学校代码： 10246

硕 士 学 位 论 文

（学术学位）

支持标准化和动态时间弯曲的时间序列子序列相似性查询算法研究

Time Series Subsequence Matching Approach Supporting Normalization and Time Warping

编　　号： 16210240016

专　　业： 计算机软件与理论

院　　系： 计算机科学技术学院

完 成 日 期： 2019 年 3 月 25 日

目　录

摘　要

随着数据中心、物联网等新兴应用的不断涌现，时间序列数据的规模呈现出爆炸式的增长。时间序列数据的分析挖掘也逐渐成为一个热门研究方向，而时间序列子序列相似性查询是时间序列数据挖掘的一个重要基础任务。子序列相似性查询分为全序列匹配和子序列匹配两种，而全序列匹配时子序列匹配的一种特殊情况，所以子序列匹配难度更高也得到更多的研究和关注。现有的所有基于索引的子序列匹配方法只考虑了在原始序列上的匹配（RSM），并没有支持子序列标准化后的匹配。虽然 UCR Suite 等方法可以处理标准化后的子序列匹配（NSM），但其需要扫描全部时间序列数据，对于处理长序列数据非常耗时。在本文中，我们首先提出了一种新的问题，称作有限制的标准化子序列匹配问题（cNSM），虽然我们在 NSM 问题的基础上添加了一些限制，但新的 cNSM 问题提供了灵活控制偏移量和幅度缩放程度的可能，这也使得用户可以通过构建的索引来处理查询。基于此，我们提出了一种新的索引结构 KV-index，以及相应的查询算法 KV-match。通过单一的索引，我们的方法可以同时支持处理在欧氏距离和 DTW 距离下的 RSM 和 cNSM 查询。索引 KV-index 是一种键值对结构，其可以很简单地实现在本地文件和 HBase 数据表上。为了支持任意长度的查询，我们进一步扩展 KV-match 为 KV-match$_{DP}$，这一增强算法可以对查询序列进行动态分段，并利用多层不同窗口长度的索引进行查询。我们在合成和真实数据集上进行了大量实验，充分验证了所提出方法的有效性和效率。

关键字：时间序列；子序列查询；标准化；动态时间弯曲；索引
中图分类号：TP311

Abstract

The volume of time series data has exploded due to the popularity of new applications, such as data center management and IoT. Data mining and analysis for time series data has also attracted many attention. Time series 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 dynamically segments the query series and 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.

Keywords: Time series Subsequence matching Normalization DTW Indexing
CLC number: TP311

符号表

X	时间序列 $(x_1, x_2, \cdots, x_n)$
$\|X\|$	时间序列 X 的长度，$\|X\| = n$
$X(i,l)$	从时间序列 X 的 i 位置开始，长度为 l 的子序列
$\hat{X}$	时间序列 X 的标准化序列
$\hat{X}(i,l)$	子序列 $X(i,l)$ 的标准化序列
$\mu_{i,l}^X$	子序列 $X(i,l)$ 的均值
$\sigma_{i,l}^X$	子序列 $X(i,l)$ 的标准差
$D(X,Y)$	时间序列 X 和 Y 间的距离
X_i	时间序列 X 的第 i 段长度为 w 的不相交窗口
μ_i^X	时间序列 X 的第 i 段不相交窗口的均值
σ_i^X	时间序列 X 的第 i 段不相交窗口的标准相差
WI	由连续窗口偏移位置组成的窗口区间
IS_i	满足第 i 段查询窗口 Q_i 的窗口区间集合
CS_i, CS	第 i 段查询窗口 Q_i 及前缀窗口 $Q_j(1 \leq j \leq i)$ 共同的的候选集
n_I, n_P	窗口区间及窗口偏移位置的数量

第 1 章　绪论

时间序列子序列匹配问题是时间序列数据挖掘的基础，是许多其他算法的核心子问题。而现有算法一方面无法适应海量数据规模和分布式计算环境，另一方面无法高效处理需要标准化和多种相似性度量的分析场景。因此，支持标准化和动态时间弯曲的子序列相似性查询问题具有很强研究价值和应用场景。

1.1　研究背景

时间序列数据在几乎所有人类活动中都是普遍存在的，包括医学、金融和科学等领域。近几年来，随着数据中心、物联网等新兴应用的普及，时间序列数据的规模也在不断扩大。因此，人们对查询分析和挖掘时间序列数据也充满了浓厚的兴趣[1, 2]。

时间序列子序列匹配问题是许多其他时间序列挖掘算法的核心子问题，其也能在大量实际问题中得到应用，如：医疗诊断、语音识别、气候分析和金融分析等。更确切地讲，时间序列子序列匹配问题就是给定一条长时间序列 X，对于任意的查询序列 Q 和距离阈值 ε，在时间序列 X 中找出所有与查询序列 Q 的距离处于阈值 ε 以内的子序列。

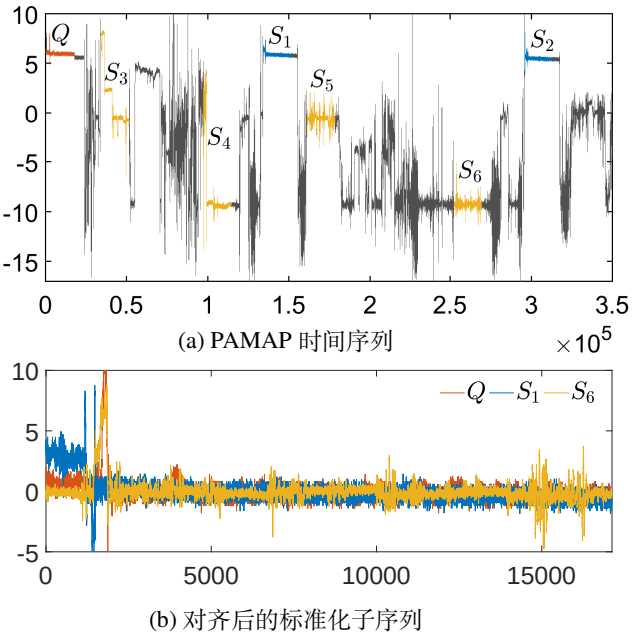
FRM[3] 是时间序列子序列匹配问题的开拓性工作。之后提出的很多方法，一些是改进了其效率[4, 5]，一些将其扩展以处理更多距离函数[6, 7]，如欧氏距离（ED）和动态时间弯曲距离（DTW）等。然而，所有这些方法只考虑了原始序列上的子序列匹配问题（RSM）。

近几年来，研究者们意识到了子序列标准化的重要性[8]。比较标准化后的子序列比直接比较原始子序列更有意义。UCR Suite[8] 是目前能解决标准化后的子序列匹配问题（NSM）的最好方法。更近一段时间，一种基于 UCR Suite 的方法[9] 通过设计更有效的下界进一步提升了效率。

然而，之前提出的 NSM 问题主要存在两方面的缺点。

首先，其需要完整扫描时间序列 X，这对于长时间序列是非常耗时的。例如，对于长度为 10^9 的时间序列，UCR Suite 就需要花费超过 100 秒来处理长度为 1,000 的查询序列。其论文[8] 也分析了不可能为 NSM 问题构建索引的原因。

其次，NSM 查询还可能得到一些不满足用户意图的结果。这是因为 NSM 问题完全忽略了幅度缩放和偏移程度。然而，在现实应用中，幅度缩放和偏移程度



(a) PAMAP 时间序列

(b) 对齐后的标准化子序列

(c) 查询序列 Q 相关信息

偏移位置 [标号]	长度
877 [Q]	17,124

(d) NSM 查询结果

偏移位置 [标号]	距离
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
...	...

★ $\varepsilon = 200.0$

图 1-1　cNSM 问题示例

可能代表了某些特定的物理机制或状态。用户可能只是希望找出那些与查询序列处于类似状态的子序列。下面我们就举例说明这个问题。

图 1-1(a) 所示的是从老年人身体活动检测数据集（PAMAP）[1] 中提取的手部 Z 方向加速度传感器的数据。被检测者交替进行了多种活动，如坐着、站立、跑步等。每一种活动持续 3 分钟，且数据采集的频率是 100Hz。我们用躺着对于的序列作为查询序列（Q 的信息如图 1-1(c) 所示），用来查找其他“躺着”的子序列。我们以 Q 发起查询请求，图 1-1(d) 列出了排名最靠前的结果。不幸的是，所有 4 个排名靠前的结果对应的是别的活动。子序列 S_3 和 S_5 对应坐着，而子序列 S_4 和 S_6 对应休息。虽然子序列 S_1 和 S_2 是想要的结果（对应躺着），但它们排在结果的前 20 名以外。我们在图 1-1(b) 中画出了标准化后的序列 Q、S_1 和 S_6，可以看到很难在标准化以后区分它们。

通过观察图 1-1(a)，我们可以简单地通过增加限制达到过滤不想要结果的目的：输出的子序列必须具有与查询序列 Q 类似的均值。事实上，这一 NSM 的新查询类型，*NSM 附加一些限制*，在许多应用中是非常有用的，以下我们给出两个例子。

- 工业应用：在风力发电领域，LIDAR 系统可以提供风扰动的预览信息[10]。极端阵风过程（EOG）是一种典型的阵风模式，代表的是一种风速在短时间内剧烈变化的现象。图 1-2显示了一条典型的 EOG 模式序列。这种模式非常重要，因为它可能造成风机的损坏。所有出现的 EOG 模式序列都具有相似的形态，而且因为风速不可能变得任意大，模式的波动程度也处在一

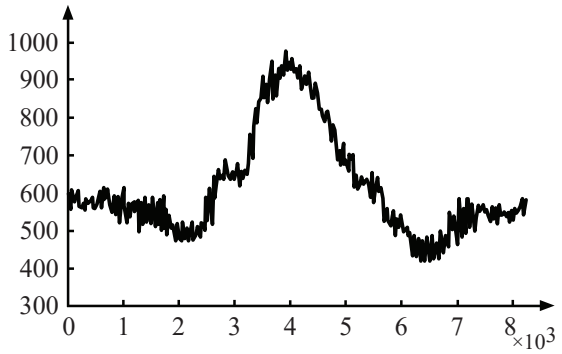


图 1-2 EOG 模式形态

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

图 1-3 模式间均值和标准差差异

定的范围以内。过于陡峭或平缓的波动都不是真实的 EOG 模式序列。如果想要在历史数据中找出所有出现的 EOG 模式，我们可以用一条典型的 EOG 模式作为查询序列，再附加上值域的范围。

- 物联网应用：当集装箱卡车通过桥梁时，桥梁上装备的压力压力传感器将会得到一种特定的波动模式序列。模式序列的值的范围是与卡车的重量有关的。如果有了一段模式序列作为查询序列，我们可以附加设定均值范围的限制，来搜索重量处在特定范围内的卡车。

注意到以上应用也都不能被 RSM 问题处理。由于幅度缩放和偏移的存在，迫使我们设定一个非常大的距离阈值，这将带来非常多的假阳性结果。

另外，为了验证提出的新查询类型的普适性，我们在一些知名真实时间序列评测数据集上调查了基本模式（Motif）对之间的关系。基本模式的挖掘[2] 是一个重要的时间序列挖掘任务，其可以找出一些具有最小标准化距离的子序列对（或集合）。对于任意一对基本模式子序列 X 和 Y，我们在图 1-3中列出了它们之间的相对均值差（ΔMean= $\frac{|\mu^X-\mu^Y|}{\max-\min}$）和标准差的差别比例（$\Delta$Std= $|\frac{\sigma^X}{\sigma^Y}|$）。从中可以看到，虽然这些基本模式子序列对是在没有任何限制的前提下找出来的（类似于 NSM 查询），但是它们之间的均值和标准差是非常接近的。所以我们可以仍然可以通过 cNSM 查询，即 NSM 查询附加一些小的限制的方法，找出它们。

1.2 研究内容

在本文中，我们正式定义一种新的子序列匹配问题，称为有限制的标准化子序列匹配问题（简称为 cNSM）。其在传统标准化子序列匹配问题 NSM 的基础上，添加了两个约束条件，一个是对于均值的限制，一个是对于标准差的限制。

cNSM 问题的一个示例查询请求类似于“给定一条查询序列 Q，其均值为

μ^Q，标准差为 σ^Q。返回满足以下条件的子序列 S：（1）$Dist(\hat{S}, \hat{Q}) \leq 1.5$；（2）$|\mu^Q - \mu^S| \leq 5$；（3）$0.5 \leq \sigma^Q/\sigma^S \leq 2$。”

通过这一约束，cNSM 问题提供了一种灵活控制偏移移位（以均值为代表）和幅度缩放（以标准差为代表）程度的方式。更进一步，cNSM 问题还为我们创造了为标准化子序列匹配问题建立索引的机会。

但是要解决提出的 cNSM 问题，主要有以下两方面的挑战。

首先，我们如何能高效地处理 cNSM 查询？一种直接的方法是首先应用 UCR Suite，找出没有限制的结果，再利用均值和标准差的限制过滤那些不符合条件的结果。然而，这仍然需要完整扫描原始序列数据。我们是否可以通过构建索引使得查询处理更为高效？

其次，在探索性和交互式的查询模式中，用户通常用相似的子序列进行搜索。用户还可能尝试不同的距离函数，例如欧氏距离（ED）或动态时间弯曲距离（DTW）等。同时，用户可能希望同时尝试 RSM 和 cNSM 查询。我们如何通过构建一个索引来支持所有这些查询类型？

1.3　主要贡献

除了提出 cNSM 问题，我们还在以下方面做出了贡献。

- 我们针对 4 种查询类型：RSM-ED、RSM-DTW、cNSM-ED 和 cNSM-DTW，分别提出了过滤条件，并证明了正确性。这些条件使得我们可以构建索引，并保证不会遗漏正确答案；

- 我们提出了一种新的索引结构 KV-index 和相应的查询处理方法 KV-match，可以同时支持所有 4 种查询类型。最大的优势是我们在一个索引上就可以同时处理不同类型的查询。另外，KV-match 对索引只有少有的几次*连续范围扫描*，而不是传统 R 树索引中对树中节点的大量随机访问，这使得查询处理过程更为高效；

- 为了高效支持任意长度的查询序列，我们还扩展 KV-match 为 KV-match$_{\mathrm{DP}}$，其可以同时使用多层不同窗口长度的索引。我们进行了大量实验，结果验证了我们所提出方法的正确性和高效。

1.4　本文组织结构

本文的组织结构如下：

第 1 章　绪论

本章主要介绍了时间序列数据挖掘的背景和重要性，时间序列子序列相似性查询是很多其他算法的基础核心。现有的时间序列子序列查询算法无法高效

地解决标准化的子序列匹配问题，也无法满足日益增长的交互式探索式的相似性查询需求。我们提出了一种新型的子序列匹配问题，在标准化子序列匹配问题的基础上，增加了一些约束条件，使得通过索引加速查询变得可能。而且实验表明，这种添加的限制并不会破坏查询的效果，在很多应用场景下反而能反映用户的真实需求。

第 2 章 相关工作与研究现状

本章主要介绍了时间序列相似性查询的相关工作和研究现状。时间序列相似性查询因为是重要的数据挖掘问题，已经得到了大量的研究。我们从相似性度量、解决不同相似性匹配问题的方法等角度详细介绍了各种现有方法，分析了它们的优缺点和适用范围，并着重阐述了与我们提出方法的不同。

第 3 章 子序列匹配理论基础研究

本章主要阐述了我们提出的有限制标准化子序列匹配问题，给出了形式化的定义。同时，我们针对多种查询类型提出了一种通用的过滤条件的形式。我们对于支持的四种查询类型分别给出了具体的过滤条件范围，并给出了证明。通过这一统一的过滤条件形式，我们可以只构建一个索引就同时支持多种查询类型。

第 4 章 子序列匹配索引与算法设计

本章主要描述了我们提出的索引结构和构建算法，以及与之配套的查询处理算法。我们较为简洁的键值对索引结构可以很好地适应不同的存储系统，构建算法也能较为简单地适配到分布式计算环境中，方便处理海量时间序列数据。查询处理算法利用统一过滤条件形式，只需读取很少几次索引，而且是连续范围读取，即可生成候选集。同时最终的候选集是所有窗口候选集的交集，与之前算法的并集相比，可以明显减少最终候选验证的负担，提高查询效率。

第 5 章 子序列匹配算法增强与优化

本章主要在上一章提出算法的基础上，进一步进行增强和优化。为了适应不同长度的查询序列，并尽可能充分利用数据序列和查询序列的特征，我们扩展出了增强版算法，其可以同时使用多层不同窗口长度的索引，利用动态规划算法将查询序列根据特点进行分段，并组合各段的候选得到最终候选集。另外，我们还提出了查询窗口顺序调整和窗口数量削减等优化手段，进一步提高了查询处理的效率。

第 6 章 实验分析

本章主要对实验过程和结果进行了论述。为了验证所提出方法的有效性和效率，我们进行了大量的实验，从多方面对提出方法进行了验证，并于现有算法进行了比较。实验结果表明，我们提出的算法具有良好的可扩展性，可以高效而准确地处理海量时间序列数据。

第 7 章 总结与展望

本章主要总结全文工作的内容，分析可以改进之处，并提出对未来子序列相似性查询领域的展望。

第 2 章　相关工作与研究现状

时间序列相似性查询分为全序列匹配和子序列匹配问题，而全序列匹配是子序列匹配的一种特殊情况，所以子序列匹配的相关研究受到了更多关注。近年来，子序列标准化匹配因其适应能力更强、可以消除扰动等特点得到了更多的认可，而现有子序列匹配算法无法高效解决这一问题，不能建立索引加速查询，只能完整扫描原始数据，这种方法无法适应海量数据规模。我们主要从相似性度量、*RSM* 问题相关方法和 *NSM* 问题相关方法等角度对现有工作进行总结。

2.1　相似性度量

确定两条时间序列是否相似的关键是如何衡量它们之间的相似性。研究者针对不同的场景和需求，设计出了各种各样的相似性度量方法，应用最为广泛的是欧氏距离（ED）和动态时间弯曲距离（DTW）。

2.1.1　欧氏距离

欧式距离（ED）是较为基础的一种距离度量方式，它需要完全将两条时间序列的各个点相应对齐，累计计算对应点之间的距离，得到的总和即为两条序列的距离。具体计算方式如下：

定义 2.1 (欧氏距离 ED) 对于给定的两条长度均为 m 的时间序列 S 和 S'，它们的欧式距离记为：

$$ED(S, S') = \sqrt{\sum_{i=1}^{m}(s_i - s'_i)^2} \tag{2.1}$$

这种计算方式非常简洁，同时可以利用三角不等式等进行估算和优化。相关的下界定理的研究也较为成熟，总体计算复杂度低，应用非常广泛。但是这种计算方式忽略了时间序列的随机可变性，例如传感器数据，序列可能符合一定的形态，但每次的具体的数据又会有不小的差异，有时还会有横向（时间方向）的偏移。如果只用欧氏距离，在某些场景下无法准确衡量相似程度。

2.1.2　动态时间弯曲距离

为了解决时间维度上的偏移，研究者提出了动态时间弯曲距离（DTW），这一度量方式运用了动态规划的思想，两台序列之间的点不再是一一对应的关系，

而是找出一种最优的匹配，使得总距离最小。具体计算方式如下：

定义 2.2 (动态时间弯曲距离 DTW) 对于给定的两条长度均为 m 的时间序列 S 和 S'，它们的 DTW 距离记为：

$$
\begin{aligned}
&DTW(\langle\rangle,\langle\rangle)=0; \qquad DTW(S,\langle\rangle)=DTW(\langle\rangle,S')=\infty;\\
&DTW(S,S')=\sqrt{(s_1-s'_1)^2+\min\begin{cases}DTW(suf(S),suf(S'))\\ DTW(S,suf(S'))\\ DTW(suf(S),S')\end{cases}}
\end{aligned}
\tag{2.2}
$$

其中，$\langle\rangle$ 表示空时间序列，而 $suf(S)=(s_2,\cdots,s_m)$ 表示时间序列 S 的后缀序列。

在 DTW 中，为了表示两条序列间的最优对应关系，定义了弯曲路径（warping path）这一矩阵结构。矩阵的元素 (i,j) 代表第一条序列的 s_i 点与第二条序列的 s'_j 点相对应。为了降低计算复杂度，我们应用 Sakoe-Chiba band[11] 来限制路径的宽度，记为 ρ。任意矩阵元素 (i,j) 需要满足 $|i-j|\le\rho$。而当宽度 $\rho=0$ 时，DTW 就将退化为欧氏距离。

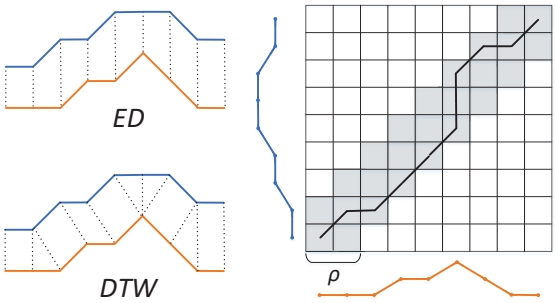


图 2-1　欧氏距离与 DTW 距离的比较

图 2-1 展示了欧氏距离与 DTW 距离的不同，并体现了 DTW_ρ 的弯曲路径。虽然 DTW 消除了一些序列间时间上的扭曲，但是其计算过程更为耗时。

2.2　解决 RSM 问题的方法

对时间序列子序列匹配的研究一开始集中在对于原始序列的匹配问题上，直接将数据序列中的子序列与查询序列进行比较，并算出距离。这种计算方式较为直接和高效，相关研究也较成熟。

2.2.1　FRM

FRM[3] 是子序列匹配的开拓性工作，其使用欧式距离作为相似性度量方法。它将每一个滑动窗口通过 DFT 变换转换为一个低维空间中的点，并存储在 R 树

中。查询序列的不相交窗口同样被转换为低维空间中的点，再通过 R 树上的范围查询得到候选集。

值得注意的是，R 树中需要存储所有滑动窗口的信息，但是这样存储的信息量太大了，所以 FRM 存储的是最小包络矩形（MBR）而不是单点，这会给查询时带来更多的假阳性候选。

2.2.2 Dual-Match

为了提高效率，之后提出的 Dual-Match[12] 与 FRM 相反，其从原始数据中提取不相交窗口，而从查询序列中提取滑动窗口，这样可以减少 R 树中存储的信息量。虽然查询效率会有一定损失，但相比 FRM 存储时用的 MBR，Dual-Match 总体上效率是更好的。

2.2.3 General Match

之后提出的 General Match[5] 通用化了 FRM 和 Dual-Match 中窗口的形式，不再是绝对的滑动窗口和不相交窗口，而是跳跃式窗口，这样也就可以同时利用 Dual-Match 中的点过滤效果和 FRM 中的窗口大小的优势了。

2.2.4 其他 RSM 方法

还有一些工作可以在其他距离度量函数下处理 RSM 问题。有工作[13] 研究了动态时间弯曲距离（DTW），并提出了两种 DTW 距离的下界函数，LB_Keogh 和 LB_PAA。另外，DMatch[14] 基于 Dual-Match[12] 进行扩展，提出了针对 DTW 距离的对偶方法。还有工作[4] 能够支持满足特定性质的多种距离函数。

另外还有方法[15] 通过建立多个索引，并根据查询序列的长度挑选其中最优的一个索引来处理查询。

GDTW[16] 是一种将 DTW 距离应用到更多点对点距离函数的通用框架。但它的主要研究方向与我们完全不同，他关注于距离度量函数，而我们是在考虑如何同时支持 RSM 和 NSM 查询。

最近，渐进式方法在全序列匹配问题中被提出[17]，其首先构建一个粗粒度的索引，之后再在查询处理过程中将其不断细化。这种机制可以减少索引的初始构建时间，也能使得索引可以根据查询的情况不断演化。但这一工作只是针对全序列匹配问题，并不能将其应用到 RSM 或 NSM 问题中去。

2.2.5 RSM 方法总结

FRM、Dual-Match 和 General Match 这些方法都是将子序列转换为低维空间的点，并构建 R 树作为索引。对于大规模数据而言，这一机制会带来大量的索

引访问。相反，我们提出的方法对于每一个查询窗口只需要对索引进行一次连续范围读取。

另外，虽然有一些工作是可以同时支持欧氏距离和 DTW 距离的，但它们都不能支持子序列标准化。

2.3　解决 NSM 问题的方法

随着子序列匹配问题研究的深入，RSM 问题的缺点也逐渐得到关注，其无法消除查询和匹配序列间的细微偏移和差异，容易受到数据波动和环境噪音的干扰。因此，研究者提出了基于子序列标准化的 NSM 问题。

2.3.1　UCR Suite

在 UCR Suite[8] 的论文中，作者提出标准化是非常重要的。UCR Suite 可以处理欧氏距离和 DTW 距离下的标准化子序列匹配问题。之后还有一些改进它的优化工作被相继提出。然而，这一方法需要完整扫描原始数据以找到符合要求的子序列，这对于处理大规模数据是无法容忍的。

2.3.2　其他 NSM 方法

之后还有不少解决 NSM 问题的方法被提出。最近，FAST[9] 提出了一种优化 UCR Suite 的方法。其在 UCR Suite 的基础上，再添加了几个下界过滤技术，以进一步减少需要验证的候选数量。但与 UCR Suite 一样，FAST 仍然需要完整扫描原始数据。

ONEX[18] 利用了欧式距离和 DTW 距离之间的关系，以支持标准化子序列匹配问题。它需要对于所有可能的子序列长度构建索引。对于每一种子序列长度，它首先标准化所有这种长度的子序列，再通过一种聚类的方法构建索引。但是它并不能同时支持 RSM 和 NSM 问题。

2.3.3　NSM 方法总结

UCR Suite 等方法需要完整扫描数据序列，这对于处理大规模数据非常耗时。相反，我们提出的 cNSM 问题提供了建立索引解决标准化子序列查询的可能。我们提出的方法可以利用索引处理 cNSM 问题，这样更为高效。

综上所述，只有 UCR Suite[8] 和 FAST[9] 是可以同时支持 RSM 和 NSM 问题的。然而，它们需要完整扫描原始数据 [1]。目前，还没有工作可以通过构建索引的方式同时支持 RSM 和 NSM 问题。

1　虽然它们是为 NSM 查询而设计的，但我们可以通过去除标准化步骤，简单将它们改为处理 RSM 查询。

2.4 本章小结

在本章中，我们回顾了现有的时间序列子序列相似性查询方法。其中有在原始数据序列上直接进行匹配的算法，也有子序列标准化的匹配算法。而目前还没有方法能够利用索引处理标准化的子序列匹配问题。

第 3 章　子序列匹配理论基础研究

时间序列子序列相似性查询工作经常是渐进式的，它需要数据分析人员探索式的精细化查询请求，不断明确自己的查询意图。这是因为，分析人员一开始很可能并不知道需要用哪一种相似性度量方法，而针对不同相似性度量不停重建索引，明显会影响查询工作的效率。所以我们提出只构建一个索引，就可以通用于不同的相似性度量方法。在本章中，我们阐述这种方法的理论基础，提出一种统一的过滤条件形式，并给出每种查询类型的具体范围和证明。

3.1　预备知识

在本节中，我们将介绍时间序列和一些相关概念，并给出本文解决问题的形式化定义。

3.1.1　相关概念

我们首先介绍时间序列的概念、标准化的定义和距离度量的计算方法等。

定义 3.1 (时间序列) 时间序列是由一系列数值元素排列组成的有序列表，记为：

$$X = (x_1, x_2, \cdots, x_n) \tag{3.1}$$

其中，$n = |X|$ 表示时间序列 X 的长度。

定义 3.2 (子序列) 时间序列 X 的长度为 l 的子序列是在时间序列 X 中截取出的一段较短的序列，记为：

$$X(i, l) = (x_i, x_{i+1}, \cdots, x_{i+l-1}) \tag{3.2}$$

其中，$1 \leq i \leq n - l + 1$。

定义 3.3 (标准化时间序列) 对于时间序列 $S = (s_1, s_2, \cdots, s_m)$，记 μ^S 和 σ^S 分别为序列 S 的均值和标准差。则时间序列 S 的标准化时间序列记为 $\hat{S}$，即：

$$\hat{S} = \left(\frac{s_1 - \mu^S}{\sigma^S}, \frac{s_2 - \mu^S}{\sigma^S}, \cdots, \frac{s_m - \mu^S}{\sigma^S} \right). \tag{3.3}$$

3.1.2　问题定义

接着，我们介绍时间序列子序列相似性查询的问题定义，并定义我们提出的 cNSM 问题。

定义 3.4 (RSM) 对于给定的长时间序列 X、查询序列 $Q(|X| \geq |Q|)$ 和距离阈值 ε $(\varepsilon \geq 0)$，在时间序列 X 中找出所有长度为 $|Q|$ 且满足以下条件的子序列 S。

$$D\big(S, Q\big) \leq \varepsilon \tag{3.4}$$

对于这种情况，我们称时间序列 S 与 Q 满足 ε-匹配。

定义 3.5 (NSM) 对于给定的长时间序列 X、查询序列 $Q(|X| \geq |Q|)$ 和距离阈值 ε $(\varepsilon \geq 0)$，在时间序列 X 中找出所有长度为 $|Q|$ 且满足以下条件的子序列 S。

$$D\big(\hat{S}, \hat{Q}\big) \leq \varepsilon \tag{3.5}$$

其中，时间序列 $\hat{S}$ 和 $\hat{Q}$ 分别为时间序列 S 和 Q 的标准化时间序列。

在 NSM 问题的基础上，cNSM 问题增加了两个限制条件。引入的阈值 α $(\alpha \geq 1)$ 和 β $(\beta \geq 0)$ 分别约束了序列的幅度缩放和偏移程度。

定义 3.6 (cNSM) 对于给定的长时间序列 X、查询序列 $Q(|X| \geq |Q|)$ 、距离阈值 ε $(\varepsilon \geq 0)$ 以及约束阈值 α 和 β，在时间序列 X 中找出所有长度为 $|Q|$ 且满足以下条件的子序列 S。

$$D\big(\hat{S}, \hat{Q}\big) \leq \varepsilon \ \cap \ \frac{1}{\alpha} \leq \frac{\sigma^S}{\sigma^Q} \leq \alpha \ \cap \ -\beta \leq \mu^S - \mu^Q \leq \beta. \tag{3.6}$$

阈值 α 和 β 越大，约束程度越松。对于这种情况，我们称时间序列 S 和 Q 满足 $(\varepsilon, \alpha, \beta)$-匹配。

如果阈值 $\alpha = 1$，那么这将要求时间序列 $\hat{S}$ 与 $\hat{Q}$ 具有相等的标准差。同理，如果阈值 $\beta = 0$，那么时间序列 $\hat{S}$ 和 $\hat{Q}$ 将具有相等的均值。

以上公式中的距离 $D(\cdot, \cdot)$ 既可以是欧氏距离，也可以是 DTW 距离。在本文中，我们将建立一种统一的索引，同时支持 4 种类型的查询，分别是：欧氏距离下的 RSM（RSM-ED）、DTW 距离下的 RSM（RSM-DTW）、欧氏距离下的 cNSM（cNSM-ED）和 DTW 距离下的 cNSM（cNSM-DTW）。

3.2　统一过滤条件形式

我们首先定义一种统一的过滤条件形式，用来过滤不符合条件的子序列。对于我们支持的 4 种查询类型，这个过滤条件的形式是一样的，这种统一的特性使得后文的查询算法可以在一种索引上同时支持多种查询类型。

更确切的讲，对于查询 Q 和长度为 m 的子序列 S，我们将它们对齐地切分为长度为 w 的多组不相交窗口。Q（或者 S）的第 i 个窗口可以表示为 Q_i（或者 S_i），即：$Q_i = (q_{(i-1)*w+1}, \cdots, q_{i*w})$，其中 $1 \le i \le p = \left\lfloor \frac{m}{w} \right\rfloor$。

对于每组窗口，我们期望找出一些可以用来建立过滤条件的特征。在本文中，我们选择利用一个单值特征：窗口的均值。

这种选择有两方面的好处：首先，单值特征允许我们只建立一维的索引，这会给索引读取带来很大的性能提升；其次，我们能在 RSM 和 cNSM 上都设计出关于均值的过滤条件。

我们将 Q_i 和 S_i 的均值分别表示为 μ_i^Q 和 μ_i^S。这个过滤条件一共有 p 段，第 i 段可表示为 $[LR_i, UR_i]$，其中 $1 \le i \le p$。如果 S 是一个符合条件的子序列，那么对于任意的 i，μ_i^S 必然落在 $[LR_i, UR_i]$ 范围内。如果任何一个 μ_i^S 处于这个范围以外，我们就可以放心地将 S 排除，而不会丢失任何符合条件的结果。

3.3 欧式距离下的 RSM 查询

我们先介绍欧氏距离下的 RSM 查询的过滤条件，这是过滤条件最简单的形式。同时我们借助 RSM 查询的一个例子，简要说明整体的查询处理方法。

以下是欧式距离下的 RSM 查询的过滤条件，以及证明过程。

引理 3.1 (RSM-ED) 如果序列 S 和 Q 符合欧氏距离下的 ε-匹配，即：

$$ED(S, Q) \le \varepsilon \tag{3.7}$$

那么 μ_i^S 必须满足：

$$\mu_i^S \in \left[\mu_i^Q - \frac{\varepsilon}{\sqrt{w}}, \mu_i^Q + \frac{\varepsilon}{\sqrt{w}} \right] \tag{3.8}$$

其中 $1 \le i \le p$。

证明 基于欧氏距离的定义，可得：

$$ED^2(S, Q) = \sum_{k=1}^{n} \left(s_k - q_k\right)^2 \ge \sum_{j=(i-1)*w+1}^{i*w} \left(s_j - q_j\right)^2 \tag{3.9}$$

根据 APCA[19] 中的推论，可得：

$$\sum_{j=(i-1)*w+1}^{i*w} \left(s_j - q_j\right)^2 \ge w * \left(\mu_i^S - \mu_i^Q\right)^2 \tag{3.10}$$

如果 $D(S, Q) \le \varepsilon$，那么经过不等式变形，以下公式依然成立：

$$\left(\mu_i^S - \mu_i^Q\right)^2 \le \frac{\varepsilon^2}{w} \tag{3.11}$$

所以，公式 (3.8) 得证。 ∎

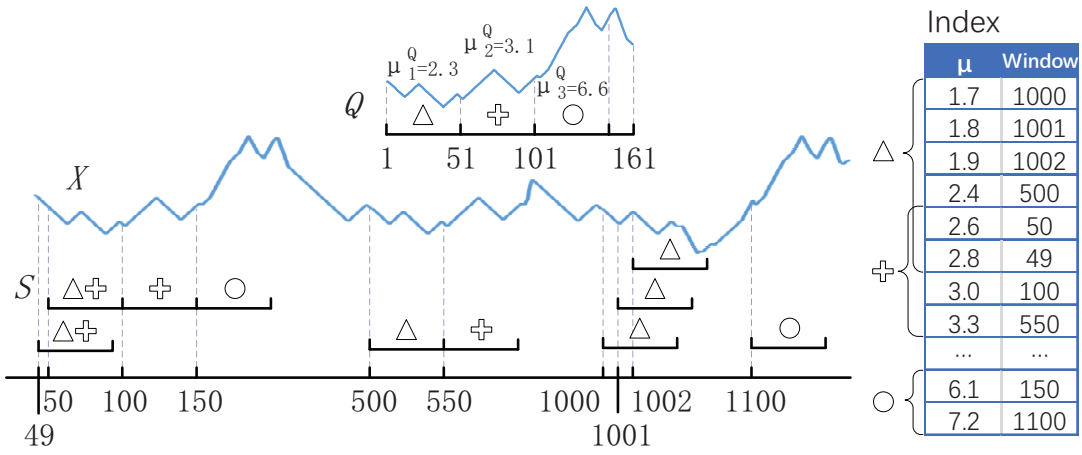


图 3-1　KV-match 查询处理过程示例

现在，我们用图 3-1中的例子来简要说明我们的查询处理方法。

假设 X 是一条很长的时间序列，Q 是长度为 161 的查询序列。我们的目标是从 S 中找出所有长度也为 161，同时满足 $ED(S,Q) \le \varepsilon$ 的所有子系列 X。窗口长度参数 w 设定为 50。

我们将 Q 切分为 3 段长度为 50 的不相交窗口，Q_1、Q_2 和 Q_3[1]。根据引理 3.1，对于任意符合条件的子序列 S，其第 i 个不相交窗口的均值一定落在 $\left[\mu_i^Q - \frac{\varepsilon}{\sqrt{50}}, \mu_i^Q + \frac{\varepsilon}{\sqrt{50}}\right]$ 范围内，其中 $i = 1, 2, 3$。

我们按以下方式建立索引以加速查询满足过滤条件窗口的过程。我们计算所有滑动窗口 $X(j,w)$ 的均值，表示为 $\mu(X(j,w))$，并建立一个有序列表存储 $\langle \mu(X(j,w)), j \rangle$ 条目。

有了这样的结构以后，我们通过以下两个步骤查找出候选。首先，对于每一个窗口 Q_i，我们通过一次范围扫描操作，获取所有均值落在 $\left[\mu_i^Q - \frac{\varepsilon}{\sqrt{50}}, \mu_i^Q + \frac{\varepsilon}{\sqrt{50}}\right]$ 范围内的滑动窗口，我们把这些根据 Q_i 找到的窗口表示为候选集 CS_i；然后，我们通过求取三个候选集 CS_1、CS_2 和 CS_3 的交集以生成最终的候选。

在图 3-1中，候选集 CS_1、CS_2 和 CS_3 的滑动窗口分别被标记为三角、叉和圆圈。唯一的候选是 $X(50, 161)$，因为相应窗口 $X(50, 50)$ 属于 CS_1、$X(100, 50)$ 属于 CS_2，而且 $X(150, 50)$ 属于 CS_3。

3.4　欧氏距离下的 cNSM 查询

因为我们的过滤条件的范围是建立在原始序列之上，而距离计算是基于标准化后的序列，所以设计欧式距离下的 cNSM 查询的过滤条件具有更大的挑战。

对于 NSM 查询而言，这个范围是完全不存在的，通过以下这个简单的例子

1　我们可以忽略剩余部分 $Q(151, 11)$，这并不会牺牲正确性，因为引理 3.1只是 RSM 的一个必要条件。

即可看出。假如 $Q = (1, 1, 1, -1, -1, -1)$ 且 $S = (100, 100, 100, -100, -100, -100)$，虽然 S 的取值更大，但仍热满足 $ED(\hat{Q}, \hat{S}) = 0$。事实上，S 的取值可以任意大，这将导致窗口均值的取值也可以任意大。

幸运的是，对于 cNSM 查询，加入的均值和标准差的限制可以避免这种极端的情况，也让我们可以设计出过滤条件的范围。这是因为，如果特定的 μ_i^S 是一个非常大的正数，为了保持总体的均值在限制范围内，必然存在另一个窗口 S_j，其均值 μ_j^S 是一个非常大的负数。然而，这两个非常大的均值（一正一负）将会导致总体的标准差非常大，而不能满足标准差的限制范围。

现在我们给出欧式距离下 cNSM 查询的过滤条件范围的形式化定义，并证明。我们用 μ^S 和 μ^Q 分别表示 S 和 Q 的总体均值，用 σ^S 和 σ^Q 分别表示 S 和 Q 的总体标准差，用 $\hat{S}$ 和 $\hat{Q}$ 分别表示 S 和 Q 的标准化序列。

引理 3.2 (cNSM-ED) 如果序列 S 和 Q 符合欧氏距离下的 $(\varepsilon, \alpha, \beta)$-匹配，即：

$$ED(\hat{S}, \hat{Q}) \le \varepsilon \ \cap \ \frac{1}{\alpha} \le \frac{\sigma^S}{\sigma^Q} \le \alpha \ \cap \ -\beta \le \mu^S - \mu^Q \le \beta \tag{3.12}$$

那么 μ_i^S 必须满足：

$$\mu_i^S \in \left[v_{\min} + \mu^Q - \beta, v_{\max} + \mu^Q + \beta\right] \tag{3.13}$$

其中 $1 \le i \le p$，且：

$$v_{\min} = \min\left\{\alpha \cdot \left(\mu_i^Q - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right), \frac{1}{\alpha} \cdot \left(\mu_i^Q - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)\right\} \tag{3.14}$$

$$v_{\max} = \max\left\{\alpha \cdot \left(\mu_i^Q - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right), \frac{1}{\alpha} \cdot \left(\mu_i^Q - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)\right\} \tag{3.15}$$

证明 基于标准化欧氏距离的定义，可得：

$$ED\left(\hat{S}, \hat{Q}\right) = \sqrt{\sum_{j=1}^{m}\left(\frac{s_j - \mu^S}{\sigma^S} - \frac{q_j - \mu^Q}{\sigma^Q}\right)^2} \tag{3.16}$$

令 $a = \frac{\sigma^S}{\sigma^Q}$ 且 $b = \mu^S - \mu^Q$，其中 $a \in [\frac{1}{\alpha}, \alpha]$ 且 $b \in [-\beta, \beta]$。

如果 $ED\left(\hat{S}, \hat{Q}\right) \le \varepsilon$，那么以下公式成立：

$$\sum_{j=1}^{m}\left(\frac{s_j - \mu^Q - b}{a\sigma^Q} - \frac{q_j - \mu^Q}{\sigma^Q}\right)^2 \le \varepsilon^2 \tag{3.17}$$

根据 APCA[19] 中的推论，类似引理 3.1，对于第 i 组窗口 S_i 和 Q_i，可得：

$$\left(\frac{\mu_i^S - \mu^Q - b}{a\sigma^Q} - \frac{\mu_i^Q - \mu^Q}{\sigma^Q}\right)^2 \le \frac{\varepsilon^2}{w} \tag{3.18}$$

经过简单公式变形，对于任意的 (a,b) 组合，可以得到 μ_i^S 的范围如下：

$$\mu_i^S \in \left[\left(\mu_i^Q - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q, \left(\mu_i^Q - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q\right] \tag{3.19}$$

为了简化表述，令 $\mu_i^Q - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}} = A$ 且 $\mu_i^Q - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}} = B$。那么过滤条件范围 $[LR_i, UR_i]$ 如下：

$$\left[\min_{\substack{a\in[\frac{1}{\alpha},\alpha]\\ b\in[-\beta,\beta]}} \left\{Aa + b + \mu^Q\right\}, \max_{\substack{a\in[\frac{1}{\alpha},\alpha]\\ b\in[-\beta,\beta]}} \left\{Ba + b + \mu^Q\right\}\right] \tag{3.20}$$

如图3-2所示，矩形区域代表 a 和 b 的合法范围。

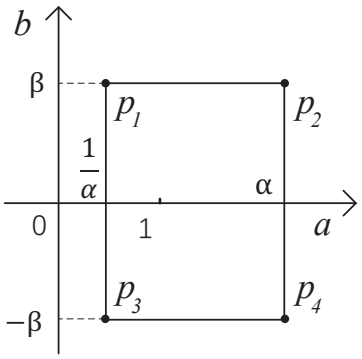


图 3-2　(a,b) 的合法范围

为了简化表述，令 $f(a,b) = Aa + b + \mu^Q$ 且 $g(a,b) = Ba + b + \mu^Q$。

显然，当 $b \in [-\beta,\beta]$ 时，$f(a,b)$ 和 $g(a,b)$ 都单调递增。而 a 的影响需分以下两种情况讨论：

- 如果 $A \geq 0$：当 $a \in [\frac{1}{\alpha},\alpha]$ 时，$f(a,b)$ 单调递增。当 $a = \frac{1}{\alpha}$ 且 $b = -\beta$ 时，$f(a,b)$ 取到最小值，如图 3-2 中的 p_3 点所示；
- 如果 $A < 0$：当 $a \in [\frac{1}{\alpha},\alpha]$ 时，$f(a,b)$ 单调递减。当 $a = \alpha$ 且 $b = -\beta$ 时，$f(a,b)$ 取到最小值，如图 3-2 中的 p_4 点所示。

所以：

$$LR_i = \min_{a\in[\frac{1}{\alpha},\alpha], b\in[-\beta,\beta]} f(a,b) = \min_{a\in\left\{\frac{1}{\alpha},\alpha\right\}} f(a,-\beta) \tag{3.21}$$

其中，$a \in \left\{\frac{1}{\alpha},\alpha\right\}$ 表示 a 可以取值 $\frac{1}{\alpha}$ 或 α。

同理，$g(a,b)$ 的最大值分为以下两种情况：

- 如果 $B \geq 0$：当 $a = \alpha$ 且 $b = \beta$ 时，$g(a,b)$ 最大，如图 3-2 中的 p_2 点所示；

- 如果 $B < 0$：当 $a = \frac{1}{\alpha}$ 且 $b = \beta$ 时，$g(a,b)$ 最大，如图 3-2 中的 p_1 点所示。

所以：

$$UR_i = \max_{a\in[\frac{1}{\alpha},\alpha],b\in[-\beta,\beta]} g(a,b) = \max_{a\in\left\{\frac{1}{\alpha},\alpha\right\}} g(a,\beta) \tag{3.22}$$

∎

3.5 DTW 距离下的 RSM 查询和 cNSM 查询

在具体介绍过滤条件的范围之前，我们首先回顾一下 DTW 距离下的查询包络和下界 LB_PAA[20] 的概念。

给定一个长度为 m 的查询序列 Q，为了处理 DTW_ρ 距离，我们需要两条长度也为 m 的查询包络序列，分别是下界包络线 L 和上界包络线 U。我们把序列 L 和 U 的第 i 个元素分别表示为 l_i 和 u_i，定义如下：

$$l_i = \min_{-\rho\le r\le\rho} q_{i+r},\ u_i = \max_{-\rho\le r\le\rho} q_{i+r} \tag{3.23}$$

LB_PAA 是定义在查询包络的基础之上的。我们将 L 和 U 切分为 p 段长度为 w 的不相交窗口，分别为 $(L_1, L_2, \cdots, L_p)$ 和 $(U_1, U_2, \cdots, U_p)$，其中 $L_i = (l_{(i-1)\cdot w+1}, \cdots, l_{i\cdot w})$ 且 $U_i = (u_{(i-1)\cdot w+1}, \cdots, u_{i\cdot w})$ $(1 \le i \le p = \lfloor\frac{m}{w}\rfloor)$。我们将 L_i 和 U_i 的均值分别表示为 μ_i^L 和 μ_i^U。对于任意长度为 m 的子序列 S，LB_PAA 的定义如下：

$$LB\_PAA(S,Q) = \sqrt{\sum_{i=1}^{p} w \cdot \begin{cases} (\mu_i^S - \mu_i^U)^2 & \text{if } \mu_i^S > \mu_i^U \\ (\mu_i^S - \mu_i^L)^2 & \text{if } \mu_i^S < \mu_i^L \\ 0 & \text{otherwise} \end{cases}} \tag{3.24}$$

同时 LB_PAA 需要满足 $LB\_PAA(S,Q) \le DTW_\rho(S,Q)$[20]。

现在，我们依次给出 DTW_ρ 距离度量下 RSM 查询和 cNSM 查询的过滤条件的范围。

引理 3.3 (RSM-DTW) 如果序列 S 和 Q 符合 DTW_ρ 距离度量下的 ε-匹配，即：

$$DTW_\rho(S,Q) \le \varepsilon \tag{3.25}$$

那么 μ_i^S 必须满足：

$$\mu_i^S \in \left[\mu_i^L - \frac{\varepsilon}{\sqrt{w}}, \mu_i^U + \frac{\varepsilon}{\sqrt{w}}\right] \tag{3.26}$$

其中 $1 \le i \le p$。

证明 已知公式 (3.24) 且 $DTW_\rho(S,Q) \le \varepsilon$，按 μ_i^S 分为以下三种情况：

(a) 如果 $\mu_i^S > \mu_i^U$：为了使得 $w \cdot (\mu_i^S - \mu_i^U)^2 \le \varepsilon$ 成立，μ_i^S 应该满足 $\mu_i^U < \mu_i^S \le \mu_i^U + \frac{\varepsilon}{\sqrt{w}}$；

(b) 如果 $\mu_i^S < \mu_i^L$：为了使得 $w \cdot (\mu_i^S - \mu_i^L)^2 \le \varepsilon$ 成立，μ_i^S 应该满足 $\mu_i^L - \frac{\varepsilon}{\sqrt{w}} \le \mu_i^S < \mu_i^L$；

(c) 其他：因为 $0 \le \varepsilon$ 始终成立，$\mu_i^L \le \mu_i^S \le \mu_i^U$。

综上，公式 (3.26) 得证。 ∎

引理 3.4 (cNSM-DTW) 如果序列 S 和 Q 符合 DTW_ρ 距离度量下的 $(\varepsilon, \alpha, \beta)$-匹配，即：

$$DTW_\rho(\hat{S}, \hat{Q}) \le \varepsilon \tag{3.27}$$

那么 μ_i^S 必须满足：

$$\mu_i^S \in \left[v_{\min} + \mu^Q - \beta, v_{\max} + \mu^Q + \beta \right] \tag{3.28}$$

其中 $1 \le i \le p$，且：

$$v_{\min} = \min \left\{ \alpha \cdot \left(\mu_i^L - \mu^Q - \frac{\varepsilon \sigma^Q}{\sqrt{w}} \right), \frac{1}{\alpha} \cdot \left(\mu_i^L - \mu^Q - \frac{\varepsilon \sigma^Q}{\sqrt{w}} \right) \right\} \tag{3.29}$$

$$v_{\max} = \max \left\{ \alpha \cdot \left(\mu_i^U - \mu^Q + \frac{\varepsilon \sigma^Q}{\sqrt{w}} \right), \frac{1}{\alpha} \cdot \left(\mu_i^U - \mu^Q + \frac{\varepsilon \sigma^Q}{\sqrt{w}} \right) \right\} \tag{3.30}$$

证明 令序列 $L' = \left(\frac{l_1 - \mu_Q}{\sigma_Q}, \cdots, \frac{l_m - \mu_Q}{\sigma_Q} \right)$ 且 $U' = \left(\frac{u_1 - \mu_Q}{\sigma_Q}, \cdots, \frac{u_m - \mu_Q}{\sigma_Q} \right)$，分别是从 L 和 U 中得到的长度为 m 的序列。

因为 L' 和 U' 只是经过了简单线性变换，易得 L' 和 U' 仍然分别是序列 $\hat{Q} = \left(\frac{q_1 - \mu_Q}{\sigma_Q}, \cdots, \frac{q_m - \mu_Q}{\sigma_Q} \right)$ 的下界和上界包络线。

类似引理3.3，如果 $DTW_\rho(\hat{S}, \hat{Q}) \le \varepsilon$ 成立，可得：

$$\hat{\mu}_i^S \in \left[\mu_i^{L'} - \frac{\varepsilon}{\sqrt{w}}, \mu_i^{U'} + \frac{\varepsilon}{\sqrt{w}} \right] \tag{3.31}$$

其中，$\hat{\mu}_i^S$ 是 $\hat{S}$ 的第 i 个窗口的均值，$\mu_i^{L'}$ 和 $\mu_i^{U'}$ 分别为 L' 和 U' 的第 i 个窗口的均值。

经过简单变形，可得：

$$\hat{\mu}_i^S = \frac{\mu_i^S - \mu^S}{\sigma^S}, \quad \mu_i^{L'} = \frac{\mu_i^L - \mu^Q}{\sigma^Q}, \quad \mu_i^{U'} = \frac{\mu_i^U - \mu^Q}{\sigma^Q} \tag{3.32}$$

所以：

$$\frac{\mu_i^S - \mu^S}{\sigma^S} \in \left[\frac{\mu_i^L - \mu^Q}{\sigma^Q} - \frac{\varepsilon}{\sqrt{w}}, \frac{\mu_i^U - \mu^Q}{\sigma^Q} + \frac{\varepsilon}{\sqrt{w}}\right] \tag{3.33}$$

在公式 (3.33) 中，μ_i^L 和 μ_i^U 分别是 L and U 的第 i 个窗口的均值。

令 $a = \frac{\sigma^S}{\sigma^Q}$ 且 $b = \mu^S - \mu^Q$。

在公式 (3.33) 中，$\sigma^S = a\sigma^Q$ 且 $\mu^S = \mu^Q + b$，可得：

$$\mu_i^S \in \left[\left(\mu_i^L - \mu^Q - \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q, \left(\mu_i^U - \mu^Q + \frac{\varepsilon\sigma^Q}{\sqrt{w}}\right)a + b + \mu^Q\right] \tag{3.34}$$

其中 $a \in [\frac{1}{\alpha}, \alpha]$ 且 $b \in [-\beta, \beta]$。

类似引理3.2的证明过程，可得 μ_i^S 的范围，公式 (3.28) 得证。 ∎

我们为每一种支持的查询类型给出了过滤条件的范围，这意味着我们可以在单一的索引上同时支持这些查询类型。在处理不同查询类型时，唯一的区别就是使用不同的过滤条件范围。这一特性可以极大地方便探索式查询搜索任务。

3.6 本章小结

在本章中，我们首先介绍了时间序列子序列查询的相关概念，并给出了所解决问题的形式化定义。之后我们设计并给出了一种统一的过滤条件的形式，并为每一种查询类型给出了具体的条件范围和证明。

第 4 章　子序列匹配索引与算法设计

根据提出的统一过滤条件形式，我们可以设计出同时支持多种查询类型的索引结构，索引虽然相较之前方法显得简单，但配合查询处理算法可以更高效地得到候选集。我们的查询算法可以只对索引进行很少的几次连续范围查询，并将各个窗口的候选的交集作为最终候选集。而现有算法大多基于 R 树，这带来大量索引随机访问，并且最终候选集还是各个窗口候选的并集。所以，我们提出的解决方案具有更高的效率，也能更好适应海量数据规模和分布式计算环境。

4.1　索引

本节中，我们介绍索引 KV-index 的组织结构，以及相匹配的索引构建算法。

4.1.1　索引组织结构

如图 3-1 所示，简易索引结构差不多有 $|X|$ 行，与原始数据规模相当，这将带来巨大的存储开销。为了避免这个问题，我们提出一种更为紧凑高效的索引结构，这种结构充分利用了数据的本地化特性，即时间上相邻的数据点的值也相近的特点。这也意味着，相邻滑动窗口的均值可能会较为相似。

逻辑上，索引结构 KV-index 由多行有序键值对组成。第 i 行的键，记为 K_i，是一个滑动窗口均值的范围，即：

$$K_i = [low_i, up_i) \tag{4.1}$$

其中，low_i 和 up_i 分别为 K_i 表示的均值范围的左端点和右端点。这是一个左闭右开的范围区间，且索引内相邻行的范围是不相交的。

索引行键的值，记作 V_i，是由均值落在 K_i 范围内的滑动窗口组成的集合。为了简化表述，我们用偏移位置代表每一个窗口，即将滑动窗口 $X(j,w)$ 记为 j。

为了进一步节省存储空间并利于之后的相似性匹配算法，我们将 V_i 中的窗口偏移位置进行重新组织。将 V_i 中的偏移位置首先递增排序，接着把相邻的窗口偏移位置合并为窗口区间，记为 WI。这样以后，V_i 包含的是一个或多个有序的不相交的窗口区间。

定义 4.1 (窗口区间) 将序列 X 的第 l 个至第 r 个长度为 w 的滑动窗口记为窗口区间 $WI = [l, r]$，其是由 $\{X(l,w), X(l+1,w), \cdots, X(r,w)\}$ 组成的滑动窗口集合，其中 $1 \le l \le r \le |X| - w + 1$。

在后面的描述中，我们用 $j \in WI$ 表示窗口偏移位置 j 属于窗口区间 $WI = [l, r]$，即：$j \in [l, r]$。另外，我们用 $WI.l$、$WI.r$ 和 $|WI| = r - l + 1$ 分表表示区间 WI 的左边界、右边界和大小。V_i 中窗口区间的总数记为 $n_I(V_i)$，窗口偏移位置的总数记为 $n_P(V_i)$，即：

$$n_I(V_i) = \left|\{WI \,|WI \in V_i\}\right| \tag{4.2}$$

$$n_P(V_i) = \sum_{WI \in V_i} |WI| \tag{4.3}$$

表4-1对应的是图3-1中示例序列的 KV-index 索引结构。其中，第一行表示原序列中存在三个均值落在 [1.5, 2.0) 范围内的活动窗口，分别为 $X(1000, 50)$、$X(1001, 50)$ 和 $X(1002, 50)$。在第二行中，三个窗口被组织为两个区间 [49, 50] 和 [500, 500]，所以 $n_I(V_2) = 2$ 且 $n_P(V_2) = 3$。值得注意的是，[500, 500] 是一个特殊的区间，其只包含单一的窗口偏移位置。

表 4-1　索引结构 KV-index 示例（对应图3-1）

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]

为了利于之后的查询处理，索引还包含一个源数据表，它的每行是一个四元组 $\langle K_i, pos_i, n_I(V_i), n_P(V_i)\rangle$，其中 pos_i 是索引第 i 行在文件中的偏移位置。因为元数据相对较小，我们可以在查询处理前将其全部载入内存。有了这些元数据，我们可以通过二分查找快速确定索引范围扫描的起始位置和长度。

物理上，索引结构 KV-index 有多种实现方式，受益于其简单的结构，可以存储在本地文件、HDFS 文件或 HBase 数据表中。在本研究中，我们实现了两个版本，一个本地文件版本和一个 HBase 数据表版本。

通常，文件系统或数据库会提供“扫描（Scan）”操作，即以给定的起始位置和长度进行连续范围读取，这非常有助于 KV-index 的存储并提升查询效率。

4.1.2　索引构建算法

我们分以下两个步骤构建索引。首先，我们构建一个所有行的均值范围宽度相同的索引结构。其次，因为数据分布在不同索引行之间不一定均衡，我们合并

相邻的索引行以优化索引。我们首先介绍一种简单的全内存算法，其能够处理中等大小的数据。接着我们将讨论如何将算法扩展到非常大的数据规模上。

在索引构建的第一步中，我们预先定义一个参数 d，表示均值范围的宽度。这张每行的均值范围将为 $[k\cdot d,(k+1)\cdot d)$，其中 $k\in\mathbb{Z}$。我们顺序读取原序列 X，用一个循环数组在过程中维护长度为 w 的滑动窗口 $X(i,w)$ 及其均值 μ_i^X。假设上一个窗口 S_{i-1} 的均值 μ_{i-1}^X 在 K_j 的范围内，而且当前窗口 S_i 的均值 μ_i^X 也在 K_j 的范围内，我们就修改当前的窗口区间 WI，将其右边界从 $i-1$ 改到 i。否则，就将一个新的区间 $WI=[i,i]$ 加入到 μ_i^X 对应的索引行中。

等宽的均值范围会造成索引相邻行的交错和不均匀。例如，$V_i=\{[5,5],[7,7]\}$ 且 $V_{i+1}=\{[6,6],[8,8]\}$。显然，一种更好的方式是将这两行合并，这样就可以得到 $V_i=[5,8]$，更节省空间。

在索引构建的第二步中，我们用一种贪心的算法合并索引的相邻行。算法从 $\langle K_1,V_1\rangle$ 和 $\langle K_2,V_2\rangle$ 开始依次检查相邻索引行。假设当前行为 $\langle K_i,V_i\rangle$ 和 $\langle K_{i+1},V_{i+1}\rangle$，将它们合并的条件为 $\frac{n_I(V_i\cup V_{i+1})}{n_I(V_i)+n_I(V_{i+1})}$ 是否小于一个预定义阈值 γ。这样做是因为我们希望合并那些含有大量临近窗口区间的索引行。如果索引行 $\langle K_i,V_i\rangle$ 和 $\langle K_{i+1},V_{i+1}\rangle$ 被合并了，新行的键为 $[low_i,up_{i+1})$，值为 $V_i\cup V_{i+1}$，且 V_i 和 V_{i+1} 中每一对相邻的窗口区间将被合并为一个新区间。窗口区间的合并实际上是两个区间序列间的合并操作，可以类似于归并排序高效实现。注意，索引行的合并过程是迭代进行的，因此连续的多行（≥2）也会被合并。

如果索引大小超过了内存容量，我们可以用以下方式构建索引。在第一阶段，我们将原时间序列分割为多段，并依次为每一段构建均值范围等宽的索引。在处理完所有分段后，再将不同分段的索引行进行合并。而索引构建的第二阶段是依次顺序访问索引行，所以这也可以拆分为多个子任务。因为每一阶段都可以拆分子任务，所以整个索引构建算法可以很方便地适配到 MapReduce 等分布式计算环境中。

4.1.3 索引构建复杂度分析

索引的构建过程分为两个阶段：生成等宽索引行、合并为变宽索引行。

在第一阶段，算法流式扫描所有数据，计算滑动窗口均值，并向哈希表插入键值对 $\langle\mu,offset\rangle$。注意窗口 $X(i,l)$ 的均值可以基于上一个窗口 $X(i-1,l)$ 的信息进行计算，计算复杂度仅为 $O(1)$。所以第一阶段的计算复杂度为 $O(n)$。

在第二阶段，算法依次检测相邻索引行并将认为有必要的行进行合并。因为每一行中的窗口区间是排好序的，合并操作类似归并排序，每一个窗口区间都只会被检查一次，其计算复杂度为 $n_I(V_i)+n_I(V_{i+1})$。所以，整体的计算复杂度是 $\sum_{i=1}^{D-1}n_I(V_i)+n_I(V_{i+1})$，其中 D 是第一阶段生成的索引行数。因为 $n_I(V_i)\le n_P(V_i)$

且 $\sum_{i=1}^{D} n_P(V_i) = n - w + 1$，所以可以推断第二阶段的计算复杂度为 $O(2n)$。

结合两个阶段，整体的计算复杂度为 $O(n)$。

之前基于索引的子序列匹配算法，如 FRM 和 General Match 等，它们是基于 R 树的，而 R 树构建的计算复杂度为 $O(n \cdot log_2(n))$[21]。另外，这些方法使用 DFT 去变换 X 的每个长度为 w 的窗口，而每次变换的计算复杂度为 $O(w \cdot log_2(w))$，所以一共为 $O(n \cdot w \cdot log_2(w))$。

综上所述，索引 KV-index 的构建过程是更为高效的。

4.2　基本查询算法

本节中，我们介绍相似性匹配算法 KV-match，并在算法1中给出其伪代码。

4.2.1　概述

对于给定的查询序列 Q，我们首先将其分割为长度为 w 的不相交窗口 Q_i，其中 $1 \le i \le p = \left\lfloor \frac{|Q|}{w} \right\rfloor$，并且计算窗口的均值 μ_i^Q（第1行）。我们在此假设 $|Q|$ 是 w 的整数倍。如果不是，可只保留其中是 w 整数倍的最长前缀。根据第3章中的分析，剩余部分可以直接忽略而不影响正确性。

主要的查询处理过程分为以下两个阶段：

- 索引检索（第 2-12 行）：对于查询序列的每一个窗口 Q_i，根据第 3 章中介绍的引理，从索引 KV-index 中读取相应连续的索引行。并基于这些索引行，生成一个候选子序列的集合，记为 CS；

- 候选检验（第 13-18 行）：通过读取原始数据并计算实际距离，检验候选集合 CS 中的所有子序列。

值得注意的是，所有四种支持的查询类型都遵循这样统一的查询处理过程，唯一的不同是在索引检索阶段，对于每一个窗口，不同的查询类型具有不同的行键范围，如第 3 章中所述。

4.2.2　窗口区间的生成

对于查询序列的每一个窗口 Q_i，我们计算根据查询类型计算其均值 μ_i^S 的范围 $[LR_i, UR_i]$。接着，我们以一次范围扫描的操作访问索引 KV-index，并获取一系列连续的索引行，记为 $RList_i = \left\{ \left\langle K_{s_i}, V_{s_i} \right\rangle, \left\langle K_{s_i+1}, V_{s_i+1} \right\rangle, \cdots, \left\langle K_{e_i}, V_{e_i} \right\rangle \right\}$。其应满足 $LR_i \in \left[low_{s_i}, up_{s_i} \right)$ 且 $UR_i \in \left[low_{e_i}, up_{e_i} \right)$。值得注意的是，其中第 s_i 行（或第 e_i 行）可能包含范围以外的均值。然而，这只是可能引入假阳性候选，并不会丢失任何真正的答案。

算法 1 MatchSubsequence(X, w, Q, ε)

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: $\quad RList_i \leftarrow \left\{ \left\langle K_{s_i}, V_{s_i} \right\rangle, \cdots, \left\langle K_{e_i}, V_{e_i} \right\rangle \right\}$
4: $\quad IS_i \leftarrow \varnothing$
5: $\quad$ **for all** $\left\langle K_j, V_j \right\rangle \in RList_i$ **do**
6: $\quad\quad IS_i \leftarrow IS_i \cup \{WI \mid WI \in V_j\}$
7: $\quad \text{Sort}(IS_i)$
8: $\quad CS_i \leftarrow \varnothing, shift_i \leftarrow (i-1) * w$
9: $\quad$ **for all** $WI \in IS_i$ **do**
10: $\quad\quad CS_i.\text{add}([WI.l - shift_i, WI.r - shift_i])$
11: $\quad$ **if** $i = 1$ **then** $CS = CS_i$
12: $\quad$ **else** $CS \leftarrow \text{Intersect}(CS, CS_i)$ ▷ 详见算法 2
13: $answers \leftarrow \varnothing$
14: **for all** $WI \in CS$ **do**
15: $\quad S \leftarrow X(WI.l, WI.r - WI.l + |Q|)$ ▷ 从原始数据中读取子序列
16: $\quad$ **for** $j \leftarrow 1, |S| - |Q| + 1$ **do**
17: $\quad\quad$ **if** $D(Q, S(j, |Q|)) \le \varepsilon$ **then** ▷ 对于 cNSM 查询进行附加检验
18: $\quad\quad\quad answers.\text{add}(S(j, |Q|))$
19: **return** *answers*

我们将 $RList_i$ 中的所有窗口区间记为区间集合 $IS_i = \{WI \mid WI \in V_k, k \in [s_i, e_i]\}$。用 $WI \in IS_i$ 表示窗口区间 WI 属于集合 IS_i。同样，对于 WI （$WI \in IS_i$）中的任意窗口偏移位置 j，记为 $j \in IS_i$。

根据公式 (4.2) 和公式 (4.3)，我们将 IS_i 中的窗口区间的个数记为 $n_I(IS_i)$，将 IS_i 中的窗口偏移位置个数记为 $n_P(IS_i)$。值得注意的是，IS_i 中的窗口区间都是互不相交的。为了利于之后的“相交”操作，我们将其中的窗口区间按升序排列，即：$IS_i[k].r < IS_i[k+1].l$，其中 $IS_i[k]$ 是 IS_i 中的第 k 个窗口区间（第 7 行）。

4.2.3 匹配算法

基于 $IS_i\ (1 \le i \le p)$，我们用一种“相交”的操作生成最后的候选集 CS。我们首先介绍对于查询序列 Q_i 的候选集的概念，记为 $CS_i\ (1 \le i \le p)$。

对于第一个窗口 Q_1，其窗口区间集合 IS_1 中的任意窗口偏移位置 j 映射到一条候选子序列 $X(j, |Q|)$。因此，窗口 Q_1 的候选集，记为 CS_1，是由窗口区间

集合 IS_1 中的所有偏移位置组成的。CS_1 依旧以类似 IS_1 的一系列有序且不相交的窗口区间的形式进行组织。

对于第二个窗口 Q_2，其窗口区间集合 IS_2 中的每一个窗口偏移位置同样对应一个候选子序列。然而，IS_2 中的偏移位置 j 对应的是候选子序列 $X(j-w,|Q|)$，因为 $X(j,w)$ 是其第二个不相交窗口。所以，Q_2 的候选集，记为 CS_2，可以通过将 IS_2 中的每一个窗口偏移位置向左平移 w 得到。

类似的，第三个窗口的候选集 CS_3 是通过将 IS_3 中的每一个窗口偏移位置向左平移 $2\cdot w$ 得到。

一般而言，对于窗口 Q_i $(1\le i\le p)$，其候选集 CS_i 如下所示：

$$CS_i=\{j-(i-1)\cdot w|j\in IS_i\} \tag{4.4}$$

我们将窗口 Q_i 的偏移位置平移量记为 $shift_i=(i-1)\cdot w$。所有的候选集 CS_i $(1\le i\le p)$ 依然以一系列有序不相交窗口区间的形式进行组织。另外，我们容易得出 $n_I(CS_i)=n_I(IS_i)$ 且 $n_P(CS_i)=n_P(IS_i)$。

结合第 3 章中的引理和候选集 CS_i 的定义，我们可以得到两个重要的特性：

特性 4.1 *如果 $X(j,|Q|)$ 没有被某一个候选集 CS_i $(1\le i\le p)$ 包含，那么 $X(j,|Q|)$ 与 Q 是不匹配的。*

特性 4.2 *如果 $X(j,|Q|)$ 与 Q 是匹配的，那么偏移位置 j 属于所有的候选集 CS_i 中，即 $j\in CS_i(1\le i\le p)$。*

现在我们介绍对窗口候选集 CS_i 之间计算交集并生成最终候选集 CS 的方法。方法包含 p 轮（第 2-12 行）。

在第一轮中，我们从索引获取 $RList_1$ ，并生成窗口区间集合 IS_1 和窗口候选集 CS_1。此时将全局候选集 CS 初始化为 CS_1。

在第二轮中，我们获取 $RList_2$，并通过将窗口区间集合 IS_2 中的所有窗口区间平移 $(2-1)\cdot w=w$ 的方式生成窗口候选集 CS_2 （第 9-10 行）。接着，将全局候选集 CS 与窗口候选集 CS_2 进行求交得到最新的全局候选集 CS （第 12 行）。因为 IS_i 和 CS_i 中的所有窗口区间是有序排列的，求交操作可以通过依次顺序求交 CS 和 CS_2 的中的每个窗口区间进行，这一过程非常类似归并排序。所以每一个窗口区间只会被检查一次，计算复杂度为 $O(n_I(CS)+n_I(CS_2))$。

一般而言，在第 i 轮中，我们将窗口候选集 CS_i 与上一轮留下的全局候选集 CS 进行交集计算，并生成最新的全局候选集 CS。这样在 p 轮过后，我们就得到了最终候选集 CS。

我们用图4-1中的例子进一步说明匹配算法的过程。在这个例子中，$RList_1$ 包含 WI_1、WI_2 和 WI_3 三个窗口区间。而 $RList_2$ 包含 WI_4、WI_5 和 WI_6 三个窗口区

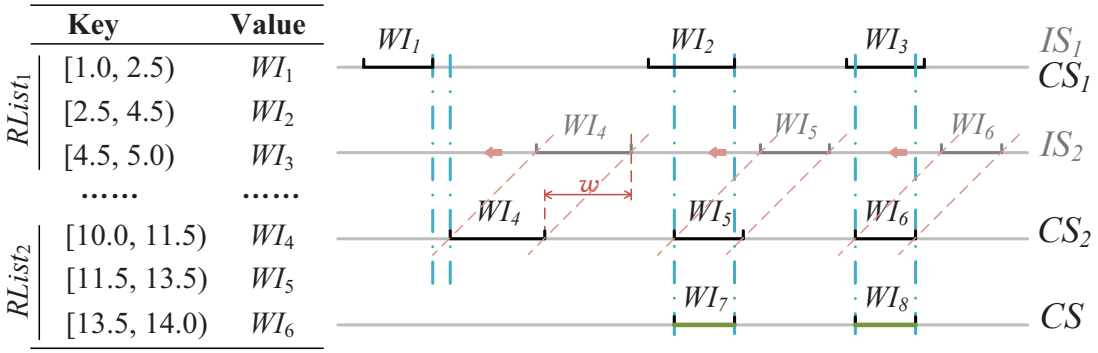


图 4-1 匹配算法示例

间。窗口区间集合 IS_1（或 IS_2）包含 $RList_1$（或 $RList_2$）覆盖的所有窗口区间。第一个窗口的候选集 CS_1 等价于 IS_1，而第二个窗口的候选集 CS_2 是通过将 IS_2 向左平移 w 得到的。之后我们在第二轮中计算 CS_1 和 CS_2 的交集以得到全局候选集 CS，其是由窗口区间 WI_7 和 WI_8 组成的。

算法 2 负责处理全局候选集 CS 和当前窗口候选集 CS_i 的交集计算。在算法中，情况 1 和情况 2 处理了类似于图 4-1中窗口区间 WI_1 和 WI_4 的情况，算法排除了没有公共部分的窗口区间（第 4-7 行）。对于情况 3 和情况 4，集合 A 和集合 B 中窗口区间的公共部分，例如图 4-1中的窗口区间 WI_7 和 WI_8，会被保留到结果候选集 CS 中，同时那些失去与后续区间相交可能的窗口区间将会被直接舍去，例如图 4-1中的窗口区间 WI_2 和 WI_6（第 9-14 行）。

4.2.4 候选检验

在第二阶段，根据最终候选集 CS，我们获取原始数据以检验并生成最终满足要求的结果（第 13-18 行）。

而言，对于候选集 CS 中的每一个窗口区间 WI，我们从原始数据中获取子序列 $X(WI.l, WI.r - WI.l + |Q|)$。值得注意的是，这条子序列包含 $|WI|$ 条长度为 $|Q|$ 的子序列。

对于每一条获取的长度为 $|Q|$ 的子序列，我们计算其与查询序列 Q 的距离，并返回符合要求的那些子序列。如果是 cNSM 查询，每一条子序列还需要在计算欧氏距离或 DTW 距离前进行标准化。

另外，UCR Suite[8] 中提出的大部分下界优化都可以应用在这里，加速候选检验过程，特别是加速 DTW 距离的计算。

4.3 本章小结

在本章中，我们首先给出了索引结构的设计，并介绍了相匹配的索引构建算法。索引结构虽然相对简单，但可以同时支持多种查询类型，并能存储到不同的

Algorithm 2 Intersect(A, B)

```
1: 将集合 B 中的窗口区间按其左端点排序
2: result ← ∅, i ← 0, j ← 0
3: while i < |A| ∩ j < |B| do
4:     if A[i].r < B[j].l then                                  ▷ 情况 1
5:         i ← i + 1
6:     else if A[i].l > B[j].r then                             ▷ 情况 2
7:         j ← j + 1
8:     else
9:         if A[i].r < B[j].r then                              ▷ 情况 3
10:            result.add({max(A[i].l, B[j].l), A[i].r})
11:            i ← i + 1
12:        else                                                 ▷ 情况 4
13:            result.add({max(A[i].l, B[j].l), B[j].r})
14:            j ← j + 1
15: return result
```

存储系统中。索引构建算法也能够很容易地扩展到分布式计算环境中去。之后介绍的子序列匹配算法，可以只通过几次连续范围读取得到候选，并且最终的候选集是各个窗口候选的交集。相比现有方法，大大提高了查询效率。

第 5 章　子序列匹配算法增强与优化

数据序列和查询序列的数据分布往往并不均匀，数据中也很可能有一些极具特征的片段，而如果我们只按基本算法那样按固定长度划分查询窗口，很多特征将埋没在窗口均值之中，无法得到有效利用。查询序列的长度也并不固定，我们需要有一些机制更好地处理不同长度的查询。另外，在基本算法中可能存在大量的查询窗口，而窗口处理的顺序又是固定的，这都不利于更高效地处理。我们在本章中针对这些问题提出了增强算法和一些优化，进一步提高了查询效率。

5.1　多层索引与增强查询算法

基本查询算法 KV-match 使用固定窗口长度 w 处理查询，忽略了查询序列的长度。这有带来了两方面的限制：首先，支持的查询长度受到限制。其次，我们不太有机会利用查询序列和原始数据的特征，以加速查询处理过程。

在本节中，我们提出算法的增强版本 KV-match$_{\mathrm{DP}}$，算法基于多层不同窗口长度的索引。规范而言，我们用两个参数 w_u 和 L 确定不同索引的构建时的窗口长度。其中，w_u 是最小窗口长度，而 L 是不同索引的数量。接着，窗口长度的集合定义为 $\Sigma = \{w_u * 2^{i-1} | 1 \leq i \leq L\}$。例如，假设 $w_u = 25$ 且 $L = 5$，我们构建窗口长度分别为 25、50、100、200 和 400 的索引。我们用 KV-index$_w$ 代表以长度为 w 的窗口构建而成的索引。

通过简单扩展第 4.1.2 节中的索引构建算法，我们可以同时构建索引集合中的各层索引。

5.1.1　动态查询分段

我们同时利用多层索引处理查询。首先将查询序列 Q 分割为一系列长度不同的不相交窗口 $\{Q_1, Q_2, \cdots, Q_p\}$，并用索引 KV-index$_{|Q_i|}$ 处理窗口 Q_i，这样可以更充分地利用数据的特征。

当查询序列 Q 分段以后，接下来的过程与基本查询算法 KV-match 是类似的。唯一的不同是，对于窗口 Q_i，我们是从索引 KV-index$_{|Q_i|}$ 获取到 $RList_i$。值得注意的是，虽然在第 3 章的引理中，查询序列 Q 是被分割为等长的窗口，但是我们可以很容易将其扩展到不等长窗口的情形，因为所有的证明都只涉及到一个窗口。

这里的挑战是如何将查询序列 Q 分段并取得最优的性能表现。我们用查询分段表示查询分割的结果。一种分段，表示为 $SG = \{r_1, r_2, \cdots, r_p\}$，意味着第一个窗口 $Q_1 = Q(1, r_1)$，第二个窗口 $Q_2 = Q(r_1 + 1, r_2 - r_1)$，依此类推。

一种高质量的查询分段应该满足以下条件：1）每一段窗口的长度属于可行长度集合 Σ；2）基于这些窗口处理查询序列 Q 能有较高的效率。

本文中，我们将分段问题作为一个优化问题，并设计一种目标函数来衡量某一种分段方式的质量，在此基础上，提出一种二维动态规划算法以解决查询分段问题。

5.1.2　目标函数

我们首先分析影响效率的关键因素。查询处理的总时间 T 由第一阶段时间 T_1 和第二阶段时间组成 T_2。如图 5-1 所示，T_2 的变化程度远大于 T_1，根据我们的理论分析和实验验证，T_2 对查询效率有更大的影响，而 T_1 则相对稳定。所以我们主要用第二阶段的效率来衡量分段的质量。而第二阶段由两部分组成：数据读取和距离计算。前者由候选集中查询窗口的数量 $n_I(CS)$ 决定，显得更为耗时。

因此，对于查询序列 Q 的一种分段 SG，在得到最终候选集 CS 以后，我们用其中窗口区间的数量 $n_I(CS)$ 来衡量分段 SG 的质量。窗口区间的数量 $n_I(CS)$ 越小，分段 SG 的质量越高。这里的挑战是，我们无法在不执行索引检索阶段的情况下，准确知道最终候选集中窗口区间数量 $n_I(CS)$。进一步，虽然我们可以通过索引元数据表获取每个窗口候选集中窗口区间的数量 $n_I(CS_i)$，但无法根据 $n_I(CS_i)$ 直接计算出 $n_I(CS)$。

为了解决这个问题，我们提出一个估计 $n_I(CS)$ 的目标函数。这里的预测基于两个假设：首先，每一个窗口的窗口区间集合 IS_i $(1 \le i \le p)$ 相互独立；其次，IS_i 中的每一个窗口区间的大小远小于数据序列长度 $|X|$。这样，我们可以将每个窗口区间看作数据序列 X 中的一个数据点，而且这些点的位置是均匀分布的。

下面，我们介绍提出的目标函数，记为 $\mathcal{F}$。假设我们用分段 SG 将查询序列 Q 分割为 $Q_1, Q_2, \cdots, Q_p$ 多段窗口，并通过索引元数据表得到每个窗口对应窗口区间的数量 IS_i $(1 \le i \le p)$。随后，我们用以下方式预测最终窗口候选集中窗口区间的数量 $n_I(CS)$。

基于之前的两点假设，我们可以用 $\frac{n_I(IS_1)}{n}$ 近似表示一个窗口区间被包含在候选集 CS_1 中的概率，其中 n 是数据序列 X 的长度。而 $\frac{n_I(IS_1)}{n} * \frac{n_I(IS_2)}{n}$ 可以表示一个窗口区间被包含在两个候选集的交集 $CS_1 \cap CS_2$ 中的概率。因此 $\prod_{i=1}^{p} \frac{n_I(IS_i)}{n}$ 可以表示一个窗口区间包含在最终候选集中的概率，其与 $n_I(CS)$ 是成比例的。所以，为了消除窗口数量的影响，我们取这个表达式的集合平均数作为最终的目

标函数 $\mathscr{F}$，即：

$$\mathscr{F}(SG) = \sqrt[p]{\prod_{i=1}^{p} \frac{n_I(IS_i)}{n}} = \frac{1}{n}\sqrt[p]{\prod_{i=1}^{p} n_I(IS_i)} \tag{5.1}$$

我们要找的查询分段是那个具有最小目标函数 $\mathscr{F}$[1]的分段。

5.1.3 二维动态规划算法

我们提出一种二维动态规划算法以找出最优的查询分段 SG。我们首先定义搜索空间。因为每个窗口 Q_i 的长度必须处于可行长度集合 Σ，所以在任意的分段 $SG = \{r_1, r_2, \cdots, r_p\}$ 中，r_i 一定是单位窗口长度 w_u 的整数倍。不符合这一点的分段都是无效的。给定一个查询序列 $Q = (q_1, q_2, \cdots, q_m)$，我们定义搜索空间为序列 $Z = (1, 2, \cdots, m')$，其中 $m' = \lfloor \frac{m}{w_u} \rfloor$。注意到序列 Z 中的值并不会影响到分段 SG 的生成。Z 的唯一作用就是约束分段 SG 的搜索空间。我们从 Z 上寻找分段 SG，而不是直接在查询序列 Q 上，我们将找出的分段记为 SG_Z。找到后再通过把 Z 中每一端点乘以 w_u 的方式，将其映射到 Q 的分段 SG 上。例如，令 $|Q| = 200$、$w_u = 25$ 且 $L = 3$，这就将有三层索引 KV-index$_{25}$、KV-index$_{50}$ 和 KV-index$_{100}$。分段 $SG_Z = \{2, 6, 7, 8\}$ 对应到 $SG = \{50, 150, 175, 200\}$。如此，查询序列 Q 将被分为 $Q(1, 50)$、$Q(51, 100)$、$Q(151, 25)$ 和 $Q(176, 25)$ 四个窗口。

我们在 Z 上用二维动态规划的方法从左到右顺序搜索最优的分段 SG_Z。第一维代表分段的边界，而第二维代表分段中包含的窗口数量。我们用 $v_{i,j}$ 表示计算过程中的子状态，其等于将 Z 的前缀 $Z(1, i)$ 分为 j 段窗口的最优分段的费用。对于任意的 j $(1 \le j \le m')$，最优分段是那个具有最小 $v_{m',j}$ 的状态。在得到所有子状态 $v_{m',j}$ 后，我们可以从中找出最优的那个作为最终的分段 SG_Z，并映射到分段 SG。动态规划递推方程如下所示：

$$v_{i,j} = \begin{cases} 1 & , i = 0 \wedge j = 0 \\ +\infty & , i = 0 \vee j = 0 \\ \min\limits_{\substack{\varphi=2^{k-1} \\ 1 \le k \le \min(L, \log_2(i)+1)}} \sqrt[j]{(v_{i-\varphi,j-1})^{j-1} * C_{i-\varphi+1,\varphi}} & , 1 \le j \le i \le m' \end{cases} \tag{5.2}$$

在公式 (5.2) 中，φ 代表在 SG_Z 上结尾为 i 的窗口的可能长度，而其最多具有 L 种可能的取值。$C_{i-\varphi+1,\varphi}$ 是从索引 KV-index$_{\varphi * w_u}$ 的源数据表中得到的不相交窗口 $Q((i-\varphi) * w_u + 1, \varphi * w_u)$ 对应窗口区间的数量 $n_I(IS)$，详见第 4.2 节。

最优的分段 SG_Z 和 SG 可以通过利用反向指针的方式进行还原。算法 3 列出了算法的完整过程。

1 因为 $\frac{1}{n}$ 是一个常数，我们可以在算法中将其忽略。

算法 3 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

另外，有时数据序列 X 可能有大量窗口都处于某一均值范围内。在这种情况下，索引 KV-index 的某些行会含有大量的窗口区间，即 n_I 较大，这将在每一轮读取 *RList* 的时候引起较大的 I/O，并在合并窗口候选集时带来计算开销。本节介绍的增强查询算法 KV-index$_{\mathrm{DP}}$ 在一定程度上可以缓解这样的问题。这是因为，目标函数会偏向于选择出具有较小 n_I 的查询窗口。

5.2　查询算法优化

本节中，我们介绍一些对于查询算法的优化方法，可以显著提高查询效率。

5.2.1　窗口数量削减

在算法 1 中，我们处理查询序列 Q 的所有 p 段不相交窗口以得到最终的候选集 CS。然而，虽然把所有的窗口候选集 $CS_i\ (1 \le i \le p)$ 一起计算交集可以尽可能地过滤最终候选集 CS 中的候选，但这并不总能取得全局的最优效率。

我们用图 5-1 加以说明，其中可以看到一个示例查询的第一阶段（索引检索）的时间 T_1、第二阶段（候选检验）的时间 T_2 以及总时间 $T\ (= T_1 + T_2)$。对于不同的查询，T、T_1 和 T_2 的值可能变化，但他们的趋势保持不变。图中的 X 轴表示第一阶段运行的轮数，记为 N_r。例如，$N_r = 10$ 意味着在索引检索阶段，虽然一共存在 40 段不相交窗口，我们只处理其中的 10 段，即 $\{Q_1, Q_2, \cdots, Q_{10}\}$，

并将第 10 段处理结束后的候选集作为进入第二阶段的最终候选集 CS。

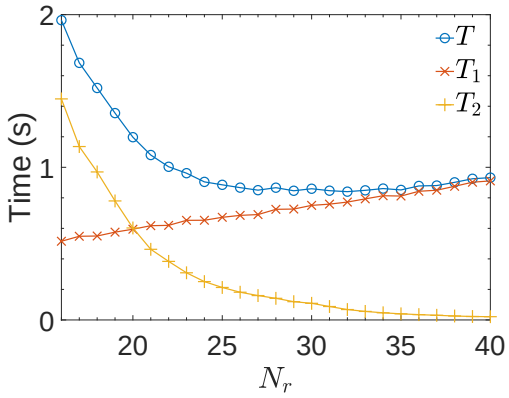


图 5-1 总时间 T、第一阶段时间 T_1 和第二阶段时间 T_2 之间的关系

一个有趣的现象是，总时间 T 并不会随着窗口数量 N_r 的增加而增加。一方面，当 N_r 小于 20 时，第二阶段时间 T_2 相对较大，这是因为候选集 CS 还包含着大量的候选，这给第二阶段带来了较高的 I/O 和计算代价。这是总时间 T 也会比较大。另一方面，当 N_r 超过 30 以后，总时间 T 变得缓慢增加。这是因为，在这种情况下第二阶段时间 T_2 保持稳定且相对较小，而第一阶段时间 T_1 因为额外的索引读取和交集计算而逐渐增加。

根据特性 4.1，仅处理一部分的窗口候选集 CS_i 以计算出最终候选集 CS 并不会丢失任何符合条件的子序列。这是因为，每一个窗口候选集 CS_i 都是真正答案的超集。所以，我们对匹配算法做如下修改：我们仍然循环处理查询窗口 Q_i。在第 i 轮中，我们获取 $RList_i$，并计算出最新的全集候选集 CS，同时计算出第一阶段到目前为止的时间 $T_1(i)$。接着我们根据当前的候选集 CS 预测出第二阶段时间 T_2，记为 $\tilde{T}_2(i)$，这是用来验证当前候选集 CS 中的所有子序列的预计时间。如果不等式 $T_1(i)+\tilde{T}_2(i) \geq T_1(i-1)+\tilde{T}_2(i-1)$ 成立，我们就停止处理剩下的查询窗口，并直接进入第二阶段。其中的挑战在于对于给定的查询如何准确预测出第二阶段时间 T_2。

本文中，我们训练了一个线性模型用来预测第二阶段时间 T_2。我们首先详细分析第二阶段，第二阶段包含两个任务：数据读取和距离计算。假设当前的候选集为 CS。数据读取的花费是与窗口区间的数量 $n_I(CS)$ 有关的。而距离计算需要计算查询序列 Q 与 $n_P(CS)$ 条候选子序列的距离，所以这部分花费是与 $n_P(CS)*|Q|$ 成正比的。因此，我们通过以下模型对第二阶段时间 T_2 进行预测：

$$\tilde{T}_2 = a*n_I(CS)+b*n_P(CS)*|Q|+c \tag{5.3}$$

在索引构建以后，我们需要训练出模型中的三个系数 a、b 和 c。为了构建训练集，我们随机生成一个查询集合，其中包含不同的查询长度和距离阈值 ε。在

每一轮中，我们同时执行索引检索和候选检验阶段。确切而言，在第 i 轮中，我们得到最新的候选集 CS，并记录其中包含的窗口区间数量 $n_I(CS)$ 和窗口偏移位置数量 $n_P(CS)$。然后，我们用候选集 CS 执行候选检验阶段，得到第二阶段时间 T_2。这样，即可向训练集中插入一条记录 $\langle n_I(CS), n_P(CS), |Q|, T_2\rangle$。等处理完所有的查询后，我们通过线性回归学习出公式 (5.3) 中的 a、b 和 c 三个系数。

5.2.2　窗口顺序调整

在基本查询算法中，我们从第一个不相交窗口 Q_1 到最后一个 Q_p 顺序处理。然而，这种顺序并不是最优的。我们以图 3-1 为例，如果我们以 Q_1、Q_2 到 Q_3 的顺序依次处理，那么计算 CS_1 和 CS_2 交集的花费是 6+4=10，因为它们的交集 $CS_1 \cap CS_2$ 的大小为 2（窗口偏移位置分别为 50 和 500），所以计算 CS_3 和 CS 交集的花费是 2+2=4。总的花费是 14。然而，如果我们将处理顺序改为 Q_3、Q_2 到 Q_1，那么总的花费将变成 (4+2)+(1+6)=13，比原先更小。

我们用启发式的方法对不相交窗口 Q_i 的顺序进行调整。基本的想法是给予包含较少窗口区间的不相交窗口 Q_i 更高的优先级。给定一个查询序列 Q，对于其中的每一个查询窗口 Q_i $(1 \le i \le p)$，我们首先根据第 3 章中的引理得出索引行键的范围 $RList_i$。因为索引的元数据表维护了索引的每一键值对 $\langle K_j, V_j \rangle$ 所包含的窗口区间的数量，所以我们可以基于元数据表计算出范围中包含的窗口区间的数量 $n_I(IS_i)$。之后，我们将所有不相交窗口根据其包含的窗口区间的数量 $n_I(IS_i)$ 递增排序。规范而言，我们将不相交窗口的顺序表示为 $(1, 2, \cdots, p)$ 的一种排列，记为 $(\psi(1), \psi(2), \cdots, \psi(p))$，其满足 $n_I(IS_{\psi(i)}) \le n_I(IS_{\psi(i+1)})$ $(1 \le i < p)$。这样，我们按从 $Q_{\psi(1)}$ 到 $Q_{\psi(p)}$ 的顺序依次处理不相交窗口。

以图 3-1 为例，因为 $IS_1 = 6$、$IS_2 = 4$ 且 $IS_3 = 2$，排列为 $\psi(1) = 3$、$\psi(2) = 2$ 和 $\psi(3) = 1$。所以我们首先处理窗口 Q_3，接着是窗口 Q_2，最后是窗口 Q_1。

在这个新的顺序下，索引检索阶段的算法与算法 1 类似，除了在第 3-6 行中将 i 替换为 $\psi(i)$，并在第 8 行中将 $shift_i$ 替换为 $(\psi(i) - 1) \cdot w$。也就是说，在第 i 轮中，我们读取 $RList_{\psi(i)}$，并以将 $IS_{\psi(i)}$ 中所有窗口区间平移 $(\psi(i) - 1) \cdot w$ 的方式生成候选集 CS_i。

这一优化让我们尽可能地提前减小候选集的大小。通过结合窗口数量削减和窗口顺序调整两个优化，我们可以将花费较高的那些不相交窗口排在最后，并跳过这些窗口，进一步加速查询处理过程。

另外，我们可以减少索引的重复访问。例如我们可以将已经访问的索引行进行缓存，这样在获取新的 $RList$ 时，如果其中部分已经在缓存中，我们只需要从索引中读取剩下的部分即可。

5.3 本章小结

在本章中，我们在基本查询算法的基础上，进行了进一步的增强和优化。针对查询序列长度不定的问题，也为了充分利用数据和查询序列数据分布的特性，我们提出了多粒度索引和动态查询分段技术，将查询序列根据统计信息切分为多段长度不等的窗口，使得总体的花费最优。另外，我们提出窗口数量削减和顺序调整等优化手段，提前处理花费较少的窗口，并在候选集趋于稳定时不再处理花费较大的窗口，这样可以进一步加速查询处理过程。

第 6 章　实验分析

我们首次提出了有限制的标准化子序列查询问题，并设计和实现了相应的索引和匹配算法。本章将在不同方面的实验中，验证我们提出方法的有效性和效率，并于现有方法进行比较。我们主要从 *RSM* 问题查询结果、*cNSM* 问题查询结果、窗口长度 w 的影响、索引大小于构建时间、可扩展性和增强算法及优化的效果等方面，对所提出的方法进行评估。

6.1　数据集和参数设置

6.1.1　真实数据集

UCR Archive[22] 是一个流行的时间序列库，其中包括许多广泛用于时间序列挖掘研究的数据集。我们讲其中的时间序列首尾相接以获得所需长度的时间序列。

6.1.2　合成数据集

我们使用合成时间序列验证所提出方法的可扩展性。合成序列是通过以下三种类型的时间序列进行生成的：

- 随机游走：序列的起点和增量步长分别在范围 $[-5,5]$ 和 $[-1,1]$ 中随机挑选；
- 高斯分布：序列的值是从一个高斯分布中选取的，高斯分布的均值和标准差分别在范围 $[-5,5]$ 和 $[0,2]$ 中随机挑选；
- 混合正弦：序列是几个正弦波的叠加混合，正弦波的周期、振幅和均值分别在范围 $[2,10]$、$[2,10]$ 和 $[-5,5]$ 中随机挑选。

为了生成时间序列 X，我们重复地执行以下步骤，直到 X 完全生成：1）随机选择序列类型 t、长度 l 和 t 类型下的相关参数；2）依据这些参数生成长度为 l 类型为 t 的一段子序列。

6.1.3　具体实现

我们实现了所提出方法的两种版本来显示其兼容性：本地磁盘版本和 HBase 数据表版本。两者都用 Java 实现的，程序代码是开源的[1]。

在本地磁盘版本中，索引存储为本地文件，其包含两个部分：索引部分和元数据部分。在索引部分中，有序索引行 $\{(K_1, V_i), (K_2, V_2), \cdots\}$ 是连续存储的。元数据部分维护了一个四元组的列表 $\langle K_i, pos_i, n_I(V_i), n_P(V_i)\rangle$ $(i = 1, 2, \cdots)$，其中 pos_i 是文件中第 i 行的偏移位置。在查询处理过程中，可以通过对元数据的二分查找得出每次扫描操作的起始索引行和结束索引行的偏移位置，再用随机读取操作读取相应索引行的数据。

为了验证 KV-match 的可扩展性，我们实现了 HBase[23] 数据表版本。其创建两种 HBase 数据表，分别存储原始数据和索引数据，两者都只有一个列族。在原始数据表中，时间序列被分割等长（默认为 1024）的不相交窗口，每个窗口存储为一行。行键是窗口的偏移位置，行值为相应的子序列。在索引数据表中，KV-index 的每一行都存储为 HBase 中的一行，K_i 作为行键，V_i 作为行值。索引的源数据表也被压缩存储为索引数据表的一行。我们使用 HBase 提供的“扫描（Scan）”API 接口直接获取 $RList$。

6.1.4　用于比较的其他方法

对于 RSM 查询，我们将所提出的方法（简称为 *KVM*）与另外两种同样基于索引的方法进行比较，分别为欧氏距离下的 General Match[5] 和 DTW 距离下与 DMatch[14]。对于 cNSM 查询，我们与 UCR Suite[8] 和 FAST[9] 进行比较。

General Match[5]（简称为 *GMatch*）是一种经典的欧氏距离下基于 R* 树的方法。我们使用了来自作者的源代码，其将索引存储在本地文件中。由于在分布式环境中构建和更新 R* 树并不简单，我们只将其与本地文件版本进行比较。

DMatch[14] 是一种反置处理的 DTW 距离下的子序列匹配方法，与其他基于树形索引的方法非常类似。由于其源代码并不开源，我们基于 General Match 的框架自己实现了它的 C++ 版本。实验中设置其窗口长度为 64，且每个窗口由 PAA 转换为 4 维空间中的点。

UCR Suite[8]（简称为 *UCR*）是可以在标准化的欧氏距离和 DTW 距离下找出最优子序列匹配的方法。它需要扫描整条时间序列的数据，并用一些下界技术来加速查询处理过程。它的代码时开源的[2]，其基于 C++ 语言并读取本地磁盘上的数据。为了使得比较更为公平，我们将其改为解决 ε-匹配问题。此外，我们基于 Java 语言实现了读取 HBase 数据表上数据的版本。我们分别对本地文件版

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

2　http://www.cs.ucr.edu/~eamonn/UCRsuite.html

本和 HBase 数据表版本进行实验，并与 KV-match 比较可扩展性。

FAST[9] 时最近提出的对 UCR Suite 的改进工作，其在原有基础上，增加了两种下界技术以进一步减少距离计算的次数。我们使用了来自作者的源代码，并将其与所提算法分别在欧氏距离和 DTW 距离下进行比较。

6.1.5 默认参数设置

在增强查询算法 KV-match$_{DP}$ 种，索引层数 L 默认设定为 5，且合法长度集合为 $\Sigma = \{25, 50, 100, 200, 400\}$。在索引构建算法中，初始均值区间定宽 d 设定为 0.5 且相邻行合并阈值 γ 设定为 80%。所有实验结果均是 100 次运行中的平均值。由于参数 d 和 γ 并不会对性能产生很大的影响，我们省略了这些参数的实验结果。

类似于其他基于树形索引的方法，我们也通过不同选择率的查询来验证性能。选择率定义为符合要求的子序列的数量与所有可能的子序列的数量 $|X| - |Q| + 1$ 的比值。我们通过从随机偏移位置截取时间序列 X 中的子序列的方式生成查询序列。

为了验证处理不同长度查询的性能，我们生成 $128, 256, \cdots, 8192$ 不同长度的查询序列。对于每一种长度，我们生成 100 条不同的查询序列。

我们通过控制距离阈值 ε 并在给定选择率范围内均匀分布选取查询序列的方式，得到所需的选择率。对于 cNSM 查询，我们通过在控制距离阈值 ε 的基础上进一步收集不同 $\langle \alpha, \beta \rangle$ 组合的查询序列。

实验是在由 8 个节点组成的集群上进行的，集群配有 HBase 1.1.5（一个 Master 节点和 7 个 RegionServer 节点），每个节点都以 Linux 作为操作系统，并具备两个 Intel Xeon E5 1.8GHz 的 CPU、64GB 的内存和 5TB 的机械硬盘存储。使用本地文件版本的实验实在集群中的一个节点上进行的。

6.2 RSM 查询的结果

我们首先将 KV-match$_{DP}$ 与 General Match 和 DMatch 进行比较。实验基于长度为 10^9 的真实数据集和不同选择率的查询序列。欧氏距离和 DTW 距离下的实验结果分别如表 6-1 和表 6-2 所示。

可以看到随着查询序列选择率的增加，General Match 的候选数量显著增加，在选择率较高的情况下，相比我们方法多出很多。虽然 General Match 将数据的每个窗口转换为一个多维数据点，这相比我们只在索引中保存均值而言多保留了原始数据的很多信息，但是它只是通过单个窗口生成候选。相反，我们的方法可以联合多个窗口的过滤能力，这样能得到更小的候选集。

General Match 对索引的访问次数大概是我们方法的 20-30 倍。得益于更少

表 6-1　欧式距离度量下 RSM 查询实验结果

算法	选择率	候选数量	索引访问次数	时间 (ms)
GMatch	$\mathbf{10^{-9}}$	13.9	279.2	852.3
	$\mathbf{10^{-8}}$	1837.5	240.1	541.2
	$\mathbf{10^{-7}}$	239,857.4	226.2	5,817.5
	$\mathbf{10^{-6}}$	1,223,370.6	338.0	30,351.7
	$\mathbf{10^{-5}}$	1,410,563.0	313.6	34,916.4
KVM-DP	$\mathbf{10^{-9}}$	2,754.9	4.6	60.4
	$\mathbf{10^{-8}}$	6,313.2	4.5	70.8
	$\mathbf{10^{-7}}$	29,853.1	4.4	138.8
	$\mathbf{10^{-6}}$	113,434.1	6.0	567.4
	$\mathbf{10^{-5}}$	153,565.1	7.0	1,200.7

的索引访问和更少的候选数量，我们的方法与 General Match 相比，总体性能有一个数量级的提升。有趣的是，对于较低选择率的查询（10^{-8} 或 10^{-9}），我们方法的候选数量虽然明显多于 General Match，但是得益于更少的索引访问，我们仍然获得了更好的总体性能。

与 General Match 类似，DMatch 也产生了大量的索引访问，并且与我们方法相比需要验证一至两个数量级的更多的候选。这背后的原因仍然与 General Match 一样，是单一窗口的候选生成机制和树形索引结构造成的。

6.3　窗口长度 w 的影响

在本实验中，我们探究了使用不同窗口长度来构建索引对过滤性能的影响。我们比较了 KV-match 和 FRM[3] [1]方法中每个窗口 Q_i 得到的候选数量。之所以选择 FRM 进行比较时因为其机制与 KV-match 较为类似。FRM 基于原始数据序列 X 的滑动窗口构建索引，并将每个窗口转换为一个 f 维空间中的点。虽有将转换后的点存入 R 树中。为了处理查询序列 Q，FRM 先将 Q 分割为 p 段不相交窗口 Q_i $(1 \le i \le p)$。对于每一段窗口，通过 R 树上的范围搜索得到一个候选集。之后，所有这些窗口候选集的并集构成最终候选集。与之相反，在我们的方法 KV-match 中，最终候选集 CS 是所有窗口候选集 CS_i 的交集。

在表 6-3中，我们列出了我们方法与 FRM 在每个窗口的平均候选数量上的比值。这个实验我们是在长度为 10^9 的时间序列上进行的，并执行了不同选择率的查询。对于每一种选择率，执行了 100 组随机选择的查询请求，并统计出它

1　FRM 是 General Match 在 $J = 1$ 时的特例。

表 6-2 DTW 距离度量下 RSM 查询实验结果

算法	选择率	候选数量	索引访问次数	时间 (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

们窗口候选数量的均值。同时，我们在不同窗口长度的索引 KV-index（50、100、200 和 400）下与 FRM 进行比较。另外，我们也列出了我们方法和 FRM 在最终候选数量上的比值。

可以清楚地看到，我们提出的方法在单个窗口 CS_i 上产生了更多的候选，特别是在窗口长度 w 较小而查询序列长度 $|Q|$ 较大的时候，这是因为下界的过滤范围与 $\frac{\epsilon}{w}$ 有关。然而，我们提出的方法在最终候选 CS 的数量上比 FRM 少了很多，这是因为在我们提出的 KV-match 中，最终候选 CS 是各个窗口候选 CS_i 的交集，而在 FRM 中，最终候选 CS 是各个窗口候选 CS_i 的并集。考虑到读取原始序列数据并计算距离是非常耗时的，减少最终候选 CS 显得更有益处。另外，对于每一个窗口 Q_i，我们只是通过一次连续范围扫描操作来访问索引，而 FRM 需要访问 R 树的多个节点，消耗更多的 I/O 资源。最后，增强查询算法 KV-index$_{DP}$ 的可以尽可能避免使用可能带来大量候选的窗口。

6.4 cNSM 查询的结果

在本实验中，我们比较了 KV-match$_{\mathrm{DP}}$ 与 UCR Suite 和 FAST 对 cNSM 查询的性能表现。实验使用本地文件版本，在长度为 10^9 的真实数据集上对不同选择率的查询进行处理。欧氏距离和 DTW 距离下的实验结果分别如表 6-4 和表 6-5 所示。对于每一组选择率，我们记录了在不同 α 和 β 情况下的运行时间。这两个限制同样嵌入到 UCR Suite 和 FAST 中，所以不符合要求的候选也会被直接丢弃。因为同一选择率分组中的运行时间相当接近，我们只列出了每种选择率下的平均时间以简化说明。

表 6-3　窗口候选平均数量与最终候选数量的比较（KV-match 与 FRM 的比值）

选择率	$\|Q\|$	窗口候选平均数量				最终候选数量			
		$w=50$	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

在 cNSM 实验中，我们使用的是相对的偏移阈值 β'，其设定为整条原始数据序列的值域的百分比。因此，引理中的 $\beta = (\max(X) - \min(X)) * \beta'\%$.

从结果中可以看出，当选择率增大时，KV-match 的运行时间增长较为平稳。在同一选择率下，运行时间随着 α 和 β 的增大而增大。因为 UCR Suite 基本总是需要扫描整个数据集，所以它的运行时间显得更加稳定并以 I/O 的消耗为主。FAST 中额外增加的几个下界似乎在欧氏距离下并没有效率的提升，可能是因为数据预处理的消耗更大。但对于 DTW 距离，FAST 相比 UCR Suite 取得了明显的性能提升，特别是对于选择率较低（10^{-8} 或 10^{-9}）的查询。在大部分情况下，我们提出的方法都相比另两种方法取得了一至两个数量级的性能提升。

表 6-4 欧氏距离度量下 cNSM 查询实验结果（单位：s）

选择率	KVM-DP				UCR	FAST
	$\alpha\backslash\beta'$	0.5	1.0	5.0	10.0	平均	平均
10^{-9}	**1.1**	0.29	0.51	2.33	4.64	59.84	86.05
	1.5	0.33	0.56	2.58	5.05		
	2.0	0.35	0.59	2.70	5.51		
10^{-8}	**1.1**	0.41	0.72	3.22	6.18	60.17	86.09
	1.5	0.56	1.00	4.60	8.98		
	2.0	0.67	1.22	5.47	10.66		
10^{-7}	**1.1**	0.85	1.30	5.46	10.29	65.25	87.79
	1.5	1.44	2.82	11.53	21.75		
	2.0	2.07	3.72	16.20	29.15		
10^{-6}	**1.1**	1.06	1.69	6.74	14.53	69.17	88.64
	1.5	1.84	3.15	15.19	27.53		
	2.0	2.52	4.39	20.77	35.75		
10^{-5}	**1.1**	0.83	1.94	7.82	12.92	70.59	89.83
	1.5	2.42	4.23	15.98	28.26		
	2.0	3.36	5.77	21.55	37.66		

6.5 索引大小与构建时间分析

在本实验中，我们比较了 KV-match$_{DP}$ 与 DMatch 在索引大小和构建时间方面的表现。因为 GMatch 具有与 DMatch 相似的索引空间占用和构建时间情况，所以我们没有将其列在结果中。实验是用不同长度的真实数据集在本地文件版本上进行的。结果如图 6-1所示，我们用深蓝色的柱形标出来原始数据序列的大小。

从结果中可以看到，DMatch 和 KV-match$_{DP}$ 的索引大小都差不多是数据大小的 10%，而 KV-match$_{DP}$ 的索引要略微大于 DMatch 的索引。然而，我们知道 KV-match$_{DP}$ 包含了 5 层 KV-index 索引，所以每一层索引要远远小于 DMatch 的索引。

我们也在 6-1中标出了两种方法的索引构建时间。我们提出方法的索引，因其简单的结构，能更高效地构建出来。在特别大的数据规模（长度为 10^{12}）下，经过 36 个小时，可以在 HBase 上为 KV-match$_{DP}$ 算法构建全部的 5 层不同窗口长度的 KV-index 索引。

另外，我们还测试了不同窗口长度 ω 对索引大小和构建时间的影响。在表 6-

表 6-5　DTW 距离度量下 cNSM 查询实验结果（单位：s）

选择率	KVM-DP				UCR	FAST
	$\alpha\backslash\beta'$	**0.5**	**1.0**	**5.0**	**10.0**	平均	平均
10^{-9}	**1.1**	0.63	0.72	2.71	3.71	139.57	77.5
	1.5	0.49	0.66	2.97	4.72		
	2.0	0.54	0.78	3.37	6.00		
10^{-8}	**1.1**	0.65	0.89	2.66	5.31	140.06	78.57
	1.5	0.88	1.24	4.89	7.89		
	2.0	0.99	1.43	5.01	9.21		
10^{-7}	**1.1**	1.58	1.88	6.61	10.02	142.99	85.07
	1.5	2.34	3.81	13.79	23.30		
	2.0	2.85	4.46	15.92	33.00		
10^{-6}	**1.1**	4.12	5.58	14.29	18.69	153.88	103.60
	1.5	7.58	11.09	30.74	60.27		
	2.0	8.02	11.40	33.72	60.56		
10^{-5}	**1.1**	15.93	19.75	36.61	49.94	177.28	137.01
	1.5	28.38	40.35	57.90	102.72		
	2.0	27.35	44.07	76.23	106.97		

6中，我们列出了对长度为 10^9 的时间序列，按固定窗口长度 w 构建的 KV-index 索引的大小和构建时间。可以看到随着窗口长度 w 的增加，索引大小和构建时间逐渐减小。这是因为更大的窗口长度 w 可以使得相邻窗口的均值更为接近，且相应使得索引行包含的窗口区间数量 $n_I(V_i)$ 变小，这会同时减少索引大小和构建时间。

6.6　可扩展性分析

为了检验我们提出方法的可扩展性，我们基于长度为 10^9 至 10^{12} 的更长的合成时间序列数据，比较 KV-match$_{DP}$ 与 UCR Suite 在 cNSM 查询类型下的性能表现。时间序列数据和我们的索引都存储在 HBase 数据表中，并分别比较了欧氏距离和 DTW 距离下的表现。实验中，我们设定 $\alpha = 1.5$ 且 $\beta' = 1.0$，并通过调整距离阈值 ε 控制选择率维持在 10^{-7}。实验结果如图 6-2所示。

可以看到 KV-match$_{DP}$ 在欧氏距离和 DTW 距离下都比 UCR Suite 快差不多两至三个数量级。对于长度为 1 万亿（10^{12}）的时间序列，我们可以平均在 127 秒内处理欧式距离下的查询，在 243 秒内处理 DTW 距离下的查询，这体现出了

表 6-6 窗口长度 w 对索引大小和构建时间的影响

窗口长度 w	索引大小 (MB)	索引构建时间 (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

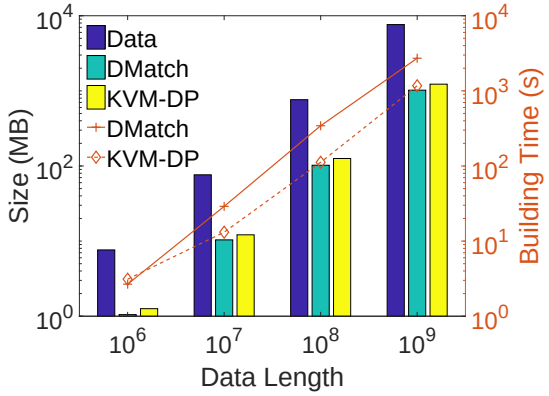


图 6-1 索引大小与构建时间

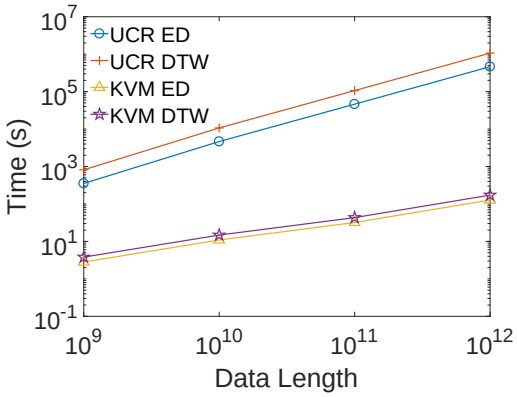


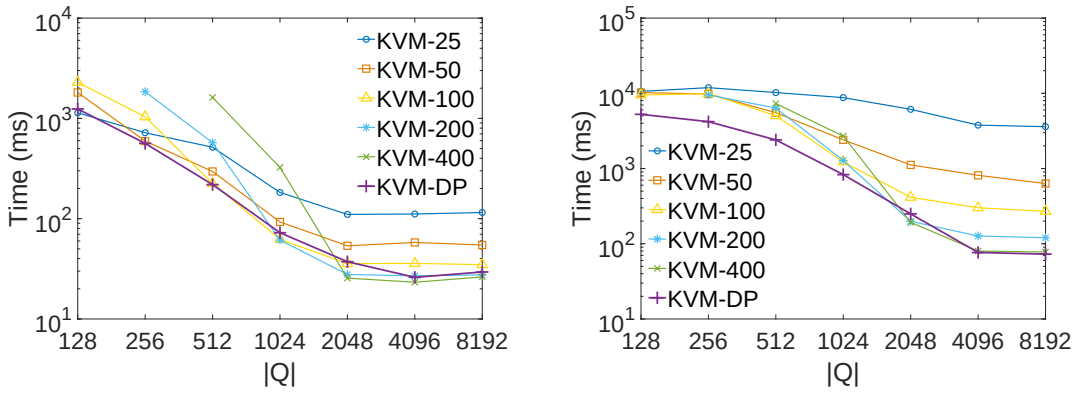
图 6-2 可扩展性

很好的可扩展性。

6.7 增强算法与基本算法的比较

在本实验中，我们比较了增强查询算法 KV-match$_{DP}$ 与基本查询算法 KV-match 对 RSM 查询的运行时间。我们分别构建了 5 层 KV-index 索引，窗口长度 w 分别为 25、50、100、200 和 400。对于 KV-match$_{DP}$ 算法，我们设定合法长度集合 $\Sigma = \{25, 50, 100, 200, 400\}$，以使用所有索引。而 KV-match 算法则分别使用不同窗口长度的索引进行实验。实验在用本地文件版本在长度为 10^9 的真实数据集上进行的。因为单层索引的查询性能与查询序列的长度高度相关，我们测试了不同查询长度下的运行时间。图 6-3 (a) 和 (b) 分别对应距离阈值 $\varepsilon = 10$（代表较低选择率）和距离阈值 $\varepsilon = 100$（代表较高选择率）的实验结果。

可以看出，在大部分情况下，增强查询算法 KV-match$_{DP}$ 比所有使用单层索引的基本查询方法表现得好。从另一方面看，使用窗口长度较短的索引更适合于长度较短的查询，而使用窗口长度较长的索引更适合于长度较长的查询。实验结果验证了我们提出的查询分段算法的有效性。增强查询算法 KV-match$_{DP}$ 可以发挥出多种窗口长度的过滤能力，并充分利用了数据和查询序列的特征。



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

图 6-3　动态窗口分段的效果

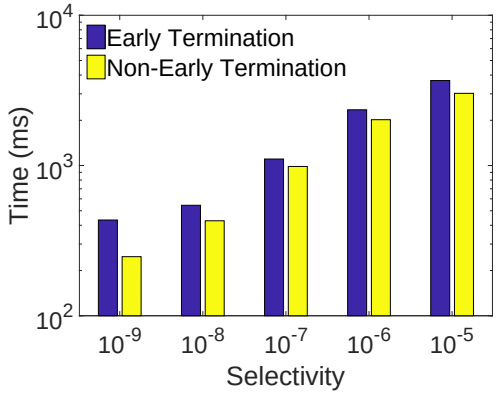


图 6-4　窗口数量削减

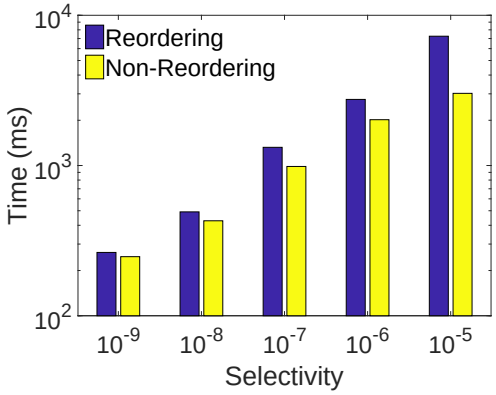


图 6-5　窗口顺序调整

6.8　窗口数量削减与顺序调整的效果

在本实验中，我们验证了窗口数量削减和窗口顺序调整两个优化技术的有效性。实验是用本地文件版本在长度为 10^9 的真实数据集上进行的。因结果较为类似，我们只列出了对欧氏距离下的 RSM 查询的实验结果。

使用和不使用窗口数量削减技术的查询处理时间对比如图 6-4所示。可以看到窗口数量削减技术可以优化查询性能大约 20%，这验证了我们在第 5.2.1 节中提出的基于学习的方法。

使用和不使用窗口顺序调整技术的查询处理时间对比如图 6-5所示。可以看到窗口顺序调整技术可以显著提升查询性能，特别是在选择率较高的情况下。

6.9　本章小结

在本章中，我们对所提出方法进行了大量实验，并于现有算法进行了比较和分析。通过实验结果，可以看到我们提出的方法具有良好的性能，对于 RSM 查询和 cNSM 查询都能高效地给出结果，与现有算法相比，得益于索引的连续范

围读取和窗口候选的交集机制，索引访问次数大大减少，最终需要检验的候选数量也明显更少，总体性能更好。文中提出的各项优化和增强算法也起到了应有的作用。

第 7 章　总结与展望

7.1　总结

时间序列子序列相似性查询在时间序列数据挖掘领域具有重要研究意义和应用价值。一部分现有算法可以解决非标准化的子序列匹配问题，但大多基于 R 树等树型索引，且最终候选集是窗口候选的并集，带来大量索引随机访问和较高的候选验证花费。另有一部分现有算法可以解决标准化的子序列匹配问题，但需要完整扫描原始数据序列，无法建立索引提高查询效率。

我们通过分析，提出了一种新型的子序列匹配问题，在标准化子序列匹配问题的基础上，增加了两个约束。添加的约束很大程度上并不会影响标准化子序列匹配的效果，因为在大部分应用场景下，想要找出的匹配序列与查询序列具有一定的均值和标准差范围，这才能对应于一致的实际状态。这些约束在不影响查询效果的同时，提供了我们控制标准化这一存在单点对全局改变可能范围无限大问题的可能，也使得我们能够构建索引解决标准化子序列匹配问题。

我们进一步提出了一种统一的候选过滤条件的形式，无论是欧氏距离、DTW 距离，还是 RSM 查询、cNSM 查询都能找到符合这一形式的过滤条件。这使得我们可以只构建一个索引，就能同时支持多种查询类型。这完全符合日常数据分析人员探索式渐进式的查询分析过程，极大方便了查询意图的迭代式细化。

在此基础上提出的索引结构和子序列匹配算法能够通过少有的几次连续范围读取，得到窗口候选集。并将这些窗口候选集的交集作为最终候选。这与现有算法相比，明显减少了索引访问次数和读取延迟，同时减少了候选验证环节的花费，具有很大性能优势。

进一步提出的增强算法和优化能更充分地利用数据和查询序列的特性，根据索引元素据，动态分割查询窗口，并根据预估花费排序，优先处理花费较小的窗口。同时，当候选集趋于稳定时，放弃后续窗口的候选排除，直接进入候选检验环节，取得总体查询效率的更优化。

通过大量实验分析和比较，可以看到我们提出的新方法相比现有方法在索引大小、构建时间、查询效率、索引访问、可扩展性等方面均有提高。部分情况下更有多个数量级的提升。本文中提出的各项查询优化和增强算法也都能起到预期的作用，为总体查询效率的进一步提升做出了贡献。

7.2　展望

展望未来，我们还可以在一些方面对算法进行改进，比如：自动根据数据分布确定索引行的均值范围宽度、更智能的索引构建第二阶段多行合并策略、共享多层不同粒度的索引间存在的冗余数据、渐进式不断细化索引以减少初始等待时间、支持不等长的 DTW 距离匹配等。

另一方面，我们可以将本文提出的问题和方法集成到开源时间序列数据管理系统中，在实践中进一步验证和完善时间序列子序列相似性查询解决方案。

参考文献

[1] RAKTHANMANON T, KEOGH E. Fast shapelets: A scalable algorithm for discovering time series shapelets[C]//ICDM. Piscataway: IEEE, 2013: 668-676.

[2] YEH C C M, ZHU Y, ULANOVA L, et al. Time series joins, motifs, discords and shapelets: A unifying view that exploits the matrix profile[J]. DMKD, 2018, 32 (1):83-123.

[3] FALOUTSOS C, RANGANATHAN M, MANOLOPOULOS Y. Fast subsequence matching in time-series databases[C]//SIGMOD '94: Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM, 1994: 419-429.

[4] ZHU H, KOLLIOS G, ATHITSOS V. A generic framework for efficient and effective subsequence retrieval[C]//volume 5. [S.l.]: VLDB Endowment, 2012: 1579-1590.

[5] MOON Y S, WHANG K Y, HAN W S. General match: A subsequence matching method in time-series databases based on generalized windows[C]//SIGMOD '02: Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM, 2002: 382-393.

[6] PAPAPETROU P, ATHITSOS V, POTAMIAS M, et al. Embedding-based subsequence matching in time-series databases[J]. TODS, 2011, 36(3):17:1-17:39.

[7] HAN W S, LEE J, MOON Y S, et al. Ranked subsequence matching in time-series databases[C]//VLDB '07: Proceedings of the 33rd International Conference on Very Large Data Bases. [S.l.]: VLDB Endowment, 2007: 423-434.

[8] RAKTHANMANON T, CAMPANA B, MUEEN A, et al. Searching and mining trillions of time series subsequences under dynamic time warping[C]//KDD '12: Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York, NY, USA: ACM, 2012: 262-270.

[9] LI Y, TANG B, U L H, et al. Fast subsequence search on time series data[C]// Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017. [S.l.]: OpenProceedings.org, 2017: 514-517.

[10] BRANLARD E. Wind energy: On the statistics of gusts and their propagation through a wind farm[J]. ECN-Wind-Memo-09, 2009, 5.

[11] SAKOE H, CHIBA S. Dynamic programming algorithm optimization for spoken word recognition[J]. TSP, 1978, 26(1):43-49.

[12] MOON Y S, WHANG K Y, LOH W K. Duality-based subsequence matching in time-series databases[C]//Proceedings of the 17th International Conference on Data Engineering. Washington, DC, USA: IEEE Computer Society, 2001: 263-272.

[13] KEOGH E, RATANAMAHATANA C A. Exact indexing of dynamic time warping[J]. KIS, 2005, 7(3):358-386.

[14] FU A W C, KEOGH E, LAU L Y H, et al. Scaling and time warping in time series querying[J]. The VLDB Journal, 2008, 17(4):899-921.

[15] LIM S H, PARK H J, KIM S W. Using multiple indexes for efficient subsequence matching in time-series databases[C]//Database Systems for Advanced Applications. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006: 65-79.

[16] NEAMTU R, AHSAN R, RUNDENSTEINER E, et al. Generalized dynamic time warping: Unleashing the warping power hidden in point-wise distances[C]//ICDE. Piscataway: IEEE, 2018.

[17] ZOUMPATIANOS K, IDREOS S, PALPANAS T. Indexing for interactive exploration of big data series[C]//SIGMOD '14: Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM, 2014: 1555-1566.

[18] NEAMTU R, AHSAN R, RUNDENSTEINER E, et al. Interactive time series exploration powered by the marriage of similarity distances[J]. Proc. VLDB Endow., 2016, 10(3):169-180.

[19] YI B K, FALOUTSOS C. Fast time sequence indexing for arbitrary lp norms [C]//VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000: 385-394.

[20] ZHU Y, SHASHA D. Warping indexes with envelope transforms for query by humming[C]//SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data. New York, NY, USA: ACM, 2003: 181-192.

[21] ALBORZI H, SAMET H. Execution time analysis of a top-down r-tree construction algorithm[J]. Inf. Process. Lett., 2007, 101:6-12.

[22] CHEN Y, KEOGH E, HU B, et al. The ucr time series classification archive [EB/OL]. 2015. www.cs.ucr.edu/~eamonn/time_series_data/.

[23] Apache HBase[EB/OL]. 2019. http://hbase.apache.org.

攻读硕士学位期间的研究成果

[1] ICDE 2019 （CCF-A 类会议长文，第一作者）

[2] ICDM 2018 （CCF-B 类会议短文，第一作者）

[3] DEXA 2018 （CCF-C 类会议长文，第三作者）

[4] DASFAA 2017 （CCF-B 类会议长文，第六作者)