Efficient *k*-Nearest Neighbor Search for Static Queries over High Speed Time-Series Streams

Bui Cong Giao(✉) and Duong Tuan Anh

Faculty of Computer Science and Engineering,
Ho Chi Minh City University of Technology, Ho Chi Minh City, Vietnam
giao.bc@cb.sgu.edu.vn, dtanh@cse.hcmut.edu.vn

Abstract. In this paper, we propose a solution to the multi-step *k*-nearest neighbor (*k*-NN) search. The method is the reduced tolerance-based *k*-NN search for static queries in streaming time-series. A multi-scale filtering technique combined with a multi-resolution index structure is used in the method. We compare the proposed method to the traditional multi-step *k*-NN search in terms of the CPU search time and the number of distance function calls in the post-processing step. The results reveal that the reduced tolerance-based *k*-NN search outperforms the traditional *k*-NN search. Besides, applying multi-threading to the proposed method enables the system to have a fast response to high speed time-series streams for the *k*-NN search of static queries.

Keywords: *k*-NN search · Streaming time-series · Multi-scale filtering · Multi-resolution index structure · Static query

1 Introduction

At present, a significant number of real-world applications deal with time-series streams: performance measurements in network monitoring and traffic management, online stock analysis, earthquake prediction, etc. The major common characteristic of these applications is that they are all time-critical. In such applications, similarity search is often a core subroutine, yet the time taken for similarity search is almost an obstacle since time-series streams might transfer huge amount of data at steady high-speed rates. As a result, time-series streams are potentially unbounded in size within a short period and the system runs out of memory soon. Due to this, if an element of time-series stream has been processed, it is quickly discarded and cannot be retrieved so that it yields to a new-coming one. To achieve real-time response, one-pass scan and low time complexity are usually required for handling streaming time-series. However, available methods used to manage static time-series are hardly to satisfy the above requirements because they commonly need to scan time-series database many times for processing time-series data and often have high time complexity. Therefore, according to Yang and Wu [1], high-speed data streams and high dimensional data are the second ranking challenge among ten top challenging problems in nowadays' data mining. In addition, Fu [2] has recently conducted a review on time-series data mining and reckoned that mining on streaming time-series is an attractive research

P.C. Vinh et al. (Eds.): ICTCC 2014, LNICST 144, pp. 83–97, 2015.
DOI: 10.1007/978-3-319-15392-6_9

direction. In the scope of this paper, we only focus on pattern discovery by similarity search in streaming context. That is similarity-based streaming time-series retrieval, which is to find those streaming time-series similar to a time-series query.

Kontaki et al. [3] summarized that there are three similarity search types extensively experimented in the literature: range search, *k*-NN search and join search. Liu and Ferhatosmanoglu [4] reckoned that applications of streaming time-series might involve two query kinds: static queries that are predefined patterns as well as ad hoc and streaming queries that are continuously changed. In our previous work [5], we proposed a solution to range search for static queries in streaming time-series. In the paper, we will address a problem of improving the performance of the multi-step *k*-NN search for static queries in streaming time-series. Specifically, we deal with an important scenario in streaming applications where incoming data are from concurrent time-series streams at high speed rates, and queries are a fixed set of time-series patterns.

The existing multi-step *k*-NN search method [6] is to achieve a fixed tolerance and use the tolerance in the whole range query process. As a result, a large number of candidate sequences are retrieved by a range query with a large tolerance. Too many candidates incur CPU overheads in the post-processing step and eventually degrade the overall *k*-NN search performance. To reduce the number of candidates, we apply a tolerance reduction-based approach, which improves the search performance by tightening the tolerance of a query when its *k*-NN set is modified. The approach is similar in spirit to one suggested by Lee et al. [7]. Our method also uses a multi-resolution index structure is built on an array of R^*-trees [8] that supports the multi-scale filtering in similarity search. The index structure stores features of time-series subsequences of queries, extracted by any transform that satisfies the lower bounding condition and has multi-resolution property.

The main contributions of the paper are

- Using a tolerance reduction-based approach in *k*-NN search for static queries over high speed time-series streams;
- Adjusting range search in an R^*-tree for many queries at a time.

The rest of paper is organized as follows. Section 2 presents supporting techniques for our approach. Section 3 describes the proposed method. Section 4 discusses experimental evaluation, and Section 5 gives conclusions and future work.

2 Supporting Techniques

The section briefly describes supporting techniques for the proposed method. Some of the techniques were presented in detail in [5].

2.1 Multi-resolution Dimensionality Reduction Methods

A time-series X of length l might be considered as a point in l-dimensional space. It would be costly if we perform similarity search directly on the time-series X. Therefore, X is often transformed into a new space with lower dimensionality.

We use three common dimensionality reduction transforms: Discrete Fourier Transforms (DFT) [9], Harr Wavelet Transform (Haar DWT) [10], and PAA [11]. They all satisfy the lower bounding condition and have multi-resolution property.

2.2 Storing Coefficients at Resolutions

Each predefined query is segmented into non-overlapped segments. This makes sure that segments of queries are filtered continuously and this is suitable for streaming environment. These segments are normalized and transformed into coefficients by Haar DWT, PAA, or DFT.

Let denote *maxlevel* as the maximum level of the multi-resolution structure and *l* as the length of a query. We assume that the minimum length of queries is *min*. A query is separated into segments: *min*, 2*min*, 4*min*… from backward direction, such that $\exists n$, n is largest and $min \times (2n - 1) \leq \text{MIN}(l, maxlevel \times min)$. Therefore, n is the maximum level to which the query might be filtered. Fig. 1 depicts the segmentation from the backward direction of a query.

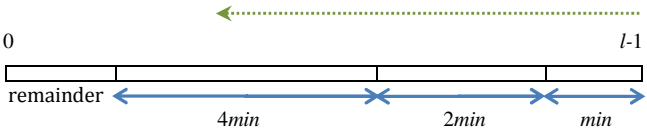

Fig. 1. The query segmentation [5]

The number of coefficients must be an integer power of two (2^i) since we want to compare the results of three transform methods: Haar DWT, PAA, and DFT. There is a maximum value of the number of coefficients; in [5] we recommend the maximum is 16. The number of coefficients at levels might be different. Assume that *min* = 8 and *maxlevel* = 5, we have the number of coefficients as shown in Table 1.

Table 1. Coefficient table

Level	Length	Number of Coefficients
1	8	2
2	16	4
3	32	8
4	64	16
5	128	16

2.3 Multi-resolution Index Structure

An array of R*-trees is used to store coefficients extracted from query segments. The index of the array corresponds to the resolution level that filters out the data. For example, the coefficients of the first segment (its length is *min*) of the query are stored

in the first R*-tree of the array (i.e. level 1). The coefficients of the second segment (its length is 2*min*) of the query are stored in the second R*-tree (i.e. level 2), etc. Notice that all static queries share the same array of R*-trees as their index.

2.4 Improved Range Search in R*-tree

R*-tree is an index structure organized as B-tree. It contains hierarchical nodes and parent (upper) nodes link to their child (lower) nodes. Only leaf nodes refer to spatial data object. Each node has its own minimum bounding rectangle (MBR). Because we use R*-tree for point (coefficients of query segments) query, we can have the information about whichever points lie in a MBR. The minimum distance h_{min} from a query point in a MBR to the margin of the MBR can be calculated after the R*-tree is constructed.

During the stage of *k*-NN search, a time-series point is checked to determine whether the time-series subsequence is a *k*-NN candidate of a query within its own tolerance *tl*. The procedure is done as follows: Firstly, the distance *dt* from a time-series point to the MBR is calculated. Next, we consider the query points in the MBR; if $dt < tl$, the time-series subsequence might be a *k*-NN candidate of that query. To reduce false alarms we additionally note that if $dt + h_{\min} < tl$ then the time-series subsequence is a more probable candidate. Fig. 2 depicts the improved range search in 2-dimensional space.

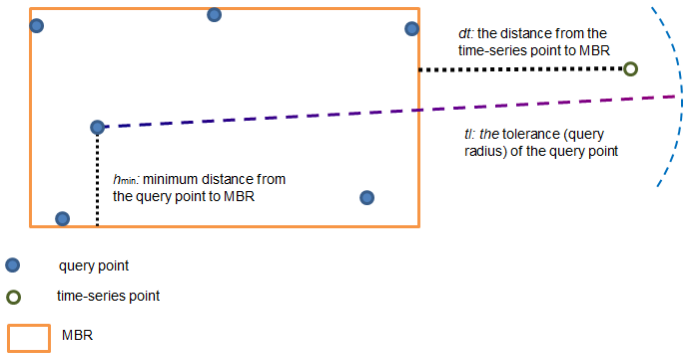

Fig. 2. The improved range search for a query point to a time-series point in the 2-dimensional space

2.5 Data Structures

To simulate data streams, we use a round-robin buffer to contain data points from a time-series stream. When the round-robin buffer is full, the new-coming element is put into the position of the oldest one; whereas in case of a common buffer, all elements of the buffer are forcibly shifted forward to have an available position for the new-coming one.

Each query has its own *k*-NN candidate set. The *k*-NN candidates of a query have their distance to the query less than a tolerance. We expect that the tolerance of a query is the maximum distance in the *k*-NN candidate set. When a time-series subsequence has its distance to the query less than the tolerance, it is added into the *k*-NN candidate set. Because time-series streams are potentially unbounded in size, so if the tolerance is large, the number of candidates in the *k*-NN set is too numerous and this incurs memory overflow in the execution of similarity search. For this reason, we cannot use a common in-memory set to store all candidates that have their distances less than the tolerance of a query. We propose using a priority queue organized as a max-heap to keep *k* candidates of a query. Therefore, the top item in the priority queue is the time-series subsequence that has the maximum distance to the corresponding query. Another advantage of using priority queues as the *k*-NN sets of queries is that the system can return the sorted *k*-NN items of a query at any time, while in the traditional *k*-NN search it takes time to sort the final candidates before the *k*-NN items are returned. However, using priority queues also costs some time for removing the top item to yield to a new time-series subsequence whose distance is less.

3 Proposed Method

The *k*-NN search algorithm consists of three main phases as follows:

Phase 1: **Preprocessing**

At the beginning of the program, the *k*-NN sets of all queries are initialized with *k* false items whose distances are a maximum value (∞). Let denote *NotFullQueryList* as the global list of queries whose *k*-NN set has still false items; initially the list contains all queries. Also, let *kNNinfo* be the global list of queries whose entire *k*-NN sets contain true items; initially the list is empty.

Next, queries are segmented and normalized. Their coefficients are calculated and stored in a multi-resolution index structure which is an array of R^*-trees. Notice that segments of predefined queries only need normalizing once in this phase, but segments of time-series stream have to be normalized at every new-coming data point.

Phase 2: ***k*-NN search**

When there is a new-coming data point of a time-series stream, segments on the streaming time-series are incrementally normalized. Using the incremental data normalization based on *z*-score [5] enables the method not to compute data normalization from the scratch. After that, the coefficients of the normalized segments are calculated. These coefficients are matched with the coefficients of query segments already stored in the nodes of R^*-tree within each tolerance of the queries from the lowest to upper levels. Fig. 3 illustrates the backward matching of each pair of the coefficients. This matching step, which applies multi-step filtering, helps to prune unsatisfying queries. Lastly, the query candidates are checked in the post-processing phase.

Phase 3: **Post-processing**

For each query candidate, the real distance between the query candidate and the corresponding piece of streaming time-series is calculated to find a true *k*-NN item

(time-series subsequence). If a true k-NN item is found, the k-NN set is updated and the tolerance might be reduced.

Table 2 shows some notations we will use in the following.

Table 2. Frequently used notations

Notation	Meaning
q	a query sequence
$q.kNN$	the priority queue of q
$q.normal$	the normalized sequence of q
m	the number of queries
$SortedByToleranceList$	data structure of lists sorted by descending tolerance
$S = \{\ldots T_{n-2}, T_{n-1}, T_n\}$	a streaming time-series
s_i	the i^{th} time-series segment of S, i: 0..$maxlevel$-1
T_n	new-coming data at time point n
T_{n-1}	new-coming data at time point n-1

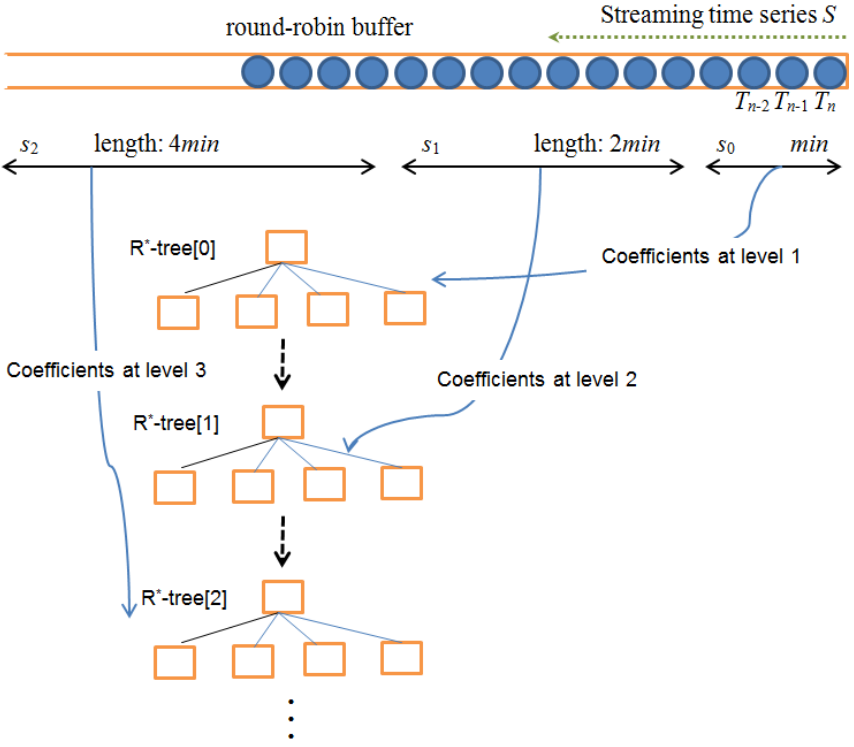

Fig. 3. Query filter through resolution levels [5]

The algorithm has two options. Firstly, when the *k*-NN set of a query is full (that means all items in the set are real), the tolerance is the maximum distance in the set and since then, it is not changed. Secondly, when the *k*-NN set of a query is full, the tolerance is the maximum distance in the set and it might be changed whenever the set is updated. We will evaluate the performance of the two options in section 4. The following pseudo codes illustrate the *k*-NN search for static queries over a time-series stream.

Algorithm *k*-NNSearch(Streaming time-series *S*)

---

variables

- *SCandidates*: a static local list contains candidate queries whose *k*-NN sets are full. Its type is *SortedByToleranceList*.
- *SNotFullQueryList*: a static local list contains candidate queries whose *k*-NN sets are not full.
- *postCheckSet*: a set of queries for post-processing
- *tempList*: a query list whose type is *SortedByToleranceList*
- *res*: a query candidate list
- d_{max}: the maximum distance of *q.kNN*
- s: a corresponding piece of *S* with a current query

begin

```
When there is a new-coming data of S: T_n          // Phase 2
1. if SCandidates is different from kNNinfo then
2.     Copy kNNinfo to SCandidates
3.     if |SCandidates| < m then
4.        Copy NotFullQueryList to SNotFullQueryList
5. if |SCandidates| > 0 then
6.     postCheckSet = ∅
7.     tempList = SCandidates
8.     for i = 0 to maxlevel - 1
9.        res = Searchk-NN(Coef(IncNormalize(s_i)),tempList,R*-
tree[i])
10.       foreach (Query q in res)
11.         if i is the maximum resolution level of q then
12.            postCheckSet = postCheckSet ∪ q
13.            Remove q from res
14.       end foreach
15.       if res is empty then
16.            break
17.       tempList = Sort res by descending tolerances
18.    end for
19.    foreach (Query q in postCheckSet)        // Phase 3
20.       d = d_o(q.normal, Normalize(s))
21.       Get d_max from q.kNN
```

```
22.       if d < d_max then
23.          q.kNN.Dequeue
24.          q.kNN.Enqueue(s) with d
25.          Get d_max from q.kNN
26.          Update q and its d_max in kNNinfo
27.    end foreach
28. foreach (Query q in SNotFullQueryList) //Phase filling k-NN
sets
29.    d = d_o(q.normal, Normalize(s))
30.    q.kNN.Dequeue
31.    q.kNN.Enqueue(s) with d
32.    Get d_max from q.kNN
33.    if d_max < ∞ then
34.       Add q and its d_max into kNNinfo
35.       Remove q from NotFullQueryList
36. end foreach
end
```

There are some noticeable issues in the phases of *Algorithm k-NNSearch*:

- *Phase 2:* If *SCandidates* does not contain all queries then *NotFullQueryList* is copied to *SNotFullQueryList* (lines 3-4). Copies (lines 2 and 4) ensure that the algorithm manipulates local resources, not shared global ones. Next, the two static local resources are *SCandidates* and *SNotFullQueryList* considered one by one. Line 9 implies that segments s_i is incrementally normalized and a coefficient vector is extracted from the normalized segment; then a *k*-NN search in the R*-tree of the i^{th} filter level is performed. *Searchk-NN* calls *Algorithm Searchk-NN* with *node* as the root node of the R*-tree. The query results are checked (line 11) to create a candidate set for phase 3. Going through filter levels might ends early if the candidate set for the next traverse is empty (lines 15-16). Because the return results of the *k*-NN search in the R*-tree is not to follows the type of *SortedByToleranceList*, the results need sorting by descending tolerance for the next filter level (line 17).

- *Phase 3:* Line 20 implies that the real distance between the normalized query sequence, which is calculated beforehand, and the time-series subsequence, which is normalized at the moment, is performed. As mentioned before, the algorithm has two options: the reduced tolerance-based *k*-NN search and the traditional *k*-NN search. Lines 25-26 exist for the first option while these codes are omitted for the second option. It is obvious that when we compare the two options, the execution cost of lines 2-4 is less in the second option.

- *Phase filling k*-NN *sets:* the phase illustrates filling *k*-NN queues of queries with real items. If a *k*-NN queue is full, that means the condition of line 33 is true, the query information is added into *kNNinfo* (line 34) and the query is removed out of *NotFullQueryList* (line 35).

To support the *k*-NN search in an R*-tree, the nodes in the index structure need to include the information of points that lies in the MBR of the nodes. The information is a list of items whose structure consists of *pointID*, *entryID*, and *hmin.* The list is in

order of *pointID*. *pointID* is the *queryID*. *entryID* is *ID* of the entry that refers to the child node containing the point. *hmin* is the minimum distance from the point to the MBR margin of the node. For example, a node has *entrieIDs*: 10, 14, 21 and *pointIDs*: 2, 4, 9, 10, 15, 22, and 35. *entryID* 10 contains 9 and 22; *entryID* 14 contains 2, 15, and 35; and *entryID* 21 contains 4 and 10. Let *node.information* be the information of the points in the node. Fig. 4 illustrates an example of *node.information*.

pointID	2	4	9	10	15	22	35
entryID	14	21	10	21	14	10	14
hmin	0	1.2	2.3	0	1.5	4.6	2.8

Fig. 4. An illustration of *node.information*

We note that *hmin* is 0 if the point lies at the margin of the MBR of the node.

Algorithm Searchk-NN elaborates the function *Searchk-NN* used in line 9 of *Algorithm k-NNSearch*. This important algorithm performs the *k*-NN search of a coefficient point against a query list, whose type is *SortedByToleranceList*, on a node of an R*-tree.

Algorithm Search*k*-NN(*point, ql, node*)

variables

- *cl:* an array of lists of query candidates in *node*, which needs considering in the lower level of the R*-tree. The indexes of the array are entry IDs of *node*.
- *res:* a query candidate list. Initially, *res* is empty.

```
begin
1.  calculate distance dt between point and the MBR of  node
2.  if dt = 0 then
3.        foreach (item in ql)
4.           Get entryID of item from node.information
5.           Add item into cl[entryID]
6.        end foreach
7.  else
8.        foreach (item in ql)
9.          if dt >= item.tolerance then
10.            break
11.         Get hmin of item from node.information
12.         if dt + hmin < item.tolerance then
13.            Get entryID of item from node.information
14.            Add item into cl[entryID]
15.       end foreach
16.       if cl is empty then
17.         return res
18. if node is leaf then
```

```
19.     foreach (item in cl)
20.        calculate distance dt between point and the point of
item
21.        if dt < item.tolerance then
22.           item.tolerance = item.tolerance - dt;
23.           Add item into res
24.     end foreach
25. else                              // internal node
26.     foreach (SortedByToleranceList el in cl)
27.        child is a child node of node, entryID of el refers to
child
28.        Add results of Searchk-NN(point,el,child) to res
29.     end foreach
30. return res
end
```

We have some notes about *Algorithm Searchk-NN*:

In the first stage, if the condition in line 2 is true, that means the point lies in the MBR or at the margin of the MBR, an array of lists of query candidates is created from items of *ql* (line 5). Otherwise, if distance *dt* is larger than the tolerance of an item (query), the loop ends early because surely this item and the remaining items are not candidates (lines 9-10). If not, an additional check is performed to make sure that *item* is a candidate (see more in section 2.4). If the condition in line 12 is true, *item* is added to the list that is identified by the *entryID* of *item* (lines 13-14). At the end of the stage, if the array of lists of query candidates does not have any item, the algorithm ends early (line 17).

In the second stage, if *node* is leaf, entries of *node* refer to point objects (coefficient vectors). If the condition in line 21 is true, *item* is a candidate of the *k*-NN set (line 23). The tolerance of *item* is reduced by distance *dt* for the next filter level (line 22). If *node* is not a leaf, each list in array *cl* has an *entryID* as the index of the list and the entry refers to a child node (line 27). The algorithm is called recursively again and the results are added into *res* (line 28).

The process of the *k*-NN search must begin from the root node. So the function *Searchk-NN (Algorithm Searchk-NN)* should be invoked at the first time by the following statement: *Searchk-NN(point, ql, Index-tree.root)*.

To clarify the structure of *ql* and *cl*, we reconsider the example in Fig. 4. Fig. 5 is an example of *ql* and Figure 6 is one of *cl*.

pointID	15	10	2	35	9
tolerance	16.5	12.2	8.7	0.4	0.1

Fig. 5. An illustration of *ql*

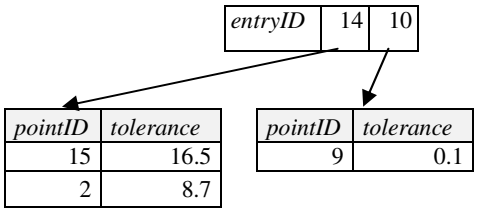

Fig. 6. An illustration of *cl*

4 Experimental Evaluation

In this section, we present experiments on the proposed method and the traditional *k*-NN search method, and evaluate the performance of the system. All experiments have been conducted on an Intel Dual Core i3 M350 2.27 GHz, 4GB RAM PC.

To take advantage of the strength of today's CPU and due to the characteristic of the search method, we use multi-threaded programming. Each threading process handles one time-series stream to implement *Algorithm k-NNSearch*. For simplicity, all threading processes have the same priority. The programming language used in this work is C# since the language is powerful for multi-threading. Because threading processes can compete to update the same global resources (e.g. *k*-NN sets) at a time, the system must lock the shared resources before updates can be done. However, this technique degrades the performance of the system. To mitigate the problem, locks for update must occur as quickly as possible by optimizing update operations.

We used ten text files containing the time-series datasets that are input for ten time-series streams. The sources of the datasets are given in column 4 of Table 3. One thousand text files created from ten above datasets play a role of static queries. The number of queries created from a dataset is proportional to the number of points in the text file that simulates the dataset. The size of the queries varies from 8 to 256. The number of filter levels is 5 for these queries. Level 1 can filter queries whose lengths are greater than or equal to 8. Level 2 can filter queries whose lengths are greater than or equal to 24, and so on. The total number of data points in the queries is 133,771. Other parameter setting in the experiments is as follows. Buffer length of each time-series stream is 1,024. R*-tree has the setting: $m = 4$ and $M = 10$.

Table 3. Text files used to simulate time-series streams

No	Datasets	Number of Points	Source
1	carinae.txt	1,189	[12]
2	D1.txt	8,778	[13]
3	D2.txt	50,000	[13]
4	darwin.slp.txt	1,400	[12]
5	eeg7-6.txt	3,600	[12]
6	Hawea-91757.txt	7,487	[14]
7	infraredwave.txt	4,096	[13]
8	lightcurve.txt	27,204	[13]
9	Pukaki-877571.txt	7,487	[14]
10	wirewave.txt	4,096	[13]
	Total points	115,337	

We have implemented experiments to compare the reduced tolerance-based *k*-NN search to the traditional *k*-NN search. The criteria for comparing the two approaches

are the CPU search time and the number of distance function calls in post-processing step of Haar DWT, PAA, and DFT. The parameter k is from 1 to 3.

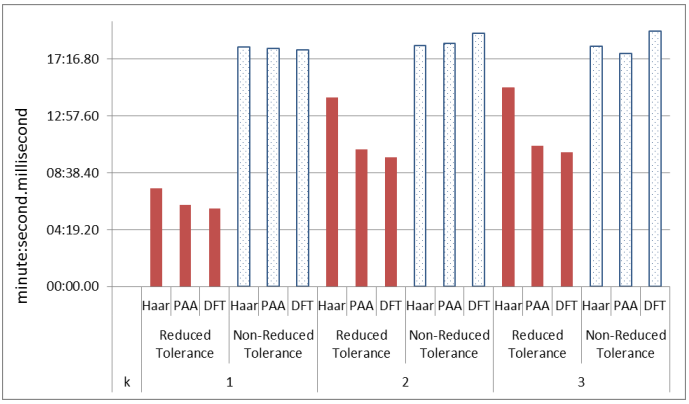

Fig. 7. CPU search times of k-NN search for the two methods

Fig. 7 shows the first approach is better than the second one in the CPU search time. For the first method, the CPU search time has increasing tendency when k increases and the CPU search time of Haar DWT is largest and that of DFT is least. As for the second method, the CPU search times of three dimensionality reduction transforms are nearly the same.

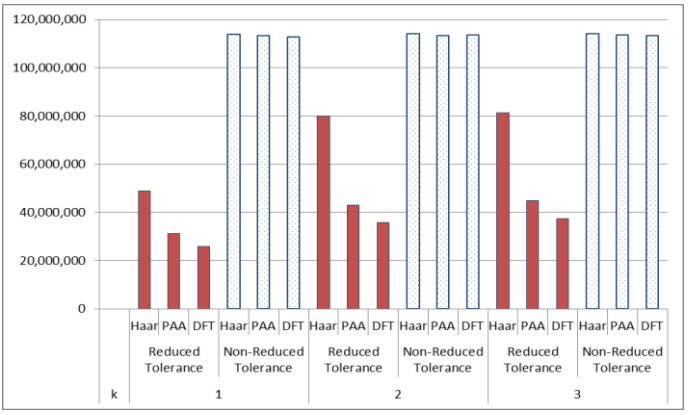

Fig. 8. The number of distance function calls in the post- processing step for the two methods

Fig. 8 shows that in the post-processing step, the first approach has the number of distance function calls less than the second does. For the first approach, the number increases when k increases and DFT has the least number and Haar DWT has the largest one. However, in the second approach, these numbers are almost the same thought k increases.

From Fig. 7 and Fig. 8 we can conclude that although the tradition *k*-NN search does not change tolerances, (that means the locking time for the global resource (*kNNInfo*) is less than our approach), tolerances in this approach are often large and the number of distance function calls in the post-processing step is too much. This degrades the overall performance.

We have already implemented the *k*-NN search method without multi-threading; in the case, the system only has a process to scan time-series streams sequentially for handling new-coming data points, whereas the proposed method has every threading process for a time-series stream. Fig. 9 shows the average CPU times for processing a new-coming data point over the ten time-series streams with *k* varying from 1 to 10 in case of the system handling time-series streams sequentially. We note that when *k* increases, in general, the average CPU times of the dimensionality reduction transforms also increase. With each *k*-NN search, the average CPU time of DFT is least, while that of Haar DWT is largest; the average CPU time of PAA is slightly larger than that of DFT.

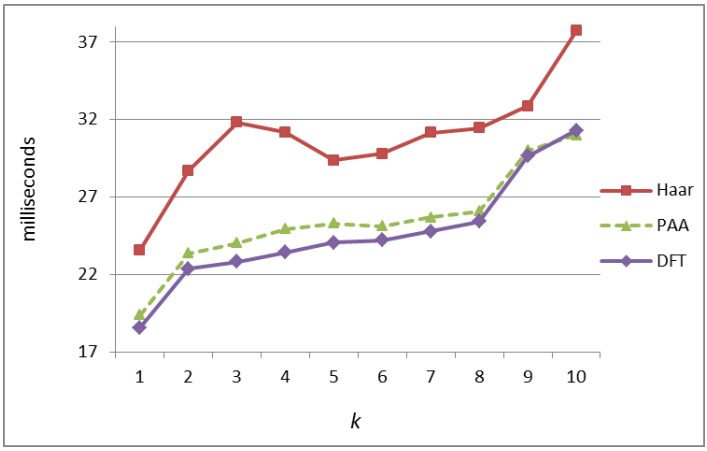

Fig. 9. The average CPU times for processing a new-coming data point when the system handles time-series streams sequentially

Since the proposed method processes time-series streams simultaneously, the performance of the system is improved significantly. The average CPU times for processing a new-coming data point of the system are slightly small in case of the system handling time-series streams simultaneously. For the 1-NN search, the average CPU time of DFT in Fig. 9 is 18 milliseconds, while in Fig. 10 the value is only 3 milliseconds. For the 5-NN search, the average CPU time of PAA in Fig. 9 is 25 milliseconds, while in Fig. 10 the value is about 6 milliseconds. For 10-NN search, the average CPU time of Haar DWT in Fig. 9 is 38 milliseconds, while in Fig. 10 the value is 10 milliseconds. Therefore, multi-threaded programming as a whole offers dramatic improvements in speed (up to roughly 4 times) over traditional programming. The comparison demonstrates the usability of multi-threading to proposed method for real-time applications that need perform *k*-NN search for static queries over time-series streams at high-speed rates.

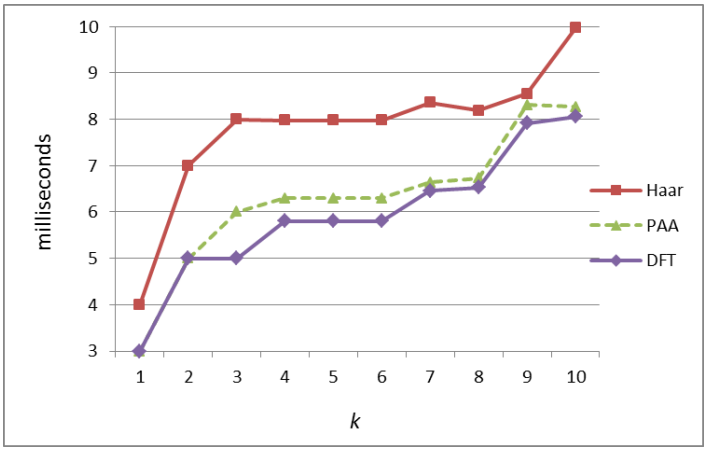

Fig. 10. The average CPU times for processing a new-coming data point when the system handles time-series streams simultaneously

5 Conclusions and Future Work

We have introduced an efficient method to the multi-step *k*-NN search for static queries over streaming time-series. In the method, the tolerance of each query is reduced when the maximum distance from that query to the top item in its *k*-NN queue is reduced. Moreover, in order to make the approach meaningful, we carry out the incremental data normalization before the *k*-NN search. A salient feature of the proposed method is using multi-scale filtering technique combined with a multi-resolution index structure that are an array of R*-trees. In the range step of *k*-NN search, the method introduces improved range search by including the information of point objects in nodes of R*-trees, and range search for many queries can be performed simultaneously. The experimental results show that for static queries in streaming time-series, the reduced tolerance-based *k*-NN search outperforms the traditional *k*-NN search. Finally yet importantly, with the proposed method, we have recorded the average CPU times for processing a new-coming data point of in case of the system handling time-series streams sequentially, and in case of the system handling time-series streams simultaneously. The results show multi-threading makes the approach increase approximately 4 times in speed. From the extensive experiments, the proposed method combined with multi-threading presents a fast response to process high-speed time-series streams for *k*-NN search of static queries.

As for future work, we plan to adjust the proposed method to handle *k*-NN search for streaming queries in high-speed streaming time-series.

References

1. Yang, Q., Wu, X.: 10 challenging problems in data mining research. International Journal of Information Technology and Decision Making (2006)
2. Fu, T.-C.: A review on time series data mining. Journal of Engineering Applications of Artificial Intelligence (24), 164–181 (2011)
3. Kontaki, M., Papadopoulos, A., Manolopoulos, Y.: Adaptive similarity search in streaming time series with sliding windows. Data and Knowledge Engineering **16**(6), 478–502 (2007)
4. Liu, X., Ferhatosmanoglu, H.: Efficient k-NN search on streaming data series. In: Hadzilacos, T., Manolopoulos, Y., Roddick, J., Theodoridis, Y. (eds.) SSTD 2003. LNCS, vol. 2750, pp. 83–101. Springer, Heidelberg (2003)
5. Giao, B., Anh, D.: Efficient similarity search for static queries in streaming time series. In: Proceedings of the 2014 International Conference on Green and Human Information Technology, HoChiMinh City, pp. 259–265 (2014)
6. Korn, F., Sidirapoulos, N., Faloutsos, C., Siegel, E., Protopapas, Z.: Fast nearest neighbor search in medical databases. In: Proceedings of the 22nd International Conference on Very Large Data Bases, Bombay, India, pp. 215–226 (1996)
7. Lee, S., Kim, B.-S., Choi, M.-J., Moon, Y.-S.: An approximate multi-step k-NN search in time-series databases. Advances in Computer Science and its Applications **279**, 173–178 (2014)
8. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-tree: an efficient and robust access method for points and rectangles. In: ACM SIGMOD International Conference on Management of Data, Atlantic City, New Jersey, USA, pp. 322–331 (1990)
9. Agrawal, R., Faloutsos, C., Swami, A.: Efficient similarity search in sequence databases. In: Lomet, D.B. (ed.) FODO 1993. LNCS, vol. 730, pp. 69–84. Springer, Heidelberg (1993)
10. Chan, K.-P., Fu, A.: Efficient time series matching by wavelets. In: Proceedings of the 15th IEEE International Conference on Data Engineering, pp. 126–133 (1999)
11. Keogh, E., Chakrabarti, K., Mehrotra, S., Pazzani, M.: Locally adaptive dimensionality reduction for indexing large time series databases. In: Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data, pp. 151–163 (2001)
12. West, M.: http://www.isds.duke.edu/~mw/data-sets/ts_data/ (accessed December 2013)
13. Weigend, A.: Time series prediction: Forecasting the future and understanding the past. http://www-psych.stanford.edu/~andreas/Time-Series/SantaFe.html (accessed December 2013)
14. Group, M.: Electricity Authority's market data and reporting portal. ftp://ftp.emi.ea.govt.nz/Datasets/ (accessed December 2013)