

Improving the Performance of Continuous Queries on Fast Data Streams: Time Series Case*

Like Gao, X. Sean Wang
{lgao, xywang}@gmu.edu

Department of Information and Software Engineering, George Mason University, Fairfax, Virginia

Abstract

A continuous query on a data stream is a standing request that is processed whenever a new data value arrives. For such a query, response time and throughput are two important performance indicators. This paper concerns the performance of continuous queries under different streaming rates. When the streaming rate is low and system can start evaluating the query without delay for each newly arrived data value, response time is the dominant performance indicator. When the streaming rate is high, both response time and throughput become key issues since the query cannot get evaluated for each arrived data value without creating ever longer response time. This paper studies the scenario of a streaming time series. Every time a new value arrives, the query is to locate, in a time series database, the nearest neighbor of the streaming series. The paper uses a pre-fetching strategy that predicts and prepares for the next evaluation based on the previous one. Experiments show that the strategy decreases response time and increases throughput.

1 Introduction

In various emerging applications, sensors are deployed in fields or installed on moving objects. The goal is to recognize and detect events from the incoming sensor data, and to provide situation awareness for systems in order to facilitate smart decision and quick reaction. Obviously, the performance of such a recognition and detection subsystem is critical to the success of the applications. The purpose of this paper is to study strategies that enhance the system performance.

We study the scenario where a sensor sends back a stream of values in the form of a time series. The recognition task is as follows: at each time when a new value arrives, the system needs to find the nearest neighbor of the time series, formed by using a fixed number of most recent values, from among the time series in a large database.

The scenario mentioned above appear in many applications, ranging from stock monitoring and target tracking. For example, a target identification may rely

on the movement of the target. A database may be available with different movement signatures of many different types of vehicles. The system is to monitor the movement of a target in order to find out the type of vehicles that the target resembles the most, in terms of movement patterns. Obviously, movement can be modeled as time series of positions, and it is necessary to continuously evaluate the nearest neighbor queries on time series database.

Two performance measures are important. The first is the response time, i.e., the time between the arrival of a value to the time the nearest neighbor is found. The second is the throughput, i.e., the number of times that the nearest neighbors are indeed found.

When data values come into the system at too fast a rate, response time will degenerate quickly as the time goes by if we insist on finding the nearest neighbor for each and every newly arrived value. Indeed, if the processing time of nearest neighbor search is longer than the time interval between two successive values, the response time will get successively longer. In this case, the system may abandon the search and move on to the next data value, i.e., a “drop” occurs. The more often the drop occurs, the worse the throughput will be. The goal of the system is to have the least number of drops and the fastest response.

To obtain better response time and lower drop ratio, we use a pre-fetching strategy. When there is time before the next data value arrives, the system will pre-fetch necessary disk pages into the main memory. Since in our scenario, successive values from the data stream have certain continuity, this preparation is quite effective. The response time is reduced greatly when the streaming rate is low. When the streaming rate is high, the strategy also works well resulting in increased throughput *and* decreased response time.

We conducted experiments to study the above pre-fetching method against a method that is a direct extension of the traditional algorithms. Experiments confirm that our method is superior in all cases and enhances the performance greatly.

The contribution of the paper is in two aspects. Firstly, we study the performance of continuous queries under different streaming rates. Secondly, we study the performance gain of pre-fetching for the con-

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

tinuous nearest neighbor queries.

The remainder of the paper is organized as follows. In Section 2, we precisely define the problem and introduce some basic notions and notation. In Section 3, we outline the algorithm with pre-fetching and the one without it. We report our experimental results in Section 4, and discuss related work in Section 5. In Section 6, we conclude the paper with some final remarks.

2 Preliminary

In this section, we introduce some basic concepts and definitions. We will then give the definitions of *drop ratio* and *response time* of continuous queries.

Definition A *streaming time series*, denoted \mathcal{S} , is an infinite number-sequence. Its values are obtained by sampling the underlying process at a fixed time interval and arrive at the query system sequentially.

The above definition says that the values of a streaming time series are sampled with the same interval, while they may come to the query system at different rates and thus each value will be associated with its arrival time. Without loss generality, we assume the first value of \mathcal{S} is sampled at time 0, and the $(i + 1)^{th}$ value is sampled at time i . If we use $\mathcal{S}[i]$ to denote the sampled value of \mathcal{S} at time i , then $\mathcal{S} = \langle \mathcal{S}[0], \mathcal{S}[1], \mathcal{S}[2] \dots, \mathcal{S}[i], \dots \rangle$. Correspondingly, from the point of view of the query system, these values will arrive at time $t_0, t_1, t_2, \dots, t_i, \dots$. The values of \mathcal{S} that arrive between time t_i and t_j , inclusively, form a finite time series, denoted $\mathcal{S}[i : j]$, i.e., $\mathcal{S}[i : j] = \langle \mathcal{S}[i], \mathcal{S}[i + 1], \dots, \mathcal{S}[j] \rangle$. The number of values in $\mathcal{S}[i : j]$ is $j - i + 1$. We assume a fixed integer L throughout the paper. When we mention an arrival time t_i , if not stated explicitly, we assume $i \geq L - 1$, i.e., at least L values have arrived. At time t_i when the i^{th} value arrives, the most recent L values form a sub-series, namely $\mathcal{S}[i - L + 1 : i]$, and the system is to find the nearest neighbor (in terms of Euclidean distance) of $\mathcal{S}[i - L + 1 : i]$ from a time series database. We assume further that each database series is of length L , and we denote $\mathcal{S}[i - L + 1 : i]$ as \mathcal{T}_{t_i} when L is understood.

Definition Given an integer $L > 0$, the *continuous nearest neighbor query* for a streaming time series \mathcal{S} is the standing request that asks for the nearest neighbor of \mathcal{T}_{t_i} at each time t_i when $\mathcal{S}[i]$ arrives ($i \geq L - 1$) from the database.

In real world situations when the stream data come very fast, the evaluation of the query at one data arrival time t_i may not finish before the next one or more data values arrive. Response time for these next values will be longer than their processing time since some

waiting occurred, and this causes more delays as time goes by. In order to keep the overall response time reasonable, some strategy needs to be adopted. One possibility is that we abort the evaluation to avoid postponing the query processing for the successive data. When this happens, we say a *drop* occurred. A drop strategy is to maintain the query process to move forward smoothly. Dependent on user requirements, there can be many different drop strategies. In this paper, we assign a response time upper bound, called the *response time threshold* and denoted Θ , and we have the following *limited response strategy*.

Definition Given the response time threshold Θ , the *limit response strategy* is a drop strategy so that if the query processing for \mathcal{T}_{t_i} cannot be finished before $t_i + \Theta$, then the evaluations for the data values arrived between t_i and $t_i + \Theta$, inclusively, will all be dropped.

Once a drop occurs (for t_i and the drop decision made at $t_i + \Theta$), this strategy clears up the waiting queue, and guarantees that each answered query has the response time no more than the threshold.

Note that the limited response strategy does not mean that each time new value arrives, the query gets the same Θ time to process. For example, if Θ is greater than the interval between two successive values, the query does not have that much time to process the second value. Indeed, the response time for processing the value arrived at t_i starts counting at t_i no matter when the processing for it actually starts.

Definition Given a streaming time series \mathcal{S} , and integers $L \leq i \leq j$, where L is the fixed integer, the *drop ratio* of a continuous query of \mathcal{S} between t_i and t_j , denoted \mathcal{D}_{ij} , is the ratio of times that the processing is dropped during this period, i.e., $\mathcal{D}_{ij} = \sum_{s=i}^j \sigma(t_s) / (j - i + 1)$, where $\sigma(t_s)$, called *drop indicator*, is 1 if processing at t_s is dropped, or 0 otherwise.

For our limited response strategy, when Θ is too small, 100% drop ratio may occur. This would mean that we have to set up Θ somewhat bigger to allow time to process the query. In this paper, we assume Θ is big enough so the drop ratio is always smaller than 100%, i.e., the query will get some answers for the period of time in question.

Definition Given a streaming time series \mathcal{S} , integers $L \leq i \leq j$ and a drop strategy, the *response time of a continuous nearest neighbor query* between t_i and t_j , denoted $t_{\mathcal{R}_{ij}}$, is the average response time during this period, that is

$$t_{\mathcal{R}_{ij}} = \frac{\sum_{s=i}^j t_{r_s} (1 - \sigma(t_s))}{\sum_{s=i}^j (1 - \sigma(t_s))}$$

where t_{r_s} is the response time to evaluate the nearest neighbor query of \mathcal{T}_{t_s} , and $\sigma(t_s)$ is the drop indicator.

Note t_{r_s} is unknown if the processing at t_s is dropped, but this doesn't affect $t_{\mathcal{R}_{ij}}$ since $1 - \sigma(t_s) = 0$.

The response time $t_{\mathcal{R}_{ij}}$ measures the average performance of the query at the times when the query has answers. For the limited response strategy with threshold Θ , it is easy to see that $t_{\mathcal{R}_{ij}} \leq \Theta$.

Note that with this strategy, drop ratio is more important an indicator than the response time, since we know that each answered query must be finished within the response time threshold. For example, given the response time threshold of one second, an algorithm that yields 80% drop ratio, with 0.5 seconds response time, will be considered worse than another algorithm that achieves only 10% drop ratio but one second response time. This is because in this case, both algorithms satisfy the one second response time threshold, while the first drops considerably more than the second.

3 Continuous query algorithms

In this section, we will first give and discuss the straightforward way to evaluate the continuous nearest neighbor queries, which is the direct extension of the traditional algorithms based on index techniques. We then propose the continuous nearest neighbor query algorithm with pre-fetching.

-
- 1 **(Nearest neighbor search in the feature space)**: transform the query time series into a point in the feature space and issue the nearest neighbor search in this space.
 - 2 **(Determining the threshold)**: calculate the real distance from the query series to the "nearest neighbor" found from step 1 in the original space.
 - 3 **(Range query)**: use this real distance as the threshold to issue a range query in feature space to find the near neighbors of the query series in the feature space, which are called *candidates*.
 - 4 **(Verification)**: evaluate the real distances from each *candidate* to the query series, and thus find the actual nearest neighbor.
-

Figure 1: Traditional algorithm

Before we move to the continuous nearest neighbor query for a streaming time series, we review the traditional algorithms [5, 14] dealing with the nearest neighbor queries of stationary time series. The commonly used approaches involve dimensionality reduction techniques [9] that map the database series from the original space into low dimensional points and the spatial indices [8] that are built on these points. There are many options to choose the reduction meth-

ods and the indices, while the query evaluation algorithms based on these techniques are similar and can be generalized as the four steps in Figure 1.

The *Direct* algorithm

The continuous nearest neighbor queries for streaming time series have a unique property, that is, the querying series at current time is very similar to the one at the previous time. This can be observed directly from the fact that the current querying series is formed by shifting each value of the last querying series, and appending a new value to its tail. In real world cases, the data that arrive at two successive times are close to each other, and we can expect that two successive querying series are also similar. This similarity between two querying series leads to the fact that they are very likely to result in the same nearest neighbor.

```

initialize the time index  $i = L - 1$ ;
//no definition of the nearest neighbor if  $i < L - 1$ 
do {
  if  $t_i > t_c$ , where  $t_c$  is the system clock time
    • wait for the new data  $\mathcal{S}[i]$  to come.
  else {
    • start searching for the nearest neighbor of  $\mathcal{T}_{t_i}$ 
    • two cases:
      Case 1) search is finished before  $t_i + \Theta$ :
        report the nearest neighbor of  $\mathcal{S}$  at  $t_i$ 
        and update  $i$  with  $i + 1$ .
      Case 2) search is not finished before  $t_i + \Theta$ :
        break the search at time  $t_i + \Theta$ . Let  $l$  be the
        greatest index that  $t_l < t_i + \Theta$ , output "drop"
        for query from  $t_i$  up to  $t_l$  and update
         $i$  with  $l + 1$ .
    } (end of else) } (end of do loop)

```

Figure 2: The *Direct* algorithm

Since it is very likely that the query at t_i results in the same result of the query at t_{i-1} , we can use this information to refine the threshold (step 2 in Figure 1) for the range query. This is done by calculating the distance from current querying series \mathcal{T}_{t_i} to the nearest neighbor of $\mathcal{T}_{t_{i-1}}$, comparing the distance found with the traditional approach (step 1 and 2 in Figure 1) and taking the smaller one as the threshold.

The *Direct* algorithm is the traditional algorithm with the limited response strategy. Note the step "searching for the nearest neighbor of \mathcal{T}_{t_i} " in Figure 2 is the same as in the traditional algorithms, except we take the nearest neighbor at time t_{i-1} as "reference" as discussed earlier if query processing at t_{i-1} is not dropped. Details are omitted due to space limitation.

Some resources used by one evaluation are very likely to be used by the next. We assume the operating system will cache these resources and enhance

the performance.

The *Pre-fetch* algorithm

At the beginning of each loop in Figure 2, there are two cases. One is that the query has used up all arrived data and the *Direct* algorithm will wait for the next value to come. The other case is that there is one or more querying series in a waiting queue.

For the first case, we should use this waiting time to pre-process the next querying series. In this paper, we predict the next querying series and evaluate the query as if it is the real one. This step fetches the necessary pages into main memory and thus makes the processing faster when the real value arrives.

In this paper, we use the following prediction method. Consider \mathcal{T}_{t_i} before time t_i . The first $L - 1$ values $\mathcal{S}[i - L + 1], \dots, \mathcal{S}[i - 1]$ to form the querying series \mathcal{T}_{t_i} have already arrived, and we only need to predict one value $\mathcal{S}[i]$ to get the predicted querying series $\hat{\mathcal{T}}_{t_i}$. The greater the length L is, the less impact the accuracy of the predicted value. So the prediction need not to be very precise, and in this paper we simply use the value $\mathcal{S}[i - 1]$ as the prediction for $\mathcal{S}[i]$.

For real world applications, we may apply various prediction models to forecast the next incoming value, either linear or non-linear predictive models. In this paper, we concentrate on the algorithm itself and do not dig into the detail on how to select or build the prediction model, although definitely, a better prediction model will improve the performance of our proposal.

Once we get the predicted querying series $\hat{\mathcal{T}}_{t_i}$, we can issue a nearest neighbor search with $\hat{\mathcal{T}}_{t_i}$. The evaluation with this querying series must be aborted immediately when the real data value $\mathcal{S}[i]$ comes, and thus there is no delay to begin the evaluation of actual querying time series \mathcal{T}_{t_i} .

For the second case, that is, when there are querying series in the waiting queue, i.e., \mathcal{T}_{t_i} is waiting, there is no need to predict $\hat{\mathcal{T}}_{t_i}$ and issue a search for the nearest neighbor of $\hat{\mathcal{T}}_{t_i}$ since all values of \mathcal{T}_{t_i} have already arrived. The query process needs to keep on processing the waiting querying series without involving the prediction.

Combining these two cases, we give the proposed algorithm in Figure 3, named *Pre-fetch* algorithm.

If the query evaluation of the predicted series $\hat{\mathcal{T}}_{t_i}$ is finished before the new data $\mathcal{S}[i]$ comes, then its pre-loaded memory pages will be used in the evaluation of \mathcal{T}_{t_i} directly. Even if only part of steps of it are finished, these steps are still useful for the evaluation of the actual querying series.

As mentioned before, the resources can be shared by two successive evaluations of the continuous query. It is easy to see that the more similar the two query-

```

initialize the time index  $i = L - 1$ ;
do {
  if  $t_i > t_c$  where  $t_c$  is the system clock time
    • predict the querying series  $\hat{\mathcal{T}}_{t_i}$  for time  $t_i$ ;
    • start searching for the nearest neighbor of  $\hat{\mathcal{T}}_{t_i}$ ;
    • if the search is not finished before  $\mathcal{S}[i]$  comes,
      break the search as soon as  $\mathcal{S}[i]$  comes;
    • wait for the new data  $\mathcal{S}[i]$  to come if it hasn't come;
      else do the same as the corresponding part in Figure 2
} end of do loop

```

Figure 3: The *Pre-fetch* algorithm

ing series are, the more resources they will share. The simple prediction method used in this paper, that is, forming $\hat{\mathcal{T}}_{t_i}$ by repeating the last value of $\mathcal{T}_{t_{i-1}}$, guarantees that $\hat{\mathcal{T}}_{t_i}$ is closer to the ongoing querying series \mathcal{T}_{t_i} than series $\mathcal{T}_{t_{i-1}}$ is. This fact means that the evaluation of the predicted querying series can provide more shareable resources for the ongoing evaluation.

If the stream rate is very fast, then the chance to process the predicted querying series is small. In this case *Pre-fetch* algorithm will not help too much. As the stream rate becomes slower, the chance and the time will become higher and larger. We can expect that *Pre-fetch* algorithm will outperform the *Direct* algorithm in this case. The experiments in the next section confirm this trend.

Both *Direct* and *Pre-fetch* algorithms are based on index techniques. There is another way to use the idle time before the new data comes, that is, we can calculate the partial distances for the next querying time series with each pattern series. Specifically, given the pattern length of L , at current time while waiting for data to come, we calculate the partial distance only with the first $L - 1$ values of the next querying series. When the actual value comes, we can easily find out the nearest neighbor by taking into account both these partial distances and the differences of the L -value. Although this method is straightforward and can yield very fast response time if the partial distances can be obtained in time, the cost to find out these partial distances is high and very close to the sequential scan method. Thus, it's only useful when the stream rate is low. Our experiment will show that this straightforward strategy is worse than the index based algorithms when stream ratio is high.

4 Experimental results

In this section, we use experiments to demonstrate the performance improvement of the *Pre-fetch* algorithm over the *Direct* algorithm. The experiments consists of two parts, the drop ratio comparison and response time comparison. We use different stream rate and

response time threshold. For simplicity, the stream rate is fixed during one continuous query evaluation.

The data set consists of 10^6 random walking time series with length of 128, and total size of data is about 500MB. We use Rtree to build the index on the first 6 coefficients of the wavelet transformation of the database series. The streaming series is also a random walking series with length of 1024 and thus the total number of times that the nearest neighbor needs to be found is 897 ($1024 - 128 + 1$).

The experiments are coded in program language C/C++ and are performed on a DELL Dimension 8200 desktop box P4 2.0G, 512MB and WINDOWS XP OS. We adjust the stream rate by setting the interval between two successive values from 0.5 to 5 seconds, and the response time threshold also from 0.5 to 5 seconds, independently. We record the wall-clock times for the two algorithms, and report the average response times of the evaluations.

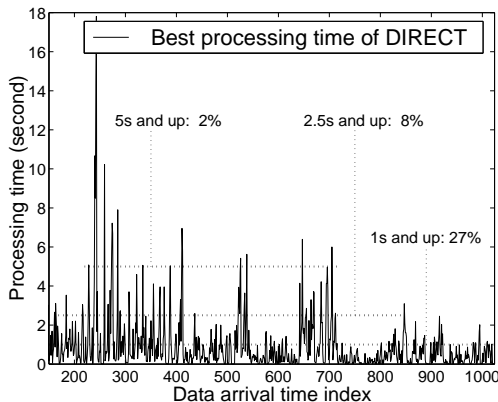


Figure 4: The best processing times of *Direct*

We first give the *processing time* needed to process the query with *Direct* algorithm in Figure 4. In this experiment, we assume that new data arrives as soon as the evaluation of the previous value is finished. No drop occurs. These are the best processing times that the *Direct* algorithm can achieve. Note the time variability: the processing times vary a lot. In Figure 4, three percentages of times that the processing time exceeds certain response time thresholds are given. As an example, when the threshold is 1 second, 27% of all evaluations is greater than 1 second.

As a comparison, the sequential scan method to find out the nearest neighbor needs about 14-15 seconds to evaluate the query for each arrived data. The pre-computation of the $L - 1$ partial distances also needs the same cost. This means that processing time for these methods will be about 15 seconds and thus the response time threshold must greater than it. From

Figure 4, we can see almost all processing times are far less than this value. This experimental result proves that the algorithms based on index techniques are more efficient than those without index.

The comparison results for the two index methods, *Direct* and *Pre-fetch*, are shown in Figure 5, with the response time threshold equal to 1, 2.5 and 5 seconds, respectively. We use “stream interval”, meaning the time between the arrivals of two successive values, instead of streaming rate in our experiments. The upper three graphs give the average drop ratios, and the lower ones show the response times. In both cases, the smaller the number, the better the performance.

Given a response time threshold, the drop ratio of *Direct* algorithm will become smaller as the stream data come slower, but it cannot be smaller than a certain value, no matter how slow the stream rate is. For example, in Figure 5(a.1) with threshold equal to 1 second, the drop ratio is a constant of about 27% when stream interval is greater than 1 second. This number can be observed from Figure 4 by counting the number of times where the processing time is greater than 1 second. Similar observations can be made with threshold equal to 2.5 and 5 seconds, too.

From Figure 5(a.1, a.2 and a.3), we can see that the *Pre-fetch* algorithm outperforms the *Direct* algorithm in terms of drop ratio with all combinations of stream rates and thresholds. Given the same stream rate, the difference of drop ratios becomes smaller when the threshold is bigger. This is because with bigger threshold, the relative times to launch the process of the predicted querying series is smaller. The absolute value of drop ratio will decrease as the stream rate is slower, while it is more important when the stream data is fast. In these graphs, we can see that when the stream rate is too fast, the drop ratios are close to each other. Then as the stream rate becomes a little slower, the difference becomes larger. Note in Figure 5(a.2 and a.3), especially (a.2), the drop ratios have a great difference of 3 – 6%.

The average response time of *Direct* changes little for different stream rates, and is less than the response time threshold, as shown in Figure 5(b.1, b.2 and b.3). Although the smaller stream interval will accumulate the delay of the evaluations, the algorithm will drop the query at some times and thus relieve the delay. When the stream interval is greater than response time threshold, the response time becomes a constant.

The response time with *Pre-fetch* algorithm will increase when the stream rate is faster. But *Pre-fetch* is much better than *Direct* algorithm, especially when the stream rate is slow and the threshold is large. For example, in Figure 5(b.3) with stream interval and the threshold both of 5 seconds, the average response time

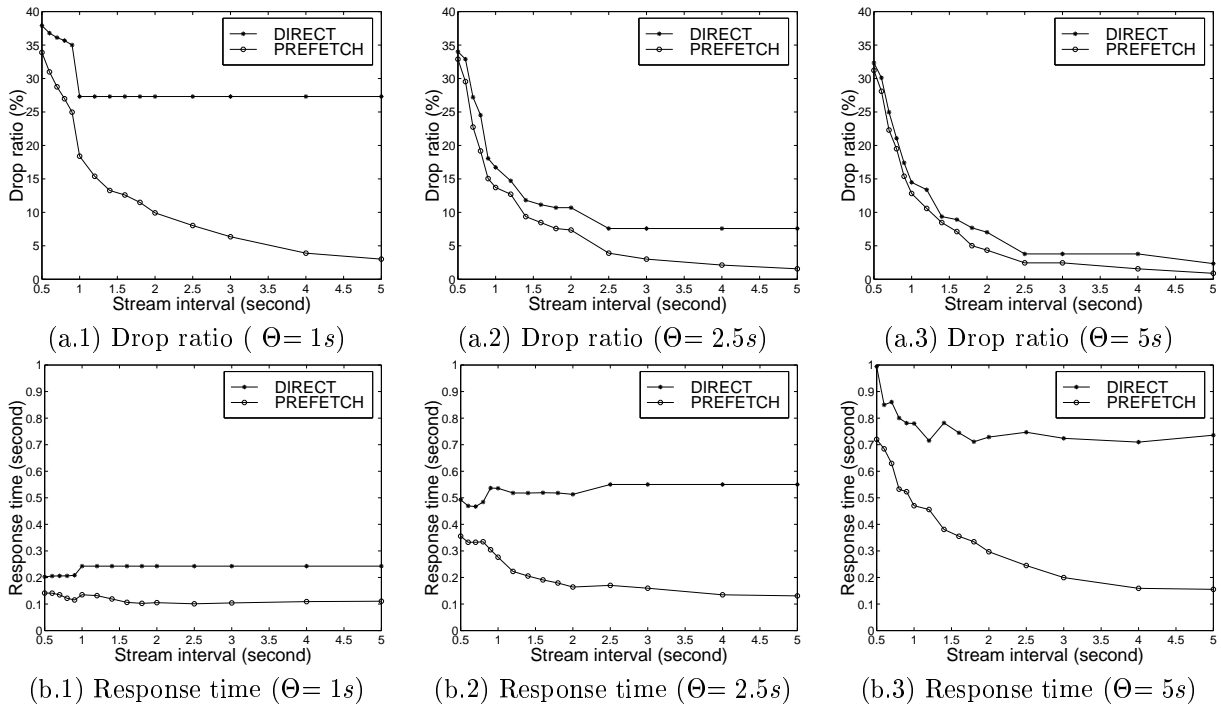


Figure 5: Performance comparison: response time and drop ratio (Θ = response time threshold, s = second)

of *Direct* algorithm is about 0.73 seconds, while *Pre-fetch* algorithm is only 0.16 seconds. We can see that *Pre-fetch* algorithm outperforms the *Direct* algorithm a lot. Note the corresponding drop ratio for *Pre-fetch* and *Direct* is 1% and 2% respectively, and *Pre-fetch* algorithm is still a winner.

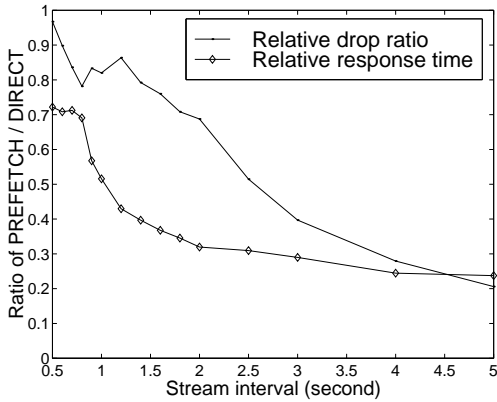


Figure 6: Relative ratios (threshold is 2.5 seconds)

From the description in Section 3, the *Pre-fetch* algorithm has no overhead as compared with *Direct* algorithm in terms of response time. The graphs in Figure 5 have demonstrated that under the same threshold and stream rate, *Pre-fetch* algorithm can decrease the drop ratio and response time simultaneously. In

Figure 6, we give the comparison with relative values of drop ratio and response time of these two algorithms. The response time threshold is 2.5 seconds in this graph. We can see that that both ratio curves are always below 1, which means that the *Pre-fetch* algorithm is always better than *Direct* algorithm.

5 Related work

The topic of this paper falls into both the general category of continuous queries and the nearest neighbor queries of time series.

Early in 1992, Terry et al. [16] first introduced the notion of “continuous queries”, where they proposed an incremental approach to evaluate the queries on append-only databases. Later, Liu et al. [10, 11] introduced notion of “continual queries”, where more general scenarios were considered. Recently, Chen et al. [3, 2] designed the NiagraCQ system in which the incremental query evaluation method was no longer restricted to append-only data sources. There are also some research on system architecture and related issues for continuous queries [1, 12, 7]. Specific queries, like continuous aggregation queries [7, 4], have also been studied. In this paper, we focus on streaming time series.

Near and nearest neighbor queries of time series have been a very interesting research topic in recent years. The general approach, see [5, 14, 13, 15], is to map the time series into the frequency domain,

with a Discrete Fourier Transform, and then index the significant part of the coefficients using certain high-dimensional indexing structure. These researches are the base for our approach to extend the stationary nearest neighbor queries into the continuous ones.

In our prior research [6], we considered the algorithms of the continuous nearest and near neighbor queries with a small number of database series, where the computation cost is the main concern. Later, we extended the research to deal with the situations that the number of database series is large and the I/O cost is the dominating factor. In this paper, we study the performance issue related to streaming rates.

Recently, S. D. Viglas and Jeffrey F. Naughton presented a rate-based query optimization for streaming data source [17]. They aimed at maximizing the output rate of query evaluation plans. In our work, we consider to improve both the output and the response time simultaneously. Recently, Zdonik et al. [18] introduced the Aurora system model for monitoring streams in which the performances are measured with response times, tuple drops and values produced. Our paper concerns with similar performance measures.

6 Conclusion

This paper is concerned with the performance improvement of the continuous nearest neighbor queries for streaming time series. With the limited response strategy to control the maximum response time, the first algorithm, *Direct* algorithm, is the extension of the traditional nearest neighbor query algorithms on time series. It uses the nearest neighbor of the previous evaluation to refine the current one. The second algorithm, *Pre-fetch* algorithm, efficiently uses the idle time between the arrivals of two successive data to predict the next querying series and process the predicted series, thus provides useful information (the nearest neighbor of predicted querying series) and resources (memory sharing) for the actual querying series. As the experiments show, this algorithm has no overhead in terms of response time and outperforms the *Direct* algorithm greatly.

The idea behind *Pre-fetch* algorithm is not limited to the continuous nearest neighbor queries for streaming time series. It may be applied to many other continuous queries as long as the next query is predictable to some degree. One obvious direction of the future work is to apply this strategy to continuous queries with other types of streaming data.

References

- [1] S. Babu and J. Widom. Continuous queries over data streams. In *SIGMOD Record, Sept. 2001*, 2001.
- [2] 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.
- [3] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proc. of SIGMOD*, pages 379–390, 2000.
- [4] A. Dobra, M. Garofalakis, J. E. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of SIGMOD*, 2002.
- [5] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of SIGMOD*, pages 419–429, 1994.
- [6] L. Gao and X. S. Wang. Continually evaluating similarity-based pattern queries. In *Proc. of SIGMOD*, 2002.
- [7] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of SIGMOD*, 2001.
- [8] A. Guttman. R trees: A dynamic index structure for spatial searching, 1984.
- [9] E. J. Keogh, K. Chakrabarti, S. Mehrotra, and M. J. Pazzani. Locally adaptive dimensionality reduction for indexing large time series databases. In *Proc. of SIGMOD*, 2001.
- [10] 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.
- [11] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):610–628, 1999.
- [12] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE Conference*, 2002.
- [13] D. Rafiei and A. Mendelzon. Similarity-based queries for time series data. In *Proc. of SIGMOD*, pages 13–25, 1997.
- [14] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. pages 71–79, 1995.
- [15] T. Seidl and H.-P. Kriegel. Optimal multi-step k-nearest neighbor search. In *Proc. of SIGMOD*, pages 154–165, 1998.
- [16] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. of SIGMOD*, pages 321–330, 1992.
- [17] S. D. Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of SIGMOD*, 2002.
- [18] Stan Zdonik, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Donald Carney. Monitoring streams - a new class of data management applications. In *VLDB*, 2002.