arXiv:2304.08264v1 [cs.DB] 17 Apr 2023

Dumpy：面向大型数据序列集合的紧凑自适应索引

ZEYU WANG，复旦大学计算机科学学院，中国QITONG WANG，巴黎西岱大学 LIPADE，法国PENG WANG*，复旦大学计算机科学学院、上海市数据科学重点实验室，中国THEMIS PALPANAS，巴黎西岱大学 LIPADE 与 IUF，法国WEI WANG，复旦大学计算机科学学院、上海市数据科学重点实验室，中国

数据序列索引对于管理和分析如今日益增长、随处可见的数据序列集合至关重要。这些索引支持精确和近似相似性搜索；近似搜索能够在毫秒内提供高质量结果，因此对某些现代应用极具吸引力。缩短预处理（即索引构建）时间并提高搜索结果的准确率，是两项主要挑战。DSTree 和 iSAX 索引系列是解决这一问题的最先进方案。然而，DSTree 的索引构建时间较长，而 iSAX 的搜索准确率较低。本文指出了 iSAX 索引系列中两个会对整体性能产生不利影响的问题。首先，我们发现，索引结构设计（即节点扇出度）存在邻近性与紧凑性权衡，这显著限制了所得索引的效率和准确率。其次，偏斜的数据分布会对 iSAX 的性能产生负面影响。为解决这些问题，我们提出了 Dumpy，这是一种采用新型多叉数据结构、自适应节点分裂算法和高效构建工作流的索引。此外，我们还设计了 Dumpy-Fuzzy 作为 Dumpy 的变体，通过适当复制序列进一步提高搜索准确率。在多个大型真实数据集上进行的实验表明，与竞争方案相比，Dumpy 系列方案在效率、可扩展性和搜索准确率方面均显著更优。本文发表于 SIGMOD'23。

CCS 概念：• **信息系统** → **数据管理系统**；**时序数据**；**数据库查询处理**。

补充关键词与短语：数据序列索引、相似性搜索

ACM 引用格式：
Zeyu Wang、Qitong Wang、PENG WANG、Themis Palpanas 和 WEI WANG。2023。Dumpy：面向大型数据序列集合的紧凑自适应索引。*ACM*数据管理会议录1，1，第111号文章（2023年5月），共27页。
https://doi.org/10.1145/3588965

*通讯作者

作者地址：Zeyu Wang，复旦大学计算机科学学院，中国上海，zeyuwang21@m.fudan. edu.cn；Qitong Wang，巴黎西岱大学 LIPADE，法国巴黎，qitong.wang@etu.u-paris.fr；PENG WANG，上海市数据科学重点实验室、复旦大学计算机科学学院，中国上海，pengwang5@fudan.edu.cn；Themis Palpanas，巴黎西岱大学 LIPADE 与 IUF，法国巴黎，themis@mi.parisdescartes.fr；WEI WANG，上海市数据科学重点实验室、复旦大学计算机科学学院，中国上海，weiwang1@fudan.edu.cn。

允许免费制作本作品的全部或部分内容的数字或纸质副本，供个人或课堂使用，但不得为盈利或商业利益而制作或分发副本，且副本须附带本声明及首页上的完整引用。必须尊重不属于作者的本作品组件的版权。注明出处后允许摘录。若要以其他方式复制、重新出版、发布到服务器或分发至列表，须事先获得明确许可和／或支付费用。请联系 permissions@acm.org 申请许可。© 2023 版权归所有者／作者所有。出版权授权予 ACM。
2836-6573/2023/5-ART111 $15.00 https://doi.org/10.1145/3588965

1 引言

如今，几乎所有科学和社会领域的应用都在产生海量数据序列集合 [31, 43, 74], , 使数据序列成为最常见的数据类型之一。过去三十年来，如何管理和分析大容量数据序列的问题一直受到数据管理领域的研究关注 [8, 44]。在此背景下，相似性搜索是一项基础操作，也是多种高层算法的核心，例如分类、聚类、模式发现和离群点检测 [10, 11, 43, 46, 59, 64]。

相似性搜索旨在给定一个查询序列和一种距离度量后，在数据集中寻找最近邻。最直接的解决方案是依次计算所有序列与查询序列之间的距离。然而，随着数据集大小增加，顺序扫描很快就会变得难以执行。为支持大规模相似性搜索，可以使用数据序列索引剪除无关数据，从而减少回答查询所需的工作量。此外，随着研究人员越来越关注数据探索，近似相似性搜索的重要性迅速提升 [21, 22]。它广泛应用于实际场景，因为它能在交互式响应时间内（以毫秒计）提供高质量的近似查询结果 [5, 7, 15, 37, 62]。在这类应用中，近似查询结果的质量足以支持下游应用 [16]。近期的例子包括：1）一种k近邻（kNN）分类器 [3], , 当kNN平均精度均值（MAP）达到60%时，其准确率趋近于最佳水平；2）一种离群点检测器 [54] , MAP达到50%时即可取得最佳ROC曲线下面积；以及3）一种用于大规模分类的基于kNN的SoftMax近似技术，当kNN召回率达到80%时，其准确率几乎与精确SoftMax相同 [69]。对于这些应用，kNN索引的核心要求包括：在上述精度要求下的查询时间（应为毫秒级）、索引构建时间，以及支持大型数据集的可扩展性。

尽管文献中有数十种可为数据序列 [22], 建立索引的方法，但只有少数方法能够稳健地支持大型数据序列集合，例如超过 100GB 的集合（因此，人们研究了近似搜索 [22], 以及用于精确 [20] 和近似 [29] 查询应答的渐进式搜索技术）。其中，DSTree [65] 和 iSAX 索引系列 [45] 在近似搜索方面展现出最佳查询性能，同时也支持精确搜索。由于采用动态分段技术，DSTree 的索引构建时间较长（比 iSAX 慢一个数量级以上），且难以优化。相反，得益于快速的索引构建和丰富的优化 [13, 14, 49, 50, 52, 67, 70],, iSAX 索引系列在过去十年中已成为最流行的数据序列索引。尽管如此，当访问少量数据（一或数个节点，从而确保毫秒级延迟）时，iSAX 的近似搜索准确率仍不理想。例如，访问一个节点时，其 MAP 不到 10%；而若要将 MAP 提高到 ≥50% [22]，查询时间则会超过一秒。在这项工作中，我们找出了 iSAX 索引系列的索引结构和构建工作流中固有的问题，并提出了全新的解决方案 Dumpy 来解决这些问题。

首先，我们发现，索引结构的设计是一个固有却被忽视的问题，它显著限制了 iSAX 索引系列的性能。尽管 iSAX [56] 原则上并未限制节点的扇出，但流行的 iSAX 系列索引 [12, 14, 38, 48, 49, 52, 67, 70]仍采用二叉结构（第一层具有完整扇出，除此之外均如此）。当节点包含的序列数量超过叶节点大小阈值 th时，系统会选取一个 SAX 段，并将该节点分裂为两个子节点。

然而，在这种二叉结构下，所采用的划分策略 [12, 56] 往往会导致次优决策（参见 [68], 第 4 节），从而损害节点内序列的邻近性（即相似度），最终影响近似查询结果的质量。

最近，有人提出了一种基于 SAX 的全多叉索引来解决这一问题 [68]。全多叉结构会在所有分段上分裂一个完整节点，因此避免了聚焦于单个分段而导致次优分裂决策的问题。然而，每次分裂最多会生成 2^w 个节点，其中 w 是分段总数，导致叶节点数量过多，进而使叶节点填充因子极低。这会由于磁盘利用率低下和磁盘访问次数过多，导致（驻留于磁盘的）索引性能不佳。尽管可以将索引中的子树合并为更大的分区（例如，128MB）[68] 以减少随机 I/O，但存储和加载其庞大的内部索引结构仍会产生大量开销，同时也会带来诸多应用限制。

我们将上述问题称为邻近性与紧凑性权衡。邻近性和紧凑性都有助于相似性搜索：邻近性使访问到的序列更接近查询，而紧凑性则使访问节点时能获得更多候选序列。

二进制索引结构旨在提供紧凑的子节点，但会降低查询结果的准确率；而全多叉结构通过分裂节点来保留节点内序列的邻近性，却无法提供填充因子较高（即紧凑性较高）的叶节点。因此，这两种结构都未能充分利用邻近性与紧凑性权衡，限制了其搜索准确率和构建效率。

在这项工作中，我们突破了 iSAX 系列索引采用单一固定扇出的局限，并提出一种能够构建多叉索引结构的自适应分裂策略。具体而言，我们设计了一种新颖的目标函数，从邻近性和紧凑性两个方面评估候选分裂方案的质量。我们使用所选分段上数据的平均方差来衡量节点内序列的邻近性，并使用子节点填充因子的方差来衡量紧凑性。此外，我们提出了一种由三种加速技术构成的高效搜索算法，根据质量评估结果寻找最优划分方案。

除了索引结构设计之外，我们还发现了 iSAX 索引系列存在另外两个问题，阻碍了对这种新型自适应多叉索引结构的充分利用。第一个观察是，当扇出较大时（例如二叉结构中的第一层以及全多叉结构中的所有层），数据序列通常以高度不均衡的方式分布在各个子节点中；即使选择最佳划分方案，也无法完全避免这种情况。也就是说，大多数数据序列集中在少数几个节点中，而大多数节点的规模都很小。这通常会导致大量小节点，从而损害最终索引的性能。另一个问题是，常见的 iSAX 索引构建流程仅依据极少部分数据的分布来分裂节点，实际上使得针对整体数据的分裂决策并非最优。例如，iSAX2+ 尝试根据第一个$th+1$ 序列在分裂时平衡两个子节点（即节点填满后才进行分裂），但正如我们的实验所验证的，最终平均填充因子通常低于 20%。

为避免这两个问题，我们设计了一种灵活高效的索引构建工作流，并提出了一种叶节点打包算法。得益于 iSAX 的静态分段，我们的工作流无需任何额外开销即可收集全局 SAX 单词表，并使自适应分裂策略更好地适配整个数据集。此外，我们的叶节点打包算法能够在不损失剪枝能力的前提下打包较小的兄弟叶节点，从而减少索引构建和查询过程中的随机磁盘访问。

总而言之，通过将自适应分裂策略与全新的索引构建流程相结合，我们提出了数据序列索引解决方案 Dumpy（因其结构简短紧凑而得名）。在索引构建效率、近似搜索准确率和精确搜索性能方面，Dumpy 推进了最先进水平，使其成为适用于大规模数据序列管理和分析应用的功能完备且实用的解决方案。

此外，作为一种基于空间划分的方法，Dumpy 通常也存在一个常见的边界问题 [17, 24]。也就是说，查询的 k近邻可能位于相邻节点或子树中，且靠近划分边界。由于我们只搜索一个或几个节点，这些真实近邻可能会被遗漏。为减轻这一影响，我们提出了 Dumpy 的变体 Dumpy-Fuzzy：将硬 划分边界转变为模糊范围，并在每次分裂时采用复制策略，进一步提高搜索准确率，代价是在索引构建和存储方面产生少量开销。

我们的贡献可概括如下。

(1) 我们发现了当前最先进的 iSAX 索引系列结构设计中固有的邻近性与紧凑性权衡，并证明这一权衡限制了近似查询结果的质量以及索引构建效率。

(2) 我们提出了 Dumpy，这是一种新型多叉数据序列索引，它能够通过在线自适应、智能地确定分裂策略，在邻近性与紧凑性权衡之间取得恰当平衡。

(3) 我们为 iSAX 索引系列设计了一套强大而高效的索引构建工作流，并提出了一种新颖的叶节点打包算法，以处理数据倾斜并实现稳定的性能。

(4) 我们设计了 Dumpy-Fuzzy，通过适当的数据复制进一步提高搜索准确率。

(5) 我们在多种合成数据集和真实数据集上进行的实验评估表明，与最先进方法竞争者相比，Dumpy 及其变体的索引构建时间始终更快（平均快 4 倍），近似查询准确率也更高（平均 MAP 高 65%），而查询应答时间仅为毫秒级。

2 相关工作

[数据序列索引] 已有数十种方法被提出，用于为海量数据序列集合建立索引 [21, 22]。其中，基于SAX的索引 [45] 广受欢迎，并取得了最先进的性能。在最初的 iSAX [56] 索引之后，iSAX2.0 和 iSAX2+ [12, 13] 通过新颖的批量加载和节点拆分策略加快了索引构建；ADS [70] 优化了索引构建与查询应答的综合耗时；ULISSE [39] 支持子序列相似性搜索；SEAnet [63]利用深度学习嵌入提升了高频时间序列的查询结果质量；而 DPiSAX [66], Odyssey [14], PARIS [50], MESSI [51], SING [52], 和 Hercules [19] 则利用数据分布和现代硬件的并行能力。这些索引都沿用了 iSAX 原有的二叉结构，限制了节点内序列邻近性。

ADS [70], 作为一种查询自适应索引，只构建并物化被已检查查询访问过的叶节点。然而，当查询工作负载极大且会访问索引的所有叶节点时，ADS 就会与 aniSAX 索引相同，具有相同的查询应答特性。相反，作为一种数据自适应索引，Dumpy 会根据数据集合而非查询来调整其结构。因此，其性能不受工作负载影响。（实验中省略 ADS，因为它不如 iSAX2.0 和 DSTree [21]。）

TARDIS [68] 最先注意到二叉结构的缺点，并提出全多叉结构以及基于大小的分区策略，以合并不同的子树，供分布式集群使用。然而，TARDIS 仅适用于分析静态数据集，而且庞大的结构降低了构建和查询效率。我们在实验中实现了 TARDIS 的独立版本。Coconut [33, 34] 使用 InvSAX 表示对数据集排序后构建 B+-树，并在批量加载时凭借顺序 I/O 获得了显著的性能提升。然而，后续插入操作会破坏磁盘上的顺序布局；此外，基于扫描的精确搜索算法要求将完整的 InvSAX 表保留在内存中，并且原始数据集必须在位。看起来也没有简单的方法能在 Coconut 中恢复经典的基于树的剪枝。因此，我们的实验未纳入 Coconut。

DSTree [65] 通过采用高度自适应的 EAPCA 摘要，并动态增加分段数量，实现了出色的搜索准确率。但其副作用是，DSTree 无法跳过针对原始数据序列的高开销分裂操作。因此，批量加载算法以及我们提到的许多其他优化难以应用于 DSTree。正如实验结果所示，即使采用静态摘要 iSAX，Dumpy 提供的查询结果质量也高于 DSTree，且构建速度快得多。

[高维向量索引] 根据近期研究 [21, 22],，数据序列和高维向量的相似性搜索算法可以互换使用。高维向量搜索的代表性算法包括基于邻近图的方法 [41, 42, 62],，这类方法在小型数据集上查询性能出色，但构建和存储图需要耗费大量时间和内存。目前，在普通机器上，这些方法难以扩展到十亿级数据集 [23, 24, 57]。在前沿研究中，与数据序列索引相比，乘积量化系列方法 [6, 25, 28, 47]能够在分钟级近似精确搜索中取得更好的查询性能。然而，其构建时间仍比 DSTree 慢一个数量级以上 [22]。LSH（局部敏感哈希）系列方法 [26, 27, 36, 40, 58], 虽然提供概率保证，但在时间和空间开销方面已被证明不如数据序列索引 [22]。

3 背景

我们首先给出本文其余部分所需的一些定义，然后介绍 iSAX 摘要和索引。

定义1（数据序列）。数据序列 $s = \{p_1, p_2, \ldots, p_l\}$ 是由点组成的序列 $p_i = (x_i, t_i)$，其中每个点都与一个值 x_i和一个位置 t_i相关联，并满足 $t_1 < t_2 < \cdots < t_l$。 l表示数据序列 s的长度。

本文假设一个数据序列数据库 db 包含大量长度相同n的数据序列。我们用 kNN（k近邻）查询表示一种指定了最近邻数量的特定相似性搜索查询。

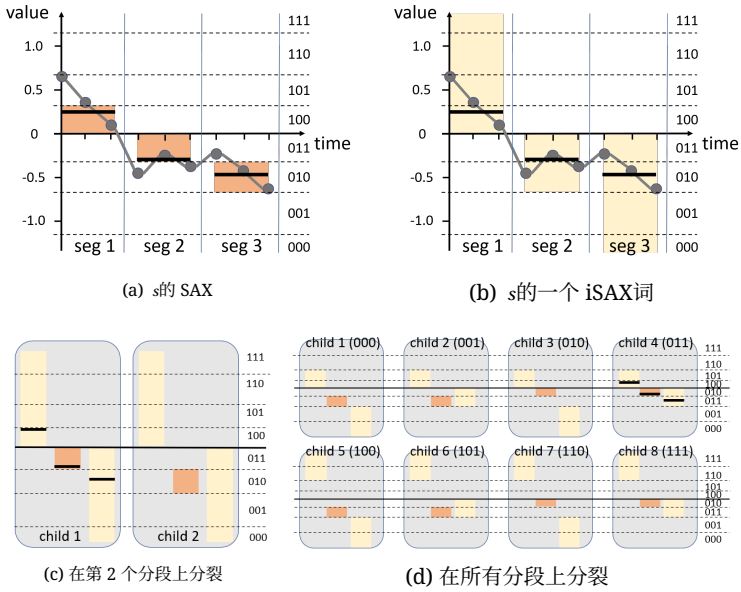
定义2（k近邻查询）。给定一个整数 k、一个查询数据序列 q和一个距离度量 $dist$，***kNN*** 查询从数据库中检索序列集合 $R = \{r_1, r_2, \ldots, r_k\}$，使得对于数据库中的任意其他序列 s以及任意 $r_i \in R$, $dist(r_i, q) \leq dist(s, q)$。

距离度量的选择取决于具体应用。不过，欧几里得距离（ED）是大型数据序列集合中最受欢迎、研究最广泛且效果出色的相似度度量之一 [18]。与此同时，我们还支持动态时间规整（DTW） [53]，这是 iSAX 索引系列 [49]的一项固有特性。除精确 k近邻查询外，通过检查整个数据库的一个小型子集来加速查询处理的近似kNN查询，也引起了研究人员的广泛关注。近似查询结果$A = \{a_1、\ldots\ldots、 a_k\}$预期应接近真实结果 R。

[iSAX 摘要] 本文采用 iSAX 摘要技术 [56]构建 Dumpy。iSAX 是一种 SAX词的动态前缀，而 SAX 是分段聚合近似（PAA）的一种符号化表示 [32]。我们通过图 1 中的示例简要回顾这些技术。

PAA(s,w) 将数据序列 s划分为 w个互不重叠且长度相等的分段，并用每个分段的均值表示该分段。因此，PAA 将 s 降维概括。如图 1(a)中的黑色实线所示，PAA(s,3)=[0.28,-0.31,-0.49]。

SAX(s,w,c) 是对 PAA 的表示，由取自基数为 c 的字母表的 w 个离散符号构成。SAX 的主要思想是，实值空间可由 $c-1$ 个断点（满足 $N(0,1)$）划分为 c 个区域，并用不同符号标记。例如，当可用符号（以位编码表示）为 {00,01,10,11} 时，SAX 会为每个分段上的 PAA



(a) s的 SAX　(b) s的一个 iSAX词

(c) 在第 2 个分段上分裂　(d) 在所有分段上分裂

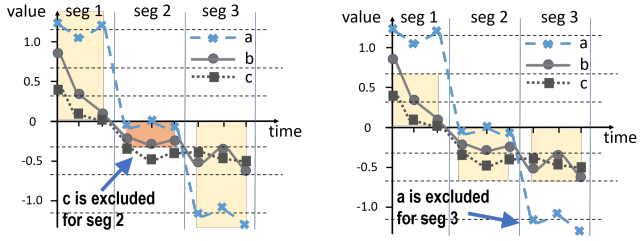
图 1. (a) 和 (b) 分别为 PAA、SAX 和 iSAX 表示（$w = 3, b = 3$）。(c) 和 (d) 分别展示了 iSAX 索引系列在二叉和全多叉结构中的节点拆分。

系数分配符号。在图 1(a) 中，SAX(s,3,8)=[100,011,010]。SAX词表示由各分段中的值范围构成的区域，以橙色背景绘出。iSAX(s,w,c) 在每个分段中采用可变基数（$\leq c$）。也就是说，iSAX词是对应 SAX词的前缀。图 1(b) 中的 iSAX词为 iSAX(s,3,8)=[1,01,0][1]。由于字母表的基数降低，iSAX词表示的范围比对应的 SAX词更大（粒度更粗）。

[iSAX 索引系列] iSAX 索引系列 [45] 使用树结构来组织数据序列，其中包含三种类型的节点。表示整个值空间的根节点通过对所有 w 分段进行分裂，最多指向 2^w 个子节点。每个内部节点都包含其中所有序列共有的 iSAX 词，以及指向其子节点的指针。每个叶节点除了存储共有的 iSAX 词外，还存储其中每个序列的原始数据和 SAX词。当叶节点的大小超过叶节点大小阈值 th时，该叶节点会转换为内部节点，其中的所有序列也会相应地进行分裂。iSAX 索引系列有两种分裂策略。一种是二分裂（见图 1(c)），通过将一个分段上的 iSAX 符号基数加倍来拆分节点。这两个细化后的 iSAX 词表示该分段上更小的空间，并分别分配给两个新的叶节点。

以图 1(b) 中包含 iSAX词 [1,01,0] 的节点为例。二分策略在第二个分段上进行分裂，并生成两个新的叶节点，其 iSAX词分别为 [1,010,0] 和 [1,011,0]。经典 iSAX 索引 [56] 采用二分裂，因此第一层以下形成二叉树结构。第二种策略是在所有 w 个分段上进行分裂（即将每个分段上的 iSAX 符号基数加倍），并生成至多 2^w 个子节点（见

[1]iSAX词的符号有一种特殊情况，即 $*$，在该分段中，我们仅使用一个符号 $*$（$c = 1$）来表示整个值范围。



(a) 偏斜区域：a和b被错误地分组到节点 1-011-0 中

(b) 均匀区域：b和c被正确地分组到节点 10-01-01 中

图 2. 偏斜分裂对节点内序列邻近性的负面影响示意图。序列 b 和c 彼此相似，而序列a 与它们并不相似。

图 1(d)）。这种分裂模式会形成完整的全多叉结构，最近提出的 iSAX 系列索引 TARDIS [68] 采用了这种结构。

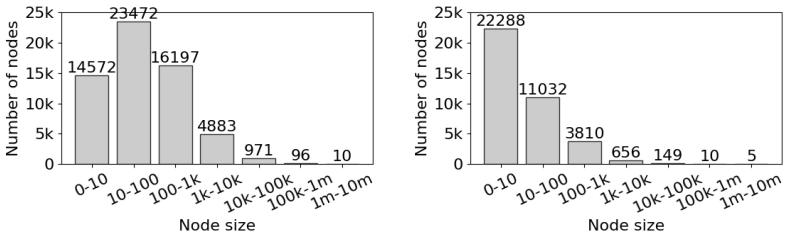
4 邻近性与紧凑性权衡

现在，我们基于对二叉结构索引和全多叉索引结构的分析，介绍邻近性与紧凑性权衡。更具体地说，我们认为，这两类结构都无法同时实现高叶节点填充因子（即高紧凑性）和高节点内序列相似度（即高邻近性），从而限制了索引构建效率和近似查询准确率。

[二叉结构的邻近性问题] 在类似 iSAX 的二叉结构索引中，最先进的分裂策略 [12] 旨在通过选择均值接近断点的分段，使两个子节点中的序列数量保持平衡。然而，这种策略会导致偏斜分裂：它可能在若干特定分段上多次进行分裂，从而生成一个包含多个粒度非常高、而其他分段粒度非常低的 iSAX 单词。这种情况如图 2(a) 所示，其中分段 2 已被分裂三次，而其他分段仅被分裂一次。对于父节点而言，选择分段 2 可能是最佳选择；然而，这一选择并不利于子节点内部序列的整体邻近性。在我们的示例中，序列 b 和c 整体上相似，但由于分段 2 存在细微差异，它们未能被归入同一组；与此同时，相距较远的序列a 和b 却被归入同一个节点。

直观地看，这是因为分裂决策考虑的是单个分段中的序列相似度（图 2(a) 中的分段 2），而邻近性取决于所有分段中序列之间的整体相似度。换言之，所有分段的粒度都应大致相同，这样才能更好地判断相似度（或等价地说，邻近性）。相反，如图 2(b) 所示，划分更均匀的节点能够成功地将序列 b 和 c 归为一组。需要注意的是，在二叉扇出的条件下，任何划分策略都无法在避免偏斜问题的同时实现均衡划分。因此，二叉扇出结构本质上都会受到邻近性问题的影响。

[全多叉结构的紧凑性问题] 与二叉扇出不同，全多叉结构会在所有分段上分裂节点。因此，它本质上避免了倾斜问题，形成严格均匀的区域。然而，它很快就会生成过多填充因子较低的小节点，严重损害索引紧凑性。我们的实验中的表1显示，在四个公开的大型数据集上，全多叉结构（TARDIS）的填充因子低于0.5%。因此，生成的索引在近似搜索中无法提供足够的候选序列，访问少量节点时准确率较低。就效率而言，尽管将子树合并为



(a) 随机游走 (b) DNA

图 3. 两个 100GB 数据集 ($w = 16$) 第一层中的节点大小分布。

更大的分区可以显著减少随机I/O，但在分区文件中存储和加载这种庞大的结构会给索引构建和查询带来很大开销，更不用说如此密集的节点数据包几乎会阻碍后续插入操作。

5 DUMPY

本节介绍我们的解决方案 Dumpy。Dumpy 基于一种新颖的自适应多叉结构，能够在邻近性与紧凑性权衡之间取得恰当的平衡。

接下来，我们将在第 5.1 节介绍基本思路，并在第 5.2 节描述 Dumpy 的构建流程。在第 5.3 节中，我们将介绍自适应分裂策略，并在第 5.4 节介绍叶节点打包算法。在第 5.5 节和第 5.6 节中，我们将分别介绍 Dumpy 的搜索和更新算法。

5.1 索引结构与设计概述

Dumpy 以层次结构组织数据序列，并采用与其他基于SAX的索引相同的自顶向下插入和分裂方式。一旦节点 N已满（其大小 c_N超过叶节点容量 th），Dumpy 就会自适应地选择 λ_N 个分段，并在这些分段上对节点 N进行分裂，以生成子节点。因此， $N \leq 2^{\lambda_N}$的扇出为。

我们在图4中展示了一棵包含 $w = 4$ 个分段的 Dumpy树示例。Dumpy N的内部节点维护一个所选分段列表， $csl(N) = [cs_1, cs_2, \ldots, cs_{\lambda_N}]$其中， cs_i是分段的标识符 （编号从 1 到 w），而 $csl(N)$按照分段标识符 的升序排列。将 $csl(N)$中每个符号增加的位串联起来后，可以得到一个长度为 λ_N 的位编码，在 Dumpy 中记为sid。节点 N_1 是一个包含 iSAX词 [0,1,1,1]的内部节点。 $csl(N_1) = [1, 3, 4]$表示 $\lambda_{N_1}=3$ ，我们在分段 1、3、4 上对 N_1 进行分裂。

在物理布局中，叶节点对应连续的磁盘页，用于存储原始序列和 SAX词。内部节点维护一个哈希表以支持树遍历，该表称为路由表，用于将标识符 映射到对应的子节点。

现在，我们介绍自适应节点分裂算法背后的直觉。为了在邻近性与紧凑性权衡之间取得最佳平衡，我们设计了一个目标函数来评估每种可能的划分方案：使用特定子空间中数据的方差来估算子节点内序列的邻近性，并使用叶节点填充因子的方差来替代紧凑性。考虑到整个搜索空间为 2^w+1，我们先剔除不理想的方案，再利用不同划分方案之间的关系来加速搜索（参见第 5.3 节）。

为了更好地实现我们的自适应分裂算法，我们提出了一种基于所有序列信息的新索引构建工作流（参见第 5.2 节）。以往基于SAX的索引的构建工作流

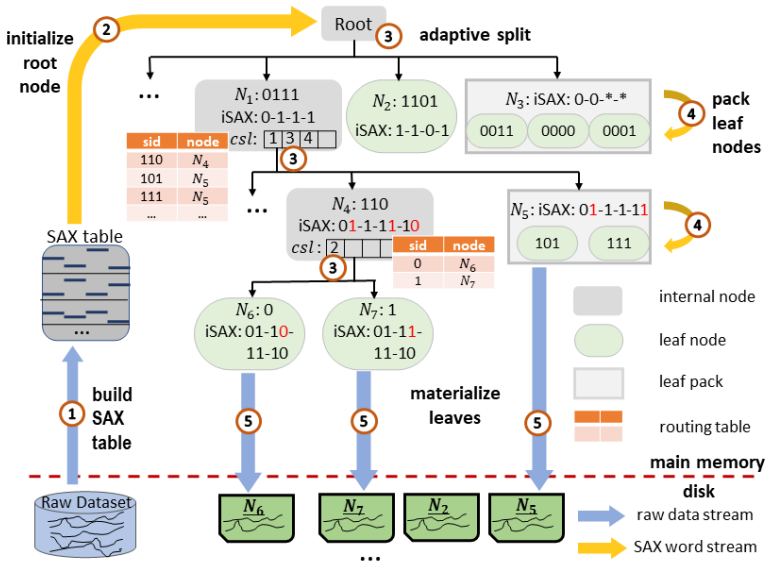


图 4. 索引结构（w=4）与构建工作流。

会在节点刚 满时对其进行分裂，也就是说，当第 th+1 个序列到达时。考虑到索引中的一个节点可能映射到远多于 th个序列，随着最初的第 $th+1$ 个序列很快只占落入该节点的所有序列中的一小部分，传统的分裂决策将失去效果。

最后，即使采用了最优的自适应分裂，仍可能存在大量小型叶节点。这是因为数据序列与高维向量类似，通常分布不均，形成许多不同的稠密和稀疏区域 [35]。图 3 展示了 iSAX 类型索引第一层的节点大小分布。Rand 中有 >60% 的节点、DNA 中有 >80% 的节点包含 <100 个序列，而 <2% 的节点则覆盖了 80% 的序列。为了彻底避免这一问题，我们提出了一种新颖的叶节点打包算法，通过限制叶节点包的最大降级位数来生成高质量的叶节点包（参见第 5.4 节）。

5.2 Dumpy 的构建工作流

图 4 展示了 Dumpy 的索引构建工作流。Dumpy 沿用了 iSAX 系列索引最先进的构建框架 [71]，但调整了两项关键设计，从而获得更优的索引结构和更高的构建效率。经典框架采用两遍处理流程。第一遍中，系统从原始数据集中批量读取数据序列，并计算每个序列的 SAX 词。随后，将这些 SAX 词逐一插入目标叶节点，并在节点满时分裂节点。第一遍结束后，索引结构便定型。第二遍中，系统再次批量读取数据序列，将其路由至正确的叶节点，最后写入相应文件。

[使用完整的 SAX词表拆分节点] 上述框架的一个关键点是仅在索引中保留 SAX词（第一遍），并用 SAX词拆分节点。这充分利用了 iSAX 摘要的静态特性，并显著减少了磁盘输入/输出。Dumpy 进一步扩展了这一工作流程，将 SAX 字词收集与节点拆分分成两个互不重叠的步骤：第一遍仅将所有 SAX词收集到 SAX 表中，然后在第二遍之前使用 SAX 表构建索引结构。因此，

算法1 Dumpy构建

输入： 数据集 db，叶节点容量 th，参数 r,ρ
输出： Dumpy 根节点 N_r
1: **对于** 按顺序读取的每个序列 s db **执行**
2: 计算 s 的 SAX词，并将其插入 SAX 表 T_{sax}
3: 初始化根节点 N_r.
4: $Split(N_r, T_{sax}, th)$
5: $NodePacking(N_r, r, \rho, th)$ 并标记新叶节点。
6: 将空哈希表初始化为 buf
7: **对**按顺序从 db 读取的每个序列 s **执行**
8: **如果** buf 已满，**则**
9: 将 buf 中的数据刷新到对应的叶节点文件。
10: 叶节点 $N_l = N_i.route2Leaf(T_{sax}[s])$
11: $buf[N_l].insert(s, T_{sax}[s])$
12: 刷新 buf 并将其清空。
13: 释放(T_{sax})
14: **返回** N_r

拆分节点时，我们知道其中序列的确切大小和分布，因此自适应分裂算法能够按预期真正发挥作用。

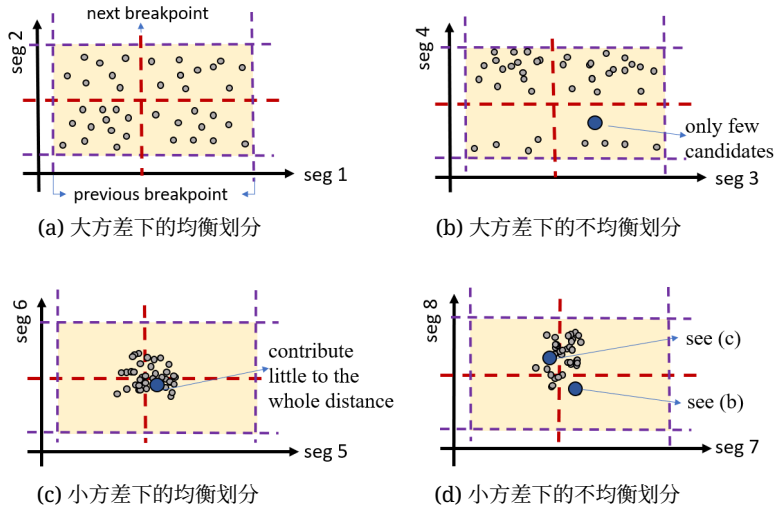
[叶节点打包后写入磁盘] 如图 3 所示，较大的扇出通常会在叶节点打包之前产生许多小节点。考虑到在第二遍中，我们会批量刷新每个相关叶节点中的序列，因此叶节点的数量大致决定了每批需要进行多少次随机磁盘写入。为减少随机写入，在第二遍中将叶节点实体化之前，Dumpy 会以恰当的方式合并同级小叶节点，形成更大的数据包，并为内部节点构建路由表。随后在第二遍中，序列将通过路由表直接路由到叶节点包，从而大幅减少随机磁盘写入。

现在，我们使用算法1介绍完整的 Dumpy 索引构建流程。在阶段1，从原始数据集构建 SAX 表（第 1–2 行）；在阶段2，初始化根节点（第 3 行）。在阶段3，首先对根节点执行自适应分裂，然后递归地对所有大小超过 th 的内部节点执行自适应分裂（第 4 行和算法2）。在阶段4，我们遍历索引树，并将同一父节点下的叶节点打包（第 5 行和算法 3）。至此，索引结构构建完成。在阶段5，我们将叶节点实体化（第 6–12 行）。第 6 行为叶节点准备一个缓冲区。随后，从磁盘读取原始序列，将其路由到叶节点，并连同对应的 SAX词一起缓存在缓冲区中（第 7、10–11 行）。缓冲区满后，会将数据刷新到每个节点对应的文件中（第 8–9 行）。

5.3 自适应节点分裂

现在，我们将介绍一种自适应策略：根据所有相关序列的 SAX词，确定扇出数和分裂位置（即在哪些分段上进行分裂）。我们的策略是基于一种新颖的目标函数选择最佳分裂方案，该目标函数同时考虑子节点内序列的邻近性以及子节点的紧凑性。由于所有可能的分裂方案数量多达 $2^w - 1$，我们还通过限制候选空间并重复利用共享信息，提出了一种高效的搜索算法。

5.3.1 目标函数。 我们的目标函数旨在使邻近性与紧凑性之间的权衡达到最佳平衡。我们根据候选分段上数据的平均方差来衡量邻近性。衡量紧凑性时，我们同时考虑子节点的填充因子方差以及溢出节点所占比例，以实现均衡划分，并避免偏向较小或较大的扇出数。下面给出所提出目标函数的形式化定义，并详细介绍其设计原则。



(a) 大方差下的均衡划分 (b) 大方差下的不均衡划分

(c) 小方差下的均衡划分 (d) 小方差下的不均衡划分

图5. 选择要分裂的分段时的四种典型情形。灰色点表示四种不同分裂方案下的同一组数据序列。蓝色点表示查询序列。

给定一个包含 c_N 个序列 $\mathrm{X}_N = \{\boldsymbol{x}^1, \boldsymbol{x}^2, \ldots\ldots, \boldsymbol{x}^{c_N}\}$ 的节点 N，其中 $\boldsymbol{x}^i = SAX(s^i, w, c)$ 是序列 s^i 的 SAX 词，且 x^i_j 是 $\boldsymbol{x}^i$ 的第 j 个符号；再给定一个划分方案 $csl(N) = [cs_1, cs_2, \ldots, cs_{|csl(N)|}]$，我们首先将 X_N 中的每个序列投影到 $csl(N)$ 的分段上，得到 $\mathrm{X'}_N$，也就是说，对于任意 i 和 $1 \le j \le |csl(N)|$，均有 $x'^i_j = x^i_{cs_j}$。然后，目标函数如下：

$$\max_{csl(N)} \; e^{\sqrt{\frac{1}{|csl(N)|} Var(\mathrm{X'}_N)}} + \alpha * e^{-(1+o)\sigma_F} \tag{1}$$

其中，e是欧拉数，投影数据的方差定义为 $Var(\mathrm{X'}_N) = \frac{1}{c_N} \sum_{i=1}^{c_N} \|\boldsymbol{x'^i} - \boldsymbol{\mu}\|^2$，$\boldsymbol{\mu}$是所选各分段上数据均值的向量[2]，$o \in [0, 1]$ 是溢出子节点的比例（大小为 $> th$），σ_F 是子节点填充因子的标准差，而 α 是用于平衡这两项度量影响的权重因子。

第一项用于估计划分方案的邻近性。它评估投影SAX空间中相关数据序列的平均方差，这等价于所有数据序列到其质心（即均值向量 $\boldsymbol{\mu}$）的平均距离。方差是衡量数据在特定维度上信息量的指标 [9, 25, 47], ，因为较大的方差通常意味着较大的信息熵 [55]。我们也在图5中对此作了解释。由于不同方案选择的分段数量可能不同，我们将方差除以所选分段的数量，以确保评估公平。

第二项用于评估划分方案的紧凑性。子节点填充因子的标准差可以防止出现极度不均衡的划分，并避免图 3 所示的严重数据倾斜：在这种情况下，该值会非常大。非正式地说，填充因子向量定义为 $\boldsymbol{F} = (F_1, F_2, \ldots\ldots, F_{2^{|csl(N)|}})$，其中 $F_i = c_{N_i}/th$，且 N_i 是第 i 个子节点。不过，该方法在扇出较小时会产生偏差，导致生成较少但较大的子节点，并造成不必要的树深度。为解决这个问题，我们添加了一个惩罚项（$1+o$），利用溢出子节点的比例来避免偏向扇出较小的方案。

[2]我们使用SAX 符号所表示范围的中点来计算均值及其他统计量。

我们在图5中进一步说明目标函数背后的启发式原理，其中展示了针对同一组数据序列的四种不同划分方案。不失一般性，我们假设所有方案都选择两个分段，即 $|csl(N)| = 2$。然后，我们将相关数据投影到二维 SAX空间中（不同方案对应不同空间）。目标函数选出的方案是图5(a)，其方差较大，且子节点均衡。图5(b)中的方案不如图5(a)，因为其子节点不均衡，导致候选序列不足，搜索准确率较低。图5(c)中的方案的问题在于方差较小，这意味着序列在这些维度上彼此相当接近；在这些维度上，序列之间的距离对所有 w 维度累积得到的整体距离贡献很小。也就是说，在这个子空间中，查询与序列之间的距离（无论大小）都无法反映它们在整个空间中的实际距离。具体而言，在这个子空间中彼此接近的序列，实际距离可能很大。因此，节点内序列的邻近性会显著减弱。图5(d)展示了最差的划分方案候选，而我们的目标函数可以轻松将其排除。

5.3.2 求解最优划分方案。一种朴素的求解方法是在我们的目标函数下遍历并评估所有 2^w-1 种划分方案。为了评估每种方案的目标函数，需要对所有序列至少进行四次遍历，这使中央处理器计算成为索引构建的瓶颈。为降低复杂度，我们提出了一种由三项实用加速技术组成的新型搜索算法：预先计算每个分段的方差、利用用户定义的填充因子范围限制搜索空间，以及分层计算子节点的大小。我们首先详细介绍每项技术，然后将它们整合到算法2中的完整节点划分算法中。

[预计算方差] 我们发现，在目标函数的第一项中，$Var(\mathrm{X'}_N)$可以通过线性累加各分段上的数据方差来计算。

$$Var(\mathcal{X'}_N) = \sum_{cs \in csl(N)} Var(\Pi_{cs}(\mathcal{X}_N)) \tag{2}$$

其中，$\Pi_{cs}(\mathrm{X}_N)$表示X_N在分段 cs上的投影。

证明。

$$\sum_{cs \in csl(N)} Var(\Pi_{cs}(\mathcal{X}_N)) = \frac{1}{c_N} \sum_{cs} \sum_{i=1}^{c_N} (x_{cs}^i - \mu_{cs})^2$$

$$= \frac{1}{c_N} \sum_{i=1}^{c_N} \sum_{cs} (x_{cs}^i - \mu_{cs})^2 = \frac{1}{c_N} \sum_{i=1}^{c_N} \|\boldsymbol{x'^i} - \boldsymbol{\mu}\|^2 = Var(\mathcal{X'}_N)$$

因此，我们可以在开始分裂节点时，预先计算每个分段上数据序列的方差。在评估特定方案时，我们只需获取对应分段的方差并将其相加，复杂度为常数。

[限制搜索空间] 现在，我们考虑基于简单的启发式方法剪除不切实际的方案。以两个极端情况为例：在 w 个分段上拆分大小为 th+1 的节点会生成过小的节点，而在一个分段上拆分一个大小为一百万的节点，则会生成两个规模庞大、远超 th的节点。因此，自然应将子节点的平均填充因子限制在合理范围内，并避免进行这种特定的评估。我们引入一对参数 F_l, F_r，用于限定子节点平均填充因子的范围。由此可以推导出所选分段数量 $|csl(N)|$的范围：

$$\max(1, \log \frac{c_N}{F_r * th}) \leq |csl(N)| \leq \min(w, \log \frac{c_N}{F_l * th}) \tag{3}$$

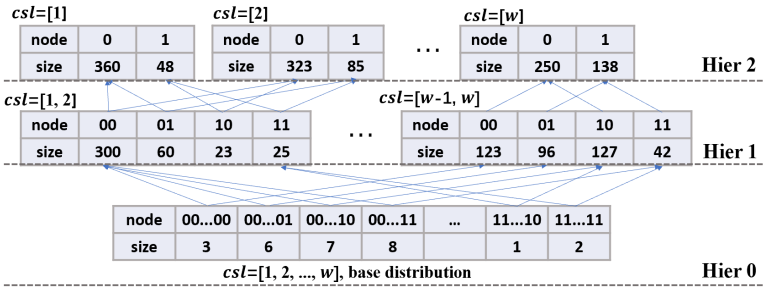


图6. 为每个候选分裂方案分层计算子节点大小。

在实践中，我们通过经验将 $F_l = 50\%$ 和 $F_r = 300\%$ 设为相应值，通常可实现 16 倍加速比，平均准确率达到 99%。对于相对较小的节点，加速效果尤为显著，因为剪枝后的搜索空间非常小。

[分层计算子节点大小] 到目前为止，对于每个方案，我们仍需遍历所有数据序列，以获取子节点的大小。我们观察到，如果一个分裂方案 $csl^i(N)$ 是另一个方案 $csl^j(N)$ 的子集，那么方案 $csl^j(N)$ 的子节点大小分布可以复用于计算方案 $csl^i(N)$ 的分布。假设 $csl^i(N) = [1]$ 和 $csl^j(N) = [1, 2]$（图6左上部分），且 $csl^j(N)$ 下四个子节点的大小分别为 $c_{N_{00}} = 300$、$c_{N_{01}} = 60$、$c_{N_{10}} = 23$、$c_{N_{11}} = 25$（节点用其 sid 表示），那么 $csl^i(N)$ 下两个子节点的大小可计算为 $c_{N_0} = 300 + 60 = 360$ 和 $c_{N_1} = 23 + 25 = 48$。由于整个 w分段是各个分裂方案的超集，我们首先计算每个分裂中 w 分段的子节点大小，作为基分布；然后从扇出最大的方案开始，以深度优先方式遍历其他方案，直至扇出最小的方案。因此，我们可以分层复用已获取的大小分布，避免针对每个方案遍历所有序列。

我们在算法2中总结了我们的分裂策略。其主干位于第4—13行和第19—42行，而第14—16行根据划分方案对数据进行分区，第17—18行则递归地分裂溢出子节点。我们首先准备每个分段上的数据方差（第4—6行）和基分布（第7—10行），然后确定扇出的范围（第11行）。接着，我们以层次化的方式评估每个划分方案（第19—42行），并使用哈希集合避免重复评估。第21—26行遍历基准方案中的每种组合，将其作为当前划分方案并计算其分布。随后，评估目标函数以更新当前最优结果。最后，以类似的方式评估当前方案的子方案。

5.4 叶节点打包

如第5.1节所述，当扇出较大时，数据序列通常会在子节点中分布不均，从而产生大量小型叶节点，进而导致索引性能退化。TARDIS [68] 采用基于节点大小的策略，将子树合并为一个分区；而 Coconut [33] 提出了针对iSAX词的排序技术，这两种方法都会严重削弱甚至完全放弃所得索引的剪枝能力。为了解决这一问题，我们提出了一种简单而有效的算法，在不损失打包节点剪枝能力的情况下，对小型叶节点进行打包。

其核心思路是尽可能缩小打包节点在 SAX空间中占据的值域，也就是让它们具有最紧凑的iSAX表示。更紧凑的 iSAX表示能直接转化为更强的剪枝能力。例如，与合并 0010 和 0101 相比，合并 sid 0010 和 0100 更好，因为这样只需降级两位，就能得到 $0**0$，而不是降级三位

算法2分裂

输入： 节点 N，SAX 表 T_{sax}，叶节点容量 th
如果 N 是根节点 **则**
  $csl(N)$ = $[1, 2, \ldots, w]$
否则
  分段方差 = []
  对于 i = 1 至 w，**执行**
    分段方差.push(分段 i中的数据方差)
  将 $baseDist$ 初始化为大小为 2^w 的列表：[0,0,...,0]。
  对于 每个数据序列 s，在 N**中执行**
    $sid, _ = promoteiSAX(iSAX(N)\ T_{sax}[s], [1, 2, ..., w])$
    $baseDist[sid]$++
  根据公式3计算 λ_{max},λ_{min}
  将 $visit$ 初始化为空集
  $csl(N)$ = calcDist($baseDist$, $[1, 2, ..., w]$, λ_{max}, λ_{min}, $visit$, $segVar$, null)
对于每个数据序列 s，在 N**执行**
  $sid, isax_{new}$ = $promoteiSAX(iSAX(N)\ T_{sax}[s], csl(N))$
  插入 s到 $N.children[sid]$
  将 $N.children[sid]$标记为 $isax_{new}$
对于满足 $c_{N_i} > th$ 的 N 的每个子节点 N_i**执行**
  $Split(N_i,\ T_{sax},\ th)$

函数calcDist($baseDist$, $basePlan$, λ_{cur}, λ_{min}, $visit$, $segVar$,bsf)
如果 $\lambda_{cur} < \lambda_{min}$ **then**
  返回 bsf
对于 基数为 λ_{cur}的每种组合，从 $basePlan$ 中取出为 csl^{cur} **执行**
  如果 $visit$ 不包含(csl^{cur}) **则**
    将 $Dist$ 初始化为大小为 $2^{\lambda_{cur}}$的列表：[0,0,...,0]
    对于 每个 $item$，在 $baseDist$ **中执行**
      sid_{new} = 从 $item.sid$中提取 csl^{cur} 中的位
      $Dist[sid_{new}]$+=$item.size$
    根据方程 1 计算 $score$
    如果 分数 > bsf.分数 **那么**
      bsf = csl^{cur}
    calcDist($Dist$,csl^{cur}, λ_{cur}-1, λ_{min}, $visit$, $segVar$, bsf)
    $visit$.插入(csl^{cur})
返回 bsf

函数 promoteiSAX(iSAX词 $isax$, SAX词 sax, 选定分段列表 csl)
sid = 0
$isax_{new}$ = $isax$
对于每个分段 seg，在 csl**执行**
  nb = $len(isax[seg])$
  sid = sid << 1 + $sax[seg]$的（$nb + 1$）位
  $isax_{new}[seg]$ = $sax[seg]$的前 $nb + 1$ 位
返回 sid ，$isax_{new}$

得到较粗略的 0 * **。我们将降级位定义为两个或多个拟合并到同一数据包中的节点的 sid 之间不同的位。在我们的节点打包算法中，我们将降级位数量限制为小于 $\rho\lambda$，其中 ρ 是一个用户定义参数，用于权衡打包质量和填充因子。具体而言，给定一个数据包列表和一个待打包的小节点 N，我们根据降级成本决定 N 归属哪个数据包。降级成本指将 N 加入数据包后，该数据包增加的降级位数量。如果插入任何节点都会导致叶节点包降级超过 $\rho\lambda$ 位或溢出（大小为 $> th$），则该叶节点包会拒绝该节点的插入请求。最后，如果没有现有数据包能满足插入 N 的要求，我们就创建一个新数据包，并将 N 插入其中。

该算法的具体细节如算法 3 所示。我们首先汇总所有小型叶节点的大小（第 1–5 行），并使用 $\lfloor sum_size/th \rfloor$ 个叶节点随机初始化打包列表，这一数量是容纳小节点所需的最小数量（第 6 行）。然后，我们遍历其余的小节点，并为它们选择或创建一个数据包（第 7–20 行）。对于节点 N，我们检查打包列表中的每个现有数据包，并选择降级成本最低的数据包（第 14–16 行、第 19–20 行）。当数据包的降级超过 $\rho * \lambda_{N_p}$ 位时，我们便放弃这一选择（第 11–12 行）。此外，我们还确保大小

算法 3节点打包

输入： 父节点 N_p、小节点阈值 r、降级比率 ρ、叶节点容量 th

1: $nodes$ = [], sum_size = 0
2: **对于** N_p 的每个未分裂的子节点 N_i **执行**
3: **若** $c_{N_l} < r * th$ **则**
4: $nodes$.push(N_l)
5: sum_size += c_{N_l}
6: 用 $\lfloor sum_size/th \rfloor$ 叶节点初始化打包列表 pl
7: **对**每个节点 N in $nodes$ **执行**
8: bsf = (null, λ_{N_p})
9: **对**每个数据包 p in pl **执行**
10: p' = $p.insert(N)$
11: **如果** $c'_p > th$ 或 $p' > \rho * \lambda_N$ 的降级位数 $_p$ **then**
12: 继续
13: **else**
14: $cost$ = p' 中降级位数增加
15: **如果** $cost < bsf$.cost **那么**
16: bsf = (p, $cost$)
17: **如果** bsf.pack == 为空 **则**
18: pl.push(使用 N 初始化的数据包)
19: **else**
20: bsf.pack.插入(N)
21: 更新 N_p 的路由表
22: **对于** N_p 的每个满足 $c_{N_i} > th$ 的子节点 N_i **执行**
23: $NodePacking(N_i, r, \rho, th)$

所有数据包的大小不会超过 th，从而避免产生新的分裂（第 11–12 行）。当不存在符合条件的数据包时，我们将创建一个以节点 N 初始化的新数据包（第 17–18 行）。最后，我们更新节点 N_p 的路由表。

小节点的定义可根据具体用例确定。对于静态数据集，可将小节点阈值 r 设为 1 以提升性能；对于动态数据集，r 则可根据历史更新频率动态设置。也就是说，对于更新密集型用例，应采用较小的 r，反之亦然。

5.5 搜索算法

Dumpy 支持两种查询回答算法。第一种遵循经典的基于剪枝的搜索算法 [22]。作为一种基于SAX的索引，Dumpy 可以通过使用 iSAX词的下界距离剪去无关叶节点，从而高效地进行搜索（包括精确搜索、 δ-ϵ-近似搜索等 [22, 56]）。此外，Dumpy 还支持传统近似搜索，即查询一个目标叶节点。我们还将其扩展为允许搜索更多节点，称为扩展近似搜索，从而在保持毫秒级响应时间的同时提升查询结果质量。我们将扩展近似搜索的范围限制在目标叶节点所在的最小子树中，以降低复杂度，并避免像基于界限的搜索方式那样遍历整棵树并逐一评估节点。得益于 Dumpy 的多叉结构和填充因子，该方法显著提高了搜索准确率。

如算法4所示，输入中还包含一个额外参数，用于限制访问的节点数。搜索过程从根节点开始，并在遇到节点数少于输入值的节点或空节点时结束（第1–4行）。然后，我们根据结束节点的兄弟节点的下界距离（ED见 [56]，动态时间规整见 [49]）按升序排序（第5行）。最后，我们依次遍历排好序的兄弟节点或子树，直到达到最大访问节点数，并返回 kNN（第6–9行；具体搜索过程略去，因为它与其他搜索算法相同）。

算法4扩展近似搜索

输入： 根节点 N_r，节点数 nbr，查询序列 q

1 $N = N_r$: 节点 : **当** ! null 且 N.leafNbr $>$ nbr **执行**
2 $N =$
3: $sid = promoteiSAX(iSAX(N)\ SAX(q)\ csl(N))$ 4:
$N = N.routingtable[sid]$ 5: 按照下界距离对 N 的兄弟节点
排序，得到列表 l 6: **当** 已搜索节点数 $<$ nbr **执行** 7: N_c = 弹出
l 的头节点 8: 获取以 N_c 为根的所有节点，并搜索其中的序
列 9: **返回** 已访问序列中的 kNN

5.6 更新

作为一个功能完备的索引，除了批量加载外，Dumpy 还支持更新（插入和删除）。一个主要区别是，Dumpy 不再预先收集动态数据集中的所有序列信息。不过，由于所有 SAX 词都与原始序列一起存储在叶节点中（与 iSAX 索引系列 [56] 相同），当内部节点的扇出和大小不满足公式3中的约束时，我们可以重新组织索引结构。

具体来说，我们读取以该节点为根的叶节点中的所有 SAX 词，并遵循算法1中的相同工作流程。请注意，此过程可以在后台线程中执行，而不会阻塞前端服务。在此期间，更新查询序列会同时进入旧结构和尚未完成的新结构。后台工作完成后，我们用新子树替换旧子树，并释放旧子树占用的所有空间。另一个区别是查询序列落入满包时的处理方式。此时，我们只需从数据包中提取目标叶节点，并在经过大量此类提取后，对其兄弟节点重新进行节点打包。这些操作非常快，因为这些小节点通常只包含少量序列。

删除操作与 iSAX 索引系列 [56, 70] 几乎相同。具体而言，我们通过位向量在相应的叶节点中标记已删除的数据序列，后续的插入操作可以利用已删除序列占用的空间，而查询会忽略这些条目。当节点为空时，我们会释放其占用的空间。Dumpy 唯一不同之处在于需要更新路由表。

5.7 复杂度分析

本节首先分析索引构建和查询的时间复杂度，然后分析空间复杂度。

[时间复杂度] 作为一种基于磁盘的索引，Dumpy 的时间开销取决于内存内复杂度和磁盘访问。接下来，我们先讨论索引构建的理论时间开销，再讨论查询的理论时间开销。

Dumpy 索引构建的复杂度可由各个子模块的复杂度相加得到。在自适应分裂算法中，设节点 N需要分裂，且根据公式3，$|csl(N)|$介于 λ_{min} 和 λ_{max} 之间。随后，计算每个分段的方差、基分布和路由目标的开销为 $O(4wc_N)$。$calcDist$函数的计算次数为 $\binom{w}{\lambda_{max}} * 2^w + \sum_{i=\lambda_{min}}^{\lambda_{max}-1} \binom{w}{i} 2^{\lambda_{i+1}}$，其中第一项对应于评估最大扇出下的所有可能方案，即使用 λ_{max}个分段（参见图6中层级结构的 Hier 1 层）；第二项对应于评估较小扇出数下的所有可能方案（参见图 6 中的 Hier 2 层）。在叶节点打包算法中，假设最终的数据包数量为 np，节点打包的时间复杂度为 $O(2^{\lambda_N} * np)$。总之，内存内复杂度为 $O(\sum_N(4wc_N + \binom{w}{\lambda_{max}} * 2^w + \sum_{i=\lambda_{min}}^{\lambda_{max}-1} \binom{w}{i} 2^{\lambda_{i+1}} + 2^{\lambda_N} * np))$。

构建 Dumpy 时，随机磁盘写入可能会产生显著开销（参见图4的阶段5）。假设数据库中的数据序列数量为 $|db|$，叶节点数量为 n_l，且

内存缓冲区可容纳 B 个序列。每个批次中，Dumpy 最多生成 n_l 次随机写入，整个索引构建过程总共会产生 $O\left(\frac{|db|}{B} * n_l\right)$ 次随机写入。

对于查询，近似搜索会从根节点沿单一路径向下到达目标叶节点。设该路径长度为 $|p|$，则其代价为 $O(|p|w)$。I/O 代价为一次大小为 $O(th)$ 的磁盘读取。与 iSAX 索引（二叉扇出）相比，Dumpy 路径的长度 $|p|$ 小 $2/\lambda$ 倍，其中 λ 表示 Dumpy 的平均扇出。除搜索目标叶节点的代价外，精确搜索的复杂度还包括 $O((1-pr) * n_l)$ 次随机磁盘读取，每次读取大小为 $O(th)$；其中 pr 为剪枝率；以及 $O(w * n_{total} \log n_{total})$ 次内存计算，其中 n_{total} 为节点总数。

实际上，Dumpy 是一种比其他基于 SAX 的索引更紧凑的索引（n_l 和 n_{total} 的值更小；参见第 7.1 节），因此构建和查询速度都更快。

[空间复杂度] Dumpy占用的空间（不包括原始数据大小）如下。SAX词持久化存储在磁盘上，占用 $\lceil wb * |db|/8 \rceil$ 字节。索引的内部节点存储路由表、iSAX词以及用于分裂的分段列表（即选定的分段），总计 $\sum_N (8 * 2^{\lambda_N} + wb/8 + \lambda_N)$ 字节。叶节点存储单个iSAX词摘要，总计 $n_l * (wb/8)$ 字节。由于节点数量较少，Dumpy在实践中只会带来极少的额外存储开销。

6 DUMPY-FUZZY

作为基于分区的索引，数据序列索引在近似搜索中也会遇到所谓的边界问题 [17, 24]。也就是说，位于查询所属节点边界附近的数据序列也是很好的候选项，但由于它们可能位于不同的子树中，因此无法纳入近似搜索。为了解决这一问题，我们提出了Dumpy的一种变体，称为Dumpy-Fuzzy。它将静态分区边界（即SAX断点）视为一个范围（模糊边界），并将落在该范围内的序列放入边界两侧的节点中。与Dumpy相比，Dumpy-Fuzzy进一步提高了近似搜索的准确率，代价是在索引构建和磁盘空间方面带来少量开销。

具体而言，Dumpy-Fuzzy 在分裂后增加了一个复制过程。对于每个新生成的内部节点 N，它会检查位于 N 的相邻节点上的序列（即 sid 与 N 相差 1 位的节点），并将边界附近的序列复制到该节点中。例如，对于一个带有 $sid = 000$ 的节点，它会检查第一个选定分段上相邻节点 100 中的序列，并将非常接近 000 的序列复制到 000 中。相同的过程也分别适用于第二和第三个选定分段上的节点 010 和 011。

我们引入一个超参数 $f \in (0, 1)$，即相对于原始节点范围的模糊边界范围，用于控制哪些序列符合复制条件。此外，叶节点打包后也会进行复制。叶节点包边界附近的序列也可以按照上述相同方式冗余地放入数据包中。我们确保此过程中不会引入额外的分裂（即叶节点包不会溢出）。

请注意，Dumpy-Fuzzy 不会损害 Dumpy 的精确搜索剪枝能力。重复的序列不会改变节点或数据包的 iSAX 词。因此，下界计算保持不变。由此，Dumpy-Fuzzy 在不违反基于剪枝的精确搜索的前提下，通过检查更有希望的候选项提高了近似搜索准确率。

我们还声称，二叉结构和全多叉结构都难以采用类似的模糊边界优化。在二叉结构中，由于每个叶节点只有一个兄弟节点，因此容易产生过多的重复序列，并导致更多分裂。这会使索引变得更深，反而降低搜索准确率。至于全多叉结构，叶节点数量过多会转化为数量庞大的复制目标，从而带来

无法接受的存储开销。我们已通过实验验证这些观察结果，但由于篇幅有限，省略了相关结果和详细算法。

7 实验

[实验环境] 实验在一台搭载 Intel Core(R) i9-10900K 2.80GHz 10 核中央处理器、配备 4*32GB 2400MHz 主存的计算机上进行，运行 Windows 子系统 Linux（Ubuntu Linux 20.04 LTS）。该计算机配有一块三星 PCIe 2TB 固态硬盘（默认）和一块希捷 SATA III 7200RPM 2TB 机械硬盘。我们的代码可在 https://github.com/DSM-fudan/Dumpy 获取。

[数据集] 我们使用一个合成数据集和三个真实数据集。所有序列在建立索引和查询前均经过 z 标准化。对于每个数据集，我们准备了 200 个不属于该数据集的查询，以确保具有足够的难度 [73], , 并使用暴力搜索获得真实值 k近邻结果。**Rand**是一个合成数据集，由符合标准高斯分布 $N(0, 1)$的随机游走步长累加生成。现有研究已广泛使用该数据集 [2, 21, 22, 72]。我们生成了 5000 万至 8 亿条不同长度的 Rand 序列（50GB–800GB）。**DNA** [1] 是一个真实数据集，采集自两种植物 Allium sativum 和 Taxus wallichiana 的 DNA 序列。它包含 2600 万条长度为 1024 的数据序列（ ~113GB ）。第二个真实数据集 **ECG** （心电描记术）提取自 MIMIC-III 波形数据库 [30]。它包含超过 9700 万条长度为 320 的序列（~117GB），这些序列以 125Hz 的频率采集自 6146 名重症监护病房患者。最后一个真实数据集**Deep** [61],包含 10 亿个大小为 96 的向量，提取自图像卷积神经网络（CNN）的最后几层。

[算法] 在 iSAX 索引系列中，我们将 **iSAX2+** 作为最先进的二叉结构 [22]。我们还实现了 **TARDIS** 的独立版本，将其作为最先进的全多叉结构，并采用 100% 的采样比例。**DSTree** [65] 也作为最先进的数据序列索引之一 [22]被纳入其中。为简单起见，参数为 f 的 Dumpy-Fuzzy 简写为 **Dumpy**-f。为评估这些索引的质量，我们实现了扩展近似搜索和基于剪枝的搜索。

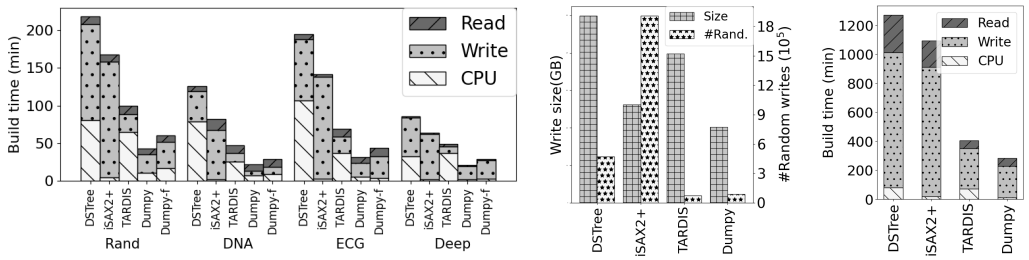
所有代码均为开源代码，使用 C/C++ 实现，并通过 g++ 9.4.0 使用 -O3 优化进行编译。我们使用经过优化的 DSTree 和 iSAX 版本2+ [21]。

[参数] 我们设置分段数量为 $w = 16$、SAX 基数为 $c = 64$ （即 $b = 8$）以及叶节点大小阈值为 $th = 10000$。除非另有说明，用于索引构建的内存缓冲区大小设置为 $4GB$。Dumpy-f 中每个序列的复制次数最多设置为 3。

[度量指标] 与其他研究类似， [4, 22, 47], 我们使用平均精度均值（MAP） [60] 作为准确率度量，它定义为一组查询的 AP 均值。对于查询s_q，AP 等于 $\frac{1}{k}\sum_{i=1}^{k} P(s_q, i) * rel(i)$，其中 $P(s_q, i)$ 是前 i 个最近结果中真实近邻所占的比例，而 $rel(i)$ 在第 i 个最近结果为真实的精确 kNN 结果时取 1，否则取 0。可以证明，当返回结果按实际距离排序时，MAP 等价于平均召回率。我们使用的另一种相似度度量是平均误差比，用于衡量近似结果与精确结果之间的差异，通常用于近似搜索， [4, 68], 其定义为 $\frac{1}{k}\sum_{i=1}^{k} \frac{dist}{dist(a_i, s_q)(r_i, s_q)}$。我们对 ED 和 DTW 均进行度量，其中 DTW 的扭曲窗口大小设为序列长度的~~10%,~~ 这是常见的设置 [49, 53]。

7.1 索引构建

7.1.1 效率。首先，我们评估了四个数据集在固态硬盘上的索引构建效率，结果如图7(a)所示。在所有四个数据集中，Dumpy 的表现都远胜其他三种方法，平均速度分别比 DSTree 快 5.3 倍、比 iSAX2+ 快 3.8 倍、比 TARDIS 快 2.5 倍。Dumpy-f只会给 Dumpy 带来少量开销（约**38%**），且明显快于 DSTree 和 iSAX2+。Dumpy 的优势主要来自随机磁盘写入的减少。如图 7(b) 所示，iSAX2+ 的随机写入次数（#Rand.）是 ~**3.5x**



(a) 在固态硬盘上构建四个数据集的耗时 (b) 构建过程中的磁盘写入 (c) 在机械硬盘上的构建耗时

图 7. 索引构建（4GB内存）

表1. 索引结构统计

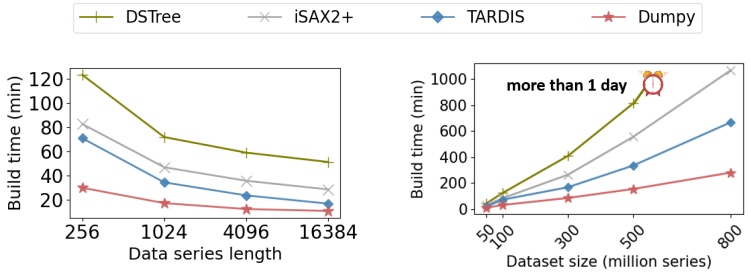
数据集	方法	#叶节点	#节点	高度	填充因子	索引大小（MB）[1]
Rand	iSAX2+	73563	86945	20	13.59%	16
	DSTree	17847	35693	32	56.03%	9
	TARDIS	8516867	8520065	**3**	0.11%	732
	Dumpy	**14106**	**19418**	7	**70.89%**	**3**
DNA	iSAX2+	42906	47885	25	6.14%	9
	DSTree	5833	11665	43	45.16%	3
	TARDIS	1011436	1312989	**5**	0.26%	278
	Dumpy	**4367**	**6228**	9	**60.32%**	**1**
ECG	iSAX2+	69786	74042	9	13.98%	14
	DSTree	20740	41479	48	47.04%	13
	TARDIS	3178628	3182368	**4**	0.33%	749
	Dumpy	**12112**	**15050**	7	**80.55%**	**3**
Deep	iSAX2+	68096	71188	8	19.08%	11
	DSTree	16324	32647	33	61.26%	8
	TARDIS	824458	827094	**3**	0.27%	546
	Dumpy	**11590**	**13664**	8	**86.28%**	**3**

[1] 仅内存索引结构的大小。

比 DSTree 多，且是 Dumpy 和 TARDIS 的 ~**17** 倍，因此无论在固态硬盘还是机械硬盘上，随机写入次数都是索引构建时间的主导因素，而非写入字节数（#Seq.）。尽管 TARDIS 的分区紧凑且较大，因而 I/O 成本也较低，但它需要耗费更多 CPU 时间，将庞大的索引结构序列化到每个分区中。图 7(c) 中机械硬盘上的结果与固态硬盘上的结果相似。

我们在表1中展示了详细的索引信息。Dumpy的叶节点最少（即填充因子最高），这证实了Dumpy具有良好的紧凑性。DSTree的叶节点数略多于Dumpy，而iSAX2+的叶节点数通常是DSTree的 >3倍。这验证了第5.2节中的复杂度分析。值得注意的是，DSTree的索引构建时间甚至比iSAX2+更长（图7(a)），这是由于EAPCA计算造成了较高的CPU开销和较大的写入字节数。TARDIS会生成百万级叶节点，且其全多叉结构的填充因子较低。这些节点被组织在大型分区中，每个分区为128MB，与 [68]的设置一致。

7.1.2 可扩展性。 接下来，我们通过将数据量从 50GB 增加到 800GB，并将序列长度从 256 增加到 16384，测试 Rand数据集上的可扩展性。改变数据集大小时，序列长度保持为 256；改变序列长度时，数据集大小保持为 100GB，采用与基准测试 [21]相同的设计。



(a) 对数据序列长度的影响 (b) 对数据集大小的影响

图 8. 索引可扩展性（32GB 内存）

图 8 展示了在 32GB 内存下的索引构建时间。Dumpy 在两种情况下均表现出最佳的可扩展性。在线性回归测试中，Dumpy 的构建时间与数据集大小的决定系数 R^2 为**0.9904**，验证了其构建时间呈线性增长。这是因为随着数据集扩大，叶节点数量也线性增加，表明平均填充因子几乎保持不变。这也支持复杂度分析。改变序列长度时的性能同样符合这一规律。

7.2 查询处理

在本节中，我们验证查询处理的准确率。

7.2.1 近似搜索。首先，我们评估了四个数据集上的近似结果准确率。

[搜索一个节点] 我们在三个数据集上使用 ED距离搜索一个节点，以获得近似的前k 个结果，并比较这些方法。结果如图9所示。可以看出，Dumpy 的表现始终优于其他方法。具体而言，与 TARDIS、iSAX2+ 和 DSTree 相比，Dumpy 在三个数据集上的平均准确率分别提高了**84%**、**46%**和**11%**，平均误差比分别降低了**7.3%**、**3.4%**和**1.4%**。TARDIS 的性能最低，这是由于其填充因子较低。iSAX2+ 的节点内序列邻近性不足，其特点是存在数量不均衡的节点，如图5所述（>**20%** 的叶节点中，有一个分段使用的位数比其他分段多 4 位以上）。此外，Dumpy-Fuzzy 的准确率高于 Dumpy 和其他方法，这验证了我们的复制策略。

[搜索多个节点] 在图 10 中，我们比较了使用 ED距离搜索多个节点（1 至 25 个）时前1结果的准确率。得益于多叉结构能够提供距离更近的兄弟节点，Dumpy 和 Dumpy-f的 MAP 值相比其他方法提升得快得多。访问 25 个节点时，与次优方法 DSTree 相比，Dumpy 和 Dumpy-f在四个数据集上的平均准确率分别提高了 **58%** 和 **65%**，平均误差比分别降低了 **3.6%** 和 **3.7%**。我们还比较了序列长度变化时的准确率（图 11）。不同长度下的准确率排名相似。

图 12 展示了不同方法处理近似前50查询的平均查询时间。可以看出，Dumpy 的查询处理时间与 iSAX2+ 相近，且快于 DSTree。也就是说，我们的方法能够以更低的时间成本实现更高的准确率。TARDIS 的速度慢得多，因为反序列化大型索引分区耗费了大量时间。

我们的结果表明，在 TB 级数据集上，Dumpy 能在 100 毫秒内达到 60%–70% 的平均准确率均值，同时其索引构建时间比最先进方法竞争者快 4 倍。这些结果表明，Dumpy 满足许多基于 k近邻的应用的需求。

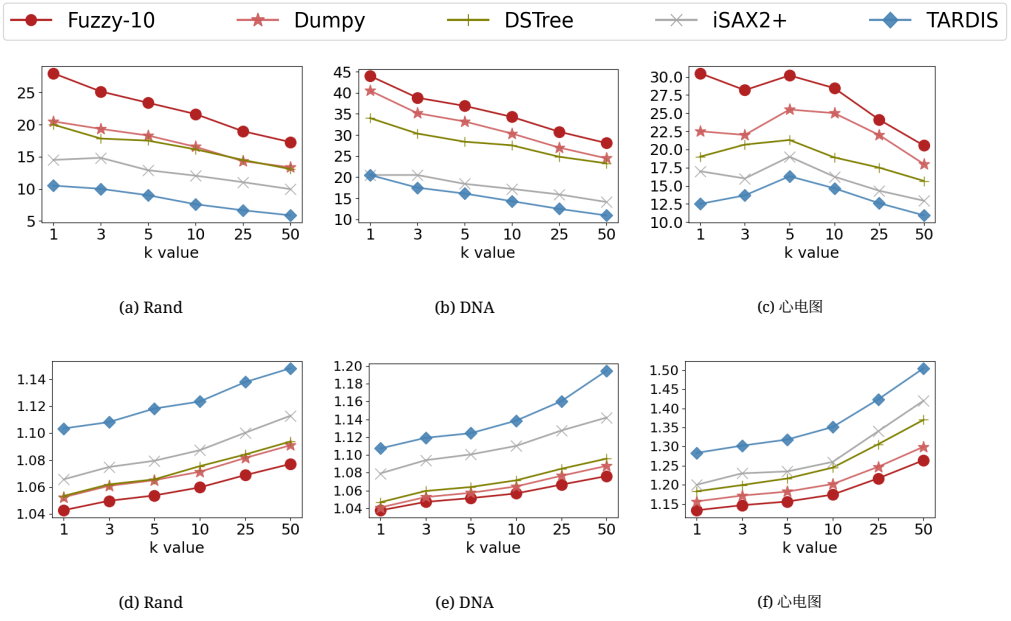


图 9. 基于欧氏距离的近似搜索（搜索单个节点）。

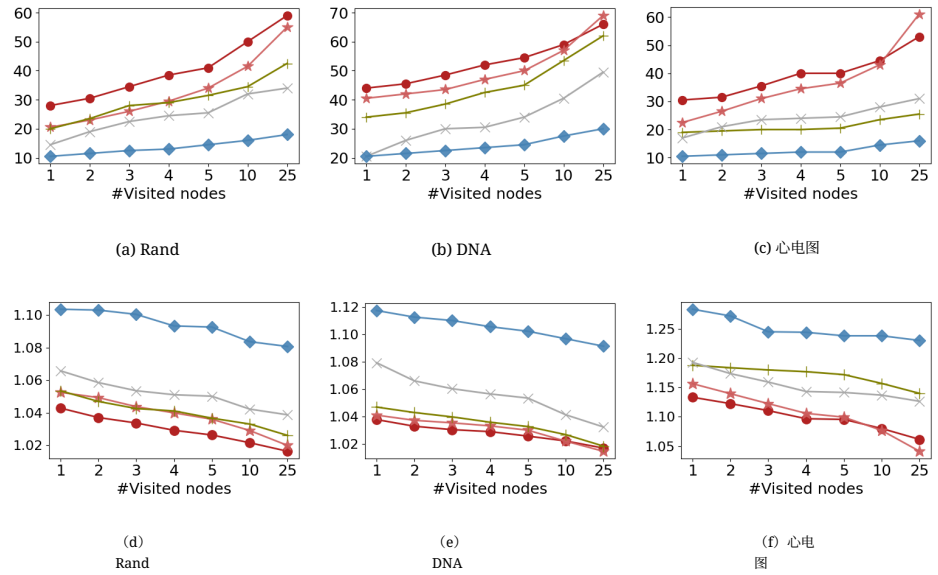


图10。ED(k=1)下的扩展近似搜索。

[近似最近邻距离的上界] Dumpy（与其他索引一样）不对其（ng-）近似搜索结果提供任何准确率保证 [21, 22]。不过，我们可以给出近似最近邻与查询序列之间距离的上界。这

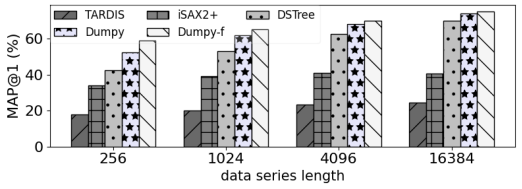


图11. 扩展的近似搜索与序列长度（搜索25个节点）

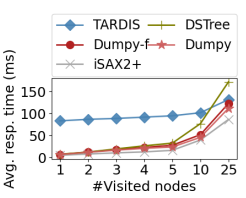


图12. 查询时间

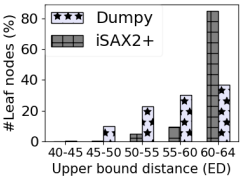
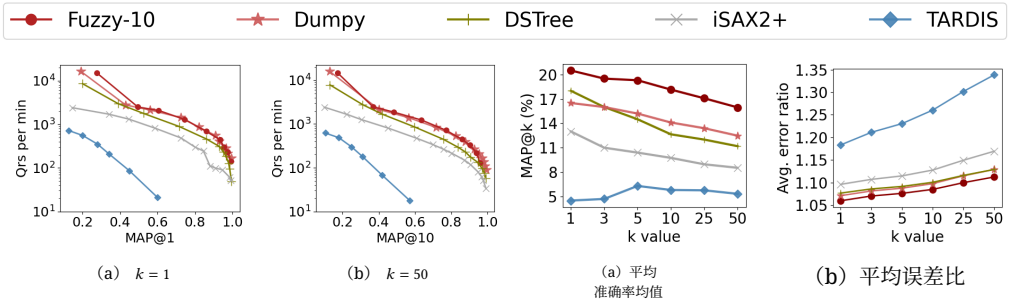


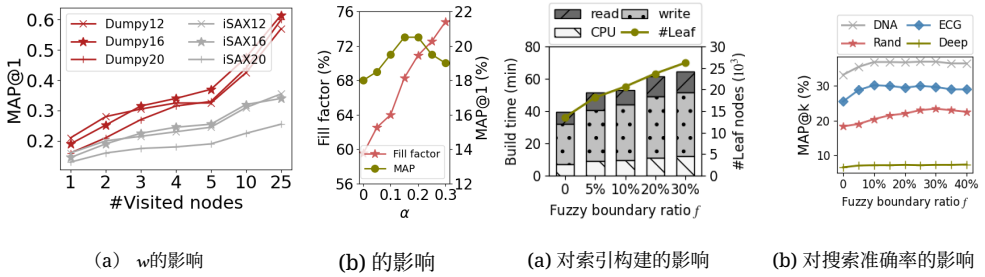
图13. 距离上界的直方图



（a） $k = 1$ （b） $k = 50$ （a）平均准确率均值 （b）平均误差比

图14。效率与准确率

图15。基于动态时间规整的近似搜索



（a） w的影响 (b) 的影响 (a) 对索引构建的影响 (b) 对搜索准确率的影响

图16. 参数的影响

图17. f 的影响

上界可解释为 Dumpy 近似搜索结果在最坏情况下的准确率。

给定目标叶节点，该上界距离定义为 $\sqrt{\frac{l}{w}\sum_{i=1}^{w} range_i^2}$，其中 ~~$range_i$是 i 个分段~~中两个断点之间的距离。它对应于查询和最近邻序列分别位于每个分段相对边界时的距离上 。

在图13中，我们用直方图展示了 Dumpy 和 iSAX 索引的叶节点上界距离分布。直方图显示，超过**80%** 的 iSAX 叶节点具有较宽松的界限（60-64），而**60%** 以上的 Dumpy 叶节点具有较紧的界限（40-60）。这是因为 Dumpy 选择分裂较粗的分段，因此其最坏情况距离比 iSAX 更小。

[效率与准确率] 我们将近似搜索扩展至所有叶节点，并采用下界剪枝，以评估索引在整个 MAP 范围内的响应时间。结果

如图14所示。得益于邻近性高的节点和紧凑的索引结构，Dumpy 在低精度和高精度区间均优于竞争对手。

[在 DTW 下搜索] 在此实验中，我们在图15中比较了动态时间规整距离下的准确率。由于其固有的难度，整体而言，精度低于欧氏距离。不过，Dumpy 和 Dumpy-f在动态时间规整下仍能取得更高的精度和更低的误差率。由于动态时间规整的绝对距离小于欧氏距离，各方法之间的误差率差异往往较小（TARDIS 除外）。

[参数的影响]我们还研究了参数对准确率的影响，包括分段数量 w、目标函数方程 1 中的权重因子 α，以及 Dumpy-Fuzzy 的边界比率 f。

在图 16(a)中，我们改变 iSAX 和 Dumpy 的 w，并观察到 iSAX 的最佳表现（当 $w = 16$ 时）仍不如 Dumpy（无论分段数量为 $w \in \{12, 16, 20\}$）。这是因为 Dumpy 采用了一种策略，搜索质量更高的分裂，从而充分发挥可用分段的 iSAX 摘要能力。我们还观察到，在这两种索引中，分段较少会降低 iSAX 摘要的精度，而分段较多则会产生大量叶节点，从而损害索引性能。

在图16(b)中，当 α 从 0 增加到 0.3 时，填充因子也持续上升，从 59% 增至 75%。精确率起初会上升，因为访问节点中的序列更多；随后则因节点内邻近度降低而下降。如图5所示，当 α 减小时，划分方案趋向于图5(c)所示的形式，这意味着节点中的序列彼此不那么相似。 α 约为 0.2 时，精确率达到最佳；我们的实验在不同数据集上均采用这一数值，表现出良好的稳健性。

其次，我们评估了 Dumpy-Fuzzy 在不同 f（即模糊边界比率）下的稳定性。从图17(a)中可以看出，随着 f 增加，叶节点数量和构建时间缓慢增加。在图17(b)（$k = 5$）中，随着 f 增加至**10%**，精确率提高，误差率降低，之后则保持相对稳定。这表明 Dumpy-Fuzzy 在 f 方面具有稳定性；对于心电图和 Deep，选择 f=10，对于 Rand 和 DNA，则选择 f=30。

7.2.2 精确搜索。我们在表2中评估了 Dumpy 相较于其他方法的精确搜索效率。由于 TARDIS 在原论文中不支持精确 k近邻搜索，我们实现了一种与 iSAX 索引系列类似的算法。但搜索过程中只有由 iSAX词概括的节点才能被剪枝。结果采用 40 个查询在 $k = 50$下基于欧氏距离和动态时间规整所得的平均值报告。总体而言，Dumpy 在所有情况下都实现了最高效率。值得注意的是，尽管 DSTree 的剪枝率高于 Dumpy，其响应时间仍比 Dumpy 慢。原因如下：DSTree 需要频繁计算标准差，因此计算距离下界耗时更长。iSAX2+受低填充因子的影响，需要读取的节点数量约为 Dumpy 和 DSTree 的3倍。

7.3 完整工作负载

最后，我们比较了插入新数据序列时的不同方法（图18）。由于 TARDIS 是为静态数据集设计的，且不易扩展，因此我们不将其纳入比较。为确保公平，我们使用单线程实现所有方法（尽管 Dumpy 支持多线程）。我们采用不同的合成工作负载，其中包含 100 次精确查询和总计 1 亿个序列，查询之间穿插着批量插入操作。结果表明，得益于紧凑的结构，Dumpy 在所有工作负载下都优于其他方法。尽管重新拆分和重新打包过程会带来额外开销，但这部分开销可由这两种设计所带来的效率提升抵消。此外，当初始

表2. 精确搜索。对于响应时间，括号中的第一个数字是 I/O时间，第二个数字是 CPU 时间。

	方法	响应时间（秒）	#已加载节点	剪枝率
Rand-ED	iSAX2+	65 (50+15)	7595	81.51%
	DSTree	33 (20+13)	2027	86.06%
	TARDIS	53 (15+38)	1665744	59.30%
	Dumpy	**17** (**12**+5)	2641	83.70%
Rand-DTW	iSAX2+	151 (85+66)	18660	72.58%
	DSTree	79 (24+65)	3678	75.12%
	TARDIS	208 (22+186)	2846868	49.24%
	Dumpy	**58** (**18**+**40**)	3997	73.61%
DNA-ED	iSAX2+	42 (26+16)	1077	91.04%
	DSTree	21 (16+5)	326	94.00%
	TARDIS	40 (11+29)	161959	71.64%
	Dumpy	**12** (**10**+2)	433	91.69%
DNA-DTW	iSAX2+	116 (60+56)	2163	87.77%
	DSTree	63 (18+45)	497	90.93%
	TARDIS	143 (16+127)	194645	68.78%
	Dumpy	**53** (**14**+**39**)	528	89.41%

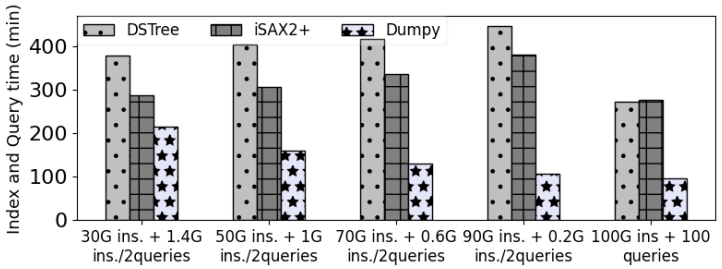


图18. 更新性能（4GB内存）

批次大小增大时，Dumpy 的性能会更好（而 iSAX 和 DSTree 的性能则会变差），因为插入操作越少，重新拆分和重新打包的次数也越少。

8 结论与未来工作

我们提出了一种新颖的多叉数据序列索引 Dumpy，它采用自适应分裂策略，在邻近性与紧凑性权衡之间实现了恰当的平衡。Dumpy 的一个变体 Dumpy-Fuzzy 可通过巧妙的部分复制实现更高的准确率。针对多种大型合成数据集和真实数据集开展的实验表明，我们的解决方案具有良好的效率、可扩展性和准确率。

未来，我们计划扩展 Dumpy，使其支持子序列匹配。通过设计合适的代价函数和高效的评估算法，Dumpy 的自适应分裂策略可以增强最先进的子序列匹配索引 ULISSE [39], , 使其节点具有更高的邻近性，从而提升性能。此外，借鉴 PARIS [50], 、SING [52] 和 TARDIS [68], 的并行范式，并利用现代硬件，还可以进一步提升 Dumpy 的整体性能。

致谢

本研究得到中国科学技术部国家重点研发计划（编号：2020YFB1710001）的资助。Qitong Wang 得到国家留学基金管理委员会的资助。我们还要感谢 Karima Echihabi 教授慷慨分享 iSAX2+和 DSTree 的优化实现。

参考文献

[1] [无日期]。美国国家生物技术信息中心（NCBI）[互联网]。https://www.ncbi.nlm.nih.gov/ 访问日期：2022年3月14日。[2] Rakesh Agrawal、Christos Faloutsos 和 Arun Swami。1993年。序列数据库中的高效相似性搜索。载于《国际数据组织与算法基础会议》。Springer，69–84。[3] Panagiotis Anagnostou、Petros Barbas、Aristidis G. Vrahatis 和 Sotiris K. Tasoulis。2020年。生物医学数据的近似 k 近邻分类。载于《2020 IEEE 国际大数据会议（大数据）》。3602–3607。https: //doi.org/ 10 1109 50022 2020 9378126 . /BigData . . Akhil Arora、Sakshi Sinha、Piyush Kumar 和 [4] Arnab Bhattacharya。2018年。HD-Index：突破高维空间中近似 k 近邻搜索的可扩展性与准确率边界。《VLDB 汇刊》11卷8期（2018）。[5] Ilias Azizi、Karima Echihabi 和 Themis Palpanas。2023年。ELPIS：面向可扩展数据科学的基于图的相似性搜索。《VLDB 汇刊》16卷6期（2023）。[6] Artem Babenko 和 Victor Lempitsky。2014年。倒排多索引。《IEEE 模式分析与机器智能汇刊》37卷6期（2014），1247–1260。[7] Artem Babenko 和 Victor Lempitsky。2016年。十亿级深度描述符数据集的高效索引。载于《IEEE 计算机视觉与模式识别会议论文集》。2055–2063。[8] Anthony J. Bagnall、Richard L. Cole、Themis Palpanas 和 Konstantinos Zoumpatianos。2019年。数据序列管理（Dagstuhl 研讨会 19282 9, 7 2019 24 39 10 4230 9 7 24 ）。《 [9] 1997 Dagstuhl 报告》（），–。https://doi.org/ . /DagRep. . J.S. Beis 和 D.G. Lowe。 . 利用高维空间中的近似最近邻搜索进行形状索引。载于《IEEE 计算机学会计算机视觉与模式识别会议论文集》。IEEE，圣胡安，波多黎各，美国，1000–1006。[10] Paul Boniol、Michele Linardi、Federico Roncallo 和 Themis Palpanas。2020年。大型序列中的自动异常检测。载于《2020 IEEE 第36届国际数据工程会议（ICDE）》。IEEE，1834–1837。[11] Paul Boniol 和 Themis Palpanas。2020年。Series2graph：面向时间序列的基于图的子序列异常检测。《VLDB 汇刊》13卷12期（2020），1821–1834。[12] Alessandro Camerra、Themis Palpanas、Jin Shieh 和 Eamonn Keogh。2010年。iSAX 2.0：索引和挖掘十亿条时间序列。载于《2010 IEEE 国际数据挖掘会议》。IEEE，58–67。[13] Alessandro Camerra、Jin Shieh、Themis Palpanas、Thanawin Rakthanmanon 和 Eamonn Keogh。2014年。超越十亿条时间序列：使用 i sax2+对超大型时间序列集合进行索引和挖掘。《Knowledge and Information Systems》39卷1期（2014），123–151。[14] Manos Chatzakis、Panagiota Fatourou、Eleftherios Kosmas、Themis Palpanas 和 Botao Peng。2023年。Odyssey：分布式数据序列相似性搜索之旅。《VLDB 汇刊》（2023）。[15] George Chen、Christina Lee 和 Shah Devavrat。2017年。海量数据现代应用的最近邻。https://nn2017.mit.edu/。载于《第31届国际神经信息处理系统会议研讨会论文集》。[16] George H. Chen 和 Devavrat Shah。2018年。阐释最近邻方法在预测中的成功。《机器学习基础与趋势》 10, 5 6 2018 337 588 10 1561 2200000064 - （），–。 [17] https://doi.org/ . / Qi Chen、Bing Zhao、Haidong Wang、Mingqin Li、Chuanjie Liu、Zhiyong Zheng、Mao Yang 和 Jingdong Wang。2021年。SPANN：高效的十亿级近似最近邻搜索。《神经信息处理系统进展》34（2021）。[18] Hui Ding、Goce Trajcevski、Peter Scheuermann、Xiaoyue Wang 和 Eamonn Keogh。2008年。时间序列数据的查询与挖掘：表示方法和距离度量的实验比较。《VLDB 汇刊》1卷2期（2008），1542–1552。[19] Karima Echihabi、Panagiota Fatourou、Kostas Zoumpatianos、Themis Palpanas 和 Houda Benbrahim。2022年。Hercules Against Data Series Similarity Search。《VLDB 汇刊》15卷10期（2022），2005–2018。[20] Karima Echihabi、Theophanis Tsandilas、Anna Gogolou、Anastasia Bezerianos 和 Themis Palpanas。2023年。ProS：具有概率质量保证的数据序列渐进式 k 近邻相似性搜索与分类。《VLDBJ》（2023）。[21] Karima Echihabi、Kostas Zoumpatianos、Themis Palpanas 和 Houda Benbrahim。2018年。数据序列相似性搜索的勒拿九头蛇：对最先进水平的实验评估。《VLDB 汇刊》12卷2期（2018年10月），112–127。[22] Karima Echihabi、Kostas Zoumpatianos、Themis Palpanas 和 Houda Benbrahim。2019年。勒拿九头蛇归来：数据序列近似相似性搜索的实验评估。《VLDB 汇刊》13卷3期（2019年11月），403–420。[23] Cong Fu、Changxu Wang 和 Deng Cai。2021年。利用卫星系统图进行高维相似性搜索：效率、可扩展性与未索引查询兼容性。《IEEE 模式分析与机器智能汇刊》（2021），1–1。[24] Cong Fu、Chao Xiang、Changxu Wang 和 Deng Cai。2019年。利用导航扩展图进行快速近似最近邻搜索。《VLDB 汇刊》12卷5期（2019年1月），461–474。

[25] Tiezheng Ge、Kaiming He、Qifa Ke 和 Jian Sun。2013。优化乘积量化。IEEE 模式分析与机器智能汇刊 36, 4 (2013), 744–755。[26] Long Gong、Huayi Wang、Mitsunori Ogihara 和 Jun Xu。2020。iDEC：用于近似最近邻搜索的可索引距离估计码。VLDB 汇刊 13, 9 (2020)。[27] Qiang Huang、Jianlin Feng、Yikai Zhang、Qiong Fang 和 Wilfred Ng。2015。用于近似最近邻搜索的查询感知局部敏感哈希。VLDB 汇刊 9, 1 (2015), 1–12。[28] Herve Jegou、Matthijs Douze 和 Cordelia Schmid。2010。用于最近邻搜索的乘积量化。IEEE 模式分析与机器智能汇刊 33, 1 (2010), 117–128。[29] Jaemin Jo、Jinwook Seo 和 Jean-Daniel Fekete。2020。PANENE：用于近似k近邻索引与查询的渐进算法。IEEE Trans. Vis. Comput. Graph. 26, 2 (2020), 1347–1360。[30] Alistair EW Johnson、Tom J Pollard、Lu Shen、Li-wei H Lehman、Mengling Feng、Mohammad Ghassemi、Benjamin Moody、Peter Szolovits、Leo Anthony Celi 和 Roger G Mark。2016。MIMIC-III：一个免费开放的重症监护数据库。Scientific data 3, 1 (2016), 1–9。[31] Eamonn Keogh。2006。大型时间序列数据库索引与挖掘的十年进展。载于《第32届超大型数据库国际会议论文集》。1268–1268。[32] Eamonn Keogh、Kaushik Chakrabarti、Michael Pazzani 和 Sharad Mehrotra。2001。大型时间序列数据库中用于快速相似性搜索的降维。Knowledge and Information Systems 3, 3 (2001), 263–286。[33] Haridimos Kondylakis、Niv Dayan、Kostas Zoumpatianos 和 Themis Palpanas。2018。Coconut：构建数据序列索引的可扩展自底向上方法。VLDB 汇刊 11, 6（2018年2月），677–690。[34] Haridimos Kondylakis、Niv Dayan、Kostas Zoumpatianos 和 Themis Palpanas。2019。Coconut：适用于静态与流式数据序列可扩展索引的可排序摘要。VLDB J. 28, 6 (2019), 847–869。https://doi.org/10.1007/s00778- 019-00573-w[35] Flip Korn、Bernd-Uwe Pagel 和 Christos Faloutsos。2001。论“维度灾难”与“自相似性福祉”。IEEE Trans. Knowl. Data Eng. 13, 1 2001 96 111 10 1109 69 908983 ( ), – 。https://doi.org/ . / . [36] Oleksandra Levchenko 、 Djamel-Edine Yagoubi 、 Reza Akbarinia 、 Florent Masseglia、Boyan Kolev 和 Dennis Shasha。2018。Spark-parsketch：大规模分布式时间序列数据集索引。载于《第27届ACM信息与知识管理国际会议论文集》。1951–1954。[37] Wen Li、Ying Zhang、Yifang Sun、WEI WANG、Mingjie Li、Wenjie Zhang 和 Xuemin Lin。2019。高维数据上的近似最近邻搜索——实验、分析与改进。IEEE知识与数据工程汇刊 32, 8 (2019), 1475–1488。[38] Michele Linardi 和 Themis Palpanas。2018。数据序列中的可扩展可变长度相似性搜索：ULISSE方法。VLDB 汇刊 11, 13 2018 2236 2248 10 14778 3275366 3284968 ( ), – 。https://doi.org/ . / . [39] Michele Linardi 和 Themis Palpanas。2020。数据序列子序列匹配的可扩展方法：ULISSE。VLDB J. 29, 6 (2020), 1449–1474。[40] Wanqi Liu、Hanchen Wang、Ying Zhang、WEI WANG、Lu Qin 和 Xuemin Lin。2021。EI-LSH：由提前终止驱动的 I/O高效增量式c-近似最近邻搜索。The VLDB Journal 30, 2 (2021), 215–235。[41] Yury Malkov、Alexander Ponomarenko、Andrey Logvinov 和 Vladimir Krylov。2014。一种基于可导航小世界图的近似最近邻算法。Information Systems 45 (2014), 61–68。[42] Yu A Malkov 和 Dmitry A Yashunin。2018。使用分层可导航小世界图进行高效且稳健的近似最近邻搜索。IEEE 模式分析与机器智能汇刊 42, 4 (2018), 824–836。[43] Themis Palpanas。2015。数据序列管理：迈向大规模序列分析之路。ACM SIGMOD Record 44, 2 (2015), 47–52。[44] Themis Palpanas。2016。大规模序列管理：过去、现在与未来一瞥。载于《SOFSEM 2016：计算机科学理论与实践——第42届计算机科学理论与实践当前趋势国际会议》（Lecture Notes in Computer Science，第9587卷）。63–80。[45] Themis Palpanas。2020。数据序列索引的演进。载于《信息检索、集成与个性化》。Springer International Publishing，Cham，68–83。[46] Themis Palpanas 和 Volker Beckmann。2019。第一届和第二届跨学科时间序列分析研讨会（ITISA）报告。SIGMOD Rec. 48, 3 2019 36 40 10 1145 3377391 3377400 ( ), – 。https://doi.org/ . / . [47] John Paparrizos、Ikraduya Edian、Chunwei Liu、Aaron J. Elmore 和 Michael J. Franklin。2022。快速自适应相似性搜索：方差感知量化方法。载于《2022 IEEE 第38届国际数据工程会议（ICDE）》。IEEE。[48] Botao Peng、Panagiota Fatourou 和 Themis Palpanas。2018。Paris：快速数据序列索引与查询应答的新方向。载于《2018 IEEE国际大数据会议（Big Data）》。IEEE，791–800。[49] Botao Peng、Panagiota Fatourou 和 Themis Palpanas。2020。Messi：内存数据序列索引。载于《2020 IEEE 第36届国际数据工程会议（ICDE）》。IEEE，337–348。

[50] Botao Peng、Panagiota Fatourou 和 Themis Palpanas。2020。Paris+：多核架构上的数据序列索引。IEEE知识与数据工程汇刊 33, 5 (2020), 2151–2164。[51] Botao Peng、Panagiota Fatourou 和 Themis Palpanas。2021。面向内存数据的快速数据序列索引。VLDB J. 30, 6 (2021), 1041–1067。[52] Botao Peng、Panagiota Fatourou 和 Themis Palpanas。2021。SING：使用GPU的序列索引。载于《2021 IEEE 第37届国际数据工程会议（ICDE）》。IEEE，1883–1888。[53] Thanawin Rakthanmanon、Bilson Campana、Abdullah Mueen、Gustavo Batista、Brandon Westover、Qiang Zhu、Jesin Zakaria 和 Eamonn Keogh。2012。在动态时间规整下搜索和挖掘数万亿条时间序列子序列。载于《第18届ACM SIGKDD知识发现与数据挖掘国际会议论文集（KDD ’ *12* 262 270 10 1145 2339530 2339576 ）》。– 。https://doi.org/ . / . Erich Schubert、[54] 2015
Arthur Zimek 和 Hans-Peter Kriegel。使用近似最近邻集成进行快速且可扩展的离群点检测。载于《高级应用数据库系统》。19–36。[55] Claude Elwood Shannon。1948。通信的数学理论。贝尔系统技术期刊 27, 3 (1948), 379–423。[56] Jin Shieh 和 Eamonn Keogh。2008。iSAX：太字节规模时间序列的索引与挖掘。载于《第14届ACM SIGKDD知识发现与数据挖掘国际会议论文集》。623–631。[57] Suhas Jayaram Subramanya、Rohan Kadekodi、Ravishankar Krishaswamy 和 Harsha Vardhan Simhadri。2019。Diskann：在单个节点上快速、精准地搜索十亿级数据点的最近邻。载于《第33届神经信息处理系统国际会议论文集》。13766–13776。[58] Yifang Sun、WEI WANG、Jianbin Qin、Ying Zhang 和 Xuemin Lin。2014。SRS：在高维欧几里得空间中使用微型索引解决 *c*-近似最近邻查询。VLDB 汇刊 8, 1（2014年9月），1–12。[59] Chang Wei Tan、Geoffrey I Webb 和 François Petitjean。2017。在时间扭曲条件下索引和分类数GB时间序列。载于《2017年SIAM国际数据挖掘会议论文集》。SIAM，282–290。[60] Andrew Turpin 和 Falk Scholer。2006。简单搜索任务中用户表现与精确率指标的比较。载于《第29届ACM SIGIR信息检索研究与发展年度国际会议论文集》。11–18。[61] Skoltech计算机视觉。 [未注明日期。]。十亿级深度索引。http://sites.skoltech.ru/compvision/noimi 访问日期：2022年3月14日。[62] Mengzhao Wang、Xiaoliang Xu、Qiang Yue 和 Yuxiang Wang。2021。基于图的近似最近邻搜索：综合综述与实验比较。arXiv预印本 arXiv:2101.12631 (2021)。[63] QITONG WANG 和 Themis Palpanas。2021。用于数据序列相似性搜索的深度学习嵌入（KDD ’21）。ACM，美国纽约， 1708 1716 10 1145 3447548 3467317 – 。[64]
https://doi.org/ . / . Qitong Wang、Stephen Whitmarsh、Vincent Navarro 和 Themis Palpanas。2022。iEDeaL：用于检测高度不平衡发作间期癫痫样放电的深度学习框架。VLDB 汇刊 16, 3 (2022), 480–490。[65] Yang Wang、Peng Wang、Jian Pei、WEI WANG 和 Sheng Huang。2013。一种用于时间序列整体匹配的数据自适应动态分段索引。VLDB 汇刊 6, 10 (2013), 793–804。[66] Djamel Edine Yagoubi、Reza Akbarinia、Florent Masseglia 和 Themis Palpanas。2017。DPiSAX：大规模分布式分区iSAX。载于《2017 IEEE国际数据挖掘会议（ICDM）》。IEEE，1135–1140。[67] Djamel Edine Yagoubi、Reza Akbarinia、Florent Masseglia 和 Themis Palpanas。2020。大规模分布式时间序列索引与查询。IEEE Trans. Knowl. Data Eng. 32, 1 2020 108 120 10 1109 2018 2880215 ( ), – 。[68] 2019
https://doi.org/ . /TKDE. . Liang Zhang、Noura Alghamdi、Mohamed Y Eltabakh 和 Elke A Rundensteiner。TARDIS：大规模时间序列数据的分布式索引框架。载于《2019 IEEE 第35届国际数据工程会议（ICDE）》。IEEE，1202–1213。[69] Kang Zhao、Liuyihan Song、Yingya Zhang、Pan Pan、Yinghui Xu 和 Rong Jin。2021。ANN Softmax：加速极端分类训练。VLDB 汇刊 15, 1 2021 1 10 10 14778 3485450 3485451 ( ), – 。[70] 2014
https://doi.org/ . / . Kostas Zoumpatianos、Stratos Idreos 和 Themis Palpanas。面向大数据序列交互式探索的索引。载于《2014年ACM SIGMOD数据管理国际会议论文集》。1555–1566。[71] Kostas Zoumpatianos、Stratos Idreos 和 Themis Palpanas。2016。ADS：自适应数据序列索引。The VLDB Journal 25, 6 (2016), 843–866。[72] Kostas Zoumpatianos、Yin Lou、Ioana Ileana、Themis Palpanas 和 Johannes Gehrke。2018。生成数据序列查询工作负载。VLDB J. 27, 6 (2018), 823–846。https://doi.org/10.1007/s00778-018-0513-x [73] Kostas Zoumpatianos、Yin Lou、Themis Palpanas 和 Johannes Gehrke。2015。数据序列索引的查询工作负载。载于《第21届ACM SIGKDD知识发现与数据挖掘国际会议论文集》。1603–1612。[74] Kostas Zoumpatianos 和 Themis Palpanas。2018。数据序列管理：满足大规模序列分析的需求。载于《2018 IEEE 第34届国际数据工程会议（ICDE）》。IEEE，1677–1678。

收稿于2022年7月；2022年10月修订；2022年11月录用