes the continuous query for the  $h$ -near neighbors.

#### 4. PERFORMANCE EVALUATION

In this section we study the performance of CQP algorithm through experiments. The experiments are coded with the programming language C++ and the FFT algorithm used is FFTW [10]. Experiments are performed on a dedicated desktop computer (Dell Dimension 4100 with 256 MB memory and PentiumIII 766 CPU).

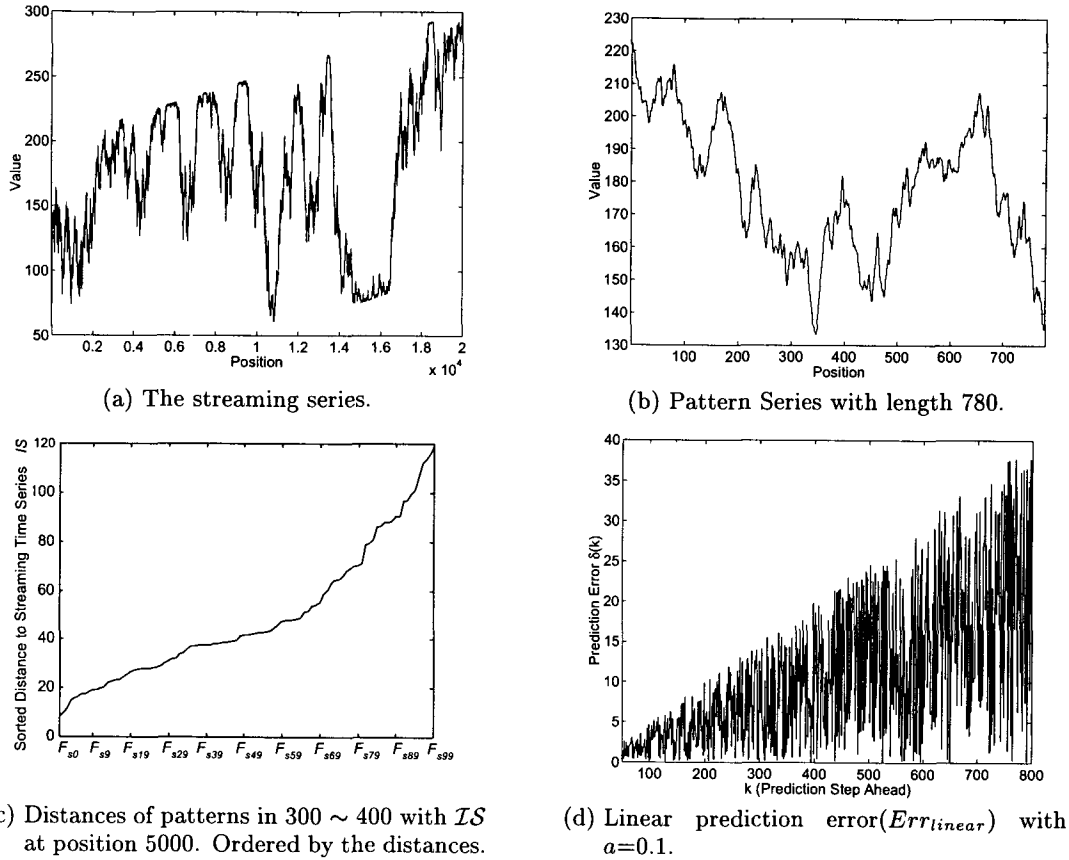


Figure 8: Synthetic data and prediction error models used in the experiment.

In order to control the test environment, we use synthetic data. We generate the streaming time series with a function of the random-walk series, defined as  $IS[i] = 100 * (\sin(0.1 * RandomWalk[i]) + 1 + i/20000)$ ,  $i = 0, \dots, 19999$ , where  $RandomWalk[0 : 19999]$  is a random-walk series. The streaming time series generated and used in the experiment is shown as Figure 8(a).

Four sets of pattern series with length ranges of 300 ~ 400, 500 ~ 600, 700 ~ 800 and 300 ~ 800, respectively, are tested in the experiments. These patterns are also synthetically generated and each is made similar to a portion of the streaming time series. Specifically, pattern series are generated by taking random positions in the streaming series and obtaining subsequences of given lengths from those positions. Each pattern set consists of 100 patterns with their lengths being uniformly distributed, i.e., for pattern series set 300 ~ 800, the lengths are  $\{300, 305, 310, \dots, 795\}$ . A sample pattern series is shown in Figure 8(b). In Figure 8(c), we show the distances of the  $IS$  with each of the pattern series in the set 300 ~ 400 at a sample position. The patterns are ordered by the distance to  $IS$  at the position.

In order to control the experiments in terms of prediction errors, we also generate predicted series based on the streaming series. Suppose at the position  $p_s - 1$ , we need to launch batch process. Assume we use an  $n$ -step prediction model, and for each integer  $k$  ( $1 \leq k \leq n$ ), we denote as  $\delta[k]$  the

absolute error between the streaming time series  $IS$  and the predicted series  $PS$  at position  $p_s - 1 + k$ . This error  $\delta[k]$  tends to be bigger when the looking forward step  $k$  becomes larger up to  $n$ . In order to simulate this, we assume that  $\delta[k]$  is a uniformly distributed variable at each prediction step  $k$  with the range of this variable growing increasingly as  $k$  becomes greater.

Three kinds of prediction error models are implemented in the experiment. The absolute error of the first one increases in the order of  $O(\sqrt{k})$ , which is defined as  $Err_{sqrt}[k] = a * RAND * \sqrt{k}$ ; the second follows the linear increasing trend with the form of  $Err_{linear}[k] = a * RAND * k$ ; and the third error model is  $Err_{square}[k] = a * RAND * k^2$ , which increases in the order of  $O(k^2)$ . In these functions,  $k$  is the prediction step,  $RAND$  is a uniformly distributed random variable and its values are within  $-0.5$  to  $0.5$ , and  $a$  is named the *error control* which can scale up the prediction error as needed. In the experiments we report here, we fix  $a = 1$  in the  $Err_{sqrt}$  model, and  $a = 0.1$  in the  $Err_{linear}$  and  $Err_{square}$  models. Figure 8(d) shows a sample linear prediction error function with  $a = 0.1$  and prediction steps from 50 to 800. Intuitively, with the same  $a$  value and prediction step  $k$ ,  $Err_{sqrt}$  model gives the best prediction accuracy, meaning  $PS$  is very close to the actual streaming time series  $IS$ ;  $Err_{linear}$  gives a moderate prediction accuracy and  $Err_{square}$  yields the worst prediction accuracy.



FFT Length Pattern Set (lengths)	Prediction Length								
	50	100	200	300	400	500	600	700	800
300 ~ 400	512	512	1024	1024	1024	1024	1024	2048	2048
500 ~ 600	1024	1024	1024	1024	1024	2048	2048	2048	2048
700 ~ 800	1024	1024	1024	2048	2048	2048	2048	2048	2048
300 ~ 800	1024	1024	1024	2048	2048	2048	2048	2048	2048

Table 2: Pattern set, prediction length and FFT length used in the experiments.

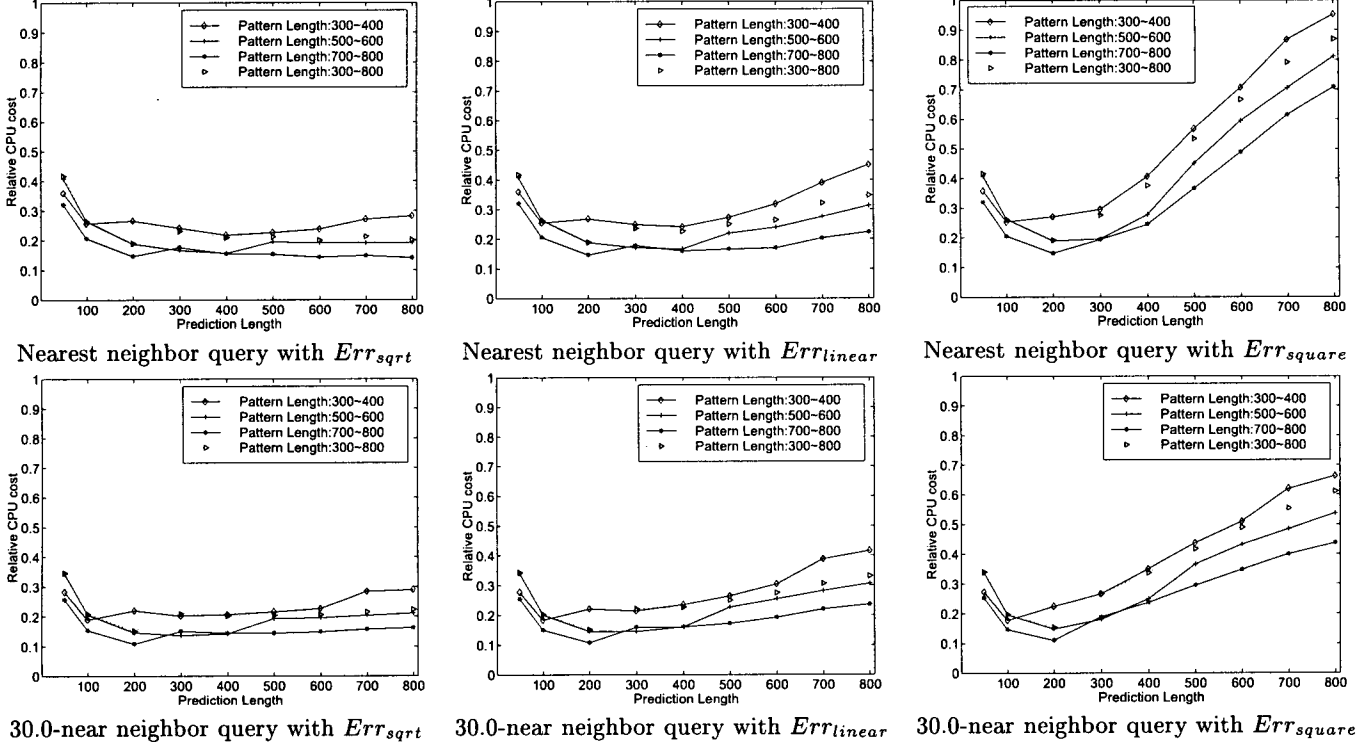


Figure 9: Relative CPU cost to evaluate query with different prediction error models.

The following queries are evaluated in the experiments:

- Q1: Find the nearest neighbor of the incoming stream time series at each time position.
- Q2: Find the 30.0-near neighbors of the streaming time series at each position  $p$ .

We define the *CPU cost* as the averaged computation time at each time position (hence this is the average response time). We show the *relative CPU cost* as the above CPU cost relative to the CPU cost of the naive method, which computes the distances at each time position directly using the distance formula. Note the CQP algorithm favors the query at the beginning positions in one prediction period, while the query at the tail end may need more time to perform the verification because of the larger prediction errors.

In our experiments with a given prediction error model, the relative CPU cost is a function of two parameters: the choice

of the pattern sets and the prediction length. The FFT length is picked as the value  $2^k$ , for some  $k$  such that  $2^k$  is the least value no less than the maximum length of pattern series plus the prediction length. All combinations tested in the experiments are shown in Table 2. Each combination is also done with each of the three prediction error models  $Err_{sqrt}$ ,  $Err_{linear}$  and  $Err_{square}$ .

Figure 9 shows the performance of CQP. Efficiency is mostly dependent on prediction models. As long as the error of the  $n$ -step prediction is small enough, the CQP algorithm can achieve good performance and outperform the naive method. Compared with the naive method, with the given prediction model and the prediction length, CQP algorithm works better as the pattern series lengths get longer. This is because that only a little more time is needed to perform the FFT on a longer time series, while the cost of the naive method is proportional to the length of the patterns. However, if the prediction model and the pattern set are fixed, the relative cost grows as the prediction length be-

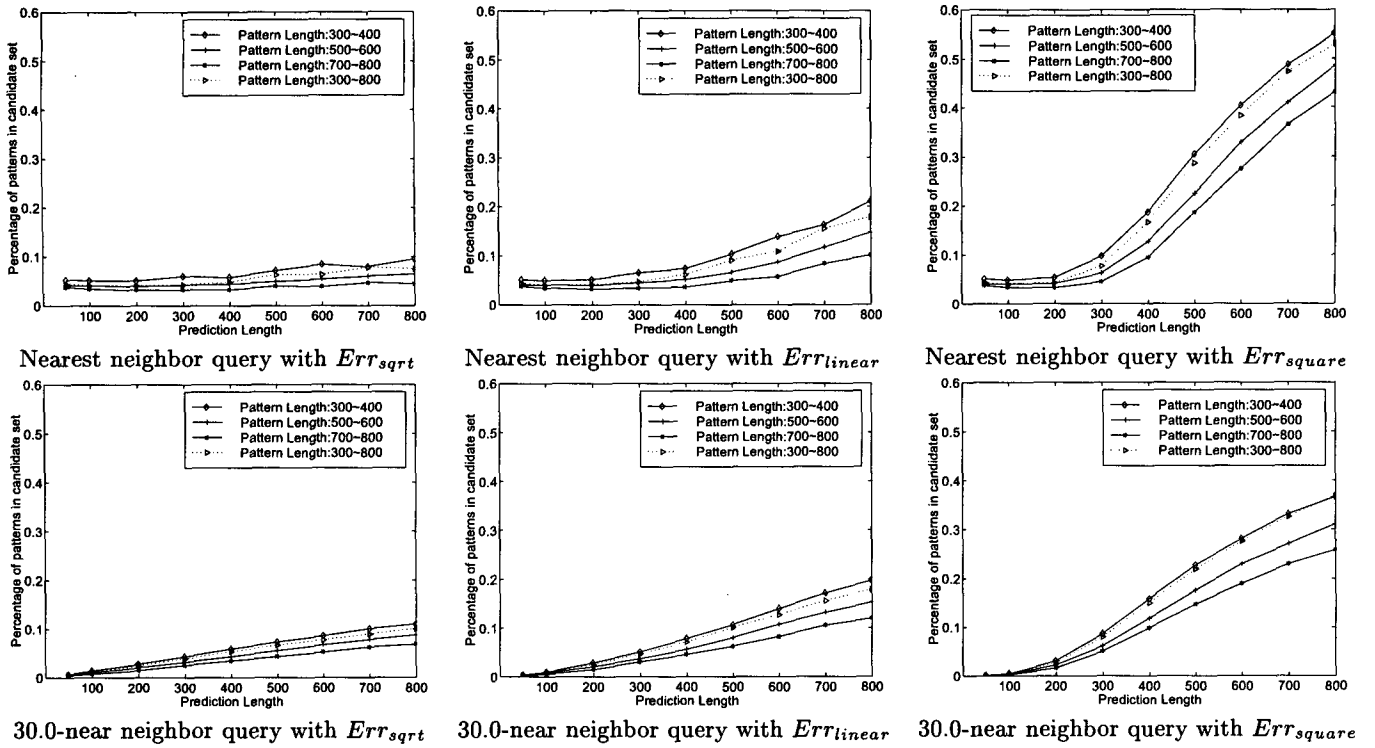


Figure 10: Percentage of patterns in the candidate set.

comes longer due to the increasing prediction error between the actual and predicted time series.

The wall clock time for the whole process (with all the time positions) takes approximately 29 to 65 seconds with the naive method depending on the data sets. Here we assume that a value in the streaming time series arrives as fast as we finish handling the previous value. Hence, the response is about 1.5 to 3.3 milliseconds on average for the naive method. The clock time of our method is much shorter than this, and can be derived from Figure 9. These times are real response times if the data values do not come faster than the speed of the processing.

The **CQP** algorithm will not be useful if the prediction error is too large. As the CPU cost of the nearest neighbor query in Figure 9 with  $Err_{square}$  model shows, the performance drops quickly when the prediction length increases. So choosing a good prediction model as well as the prediction length is critical to use this algorithm.

As mentioned above, the performance of **CQP** mostly depends on the accuracy of the prediction. The FFT batch processing contributes to the performance as well, but is not very sensitive to the accuracy of prediction. In order to see the relationship of performance and prediction power, we show the average percentage of pattern series in the candidate set over all the time positions. See Figure 10.

Note that for the nearest neighbor query, the answer of the query is always one pattern at each time position. However, for the near-neighbor queries, the patterns in the answer de-

pends on the threshold value. In our experiments, we used threshold  $h = 30.0$ , and the average number of patterns in the answer at each position is 14 out of 100 patterns. Note that for the  $h$ -near neighbor query, the number of candidates may be different from the number of patterns in the answer, and is dependent on the prediction model used. For example, the number of actual 30.0-near neighbors at all 20,000 time positions is totally 282,300, but **CQP** algorithm only verifies totally 64,938, 77,518, and 130,493, respectively, for the three prediction models with a prediction length of 300. Indeed, as discussed in Section 3, some patterns, category (1), may already be in the answer without verification.

It should also be mentioned that the CPU cost of **CQP** algorithm to find the  $h$ -near neighbors is not sensitive to the  $h$  value. Instead, it is more dependent on the distance distribution of the patterns to the streaming series at each position. For example, in Figure 8(c), if the maximum prediction error is 10, then the number of candidates is 36 for 30.0-near neighbors, while only about 10 for 80.0-near neighbors. There is no direct relationship between the number of candidates (thus the CPU cost) and the query threshold  $h$ .

From Figure 9, we can also find that there exists a best prediction length to best outperform the naive method for a given pattern set. The averaged CPU cost at each position to perform the batch processing may decrease as the prediction length becomes longer, but the verification procedure cost will increase at the same time due to the increased prediction error. The existence of the best prediction length is the trade-off between these two factors.

## 5. RELATED WORK

Long standing queries over a changing database have long attracted the attention of researchers. Perhaps Terry et al. [29] were the first ones who introduced the notion of “continuous queries” for a class of queries that are issued once and then run “continuously” over databases. They proposed an incremental approach to evaluate the queries on append-only databases with only new results returned to the user. Also, Parker et al. (e.g., [21]) worked on answering queries on data streams. A similar concept, but termed “continual query”, was also presented and studied in the work of Liu et al. [17, 18].

Recently, due to the ever growing use of information systems in many new types of applications, continuous queries have again become the focal point of several research projects. Chen et al. reported, in [7, 6], the design of NiagraCQ system in which incremental query evaluation method was no longer restricted to append-only data sources. Babu and Widom [3], and Madden and Franklin [19] also reported system architecture and related issues for dealing with continuous queries.

In this paper, we study a particular but important type of continuous query, namely similarity-based pattern queries on streaming time series. Instead of using the idea of incrementally updating the query result, we use a new prediction-based approach. We believe many continuous queries will be able to take advantage of prediction and provide faster response time.

Our work is also related to that on time series similarity search. However, work has been concentrated on stationary data sources. Recent work in this area is categorized into two general approaches [13]. The first is to map time sequence into frequency domain [1, 9, 26, 15], generally with a Discrete Wavelet Transform, and then use the significant part of the coefficients to index the original time series. The second approach [2, 28] performs the operation in time domain directly. This work is also related to indexing high dimensional data. A recent survey can be found in [4]. However, in the above approaches, the database series are supposed to have the same lengths. In contrast, we deal with (pattern) time series with variable lengths. If the database pattern series are all of the same length, we believe our prediction-based method can be combined with the use of indexing structures in the literature.

Work has also been done in dealing with variable lengths and subseries, which may be adapted to deal with series with various lengths. Subseries matching [9, 14] in time-series databases is a more difficult problem. The time series stored in databases have variable length and are longer than the query sequences. The query is to find the subsequences in the database that have a similar pattern in the same length as the query series. Generally, the approach is to split the series into shorter ones and to index these shorter ones to answer queries. In general, splitting long series into shorter ones works only for near-neighbor queries, not for nearest neighbors. In this paper, we deal with both nearest neighbor and near neighbor queries on time series with different lengths.

## 6. CONCLUSION

In this paper, we introduced a new strategy of processing continuous queries, namely continuous query with prediction. We showed this strategy works well in the scenario where we need to quickly find similarity-based patterns from a streaming time series. We detailed the algorithms and experiments used for this scenario. The experiments are done using synthetic data to control the environment and to see when the strategy works.

For the scenario we consider, we may easily extend the queries to find nearest  $k$  neighbors. Another improvement we may consider is that the time to launch the batch process can be adjusted based on the size of the candidate patterns. Indeed, if the prediction model is not good enough, the number of candidates will increase dramatically at the positions of the tail end of one prediction period. In this case, we should consider re-predict the time series and re-launch the batch processing. In this paper, we only considered the launch of the prediction and batch processing at fixed time positions. An additional extension to our scenario we may consider in the future is the case when the number of patterns is large, and when disk accesses are necessary. In this case, we believe prediction can be helpful as well.

In our algorithm, at certain time positions, the FFT batch process is launched. At these positions, the response time will be slower than the other time positions, where no batch process is used. For a faster response time at these positions, we may consider to perform the batch process before all the predicted values are exhausted. That is, we may overlap batch process with the normal steps. How well this strategy works is an interesting future research direction.

The approach in our work follows the traditional similarity models that rely on pointwise Euclidean distance [1, 9, 26, 8]. Non-Euclidean metrics have also been used to compute the similarity for time series. Perng [22] proposed “Landmarks Similarity” as a general model to measure the similarity among time series. Rafiei [27] proposed a class of linear transforms on the Fourier transformations of original series. Several important notions of similarity can be expressed with this class and the corresponding queries can be efficiently implemented on top of R-tree index. Huang [13] proposed another approach that transforms time series into symbol strings and then builds a suffix tree to index all suffixes of the symbol strings. As a future research direction, it will be interesting to incorporate prediction-based method with these non-Euclidean distances.

It will be most interesting to extend our strategy to other kinds of continuous queries. The critical ingredients for this strategy to work is the prediction capability and the use of prediction to increase the speed of processing when actual values arrive. However, it should be noted that the prediction need not be precise *all* the time. If the prediction is not good at certain period of time, the strategy may still work well. Indeed, we only need the have the predictions relatively precise at relatively most of the time. How these “relativities” are measured and how much is sufficient depends on the particular application considered.

## 7. REFERENCES

- [1] R. Agrawal, C. Faloutsos, and A. N. Swami. Efficient similarity search in sequence databases. In *Proceedings of the 4th FODO*, pages 69–84, 1993.
- [2] R. Agrawal, K.-I. Lin, H. S. Sawhney, and K. Shim. Fast similarity search in the presence of noise, scaling, and translation in time-series databases. In *The VLDB Journal*, pages 490–501, 1995.
- [3] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record*, Sept. 2001.
- [4] S. Berchtold and D. A. Keim. High-dimensional index structures, database support for next decade's applications (tutorial). In *SIGMOD Conference*, 1998.
- [5] C. Burrus and T. Parks. *DFT/FFT and Convolution Algorithms*. John Wiley and Sons, 1985.
- [6] J. Chen, D. J. DeWitt, and J. F. Naughton. Design and evaluation of alternative selection placement strategies in optimizing continuous queries. In *ICDE Conference*, 2002.
- [7] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proc. of the ACM SIGMOD Conference*, pages 379–390, 2000.
- [8] K. K. W. Chu and M. H. Wong. Fast time-series searching with scaling and shifting. In *Proc. of the 18th ACM PODS 1999, Philadelphia*, pages 237–248, 1999.
- [9] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *ACM SIGMOD Conference*, pages 419–429, 1994.
- [10] M. Frigo and S. G. Johnson. FFTW: C subroutine library for computing the Discrete Fourier Transform (DFT). On-line. <http://www.fftw.org/>, 2001.
- [11] T. V. Gestel, J. Suykens, D.-E. Baestaens, A. Lambrechts, G. Lanckriet, B. Vandaele, D. B. Moor, and J. Vandewalle. Financial time series prediction using least squares support vector machines within the evidence framework. *IEEE Transactions on Neural Networks*, 12(4):809–821, 2001.
- [12] L. Györfi, G. Lugosi, and G. Morvai. A simple randomized algorithm for sequential prediction of ergodic time series. *IEEE Transactions on Information Theory*, 45(7):2642–2650, 1999.
- [13] Y.-W. Huang and P. S. Yu. Adaptive query processing for time-series data. In *Proceedings of the 5th International Conference of Knowledge Discovery and Data Mining*, pages 282–286, 1999.
- [14] T. Kahveci and A. K. Singh. Variable length queries for time series data. In *ICDE 01*, pages 273–282, 2001.
- [15] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proceedings of ACM SIGMOD Conference on Management of Data*, pages 151–162, 2001.
- [16] I. Kim and S.-R. Lee. A fuzzy time series prediction method based on consecutive values. In *Fuzzy Systems Conference Proceedings*, volume 2, pages 703–707, 1999.
- [17] L. Liu, C. Pu, R. S. Barga, and T. Zhou. Differential evaluation of continual queries. In *International Conference on Distributed Computing Systems*, pages 458–465, 1996.
- [18] L. Liu, C. Pu, and W. Tang. Continual queries for Internet scale event-driven information delivery. *IEEE TKDE*, 11(4):610–628, 1999.
- [19] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE Conference*, 2002.
- [20] A. Oppenheim and R. Schaffer. *Digital Signal Processing*. Prentice-Hall, Inc., 1975.
- [21] D. S. Parker, R. R. Muntz, and H. L. Chau. The tangram stream query processing system. In *ICDE Conference*, 1989.
- [22] 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*, pages 33–42, 2000.
- [23] B. Plale and K. Schwan. Optimizations enabled by a relational data model view to querying data streams. In *Proc. of 15th International Parallel and Distributed Processing Symposium*, 2001.
- [24] S. Policker and A. Geva. A new algorithm for time series prediction by temporal fuzzy clustering. In *Proceedings. 15th International Conference on Pattern Recognition*, volume 2, pages 728–731, 2000.
- [25] A. D. Poularikas, editor. *The transforms and applications handbook*. CRC Press LLC, 2000.
- [26] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 13–25, 1997.
- [27] D. Rafiei and A. O. Mendelzon. Querying time series data based on similarity. *IEEE TKDE*, 12(5):675–693, 2000.
- [28] H. Shatkay and S. B. Zdonik. Approximate queries and representations for large data sequences. In *ICDE*, pages 536–545, 1996.
- [29] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 321–330, 1992.
- [30] L. Wang, K. K. Teo, and Z. Lin. Predicting time series with Wavelet packet neural networks. *Proc. International Joint Conference on Neural Networks*, 3:1593–1597, 2001.
- [31] S. Winograd. Some bilinear forms whose multiplicative complexity depends on the field of constants. *Mathematical Systems Theory*, 10:169–180, 1977.