A New Approach for Processing Ranked Subsequence Matching Based on Ranked Union

Wook-Shin Han
Kyungpook National University, Korea
wshan@knu.ac.kr

Jinsoo Lee
Kyungpook National University, Korea
jslee@www-db.knu.ac.kr

Yang-Sae Moon
Kangwon National University, Korea
ysmoon@kangwoon.ac.kr

Seung-won Hwang
POSTECH, Korea
swhwang@postech.ac.kr

Hwanjo Yu
POSTECH, Korea
hwanjoyu@postech.ac.kr

ABSTRACT

Ranked subsequence matching finds top-k subsequences most similar to a given query sequence from data sequences. Recently, Han et al. [12] proposed a solution (referred to here as HLMJ) to this problem by using the concept of the minimum distance matching window pair (MDMWP) and a global priority queue. By using the concept of MDMWP, HLMJ can prune many unnecessary accesses to data subsequences using a lower bound distance. However, we notice that HLMJ may incur serious performance overhead for important types of queries. In this paper, we propose a novel systematic framework to solve this problem by viewing ranked subsequence matching as *ranked union*. Specifically, we propose a notion of the *matching subsequence equivalence class* (MSEQ) and a novel lower bound called the *MSEQ-distance*. To completely eliminate the performance problem of HLMJ, we also propose a *cost-aware density-based scheduling* technique, where we consider both the density and cost of the priority queue. Extensive experimental results with many real datasets show that the proposed algorithm outperforms HLMJ and the adapted PSM [22], a state-of-the-art index-based merge algorithm supporting non-monotonic distance functions, by up to two to three orders of magnitude, respectively.

Categories and Subject Descriptors

H.3.1 [**Information storage and retrieval**]: Content Analysis and Indexing—*Indexing methods*; H.3.3 [**Information storage and retrieval**]: Information Search and Retrieval—*Search process*

General Terms

Algorithm, Performance

Keywords

Time-series data, Ranked subsequence matching, Ranked union

1. INTRODUCTION

Time-series data are of growing importance in data mining and data warehousing [14, 19]. A time-series is a sequence of real numbers representing values at specific points in time. Typical examples include music data, stock prices and network traffic data. The time-series data stored in a database are called data sequences.

Various similarity models have been studied in similar sequence matching (finding data sequences from a database which are similar to a given query sequence). In this paper, we use the similarity model based on the dynamic time warping (DTW) distance [4]. The DTW distance is one of the most robust and widely used measures for various applications such as query by humming [24], image searching [3], and speech recognition [18].

In this paper, we study ranked subsequence matching, identifying top-k subsequences in the database that are similar to a query sequence by using DTW. The state-of-the-art solution to this problem is by Han et al. [12] (referred to here as HLMJ). However, we notice that HLMJ may incur serious performance overhead for important types of queries. We briefly illustrate HLMJ below to depict the problems which motivate our solution.

In the index building process, HLMJ divides each data sequence into disjoint windows, transforms each window into a lower-dimensional data point (say, an f-dimensional point), and stores it as a leaf entry in the R*-tree. In the ranked subsequence matching process, HLMJ first divides a query sequence into sliding windows and transforms each *query* window into an f-dimensional query point. HLMJ maintains a global priority queue to store pairs of f-dimensional query and data points with their distances. Next, HLMJ pops a query/data point pair from the priority queue where their distance is the minimum, then computes a lower bound distance using those points. If this lower bound distance is greater than the DTW distance between the query sequence and the top k-th data subsequence obtained thus far, that pair is pruned. Otherwise the corresponding data subsequence for the data point is accessed to compute the DTW distance. HLMJ repeats this process until it finds top-k subsequences. Figure 1 illustrates how HLMJ works. A data sequence S is divided into disjoint windows (s_1, s_2, $\cdots$), and these windows are transformed into 2D points and stored in the R*-tree. In query processing, HLMJ divides a query sequence Q into eight sliding windows ($q_1 \sim q_8$) and transforms these windows into 2D points. After that, HLMJ pops a pair (q_1, s_5) from the global priority queue, retrieves a data subsequence for s_5, and computes the DTW distance.

As a lower bound distance for pruning, HLMJ exploits the minimum-distance matching-window pair (MDMWP) distance. The MDMWP for a query sequence and a data subsequence is a matching window pair, where the distance between the two windows in the pair is the minimum among all matching window pairs. If there are r matching windows pairs for a query sequence and a data sub-

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.

SIGMOD'11, June 12–16, 2011, Athens, Greece.

Copyright 2011 ACM 978-1-4503-0661-4/11/06 ...$10.00.

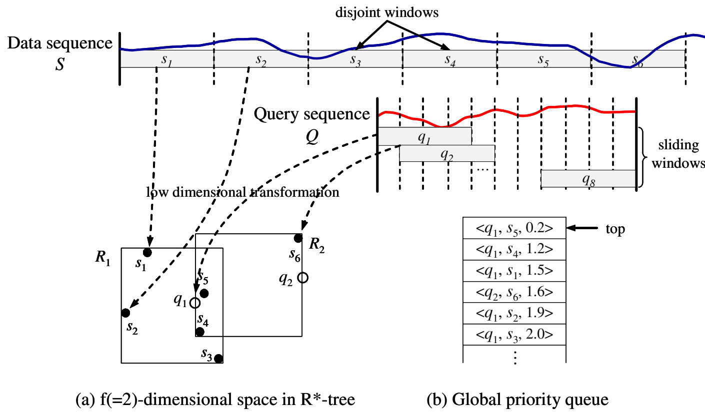


Figure 1: An example of HLMJ.

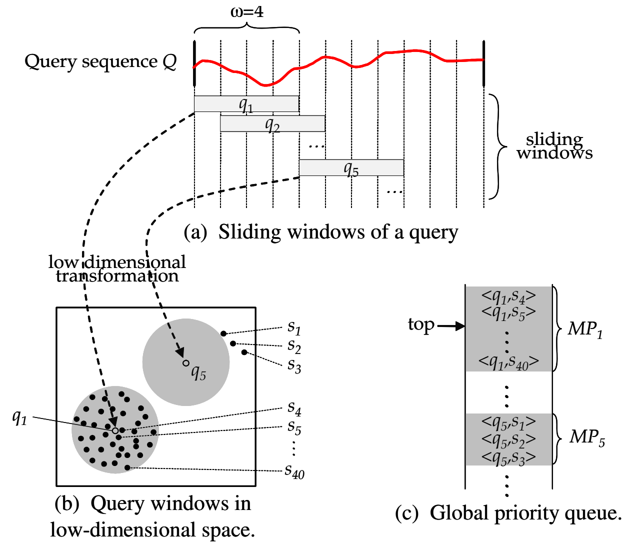


Figure 2: Example of the MDMWP-based scheduling problem.

sequence, and (q_m, r_m) is the MDMWP, then the MDMWP distance is $\sqrt[p]{r \times d(q_m, s_m)^p}$ [12]. In Figure 1 for example, after popping the next entry $(q_1,\ s_4)$ from the global priority queue, HLMJ computes the MDMWP distance for this entry. Since the popped entry is $(q_1,\ s_4)$, the other matching window pair becomes $(q_5,\ s_5)$, and thus, its distance must be $\geq$ 1.2 according to the definition of MDMWP. Thus, the MDMWP distance for this pair is $\sqrt[p]{d(q_1, s_5)^p \times 2} = \sqrt[p]{1.2^p \times 2}$. If $p = 2$, the MDMWP distance is nearly 1.7. Assume that the DTW distance computed for the first entry $(q_1,\ s_5)$, i.e, the DTW distance between the query sequence and the top first data subsequence obtained thus far, is 1.5. Thus, if we want to find only the top-1 data subsequence, the entry $(q_1,\ s_4)$ is pruned without accessing its corresponding data subsequence since its lower bound distance is greater than 1.5.

However, in HLMJ using MDMWP-based scheduling and a global priority queue, we observe the following serious performance degradation, which occurs in real datasets as we will see in Section 6. Figure 2 illustrates one such example, where a query point q_5 is placed in a sparse region while q_1 is placed in a dense region. In the global queue, all matching pairs MP_1 of q_1 and the data points in the dense region are placed in the upper part of the queue whereas such pairs for q_5 and the data points in the sparse region are placed below MP_1. Since it pops iteratively from the top of queue, HLMJ cannot reach MP_5 until it finishes the processing of all pairs in the entire MP_1. Meanwhile, accessing MP_5 first is highly effective in quickly increasing a lower bound. For example, when $\langle q_1, s_4 \rangle$ is popped, its MDMWP distance is $\sqrt[p]{d(q_1, s_4)^p \times 2}$. Instead, if we first access $\langle q_5, s_1 \rangle$, its lower bound can grow up to $\sqrt[p]{d(q_1, s_4)^p + d(q_5, s_1)^p}$ such that $d(q_5, s_1) \gg d(q_1, s_4) \simeq 0$, since the matching window pairs for q_1 and q_5 are s_4 and s_1, respectively. However, since there can be as many query points as there are sliding windows, it is non-trivial to extend HLMJ to 1) optimize the access order of query points and 2) derive a tight lower bound. Furthermore, since it is highly likely to grow very slowly as well, the MDMWP-distance can be of little use for pruning in this example. In the worst case, even if the selectivity of a query is very low, the algorithm might end up accessing too many candidates, incurring even higher cost than sequential scan.

Alternatively, one may consider applying a general ranked retrieval algorithm, such as TA-family algorithms [5, 6, 9, 20]. However, unlike these algorithms which assume that ranking functions are monotonic, the scoring function DTW and its lower bound function LB_{PAA} [24] are non-monotonic. Furthermore, only *lower bound distances* are computed at the index level since lower-dimensional data points are stored in the R*-tree. Thus, the corresponding data subsequences must be accessed additionally to calculate exact distances. More recently, algorithms for supporting non-monotonic ranking functions have been studied [22]. We adopt the best algorithm of [22] as another baseline approach. However, as we will see in Experiment 6, this method suffers from bad performance whenever the number of disjoint query windows > 3.

In this paper, we propose the first systematic framework which viewes a ranked subsequence matching query as a ranked union query over ω ranked subqueries, where ω is the size of a sliding window, and ranked union merges the results from the subqueries and returns top-k results. Specifically, we propose the novel notion of *matching subsequence equivalence class* (MSEQ), with which we partition the query points into ω MSEQs so that each MSEQ consists of only disjoint query windows. If we regard each query point as an f-dimensional attribute, then each equivalence class $MSEQ_i$ corresponds to a ranked subquery having $|MSEQ_i|$ f-dimensional attributes. Instead of maintaining one global queue, we allocate a priority queue for each query point, so that each f-dimensional attribute has a dynamically generated and sorted list. This novel interpretation enables us to carefully control the scheduling order for the consumption of the priority queues for each subquery, i.e., sophisticated scheduling over the priority queues.

This interpretation, however, presents several challenging questions. 1) How can we derive a lower bound for ranked union? 2) What is the optimal access order for selecting subqueries? 3) If a certain subquery is selected, which priority queue should be processed? More specifically, how can we avoid the MDMWP-based scheduling problem? To answer the first two questions, we propose an execution model for ranked union and derive a new lower bound called MSEQ-distance under this execution model. We propose a cost-aware, density-based scheduling to approach the third question. This way, query points having relatively lower density can be processed quickly. The reason we consider cost-awareness is that it would be better to select a priority queue PQ_1 if the disk I/O cost of accessing the subsequences corresponding to the entries of PQ_1 is much lower than other priority queues.

Our contributions are as follows: 1) We propose the first framework to view ranked subsequence matching as ranked union using (a) a novel notion of matching subsequence equivalence class (MSEQ) and (b) a new lower bound distance, the MSEQ-distance; This can also be viewed as a generalized framework for ranked subsequence matching. 2) By doing so, we show that the state-of-the-art solution, HLMJ using MDMWP-based scheduling with a

global priority queue [12] may lead to severe performance degradation. 3) To remedy this problem, we propose a cost-aware density-based scheduling technique. 4) Extensive experimental results with many real datasets show that the proposed algorithm outperforms (a) HLMJ and (b) the adapted PSM [22], a state-of-the-art index-based merge algorithm supporting non-monotonic distance functions, by up to two to three orders of magnitude.

The rest of this paper is organized as follows. Section 2 reviews DTW and HLMJ. Section 3 presents translation of the ranked subsequence matching query into the ranked union query. In Section 4, we present a cost-aware density-based scheduling method that avoids the MDMWP-based scheduling problem by minimizing the cost of accessing entries from priority queues. Section 5 presents an overview of related work, and Section 6 presents the results of performance evaluations. Section 7 summarizes and concludes the paper.

2. BACKGROUND

Table 1: Summary of notation.

Symbols	Definitions
S_{sid}	A sequence whose identifier is *sid*
$Len(S)$	Length of sequence S
$S[i]$	The i-th entry of sequence S ($1 \leq i \leq Len(S)$)
$S[i:j]$	A subsequence including entries from the i-th one to the j-th
ω	Length of the sliding/disjoint window
s_i	The i-th disjoint window of sequence S
$\langle\rangle$	empty sequence
$Rest(S)$	A subsequence of S, including entries from the second

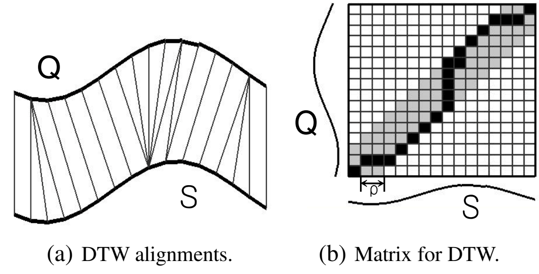
We first define the DTW distance, which is recursively defined as follows. Given two sequences S and Q of the same length, there exist three cases in computing the minimal cost of possible matches of postfixes of S and Q: 1) matching postfixes of S and Q (=$DTW_\rho(Rest(S), Rest(Q))$); 2) matching Q with the postfix of S (=$DTW_\rho(Rest(S), Q)$); or 3) matching S with the postfix of Q (=$DTW_\rho(S, Rest(Q))$) [1].

$$DTW_\rho(S,Q) = \sqrt[p]{D(S[1],Q[1]) + min\begin{cases} DTW_\rho(Rest(S), Rest(Q))^p \\ DTW_\rho(Rest(S), Q)^p \\ DTW_\rho(S, Rest(Q))^p \end{cases}}$$

$$DTW_\rho(\langle\rangle,\langle\rangle) = 0, DTW_\rho(S,\langle\rangle) = DTW_\rho(\langle\rangle, Q) = \infty$$
$$D(S[i],Q[j]) = \begin{cases} |S[i]-Q[j]|^p & \text{if } |i-j| \leq \rho \\ \infty & \text{otherwise} \end{cases} \quad (1)$$

Given two sequences S and Q of the same length, DTW enables "nonlinear" alignments between S and Q. Figure 3(a) illustrates the strength of DTW. Specifically, in Figure 3(b), a cell (i,j) in the matrix represents an alignment between a query point $Q[i]$ and a data point $S[j]$. The goal is to find such a path with minimal warping cost, i.e., the *warping path*. Here, the warping path consists of dark cells in the matrix. In order to prevent pathological warpings where a small portion of one sequence matches with a large portion of another sequence [13], we limit the scope of the warping path using the *warping width* ρ [1, 12, 13, 24], which is known as the Sakoe-Chiba band constraint, where matrix entry (i,j) becomes ∞ if $|i-j| > \rho$. Thus, only shaded cells in the matrix in Figure 3(b) are computed. Note that, the DTW distance becomes the L_p distance when $\rho = 0$.

In computing DTW_ρ, an element $S[i]$ can match one of $Q[i-\rho : i+\rho]$. Thus, for each element in Q, we store the maximum and the



(a) DTW alignments. (b) Matrix for DTW.

Figure 3: Illustration of DTW.

minimum values from $Q[i-\rho : i+\rho]$ and thus compute a lower bound distance using pairwise computation without using expensive dynamic programming. Since we have two values for each query element $Q[i]$, these sequences are called *envelopes* [13]. Figure 4 illustrates the lower and upper envelopes of Q. The shaded area shows the lower-bound distance between Q and S. The concepts of the *query envelope* and the *lower bound distance* are formally described in subsequent paragraphs.

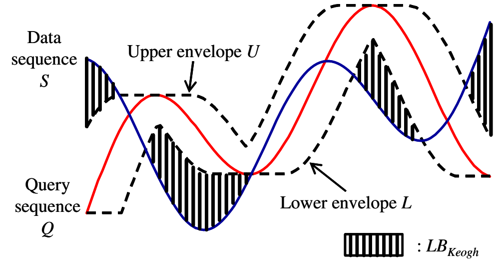


Figure 4: Example of envelopes and LB_{Keogh}.

We now define the notion of query envelope [24], which can be formally defined as below:

Definition 1. *The query envelope of a query Q, $\mathbb{E}(Q)$ consists of the upper and lower envelopes of Q and represents the region defined between the upper envelope U and the lower envelope L. The i-th element $(L[i], U[i])$ in $\mathbb{E}(Q)$ is defined as follows:*

$$L[i] = \min_{-\rho \leq r \leq \rho}(Q[i+r]), \quad U[i] = \max_{-\rho \leq r \leq \rho}(Q[i+r]) \quad \Box$$

The tightest lower bound of DTW, between a query envelope $\mathbb{E}(Q)$ and a data sequence S, is known as LB_{Keogh} [13] below.

$$LB_{Keogh}(\mathbb{E}(Q), S) = \sqrt[p]{\sum_{i=1}^{N} \begin{cases} |S[i]-U[i]|^p & \text{if } S[i] > U[i] \\ |S[i]-L[i]|^p & \text{if } S[i] < L[i] \\ 0 & \text{otherwise} \end{cases}}$$

We now discuss piecewise aggregate approximation (PAA) [13, 23], a technique to reduce the dimensionality of a data sequence S of length N down to f ($f \ll N$), by dividing the sequence of length N into f equal sized segments then storing the mean values. We notate PAA of such S, $\mathcal{P}(S)$, as a time-series [$\overline{S[1]}$,...,$\overline{S[f]}$] of length f, where

$$\overline{S[i]} = \frac{f}{N} \sum_{j=\frac{N}{f}(i-1)+1}^{\frac{N}{f}i} S[j].$$

Similarly, the PAA of the query envelope $\mathbb{E}(Q)$ can be computed, by applying PAA to the upper and lower query envelopes. That is, the i ($1 \leq i \leq f$)-th element ($\overline{L[i]}$, $\overline{U[i]}$) in $\mathcal{P}(\mathbb{E}(Q))$ is:

$$\overline{L[i]} = \frac{f}{N} \sum_{j=\frac{N}{f}(i-1)+1}^{\frac{N}{f}i} L[j], \quad \overline{U[i]} = \frac{f}{N} \sum_{j=\frac{N}{f}(i-1)+1}^{\frac{N}{f}i} U[j].$$

As a lower bound that can be used at the index level, LB_{PAA} [24] is defined as Eq. (2), which is the distance between the query envelope $\mathcal{P}(\mathbb{E}(Q))$ and the PAA of the data sequence $\mathcal{P}(S)$. Figure 5 illustrates the PAAs of a query envelope and the a data sequence and LB_{PAA}.

$$\sqrt[p]{\sum_{i=1}^{f} \frac{N}{f} \begin{cases} |\overline{S[i]} - \overline{U[i]}|^p & \text{if } \overline{S[i]} > \overline{U[i]} \\ |\overline{S[i]} - \overline{L[i]}|^p & \text{if } \overline{S[i]} < \overline{L[i]} \\ 0 & \text{otherwise} \end{cases}} \quad (2)$$

Figure 5: Illustration of LB_{PAA}

To guarantee no false dismissal under DTW, we have the following lemma [24].

Lemma 1. *Given two sequences Q and S of the same length and a warping width ρ, the following equation holds:*

$$DTW_\rho(Q, S) \geq LB_{Keogh}(\mathbb{E}(Q), S) \geq LB_{PAA}(\mathcal{P}(\mathbb{E}(Q)), \mathcal{P}(S))$$

Now, we define the MDMWP-distance in Definition 2 which can be used for pruning during ranked subsequence matching, and Lemma 2 shows the lower boundness of the MDMWP-distance.

Definition 2. [12] *Given a query envelope $\mathbb{E}(Q)$ and a data subsequence $S[i:j]$, if MDMWP of $\mathcal{P}(\mathbb{E}(Q))$ and $\mathcal{P}(S[i:j])$ is $(\mathcal{P}(\mathbb{E}(q_m)), \mathcal{P}(s_m))$, then the MDMWP-distance of $\mathcal{P}(\mathbb{E}(Q))$ and $\mathcal{P}(S[i:j])$ is defined as $\sqrt[p]{r} \times LB_{PAA}(\mathcal{P}(\mathbb{E}(q_m)), \mathcal{P}(s_m))$ where $r = \lfloor (Len(Q)+1)/\omega \rfloor - 1$. That is, r is the minimum number of disjoint windows in $S[i:j]$ regardless of the starting offset i.* □

Lemma 2. [12] *Given a query envelope $\mathbb{E}(Q)$ and a data subsequence $S[i:j]$, the following Eq. (3) holds:*

$$DTW_\rho(Q, S[i:j]) \geq MDMWP\text{–}distance(\mathcal{P}(\mathbb{E}(Q)), \mathcal{P}(S[i:j])) \quad (3)$$

3. RANKED UNION QUERY PROCESSING FRAMEWORK

We propose a systematic framework for ranked union. We first give a formal definition of our problem below. As in [12], in order to match data subsequences of length $l \neq |Q|$, one can scale Q with reasonable scale factors.

Problem Definition 1. [12] *Given n data sequences S_1, ..., S_n of variable lengths, a query sequence Q, a number k, and a warping width ρ, find k-nearest data subsequences for Q by using DTW_ρ.*

We first give an insight into our framework. We notice that we can partition sliding query windows into several groups since 1) query windows in the same group match the same corresponding subsequences, and 2) query windows in different groups do not. Consider the sliding windows in Figure 6. Sliding windows $\mathcal{P}(\mathbb{E}(q_2))$ and $\mathcal{P}(\mathbb{E}(q_6))$ match the same corresponding data subsequences, but $\mathcal{P}(\mathbb{E}(q_1))$ and $\mathcal{P}(\mathbb{E}(q_2))$ do not match. That is, if $\mathcal{P}(\mathbb{E}(q_2))$ matches a disjoint window $S[i+1:i+4]$, $\mathcal{P}(\mathbb{E}(q_6))$ matches a disjoint window $S[i+5:i+8]$. Both match the same subsequence $S[i:i+10]$. For each group, we can find top-k matching data subsequences for the query windows in this group. Thus, if we select top-k results from the results of all groups, these results become the answer to the ranked subsequence matching. Furthermore, if we can derive a lower bound for each group during execution, we can provide a tighter lower bound than MDMWP. We present formal concepts in the next subsections where such the group is formalized using the concept of the equivalence class, since any two windows in the same group are equivalent to each other in that both match the same data subsequence.

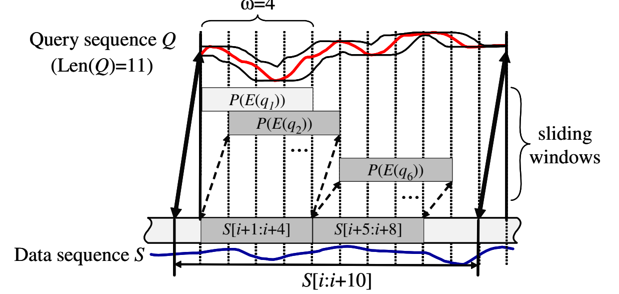


Figure 6: Matching window pairs and their corresponding data subsequences.

3.1 Translating into Ranked Union Queries

Our framework uses the windowing scheme proposed from Dual-Match [17]– This scheme divides data sequences into disjoint windows of the same size ω. Data windows are then transformed into lower-dimensional data points and indexed to an R*-tree. To support DTW, we first construct the query envelope $\mathbb{E}(Q)$ for a query sequence Q, divide the query envelope into sliding windows $\mathbb{E}(q_i)$ of size ω, and transform each query window $\mathbb{E}(q_i)$ into a lower-dimensional window $\mathcal{P}(\mathbb{E}(q_i))$. In the remainder of this paper, we will use the terms "window," "enveloped window," and "point" interchangeably unless necessary for clarity.

Our framework views a ranked subsequence matching query as a ranked union query of subqueries, where each subquery contains only disjoint windows. This enables us to formally analyze the performance issues of HLMJ and develop new techniques for solving these performance issues. Definition 3 provides a formal definition of the ranked union query.

Definition 3. *Given a query sequence Q, a ranked union query over ranked subsequence subqueries merges results from all subqueries and returns k-nearest data subsequences for Q.* □

Definition 4. *Let $\mathcal{QS}$ be the set of all sliding windows for a query envelope $\mathcal{P}(\mathbb{E}(Q))$. Let $\cong$ be an equivalence relation over $\mathcal{QS}$ such that $\mathcal{P}(\mathbb{E}(Q[i:i+\omega-1])) \cong \mathcal{P}(\mathbb{E}(Q[j:j+\omega-1]))$ if and only if $|i-j|$* ***mod*** *$\omega = 0$.* □

The equivalence class of $\mathcal{P}(\mathbb{E}(Q[i:i+\omega-1]))$ is a set of query windows which are equivalent to (i.e., $\cong$) $\mathcal{P}(\mathbb{E}(Q[i:i+\omega-1]))$. This class is called a *matching subsequence equivalence class* (*MSEQ*) since all windows in an MSEQ match the same data subsequences. For convenience of notation, we use $MSEQ_i$ $(1 \leq i \leq \omega)$ to denote the equivalence class of $\mathcal{P}(\mathbb{E}(Q[i:i+\omega-1]))$. $MSEQ_{i,j}$ denotes a query window $\mathcal{P}(\mathbb{E}(Q[(i+(j-1)\times\omega) : (i+(j-1)\times\omega)+\omega-1]))$. Consider a query sequence Q in Figure 2. The length of the query sequence is 11, and ω is set to 4. The equivalence class of $\mathcal{P}(\mathbb{E}(Q[1:\omega]))$, denoted as $[\mathcal{P}(\mathbb{E}(Q[1:\omega]))]_{\cong}$, consists of two windows, $\{\mathcal{P}(\mathbb{E}(Q[1:\omega])), \mathcal{P}(\mathbb{E}(Q[5:\omega]))\}$. In this example, there exist four equivalence classes under $\cong$, and each class has two windows.

In the proposed framework, we try to evaluate a ranked subsequence matching query using the corresponding ranked union

query. To do this, we need to show that both queries produce the same result, that is, their query results are equivalent to each other. The following lemma shows this equivalence.

Lemma 3. *The evaluation result of a ranked subsequence matching query using sliding windows of size ω is equivalent to that of the ranked union query over ω ranked subsequence matching subqueries, where each subquery corresponds to an MSEQ.*

PROOF: To evaluate a ranked subsequence matching query correctly, we need to find k-nearest data subsequences by considering all possible data subsequences. Also, to guarantee that a ranked union query of Definition 3 produces the same result by the ranked subsequence matching query, we need to show that the results of subqueries of MSEQs cover all possible data subsequences. We thus prove every data subsequence is a result of one subquery (i.e., an MSEQ) of the ranked union query.

We prove it using Figure 7. We first denote the nth sliding window of a query Q by q_n and the mth disjoint window of a data sequence S by s_m. As illustrated in Figure 7, suppose that the jth query window $MSEQ_{i,j}$ of $MSEQ_i$ matches with the mth data disjoint window s_m. We note here that $MSEQ_{i,j}$ corresponds to the $swn(i,j)^{th}$ sliding window $q_{swn(i,j)}$, where $swn(i,j) = (j-1) \times \omega + i$, and swn stands for sliding window number. Then, the starting offset of the subsequence corresponding to this matching window pair is smaller by $swn(i,j)$ compared to the offset of s_m. That is, the start offset of the matching data subsequence is (m-1)×ω-($swn(i,j)$-1)+1, and the subsequence itself is S[(m-1)×ω-($swn(i,j)$-1)+1:(m-1)×ω-($swn(i,j)$-1)+$Len(Q)$]. According to Definition 4, we have ω MSEQs, i.e., $1 \leq i \leq \omega$. Thus, every subsequence of S with $Len(Q)$ is a result of $MSEQ_i$ ($1 \leq i \leq \omega$). This completes the proof. □

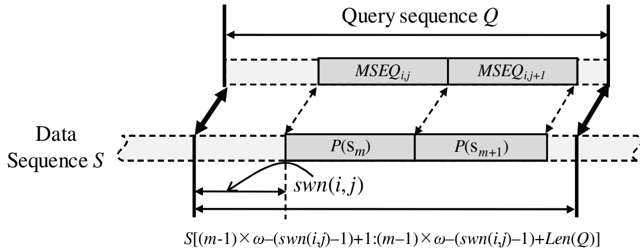


Figure 7: Matching window pairs.

3.2 Processing of Ranked Union Query

With our problem translated into a ranked union query problem, we now discuss an efficient execution model.

3.2.1 Extended Iterator Model

To process a ranked union query, we generate a query execution plan (QEP) using physical operators. Algorithms for executing such a plan can be constructed using the iterator model [8] commonly used in query processors of commercial DBMSs. In the iterator model, the operators composing the QEP receive a tuple from the child operators, which is then passed on to their parent operator if necessary. To obtain one tuple at a time from each operator, the operator provides the $GetNext()$ function as the interface. Each operator also provides the $Start()$ and $End()$ functions to initialize and finalize the operator. In our framework, a subsequence is modeled as a tuple.

In processing a ranked query, the scheduling order for consuming the priority queues from child nodes significantly affects the overall performance. Thus, we need to carefully control the scheduling of the priority queue selection, which motivates us to extend the iterator model. For such optimization, whenever each operator consumes an entry from a priority queue, it returns a lower bound as a "scheduling hint" even when no subsequence is ready to be returned. This enables finer granular scheduling.

Definition 5. *Each operator in the* ***extended iterator model*** *returns a pair of ⟨status, obj⟩. Status is one of the following values:*

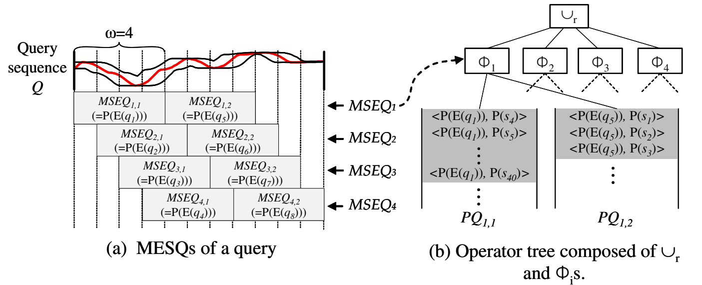
- *TUPLE: obj contains the next result tuple.*
- *LB: The next result tuple is yet to be found, and obj contains the lower bound of the next result tuple instead.*
- *EOR: all tuples of the child operator are consumed.* □

3.2.2 Execution Model of Ranked Union

Our execution model provides two novel operators which follow the extended iterator model: 1) multi-way ranked union operator denoted as $\cup_r$ and 2) ranked subsequence matching subquery operator denoted as Φ. $\cup_r$ merges the results from ω subqueries and returns top-k subsequences. $\cup_r$ maintains the current top-k subsequences obtained thus far along with the current lower bounds for the subqueries. Thus, $\cup_r$ stops the processing of subqueries whenever the distance of the top k-th subsequence obtained so far $\leq$ all the current lower bounds of the subqueries. Φ_i (corresponding to the i-th subquery) allocates a priority queue $PQ_{i,j}$ for each query window $MSEQ_{i,j}$ in $MSEQ_i$, which maintains matching pairs of $MSEQ_{i,j}$ with data points or MBRs in distance order.

Figure 8 shows an example of how the ranked subsequence query Q in Figure 2 is executed using ranked union. Since there are four MSEQs for Q, $\cup_r$ has four child operators, each of which corresponds to an MSEQ. Φ_1 has two priority queues, $PQ_{1,1}$ and $PQ_{1,2}$. $\cup_r$ selects one child operator Φ_i, and Φ_i then selects a priority queue $PQ_{i,j}$ to find a matching window pair of $MSEQ_{i,j}$ with a data point or an MBR. If the matching window pair contains a data point, Φ_i computes a lower bound distance called the MSEQ-distance. If this lower bound distance is greater than the DTW distance between the query sequence and the top k-th data subsequence obtained thus far, that pair is pruned. Otherwise the corresponding data subsequence for the data point is accessed to compute the DTW distance. We repeatedly call $\cup_r$ until it finds top-k subsequences.

More specifically, the main algorithm invokes $\cup_r.GetNext()$ to retrieve top-k subsequences one at a time. $\cup_r.GetNext()$ chooses one of the subqueries and invokes $\Phi_i.GetNext()$. In $\Phi_i.GetNext()$, we choose one of its priority queues, based on a scheduling strategy we will discuss later. Entries in the chosen queue are then consumed to return either a new tuple or a new lower bound.



(a) MESQs of a query

(b) Operator tree composed of $\cup_r$ and Φ_is.

Figure 8: Illustration of the execution model.

Our execution model, together with the concept of the MSEQ, enables us to significantly prune the index search space of ranked union. Since $PQ_{i,j}$ of Φ_i maintains all matching pairs to be popped in the order of distances, we can then compute a lower bound of the distance between the query sequence and the corresponding subsequence in advance. To further illustrate this concept, we provide Definition 6 and Lemma 4.

Definition 6. *Suppose that the current top entry of a priority queue $PQ_{i,j}$ of Φ_i contains a matching pair of $MSEQ_{i,j}$ and obj_j. MINDIST is a lower bounding distance between an MBR and a transformed window [12]. Then, the MSEQ-distance between $MSEQ_{i,j}$ and obj_j is $\sqrt[p]{\sum_{j=1}^{|MSEQ_i|}\mathcal{D}(MSEQ_{i,j}, obj_j)^p}$,*

$$\text{where } \mathcal{D} = \begin{cases} \text{MINDIST} & \text{, if } obj_j \text{ is an MBR} \\ LB_{PAA} & \text{, if } obj_j \text{ is a leaf entry.} \end{cases}$$

Lemma 4. *Suppose a matching pair mp is to be popped from a priority queue $PQ_{i,j}$ of Φ_i such that mp contains a leaf entry, and its corresponding subsequence $S[a:b]$ has not been retrieved yet. Then, $DTW_\rho(Q, S[a:b]) \geq$ MSEQ-distance(mp).*

PROOF: By Lemma 2, we have the following inequality.

$$DTW_\rho(Q,S) \geq LB_{PAA}(\mathcal{P}(\mathbb{E}(Q)), \mathcal{P}(S[a:b]))$$

Since the data subsequence $S[a:b]$ matches $MSEQ_i$, $S[a:b]$ must include $|MSEQ_i|$ disjoint windows, (possibly null) subsequences s_h (at the head), and s_t (at the tail). Thus, $S[a:b]$ can be represented as $s_h s_1 ... s_{|MSEQ_i|} s_t$. Similarly, $\mathbb{E}(Q)$ can be represented as $\mathbb{E}(q_h q_i ... q_{(i+|MSEQ_i|-1)} q_t)$. Thus, we have

$$LB_{PAA}(\mathcal{P}(\mathbb{E}(Q)), \mathcal{P}(S[a:b])) =$$
$$LB_{PAA}(\mathcal{P}(\mathbb{E}(q_h q_i ... q_{(i+|MSEQ_i|-1)} q_t)), \mathcal{P}(s_h s_1 ... s_{|MSEQ_i|} s_t))$$

Here, if we omit the two matching subsequence pairs, $(\mathbb{E}(q_h), s_h)$ and $(\mathbb{E}(q_t), s_t)$, from the above equation, we have the following inequality, since LB_{PAA} is a monotonic increasing function of the sequence length:

$$\geq LB_{PAA}(\mathcal{P}(\mathbb{E}(q_i \dots q_{(i+|MSEQ_i|-1)})), \mathcal{P}(s_1 \dots s_{|MSEQ_i|}))$$

$$= \sqrt[p]{\sum_{k=1}^{|MSEQ_i|} LB_{PAA}(MSEQ_{i,k}, \mathcal{P}(s_k))^p}$$

Since $S[a:b]$ has not been retrieved yet, the distance of the top entry of $PQ_{i,k}$, $(= \mathcal{D}(MSEQ_{i,k}, obj_k)) \leq LB_{PAA}(MSEQ_{i,k}, \mathcal{P}(s_k))$, where $k \neq j$. Then, the following equations hold:

$$\geq \sqrt[p]{\sum_{k=1}^{|MSEQi|} \begin{cases} \mathcal{D}(MSEQ_{i,k}, obj_k))^p & \text{, if } k \neq j \\ LB_{PAA}\,(mp)^p & \text{, if } k = j \end{cases}}$$

$$= \text{MSEQ-distance}(mp) \qquad \square$$

The following Lemma 5 shows that the MSEQ-distance is at least as tight as the MDMWP-distance when MDMWP-based scheduling is used for our ranked union framework.

Lemma 5. *Suppose that MDMWP-based scheduling is used, and that a matching pair $(MSEQ_{i,j}, \mathcal{P}(s_k))$ is to be popped from a priority queue $PQ_{i,j}$ of Φ_i where its corresponding subsequence $S[a:b]$ has not been retrieved yet. Then, MSEQ-distance $(MSEQ_{i,j}, \mathcal{P}(s_k)) \geq$ MDMWP-distance$(MSEQ_{i,j}, \mathcal{P}(s_k))$.*

PROOF: The MDMWP-distance of a matching pair $(MSEQ_{i,j}, \mathcal{P}(s_k))$ is defined as $\sqrt[p]{\lfloor (Len(Q)+1)/\omega \rfloor - 1} \times LB_{PAA}(MSEQ_{i,j}, \mathcal{P}(s_k))$. First, $|MSEQ_i| \geq \lfloor (Len(Q)+1)/\omega \rfloor - 1$. Next, since MDMWP-based scheduling is used, $LB_{PAA}(MSEQ_{i,j}, \mathcal{P}(s_k))$ is less than or equal to the distance of the current top entry in $PQ_{i,k}$.Thus, MSEQ-distance$(MSEQ_{i,j}, \mathcal{P}(s_k)) \geq$ MDMWP-distance$(MSEQ_{i,j}, \mathcal{P}(s_k))$. $\square$

Now, we explain how to select one of the subqueries of Φ_i. To minimize the cost of the ranked union execution model, any unnecessary $\Phi_i.GetNext()$ invocations must be avoided. We have the following Lemma 6 for this purpose.

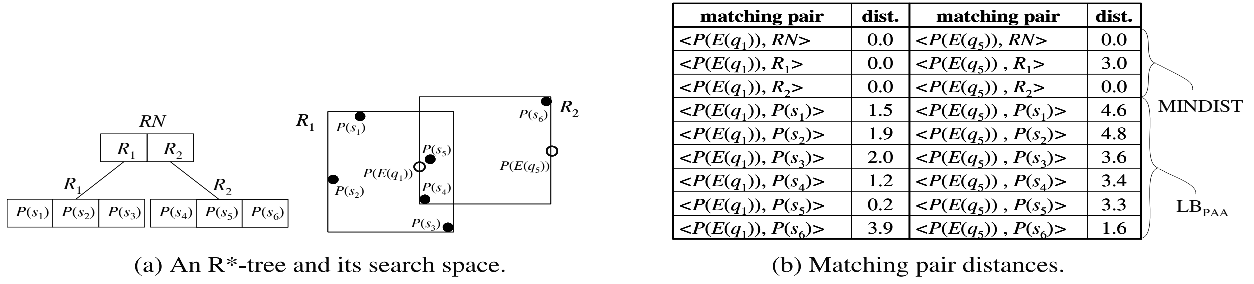
Lemma 6. *Let CLB_i be a current lower bound for Φ_i. To minimize the total number of $\Phi.GetNext()$ invocations, $\cup_r$ must choose Φ_j such that $j = \arg\min_{i=1}^{\omega}(CLB_i)$.*

PROOF: Let δ_{cur} represent the DTW$_\rho$ distance between the query sequence and the top k-th data subsequence obtained so far. Until δ_{cur} converges to the distance between the query sequence and the actual top-kth subsequence, the termination condition $\min_{i=1}^{\omega} CLB_i \geq \delta_{cur}$ does not hold. This condition cannot be reached without selecting Φ_j such that $j = \arg\min_{i=1}^{\omega}(CLB_i)$. $\square$

Algorithm for $\cup_r.Start()$: $\cup_r$ maintains three global variables: δ_{cur}, $candMinQ_{\cup_r}$, and $CLBs$. δ_{cur} represents the DTW distance between Q and the top k-th subsequence obtained so far. $candMinQ_{\cup_r}$ is implemented as a minimum priority queue whose entry is a candidate subsequence. $CLBs$ is a set of current lower bounds of subqueries. Initially, since there is no candidate obtained so far, δ_{cur} is set to ∞, $candMinQ_{\cup_r}$ is set to empty, and each CLB_i is set to zero. We then call $Start()$ of each child Φ_i of $\cup_r$.
Algorithm for $\Phi_i.Start()$: In order to find the nearest leaf entries (i.e., f-dimensional points) for each window $MSEQ_{i,j}$, Φ_i allocates a minimum priority queue $PQ_{i,j}$ for each query window in Φ_i. An entry in $PQ_{i,j}$ is in the form of a triple $\langle MSEQ_{i,j}, obj, dist\rangle$. Here, $MSEQ_{i,j}$ and obj are a matching pair where obj is either a subsequence, a leaf entry, a leaf node, or a non-leaf node; $dist$ is the distance between the matching pair. Φ_i also maintains a local minimum priority queue $candMinQ_{\Phi_i}$ to store top-k candidates for Φ_i. $\Phi_i.Start()$ pushes each pair of $MSEQ_{i,j}$ and the root node MBR annotated with their MINDIST distances.
Algorithm for $\cup_r.GetNext()$: If the distance of the top entry in $candMinQ_{\cup_r} \leq$ all CLB_i $(1 \leq i \leq \omega)$, we return a subsequence as a tuple by popping the top entry of $candMinQ_{\cup_r}$. Next, we select a subquery Φ_i according to Lemma 6. Then, we invoke $\Phi_i.GetNext()$. If Φ_i generates a subsequence (i.e., $status$ is equal to TUPLE), we update the lower bound of $CLB_i.dist$ with $obj.dist$ and push the subsequence into the $candMinQ_{\cup_r}$ if $obj.dist \leq \delta_{cur}$. If Φ_i generates a new lower bound, we update the lower bound of $CLB_i.dist$ with $obj.dist$. If Φ_i returns EOR, we remove CLB_i from $CLBs$ since Φ_i will return no more candidates. (A detailed algorithm is in the Appendix in [11]).
Algorithm for $\Phi_i.GetNext()$: We denote as MSEQ-dist$_{\text{next}}$, the MSEQ-distance for the entry to be popped next from a $PQ_{i,j}$ in Φ_i, i.e, the lower bound distance for the entry to be popped next. If $candMinQ_{\Phi_i}$.Top().$dist$ (=the distance of the top entry of $candMinQ_{\Phi_i}$) $> \delta_{cur}$ and so is MSEQ-dist$_{\text{next}}$, we can guarantee that Φ_i does not contribute in generating top-k candidates. If $candMinQ_{\Phi_i}$.Top().$dist \leq$ MSEQ-dist$_{\text{next}}$, we pop an entry from $candMinQ_{\Phi_i}$ and return it as a tuple; otherwise, we pop an entry from one of the priority queues using $SelectPriorityQueue()$. Here, an entry contains either a node or a leaf entry. We then perform one of the following actions depending on the type of the pair:

- *matching pair with a leaf/non-leaf node*: Retrieve the entries in the node and push the pairs of these entries along with the query windows annotated with their MSEQ-dist distances into the priority queue. Here, pairs whose MSEQ-dist $> \delta_{cur}$ are prunned.
- *matching pair with a leaf entry*: Retrieve the matching subsequence and compute the LB_{Keogh} and DTW$_\rho$ distances. If both distances $\leq \delta_{cur}$, push back to the priority queue.

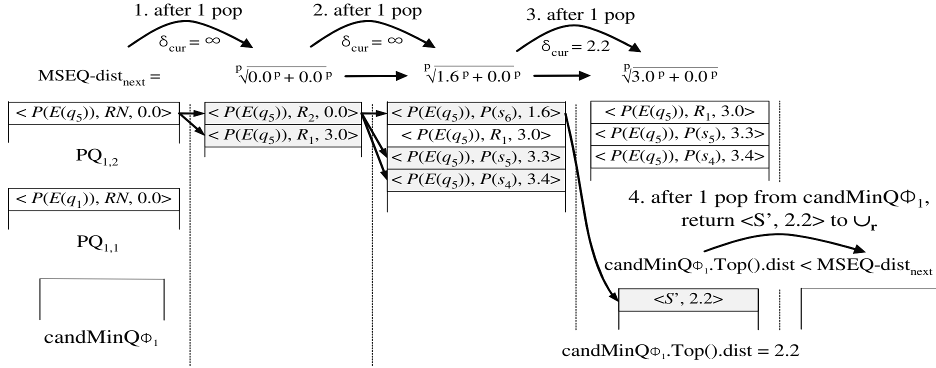
Lastly, we compute MSEQ-dist$_{\text{next}}$ and return a new lower bound value to $\cup_r$. (A detailed algorithm is in the Appendix in [11]).
Example: Figure 9 illustrates how our rank union framework operates using an example, where a data sequence S of length 27 and a query sequence Q of length 11 are divided into disjoint and sliding windows of size $\omega = 4$. Suppose that k is set to 1. As illustrated in Figure 9(a), S is divided into six disjoint windows $\{s_1, \cdots, s_6\}$,



matching pair	dist.	matching pair	dist.
$<P(E(q_1)), RN>$	0.0	$<P(E(q_5)), RN>$	0.0
$<P(E(q_1)), R_1>$	0.0	$<P(E(q_5)), R_1>$	3.0
$<P(E(q_1)), R_2>$	0.0	$<P(E(q_5)), R_2>$	0.0
$<P(E(q_1)), P(s_1)>$	1.5	$<P(E(q_5)), P(s_1)>$	4.6
$<P(E(q_1)), P(s_2)>$	1.9	$<P(E(q_5)), P(s_2)>$	4.8
$<P(E(q_1)), P(s_3)>$	2.0	$<P(E(q_5)), P(s_3)>$	3.6
$<P(E(q_1)), P(s_4)>$	1.2	$<P(E(q_5)), P(s_4)>$	3.4
$<P(E(q_1)), P(s_5)>$	0.2	$<P(E(q_5)), P(s_5)>$	3.3
$<P(E(q_1)), P(s_6)>$	3.9	$<P(E(q_5)), P(s_6)>$	1.6

MINDIST (rows RN–R_2), LB_{PAA} (rows $P(s_1)$–$P(s_6)$)

(a) An R*-tree and its search space. (b) Matching pair distances.



(c) Changes to $PQ_{1,j}$ and candMinQ$_{\Phi_1}$ over time.

Figure 9: An example of the ranked union framework.

and each window of size 4 is transformed into a lower-dimensional data point (say, a 2D point). The points are indexed in an R*-tree with two MBRs– MBR R_1 with $\{\mathcal{P}(s_1), \mathcal{P}(s_2), \mathcal{P}(s_3)\}$ and MBR R_2 with $\{\mathcal{P}(s_4), \mathcal{P}(s_5), \mathcal{P}(s_6)\}$. The root node *RN* consists of these two MBRs. Figure 9(c) describes the changes in $PQ_{1,j}$ and $candMinQ_{\Phi_1}$ over time. We illustrate two query points, q_1 and q_5, and assume $SelectPriorityQueue()$ always chooses $PQ_{1,2}$. Since $\omega = 4$, $MSEQ_{1,1} = \mathcal{P}(\mathbb{E}(q_1))$ and $MSEQ_{1,2} = \mathcal{P}(\mathbb{E}(q_5))$, as shown in Figure 8. Note that the ranked union framework can terminate the processing much earlier than HLMJ, i.e., after only 4 pops as opposed to 12 pops required by HLMJ!

• initial state: $PQ_{1,1}$ and $PQ_{1,2}$ are initialized with the pairs of RN with $\mathcal{P}(\mathbb{E}(q_1))$ and $\mathcal{P}(\mathbb{E}(q_5))$, annotated with their MINDISTs. That is, $\langle \mathcal{P}(\mathbb{E}(q_1)), RN, 0.0\rangle$ and $\langle \mathcal{P}(\mathbb{E}(q_5)), RN, 0.0\rangle$ are inserted into $PQ_{1,1}$ and $PQ_{1,2}$. Initially, $candMinQ_{\Phi_1}$ is empty.

• after 1 pop: After $\langle \mathcal{P}(\mathbb{E}(q_5)), RN, 0.0\rangle$ is popped, it is expanded into $\mathcal{P}(\mathbb{E}(q_5)) \times \{R_1, R_2\}$, which are inserted back into $PQ_{1,2}$.

• after 2 more pops: After $\langle \mathcal{P}(\mathbb{E}(q_5)), R_2, 0.0\rangle$ is popped, it is expanded into $\{\mathcal{P}(\mathbb{E}(q_5))\}\}\times\{\{\mathcal{P}(s_4), \mathcal{P}(s_5), \mathcal{P}(s_6)\}$, which are inserted back into $PQ_{1,2}$. Next, $\langle \mathcal{P}(\mathbb{E}(q_5)), \mathcal{P}(s_6), 1.6\rangle$ is popped. Since $\mathcal{P}(s_6)$ is a leaf entry, we retrieve the corresponding subsequence S' from the database, which then is inserted into $candMinQ_{\Phi_1}$.

• after 1 more pop: Since $candMinQ_{\Phi_1}$.Top().$dist <$ MSEQ-dist$_{next}$ (i.e., $2.2 < \sqrt[p]{3^p}$), we pop an entry from $candMinQ_{\Phi_1}$ and return it to $\cup_r$ as a tuple. □

Now, we explain how to select a priority queue using $SelectPriorityQueue()$. Although DTW and LB_{PAA} distances themselves are not monotonic, our proposed translation to the ranked union framework enables the adoption of scheduling strategies developed for ranking queries, such as the scheduling heuristics of [10]. We thus adopt one such strategy for choosing a priority queue $PQ_{i,j}$ where the difference in the distances between the last top entry and the current top entry is the maximum.

Meanwhile, implementing MDMWP-based scheduling may incur serious performance degradation when some query points are placed in dense regions while other points are placed in sparse regions, as explained in Section 1. This observation motivates us to develop a density-based scheduling method, which we describe in the next section.

4. COST-AWARE DENSITY-BASED SCHEDULING WITH SELECTIVE EXPANSION

We propose a novel priority queue selection strategy called RU-COST, exploiting cost-aware scheduling with selective expansion. This method avoids the MDMWP-based scheduling problem by minimizing the cost of accessing entries from priority queues. Unlike scheduling problems explored for TA-family algorithms, this problem poses two additional challenges: 1) the list for each query window is dynamically generated; and 2) the cost of generating one leaf entry from the priority queue varies significantly, depending on how many MBRs must be expanded to reach the leaf entry.

Intuitively, a good scheduling algorithm will identify a queue whose top-h leaf entries are sparsely populated near the query point, as noted in Section 1. For example, consider the ranked union execution framework in Figure 8. Clearly, selecting $PQ_{1,2}$ rather than $PQ_{1,1}$ would quickly grow the MSEQ-distance for Φ_1. However, if the disk I/O cost of accessing the corresponding data subsequences for the top-h leaf entries in $PQ_{1,1}$ is much cheaper than that of $PQ_{1,2}$, selecting $PQ_{1,1}$ would seem to be more beneficial. To solve this challenging problem, We propose a novel concept of cost-aware density. We formally define this concept in Definition 7.

Definition 7. *Let the most recently popped leaf entry from $PQ_{i,j}$ be le_p. Let the next top-h leaf entries to be popped from $PQ_{i,j}$ be $le_1, le_2, \cdots, le_h$. Then, the cost-aware density $CDens(PQ_{i,j})$ is defined as follows. NUM_IO returns the number of I/Os when accessing the subsequences corresponding to $le_1, le_2, \cdots, le_h$.*

$$\frac{\alpha \times NUM_IO(le_1, le_2, \cdots, le_h) + \beta \times h}{LB_{PAA}(MSEQ_{i,j}, le_h) - LB_{PAA}(MSEQ_{i,j}, le_p)},$$

where α and β are user-specified constant values.

Note that, the denominator of $CDens(PQ_{i,j})$ (or "volume" in the density) is the difference in LB_{PAA} between the last popped entry le_p and the top h^{th} entry le_h, as leaf entries are sorted in one-dimensional distance space using LB_{PAA}. The numerator of $CDens(PQ_{i,j})$ (or "mass" in the density) is the cost for leaf entries, which is a combination of the cost of disk I/Os NUM_IO and the cost of computing distances. As disk I/Os are dominant in performance of ranked union, we set α to 1 and β to 0. Finding more accurate values of α and β will be our future work. Note also that, when there is a need to choose among queues with zero density, we choose the one with the smallest denominator. For efficient computation of NUM_IO, we allocate a bitmap for pages, where each bit is set to 1 when the corresponding page is in the buffer. Thus, we can accurately count entries whose corresponding subsequences are not in the buffer. The memory overhead for the bitmap is marginal, as it requires only 8KBytes for a 256MBytes database.

However, two challenges still remain: First is the question of the optimal lookahead h. If h is too small, the estimated value cannot reflect the overall density. Meanwhile, if it is too large, scheduling does not adapt well to changes in densities, leading to suboptimal performance. To address this challenge, we considered an adaptive approach setting $h = 1$ and increasing it dynamically. However, in our extensive experiments with many real datasets, if h is set to the blocking factor of index pages, the overall performance is very stable regardless of datasets. We therefore set h as the blocking factor.

The second challenge is how to efficiently compute $CDens(PQ_{i,j})$. One naive way is to obtain the next top-h leaf entries from all priority queues of Φ_i and compute exact densities. However, this may impose serious performance overhead if a query point is placed in a dense region, in which case, all index pages in the dense region need to be accessed to obtain the top-h leaf entries.

To address the second challenge, we propose a novel *selective expansion* technique of priority queues, which consists of two steps: 1) we select one priority queue as a pivot and obtain the top-h leaf entries from this pivot queue only to compute its exact density $CDens(PQ_{i,pivot})$. 2) For the remaining queues, we selectively expand them, by choosing only the queue $PQ_{i,j}$ such that LB_{CDens} $(PQ_{i,j}) < CDens(PQ_{i,pivot})$; that is, we stop expanding priority queues whose $LB_{CDens} \geq$ the $CDens$ of the pivot priority queue. After expanding MBRs in $PQ_{i,j}$, if $CDens(PQ_{i,j}) < CDens(PQ_{i,pivot})$, $PQ_{i,j}$ becomes the pivot. We iterate this selective expansion process until $CDens(PQ_{i,pivot}) \leq LB_{CDens}$ $(PQ_{i,j})$ or $CDens(PQ_{i,j})$ for any $j \neq pivot$.

Now, we define $LB_{CDens}(PQ_{i,j})$ in the following definition.

Definition 8. *Let the most recently popped leaf entry from $PQ_{i,j}$ be le_p. Let $le'_1, le'_2, \cdots, le'_{m-1}, MBR_1, \cdots, le'_m, \cdots, le'_h$ be the entries in the priority queue ordered by their distances where le'_h is the top-h^{th} leaf entry in the priority queue. Then, $LB_{CDens}(PQ_{i,j})$ is defined as following :*

$$\frac{\alpha \times NUM_IO(le'_1, le'_2, \cdots, le'_{m-1}) + \beta \times h}{LB_{PAA}(MSEQ_{i,j}, le'_h) - LB_{PAA}(MSEQ_{i,j}, le_p)}$$

Lemma 7. $LB_{CDens}(PQ_{i,j}) \leq CDens(PQ_{i,j})$

PROOF: Let le_p denote the last popped leaf entry from $PQ_{i,j}$ and $le_1, \cdots, le_h$ denote the real top-h leaf entries in order of distance. When le'_h is the hth leaf entry in the current queue, and its preceding entries are $le'_1, \cdots, le'_{m-1}, MBR_1, \cdots, le'_m, \cdots$, since MBR_1 must be extended to find real top-h leaf entries, $LB_{PAA}(MSEQ_{i,j}, le_h) \leq LB_{PAA}(MSEQ_{i,j}, le'_h)$. However, as such expansion cannot affect the ordering of the preceding entries $le'_1, \cdots, le'_{m-1}$, we can conclude that $(le_1, \cdots, le_{m-1})$ are equivalent to $(le'_1, \cdots, le'_{m-1})$. That is, $NUM_IO\,(le_1, \cdots, le_h) \geq NUM_IO\,(le'_1, \cdots, le'_{m-1})$. Therefore, we can conclude that $LB_{CDens}(PQ_{i,j}) \leq CDens(PQ_{i,j})$. □

For this selective expansion to be effective, we must avoid selecting an initial pivot with a very high density. To efficiently approximate density for pivot selection, we first access a set of leaf-node MBRs by accessing only the parent non-leaf nodes of the leaf nodes. For each query window $\mathcal{P}(\mathbb{E}(q))$, we can then determine the distance range $[d_1,\ d_2]$ for each leaf-node MBR by computing MINDIST and MAXDIST. Here, MINDIST and MAXDIST are the minimum and maximum distances between an MBR and a query window, respectively. For simplicity, we assume that the distances between leaf entries and the query envelope are uniformly distributed in the range $[d_1,\ d_2]$. After computing the distance range for each leaf MBR, we merge all distance ranges into a global range. Thus, we can estimate the distance range of the top-h entries of the priority queue using this global range.

Figure 10 illustrates how our proposed selective expansion technique works to find real top-4 leaf entries (i.e., $h = 4$). In the figure, the distances of the last popped leaf entries le_p from $PQ_{i,1}$ and $PQ_{i,2}$ are 4.0 and 3.0 respectively. Leaf entries incurring I/O costs are shaded. The first step of the selective expansion technique is to find a pivot queue, from which we find top-4 entries. In this figure, $PQ_{i,1}$ is chosen, and four leaf entries $(le_1, \cdots, le_4)$ are accessed. In this case, based on Definition 7, $CDens(PQ_{i,1})$ is computed as $\frac{2}{9.0-4.0}$ (when $\alpha = 1$ and $\beta = 0$). In the second step, we compute $LB_{CDens}(PQ_{i,j})$ to selectively expand another queue $PQ_{i,2}$. Observe from this example that $PQ_{i,2}$ has entries $(le'_1, le'_2, MBR_1, MBR_2, \cdots, le'_3, le'_4)$ with distance values $(3.1, 3.2, 3.3, 3.4, \cdots, 7.0, 9.0)$. In this queue, $LB_{CDens}(PQ_{i,2})$, stated in Definition 8, is computed as $\frac{2}{9.0-3.0}$. Since $CDens(PQ_{i,1}) > LB_{CDens}(PQ_{i,2})$, we expand MBR_1, after which a leaf entry with distance 3.3 is inserted into the queue. This updates the queue entries to $(le'_1, le'_2, le'_3, MBR_2, \cdots, le'_4)$ with distances (3.1, 3.2, 3.3, 3.4, $\cdots$, 7.0) and also updates $LB_{CDens}(PQ_{i,2})$ to $\frac{2}{7.0-3.0}$. As $CDens(PQ_{i,1}) < LB_{CDens}(PQ_{i,2})$, we can stop expanding $PQ_{i,2}$.

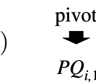
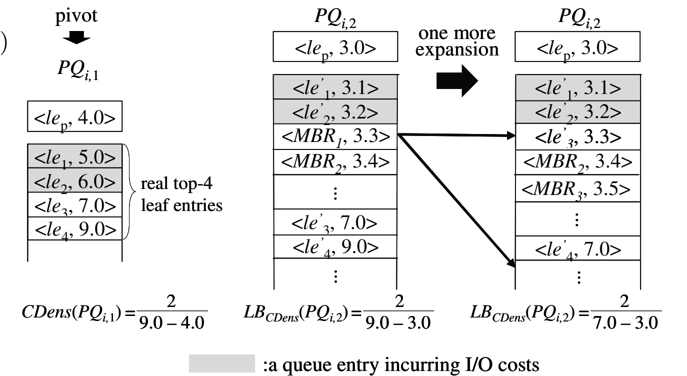



Figure 10: An example of the selective expansion technique.

5. RELATED WORK

Range subsequence matching and its solution under the Euclidean distance were first proposed by Faloutsos et al. [7] (FRM). FRM exploits the window construction mechanism of dividing data sequences into sliding windows and a query sequence into disjoint windows. FRM uses an R*-tree with the lower-dimensional transformation to store the sliding windows of data sequences and to find candidate subsequences from the disjoint windows of a query

sequence. DualMatch [17] and GeneralMatch [16], improve this subsequence matching performance by using window construction methods which differ that of FRM. DualMatch divides the data sequences into disjoint windows and the query sequence into sliding windows; GeneralMatch generalizes the sliding and disjoint windows. The approach proposed here is quite different from these solutions: 1) we deal with the *ranked* subsequence matching rather than the *range* subsequence matching; and 2) we use the DTW distance.

Regarding the DTW distance, [21] proposed a range subsequence matching solution based on FRM. However, their method cannot be used for the ranked subsequence matching due to a loose lower bound and dimensionality curse on long queries [12]. [1] proposed a technique called *anticipatory DTW* to speed up DTW calculations by progressively computing DTW matrix cells. This technique is orthogonal to index-based ranked subsequence matching [1]. [2] proposed an *approximate* subsequence matching using embedding. Unlike our solution, this approximate method allows false dismissals, which cannot be used for many important time-series applications, such as electrocardiogram and gas pipeline inspection. [12] proposed an *exact* solution by introducing two tight lower bounds, the *mdmwp-distance* and the *window-group distance*. If the data points are uniformly distributed in the indexing space, this solution works well; if not, however, it shows bad performance due to the MDMWP-scheduling problem.

Top-k processing in other applications has been actively studied lately.Pioneering algorithms for the efficient processing of such queries are known as TA-family algorithms [6], followed by more algorithms [5, 9, 20] which consider a wide range of access scenarios. However, these algorithms cannot be readily applied to our proposed problem, as they assume the ranking function is monotonic over attribute values.

Using rank-awareness, Li et al. [15] extended the original semantics of relational operators including the union operator. However, our framework is significantly different from this framework: 1) they assume the ranking function is monotonic; 2) as opposed to their ranked union operator which first obtains top-k results for each *relational subquery* and then returns global top-k results (coarse-level scheduling), our ranked union framework provides a tailored execution model for *subsequence matching subqueries* by exploiting finer granular scheduling; 3) For fast subsequence matching, our framework presents a novel subsequence matching operator Φ and its lower bound MSEQ-distance. Thus, our underlying processing mechanism is completely different from their framework which is based on relational operators.

More recently, state-of-the art algorithms for supporting non-monotonic ranking functions have been studied [22]. These algorithms support non-monotonic functions using an index-merge framework with B+-trees or R*-trees. Thus, they can be adapted to ranked subsequence matching using R*-trees. To join multiple indexes efficiently, they first enumerate candidate join states progressively by increasing order of their lower bound scores from the root nodes of the indexes. For each join state, they check the state, whether it can produce the join results, by computing the join signature. If the join state cannot produce the join results, they stop enumerating the join states further. However, as we will see in Section 6, computing join signatures requires prohibitive bloom filter calls if the number of joins is larger than four (i.e., the number of query windows > 4 in our target problem). More specifically, if there are n joining indexes whose fan-outs are f, in the worst case, we need to invoke bloom filters f^n times to compute all valid join signatures from one join state.

6. PERFORMANCE EVALUATION

We evaluate the performance of the state-of-the-art algorithms against our ranked union algorithms. The algorithms considered are as follows: (1) SeqScan, a sequential scan algorithm, (2) HLMJ, (3) PSM, a top-k subsequence matching solution adopted from the recent index-based merge algorithm [22], (4) RU, the ranked union with the default scheduling strategy, and (5) RU-COST, the ranked union with the cost-aware density-based scheduling strategy.

Our main objective is to show that our algorithms 1) *consistently* outperform the competing algorithms even for query sets having no dense windows and 2) *significantly* outperform the competing algorithms for query set having dense windows.

6.1 Experiment Setup

We use five real and synthetic data sets as shown in Table 2. Each data set consists of a long data sequence and has the same effect as one consisting of multiple data sequences [7, 16]. (Refer to the Appendix in [11] for the detailed explanation on data sets.) We use the number of candidates, the number of page accesses, and the wall clock time as the performance metrics. Table 3 summarizes parameters and their values. We generate query sequences from the data sequence by taking subsequences of length $Len(Q)$ starting from random offsets as in [7, 12, 16]. Regarding the query types of the PIPE data set, refer to the Appendix in [11]. We measure the three performance metrics using all query sequences, and use their averages.

Table 2: Data sets used.

Data set	UCR	PIPE	WALK	STOCK	MUSIC
Size($\times 1,000$)	1,056	24,307	1,000	328	2,373

Table 3: Experimental parameters and their values.

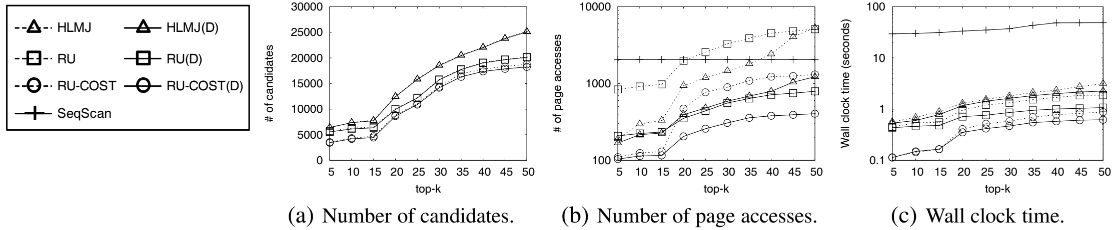
Parameter	Default	Range
k	25	$5 \sim 50$
Buffer size	5%	$1\% \sim 10\%$
$Len(Q)$	384	256, 384, 512
ω(window size)	64	32, 64, 128

We conduct all the experiments on a Linux server with Xeon Quad Core 1.6GHz CPU and 8 Gbytes RAM. We use LRU as the buffer replacement algorithm, and set the page size to 4 KBytes. For each query sequence, we set the warping width to 5% of the query length. We also performed experiments for varying both the number of PAA features and the warping width, but the performance trends are similar, and thus we omit the results for brevity.

6.2 Experiments and Results

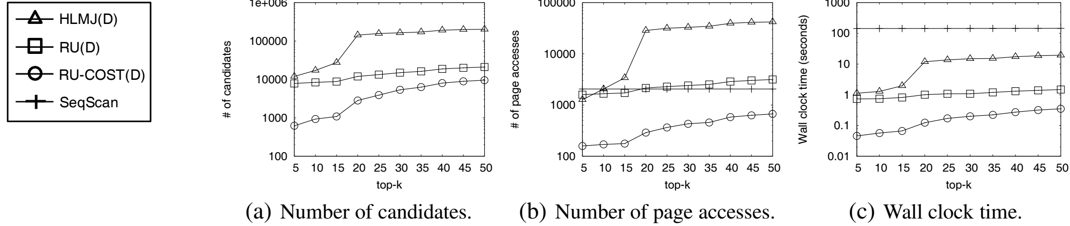
Experiment 1) (effect of k) Figure 11 shows the experimental results for UCR. We use the same query set consisting of 20 sequences that are used in [12]. We call this query set *UCR-REGULAR*. Note that PSM does not finish with reasonable times due to excessive bloom filter calls if $Len(Q) > 256$. Thus, we can show the performance results of PSM in Experiment 6 when the $Len(Q) = 256$. In addition, we can use the deferred retrieval mechanism [12] for HLMJ, RU, and RU-COST. This mechanism delays a set of subsequence retrieval requests, groups the requests by their corresponding subsequences, and enables batch retrieval, thereby avoiding excessive random disk I/Os. Thus, for each of these three algorithms, we experiment with two different versions: one uses the deferred retrieval mechanism (denoted as "(D)"); the other does not. In the deferred retrieval mechanism we allocate memory of only 0.5% of the database size for storing the delayed requests.

As shown in Figure 11(a), our RU-COST consistently reduces the number of candidates by up to 1.8 times compared with HLMJ. This indicates that 1) the MSEQ-distance is tighter than the lower



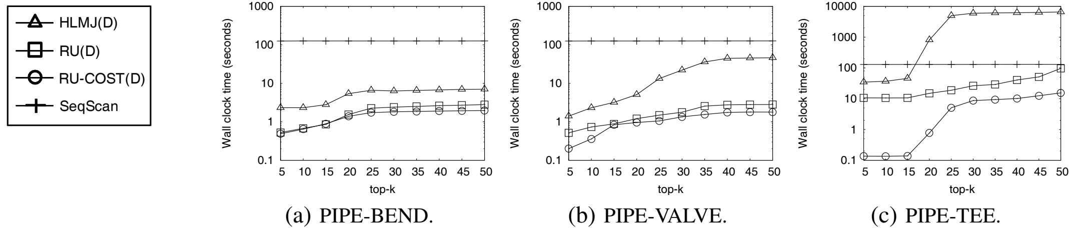
(a) Number of candidates. (b) Number of page accesses. (c) Wall clock time.

Figure 11: Experimental results with UCR-REGULAR by varying k.



(a) Number of candidates. (b) Number of page accesses. (c) Wall clock time.

Figure 12: Experimental results with UCR-DENSE by varying k.



(a) PIPE-BEND. (b) PIPE-VALVE. (c) PIPE-TEE.

Figure 13: Wall clock time for different query types on PIPE by varying k.

bound of HLMJ, and 2) cost-aware density scheduling is more efficient than MDMWP-scheduling. In particular, RU-COST outperforms RU by up to 1.6 times in the number of candidates. This means that RU-COST increases the lower bound more rapidly even for *UCR-REGULAR* having no very dense windows.

In terms of the number of page accesses, RU-COST(D) provides significant reduction compared with other algorithms as shown in Figure 11(b). This outcome is due to two causes: (1) RU-COST shows the best result in reducing the number of candidates; (2) the deferred retrieval mechanism groups many random accesses into a series of sequential accesses. The first reason explains why RU-COST(D) can reduce the number of page accesses by up to 3.0 and 2.0 times compared with HLMJ(D) and RU(D), respectively, in Figure 11(b). The second reason explains why RU-COST(D) outperforms RU-COST as well as RU and HLMJ.

As shown in 11(c), RU-COST(D) reduces the wall clock time by up to 3.8, 4.7, 260 times compared with RU(D), HLMJ(D), and SeqScan. In subsequence matching, the wall clock time is determined by two major factors: 1) the time for accessing disk pages and 2) the time for computing distances between query and candidate sequences. This explains why Figure 11(c) looks like a merged version of Figures 11(a) and 11(b). In Figures 11(a) and 11(b), RU-COST(D) shows the best result in terms of both candidates and page accesses, and therefore it consistently outperforms any other algorithms in the wall clock time of Figure 11(c).

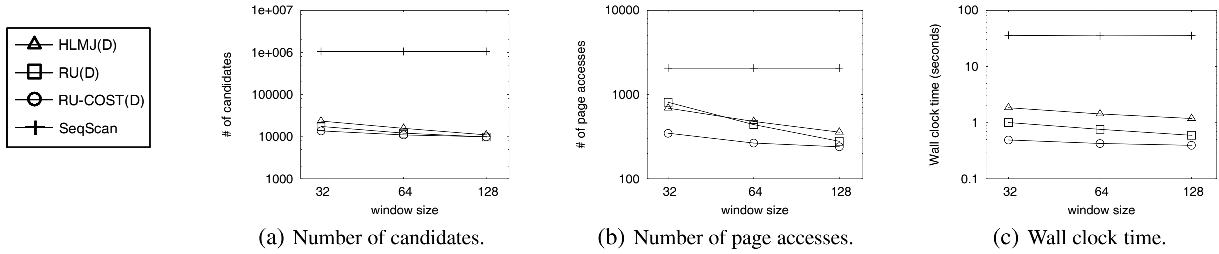
As in Figures 11(b) and 11(c), a deferred algorithm is always superior to its corresponding non-deferred algorithm. Thus, hereafter we consider only the deferred algorithms. For brevity, we omit the non-deferred algorithms from the experimental graphs.

Experiment 2) (effect of the data point density) In this experiment, we show that our ranked union approach clearly overcomes the MDMWP-based scheduling problem of HLMJ. As shown in Figure 2, this problem occurs by the density difference of data points. For this experiment, we construct another query set consisting of 20 query sequences from UCR, called *UCR-DENSE*, where each query sequence consists of two types of query points: one in the dense region, and the other in the sparse region. Thus, UCR-DENSE incurs the MDMWP-based scheduling problem.

Figure 12 shows the experimental results for UCR with UCR-DENSE. Comparing Figure 12(a) with Figure 11(a), the differences between algorithms are much larger in UCR-DENSE. In particular, the difference between RU-COST(D) and HLMJ(D) in UCR-DENSE is much larger than that in UCR-REGULAR. In Figure 12(a), RU-COST(D) reduces the number of candidates by up to 50.4 times compared with HLMJ(D). This significant reduction means that our ranked union approach resolves the MDMWP-based scheduling problem. Due to this reduction, the difference in the number of page accesses becomes large, and accordingly, the difference in the wall clock time also becomes large.

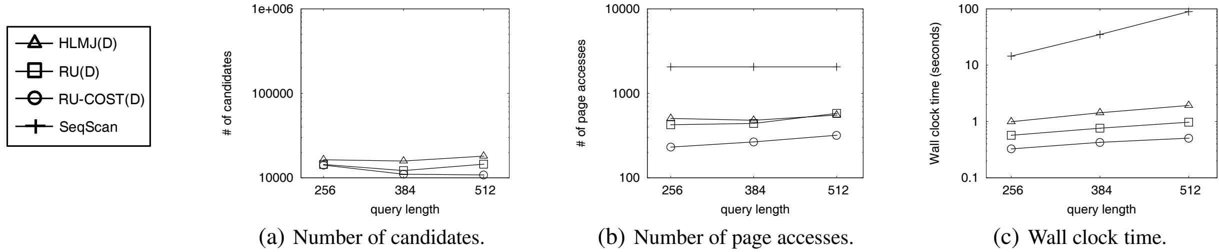
For a clearer comparison of RU-COST(D) and HLMJ(D), we use PIPE which contains lots of dense regions. Figure 13 shows the wall clock time for three representative queries sequences, *PIPE-BEND*, *PIPE-VALVE*, and *PIPE-TEE*. These query sequences are eventually mapped into dense and sparse regions in a mixed way and incur the MDMWP-based scheduling problem. Compared with HLMJ(D) and RU(D), RU-COST(D) significantly improves the performance by up to 980.9 and 78.3 times! In particular, the improvement is remarkable in PIPE-TEE and PIPE-VALVE (see Figures 13(b) and 13(c)). This is because a query sequence of PIPE-TEE or PIPE-VALUE contains many irregular patterns, i.e., its sliding windows are mapped into both sparse and dense regions, incurring the MDMWP-based scheduling problem.

Experiment 3) (effect of the buffer size) Figure 14 shows the wall clock time for UCR by varying the buffer size. As the query set, we use UCR-REGULAR in Figure 14(a) and UCR-DENSE in Fig-



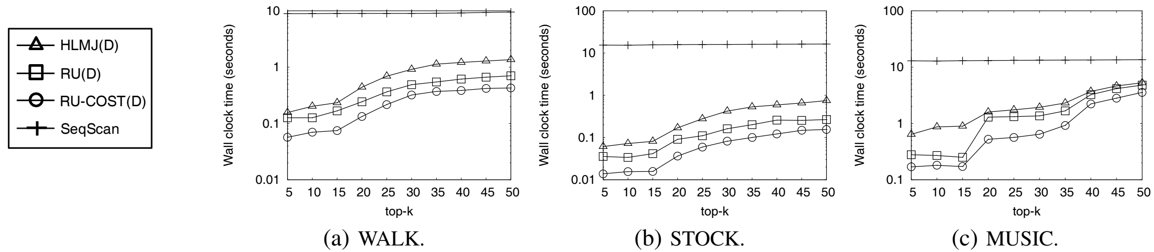
(a) Number of candidates. (b) Number of page accesses. (c) Wall clock time.

Figure 15: Experimental results for UCR with UCR-REGULAR by varying the window size.



(a) Number of candidates. (b) Number of page accesses. (c) Wall clock time.

Figure 16: Experimental results for UCR with UCR-REGULAR by varying the query length.



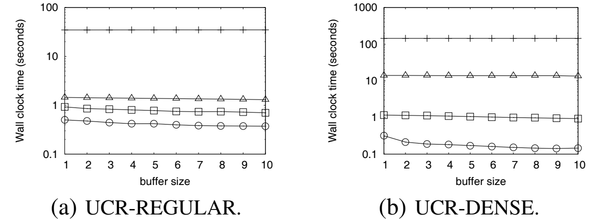
(a) WALK. (b) STOCK. (c) MUSIC.

Figure 17: Wall clock time for WALK, STOCK, and MUSIC by varying k.

ure 14(b). Like the previous experimental results, RU-COST(D) shows the best result by enlarging the lower bound aggressively.

As shown in Figure 14, SeqScan shows the constant time for all buffer sizes since it just accesses all subsequences with a small size buffer sequentially. In contrast, the wall clock time of other buffer-based algorithms slightly decreases as the buffer size increases. This is obvious since, as the buffer size increases, the buffer hit-ratio in accessing candidate subsequences increases. Note that there is a considerable performance degradation in the wall clock time for UCR-DENSE. This is because SeqScan exploits LB_{Keogh} before DTW computations, and LB_{Keogh} is much more effective in UCR-REGULAR than in UCR-DENSE. We also note that the buffer size incurs only a small performance difference in all of the buffer-based algorithms. This is because all these algorithms use the deferred retrieval mechanism which accumulates many random subsequence requests in the group subsequence access list being rescheduled to a sequential scan [12]. In general, the small buffer size is very crucial for both large (hundreds of gigabytes to terabytes) time-series data and multi-user environment. As we can see in Figure 14, our RU-COST(D) and RU(D) require only a small buffer size, and accordingly, we can say that our algorithms have the most desirable characteristic in the large database and multi-user environment.

Experiment 4) (Effect of Window Size and Query Length) Figure 15 shows the experimental results for UCR with UCR-REGULAR by varying the window size. The experimental results of Figure 15 are summarized as follows. Like the previous experiments, SeqScan shows constant values in all three measures, since it fully scans the entire database and considers all possible subsequences, regardless of the window size. On the other hand, the results of the other three algorithms are changed according to the window size, since they use window construction mechanism [7, 12, 16] of subsequence matching. We note that, as the window size increases, all three measures of these three algorithms decrease. This decreasing trend is well explained by the *window size effect* [17, 16]. That is, the larger window incurs the smaller number of candidates, and accordingly, the larger window causes the smaller number of page accesses and the shorter wall clock time. RU-COST(D) still outperforms the other three algorithms, regardless of the window size. Compared with SeqScan, HLMJ(D), and RU(D), the last RU-COST(D) improves matching performance by up to 90.6 times, 3.8 times, and 2.1 times for UCR-REGULAR; and 846.8 times, 80.2 times, and 11.6 times for UCR-DENSE (The figures of the results for UCR with UCR-DENSE are omitted due to space limit).



(a) UCR-REGULAR. (b) UCR-DENSE.

Figure 14: Wall clock time by varying the buffer size.

Figure 16 shows the experimental results by varying the query length. The experimental results of Figure 16 are summarized as follows. In the case of SeqScan, the number of candidates and the number of page accesses change only very slightly according to the query length. Its wall clock time, however, increases according to the query length, since the longer length requires more operations

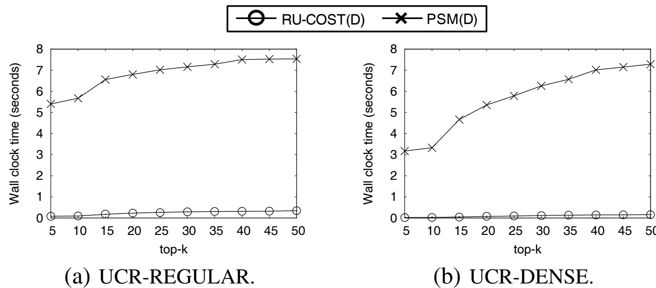
in computing the DTW distance. With other algorithms including RU-COST(D), the number of candidates slightly increases according to the query length. This increasing trend is also explained by the window size effect. That is, as the query length increases, the relative size of the corresponding window decreases, and thus, the more candidates occur due to the window size effect. Due to the increase in the number of candidates, the number of page accesses and the wall clock time also increase for the larger query length. Like the previous experimental results, RU-COST(D) significantly improves matching performance compared with other algorithms.

Experiment 5) (Effect of Different Data Sets) Figures 17(a), 17(b), and 17(c) show the wall clock time for WALK, STOCK, and MUSIC, respectively. (We omit the results for candidates and page accesses since their trends are very similar to that of UCR with UCR-REGULAR.) As shown in the figures, RU-COST(D) shows the best performance, and we can say that our ranked union approach always outperforms the previous algorithms, regardless of the data set. Because the results in the figures show such a similar tendency as that of UCR with UCR-REGULAR (compare Figure 17 with Figure 11(c)), we omit the detailed explanation of the performance results.

Experiment 6) (Comparison with PSM(D)) Figure 18 shows the wall clock times of PSM(D) and RU-COST(D) for UCR by varying k when the query size is 256. Figures 18(a) and 18(b) show performance results for UCR-REGULAR and UCR-DENSE, respectively. RU-COST(D) significantly outperforms PSM(D) by up to 62.5 and 135.7 times for UCR-REGULAR and UCR-DENSE, respectively. Since PSM exploits all windows using index-merge at the index level, there are no drastic performance differences for either query set. Note that we implemented the best version of PSM.

PSM maintains join signatures using the bloom filter [22], and computing join signatures requires a lot of bloom filter calls. Thus, it cannot support more than four-way joins with reasonable times due to excessive bloom filter calls. In our experiment, we observe that, on average, more than 4 billion bloom filter calls are performed to compute the join signatures.

We also performed experiments with k=25 and the query length=320. RU-COST(D) outperforms PSM(D) by up to 2440.5 times and 5352.1 times for UCR-REGULAR and UCR-DENSE, respectively.



(a) UCR-REGULAR. (b) UCR-DENSE.

Figure 18: Wall clock time for PSM(D) and RU-COST(D).

7. CONCLUSIONS

In this paper, we presented a novel framework, *ranked union*, for processing ranked subsequence matching in time-series databases. For this framework, we proposed a notion of the *matching subsequence equivalence class* (MSEQ), formally derived a novel lower bound *MSEQ-distance*, and showed its superior pruning power against the MDMWP-distance. After that, we proposed a novel execution model for ranked union along with two novel operators: 1) multi-way ranked union operator and 2) ranked subsequence matching subquery operator. We then enhanced the basic ranked union with a novel priority queue selection strategy RU-COST. By exploiting both a novel concept of the *cost-aware density* and the *selective expansion* technique with another lower bound, LB_{CDens}, for the cost-aware density, RU-COST can completely eliminate the MDMWP-scheduling problem with immaterial overhead. We also showed the lower boundness of LB_{CDens}. Through extensive experiments on both real and synthetic datasets, we showed that our ranked union solutions outperform the state-of-the-art algorithms by up to two to three orders of magnitude.

8. ACKNOWLEDGMENTS

This paper is based on research supported by the R&D program of MKE/KEIT (KI10033545). This paper is also supported in part by MEST/KOSEF (R11-2008-007-03003-0).

9. REFERENCES

[1] I. Assent, M. Wichterich, R. Krieger, H. Kremer, and T. Seidl. Anticipatory dtw for efficient similarity search in time series databases. *PVLDB*, 2(1):826–837, 2009.

[2] V. Athitsos, P. Papapetrou, M. Potamias, G. Kollios, and D. Gunopulos. Approximate embedding-based subsequence matching of time series. In *SIGMOD Conference*, pages 365–378, 2008.

[3] I. Bartolini and M. Patella. Warp: Accurate retrieval of shapes using phase of fourier descriptors and time warping distance. *IEEE PAMI.*, 27(1):142–147, 2005.

[4] D. J. Berndt and J. Clifford. Finding patterns in time series: a dynamic programming approach. pages 229–248, 1996.

[5] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE 2002*, 2002.

[6] R. Fagin, A. Lote, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[7] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *SIGMOD*, 1994.

[8] G. Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[9] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries in image databases. In *VLDB*, 2000.

[10] U. Güntzer, W.-T. Balke, and W. Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC*, pages 622–628, 2001.

[11] W. Han, J. Lee, Y. Moon, S. Hwang, and H. Yu. A new approach for processing ranked subsequence matching based on ranked union. http://www-db.knu.ac.kr/rankedunion/runion-tr.pdf, 2011.

[12] W. Han, J. Lee, Y. Moon, and H. Jiang. Ranked subsequence matching in time-series database. In *VLDB*, 2007.

[13] E. Keogh. Exact indexing of dynamic time warping. In *VLDB*, 2002.

[14] E. Keogh. A decade of progress in indexing and mining large time series databases. In *VLDB*, 2006.

[15] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. Ranksql: Query algebra and optimization for relational top-k queries. In *SIGMOD Conference*, pages 131–142, 2005.

[16] Y. Moon, K. Whang, and W. Han. General match: a subsequence matching method in time-series databases based on generalized windows. In *SIGMOD*, 2002.

[17] Y.-S. Moon, K.-Y. Whang, and W.-K. Loh. Duality-based subsequence matching in time-series databases. In *ICDE*, 2001.

[18] L. Rabiner and B.-H. Juang. *Fundamentals of speech recognition*. Prentice-Hall, Inc., 1993.

[19] D. Rafiei and A. O. Mendelzon. Querying time series data based on similarity. *TKDE*, 12(5):675–693, 2000.

[20] S. won Hwang and K. C.-C. Chang. Optimizing top-k queries for middleware access: A unified cost-based approach. *ACM Trans. Database Syst.*, 32(1):5, 2007.

[21] T. S. F. Wong and M. H. Wong. Efficient subsequence matching for sequences databases under time warping. *IDEAS*, 2003.

[22] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *SIGMOD*, 2007.

[23] B.-K. Yi and C. Faloutsos. Fast time sequence indexing for arbitrary lp norms. In *VLDB*, 2000.

[24] Y. Zhu and D. Shasha. Warping indexes with envelope transforms for query by humming. In *SIGMOD*, 2003.