# KV-match: A Subsequence Matching Approach Supporting Normalization and Time Warping

[Extended Version]

Jiaye Wu #, Peng Wang #, Ningting Pan #, Chen Wang *, Wei Wang #, Jianmin Wang *

# *School of Computer Science, Fudan University, Shanghai, China*
{wujy16, pengwang5, ntpan17, weiwang1}@fudan.edu.cn

* *School of Software, Tsinghua University, Beijing, China*
{wang_chen, jimwang}@tsinghua.edu.cn

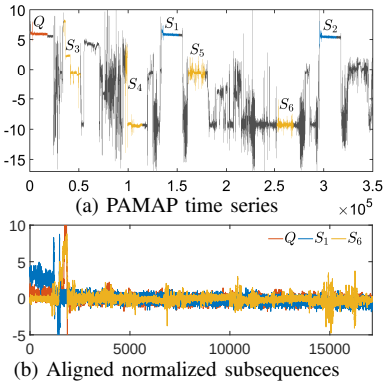
***Abstract*—The volume of time series data has exploded due to the popularity of new applications, such as data center management and IoT. Subsequence matching is a fundamental task in mining time series data. All index-based approaches only consider raw subsequence matching (RSM) and do not support subsequence normalization. UCR Suite can deal with normalized subsequence matching problem (NSM), but it needs to scan full time series. In this paper, we propose a novel problem, named constrained normalized subsequence matching problem (cNSM), which adds some constraints to NSM problem. The cNSM problem provides a knob to flexibly control the degree of offset shifting and amplitude scaling, which enables users to build the index to process the query. We propose a new index structure, KV-index, and the matching algorithm, KV-match. With a single index, our approach can support both RSM and cNSM problems under either ED or DTW distance. KV-index is a key-value structure, which can be easily implemented on local files or HBase tables. To support the query of arbitrary lengths, we extend KV-match to KV-match$_{DP}$, which utilizes multiple varied-length indexes to process the query. We conduct extensive experiments on synthetic and real-world datasets. The results verify the effectiveness and efficiency of our approach.**

## I. Introduction

Time series data are pervasive across almost all human endeavors, including medicine, finance and science. In consequence, there is an enormous interest in querying and mining time series data. [1], [2].

Subsequence matching problem is a core subroutine for many time series mining algorithms. Specifically, given a long time series $X$, for any query series $Q$ and a distance threshold $\varepsilon$, the subsequence matching problem finds all subsequences from $X$, whose distance with $Q$ falls within the threshold $\varepsilon$.

FRM [3] is the pioneer work of subsequence matching. Many approaches have been proposed, either to improve the efficiency [4], [5] or to deal with various distance functions [6], [7], such as Euclidean distance and Dynamic Time Warping. However, *all* these approaches only consider the raw subsequence matching problem (RSM for short). In recent years, researchers realize the importance of the subsequence normalization [8]. It is more meaningful to compare the z-normalized subsequences, instead of the raw ones. UCR Suite [8] is the state-of-the-art approach to solve the normalized subsequence matching problem (NSM for short).




(a) PAMAP time series

(b) Aligned normalized subsequences

(c) Information of Q

| Offset [Label] | Length |
|---|---|
| 877 [$Q$] | 17,124 |

(d) Results of NSM

| Offset [Label] | Distance |
|---|---|
| 252,492 [$S_6$] | 117.78 |
| 97,458 [$S_4$] | 130.80 |
| 34,562 [$S_3$] | 138.12 |
| 161,416 [$S_5$] | 149.37 |
| ... | ... |
| 134,456 [$S_1$] | 164.88 |
| 296,063 [$S_2$] | 166.74 |
| ... | ... |

$\star$ $\varepsilon = 200.0$

Fig. 1. Illustrative example of cNSM

The NSM approach suffers from two drawbacks. First, it needs to scan the full time series $X$, which is prohibitively expensive for long time series. For example, for a time series of length $10^9$, UCR Suite needs more than 100 seconds to process a query of length 1,000. [8] analyzed the reason why it is impossible to build the index for the NSM problem. Second, the NSM query may output some results not satisfying users' intent. The reason is that NSM fully ignores the offset shifting and amplitude scaling. However, in real world applications, the extent of offset shifting and amplitude scaling may represent certain specific physical mechanism or state. Users often only hope to find subsequences within similar state as the query. We illustrate it with an example.

**Example 1.** The time series in Fig. 1(a) comes from the Physical Activity Monitoring for Aging People (PAMAP) dataset [1] collected from z-accelerometer at hand position. The monitored person conducts various activities alternatively, like sitting, standing, running and so on. Each activity lasts for about 3 minutes, and the data collection frequency is 100Hz. We use one subsequence corresponding to lying activity as the query ($Q$ in Fig. 1(c)) to find other "lying" subsequences. We issue a NSM query with $Q$, and Fig. 1(d) lists the top results. Unfortunately, all top-4 results corresponds to other activities. $S_3$ and $S_5$ correspond to sitting activity, while $S_4$ and $S_6$ correspond to breaking activity. Although $S_1$ and $S_2$

are the desired results (correspond to lying activity), they are ranked out of top-20. We show the normalized $Q$, $S_1$ and $S_6$ in Fig. 1(b). It is difficult to distinguish them after normalization.

By observing Fig. 1(a), one can filter the undesired results easily by adding an additional constraint: the output subsequences should have similar mean value as $Q$. In fact, this new type of NSM query, *NSM plus some constraints*, is useful in many applications. We list two of them as follows,

- (Industry application) In the wind power generation field, LIDAR system can provide preview information of wind disturbances [9]. Extreme Operating Gust (EOG) is a typical gust pattern which is a phenomenon of dramatic changes of wind speed in a short period. Fig. 2 shows a typical EOG pattern. This pattern is important because it may generate damage on the turbine. All EOG pattern occurrences have the similar shape, and their fluctuation degree falls within certain range, because the wind speed cannot be arbitrarily high. If we hope to find all EOG pattern occurrences in the historical data, we can use a typical EOG pattern as the query, plus the constraint on the range of the values.
- (IoT application) When a container truck goes through a bridge, the strain meter planted in the bridge will demonstrate a specific fluctuation pattern. The value range in the pattern depends on the weight of the truck. If we have one occurrence of the pattern as a query, we can additionally set a mean value range as the constraint to search container trucks whose weight falls within a certain range.

Note that the above applications cannot be handled by RSM query, because the existing offset shifting and amplitude scaling forces us to set a very large distance threshold, which will cause many false positive results.

Furthermore, to verify the universality of this new query type, we investigate the motif pairs in some popular real-world time series benchmarks. Motif mining [2] is an important time series mining task, which finds a pair (or set) of subsequences with minimal normalized distance. For a motif subsequence pair, say $X$ and $Y$, we show the relative mean value difference ($\Delta$Mean= $\frac{|\mu^X - \mu^Y|}{\max - \min}$) and the ratio of standard deviation ($\Delta$Std= $|\frac{\sigma^X}{\sigma^Y}|$) in Fig. 3 We can see that although these pairs are found without any constraint (like NSM query), both mean value and standard deviation of motif subsequences are very similar. So we can find these pairs by the cNSM query, a NSM query plus a small constraint.

In this paper, we formally define a new subsequence matching problem, called *constrained normalized subsequence matching problem* (cNSM for short). Two constraints, one for mean value and the other for standard deviation, are added to the traditional NSM problem. One exemplar cNSM query looks like "given a query $Q$ with mean value $\mu^Q$ and standard deviation $\sigma^Q$, return subsequences $S$ which satisfy: (1) $Dist(\hat{S}, \hat{Q}) \leq 1.5$; (2) $|\mu^Q - \mu^S| \leq 5$; (3) $0.5 \leq \sigma^Q/\sigma^S \leq 2$". With the constraint, the cNSM problem provides a knob to flexibly control the degree of offset shifting (represented by mean value) and amplitude scaling (represented by standard deviation). Moreover, the cNSM problem offers us the opportunity to build index for the normalized subsequence matching.

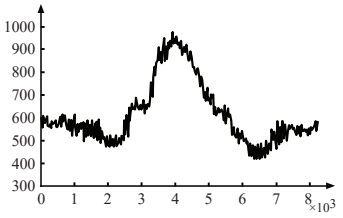



Fig. 2. EOG pattern

| Dataset | △Mean | △Std |
|---|---|---|
| Taxi | 0.01 | 1.01 |
| Power | 0.03 | 1.02 |
| Temperature | 0.04 | 1.11 |
| Penguin | 0.06 | 1.61 |
| Commute | 0.00 | 1.02 |
| ECG 308 | 0.00 | 1.01 |
| ECG 15 | 0.15 | 1.05 |
| NPRS 43 | 0.01 | 1.01 |
| Video | 0.01 | 1.03 |
| TEK 17 | 0.00 | 1.00 |

Fig. 3. Motif example

**Challenges.** Solving the cNSM problem faces the following challenges. First, how can we process the cNSM query efficiently? A straightforward approach is to first apply UCR Suite to find unconstrained results, and then use mean value and standard deviation constraints to prune the unqualified ones. However, it still needs to scan the full series. Can we build an index and process the query more efficiently?

Second, users often conduct the similar subsequence search in an exploratory and interactive fashion. Users may try different distance functions, like Euclidean distance or Dynamic Time Warping. Meanwhile, users may try RSM and cNSM query simultaneously. Can we build a single index to support all these query types?

**Contributions.** Besides proposing the cNSM problem, we also have the following contributions.

- We present the filtering conditions for four query types, RSM-ED, RSM-DTW, cNSM-ED and cNSM-DTW, and prove the correctness. The conditions enable us to build index and meanwhile guarantee no false dismissals.
- We propose a new index structure, KV-index, and the query processing approach, KV-match, to support all these query types. The biggest advantage is that we can process various types of queries efficiently with a single index. Moreover, KV-match only needs a few numbers of *sequential scans* of the index, instead of many random accesses of tree nodes in the traditional R-tree index, which makes it much more efficient.
- Third, to support the query of arbitrary lengths efficiently, we extend KV-match to KV-match$_{DP}$, which utilizes multiple indexes with different window lengths. We conduct extensive experiments. The results verify the efficiency and effectiveness of our approach.

The rest of the paper is organized as follows. We present the preliminary knowledge and problem statements in Section II. In Section III we introduce the theoretical foundation and motivate the approach. Section IV and V describe our index structure, index building algorithm and query processing algorithm. Section VI extends our method to use multi-level indexes with different window lengths. Our implementation

TABLE I
FREQUENTLY USED NOTATIONS

| Notation | Description |
|---|---|
| $X$ | a time series $(x_1, x_2, \cdots, x_n)$ |
| $X(i,l)$ | a length-$l$ subsequence of $X$ starting at offset $i$ |
| $\hat{X}$ | the normalized series of time series $X$ |
| $X_i$ | the $i$-th length-$w$ disjoint window of $X$ |
| $\mu_i^X$ | the mean value of the $i$-th disjoint window of $X$ |
| $\sigma_i^X$ | the standard deviation of the $i$-th disjoint window of $X$ |
| $WI$ | a window interval containing continuous window positions |
| $IS_i$ | a set of window intervals satisfying the criterion for $Q_i$ |
| $CS_i, CS$ | a set of candidates for $Q_i$ and for all $Q_j (1 \le j \le i)$ |
| $n_I, n_P$ | the number of window intervals and window positions |

details are described in Section VII. The experimental results are presented in Section VIII and we discuss related works in Section IX. Finally, we conclude the paper and look into the future work in Section X.

## II. Preliminary Knowledge

In this section, we introduce the definition of time series and other useful notations.

### A. Definitions and Problem Statement

A *time series* is a sequence of ordered values, denoted as $X = (x_1, x_2, \cdots, x_n)$, where $n = |X|$ is the *length* of $X$. A length-$l$ *subsequence* of $X$ is a shorter time series, denoted as $X(i,l) = (x_i, x_{i+1}, \cdots, x_{i+l-1})$, where $1 \le i \le n-l+1$.

For any subsequence $S = (s_1, s_2, \cdots, s_m)$, $\mu^S$ and $\sigma^S$ are the *mean value* and *standard deviation* of $S$ respectively. Thus the *normalized series* of $S$, denoted as $\hat{S}$, is

$$\hat{S} = \left( \frac{s_1 - \mu^S}{\sigma^S}, \frac{s_2 - \mu^S}{\sigma^S}, \cdots, \frac{s_m - \mu^S}{\sigma^S} \right)$$

Our work supports two common distance measures, *Euclidean distance* and *Dynamic Time Warping*. Here we give the definition of them.

*Euclidean Distance* (**ED**): Given two length-$m$ sequences, $S$ and $S'$, their distance is $ED(S, S') = \sqrt{\sum_{i=1}^{m} (s_i - s'_i)^2}$.

*Dynamic Time Warping* (**DTW**): Given two length-$m$ sequences, $S$ and $S'$, their distance is

$$DTW(\langle\rangle, \langle\rangle) = 0; \qquad DTW(S, \langle\rangle) = DTW(\langle\rangle, S') = \infty$$

$$DTW(S, S') = \sqrt{(s_1 - s'_1)^2 + \min \begin{cases} DTW(suf(S), suf(S')) \\ DTW(S, suf(S')) \\ DTW(suf(S), S') \end{cases}}$$

where $\langle\rangle$ represents empty series and $suf(S) = (s_2, \cdots, s_m)$ is a suffix subsequence of $S$.

In DTW, the *warping path* is defined as a matrix to represent the optimal alignment for two series. The matrix element $(i, j)$ represents that $s_i$ is aligned to $s'_j$. To reduce the computation complexity, we use the Sakoe-Chiba band [10] to restrict the width of warping, denoted as $\rho$. Any pair $(i, j)$ should satisfy $|i - j| \le \rho$. When $\rho = 0$, it degenerates into ED.

We aim to support subsequence matching for both the raw subsequence and the normalized subsequence simultaneously. The problem statements are given here.

*Raw Subsequence Matching* (**RSM**): Given a long time series $X$, a query sequence $Q$ ($|X| \ge |Q|$) and a distance threshold $\varepsilon$ ($\varepsilon \ge 0$), find all subsequences $S$ of length $|Q|$ from $X$, which satisfy $D(S, Q) \le \varepsilon$. In this case, we call that $S$ and $Q$ are *in $\varepsilon$-match*.

*Normalized Subsequence Matching* (**NSM**): Given a long time series $X$, a query sequence $Q$ and a distance threshold $\varepsilon$ ($\varepsilon \ge 0$), find all subsequences $S$ of length $|Q|$ from $X$, which satisfy $D(\hat{S}, \hat{Q}) \le \varepsilon$, where $\hat{S}$ and $\hat{Q}$ are the normalized series of $S$ and $Q$ respectively.

The cNSM problem adds two constraints to the NSM problem. Thresholds $\alpha$ ($\alpha \ge 1$) and $\beta$ ($\beta \ge 0$) are introduced to constrain the degree of amplitude scaling and offset shifting.

*Constrained Normalized Subsequence Matching* (**cNSM**): Given a long time series $X$, a query sequence $Q$, a distance threshold $\varepsilon$, and the constraint thresholds $\alpha$ and $\beta$, find all subsequences $S$ of length $|Q|$ from $X$, which satisfy

$$D(\hat{S}, \hat{Q}) \le \varepsilon \ , \ \frac{1}{\alpha} \le \frac{\sigma^S}{\sigma^Q} \le \alpha \ , \ -\beta \le \mu^S - \mu^Q \le \beta$$

The larger $\alpha$ and $\beta$, the looser the constraint. In this case, we call that $S$ and $Q$ are *in $(\varepsilon, \alpha, \beta)$-match*.

The distance $D(\cdot, \cdot)$ is either ED or DTW. In this paper, we build an index to support four types of queries, RSM-ED, RSM-DTW, cNSM-ED and cNSM-DTW simultaneously.

## III. Theoretical Foundation and Approach Motivation

In this section, we establish the theoretical foundation of our approach. We propose a condition to filter the unqualified subsequences. For all four types of queries, the conditions share the same format, which enables us to support all query types with a single index.

Specifically, for the query $Q$ and the subsequence $S$ of length-$m$, we segment them into aligned disjoint windows of the same length $w$. The $i$-th window of $Q$ (or $S$) is denoted as $Q_i$ (or $S_i$), ($1 \le i \le p = \lfloor \frac{m}{w} \rfloor$), that is, $Q_i = (q_{(i-1)*w+1}, \cdots, q_{i*w})$.

For each window, we hope to find one or more features, based on which we can construct the filtering condition. In this work, we choose to utilize one single feature, the mean value of the window. The advantages are two-folds. First, with a single feature, we can build a one-dimensional index, which improves the efficiency of index retrieval greatly. Second, the mean value allows us to design the condition for both RSM and cNSM query.

We denote mean values of $Q_i$ and $S_i$ as $\mu_i^Q$ and $\mu_i^S$. The condition consists of $p$ number of ranges. The $i^{th}$ one is denoted as $[LR_i, UR_i]$ ($1 \le i \le p$). If $S$ is a qualified subsequence, for any $i$, $\mu_i^S$ must fall within $[LR_i, UR_i]$. If any $\mu_i^S$ is outside the range, we can filter $S$ safely.

### A. RSM-ED Query Processing

In this section, we first present the condition for the simplest case, RSM-ED query, and then illustrate our approach.

**Lemma 1.** *If $S$ and $Q$ are in $\varepsilon$-match under ED measure, that is, $ED(S, Q) \le \varepsilon$, then $\mu_i^S$ ($1 \le i \le p$) must satisfy*

$$\mu_i^S \in \left[ \mu_i^Q - \frac{\varepsilon}{\sqrt{w}}, \mu_i^Q + \frac{\varepsilon}{\sqrt{w}} \right] \tag{1}$$

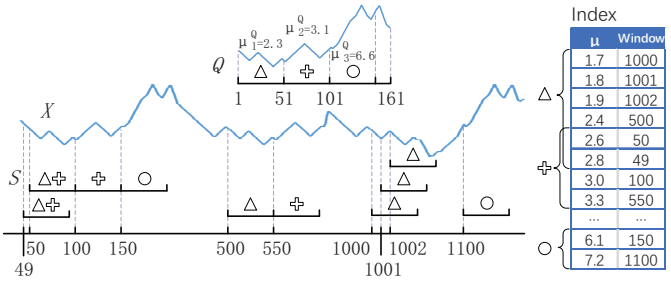



Fig. 4. Illustrative example

*Proof.* Based on the ED definition, we have

$$ED^2(S,Q) = \sum_{k=1}^{n}(s_k - q_k)^2 \geq \sum_{j=(i-1)*w+1}^{i*w}(s_j - q_j)^2$$

where $1 \leq i \leq p$. According to the corollary in [11],

$$\sum_{j=(i-1)*w+1}^{i*w}(s_j - q_j)^2 \geq w * \left(\mu_i^S - \mu_i^Q\right)^2$$

If $D(S,Q) \leq \varepsilon$, after inequality transformation, it should hold that $\left(\mu_i^S - \mu_i^Q\right)^2 \leq \frac{\varepsilon^2}{w}$, so we get Eq. (1). □

Now we illustrate our approach with the example in Fig. 4. $X$ is a long time series, and $Q$ is the query sequence of length 161. The goal is to find all length-161 subsequences $S$ from $X$, which satisfy $ED(S,Q) \leq \varepsilon$. The parameter of the window length $w$ is set to 50. We split $Q$ into three disjoint windows of length 50, $Q_1$, $Q_2$, $Q_3$[1]. According to Lemma 1, for any qualified subsequence $S$, the mean value of the $i^{th}$ disjoint window $S_i$ must fall within the range $\left[\mu_i^Q - \frac{\varepsilon}{\sqrt{50}}, \mu_i^Q + \frac{\varepsilon}{\sqrt{50}}\right]$ $(i = 1, 2, 3)$. To facilitate finding the windows satisfying this condition, we build the index as follows. We compute the mean values of all sliding windows $X(j,w)$, denoted as $\mu(X(j,w))$, and build a sorted list of $\langle \mu(X(j,w)), j\rangle$ entries. With this structure, we find the candidates in two steps. First, for each window $Q_i$, we obtain all sliding windows whose mean values fall within $\left[\mu_i^Q - \frac{\varepsilon}{\sqrt{50}}, \mu_i^Q + \frac{\varepsilon}{\sqrt{50}}\right]$ by a single *sequential scan* operation. We denote the found windows for $Q_i$ as $CS_i$. Then, we generate the final candidates by *intersecting* windows in $CS_1$, $CS_2$ and $CS_3$.

In Fig. 4, sliding windows in $CS_1$, $CS_2$ and $CS_3$ are marked with "triangle", "cross" and "circle" respectively. The only candidate is $X(50,161)$, because $X(50,50) \in CS_1$, $X(100,50) \in CS_2$ and $X(150,50) \in CS_3$.

### B. Range for cNSM-ED Query

We solve the cNSM problem based on KV-index either. For the given query $Q$, we determine whether a subsequence $S$ is $(\varepsilon,\alpha,\beta)$-match with $Q$ by checking the raw subsequence $S$ directly. Specifically, we achieve this goal by designing the range $[LR_i, UR_i]$ for each query window $Q_i$. For any subsequence $S$, if any $\mu_i^S$ falls outside this range, $S$ cannot be $(\varepsilon,\alpha,\beta)$-match with $S$ and we can filter $S$ safely. We illustrate it with an example. Let $Q = (1,1,-1,-1)$, $w = 2$, $(\alpha,\beta) = (2,1)$ and $\varepsilon = 0$ [2]. By simple calculation, we obtain $\mu_1^Q = 1$ and $\sigma^Q = 1.1547$. For any length-4 subsequence $S$, if only $\mu_1^S = 4$, we can infer that $S$ cannot be matched with $Q$ without checking the whether $\hat{S}$ satisfies the cNSM condition, as follows. To make $ED(\hat{Q},\hat{S}) = 0$, $\mu_2^S$ must be -4. If it is the case, $\sigma^S$ is 4.6188 at least. However, $\frac{\sigma^S}{\sigma^Q} > 2$, which violates the cNSM condition.

Now we formally give the range for cNSM-ED query. Let $\mu^S$ and $\mu^Q$ be the global mean values of $S$ and $Q$, $\sigma^S$ and $\sigma^Q$ be the standard deviations, $\hat{S}$ and $\hat{Q}$ be the normalized $S$ and $Q$ respectively.

**Lemma 2.** *If $S$ and $Q$ are in $(\varepsilon,\alpha,\beta)$-match under ED measure, that is, $ED(\hat{S},\hat{Q}) \leq \varepsilon$, then $\mu_i^S$ $(1 \leq i \leq p)$ satisfies*

$$\mu_i^S \in [v_{\min} + \mu^Q - \beta, v_{\max} + \mu^Q + \beta] \quad (2)$$

*where*

$$v_{\min} = \min\left(\alpha \cdot (\mu_i^Q - \mu^Q - \tfrac{\varepsilon\sigma^Q}{\sqrt{w}}), \tfrac{1}{\alpha} \cdot (\mu_i^Q - \mu^Q - \tfrac{\varepsilon\sigma^Q}{\sqrt{w}})\right),$$
$$v_{\max} = \max\left(\alpha \cdot (\mu_i^Q - \mu^Q + \tfrac{\varepsilon\sigma^Q}{\sqrt{w}}), \tfrac{1}{\alpha} \cdot (\mu_i^Q - \mu^Q + \tfrac{\varepsilon\sigma^Q}{\sqrt{w}})\right).$$

*Proof.* Based on the normalized ED definition, we have

$$ED(\hat{S},\hat{Q}) = \sqrt{\sum_{j=1}^{m}\left(\frac{s_j - \mu^S}{\sigma^S} - \frac{q_j - \mu^Q}{\sigma^Q}\right)^2}$$

Let $a = \frac{\sigma^S}{\sigma^Q}$ and $b = \mu^S - \mu^Q$, where $a \in [\frac{1}{\alpha}, \alpha]$ and $b \in [-\beta, \beta]$. If $ED(\hat{S},\hat{Q}) \leq \varepsilon$, it holds that

$$\sum_{j=1}^{m}\left(\frac{s_j - \mu^Q - b}{a\sigma^Q} - \frac{q_j - \mu^Q}{\sigma^Q}\right)^2 \leq \varepsilon^2$$

According to the corollary in [11], similar to Lemma 1, for the $i$-th window $S_i$ and $Q_i$, we have

$$\left(\frac{\mu_i^S - \mu^Q - b}{a\sigma^Q} - \frac{\mu_i^Q - \mu^Q}{\sigma^Q}\right)^2 \leq \frac{\varepsilon^2}{w}$$

By simple transformation, for any specific pair of $(a,b)$, we can get a range of $\mu_i^S$ as follows,

$$\mu_i^S \in \left[\left(\mu_i^Q - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q, \left(\mu_i^Q - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q\right]$$

For ease of description, we assign $\mu_i^Q - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}} = A$ and $\mu_i^Q - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}} = B$.

The final range $[LR_i, UR_i]$ should be

$$\left[\min_{\substack{a\in[\frac{1}{\alpha},\alpha]\\ b\in[-\beta,\beta]}}\{Aa + b + \mu^Q\}, \max_{\substack{a\in[\frac{1}{\alpha},\alpha]\\ b\in[-\beta,\beta]}}\{Ba + b + \mu^Q\}\right]$$

As illustrated in Fig. 5, the rectangle represents the whole legal range of $a$ and $b$. Let $f(a,b) = Aa + b + \mu^Q$ and $g(a,b) = Ba + b + \mu^Q$. Apparently, both $f(a,b)$ and $g(a,b)$ increase monotonically for $b \in [-\beta,\beta]$. As for $a$, we have two cases,

- If $A \geq 0$, $f(a,b)$ increases monotonically for $a \in [\frac{1}{\alpha}, \alpha]$. $f(a,b)$ is minimal when $a = \frac{1}{\alpha}$ and $b = -\beta$, which is represented by the point $p_3$ in Fig. 5;

[1] We can ignore the remain part $Q(151,11)$ without sacrificing the correctness since Lemma 1 is a *necessary condition* for RSM.

[2] To make the example simple enough, we set $\varepsilon$ as 0.

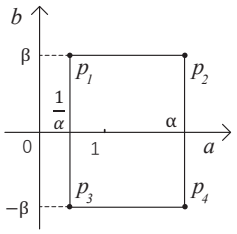



Fig. 5. Legal Range of $(a, b)$

| Key | Value |
|---|---|
| [1. 5, 2. 0) | [1000, 1002] |
| [2. 0, 3. 0) | [49, 50], [500, 500] |
| [3. 0, 4. 0) | [100, 100], [550, 550] |
| ..... | ..... |
| [6. 0, 7. 5) | [150, 150], [1100, 1100] |

Fig. 6. Index Structure

- If $A < 0$, $f(a, b)$ decreases monotonically for $a \in [\frac{1}{\alpha}, \alpha]$. $f(a, b)$ is minimal when $a = \alpha$ and $b = -\beta$, which is represented by the point $p_4$ in Fig. 5.

So $$LR_i = \min_{a\in[\frac{1}{\alpha},\alpha], b\in[-\beta,\beta]} f(a, b) = \min_{a\in\left\{\frac{1}{\alpha},\alpha\right\}} f(a, -\beta)$$

Note that formula $a \in \left\{\frac{1}{\alpha}, \alpha\right\}$ means $a$ is either $\frac{1}{\alpha}$ or $\alpha$.

Similarly, we can infer the maximal value of $g(a, b)$ as following two cases,

- If $B \geq 0$, $g(a, b)$ is maximal when $a = \alpha$ and $b = \beta$, which is represented by the point $p_2$ in Fig. 5.
- If $B < 0$, $g(a, b)$ is maximal when $a = \frac{1}{\alpha}$ and $b = \beta$, which is represented by the point $p_1$ in Fig. 5.

So $$UR_i = \max_{a\in[\frac{1}{\alpha},\alpha], b\in[-\beta,\beta]} g(a, b) = \max_{a\in\left\{\frac{1}{\alpha},\alpha\right\}} g(a, \beta)$$

□

### C. Range for RSM-DTW and cNSM-DTW Query

Before introducing the ranges, we first review the query envelop and the lower bound of DTW distance, LB_PAA [12]. To deal with DTW$_\rho$ measure, given length-$m$ query $Q$, the query envelop consists of two length-$m$ series, $L$ and $U$, as the lower and upper envelop respectively. The $i$-th elements of $L$ and $U$, denoted as $l_i$ and $u_i$, are defined as

$$l_i = \min_{-\rho\leq r\leq\rho} q_{i+r}\ ,\ u_i = \max_{-\rho\leq r\leq\rho} q_{i+r}.$$

LB_PAA is defined based on the query envelop. $L$ and $U$ are split into $p$ number of length-$w$ disjoint windows, $(L_1, L_2, \cdots, L_p)$ and $(U_1, U_2, \cdots, U_p)$, in which $L_i = (l_{(i-1)\cdot w+1}, \cdots, l_{i\cdot w})$ and $U_i = (u_{(i-1)\cdot w+1}, \cdots, u_{i\cdot w})$ $(1 \leq i \leq p = \lfloor\frac{m}{w}\rfloor)$. The mean values of $L_i$ and $U_i$ are denoted as $\mu_i^L$ and $\mu_i^U$ respectively. For any length-$m$ subsequence $S$, the LB_PAA is as follows,

$$LB\_PAA(S, Q) = \sqrt{\sum_{i=1}^{p} w \cdot \begin{cases} (\mu_i^S - \mu_i^U)^2 & \text{if } \mu_i^S > \mu_i^U \\ (\mu_i^S - \mu_i^L)^2 & \text{if } \mu_i^S < \mu_i^L \\ 0 & \text{Otherwise} \end{cases}} \quad (3)$$

which satisfies $LB\_PAA(S, Q) \leq DTW_\rho(S, Q)$ [12].

Now we give the ranges for RSM and cNSM under the DTW$_\rho$ measure in turn.

**Lemma 3.** *If $S$ and $Q$ are in $\varepsilon$-match under DTW$_\rho$ measure, that is, $DTW_\rho(S, Q) \leq \varepsilon$, then $\mu_i^S$ $(1 \leq i \leq p)$ satisfies*

$$\mu_i^S \in \left[\mu_i^L - \frac{\varepsilon}{\sqrt{w}}, \mu_i^U + \frac{\varepsilon}{\sqrt{w}}\right] \quad (4)$$

*Proof.* See Appendix A.

**Lemma 4.** *If $S$ and $Q$ are in $(\varepsilon, \alpha, \beta)$-match under DTW$_\rho$ measure, that is, $DTW_\rho(\hat{S}, \hat{Q}) \leq \varepsilon$, then $\mu_i^S$ $(1 \leq i \leq p)$ satisfies*

$$\mu_i^S \in \left[v_{\min} + \mu^Q - \beta, v_{\max} + \mu^Q + \beta\right] \quad (5)$$

*where*

$$v_{\min} = \min\left(\alpha \cdot (\mu_i^L - \mu^Q - \tfrac{\varepsilon\sigma^Q}{\sqrt{w}}), \tfrac{1}{\alpha} \cdot (\mu_i^L - \mu^Q - \tfrac{\varepsilon\sigma^Q}{\sqrt{w}})\right),$$
$$v_{\max} = \max\left(\alpha \cdot (\mu_i^U - \mu^Q + \tfrac{\varepsilon\sigma^Q}{\sqrt{w}}), \tfrac{1}{\alpha} \cdot (\mu_i^U - \mu^Q + \tfrac{\varepsilon\sigma^Q}{\sqrt{w}})\right).$$

*Proof.* See Appendix B.

**Analysis**. We provide the ranges of mean value for all four query types, which means that we can support all queries with a single index. When processing different query types, the only difference is to use different ranges of $\mu_i^S$. This property is beneficial for exploratory search tasks.

## IV. KV-INDEX

In this section, we present our index structure KV-index, and the index building algorithm.

### A. Index Structure

The index structure in Fig. 4 has approximately equal number of entries of $|X|$, which causes a huge space cost. To avoid that, we propose a more compact index structure which utilizes the data locality property, that is, the values of adjacent time points may be close. In consequence, the mean values of adjacent sliding windows will be similar too.

Logically, KV-index consists of ordered rows of key-value pairs. The key of the $i$-th row, denoted as $K_i$, is a *range* of mean values of sliding windows, that is, $K_i = [low_i, up_i)$, where $low_i$ and $up_i$ are the left and right endpoint of the mean value range of $K_i$ respectively. It is a left-closed-right-open range, and the ranges of adjacent rows are disjoint.

The corresponding value, denoted as $V_i$, is the set of sliding windows whose mean values fall within $K_i$. To facilitate the expression, we represent each window by its position, that is, we represent sliding window $X(j, w)$ with $j$. To further save the space cost and also facilitate subsequence matching algorithm, we organize the window positions in $V_i$ as follows. The positions in $V_i$ are sorted in ascending order, and consecutive ones are merged into a *window interval*, denoted as *WI*. So $V_i$ consists of one or more sorted and non-overlapped window intervals.

**Definition 1** (Window Interval)**.** *We combine the $l^{th}$ to $r^{th}$ length-$w$ sliding windows of $X$ as a window interval $WI = [l, r]$, which contains a set of sliding windows $\{X(l, w), X(l+1, w), \cdots, X(r, w)\}$, where $1 \leq l \leq r \leq |X| - w + 1$.*

In the following descriptions, we use $j \in WI$ to denote the window position $j$ belonging to the window interval $WI = [l, r]$, that is, $j \in [l, r]$. Moreover, we use $WI.l$, $WI.r$ and $|WI| = r - l + 1$ to denote the left boundary, the right boundary and the size of interval *WI* respectively. The overall number of window intervals in $V_i$ is denoted as $n_I(V_i)$, and the number of window positions in $V_i$ as $n_P(V_i)$. Formally, we have

$$n_I(V_i) = \left|\{WI \mid WI \in V_i\}\right| \quad (6)$$

$$n_P(V_i) = \sum_{WI\in V_i} \left|WI\right| \quad (7)$$

Fig. 6 shows KV-index for Fig. 4. The first row indicates that there exists three sliding windows, $X(1000, 50)$, $X(1001, 50)$ and $X(1002, 50)$, whose mean values fall within the range $[1.5, 2.0)$. In the second row, three windows are organized into two intervals $[49, 50]$ and $[500, 500]$. Thus $n_I(V_2) = 2$ and $n_P(V_2) = 3$. Note that, $[500, 500]$ is a special interval which only contains one single window position.

To facilitate the query processing, KV-index also contains a meta table, in which each entry is a quadruple as $\langle K_i, pos_i, n_I(V_i), n_P(V_i)\rangle$, where $pos_i$ is the offset of $i$-th row in the index file. Due to its small size, we can load the meta table to memory before processing the query. With the meta table, we can quickly determine the offset and the length of a scan operation by the simple binary search.

Physically, KV-index can be implemented as a local file, an HDFS file or an HBase table, because of its simple format. In this work, we implement two versions, a local file version and an HBase table version (details are in Section VIII). In general, if a file system or a database supports the "scan" operation with start-key and end-key parameters, it can support KV-index. We provide details about the index implementation in Section VII.

### B. Index Building Algorithm

We build the index with two steps. First, we build an index in which all rows use the equal-width range of the mean values. Second, because data distribution is not balanced among rows, we merge adjacent rows to optimize the index. We first introduce a basic in-memory algorithm, which works for moderate data size. Then we discuss how to extend it to very large data scale.

In the first step, we pre-define a parameter $d$, which represents the range width of the mean values. The range of each row will be $[k \cdot d, (k+1) \cdot d)$, where $k \in \mathbb{Z}$. We read series $X$ sequentially. A circular array is used to maintain the length-$w$ sliding window $X(i, w)$, and its mean value $\mu_i^X$ are computed on the fly. Assume the mean value of $S_{i-1}$, $\mu_{i-1}^X$, is in range $K_j$, and the mean value of the current window $S_i$, $\mu_i^X$, is also in $K_j$, we modify the current *WI* by changing its right boundary from $i-1$ to $i$. Otherwise, a new interval, $WI = [i, i]$, will be added into certain row according to $\mu_i^X$.

The equal-width range can cause the zigzag style of adjacent rows. For example, the $V_i = \{[5, 5], [7, 7]\}$ and $V_{i+1} = \{[6, 6], [8, 8]\}$. Apparently, a better way is to merge these two rows so that the corresponding value becomes $V_i = [5, 8]$.

In the second step, we merge adjacent rows with a greedy algorithm. We check the rows beginning from $\langle K_1, V_1\rangle$ and $\langle K_2, V_2\rangle$. Let the current rows be $\langle K_i, V_i\rangle$ and $\langle K_{i+1}, V_{i+1}\rangle$. The merging condition is whether $\frac{n_I(V_i \cup V_{i+1})}{n_I(V_i)+n_I(V_{i+1})}$ is smaller than $\gamma$, a pre-defined parameter. The rationale is that we merge the rows in which a large number of intervals are neighboring. If rows $\langle K_i, V_i\rangle$ and $\langle K_{i+1}, V_{i+1}\rangle$ are merged, the new key is $[low_i, up_{i+1})$, and the new value is $V_i \cup V_{i+1}$. Moreover, all neighboring window intervals from $V_i$ and $V_{i+1}$ are merged to one interval.

The merge operation is actually a union operation between two ordered interval sequences, which can be implemented efficiently similar to the merge-sort algorithm. Since each window interval will be examined exactly once, its time complexity is $O(n_I(V_i) + n_I(V_{i+1}))$.

If the size of index exceeds memory capacity, we build the index as follows. In the first step, we divide time series into segments, and build the fixed-width range index for each segment in turn. After all segments are processed, we merge the rows of different segments. The second step visits index rows sequentially, which can be also divided into sub-tasks. Since each step can be divided into sub-tasks, the whole index building algorithm can be easily adapted to distributed environment, like MapReduce.

**Complexity analysis.** The process of building KV-index consists of two steps, generating rows with the fixed width, and merging them into varied-width ones. The first step scans all data in stream fashion, computes the mean value, and inserts $\langle \mu, offset\rangle$ entry into hash table. Note that the mean value of $X(i, l)$ can be computed based on that of $X(i-1, l)$, whose cost is $O(1)$. So the cost of the first step is $O(n)$. In the second step, we detect adjacent rows and merge them if necessary. Since the intervals are ordered within each row, the merge operation is similar to the merge sort, whose cost is $n_I(V_i) + n_I(V_{i+1})$. Therefore, the whole cost is $\sum_{i=1}^{D-1} n_I(V_i) + n_I(V_{i+1})$ ($D$ is the number of rows in first step). Because $n_I(V_i) \leq n_P(V_i)$ and $\sum_{i=1}^{D} n_P(V_i) = n - w + 1$, we can infer that its cost is $O(2n)$. In summary, the complexity of building index is $O(n)$.

All previous index-based approaches, like FRM and General Match, are based on R-tree, whose building cost is $O(n \cdot log_2(n))$ [13]. Moreover, they use DFT to transform each $w$-size window of $X$, whose cost is $w \cdot log_2(w)$. So the total transformation cost is $O(n \cdot w \cdot log_2(w))$. Therefore, building KV-index is more efficient.

## V. KV-MATCH

In this section, we present the matching algorithm KV-match, whose pseudo-code is shown in Algorithm 1.

### A. Overview

Initially, given query $Q$, we segment it into disjoint windows $Q_i$ of length $w$ ($1 \leq i \leq p = \left\lfloor \frac{|Q|}{w} \right\rfloor$), and compute mean values $\mu_i^Q$ (Line 1). We assume that $|Q|$ is an integral multiple of $w$. If not, we keep the longest prefix which is a multiple of $w$. According to the analysis in Section III, the rest part can be ignored safely.

The main matching process consists of two phases:

Phase 1: Index-probing (Line 2-12): For each window $Q_i$, we fetch a list of consecutive rows in KV-index according to the lemmas in Section III. Based on these rows, we generate a set of subsequence candidates, denoted as *CS*.

Phase 2: Post-processing (Line 13-18): All subsequences in *CS* will be verified by fetching the data and computing the actual distance.

Note that all four types of queries have the same matching process, the only difference is that in the index-probing phase, for each window, different types have the various row ranges, as introduced in Section III.

### B. Window Interval Generation

For each window $Q_i$, we calculate the range of $\mu_i^S$, $[LR_i, UR_i]$, firstly according to the query type. Then we visit KV-index with a single scan operation, which will obtain a list of consecutive rows, denoted as $RList_i = \{\langle K_{s_i}, V_{s_i}\rangle, \langle K_{s_i+1}, V_{s_i+1}\rangle, \cdots, \langle K_{e_i}, V_{e_i}\rangle\}$, which satisfies

**Algorithm 1** MatchSubsequence($X, w, Q, \varepsilon$)

1: $p \leftarrow \left\lfloor \frac{|Q|}{w} \right\rfloor, \mu_i^Q \leftarrow \text{avg}(Q_i)\ (1 \le i \le p)$
2: **for** $i \leftarrow 1, p$ **do**
3: $RList_i \leftarrow \{\langle K_{s_i}, V_{s_i}\rangle, \cdots, \langle K_{e_i}, V_{e_i}\rangle\}$
4: $IS_i \leftarrow \varnothing$
5: **for all** $\langle K_j, V_j\rangle \in RList_i$ **do**
6: $IS_i \leftarrow IS_i \cup \{WI\ |WI \in V_j\}$
7: SORT($IS_i$)
8: $CS_i \leftarrow \varnothing, shift_i \leftarrow (i-1) * w$
9: **for all** $WI \in IS_i$ **do**
10: $CS_i$.add($[WI.l - shift_i, WI.r - shift_i]$)
11: **if** $i = 1$ **then** $CS = CS_i$
12: **else** $CS \leftarrow$ INTERSECT($CS, CS_i$)
13: $answers \leftarrow \varnothing$
14: **for all** $WI \in CS$ **do**
15: $S \leftarrow X(WI.l, WI.r - WI.l + |Q|)$ ▷ Scan from data
16: **for** $j \leftarrow 1, |S| - |Q| + 1$ **do**
17: **if** $D(Q, S(j, |Q|)) \le \varepsilon$ **then** ▷ Extra test for cNSM
18: $answers$.add($S(j, |Q|)$)
19: **return** $answers$

$LR_i \in [low_{s_i}, up_{s_i})$ and $UR_i \in [low_{e_i}, up_{e_i})$. Note that the $s_i$-th row (or the $e_i$-th row) may contain mean values out of the range. However, it only brings negative candidates, without missing any positive one.

We denote all window intervals in $RList_i$ as $IS_i = \{WI\ |WI \in V_k, k \in [s_i, e_i]\}$. We use $WI \in IS_i$ to indicate that window interval $WI$ belongs to $IS_i$. Also, for any window position $j$ in $WI$ ($WI \in IS_i$), we have $j \in IS_i$.

According to Eq. (6) and Eq. (7), we indicate the number of window intervals in $IS_i$ as $n_I(IS_i)$, and the number of window positions in $IS_i$ as $n_P(IS_i)$. Note that the window intervals in $IS_i$ are disjoint with each other. To facilitate the next "interaction" operation, we sort these intervals in ascending order, that is, $IS_i[k].r < IS_i[k+1].l$, where $IS_i[k]$ is the $k^{th}$ window interval in $IS_i$ (Line 7).

### C. The Matching Algorithm

Based on $IS_i$ ($1 \le i \le p$), we generate the final candidate set $CS$ with an "intersection" operation. We first introduce the concept of *candidate set* for $Q_i$, denoted as $CS_i$ ($1 \le i \le p$). For window $Q_1$, any window position $j$ in $IS_1$ maps to a candidate subsequence $X(j, |Q|)$. Therefore, the candidate set for $Q_1$, denoted as $CS_1$, is composed of all positions in $IS_1$. $CS_1$ is still organized as a sequence of ordered non-overlapped window intervals, like $IS_1$.

For $Q_2$, each window position in $IS_2$ also corresponds to a candidate subsequence. However, position $j$ in $IS_2$ corresponds to the candidate subsequence $X(j-w, |Q|)$, because $X(j, w)$ is its second disjoint window. So the candidate set for $Q_2$, denoted as $CS_2$, can be obtained by left-shifting each window position in $IS_2$ with $w$. Similarly, $CS_3$ is obtained by left-shifting the positions in $IS_3$ with $2 \cdot w$. In general, for window $Q_i$ ($1 \le i \le p$), the candidate set $CS_i$ is as follows,

$$CS_i = \{j - (i-1) \cdot w | j \in IS_i\}$$

The shifting offset for $Q_i$ is denoted as $shift_i = (i-1) \cdot w$. All candidate sets $CS_i$ ($1 \le i \le p$) are still organized as an ordered sequence of non-overlapped window intervals. Moreover, it can be easily inferred that $n_I(CS_i) = n_I(IS_i)$ and $n_P(CS_i) = n_P(IS_i)$.

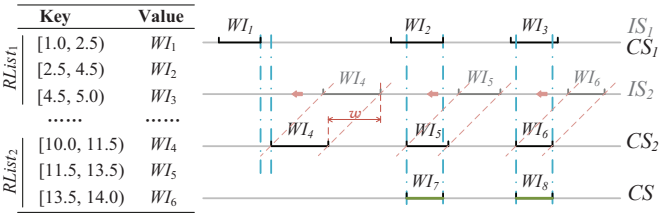



Fig. 7. Example of the matching algorithm

Through combining the lemmas in Section III and the definition of $CS_i$, we can obtain two important properties,

**Property 1.** *If $X(j, |Q|)$ is not contained by certain $CS_i$ ($1 \le i \le p$), then $X(j, |Q|)$ and $Q$ are not matched.*

**Property 2.** *If $X(j, |Q|)$ and $Q$ are matched, position $j$ belongs to all candidate sets $CS_i$, that is, $j \in CS_i$ ($1 \le i \le p$).*

Now we present our approach to intersect $CS_i$'s to generate the final $CS$. It consists of $p$ rounds (Line 2-12). In the first round, we fetch $RList_1$ from the index, and generate $IS_1$ and $CS_1$. We initialize $CS$ as $CS_1$. In the second round, we fetch $RList_2$, and generate $CS_2$ by shifting all window intervals in $IS_2$ with $(2-1) \cdot w = w$ (Line 9-10). Then we intersect $CS$ with $CS_2$ to obtain up-to-date $CS$ (Line 12). Because all intervals in $IS_i$, as well as $CS_i$, are ordered, the intersection operation can be executed by sequentially intersecting window intervals of $CS$ and $CS_2$, which is quite similar to merge-sort algorithm with $O(n_I(CS) + n_I(CS_2))$ complexity. In general, during the $i$-th round, we intersect $CS_i$ with $CS$ of the last round, and generate the up-to-date $CS$. After $p$ rounds, we obtain the final candidate set $CS$.

We illustrate the algorithm with the example in Fig. 7. $RList_1$ contains three intervals, $WI_1$, $WI_2$ and $WI_3$. $RList_2$ contains three intervals, $WI_4$, $WI_5$ and $WI_6$. $IS_1$ (or $IS_2$) contains all the intervals covered by $RList_1$ (or $RList_2$). $CS_1$ equals to $IS_1$, while $CS_2$ is generated by left-shifting $IS_2$ with offset $w$. Then we intersect $CS_1$ and $CS_2$ to get $CS$ in the second round, which is composed of $WI_7$ and $WI_8$.

In phase 2, according to $CS$, we fetch data to generate the final qualified results (Line 13-18). Formally, for each window interval $WI$ in $CS$, we fetch the subsequences $X(WI.l, WI.r - WI.l + |Q|)$ from data. Note that this subsequence contains $|WI|$ number of subsequences. For each fetched length-$|Q|$ subsequence, we calculate the distance from $Q$ and return the qualified ones. If the query is cNSM query, each subsequence needs to be normalized before computing the ED or DTW distance. Moreover, most lower bounds used in UCR Suite [8] can be also used here to speed up the verification, particularly for DTW measure.

## VI. KV-MATCH$_{DP}$

The basic KV-match uses a fixed window length $w$ to process the query, regardless of the query length. It has two limitations. First, the length of the supported query is limited. Second, we have less chance to exploit the characteristics of the query and the time series data to speed up processing.

In this section, we propose KV-match$_{DP}$, which is based on multiple indexes with variable window lengths. Formally, the lengths of windows to build the index are summarized by

two parameters, $w_u$ and $L$, where $w_u$ is the minimum window length and $L$ is the number of indexes. Then, the set of window lengths is $\Sigma = \{w_u * 2^{i-1} | 1 \leq i \leq L\}$. For example, suppose $w_u = 25$ and $L = 5$, we build indexes of length 25, 50, 100, 200 and 400 respectively. We use KV-index$_w$ to denote the index based on length-$w$ windows. The set of indexes can be built simultaneously by extending the index building algorithm in Section IV-B easily.

### A. Dynamic Query Segmentation

We process the query with multiple indexes simultaneously. That is, we split $Q$ into a sequence of disjoint windows of variable lengths, $\{Q_1, Q_2, \cdots, Q_p\}$, and process each $Q_i$ with KV-index$_{|Q_i|}$, which is more flexible to utilize the characteristics of the data. Once $Q$ is split, the following process is similar to that in KV-match. The only difference is that for window $Q_i$, we fetch $RList_i$ from index KV-index$_{|Q_i|}$. Note that although in Lemmas in Section III, $Q$ is split into equal-length windows, we can easily extend them to variable-length windows, since the proof always involves only one window.

The challenge here is how to split query $Q$ to achieve the best performance. We use *query segmentation* to represent the result of query splitting. A segmentation, denoted as $SG = \{r_1, r_2, \cdots, r_p\}$, means that $Q_1 = Q(1, r_1)$, $Q_2 = Q(r_1 + 1, r_2 - r_1)$ and so on. A high-quality segmentation should satisfy: 1) the length of each window belongs to $\Sigma$; 2) processing $Q$ with these windows results in high performance. We take the segmentation as an optimization problem and design an objective function to measure its quality.

### B. The Objective Function

We first analyze the key factors to impact the efficiency. The runtime of query processing $T$ is composed of $T_1$ and $T_2$, those of phase 1 and 2 respectively. According to our theoretical analysis and experimental verification, $T_2$ is more significant to the efficiency, while $T_1$ is more stable. So we utilize the efficiency of phase 2 to measure the segmentation quality. Phase 2 consists of two parts, data fetching and distance computation, the former of which, determined by $n_I(CS)$, is much more time-consuming.

Therefore, for a segmentation $SG$ of $Q$, after obtaining the final candidate set $CS$, we use $n_I(CS)$ to measure the quality of $SG$. The smaller $n_I(CS)$, the higher quality of $SG$. The challenge is we cannot obtain the exact value of $n_I(CS)$ without going through the index-probing phase. Moreover, although we can obtain the size of $n_I(IS_i)$'s from the meta table, we cannot compute $n_I(CS)$ with $n_I(IS_i)$'s directly.

To address this issue, we propose an objective function to estimate the value of $n_I(CS)$. The estimation is based on two assumptions. First, $IS_i$'s of disjoint windows are independent with each other ($1 \leq i \leq p$). Second, the size of each window interval in $IS_i$ is much smaller than $|X|$. So we can take each window interval as a single point in $X$, and these positions are distributed uniformly.

Next, we introduce our objective function, denoted as $\mathcal{F}$. Assume that we use $SG$ to split $Q$ into $Q_1, Q_2, \cdots, Q_p$, and obtain the size of each $IS_i$ ($1 \leq i \leq p$) based on the meta table. Then we estimate $n_I(CS)$ as follows. Based on these two assumptions, we can use $\frac{n_I(IS_1)}{n}$ to approximately represent the probability of an interval contained in $CS_1$, where $n$ is the length of $X$. It follows that $\frac{n_I(IS_1)}{n} * \frac{n_I(IS_2)}{n}$ is the probability of an interval contained in $CS_1 \cap CS_2$. Therefore, $\prod_{i=1}^{p} \frac{n_I(IS_i)}{n}$ is the probability of an interval contained in the final $CS$, which is proportional to $n_I(CS)$. It is obvious that the larger $p$, the smaller $\prod_{i=1}^{p} \frac{n_I(IS_i)}{n}$. So, to eliminate the effect of number of windows, we take geometric mean of this value as the final objective function $\mathcal{F}$, as follows,

$$\mathcal{F}(SG) = \sqrt[p]{\prod_{i=1}^{p} \frac{n_I(IS_i)}{n}} = \frac{1}{n} \sqrt[p]{\prod_{i=1}^{p} n_I(IS_i)} \tag{8}$$

The target segmentation is the one with the minimal value of $\mathcal{F}$[1].

### C. Two-dimensional DP Approach

We propose a two-dimensional dynamic programming algorithm to find the optimal $SG$. We first define the search space. Since the length of each window $Q_i$ must belong to $\Sigma$, so in any $SG = \{r_1, r_2, \cdots, r_p\}$, $r_i$ must be multiple times of $w_u$. Any $SG$ not satisfying this constraint is invalid. Given query $Q = (q_1, q_2, \cdots, q_m)$, we define the search space with sequence $Z = (1, 2, \cdots, m')$, where $m' = \lfloor \frac{m}{w_u} \rfloor$. Note that the values in $Z$ do not have impact on the generation of $SG$. The only effect of $Z$ is to constrain the search space of $SG$. Instead of finding $SG$ on $Q$ directly, we find it from $Z$, denoted as $SG_Z$, and then map it to $SG$ of $Q$ by multiplying each endpoint of $Z$ with $w_u$. For example, let $|Q| = 200$, $w_u = 25$ and $L = 3$. That is, we have three indexes, KV-index$_{25}$, KV-index$_{50}$ and KV-index$_{100}$. $SG_Z = \{2, 6, 7, 8\}$ corresponds to $SG = \{50, 150, 175, 200\}$. In this case, $Q$ is segmented into four windows, $Q(1, 50)$, $Q(51, 100)$, $Q(151, 25)$ and $Q(176, 25)$.

We search the optimal $SG_Z$ with two-dimensional dynamic programming from left to right on $Z$ sequentially. The first dimension represents the boundaries of segmentation, and the second represents the number of windows contained in a segmentation. We use $v_{i,j}$ to represent a sub-state of calculation process, which corresponds to the best segmentation of the prefix of $Z$, $Z(1, i)$, with $j$ number of windows. For any $j$ ($1 \leq j \leq m'$), the best segmentation is the one with minimum $v_{m',j}$. After obtaining all $v_{m',j}$'s, we select the minimal one as the final $SG_Z$, and map it to $SG$. The dynamic programming equation is presented as Eq. (9).

In Eq. (9), $\varphi$ represents the possible lengths of the window ending at $i$ in $SG_Z$, and it has $L$ possible values at most. $C_{i-\varphi+1,\varphi}$ is the value of $n_I(IS)$ for the disjoint window $Q((i-\varphi)*w_u+1, \varphi*w_u)$, which can be obtained from the meta table of KV-index$_{\varphi*w_u}$, as explained in Section V. The optimal $SG_Z$ and $SG$ can be recovered by leveraging backward-pointers.

$$v_{i,j} = \begin{cases} 1 & , i = 0 \wedge j = 0 \\ +\infty & , i = 0 \vee j = 0 \\ \min\limits_{\substack{\varphi=2^{k-1} \\ 1 \leq k \leq \min(L, \log_2(i)+1)}} \sqrt[j]{(v_{i-\varphi,j-1})^{j-1} * C_{i-\varphi+1,\varphi}} & , 1 \leq j \leq i \leq m' \end{cases} \tag{9}$$

The complete algorithm is shown in Algorithm 2.

*Analysis.* It happens that a large amount of windows of $X$ have similar mean values. In this case, certain rows in KV-index will have large value of $n_I$, which incurs large I/O cost to fetch $RList$ and large computation cost to merge $CS_i$ in each round. The KV-index$_{\text{DP}}$ can alleviate this phenomenon

[1]Since $\frac{1}{n}$ is a constant, we ignore it in the algorithm.

**Algorithm 2** Segment($w_u, L, Q$)

$m' \leftarrow \left\lfloor \frac{|Q|}{w_u} \right\rfloor, v_{i,j} \leftarrow +\infty, P_{i,j} \leftarrow -1 \ (0 \le i \le m')$
$v_{0,0} \leftarrow 1$
**for** $i \leftarrow 1, m'$ **do**
  **for** $j \leftarrow 1, i$ **do**
    **for** $k \leftarrow 1, \min(L, \log_2(i) + 1)$ **do**
      $\varphi \leftarrow 2^{k-1}$
      **if** $\sqrt[j]{(v_{i-\varphi,j-1})^{j-1} * C_{i-\varphi+1,\varphi}} < v_{i,j}$ **then**
        $v_{i,j} \leftarrow \sqrt[j]{(v_{i-\varphi,j-1})^{j-1} * C_{i-\varphi+1,\varphi}}$
        $P_{i,j} \leftarrow \varphi$
$SG \leftarrow \varnothing, i \leftarrow m', j \leftarrow \arg\min_x(v_{m',x}) \ (1 \le x \le m')$
**while** $i \neq -1$ **do**
  $SG$.add($i * w_u$)
  $i \leftarrow i - P_{i,j}, j \leftarrow j - 1$
**return** $SG$

to some extent, since the objective function prefer the query windows with smaller $n_I$.

Moreover, we can use some techniques to alleviate this phenomenon further. First, to reduce the duplicate index visit, we can cache the index rows already fetched. Then for each new $RList$, if partial of it is already in the cache, we only need to fetch the rest part from KV-index. Second, we can reorder $Q_i$'s to be processed according to the size of $RList_i$, which can be obtained easily from the meta data. In other words, we first process $Q_i$ with smaller $RList_i$, which can reduce both I/O cost and the merge computation cost. Third, note that each $CS_i$ is the *superset* of the true result, so we can only process a partial of query windows, instead of all of them, to obtain the final $CS$ without loss of correctness. By combining the second and third optimization, we can skip some rows with large $n_I$ by ranking them at the bottom position.

## VII. Implementation

We implement two versions of our approach to show the compatibility of our approach. One stores indexes in local disk files, and the other stores indexes on HBase [14]. Both are implemented with Java. The code and synthetic data generator are publicly available[1].

### A. Local File Version

To compare the efficiency with previous subsequence matching methods, we first implement KV-match on conventional disk files.

In data file, all time series values are stored one by one in binary format, and their offsets are omitted because they can be easily inferred from bytes' length. In index file, the rows of KV-index are also stored contiguously. The offset of each row is recorded in meta data, stored at the footer of the file. The meta data will be retrieved first before processing the query. The start offset and length of each sequential read can be inferred by binary search on the meta data, and then a seek operation will be used to fetch data from file.

### B. HBase Table Version

To verify the performance of KV-match for large data scale and test the scalability of our approach, we also implement it on HBase, where time series data and index are stored in tables respectively.

In time series table, time series is split into equal-length (1024 by default) disjoint windows, and each one is stored as a row. The key is the offset of the window, and value is the corresponding series data. In index table, a row of KV-index is stored as a row in HBase, and the meta table is also compacted to store as a row. We load the meta table to memory before processing the query. To take full advantage of the cluster, we adapt index building algorithm to the MapReduce framework.

### C. Compatibility with Other Systems

Moreover, our index structure can be easily transplanted to other modern TSDB's. The only requirement is the system provides the "scan" operation to perform sequential data retrieval. Many systems support this operation, As examples, Table II lists the API used to implement the scan operation on some popular storage systems.

TABLE II
Scan Operation on Popular Storage Systems

| System | Code Snippet of Retrieving Data in Specific Range |
|---|---|
| Local | `raf = new RandomAccessFile(file, "r");`<br>`raf.seek(offset);`<br>`raf.read(result, 0, length);` |
| HDFS | `fdis = FileSystem.get(conf).open(path);`<br>`fdis.seek(offset);`<br>`fdis.read(result, 0, length);` |
| HBase | `scan = new Scan(startKey, endKey);`<br>`results = table.getScanner(scan);` |
| LevelDB | `for (it->Seek(startKey); it->Valid() &&`<br>`it->key().ToString() < endKey;`<br>`it->Next()) ...` |
| Cassandra | `SELECT * FROM table WHERE`<br>`key >= startKey AND key < endKey` |

## VIII. Experiments

In this section, we conduct extensive experiments to verify the effectiveness and efficiency of the proposed approach.

### A. Datasets and Settings

*1) Real Datasets:* UCR Archive [15] is a popular time series repository, which includes many datasets widely used in time series mining research. We concatenate the time series in UCR Archive to obtain desired length time series.

*2) Synthetic Datasets:* We use synthetic time series to test the scalability of our approach. The series are generated by combining three types of time series as follows.

- Random walk. The start point and step length are picked randomly from $[-5, 5]$ and $[-1, 1]$ respectively;
- Gaussian. The values are picked from a Gaussian distribution with mean value and standard deviation randomly selected from $[-5, 5]$ and $[0, 2]$ respectively;
- Mixed sine. It is a mixture of several sine waves whose period, amplitude and mean value are randomly chosen from $[2, 10]$, $[2, 10]$ and $[-5, 5]$ respectively.

To generate a time series $X$, we execute the following steps repeatedly until $X$ is fully generated: i) randomly choose a type $t$, a length $l$ and the parameters according to type $t$; ii) generate a length-$l$ subsequence using type $t$ with parameters.

[1]https://github.com/DSM-fudan/KV-match

TABLE III
RESULTS OF RSM QUERIES UNDER ED MEASURE

| Approach | Selectivity | #candidates | #index accesses | Time (ms) |
|---|---|---|---|---|
| GMatch | $10^{-9}$ | 13.9 | 279.2 | 852.3 |
| | $10^{-8}$ | 1837.5 | 240.1 | 541.2 |
| | $10^{-7}$ | 239,857.4 | 226.2 | 5,817.5 |
| | $10^{-6}$ | 1,223,370.6 | 338.0 | 30,351.7 |
| | $10^{-5}$ | 1,410,563.0 | 313.6 | 34,916.4 |
| KVM-DP | $10^{-9}$ | 2,754.9 | 4.6 | 60.4 |
| | $10^{-8}$ | 6,313.2 | 4.5 | 70.8 |
| | $10^{-7}$ | 29,853.1 | 4.4 | 138.8 |
| | $10^{-6}$ | 113,434.1 | 6.0 | 567.4 |
| | $10^{-5}$ | 153,565.1 | 7.0 | 1,200.7 |

TABLE IV
RESULTS OF RSM QUERIES UNDER DTW MEASURE

| Approach | Selectivity | #candidates | #index accesses | Time (ms) |
|---|---|---|---|---|
| DMatch | $10^{-9}$ | 1,176,639.8 | 250.0 | 543.5 |
| | $10^{-8}$ | 1,278,894.9 | 276.1 | 1,424.2 |
| | $10^{-7}$ | 1,800,014.9 | 447.8 | 7,847.2 |
| | $10^{-6}$ | 2,406,697.3 | 619.2 | 29,952.9 |
| | $10^{-5}$ | 3,431,349.8 | 902.9 | 132,062.4 |
| KVM-DP | $10^{-9}$ | 25,423.9 | 4.7 | 115.3 |
| | $10^{-8}$ | 38,894.0 | 4.9 | 120.5 |
| | $10^{-7}$ | 87,002.5 | 5.3 | 634.1 |
| | $10^{-6}$ | 118,580.9 | 6.6 | 3,641.3 |
| | $10^{-5}$ | 218,965.5 | 7.1 | 21,348.2 |

*3) Counterpart Approaches:* For RSM, we compare our approach (*KVM* for short) with two index-based approaches, General Match [5] for ED and DMatch [16] for DTW. For cNSM, we compare with UCR Suite [8] and FAST [17].

General Match [5] (*GMatch* for short) is a classic R*-tree based approach for ED. We use the code from author, which stores indexes in local disk files. Since building and updating R*-tree in distributed environment is not straightforward, we only compare it with our local file version.

DMatch [16] is a duality-based subsequence matching approach for DTW, which is quite similar to other tree-style approaches. Because its code is not publicly available, we implement a C++ version based on General Match framework. The window length is set to 64 and each window is transformed to a 4-dimensional point by PAA.

UCR Suite [8] (*UCR* for short) finds the best normalized matching subsequence under both ED and DTW. It scans the whole time series data, and uses some lower-bound techniques to speed up the query processing. Its code is publicly available[1], which is implemented in C++ and reads data on local disks. To make the comparison fair, we alter it to $\varepsilon$-match problem. Moreover, we implement a Java version to retrieve data on HBase, and conduct experiments for both local file and HBase table version to compare its scalability with KV-match.

FAST [17] is a recent improvement on UCR Suite, which adds more lower-bound techniques to reduce the number of distance calculations. We use the code from author, and compare it with our local file version under both ED and DTW.

*4) Default Setting:* In KV-match$_{\text{DP}}$, $L$ is set to 5, and $\Sigma = \{25, 50, 100, 200, 400\}$. In index building algorithm, the initial fixed width $d$ is set to $0.5$ and the merge threshold $\gamma$ is set to $80\%$. All experimental results are averaged over 100 runs.

To test the performance of processing queries with arbitrary lengths, we generate queries of length $128, 256, \cdots, 8192$. For each length, 100 different query series are generated.

Experiments are executed on a cluster consisting of 8 nodes with HBase 1.1.5 (1 Master and 7 RegionServers). Each node is powered by Linux, and has two Intel Xeon E5 1.8GHz CPUs, 64GB memory, 5TB HDD storage. Experiments using local file version are executed on a single node of the cluster.

### B. Results of RSM Queries

We first compare KV-match$_{\text{DP}}$ with General Match and DMatch. The experiment is conducted on length-$10^9$ real dataset with queries of different selectivities. The results are shown in Table III and IV respectively.

[1]http://www.cs.ucr.edu/∼eamonn/UCRsuite.html

TABLE V
RESULTS OF CNSM QUERIES UNDER ED MEASURE

| Selectivity | KVM-DP (s) | | | | UCR | FAST |
|---|---|---|---|---|---|---|
| | $\alpha \backslash \beta'$ | 1.0 | 5.0 | 10.0 | Avg.(s) | Avg.(s) |
| $10^{-9}$ | 1.1 | 0.51 | 2.33 | 4.64 | 59.84 | 86.05 |
| | 1.5 | 0.56 | 2.58 | 5.05 | | |
| | 2.0 | 0.59 | 2.70 | 5.51 | | |
| $10^{-8}$ | 1.1 | 0.72 | 3.22 | 6.18 | 60.17 | 86.09 |
| | 1.5 | 1.00 | 4.60 | 8.98 | | |
| | 2.0 | 1.22 | 5.47 | 10.66 | | |
| $10^{-7}$ | 1.1 | 1.30 | 5.46 | 10.29 | 65.25 | 87.79 |
| | 1.5 | 2.82 | 11.53 | 21.75 | | |
| | 2.0 | 3.72 | 16.20 | 29.15 | | |
| $10^{-6}$ | 1.1 | 1.69 | 6.74 | 14.53 | 69.17 | 88.64 |
| | 1.5 | 3.15 | 15.19 | 27.53 | | |
| | 2.0 | 4.39 | 20.77 | 35.75 | | |
| $10^{-5}$ | 1.1 | 1.94 | 7.82 | 12.92 | 70.59 | 89.83 |
| | 1.5 | 4.23 | 15.98 | 28.26 | | |
| | 2.0 | 5.77 | 21.55 | 37.66 | | |

It can be seen that when the selectivity increases, the number of candidates of General Match explodes dramatically, and in the case of higher selectivities, it is much larger than that of ours. Although General Match converts all values in a window into a multi-dimensional point, which keeps more information than the mean value used in KV-index, it generates candidates only based on one single window. In contrast, our approach combines the pruning power of multiple windows, which can achieve smaller candidate set.

The number of index accesses of General Match is 20-30 times larger than that of ours. Due to fewer index accesses and less number of candidates, our approach achieves the overall performance improvement of one order of magnitude compared to General Match. An interesting phenomenon is that for queries of low selectivities ($10^{-8}$ or $10^{-9}$), the number of candidates of our approach is slightly larger than that of General Match. However, benefiting from fewer index accesses, we still achieve better overall performance.

Similar to General Match, DMatch also conducts large number of index accesses, and has to verify one or two orders of magnitude more candidates than ours. The reason is still the single window candidate generation mechanism and tree-style index structure, as General Match.

### C. Influence of Window Size $w$

In this experiment, we investigates the pruning performance of building index with the mean values. We compare the

TABLE VI
RESULTS OF CNSM QUERIES UNDER DTW MEASURE

| Selectivity | KVM-DP (s) | | | | UCR | FAST |
|---|---|---|---|---|---|---|
| | $\alpha \backslash \beta'$ | 1.0 | 5.0 | 10.0 | Avg.(s) | Avg.(s) |
| $10^{-9}$ | **1.1** | 0.72 | 2.71 | 3.71 | 139.57 | 77.5 |
| | **1.5** | 0.66 | 2.97 | 4.72 | | |
| | **2.0** | 0.78 | 3.37 | 6.00 | | |
| $10^{-8}$ | **1.1** | 0.89 | 2.66 | 5.31 | 140.06 | 78.57 |
| | **1.5** | 1.24 | 4.89 | 7.89 | | |
| | **2.0** | 1.43 | 5.01 | 9.21 | | |
| $10^{-7}$ | **1.1** | 1.88 | 6.61 | 10.02 | 142.99 | 85.07 |
| | **1.5** | 3.81 | 13.79 | 23.30 | | |
| | **2.0** | 4.46 | 15.92 | 33.00 | | |
| $10^{-6}$ | **1.1** | 5.58 | 14.29 | 18.69 | 153.88 | 103.60 |
| | **1.5** | 11.09 | 30.74 | 60.27 | | |
| | **2.0** | 11.40 | 33.72 | 60.56 | | |
| $10^{-5}$ | **1.1** | 19.75 | 36.61 | 49.94 | 177.28 | 137.01 |
| | **1.5** | 40.35 | 57.90 | 102.72 | | |
| | **2.0** | 44.07 | 76.23 | 106.97 | | |

number of candidates obtained from each query window $Q_i$ of KV-match and FRM [3] [1]. FRM is selected to compare because its mechanism is analogous to KV-match. FRM builds the index based on the sliding windows of $X$, and each window is transformed into an $f$-dimensional point. Then the transformed points are stored in R-tree. To process query $Q$, FRM splits $Q$ into $p$ number of disjoint windows $Q_i$ ($1 \leq i \leq p$). For each window, a set of candidates are obtained by a range query to R-tree. Then, the *union* of candidates of all windows forms the final candidate set. In contrast, in KV-match, the final candidate sets, $CS$, is the intersection of $CS_i$'s.

In Table VII, we show the ratio of number of candidates per window between our approach and FRM. The experiments are conducted on time series of length $10^9$. We run queries of different selectivities. For each selectivity, 100 randomly generated queries of length 2048 are processed, and the number of candidates are averaged. We compare KV-indexes and FRM with variable window sizes, 50, 100, 200, 400. Moreover, we also show the ratio of the number of final candidates between our approach and FRM.

It can be seen that our approach will generate more candidates per window, $CS_i$, especially for smaller $w$ and larger $|Q|$, since the range depends on $\frac{\varepsilon}{w}$. However, the number of final candidates, $CS$, of our approach is much smaller than that of FRM, because in KV-match, $CS$ is the intersection of $CS_i$'s, while in FRM, $CS$ is the union of $CS_i$'s. Consider it is more expensive to fetch the time series to compute the distance, reducing $CS$ is more beneficial. Moreover, for each $Q_i$, we only visit index with a sequential scan operation, while in FRM we need to visit multiple index nodes, which may incur more I/O cost. Finally, the mechanism of KV-match$_{\text{DP}}$ can avoid to use the query windows with many candidates.

### D. Results of cNSM Queries

In this experiment, we compare KV-match$_{\text{DP}}$ with UCR Suite and FAST for cNSM on local disk. The experiment is conducted on length-$10^9$ real dataset with queries of different selectivities. The results under ED and DTW measures are shown in Table V and VI respectively. For each selectivity, we report the runtime for different $\alpha$ and $\beta$. The constraints are also embedded into UCR Suite and FAST, so unqualified candidates are abandoned too. For simplicity, we only report the average runtime for each selectivity, because theirs runtime for queries in the same selectivity group is quite similar.

We use relative offset shifting $\beta'$ in cNSM experiments, which is the percentage of the value range of the whole data series. Therefore, $\beta = (\max(X) - \min(X)) * \beta'\%$.

It can be seen that when the selectivity increases, the runtime of KV-match increases steadily. When the selectivity is fixed, the runtime increases as $\alpha$ and $\beta$ increase. Because UCR Suite almost always scans the whole dataset, its runtime is more stable and dominated by I/O cost. The extra lower-bounds in FAST seems not efficient for ED, due to its overhead of data preparation. While for DTW, FAST achieves obvious improvement comparing to UCR Suite, especially for queries of low selectivities ($10^{-8}$ or $10^{-9}$). In most cases, our approach achieves the performance improvement of one to two orders of magnitude compared to them.

### E. Index Size and Building Time

We compare the index space cost and building time of KV-match$_{\text{DP}}$ and DMatch. GMatch has similar space cost and building time as those of DMatch, and so we do not show them in the results. The experiment is conducted on the local file version with real datasets. Results are shown in Fig. 8. We also show the size of time series data as dark blue bars.

It can be seen that the index sizes of both DMatch and KV-match$_{\text{DP}}$ are about 10% of data size, and the size of KV-match$_{\text{DP}}$ is slightly larger than that of DMatch. However, KV-match$_{\text{DP}}$ consists of 5 KV-indexes, so the size of a single KV-index is much smaller than that of DMatch. We also show the index building time as lines in Fig. 8. Our index is much more efficient to build, due to its simple structure. In the extremely large data scale (the trillion-length time series), it takes 36 hours to build all 5 KV-indexes for KV-match$_{\text{DP}}$ on HBase.

Moreover, we test the influence of window size $w$ on the index size and building time. In Table VIII, we show the index size and building time of KV-index with fixed $w$ on time series of length $10^9$. It can be seen that as $w$ increases, both index size and building time decrease gradually. This is because that larger $w$ makes the mean values of the adjacent windows more similar, and correspondingly makes $n_I(V_i)$ smaller, which reduces both the index size and the building time.

### F. Scalability

To investigate the scalability of our approach, we use longer synthetic time series, from length-$10^9$ to length-$10^{12}$, to compare KV-match$_{\text{DP}}$ and UCR Suite for cNSM queries. Both time series data and our index is stored as HBase table, and both ED and DTW measures are compared. We set $\alpha = 1.5$, $\beta' = 1.0$, and hold selectivity to $10^{-7}$ by adjusting $\varepsilon$. The results are shown in Fig. 9.

It can be seen that KV-match$_{\text{DP}}$ is faster than UCR Suite under both ED and DTW measures by almost two to three orders of magnitude. For trillion-length ($10^{12}$) series, we can process queries by 127s (under ED measure) and 243s (under DTW measure) on average, which shows great scalability.

### G. KV-match$_{DP}$ vs. the Basic KV-match

In this experiment, we compare the runtime between KV-match$_{\text{DP}}$ and KV-match for RSM queries. We build 5 KV-indexes with $w$ as $25, 50, 100, 200, 400$ respectively. For KV-match$_{\text{DP}}$, we set $\Sigma = \{25, 50, 100, 200, 400\}$ to use all these

[1] FRM is a special case of General Match when $J = 1$.

TABLE VII
THE RATIO OF KV-MATCH AND FRM ON WINDOW AVERAGED CANDIDATES VS. FINAL TOTAL CANDIDATES

| Selectivity | $\|Q\|$ | #candidates per window | | | | #candidates in final | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $w=50$ | 100 | 200 | 400 | $w=50$ | 100 | 200 | 400 |
| $10^{-6}$ | 512 | 14.3 | 21.8 | 29.7 | 31.3 | 0.002 | 0.104 | 2.626 | 31.287 |
| | 1024 | 40.5 | 58.7 | 47.9 | 20.8 | 0.081 | 0.086 | 0.750 | 7.055 |
| | 2048 | 52.1 | 65.5 | 59.3 | 21.2 | 0.010 | 0.007 | 0.041 | 0.323 |
| | 4096 | 65.5 | 69.8 | 64.4 | 37.9 | 0.112 | 0.040 | 0.029 | 0.143 |
| | 8192 | 91.9 | 82.6 | 70.6 | 57.4 | 0.108 | 0.080 | 0.049 | 0.069 |
| $10^{-5}$ | 512 | 12.4 | 8.1 | 5.9 | 8.4 | 0.091 | 0.226 | 1.561 | 8.352 |
| | 1024 | 18.3 | 10.1 | 7.0 | 5.8 | 0.184 | 0.029 | 0.062 | 1.044 |
| | 2048 | 41.0 | 18.4 | 10.0 | 10.2 | 0.209 | 0.076 | 0.002 | 0.040 |
| | 4096 | 81.1 | 33.6 | 18.2 | 15.6 | 0.247 | 0.131 | 0.025 | 0.006 |
| | 8192 | 168.7 | 69.9 | 33.9 | 24.4 | 0.354 | 0.170 | 0.043 | 0.002 |
| $10^{-4}$ | 512 | 13.1 | 7.7 | 4.7 | 4.7 | 0.183 | 0.273 | 1.138 | 4.714 |
| | 1024 | 23.7 | 10.3 | 5.5 | 3.4 | 0.204 | 0.029 | 0.080 | 0.587 |
| | 2048 | 62.3 | 23.0 | 9.6 | 5.7 | 0.483 | 0.181 | 0.026 | 0.071 |
| | 4096 | 165.0 | 60.3 | 24.5 | 11.3 | 0.752 | 0.582 | 0.388 | 0.137 |
| | 8192 | 281.4 | 103.5 | 40.2 | 17.5 | 0.535 | 0.400 | 0.196 | 0.042 |
| $10^{-3}$ | 512 | 13.5 | 5.8 | 2.7 | 2.3 | 0.149 | 0.207 | 0.577 | 2.315 |
| | 1024 | 28.9 | 11.6 | 5.5 | 2.6 | 0.340 | 0.099 | 0.171 | 0.553 |
| | 2048 | 68.5 | 26.1 | 10.7 | 5.2 | 0.531 | 0.319 | 0.087 | 0.152 |
| | 4096 | 161.8 | 61.4 | 24.3 | 10.0 | 0.728 | 0.520 | 0.280 | 0.063 |
| | 8192 | 266.2 | 152.6 | 61.3 | 24.9 | 0.940 | 0.704 | 0.508 | 0.277 |

TABLE VIII
INFLUENCE OF $w$ ON INDEX SIZE AND BUILDING TIME

| $w$ | Size (MB) | Building time (s) |
|---|---|---|
| **25** | 354.09 | 299.38 |
| **50** | 287.21 | 234.30 |
| **100** | 236.49 | 227.06 |
| **200** | 194.52 | 210.18 |
| **400** | 155.47 | 198.12 |

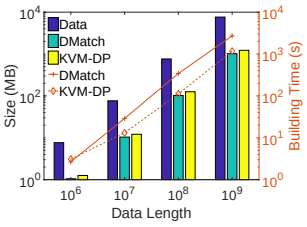



Fig. 8. Size & building time

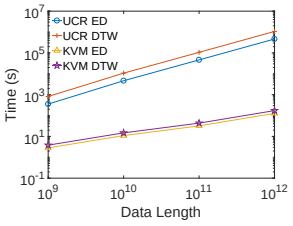
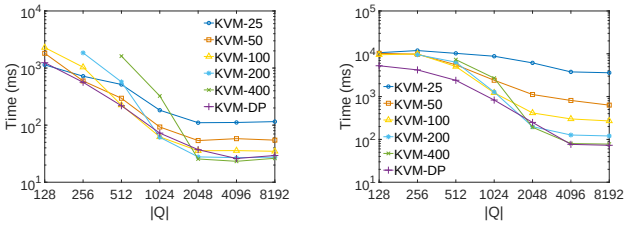



Fig. 9. Scalability




(a) $\varepsilon = 10$ (b) $\varepsilon = 100$

Fig. 10. Effect of dynamic window segmentation

indexes. The experiment is conducted with local file version on length-$10^9$ real dataset. Because the performance of a single index is highly related to the length of queries, we test the runtime of variable query lengths. Fig. 10 (a) and (b) show the results in the case of $\varepsilon = 10$ (representing low selectivity) and $\varepsilon = 100$ (representing high selectivity) respectively.

It can be seen that in most cases, KV-match$_{\text{DP}}$ outperforms all single indexes. On the contrary, the index with small window length is suitable only for shorter queries, while the index with large window length only works well on longer queries. The results verify the effectiveness of our query segmentation algorithm. KV-match$_{\text{DP}}$ can utilize the pruning power of multiple window lengths and leverage the data characteristics of the query sequences.

## IX. RELATED WORK

Subsequence matching problem has been studied extensively in last two decades.

**Approaches for RSM problem.** The pioneering work [3], FRM, used Euclidean distance as the similarity measure. It transforms each sliding window into a low-dimensional point and stores in R-tree. Disjoint windows of query series are also transformed and the candidates are retrieved by range queries on R-tree. To improve the efficiency, Dual-Match [18] extracts disjoint windows from data series and sliding windows from query series, which reduces the size of R-tree. General Match [5] generalizes both of them, and benefits from both point filtering effect in Dual-Match and window size effect in FRM. [19] builds multiple indexes and picks the optimal one to process the query according to the query length. All these approaches transform subsequences into low-dimensional points, and build R-tree as the index. This mechanism incurs large amount of index visits for large data scale. In contrast, KV-match only needs a scan operation for each $Q_i$.

Some works deal with RSM problem with other distance functions. The Dynamic Time Warping (DTW) distance is studied in [20], which proposes two lower bounds for DTW, LB_Keogh and LB_PAA. Also, DMatch [16] presents a duality-based approach for DTW by extending Dual-Match [18]. [4] supports multiple distances which satisfy specific property.

GDTW [21] is a general framework to apply the idea of DTW to more point-to-point distance functions. It is orthogonal to KV-match, because it focuses on the distance function while KV-match considers how to support both RSM and NSM queries simultaneously. Recently, adaptive approach is studied in whole matching problem [22], which first builds a coarse-

granularity index, then refines it during the query processing. This mechanism can reduce the initial construction time, and also make the index evolved according to the queries. This work deals with the whole matching problem. It is not trivial to adapt it to support both RSM and NSM problem.

Although there exist some works to support both ED and DTW measure, all these works don't support normalization.

**Approaches for NSM problem.** In [8], authors claim that normalization is vital and propose the UCR Suite to deal with normalized subsequence matching under both ED and DTW. Some optimizations are utilized to speed up. However, it needs to scan the whole sequence to find the qualifying subsequences, which is intolerable for large data scale. Recently, FAST [17] is proposed to improve the efficiency. It is based on UCR Suite, and adds some lower-bound techniques to reduce the number of candidate verification. Similar with UCR suite, FAST still needs to scan the whole time sequence. In contrast, KV-match proposes an index to deal with cNSM problem, which is more efficient. ONEX [23] utilizes the marriage of ED and DTW to support the normalized subsequence search. It builds the index for all possible subsequence lengths. For each subsequence length, it first normalizes all subsequences, and then builds the index based on a clustering approach. So it cannot support RSM and NSM problems simultaneously.

In sum, only UCR Suite [8] and FAST [17] support both RSM and NSM [1]. However, they need to scan the full time series. There is no existing work to build the index supporting both RSM and NSM problem.

## X. Conclusion and Future Work

We propose a novel constrained normalized subsequence matching problem (cNSM), which provides a knob to flexibly control the degree of offset shifting and amplitude scaling. We also propose a key-value index structure KV-index, corresponding matching algorithm KV-match, and the extended version KV-match$_{\text{DP}}$, to support both RSM and cNSM problems under either ED or DTW measure. Experimental results verify the efficiency and effectiveness. To the best of our knowledge, this is the first index-based work for normalized subsequence matching. In the future, we will try to support more distance measures, especially variable-length DTW.

## References


[1] T. Rakthanmanon and E. Keogh, "Fast shapelets: A scalable algorithm for discovering time series shapelets," in *ICDM*, 2013, pp. 668–676.

[2] C.-C. M. Yeh, Y. Zhu, L. Ulanova, N. Begum, Y. Ding, H. A. Dau, Z. Zimmerman, D. F. Silva, A. Mueen, and E. Keogh, "Time series joins, motifs, discords and shapelets: A unifying view that exploits the matrix profile," *DMKD*, vol. 32, no. 1, pp. 83–123, Jan. 2018.

[3] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos, "Fast subsequence matching in time-series databases," in *SIGMOD*, 1994, pp. 419–429.

[4] H. Zhu, G. Kollios, and V. Athitsos, "A generic framework for efficient and effective subsequence retrieval," in *VLDB*, 2012, pp. 1579–1590.

[5] Y.-S. Moon *et al.*, "General match: A subsequence matching method in time-series databases based on generalized windows," in *SIGMOD*, 2002, pp. 382–393.

[6] P. Papapetrou, V. Athitsos, M. Potamias, G. Kollios, and D. Gunopulos, "Embedding-based subsequence matching in time-series databases," *TODS*, vol. 36, no. 3, pp. 17:1–17:39, Aug. 2011.

[7] W.-S. Han, J. Lee, Y.-S. Moon, and H. Jiang, "Ranked subsequence matching in time-series databases," in *VLDB*, 2007, pp. 423–434.

[8] T. Rakthanmanon, B. Campana *et al.*, "Searching and mining trillions of time series subsequences under dynamic time warping," in *SIGKDD*, 2012, pp. 262–270.


[1] Although they aim to process the NSM query, we can easily adapt them to deal with the RSM query by removing the normalization step.


[9] E. Branlard, "Wind energy: On the statistics of gusts and their propagation through a wind farm."

[10] H. Sakoe and S. Chiba, "Dynamic programming algorithm optimization for spoken word recognition," *TSP*, vol. 26, no. 1, pp. 43–49, Feb 1978.

[11] B.-K. Yi and C. Faloutsos, "Fast time sequence indexing for arbitrary lp norms," in *VLDB*, 2000, pp. 385–394.

[12] Y. Zhu and D. Shasha, "Warping indexes with envelope transforms for query by humming," in *SIGMOD*, 2003, pp. 181–192.

[13] H. Alborzi and H. Samet, "Execution time analysis of a top-down r-tree construction algorithm," *Inf. Process. Lett.*, vol. 101, pp. 6–12, 2007.

[14] "Apache HBase," http://hbase.apache.org.

[15] Y. Chen, E. Keogh, B. Hu, N. Begum, A. Bagnall, A. Mueen, and G. Batista, "The ucr time series classification archive," www.cs.ucr.edu/~eamonn/time_series_data/.

[16] A. W.-C. Fu, E. Keogh, L. Y. H. Lau, C. A. Ratanamahatana, and R. C.-W. Wong, "Scaling and time warping in time series querying," *The VLDB Journal*, vol. 17, no. 4, pp. 899–921, Jul 2008.

[17] Y. Li, B. Tang, L. H. U, M. L. Yiu, and Z. Gong, "Fast subsequence search on time series data (Poster Paper)," in *EDBT*, 2017, pp. 514–517.

[18] Y.-S. Moon, K.-Y. Whang, and W.-K. Loh, "Duality-based subsequence matching in time-series databases," in *ICDE*, 2001, pp. 263–272.

[19] S.-H. Lim, H.-J. Park, and S.-W. Kim, "Using multiple indexes for efficient subsequence matching in time-series databases," in *DASFAA*, 2006, pp. 65–79.

[20] E. Keogh and C. A. Ratanamahatana, "Exact indexing of dynamic time warping," *KIS*, vol. 7, no. 3, pp. 358–386, Mar. 2005.

[21] R. Neamtu, R. Ahsan, E. Rundensteiner, G. N. Sarkozy, E. Keogh, A. Dau, C. Nguyen, and C. Lovering, "Generalized dynamic time warping: Unleashing the warping power hidden in point-wise distances," in *ICDE*, 2018.

[22] K. Zoumpatianos, S. Idreos, and T. Palpanas, "Indexing for interactive exploration of big data series," in *SIGMOD*, 2014, pp. 1555–1566.

[23] R. Neamtu, R. Ahsan, E. Rundensteiner, and G. Sarkozy, "Interactive time series exploration powered by the marriage of similarity distances," *Proc. VLDB Endow.*, vol. 10, no. 3, pp. 169–180, Nov. 2016.


## Appendix A
## Proof of Lemma 3

By combining Eq. (3) and $DTW_\rho(S,Q) \le \varepsilon$, we can easily infer the following three cases of $\mu_i^S$,

(a) $\mu_i^S > \mu_i^U$. In order to let $w \cdot (\mu_i^S - \mu_i^U)^2 \le \varepsilon$, $\mu_i^S$ should satisfy $\mu_i^U < \mu_i^S \le \mu_i^U + \frac{\varepsilon}{\sqrt{w}}$;

(b) $\mu_i^S < \mu_i^L$. In order to let $w \cdot (\mu_i^S - \mu_i^L)^2 \le \varepsilon$, $\mu_i^S$ should satisfy $\mu_i^L - \frac{\varepsilon}{\sqrt{w}} \le \mu_i^S < \mu_i^L$;

(c) Otherwise. Because $0 \le \varepsilon$ always holds, $\mu_i^L \le \mu_i^S \le \mu_i^U$.

Taking the union of above three cases, we will get Eq. (4). □

## Appendix B
## Proof of Lemma 4

Let $L' = \left(\frac{l_1-\mu_Q}{\sigma_Q}, \cdots, \frac{l_m-\mu_Q}{\sigma_Q}\right)$, $U' = \left(\frac{u_1-\mu_Q}{\sigma_Q}, \cdots, \frac{u_m-\mu_Q}{\sigma_Q}\right)$ be two length-$m$ series derived from $L$ and $U$. Since $L'$ and $U'$ are derived by a simple linear transformation, it can be easily inferred that $L'$ and $U'$ are still the lower and upper envelop of $\hat{Q} = \left(\frac{q_1-\mu_Q}{\sigma_Q}, \cdots, \frac{q_m-\mu_Q}{\sigma_Q}\right)$.

Similar to Lemma 3, if $DTW_\rho(\hat{S},\hat{Q}) \le \varepsilon$, we have $\hat{\mu}_i^S \in \left[\mu_i^{L'} - \frac{\varepsilon}{\sqrt{w}}, \mu_i^{U'} + \frac{\varepsilon}{\sqrt{w}}\right]$, where $\hat{\mu}_i^S$ is the mean value of the $i$-th windows of $\hat{S}$, $\mu_i^{L'}$ and $\mu_i^{U'}$ are the mean values of the $i$-th windows of $L'$ and $U'$ respectively.

By simple transformation, we have $\hat{\mu}_i^S = \frac{\mu_i^S-\mu^S}{\sigma^S}$, $\mu_i^{L'} = \frac{\mu_i^L-\mu^Q}{\sigma^Q}$ and $\mu_i^{U'} = \frac{\mu_i^U-\mu^Q}{\sigma^Q}$, so

$$\frac{\mu_i^S-\mu^S}{\sigma^S} \in \left[\frac{\mu_i^L-\mu^Q}{\sigma^Q} - \frac{\varepsilon}{\sqrt{w}}, \frac{\mu_i^U-\mu^Q}{\sigma^Q} + \frac{\varepsilon}{\sqrt{w}}\right] \tag{10}$$

In Eq. (10), $\mu_i^L$ and $\mu_i^U$ are the mean values of the $i$-th windows of $L$ and $U$ respectively.

Let $a = \frac{\sigma^S}{\sigma^Q}$ and $b = \mu^S - \mu^Q$, By replacing $\sigma^S = a\sigma^Q$ and $\mu^S = \mu^Q + b$ in Eq. (10), we can get

$$\mu_i^S \in \left[\left(\mu_i^L - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q, \left(\mu_i^U - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q\right]$$

where $a \in [\frac{1}{\alpha}, \alpha]$ and $b \in [-\beta, \beta]$. Similar to the proof of Lemma 2, we can obtain that the range of $\mu_i^S$ is exactly Eq. (5). □