arXiv:2601.00695v1 [cs.DB] 2 Jan 2026

DeXOR：在十进制空间中实现异或，以对浮点数据进行流式无损压缩

Chuanyi Lv 浙江大学，中国
chuanyi.lv@zju.edu.cn

Huan Li 浙江大学，中国
lihuan.cs@zju.edu.cn

Dingyu Yang 浙江大学，中国
yangdingyu@zju.edu.cn

Zhongle Xie 浙江大学，中国
xiezl@zju.edu.cn

Lu Chen 浙江大学，中国
luchen@zju.edu.cn

Christian S. Jensen 奥尔堡大学，丹麦
csj@cs.aau.dk

摘要

随着流式浮点数日益普及，对此类数据进行有效且高效的压缩至关重要。压缩方案必须能够利用连续数值之间的相似性或平滑性，并能够应对极端条件，例如高精度值或缺乏平滑性。我们提出了 DeXOR，这是一种新颖的框架，可通过十进制异或过程对十进制空间中的最长公共前缀和后缀进行编码，从而实现最优的前缀复用和有效的冗余消除。为确保即使存在二进制-十进制转换误差，也能准确且低成本地解压缩，DeXOR 引入了 1）带容错舍入的缩放截断，以及 2）针对十进制异或优化的不同位管理策略。此外，鲁棒的异常处理器通过管理浮点指数来增强稳定性，从而在极端条件下保持较高的压缩率。在对 22 个数据集进行的评估中，DeXOR 超越了最先进的方案，压缩率提高了 15%，解压速度提升了 20%，同时保持了有竞争力的压缩速度。DeXOR 还能够适应各种条件，并在其他方案失效的极端场景中表现出鲁棒性。

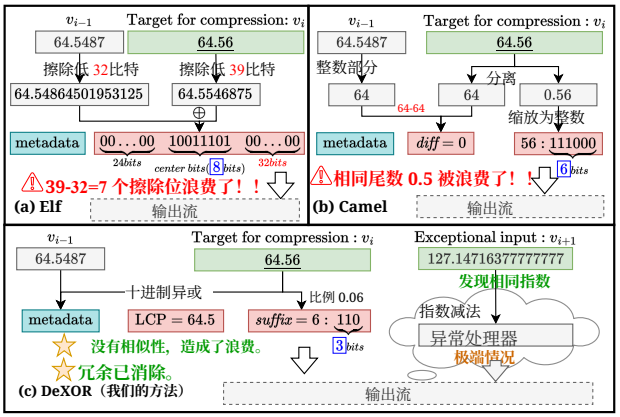

图 1：利用前一个值64.56 压缩目标值64.5487：(a) Elf [30, 32] 会移除二进制表示中的尾随零，但在异或后浪费比特。(b) Camel [45]针对十进制值的整数部分进行处理，未优化小数部分。(c) DeXOR 提取最长公共十进制前缀（64.5）以利用平滑性，并对后缀进行缩放（从0.06 到6）以消除冗余性。它还针对极端情况下的目标值（127.14716377···），加入了基于指数减法的异常处理器。

1 引言

快速发展的物联网 [28] 会产生海量流式数据，通常以时间序列的形式呈现。例如，航空系统每秒会产生数 TB 的数据 [23]。在工业自动化、金融和科学研究等诸多领域，高精度浮点数据流都需要准确解压缩，因为即便是微小误差也可能造成严重的运行后果。因此，兼具实时效率的无损压缩是一项根本且迫切的挑战。

为应对这一挑战，流式无损压缩 (SLC) 已成为一项重要的研究重点。本研究旨在解决

一种常见的设置是，只需前一个值即可压缩当前值（图1），适用于内存限制严格的场景，或无法保留较早值的流式应用。现有的SLC 方法分为两类：基于平滑性的和基于冗余的方案。基于平滑性的方案（如Gorilla [38] 和Chimp [33],利用连续值之间的相似性（称为平滑性），通过位运算（例如异或）进行压缩。尽管这些方案对序列模式效果良好，但难以处理浮点数所采用的复杂存储格式。相比之下，基于冗余的方案针对浮点数结构中的表示冗余，通过引入前导零或尾随零来减少存储位数。典型示例包括缩放为整数的方案（PDE [27] 和 ALP [8]）。然而，这类方案往往无法利用数据固有的平滑性，因此压缩率较低。

近期提出的方案，包括 Elf 系列 [29–32] 和 Camel [45]，试图结合基于平滑性和基于冗余的方案（见图 1(a) 和 (b)）。然而，这种整合会造成干扰：冗余消除涉及格式转换，破坏了平滑性。例如，在 Elf 系列中，尾数擦除会使精度不同的二进制表示错位，从而破坏平滑性。在图 1(a) 中，64.56 擦除了 39 个尾随零，而 64.5487 擦除了 32 个；异或后，64.56 又多丢失了 7 个尾随零，导致压缩后的输出中出现冗余。接下来，图 1(b) 中介绍的 Camel [45], 将整数和小数部分分开，对整数部分的相似性进行了部分利用（64 – 64），同时通过缩放减少小数部分的冗余（将 0.56 转换为 56，以整数表示）。

然而，该方案没有利用小数部分的平滑性，因而限制了其压缩率。

不同于受二进制系统约束的方案，Camel 开创了十进制域中的平滑度利用，启发我们设计了DeXOR，这是一个统一框架，通过十进制空间转换协同融合平滑度利用与基于冗余的技术。其核心是十进制异或，可将二进制浮点值转换为与十进制对齐的值，分离出最大的公共前缀以利用平滑性，同时保留残差以进行冗余消除。以图64。56 中的目标值1(c) 为例。使用十进制异或，可以识别并移除与前一个值共享的**最长公共前缀（LCP）**，例如*64.5* 是64。56 与64。5487 之间的公共部分。随后将残余值（0.06）缩放为整数后缀（6），从而实现最优冗余消除。

虽然十进制异或在概念上很简单，但在 SLC工作流中处理它面临三个关键挑战：

- **C1（无损重构）**：二进制到十进制转换会带来精度损失风险（见第2.2节），影响后缀提取的精确性，并破坏无损重构的保证。
- **C2（计算开销）**：为保证平滑性而进行的前缀提取，以及用于冗余消除的后缀处理，都会带来额外的计算和元数据开销，增加时间复杂度与存储需求。
- **C3（极端情况下的鲁棒性）**：对于精度较高、非平滑且冗余性较低的数据集，十进制异或的优势可能减弱，反而增加时间或空间开销。

解决这些挑战至关重要。DeXOR 提供了一系列复杂的优化方法。

为应对挑战 C1，DeXOR 采用缩放截断方法与容错舍入（第4.2.1节），确保准确识别前缀和后缀。不同于传统方案将二进制格式转换为十进制格式以进行字符串匹配，DeXOR 直接从原始二进制格式中计算前缀和后缀。此外，该方法引入了容错舍入机制以降低边界情况风险，确保十进制异或运算鲁棒且无损。这消除了由转换引起的前缀和后缀识别错误，同时进一步降低了计算开销。

为应对挑战 C2，DeXOR 提供了多种十进制异或专用位管理策略：（i）DeXOR 通过使用最少的坐标数据（即图1（c）中的元数据）编码最长公共前缀和后缀，从而最大限度地降低前缀存储复杂度；轻量级的 2 位情况编码还进一步实现了对这些坐标结果的高效复用（第4.2.2节）。（ii）通过识别后缀与前缀之间的符号一致性，并评估变长后缀存储的低效性，DeXOR 采用了最优的定长无符号后缀编码设计（第4.3节）。

为应对挑战 C3，DeXOR 加入了一个异常处理器，与十进制异或驱动的主流水线并行运行，专门用于处理高精度或非平滑数据流。该处理器会分离浮点指数并对指数减法进行编码（见图1(c)），利用数据的平滑性，同时跳过收益甚微的数据成分（第5.1节）。鉴于指数减法具有范围有界、低波动性和低位密度的特点，DeXOR引入紧凑自适应长度编码，以高效存储关键信息（第5.2节）。对于已知具有特殊属性的数据集，该处理器也可作为独立压缩器使用。

我们使用 22 个公开可用的数据集对 DeXOR 进行了评估，这些数据集常用于先进研究 [30, 32, 33, 38, 45]。在所有指标上，DeXOR 均表现出领先优势：与性能最佳的竞争方案相比，压缩率提高了 15%，解压缩速度提升了 20%，同时保持了相当的压缩速度。在低精度数据集上，DeXOR 的解压缩速度提升了 30%，压缩速度提高了 20+%。它在几乎所有数据集上都实现了最高压缩率，并展现出稳健的性能；即使在其他方案失效的极端情况下，也能保持出色的压缩效果。稳定实现 15%–20% 的压缩增益意义重大，可在存储成本 [38],网络带宽 [18], 查询性能 [7, 14], 和系统可持续性 [36, 43], 等方面带来收益，从而产生显著效益。

主要贡献总结如下。

- 我们重新审视现有的SLC方案，并深入探讨如何利用十进制空间操作发挥平滑性和冗余性的作用。基于这些见解，我们提出DeXOR，这是一种全面的SLC 方案，以十进制异或为核心（第2 节和第3节）。
- 我们为DeXOR的主流水线配备了十进制异或专用优化，包括采用带容错舍入的缩放截断，以高效、精确地进行前缀-后缀提取，以及采用不同的位管理策略，确保数据和元数据得以紧凑存储（第4节）。
- 我们提出了一种异常处理器，以弥补十进制异或在极端场景下的不足，包括涉及高精度和非平滑数据流的情况。该处理器能够稳定指数并采用自适应编码，即使在其他方案表现不佳时，也能保持较高的压缩率（第5节）。
- 我们报告了针对相关数据集和基线方法开展的大量实验，结果表明，DeXOR 能够实现更高的压缩率和更快的解压缩速度，在不同条件下具有良好的可扩展性，并能在极端场景中保持鲁棒性（第6节）。

2 预备知识

2.1 数值数据表示

在实际应用中，数值通常表示为浮点数，而不是整数，其中**IEEE 754 标准** [4] 是事实上的格式。此外，计算机系统必须使用二进制表示ω_i存储每个数值 v_i，最常见的是采用 64位双精度形式：

$$f_{\text{IEEE754}} : v_i \rightarrow \omega_i = \underbrace{sign}_{1\ bit} \ || \ \underbrace{exp}_{11\ bits} \ || \ \underbrace{fraction}_{52\ bits}. \tag{1}$$

原始十进制值 v_i 可以重构为：

$$v_i \approx sign \times 2^{(exp-bias)} \times (1 + fraction), \tag{2}$$

其中，符号 表示正负；指数 是存储指数；偏置 是预定义的偏移量，可将负指数映射为非负值以实现紧凑存储（实际指数为指数 – 偏置）；而尾数 以分数形式表示有效数，与 1 相加后构成完整的有效数值。这种格式可确保在不同平台上实现紧凑且高效的表示。本研究

表1：转换器中各种方法的示例（假设前一个值 $v_1 = 88.1537$ 和当前值 $v_2 = 88.1479$）：前导零、中心位和尾随零以高亮显示。开小数分行映射操作，因此产生 4 比特的索引开销，即（索引 = 0010）（见表1第 3 行）。

Type	方法	转换	二进制表示（转换后）	CBL
-	原始 IEEE 754	ω(88.1479)	0 1000000010101100000100101110111001100011000111111000101010000101	63
①	异或 (Gorilla [38] 和Chimp [33])	ω(88.1479) $\oplus$ ω(88.1537)	00000000000000000000000 10100001000010010000100110011100010011 0	39
	小数分离 (Camel [45])	(88.1479−88) %2^{-4}×10^4	00000000000000000000000000000000000000000000000000000000 11100101 ⟹ (索引=0010)	12
②	擦除（Elf [32]）	88.147**88818359375**	0 1000000010101100000100101110111 00000000000000000000000000000000	31
	缩放为整数（ALP [8]）	88.1479 ×10^6 × 10^{-2} = 881479	00000000000000000000000000000000000000000000000000 11011110111111100111	20
③	十进制异或（**我们的**）	(88.1479 ⋄ 88.1537) = 479	00000000000000000000000000000000000000000000000000000000000000000000	9

默认采用双精度（其中 偏置 = 1023 见公式 2），因为其精度更高。尽管如此，所提出的压缩技术也可直接应用于单精度值，其格式类似（8位 指数 和 23位 尾数），但 偏置 = 127。

二进制与十进制转换中的舍入误差。IEEE 754 中有限的比特数会在公式 2 所示的二进制-十进制转换过程中引入舍入误差。例如，存储十进制值 $v_i = 88.1479$ 后，可能会将其重建为 $v_i' = 88.14790000000000702\cdots$，从而产生可忽略不计的误差（例如，对于 64 位精度，舍入误差小于 $2^{(exp-bias)-52}$）。虽然舍入误差通常微不足道，但在涉及二进制-十进制转换的典型压缩方案中，舍入误差可能会不断累积 [8, 27, 45]；因此，需要谨慎处理以确保精度。我们将在第 4.2 节讨论如何应对这一问题。

2.2 问题表述

遵循先前的研究 [30, 32, 33, 38, 45],，我们假设压缩系统会处理一个由连续输入的数值构成的数据流，记为 v_1，v_2，$\cdots$。此类数据流可能源自对现有时间序列的订阅，也可能包含由多个来源发出的随机数据。在我们的框架中，如果存在时间戳，则不予考虑，因为通常会使用专门的压缩技术（例如 *Delta-of-delta* 压缩 [38]）单独处理时间戳。相反，我们只关注数值的压缩。如前所述，此类输入数据通常表示为比特流，定义如下。

定义 1（比特流）。输入比特流表示为 $\omega_1\omega_2\cdots$，其中每个 ω_i 都是十进制空间中数值 v_i 的 *IEEE 754* 二进制表示。

我们将问题定义如下。

问题陈述（流式无损压缩，SLC）。给定一个输入比特流，在不假设时间平滑性的前提下，流式压缩是一个函数C：$\omega_{i-N}\cdots\omega_i \to \omega_i'$，它使用由 N个先前值构成的缓冲区，将目标值 ω_i转换为压缩二进制表示 ω_i'。流式无损压缩确保存在反向映射$\mathcal{D}$：$\omega_{i-N}'\cdots\omega_i' \to \omega_i$，从而能够从压缩比特流中精确恢复$\omega_i$（以及其前面的 N个值ω_i'）。

我们的问题定义采用单变量流式设置，因此与当前主流的最先进方法 [8, 32, 33, 38]保持一致。对于高维数据集（例如向量数据），每个维度都可以独立压缩，并分配给单独的计算线程，从而轻松实现并行化。

本研究探讨 SLC 在 $N = 1$下的情况，其中压缩和解压缩只涉及前一个值。这种极简上下文使得实现有效压缩更具挑战性。当 $N > 1$时，更大的缓冲区可支持使用滑动窗口（利用当前值之前的 N个值进行压缩）或截断窗口（在小批量中压缩 N个数据）。如第 6.6节所示，我们提出的方案在压缩率和/或效率方面优于使用更大缓冲区的方案。

2.3 重新思考现有的 SLC 方案

在数值数据压缩中，**变长编码** [38]通过移除因固定分配而产生的不必要比特来减少存储空间，例如**前导**或**尾随零**。参见表1中的示例，v_2的 IEEE 754 表示 ω（88.1479）有一个前导零，可以省略。剩余的必要比特，即**中心位**，以一个1 比特开头并结尾。

□

为了重构原始值，需要使用辅助位（即元数据）记录中心位的长度和（起始）位置，而这些比特几乎没有压缩空间。因此，在追求更高压缩率时，缩短**中心比特长度（CBL）**往往会成为瓶颈。为解决这一问题，先前的方法 [8, 11, 32, 33, 45] 引入预处理步骤，利用数据平滑性 [11, 32, 33, 45] 或数值精度冗余 [8, 32, 45], 等属性缩短 CBL。我们将这些预处理步骤称为**转换器**，表1总结了常见类型。

类型 1：基于平滑性的转换器利用流式数据的平滑性，即相邻值通常彼此相似——这一特性在时间序列中尤为常见。

示例 1。诸如*Gorilla [38]*和*Chimp [33]*等方法使用异或运算（例如，$\omega_2 \oplus \omega_1$）来突出相似性，从而产生许多前导零，使 *CBL* 从 *63* 降至 *39*（见表*1*第 *2* 行）。

Camel [45] 将十进制数值拆分为整数部分和小数部分。对于 $v_2 = 88.1479$，整数部分 88 与前一个整数部分相减后存储（例如，$88 - 88 = 0$），而小数部分 0。1479 则使用模数 2^{-4}（对应于 v_2 的小数点后 *4* 位）映射到更小的范围内，并乘以 10^4。这样可将中心位缩减至 *8* 比特，但会增加 *4* 比特的索引额外开销（表 *1*第 *3* 行）。

类型2 基于冗余的转换器针对 IEEE 754 规范进行目标转换，并利用对数据精度的了解来减少表示冗余。

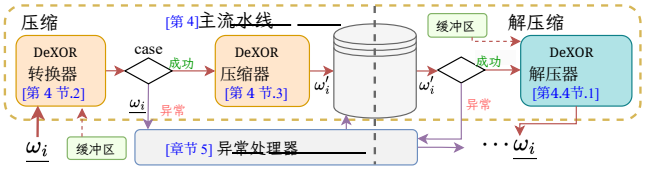

图2：“主流水线”包含一个基于十进制异或的转换器和一个专用压缩器/解压缩器，并配有一个用于处理极端情况的“异常处理器”。

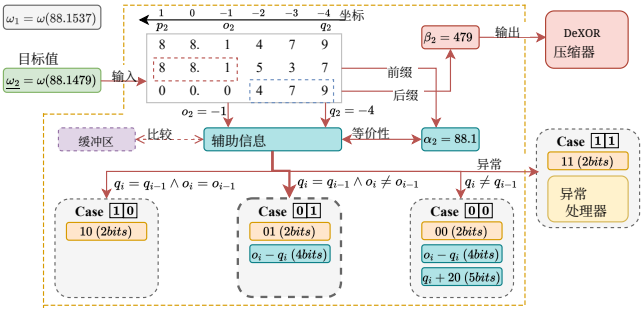

图 3：DeXOR 转换器将坐标保留为辅助信息，以替代提取出的前缀，同时将后缀传送至压缩器进行变长编码。

示例 2。*Elf [32]* 会擦除不必要的尾数，在保留原始十进制精度（单独存储）的同时引入更多尾随零，从而能够将十进制数值舍入还原（88。14788⋯表 *1* 第 *4* 行中的数值在已知 *4* 位小数时会向上舍入为 88。1479）。

ALP *[8]* $v' = v \times$ 通过将 $10^e \times 10^{-f}$ 浮点数缩放为整数，其中 $\langle e, f \rangle$ 经过仔细选择，以避免舍入误差。这种缩放会添加前导零，从而将 88.1479 的 *CBL* 优化至 *20*（表 *1*中的第 *5* 行）。

平滑性与冗余性的联合利用。大多数 SLC 转换器要么利用平滑性，要么利用冗余性，但结合这两种方法会带来挑战：

- 平滑性中断：类型 2 中基于精度的变换可能会破坏类型1中的平滑性。例如，Elf系列方案会根据精度减去最小值，从而改变二进制表示，降低其与先前值的相似性。同样，ALP 会独立缩放各个值，扭曲坐标系（例如，66.101 → 66101 和 66.1 → 661）。
- 部分解决方案：Camel 保留了整数部分的平滑性，却忽略了小数部分的相似性（例如，88.1479 和 88.1537 共有 88.1，但只保留了 88）。Elf 尝试将擦除（类型 2）与异或（类型1）结合起来，但擦除后 CBL 中的不一致会导致异或效率低下。这里的平滑性指值间连续性。在 Elf 中，根据每个值的精度对其应用可变长度比特擦除；随后，异或会将长度对齐到较长者，从而造成平滑度损失。

在本研究中，我们提出了**首个类型3 转换器**，它结合了基于平滑性和基于冗余的转换器各自的优势。我们的转换器依托一种十进制空间异或式运算，可最大限度地利用平滑性（*88.1* 是 88.1479 和 88.1537共有的部分），同时保留剩余部分（88.1479 − 88.1 = 0.0479）用于基于精度的变换。如表1第 6 行所示，该转换器名为十进制异或，其 CBL 比现有的SLC 转换器更短。

3 DeXOR 总体框架

如图 2 所示，所提出的框架通过一个 ω_i（IEEE 754 格式）对每个浮点数进行压缩，该框架围绕主流水线（第 4 节）构建十进制异或技术，并辅以异常处理器（第5节）应对极端情况。

主流水线：输入首先由 DeXOR 转换器（第 4.2 节）处理，在利用平滑性的同时消除精度冗余。此步骤会生成一个 2 位案例码，用于指示解压缩期间的执行状态。特殊案例码 1 1 会触发切换至异常处理流程，而其他情况则继续进行元数据存储和后缀编码。信息容量大幅降低的后缀随后被传递给DeXOR 压缩器（第 4.3 节），以进行专门的变长编码。

异常处理器：DeXOR 处理效果不佳的数据（例如，16+ 十进制数）会通过内联条件分支，以原始 IEEE 754 格式重定向至异常处理器。该模块会识别最稳定的组成部分（例如平滑度较高的指数）进行减法，并对减法结果采用自适应存储，以降低存储开销。异常处理过程与全局输入结构保持一致，因此在必要时能够独立执行压缩。

该框架整合了以十进制异或为核心的流水线与插件式异常处理模块，以在各种数据集上实现稳健的性能。它会动态判断是通过主流水线还是异常处理器处理数据，从而针对所有情况优化压缩。因此，如果已知数据流的先验信息，就可以直接进入异常处理器（参见第5.3节），减少转换开销，并进一步提高高精度数据的压缩率。

4 主流水线索引器

4.1 十进制异或的定义

正如第 2.3 节所述，同时利用二进制表示的平滑性和十进制数值中的精度冗余颇具挑战。这促使我们从另一个角度进行思考：浮点数在十进制空间中也表现出平滑性。这表明可以先应用类似 异或的操作，去除十进制空间中的公共部分，然后再处理残余部分中的冗余性。因此，我们提出了**十进制异或**，这是一种在全局对齐的十进制坐标系中进行浮点数据转换的方法。它将公共十进制前缀与其余部分分开处理，从而分别解决平滑性和冗余性问题。

一个十进制值 v_X可以用其坐标范围上的一个有限片段表示，即 $v_X = x_p x_{p-1} \ldots x_{q+1} x_q$，其中 $p \in \mathbb{Z}$是**首位坐标**（最高十进制位），而 $q \in \mathbb{Z}$是**尾部坐标**（最低十进制位）。我们有 $v_X = \sum_{j=q}^{p} x_j \times 10^j$。其**数据精度** dp定义为有效十进制位数，即 $dp = p - q + 1$。参见图 3，对于目标值 $v = 88.1479$，有 $p = 1$，$q = -4$，以及 $dp = 1 - (-4) + 1 = 6$。

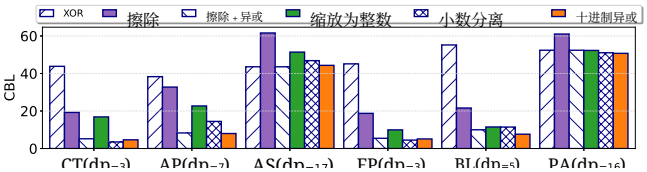

图 4：时间序列数据集的 CBL（不包括元数据开销）：City-temp (CT)、Air-pressure (AP) 和 Air-sensor (AS)；以及非时间序列数据集：Food-Price(FP)、Blockchain-tr (BL) 和 POI-lat (PA)。数据集按各类别的数据精度 (dp) 排序。

给定 $\underline{v_X}$ 目标值和 v_Y，用其十进制数字x_j和y_j（$j \in \mathbb{Z}$）表示，我们将非对称十进制异或定义为：

$$\underline{v_X} \diamond v_Y = \left(\sum_{j=-\infty}^{\infty} \left(\underline{x_j} \diamond y_j\right) \cdot 10^j \right) \cdot 10^{(-q)}, \tag{3}$$

其中，$x_j \diamond y_j$ 表示对单个数字x_j和y_j执行的十进制异或运算，定义如下：

$$\underline{x_j} \diamond y_j = \begin{cases} 0 & \text{if } \forall k \in [j, \infty)(x_k = y_k) \\ \underline{x_j} & \text{otherwise} \end{cases} \tag{4}$$

简单来说，我们有 $\overline{88.1479} \diamond \overline{88.1537} = 0.0479 \times 10^4 =$ 。由此得 479 9 1 到的 CBL 小至（见表3 中的类型）。为进一步验证其有效性，我们使用六个具有代表性的基准数据集（包括时间序列和非时间序列数据集）对不同转换器进行比较，开展了一项试点实验（数据集说明见第6.1节）。图4 报告了转换后的平均 CBL。按照设计，CBL 捕捉了各转换器的**内在压缩潜力**，且不包括任何元数据开销。

我们得出以下主要观察结果：1）尽管异或（类型1）对 dp 的增加不敏感，但所得 CBL 始终较差（CBL $\geq$ 38）。2）涉及精度冗余的方法（类型1 和类型2）的性能会随着 dp 增加而下降；当 dp 超过 15 比特时，擦除方法几乎失效（CBL> 60）；与非时间序列数据相比，缩放为整数方法在时间序列数据上的表现较差，因为它未能利用数据的平滑性。3）十进制异或在大多数情况下实现了最佳的 CBL 降幅；这归功于它成功融合了类型1 和类型2 方法的优势。有趣的是，在高dp 场景（AS 和PA）中，十进制异或的表现与异或相近；第5节将分析并讨论这些特殊情况。

4.2 TheDeXOR 转换器

尽管十进制异或具有显著优势，但要在SLC 场景中实现并非易事。压缩系统处理的是二进制 IEEE 754 数据，需要进行代价高昂的二进制-十进制转换，以识别十进制空间中的相似前缀。此外，转换误差可能会影响前缀和后缀的准确检测。为解决这些问题，我们在第4.2.1节提出了一种称为缩放截断与容错舍入的新方法，从而实现轻量且精确的前缀-后缀识别。此外，第 4.2.2 节介绍了一种高效方法，用于存储与已识别前缀和后缀相关的元数据。

4.2.1 前缀与后缀识别。令 α 为 v_X 与 v_Y 共有的**最长公共前缀（LCP）**。我们有 $\underline{v_X} \diamond v_Y = \underline{\quad} \quad \underline{\quad}$ $v_X - \alpha = \beta'$，其中 β' 是一个残余值。令 β 为十进制异或后的后缀，我们有 $\beta = \beta' \times 10^{-q}$其中 q是 v_X的尾部坐标。我们用 $o \in [p, q]$ 表示与 α在 v_X中的最长公共前缀坐标。给定图 3中的当前值 $v_X = 88.1479$和前一个值 $v_Y = 88.1537$，我们有 $\alpha = 88.1$, $o = -1$ $\underline{\quad}$，$\beta' = 0.0479$, 以及 $\beta = 479$。

为确定前缀 α和后缀 β，需要两个坐标：(1) 最长公共前缀坐标 o，即 $\alpha = \sum_{j=o}^{\infty} x_j \times 10^o$；以及 (2) 尾部坐标 q，即 $\beta = \beta' \times 10^{-q} = (v_X - \sum_{j=o}^{\infty} x_j \times 10^o) \times 10^{-q}$。

然而，v_X 由于压缩系统只接收二进制值，其十进制空间坐标是未知的。一种简单的方法是将二进制值 ω_X 重建为近似的十进制表示 v'_X，使用公式 2进行转换 v'_X 为 字符串 以便与字符串(v'_Y)进行前后缀匹配，然后将 字符串 结果转换回数值以供进一步处理。由于多次类型转换会带来额外开销，这种方法效率低下；此外，IEEE 754 转换误差可能导致无法准确识别前缀和后缀（$v'_X \approx v_X$），如本节后文所述。

为了解决这一问题，我们提出一种轻量级方法，通过简单的缩放和截断运算符来确定 o和 q。该方法通过以下两个引理进行形式化。

引理 1（最长公共前缀坐标判定）。任意公共前缀 α' 介于 v_X和 v_Y之间，均满足 $\alpha' = \sum_{j=l}^{\infty} x_j \times 10^j =$

$$\text{trunc}(v_X \times 10^{-l}) \times 10^l = \text{trunc}(v_Y \times 10^{-l}) \times 10^l, \tag{5}$$

其中，$l \geq q$ 是表示 α' 的结束坐标，trunc(·) 表示截断 函数。最长公共前缀坐标 o 是满足此条件的最小l：$\nexists\, o' < o(\text{trunc}(v_X \times 10^{-o'}) \times 10^{o'} = \text{trunc}(v_Y \times 10^{-o'}) \times 10^{o'})$。

为简洁起见，我们省略证明。

示例 3 $o = -1$ 88.1479 88.1479 ×。对于 的最长公共前缀坐标，截断 $10^1 \times 10^{-1} = 88.1 =$ $88.1537 \times 10^1 \times 10^{-1}$ $o' =$ () 截断() 。然而，对于 −2 $88.1479 \times 10^2 \times 10^{-2} = 88.14 \neq 88.15 =$ ，截断() 截断(88.1537 ×) 。对于 ， 公共前缀是 ，但不是最长的。 $10^2 \times 10^{-2}$ $o' = 0$ 88,

引理1 表明，我们可以从尾部坐标 q 开始迭代，以找到满足方程5的最小坐标。然而，由于 q 未知，必须先确定它。

引理 2（尾部坐标判定）。坐标 q 是 $v_X (v_X \neq 0)$ 的尾部坐标，当且仅当：

$$\begin{aligned} &\left(\text{trunc}(v_X \times 10^{-q}) = v_X \times 10^{-q}\right) \\ &\wedge \left(\text{trunc}(v_X \times 10^{-(q+1)}) \neq v_X \times 10^{-(q+1)}\right) \end{aligned} \tag{6}$$

证明。充分性：如果 q是尾部坐标，则缩放后的值 $v_X \times 10^{-q}$ 必须是整数，即 $\text{trunc}(v_X \times 10^{-q}) = v_X \times 10^{-q}$。此外，对于 $q+1$，缩放后的值 $v_X \times 10^{-(q+1)}$ 不可能是整数，因为更小的缩放会引入小数部分：$\text{trunc}(v_X \not\geq 10^{-(q+1)})$ $v_X \times 10^{-(q+1)}$ q 6 q。必要性：如 果满足方程 中的条件，则 是使 $v_X \times 10^{-q}$为整数的最右侧坐标。因此，q必须是尾部坐标。□

示例 4。对于 $v_2 = 88.1479$，其尾部为 $q = -4$：$\text{trunc}(v_2 \times 10^{-(-4)}) = 881479 = v_2 \times 10^{-(-4)}$，而 $\text{trunc}(v_2 \times 10^{-(-3)})$ $= 88147 \neq 88147.9 = v_2 \times 10^{-(-3)}$ 。

引理 2 给出了确定 q的条件，但其可能的范围过大，不适合进行迭代搜索。不过，

算法 1 获取_前缀_和_后缀（当前值 ω_i，前一尾坐标 q_{i-1}）

1: $v'_i \leftarrow$二进制_转为_十进制(ω_i) ▷ 方程2
2: 尾部坐标 $q_i \leftarrow$ 获取_尾部(v'_i, q_{i-1})
3: 最长公共前缀坐标 $o_i \leftarrow$ 获取_最长公共前缀(v'_i, q_i)
4: 前缀 $\alpha_i \leftarrow \lfloor v'_i \times 10^{-o_i} \rfloor \times 10^{o_i}$
5: 后缀 $\beta_i \leftarrow$ 取整$((v'_i - \alpha_i) \times 10^{-q_i})$
6: **返回** $q_i, o_i, \alpha_i, \beta_i$

输入数据流中的规律可以简化这一过程。例如，在 AP 数据集中，相邻值有 89% 的概率共享相同的尾部坐标（$q_i = q_{i-1}$）。这一观察使我们能够将前一尾坐标 q_{i-1} 用作尾部坐标搜索的初始点。

到目前为止，我们尚未讨论 **IEEE 754 转换误差**。我们不求精确的值 v_X 和 v_Y，而是使用它们的近似值 v'_X 和 v'_Y。此外，诸如 $v'_X \times 10^{-j}$（见公式 5 和 6）这样的缩放操作会引入额外误差，因为 10^{-j} 也以浮点数表示。因此，像 截断$(v'_X \times 10^{-q}) = v'_X \times 10^{-q}$这样的条件在引理 2中可能会因微小的舍入误差而不成立。为此，我们引入一个较小的容差 $\Delta = 10^{-6}$，通常设为 。如果 $|截断(v'_X \times 10^{-q}) - v'_X \times 10^{-q}| <$ 截断()，我们就视该条件为满足。引理 1中的相等条件也采用相同的容差。

示例 5 假设 $v'_2 =$ 88.147900000000007。 $v_2 = 88.1479$ 及其重构值为 $\cdots$。缩放会引入微小的舍入误差，导致精确相等条件失效：$v'_2 \times 10^4 = 881479.0000000001\cdots \neq 881479 =$ 截断$(v'_2 \times 10^4)$。因此，$v'_2.q$ 可能与实际的 $v_2.q = -4$ 不同。然而，差值 $|v'_2 \times 10^4 - v'_2 \times 10^4| =$ 0.0000000001 $< \Delta$ 截断()$\cdots$。通过允许此容差，我们确保 $v'_2.q = v_2.q = -4$。

带容错舍入的缩放截断算法见算法 1。这里，q_i通过从 q_{i-1}开始的局部性感知启发式确定，并在两步内收敛。如果 $|v'_i \times 10^{-q_{i-1}} - 截断(v'_i \times 10^{-q_{i-1}})| < \Delta$，则 $q_i \geq q_{i-1}$，并执行向前搜索，直到满足引理 2。否则，向后搜索将找到正确的 q_i。而对 o_i的搜索从 q_i开始，并持续到识别出第一个公共前缀。最长公共前缀 α_i和后缀 β_i将被提取出来进行处理（第 4–5 行）。

4.2.2 高效元数据存储。 图 3 展示了识别最长公共前缀 α_i和后缀 β_i后的处理过程。前缀 α_i无需存储，因为可以利用最长公共前缀坐标 o_i和前一个值 v'_{i-1}推导得出。存储 o_i比存储 α_i节省空间得多。相反，β_i会被传递给

DeXOR 变长编码压缩器（详见第 4.3 节）。请注意，q_i必须存储，以便在解压缩时将 β_i缩放回其十进制值，例如，$479 \times 10^{-4} = 0.0479$。因此，$o_i$和 q_i都作为元数据，使用辅助位存储。

假设十进制数值处于一个坐标范围内：$-20 \leq q \leq p \leq 11$ [1]、q_i和 o_i 各需要 5 比特来覆盖长度为 31 的区间。为尽量减少符号开销，加入偏移量 20，使 q_i 和 o_i 变为非负数。此外，我们不存储 o_i，而是存储（$o_i - q_i$），其通常处于 $[0, 15]$

[1]这一范围足以满足地理信息系统和科学计算等实际应用的需求 [23]。

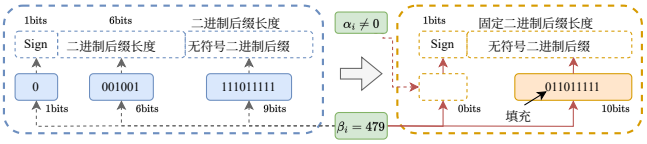

图 5：DeXOR 压缩器中的优化策略。除 $\alpha_i = 0$外，符号开销均可免除。

且只需要 4 比特。如果 $(o_i - q_i) > 15$，则将值 v_i视为高精度值，并将其交由第 5 节中的异常处理器处理。

值得注意的是，o_i和 q_i（以及（$o_i - q_i$））通常可以重复使用而无需存储，因为它们往往与前一个元素的值相同。我们使用一个两比特控制码来指示以下复用情况：

- **情况 10**（$q_i = q_{i-1} \wedge o_i = o_{i-1}$）：只需 2 比特即可指示同时复用 q_{i-1} 和 o_{i-1}。
- **情况 01**（$q_i = q_{i-1} \wedge o_i \neq o_{i-1}$）：除了控制码之外，还使用 4 比特存储（$o_i - q_i$），同时为 q_i 节省 5 比特。
- **情况 00**（$q_i \neq q_{i-1}$）：控制码占 2 比特，q_i占 5 比特，（$o_i - q_i$）占 4 比特。

当 $q_i \neq q_{i-1}$时，我们不区分 $o_i = o_{i-1}$，因为在这种情况下（$o_{i-1} - q_{i-1}$）很少能复用。相反，剩余的**情况 11** 用作异常处理器的入口点。□□

示例 6. 参见图 3，经过 *DeXOR* 转换器后的二进制流为 01000 11。前两位是控制码，而 0011 表示 $o_i - q_i = -1 - (-4) = 3$。由于复用了 $(q_i + 20)$，因此 $(q_{i-1} + 20)$节省了五个比特。

4.3 DeXOR 压缩器

DeXOR 转换器提取后缀 β_i，这是值中信息量最大的部分。由于 β_i是整数，其二进制表示通常包含前导零。为简单起见，我们将 β_i去除前导零后的二进制表示称为 **二进制后缀**。例如，$\beta_2 = 479$ 的二进制后缀为 111011111（见表 1）。随后，DeXOR 压缩器使用变长编码来减少二进制后缀的存储开销。

图5（a）展示了原始变长编码：1 比特用于符号，6 比特用于表示无符号二进制后缀的长度（最多 63 比特），其余比特用于后缀本身。例如，111011111 使用 $1 + 6 + 9 = 16$ 比特。为优化这一过程，DeXOR 压缩器采用以下两种策略。

4.3.1 优化符号管理。 利用引理3，我们降低了符号存储的额外开销：

引理 3 $\forall j\ k \in \mathbb{Z}\ x_j\ x_k \geq 0 \implies \alpha\ \beta' \geq$(符号一致性)。 0 $\implies \alpha\ \beta \geq 0$ „$\cdots$。

引理 3 表明，当 $\alpha_i \neq 0$时，β_i的符号与 α_i相同。由于 α_i通过 o_i编码，β_i的符号（及其二进制后缀）可以省略。然而，在罕见情况下，如果 $\alpha_i = 0$，则仍必须用 1 位存储β_i的符号。

4.3.2 优化的二进制后缀长度。 </style}给定整数 β_i，其无符号二进制后缀的长度为 $\ell_i = \lceil \log_2(\text{abs}(\beta_i) + 1) \rceil$，其中 abs(·) 表示绝对值。例如，$\beta_2 = 479$ 需要 $\ell_2 = 9$ 比特来存储无符号二进制后缀。事实上，记录的尾部坐标q_i 和最长公共前缀坐标 o_i 可以近似确定

值 β_i，使得 $\text{abs}(\beta_i) \in [10^{\delta-1}, 10^{\delta})$，其中 $\delta = o_i - q_i$。例如，如果 $\beta_2 = 479$，那么 $\beta_2 < 10^{(-1-(-4))} = 10^3$。这意味着表示 β_2 所需的最大比特数为 $\bar{\ell}_2 = \lceil \log_2(10^3) \rceil = 10$。通过使用固定长度 $\bar{\ell}_i = \lceil \log_2(10^\delta) \rceil$，我们无需存储 ℓ_i（其需要 6 比特存储，因为该长度最多为 63 比特）。对于无符号二进制后缀 β_i 所需比特数少于固定长度的情况，会填充前导零以匹配 $\bar{\ell}_i$。

总体而言，优化方案需要使用 4 个比特来存储 $\delta = o_i - q_i$（见第4.2.2节），并使用 $\bar{\ell}_i$ 个比特存储后缀。相比之下，原始方案需要使用 6 个比特存储 ℓ_i，并使用 ℓ_i 个比特存储后缀。下面的引理对固定长度方案的优势进行了形式化：

引理 4（无符号二进制后缀的固定比特分配）。对于任意 $\beta_i \in \mathbb{Z}$，对 $\bar{\ell}_i$ 比特进行固定分配，相较于对 ℓ_i 进行可变分配可实现更好的压缩，即 $\mathbb{E}[(4 + \bar{\ell}_i)] < \mathbb{E}[(6 + \ell_i)]$。

我们为 $\mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] < 0$ 提供证明概要。该推导以 $[10^{\delta-1}, 10^{\delta}$ $2^{j-1} < 10^{\delta-1} \leq 2^j < 2^{j+1}$ $<)$ 为条件。$2^{j+2} < 10^{\delta}$ 在，我们有 $\delta \in [(j-1)\log_{10} 2 + 1,\ j\log_{10} 2 + 1]$。我们区分两种情况：概率 $\mathbb{P}_1\{2^{j+3} > 10^{\delta}\} \approx 0.6781$，以及 $\mathbb{P}_2\{2^{j+3} \leq 10^{\delta}\} \approx 0.3219$。我们知道，$\bar{\ell}_i$ 在 $[10^{\delta-1}, 10^{\delta})$ 上为常数；ℓ_i 是一个分段函数，其期望 $\mathbb{E}_1 = \mathbb{E}(\bar{\ell}_i \mid 2^{j+3} > 10^{\delta} - \mathbb{E}\ \ell_i \mid 2^{j+3} > 10^{\delta} - 2 < j + 3 - j + \frac{7}{6} - 2 =) ( ) ( ) ( ) 。 -\frac{1}{6}$ $\mathbb{E}_2 = j + 2 - \mathbb{E}\ \ell_i \mid 2^{j+3} \leq 10^{\delta} < -\frac{1}{7}$ - 类似地，( ) 。最后，$\mathbb{E}[(\bar{\ell}_i - \ell_i - 2)$ - $] = \mathbb{E}_1 \times \mathbb{P}_1 + \mathbb{E}_2 \times \mathbb{P}_2 < -\frac{1}{6} \times \mathbb{P}_1 - \frac{1}{7} \times \mathbb{P}_2 \approx -0.159 < 0$。-

示例 7。参见图5，*DeXOR* 压缩器将后缀 $\beta_2 = 479$ 的存储空间从 *16* 比特（原始设计：*1* 比特用于符号 + *6* 比特用于 $\ell_2 = 9$，以及 $\ell_2 = 9$ 比特用于 β_2）减少到 *10* 比特（*0* 比特用于 $\alpha_i \neq 0 + \bar{\ell}_2 = 10$ 为图 β_2 中分配的固定比特5（*b*））。即使计入 δ 在*DeXOR* 转换器中已记录的 *4* 比特额外开销，优化后仍净节省 *2* 比特。

4.4 加速解压缩

解压缩过程会从压缩比特流中提取并解析各个片段。对于每个新值，DeXOR 解压缩器会读取一个两比特控制码（参见第4.2节）。如果未触发异常处理器，q_i 和 o_i 会从元数据位中复用或推导得出：q_i 将原始后缀 β'_i 缩放回原值，而 o_i 则用于确定最长公共前缀 α_i。最终值通过组合 α_i 和 β'_i 重建。当控制码为1 0 且 $q_i = q_{i-1} \wedge o_i = o_{i-1}$ 时，我们只需复用前一个最长公共前缀（$\alpha_i = \alpha_{i-1}$），无需额外计算，从而加快解压缩。在第6.2节中，我们将考察该方案的效率。

5 DeXOR 异常处理器

回看图4，所有转换器，包括 十进制异或，在高精度数据集（例如 AS 和 PA）上的表现都很差，通常与基于平滑性的 异或 转换器相当或更差。这是因为高精度数据会减少精度冗余，从而削弱基于冗余的技术（包括 十进制异或）的性能。

示例 8。高精度数据集的平滑性和精度冗余都很低，导致 *DeXOR* 主流水线中的后缀较长。例如，对于 $\delta = o_i - q_i = 16$，所需的后缀长度 $\bar{\ell}_i$ 为 *54* 比特，超过了 *IEEE 754* 格式中 尾数使用的 *52* 比特（参见公式 *1*）。这类数据使 *Elf* 等基于精度的方法几乎没有可利用的擦除机会，并导致 *ALP* 发生溢出。

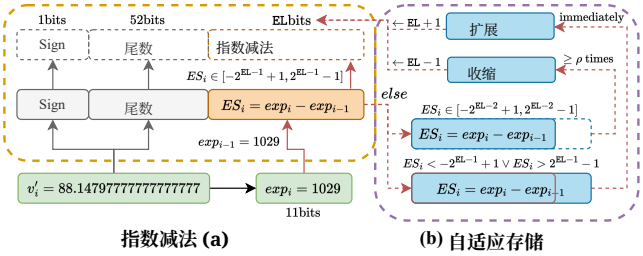

图 6：异常处理器的数据流（$exp_i = 1029$，采用预定义的 偏置 $= 1023$）；EL 是一个可调变量。

尽管 异或 转换器仅依赖平滑性且相对稳健，但由于高精度数据（CBL ≥ 38）本身的平滑性有限，其性能仍不理想。为此，所提出的异常处理器专门针对高精度情形，着重提升对平滑性的利用。如图 6（a）所示，异常处理器针对 IEEE 754 格式中的指数（指数），对相邻指数执行减法，指数$_i$ − 指数$_{i-1}$（红线），以实现数据复用。由于 符号和尾数字段的平滑性有限，因此保留原样（灰线）。为进一步优化存储，系统会根据流式数据动态调整指数减法所需的存储空间。第 5.1 节和第5.2节分别详细介绍减法和自适应存储。

5.1 指数减法

在 IEEE 754 表示（见表 1）中，尾数部分尾数通常缺乏平滑性，而指数*exp* 则包含更稳定且有意义的比特。

我们总结了 IEEE 754 表示中存储的指数的关键属性：(1) 共享值范围广：单个指数值表示给定范围内的所有实数。例如，若 指数 −偏置 $= 6$（见公式 2），则它表示 $[2^6, 2^7)$ $= [64, 128)$ 范围内的所有值，不受精度影响。(2) 低波动性：沿数据流变化时，指数值很少改变。(3) 低位密集：指数二进制表示的低位较为紧凑，因为指数均为整数。

鉴于这些属性，指数非常适合压缩，并构成第 5.2 节详述的自适应存储策略的基础。在连续指数之间采用减法而非 异或，因为在高精度数据集中，减法往往能产生更短的 CBL。对于高精度数据，IEEE 754 符号位和尾数以原始二进制形式存储，而压缩仅针对指数这一最具成本效益、最能体现平滑性的组成部分。

根据这些发现，我们将**指数减法**定义为$ES_i = exp_i - exp_{i-1}$。随后将该结果交由自适应存储模块处理。存储ES_i所需的长度定义为：$\ell^e_i = 1 + \lceil log_2(\text{abs}(ES_i) + 1) \rceil$。与第 ℓ_i 4.3.2 节中定义的相比，ℓ^e_i会额外计入 1 位来处理符号，因为ES_i可能为负数。

示例 9。考虑高精度数据（8 个 7）例子，其中 exp_i 与偏置 $7 = 6$。通常，前一个值 exp_{i-1} 等于 exp_i，因此得到 $ES_i = 0$。这使得 $\ell^e_i = 1 + \log_2(\lceil 0 + 1 \rceil) = 1$ 位长度 $ES_i = 0$ 所需的比特数为

相比之下，使用变长编码对 exp_i 进行编码需要 *4* 比特（*1* 比特用于符号，*3* 比特用于(exp_i 偏置）= 6）。

5.2 减法结果的自适应存储

仅压缩指数所能节省的空间有限，因为指数仅需 11 个比特。此外，还需要 4 个比特的元数据开销来表示 ES_i 的长度，其范围为 1 至 11 个比特，这进一步降低了压缩率。

示例 10。考虑压缩三个指数减法结果：$ES_1 = 3$、$ES_2 = 1$ 和 $ES_3 = 1$。它们源自四个不同的特殊值，这些值在原始序列中可能并不连续。它们所需的长度分别为 $\ell_1^e = 3$ 和 $\ell_2^e = \ell_3^e = 2$。采用固定元数据方案时，每个 ℓ_i^e需要 *4* 个比特，再加上残余符号和尾数所需的 *53* 个比特，因此每个条目的固定开销为 *57* 个比特。总存储成本为：$\ell_1^e + \ell_2^e + \ell_3^e + 57 \times 3 = 178$ 比特。

我们提出了一种自适应存储机制，根据历史数据动态调整存储长度 EL，以弹性地适应 ES_i。存储长度 EL 在压缩和解压缩过程中作为局部变量保存在主存中。EL 的范围为 1 至 12 个比特，足以覆盖任何 $ES_i \in [-2047, 2047]$[2]。

给定当前的 EL，范围 $[0, 2^{EL}-1]$ 用于存储整数（即 ES_i），其中最大可表示值 $2^{EL} - 1$ 保留用于指示溢出（如下所述），因此可用范围减少 1。由于 ES_i 可能为负数，因此加上偏置 $2^{EL-1} - 1$ 以抵消 ES_i的符号。实际存储的值变为：$ES_i + 2^{EL-1} - 1$，原始 ES_i 的有效范围为 $[-(2^{EL-1} - 1)\ 2^{EL-1} - 1]$。

存储当前 ES_i 后，EL 会通过两个操作动态调整，以处理下一个 ES_{i+1}，如图6(b) 所示：

• **扩展**：如果 ES_i 超出 EL 定义的有效范围，则用 1 填满当前 EL 长度的空间，表示最大可表示值 $2^{EL} - 1$。这种情况会发生**溢出**，从而触发存储 64 位 IEEE 754 表示。作为响应，EL 会为下一个条目 ES_{i+1} 增加 1（希望以此解决潜在的溢出问题）。

• **收缩**：如果 ES_i 连续落在较小范围内（$ES_i \in [-(2^{EL-2} - 1),\ 2^{EL-2} - 1]$），且持续超过 ρ 次（中断会重置计数），EL 就会减小 1（但不会小于 1），因为较小的 EL 很可能已足够。较小的 ρ 会带来过早收缩的风险，而较大的 ρ 则会延迟调整。默认情况下，我们采用 $\rho = 8$，第 6.4 节分析了其对高精度数据集的影响。请注意，如果不满足 EL 参数化条件，ρ则 EL 保持不变。

示例 11。接续示例 *10*，使用自适应存储，我们尝试存储 $ES_1 = 3$，初始长度为 EL = 1。由于 $3 > 2^1 - 1$，因此触发了溢出条件（EL 中的所有比特均为 1）。所需存储总量为 *65* 比特（用 *1* 比特的 EL 表示溢出，并存储 *+ 64*比特的原始 *IEEE 754* 表示）。

溢出后，EL 扩展为 *2*，足以存储 $ES_2 \le 2^{2-1} - 1$。这使存储成本达到 *55* 比特（EL = 2，用于存储 ES_2 *+ 53* 比特的残余符号和尾数）。同样，ES_3 也需要 *55* 比特，因为无需进一步扩展。这三个结果的总存储量为 $65 + 55 + 55 = 175$ 比特。

尽管该方法需要一个预热阶段，EL 很快就会稳定在最优值（EL $\to \ell_i^e$），使该方案能够接近理论压缩极限，且无需存储任何固定开销。此外，示例 11 相较于示例 10 的优势会愈发明显，因为额外开销被分摊到更多ES_i。

唯一的缺点是最大压缩增益上限为 11 比特。然而，在大多数极端情况下，$\delta = 16$，主流水线至少需要 56 比特，且所需比特数会随精度增加。相比之下，无论精度如何，自适应存储始终只需 56 比特存储（即流水线的下限），即使对于高精度数据也是如此。因此，当 $\delta = o_i - q_i > 15$时，我们会触发异常处理过程，其中 q_i 和 o_i 由转换器提供（参见第 4.2节），以实现最大压缩率。

5.3 异常处理器的扩展应用

异常处理器不仅能解决主流水线中的低效问题，还能处理二进制-十进制转换中的舍入误差所导致的异常（参见第 2.2 节）。在第4.2 节中，我们为 $\Delta = 10^{-6}$ 内的误差引入了容差，从而消除了大多数舍入误差。不过，仍有两种异常：

(1) 正确结果的误分类：例如，计算尾部坐标 q 时，残差极小的值 19.0000005 可能被错误地视为 19，导致得到 $q = 0$ 而非其真实值 $q = -7$。这会导致低位坐标比特被丢弃，解压值变为 19。

(2) 解压过程中的舍入误差：使用 $v_i' = \alpha_i + \beta_i \times 10^{q_i}$ 重构值时，即使 α_i、β_i 和 q_i 均正确，缩放操作仍可能引入舍入误差，导致结果与原始数据略有差异。

尽管这些错误极为罕见（在基准数据集中低于 0.01%），仍需进行特殊处理。异常处理器完全在二进制层面运行，以确保可靠性，并支持完整恢复原始二进制表示。虽然其压缩率无法达到主流水线的水平，但由于异常情况极为罕见，它能保证解压精度，同时对整体压缩的影响微乎其微。

相应的条件为 $v_i' \neq \alpha_i + \beta_i \times 10^{q_i}$。将此条件与 $o_i - q_i > 15$（第 5.2 节）结合，即可在算法 1 执行后立即识别异常情况。

通常，数据集的精度已知且保持一致，因此异常情况具有可预测性。如果精度已知，或连续出现的异常超过阈值，我们就会跳过情况代码的存储，默认仅使用异常处理器进行压缩。对于高精度的非时间序列数据集，这可减少几个百分点的比特数，并使压缩速度接近翻倍；然而，在时间序列数据集上，完整的 DeXOR 仍能实现最佳压缩效果。由于这种“先验知识”模式会带来不公平的优势，第 6 节将其禁用，仅报告不依赖精度的配置结果。

6 实验研究

6.1 实验设置

数据集：我们采用了以往研究中的一个合成数据集和 21 个真实世界数据集 [30, 32, 45]：其中 14 个为时间序列数据集，8 个为非时间序列数据集，并保留原始顺序（见表2）。后者缺少时间戳，其不规则性对基于平滑性的转换器构成挑战（类型1）；AS（合成、含噪）则对基于冗余的

[2]IEEE 754 双精度格式需要此范围，其中 $exp_i \in [0, 2047]$ [4]。

表2：ACB（↓）、压缩速度（单位：MB/s，↑）和解压速度（单位：MB/s，↑）。数据精度$dp = p - q + 1$表示每个值的平均信息量。数据集按低dp ($dp \le 7$)（标记为蓝色）和高dp (标记为红色）。Camel [45] 仅能在低dp 数据集上实现无损压缩。

	数据集	按 dp 递增排列的时间序列数据集														按 dp 递增排列的非时间序列数据集几何平均值								
		WS		CT	IR	DPT	SUSA	SUK	SDE	AP	BM	BW	BT	BP	AS	FP		BL	CA	CO	PA	PO	完整	low-dp
ACB	Gorilla	52.77	32.12	46.34	39.73	53.71	43.85	35.65	46.18	45.93	49.28	59.79	58.71	53.10	53.06	32.26	59.33	37.24	44.82	63.83	68.02	61.94	68.31	49.09	46.82
	Chimp	51.94	33.31	45.75	43.36	56.69	49.52	43.31	52.18	56.76	54.58	60.13	59.96	58.59	58.33	34.05	57.99	34.99	44.57	61.71	64.20	60.75	64.60	51.18	48.31
	Elf	16.44	13.12	20.78	18.09	27.38	24.78	24.25	27.10	36.81	36.12	39.33	39.81	43.92	62.73	17.82	24.20	18.24	23.84	36.53	40.19	64.39	69.45	29.72	23.18
	Elf+	14.27	9.70	18.35	14.91	24.32	21.86	22.82	25.29	33.37	34.61	37.80	37.21	40.97	63.69	17.30	21.58	16.24	21.21	34.56	39.49	65.34	70.36	27.31	20.57
	Camel	10.36	10.79	11.47	11.59	**12.09**	11.45	**10.18**	13.58	/	/	/	/	/	/	/	**13.85**	20.01	19.97	31.94	34.41	/	/	/	14.86
	DeXOR				**8.01**	13.22																			
Com. Speed	Gorilla	10.06	34.37	9.77	34.42	3.55	18.19	30.80	17.69	28.78	2.97	3.50	25.95	21.16	23.17	**22.68**	6.81	11.66	30.46	25.45	7.41	7.65	**12.16**	13.94	14.95
	Chimp	13.20	35.01	10.58	15.13	16.25	35.63	34.60	**26.57**	31.80	2.25	5.35	20.34	**28.19**	16.57	21.51	**25.50**	0.56	31.92	26.26	30.09	**23.01**	7.91	**15.69**	17.47
	Elf	10.94	13.36	**35.11**	36.87	**18.11**	36.20	18.45	22.77	32.79	2.89	**6.67**	19.96	1.58	**26.90**	4.14	5.07	1.12	34.45	4.98	10.02	9.90	6.37	10.93	13.53
	Elf+	17.01	10.85	11.16	39.70	17.63	39.74	31.35	26.23	**36.51**	**4.12**	4.77	**33.40**	20.86	1.22	21.01	6.15	7.82	**38.34**	29.13	**30.79**	19.16	4.81	15.14	19.92
	Camel	8.46	0.47	6.28	15.25	2.98	9.61	10.44	4.21	/	/	/	/	/	/	/	2.75	0.34	3.66	21.46	3.97	/	/	/	4.23
	DeXOR											4.65											1.86		
Decom. Speed	Gorilla	60.61	78.96	52.20	72.09	7.76	62.94	48.26	44.95	58.26	8.37	37.97	48.03	53.97	52.49	29.12	21.72	40.45	66.49	54.81	48.25	42.46	**40.66**	41.39	44.82
	Chimp	64.87	**81.76**	**69.16**	73.51	8.29	46.08	21.46	58.53	52.25	**26.50**	34.56	52.79	47.65	43.73	**71.59**	33.08	24.80	**70.72**	54.92	59.81	50.23	37.58	44.41	44.21
	Elf	60.92	69.30	21.75	67.49	26.59	28.57	57.27	23.17	36.11	6.14	27.19	51.41	47.86	51.22	56.38	24.52	15.01	37.78	37.70	7.35	43.99	25.98	31.86	30.97
	Elf+	42.92	44.18	56.77	68.25	33.23	53.84	58.62	51.62	53.78	5.36	5.75	36.37	28.64	48.47	10.05	11.26	1.04	49.29	51.56	48.81	9.67	26.43	25.90	33.11
	Camel	**95.15**	75.00	38.54	95.04	35.78	32.68	83.70	35.83	/	/	/	/	/	/	/	**67.42**	12.51	36.22	56.90	48.56	/	/	/	48.28
	DeXOR																								

转换器（类型 2）；PA/PO 提供高精度地理空间坐标，对这两种类型的转换器都构成挑战。这些数据集包含的序列最长可达数亿个点，数值精度（dp）为 3 到 17 位，数值范围从狭窄（例如 $[0.01, 1.45]$）到宽广（例如 $[13{,}068, 532{,}323]$）不等，涵盖气象学、金融、生态学、出行、区块链和地理空间分析等领域 [3]。

基线方法与指标。我们将 DeXOR 与多种最先进的 SLC 方案进行比较，包括Gorilla [38], Chimp [33], Camel [45], 和 Elf 系列（Elf [32] 及其变体 Elf+ [30]）。为扩大比较范围，第 6.6 节还涵盖了采用滑动窗口或截断窗口的方案：ALP [8], Chimp_{128} [33], SElf∗ [31]，以及 Elf∗ [31]。为了更直观地衡量压缩率，我们采用 ALP [8]中的平均压缩位数（ACB）指标。压缩率等于 ACB/64（IEEE 754 表示长度）。我们通过压缩和解压速度来评估效率，速度以每秒兆字节（MB/s）为单位。

设置与实现。实验在 Ubuntu 服务器（Intel Xeon Gold 6348, 512 GB 内存）上运行。所有 SLC 方案均基于 Java，并使用 Docker 容器化，以确保可复现性并隔离资源；该流水线支持由时间戳驱动的数据流处理。系统级 IoTDB [39] 集成情况见第 6.7 节。

6.2 总体性能

表2和图7展示了三项指标的结果。平均压缩位数。总体而言，DeXOR始终实现最佳 ACB，仅在三个低差分隐私数据集（DPT、SUK、EVC）上比 Camel 高出 ≤ 2 比特；在这些数据集上，Camel采用有损截断，目标值为$q \in [-4, -1]$。然而，这种专门优化在高差分隐私数据（红色）和离群序列（FP）上会导致精度损失或溢出。Gorilla和Chimp采用异或和变长编码策略，达到 ≥ 30，且通常超过50比特；Elf和Elf+在低差分隐私数据上实现较低的ACB（≤ 30，某些情况下甚至为 ≤ 10），但在高差分隐私数据（AS、PA、PO）上表现下降。总体而言，只有DeXOR（以及Camel）在大多数情况下都能维持ACB≤ 20，将低差分隐私输入压缩至原始大小的≤ 25%，最佳情况下可低至12.5%。

[3]所有数据集的资源、统计数据和处理脚本，以及 DeXOR源代码均已公开，地址为 https://github.com/SuDIS-ZJU/DeXOR。

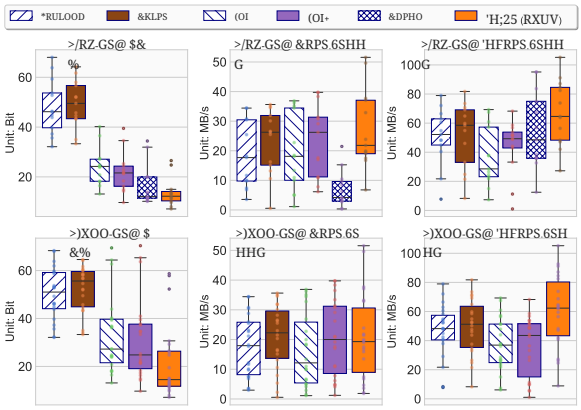

图 7：表 2 在低 dp（上图）和完整 dp（下图）下的箱线图可视化；箱体表示四分位距和中位数，须线延伸至 1。5×个四分位距，点表示各数据集的单独结果。

压缩速度。DeXOR的速度与最快的基线（Chimp）相差不超过5%，吞吐量是其最接近的压缩率竞争对手Camel的两倍。在CA上，由于不稳定的尾部坐标触发了重新计算（算法 1），DeXOR速度较慢；但在低差分隐私蓝色区域（dp ≤ 7 和 $q \in [-4, -1]$），其速度比Elf+ 快21%，比Camel快4.7×。异或方案（Chimp、Gorilla）的中央处理器开销最低；Elf和Elf+ 则在输入/输出方面取得平衡；DeXOR的动态启发式策略（见算法 1）偏向规则序列，只有少数不规则序列（IR、CA）会产生额外开销。

解压缩速度。DeXOR在 22 个数据集中有 13 个表现领先，平均速度比 Camel 快 31%，比 Gorilla 快 41%；即使在其压缩表现落后的数据集（IR、CA）上，它仍然是最快的，并且在 SSD 上的吞吐量达到最先进水平的两倍。唯一的例外是少数低dp 数据集（例如 WS），在这些数据集上 DeXOR 的运行速度约为 Camel 的一半。

这种出色的解压缩性能源于两个因DeXOR 素。首先，它只读取存储的坐标，因此不稳定的模式不会拖慢解压缩。其次，复用前一个最长公共前缀α_i 可将解压缩工作量减少 ~50%，相较于完整重新计算（第4.4节）。在低dp 数据集（PM、EVC）上，输入/输出节省和可复用步骤有限，优势略有缩小；但在高DeXOR dp 数据上，*DeXOR* 仍明显更快。

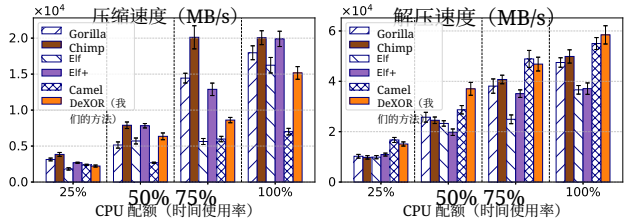

图 8：可扩展性与 CPU 配额的关系。柱形表示 10 次运行的平均值，误差线表示一个标准差。

6.3 可扩展性研究

我们从两个维度评估所有方案的可扩展性——（1）中央处理器配额限制和（2）数据采样变异性：

CPU 配额约束（图8）。我们将 CPU 使用率限制在 25% 至 100%，并记录运行速度。大多数方案呈线性扩展；异或方案（Gorilla、Chimp）在 50–75% 时提速最快。DeXOR 稳步提升：在 25% 时解压缩速度排名第二，在 50–100% 时压缩和解压缩速度均排名第一。

数据采样变异性（图9）。我们还通过压缩精度各异的代表性时间序列数据集来评估鲁棒性，采用两种采样策略：（1）连续采样，保留时间顺序；（2）随机采样，破坏上下文连续性。

总体而言，DeXOR 在两种采样策略下始终实现最低的 ACB，且方差极小。随机采样会显著影响流式数据的平滑性。基于异或的方法（如 Gorilla）对平滑性高度敏感，在 AP 数据集采用 60% 随机采样时，ACB 最差。Elf 和 Elf+比简单的基于异或的方法具有更好的容差。感知十进制的方法（Camel 和 DeXOR）在 CT 和 DPT 数据集上表现良好，但在 CT 和 DPT 上进行随机采样时，Camel 的 ACB 会显著下降（损失 10 ~ 15 比特），而 DeXOR 则保持稳定。在 WS 上，Elf 系列受到相关性破坏的影响。相比之下，DeXOR 通过有效利用数据规律性（第 4.2.2 节）保持 ACB 稳定。

6.4 参数敏感性分析

第 5.2 节 介绍了一个超参数 ρ，它用作阈值，用于确定何时收缩记录的指数减法的位长度 EL 。优化 ρ 可略微提高高 *dp* 数据集（如 AS、PA 和 PO）的压缩率。

如图10所示，三条虚线表示从未发生收缩的情况（$\rho \to +\infty$），这会导致显著的冗余性和最差的 ACB。相反，无条件收缩（$\rho = 0$）表现更好，这表明无控制扩展的负面影响大于过早收缩造成的损失。这一观察结果说明应优先选择较小的 ρ 值。不同数据集上的 ρ 对 ACB 的影响趋势一致：ACB 会迅速下降至最小值，随后缓慢上升。对于 PA 数据集，当 $\rho \geq 1$ 时，ACB 趋于稳定，直到 $\rho \geq 9$时才开始明显上升。相比之下，对于 PO数据集，$\rho = 0$ 优于 $\rho \geq 1$，这表明指数波动经常受到衰减，因此 $\rho = 0$ 是局部最优选择。

表 3：消融研究中最佳 ACB 结果以粗体显示。Δ 表示 ACB 的变化，绿色表示提升，红色表示下降。灰色区域表示停用异常处理器。

数据集		DeXOR	不含异常处理程序		不含解码异或		两者均不含	
		ACB ↓	ACB ↓	Δ (%)	ACB ↓	Δ (%)	ACB ↓	Δ (%)
Time-Series (TS)	WS	**10.35**	**10.35**	0.00	10.73	-3.67	10.73	-3.67
	PM	**7.12**	**7.12**	0.00	8.72	-22.47	8.72	-22.47
	CT	**11.33**	**11.33**	0.00	14.02	-23.74	14.02	-23.74
	IR	**8.01**	**8.01**	0.00	13.47	-68.16	13.47	-68.16
	DPT	**13.22**	**13.22**	0.00	18.18	-37.52	18.18	-37.52
	SUSA	**9.71**	**9.71**	0.00	19.06	-96.29	19.06	-96.29
	SUK	**11.59**	**11.59**	0.00	21.16	-82.57	21.16	-82.57
	SDE	**12.22**	**12.22**	0.00	21.68	-77.41	21.68	-77.41
	AP	**14.87**	**14.87**	0.00	27.68	-86.15	27.68	-86.15
	BM	**19.47**	**19.47**	0.00	29.33	-50.64	29.33	-50.64
	BW	**30.67**	**30.67**	0.00	32.21	-5.02	32.21	-5.02
	BT	**29.25**	**29.25**	0.00	31.73	-8.48	31.73	-8.48
	BP	**25.89**	**25.89**	0.00	34.35	-32.68	34.35	-32.68
	AS	**52.28**	53.13	-1.63	55.93	-6.98	64.70	-23.76
Non-TS	FP	**12.38**	**12.38**	0.00	16.99	-37.24	16.99	-37.24
	EVC	**14.13**	**14.13**	0.00	14.77	-4.53	14.77	-4.53
	SSD	**13.27**	**13.27**	0.00	17.20	-29.62	17.20	-29.62
	BL	**15.00**	**15.00**	0.00	18.62	-24.13	18.62	-24.13
	CA	**24.76**	**24.76**	0.00	24.97	-0.85	24.97	-0.85
	CO	26.53	26.53	0.00	**26.19**	**1.28**	**26.19**	**1.28**
	PA	57.86	63.97	-10.56	**57.61**	**0.43**	64.70	-11.82
	PO	58.7	65.05	-10.82	**58.49**	**0.36**	65.24	-11.14
平均值		**21.76**	22.36	-2.78	26.05	-19.74	27.08	-24.46

根据这些发现，我们将 $\rho = 8$ 设为所有数据集的默认值。对于 PA 和 PO，这一选择可能是最优的；而对于 AS，它与局部最优值的差异小于 0.01 比特。在大多数情况下，这一默认设置已足够。

6.5 消融研究

我们逐模块进行消融分析，以评估各组件的贡献，同时始终保留 DeXOR 压缩器。我们考察：(1)*w/oExcep.*，它不进行异常处理，直接以 IEEE 754 格式存储极端情况的数据；以及 (2) *w/o* 解码异或，它跳过消除最长公共前缀（第 4.2 节），但仍存储元数据和缩放后缀，包括 $\delta = (o_i - q_i)$、q_i和 $\beta_i = v_i \times 10^{-q_i}$。结果汇总于表 3。

异常处理器的影响。该插件模块主要在高 *dp* 数据集上触发，如表 3中的灰色部分所示。尽管它仅在特定情况下启用，但对 AS、PA 和 PO 等数据集上的 ACB 性能贡献显著。移除该模块后，DeXOR 在时间序列数据集 AS上的性能略有下降，而在 PA 和PO 等非时间序列数据集上的性能则显著退化，有时甚至不如原始的 64 位 IEEE 754 表示。然而，我们确认，即使没有异常处理器，DeXOR 在 ACB 方面仍与所有基线方法保持竞争力。相比之下，消融该模块后，低精度数据集上的任何缺陷都必然源于容差失效（见第 4.2.1 节），但观测值为零；因此，容差误差可以忽略不计。

十进制异或的作用。移除十进制异或会导致性能严重下降，使 DeXOR退化为缩放为整数方案。如表3所示，这会使所有数据集的 ACB 平均损失 19.74%。然而，当未对齐的尺度导致前置填充零、从而提高所需精度时，十进制异或会给特定数据集（例如PA、PO、CO）带来额外开销。例如，将 8.81479 ⋄ 88.1479 作为参考值进行处理，可能会得到最长公共前缀坐标（$o_i = 2$） > （$p_i = 0$），使

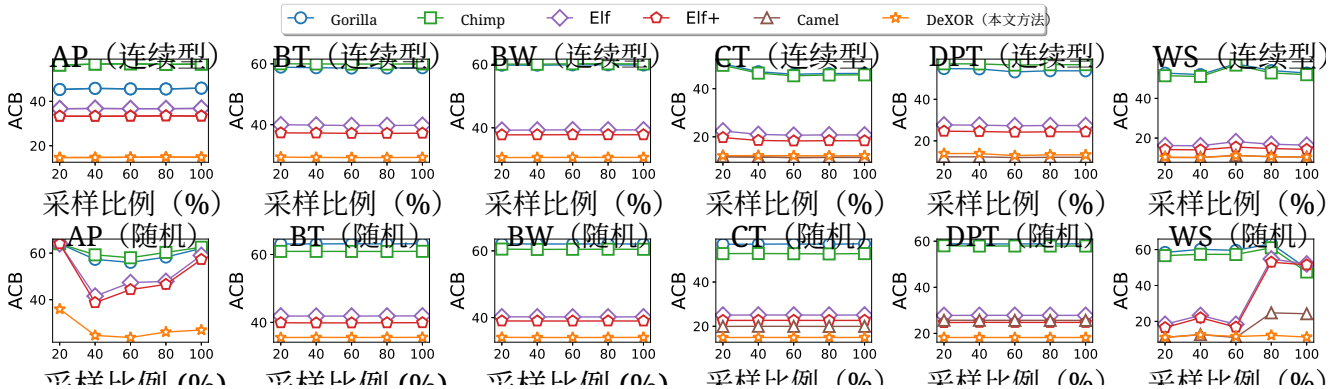

图 9：六个具有代表性的时间序列数据集上的可扩展性测试（“连续型”表示保留时间顺序，而“随机”表示上下文连续性被破坏）。Camel 不适用于高精度 数据集 AP、BT 和 BW。

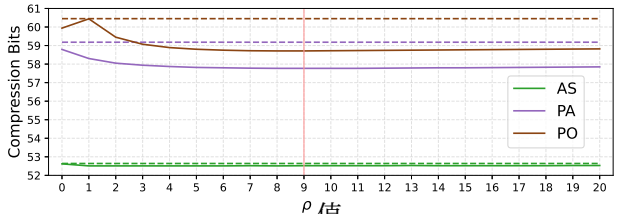

图 10：ρ 对 DeXOR 中 ACB 的影响。虚线表示 $\rho \to +\infty$时的情况。

表 4：与更大缓冲区方案的比较 **es.**

指标（几何平均值）	Chimp$_{128}$	SElf∗	ALP	Elf∗	DeXOR（我们的方法）
平均压缩比特数 ↓	31.37	19.55	52.97	**17.55**	17.82
压缩速度（MB/s） ↑	12.65	14.54	2.07	**16.06**	14.94
解压速度（MB/s） ↑	40.27	49.69	38.73	23.41	**53.12**

表 5：四个数据集上采用原始、零均值（-Z）和标准化（-S）预处理策略的 ACB（↓）。

数据集	Gorilla	Chimp	Elf	Elf+	DeXOR（我们的方法）
CT	46.34	45.75	20.78	18.35	**11.33**
CT-Z	47.95	**47.66**	52.51	53.51	57.10
CT-S	**51.48**	52.04	54.97	55.95	55.94
AS	53.06	58.33	62.73	63.69	**52.28**
AS-Z	**51.00**	58.39	60.57	61.56	53.44
AS-S	**53.21**	58.43	62.74	63.68	54.34
FP	32.26	34.05	17.82	17.30	**12.38**
FP-Z	**29.79**	34.69	37.53	38.53	56.05
FP-S	**35.99**	40.90	43.60	44.56	42.20
PA	61.94	60.75	64.39	65.34	**57.86**
PA-Z	65.77	63.69	66.48	67.44	**59.11**
PA-S	67.40	63.89	68.27	69.24	**59.10**

$\delta = (o_i - q_i = 7) > dp = (p_i - q_i + 1 = 6)$[4]。这反映了转换器中的一种权衡：复用前一个后缀可以避免显式存储该后缀，但未对齐有时会要求更高的精度。两个模块的作用。移除两个模块后，DeXOR实质上只保留了元数据复用（第4.2.2节）和压缩器（第4.3节）的功能。尽管如此，表3 显示，剩余组件的压缩率仍可与Elf、Elf+和Camel 相媲美（参见表2）。

有两个值得注意的观察结果：(1) 在 CO 数据集上，精简版 DeXOR 的性能略优于完整版本（+1.28%），这是由罕见数据情况导致的错位开销所致。在 PA 和 PO 数据集上也观察到了类似现象，不过移除异常处理器带来的收益抵消了性能损失。(2) 十进制异或通过消除最长公共前缀减少异常，使一些高dp数据落在异常情况的范围之外。例如，在 AS 数据集上，消融使 ACB 恶化了 -23.76%，远差于各项单独消融的综合影响（(-1.63%) + (-6.98%)）。这凸显了 DeXOR 的一项关键优势：能够对异常进行事后判定，从而更准确地处理高dp但平滑的数据。

6.6 带缓冲或预处理的 SLC

更大的缓冲区方案与预处理的 SLC 比较

在更大的缓冲区方案中，一些方案会利用更大的缓冲区——例如滑动窗口（Chimp$_{128}$ [33]，SElf∗ [31]）或小批量（ALP [8]，Elf∗ [31]）——以提升压缩效果。由于这些方法需要更多内存，我们在表 2中单独报告，而不将其与主要基线混在一起。各方案对缓冲区大小的敏感程度及其目标场景各不相同。为确保公平，我们采用各方案推荐的缓冲区大小：$N = 128$ 用于 Chimp$_{128}$，$N = 1024$ 用于 ALP，以及 $N = 1000$ 用于 SElf∗ 和 Elf∗。如表 4所示，尽管面对的是大量使用缓冲区的方案，DeXOR仍实现了最快的解压缩速度，并且压缩率和速度均排名第二，仅比 Elf∗ 低 0.27 比特，慢 1.12 MB/s。

预处理方法。零均值归一化和 0-1 标准化会降低局部平滑性、增加熵，并在引入额外运行时开销的同时造成不可逆的损失：例如，将

$$50.0 \to \cdots \text{转换为} \; -0.0431$$

exp会扩大精度要求，使每种 SLC 方案都承受更大压力。如表 5所示，这些预处理方法会损害所有精度冗余方案（包括 DeXOR），而简单的 异或方案（ Gorilla 、 Chimp ）则会从中受益。对于高精度的不规则数据集（AS 和PA），预处理对 DeXOR 的影响较小，它仍能保持有竞争力的结果。因此，我们建议

[4]考虑一种简化情况：$v_X = 8.8$ 和 $v_Y = 88.1$ 的首位坐标不同($v_X.p = 0$) $\neq$ ($v_Y.p = 1$)，这意味着在 $o = 2 = \max(v_X.p, v_Y.p) + 1$处，最长公共前缀为 $\alpha = 0$。这会在十进制异或中将后缀补齐为088，虽然结果正确，但效率较低。

表6：所有结果均在 Apache IoTDB Tsfileengine 中生成，并在三个异构真实世界数据集上进行评估：CT（时间序列）、FP（非时间序列）和 PA（高精度非时间序列）。

二次压缩	算法	平均压缩位数（↓）				压缩吞吐量（MB/s，↑）				查询延迟（毫秒/1k，↓）			
		CT	FP	PA	平均值	CT	FP	PA	平均值	CT	FP	PA	Avg.
未压缩	Sprintz	51.94	52.01	61.85	55.27	1.344	4.160	3.136	2.880	0.67	0.188	0.132	0.330
	Gorilla	58.17	40.28	67.57	55.34	6.016	5.888	**4.096**	**5.205**	0.174	**0.073**	**0.090**	0.112
	DeXOR	**12.91**	**13.96**	**59.45**	**28.78**	**6.144**	**6.016**	3.072	4.949	**0.093**	0.083	0.106	**0.094**
Lz4	Sprintz	37.57	34.93	60.95	44.48	4.032	5.696	3.264	4.331	0.105	0.118	0.127	0.116
	Gorilla	29.00	27.97	66.70	41.22	5.120	5.760	**4.480**	**5.120**	0.095	**0.081**	0.101	0.092
	DeXOR	**11.56**	**12.83**	**58.54**	**27.64**	**5.952**	**5.824**	3.008	4.928	**0.079**	0.086	**0.090**	**0.085**
Snappy	Sprintz	36.70	34.17	60.72	43.86	3.840	5.632	**4.032**	4.501	0.119	0.099	0.091	0.103
	Gorilla	30.13	28.89	66.44	41.82	5.120	**6.208**	3.968	5.099	0.140	**0.077**	**0.080**	0.098
	DeXOR	**11.69**	**12.83**	**58.32**	**27.61**	**6.272**	5.760	3.328	**5.120**	**0.073**	0.092	0.092	**0.085**

表 7：Gorilla 和 DeXOR 在 IoTDB 上对代表性向量数据集的压缩与查询性能。

指标	算法	数据集		
		SIFT	WR	WW
平均压缩位数（↓）	Gorilla	19.65	42.84	43.03
	DeXOR	**12.99**	**13.47**	**13.11**
压缩吞吐量（MB/s，↑）	Gorilla	13.510	4.933	6.958
	DeXOR	**18.493**	**11.167**	**14.724**
查询延迟 (毫秒/1k, ↓)	Gorilla	0.017	0.014	0.005
	DeXOR	**0.0002**	**0.010**	**0.004**

在使用 DeXOR 之前，避免采用会提高精度或引入损失的变换；如果必须进行归一化，应在 DeXOR 无损解压缩后再执行，以获得更好的性能。

6.7 真实世界时序数据库测试

Apache IoTDB 的列式引擎 TsFile [39] 采用可插拔的编码层，使我们能够在不更改编排的情况下添加 DeXOR 。在三个不同的数据集（CT、FP、PA）上，我们将 DeXOR 与 IoTDB 的 Gorilla 和 Sprintz 方案进行比较，并分别与下游 Lz4 [3] 或 Snappy [6] 级联，以测试二次压缩堆叠效果。表 6 显示，DeXOR 的压缩效果始终优于 Gorilla 和 Sprintz（最高可达 4。5×），同时保持摄取吞吐量，并能更快地响应 1,000 次单值查询；二次压缩器 Lz4 和 Snappy 仅带来 < 2% 的额外空间缩减，且运行时开销可以忽略不计，因此对于 DeXOR 而言是可选的。

向量数据压缩支持。虽然主要研究涵盖 22 个一维序列，但向量也被广泛使用。为确定 DeXOR 能否直接复用，我们选取了两个有代表性的数据集：SIFT [2]（10k 个向量，每个向量有 128 个维度，用于计算机视觉）和 Wine-Quality [15]（4,898 个向量，每个向量有 11 个维度，表示红葡萄酒和白葡萄酒——WR 和 WW）。每个维度独立压缩。一次查询对应于检索一条完整的向量记录，延迟通过 1,000 次此类查询进行测量。如表 7所示，无需针对向量进行任何专门的重新设计，DeXOR 仍能实现 ~ 3.5× 更好的压缩效果、 ~ 2× 更高的吞吐量，并且相比按维度压缩的 Gorilla延迟大幅降低，表明 DeXOR适用于向量数据。

7 相关工作

通用压缩。LZ4 [3], LZMA [1], Snappy [6],XZ [5], 和 Brotli [9] 利用冗余性，通过霍夫曼编码和字典编码实现压缩，在静态数据上压缩效果更好，但需要反复解压和重新压缩，妨碍实时应用。

流式有损压缩结合数据清理技术 [17, 26, 35] 和误差有界机制 [12, 16, 29, 34, 44] 以精度换取压缩率。尽管我们的目标是无损压缩，但这些技术也很容易扩展到有损场景。

学习式压缩 [16, 19, 24, 35, 44] 对值进行预测，并对高精度随机残差进行编码，因此大多数方法都是有损的（例如 SZ [16], MOST [44]）。NeaTS [19] 则因提供了一个可配置的 开关而脱颖而出，既支持有损压缩，也支持无损压缩。这些技术高度依赖模型且对硬件要求较高，虽前景可期，但仍需进一步探索。

无损压缩已针对**流式数据** [8, 10, 11, 13, 20–22, 24, 25, 27, 30–33, 37, 38, 41, 42, 46] 以及**浮点数据** [8, 27, 30–32, 38]开发。流式数据压缩通常会利用上下文统计信息。ALP [8]和Elf ∗ [31] 等批处理方案采用熵编码和字典编码等离线技术，以更高的延迟换取更好的压缩效果。对于浮点数据，异或 [38] 和缩放为整数 [8]等技术可缓解 IEEE754 编码带来的问题。DeXOR将这些技术扩展到整数压缩，并加入可配置的异常处理器，以便在线适应不同的数据集。

向量数据压缩。低维向量可以利用 SIMD 并行处理按维度进行压缩 [8]；高维向量（例如 DeltaPQ [40]）的压缩依赖于量化。压缩向量中的连续值非常直观（参见我们在第 6.7 节中的实现），而利用向量间平滑性进行无损压缩仍是一个悬而未决的问题。

8 结论

我们提出了 DeXOR，这是一种用于浮点数据的流式无损压缩方案，可在十进制空间中应用轻量级类异或转换。通过利用常见的十进制前缀和后缀，它统一了基于平滑性和基于冗余的技术，同时缓解了 IEEE 754 转换误差、异常值及高精度情形带来的问题。DeXOR 在十进制数值压缩方面实现了最先进的性能，并具有出色的灵活性。

针对不同场景构建 DeXOR 系列已有明确方向：（1）批处理模式 DeXOR：采用先进的并行处理技术（如 SIMD 向量化），实现高性能的原生向量数据压缩。（2）基于模型的无损 DeXOR：改进十进制异或，以压缩轻量级时间序列预测模型生成的预测残差。（3）有损 DeXOR：开发功能强大的误差有界变体，以适应特定的数据条件。

参考文献

[1] 2001. Go 语言中的 LZMA 压缩/解压缩。https://code.google.com/archive/p/lzma/. [2] 2010. ANN_SIFT10K（siftsmall）。http://corpus-texmex.irisa.fr/ [3] 2013. LZ4：极快的压缩算法。https://github.com/lz4/lz4. [4] 2019. IEEE 浮点算术标准。IEEE Std 754-2019（IEEE 754-2008 的修订版）（2019），1–84. [5] 2022. XZ Utils。https://github.com/tukaani-project/xz. [6] 2023. Snappy：一种快速压缩器/解压缩器。https://github.com/google/ snappy. [7] Daniel J. Abadi、Samuel Madden 和 Miguel Ferreira。2006。将压缩与执行集成到列式数据库系统中。载于 SIGMOD。671–682。[8] Azim Afroozeh、Leonardo X. Kuffo 和 Peter Boncz。2023。ALP：自适应无损浮点压缩。SIGMOD 1，4（2023），1–26。[9] Jyrki Alakuijala、Andrea Farruggia、Paolo Ferragina、Eugene Kliuchnikov、Robert Obryk、Zoltan Szabadka 和 Lode Vandevenne。2018。Brotli：一种通用数据压缩器。TOIS 37，1（2018），1–30。[10] Noura Alghamdi、Liang Zhang、Huayi Zhang、Elke A. Rundensteiner 和 Mohamed Y. Eltabakh。2020。ChainLink：为长子序列匹配建立大型时间序列数据索引。载于 ICDE。529–540。[11] Davis Blalock、Samuel Madden 和 J ohn Guttag。2018。Sprintz：物联网时间序列压缩。IMWUT 2，3（2018），1–23。[12] Shubham Chandak、Kedar Tatwawadi、Chengtao Wen、Lingyun Wang、Juan Aparicio Ojea 和 Tsachy Weissman。2020。LFZip：通过改进预测对多变量浮点时间序列数据进行有损压缩。载于 DCC。342–351。[13] Giacomo Chiarot 和 Claudio Silvestri。2023。时间序列压缩综述。计算机调查 55，10（2023），1–32。[14] ClickHouse。2025。ClickHouse 中的压缩。https://clickhouse.com/docs/data-compression/compression-in-clickhouse. [15] Paulo Cortez、A. Cerdeira、F. Almeida、T. Matos 和 J. Reis。2009。葡萄酒质量。https://archive.ics.uci.edu/dataset/186/wine+质量 [16] Sheng Di 和 Franck Cappello。2016。使用 SZ 实现快速、误差有界的有损高性能计算数据压缩。载于 IPDPS。730–739。[17] Eugene Fink 和 Harith Suman Gandhi。2011。通过提取主要极值压缩时间序列。JETAI 23，2（2011），255–270。[18] 谷歌。2018。移动速度的必要性：移动延迟如何影响出版商收入。谷歌研究白皮书。[19] Andrea Guerra、Giorgio Vinciguerra、Antonio Boffa 和 Paolo Ferragina。2025。通过随机访问对非线性时间序列进行学习式压缩。载于 ICDE。1579–1592。[20] Adrián Gómez-Brandón、José R. Paramá、Kevin Villalobos、Arantza Illarramendi 和 Nieves R. Brisaboa。2021。支持直接访问的工业时间序列无损压缩。《工业计算机》132（2021），103503。[21] Huamin Chen、Jian Li 和 P. Mohapatra。2004。RACE：面向传感器网络、具备速率自适应和误差界的时间序列压缩。载于 MASS。124–133。[22] Sang-Ho Hwang、Kyung-Min Kim、Sungho Kim 和 Jong Wook Kwak。2023。基于动态位打包的时间序列传感器数据无损压缩。Sensors 23，20（2023），8575。[23] Søren Kejser Jensen、Torben Bach Pedersen 和 Christian Thomsen。2017。时间序列管理系统：综述。TKDE 29，11（2017），2581–2600。[24] Nan Jiang、Qingping Xiang、Hongzhi Wang 和 Bo Zheng。2023。基于强化学习的时间序列压缩。《信息科学》648（2023），119490。[25] Abdelouahab Khelifati、Mourad Khayati 和 Philippe Cudre-Mauroux。2019。CORAD：使用稀疏字典编码进行海量时间序列的相关性感知压缩。载于大数据。2289–2298。

[26] Lucie Klus、Roman Klus、Elena Simona Lohan、Carlos Granell、Jukka Talvitie、Mikko Valkama 和 Jari Nurmi。2021 年。《面向可穿戴传感器数据的直接轻量级时序压缩》。Sensors Letters 5，2（2021），1–4。[27] Maximilian Kuschewski、David Sauerwein、Adnan Alhomssi 和 Viktor Leis。2023 年。《BtrBlocks：面向数据湖的高效列式压缩》。SIGMOD 1，2（2023），1–26。[28] Huan Li、Hua Lu、Christian S Jensen、Bo Tang 和 Muhammad Aamir Cheema。2022 年。《物联网中的空间数据质量：管理、利用与前景》。CSUR 55，3（2022），1–41。[29] Ruiyuan Li、Zechao Chen、Ruyun Lu、Xiaolong Xu、Guangchao Yang、Chao Chen、Jie Bao 和 Yu Zheng。2025 年。《Serf：流式误差有界浮点压缩》。SIGMOD 3，3（2025），1–27。[30] Ruiyuan Li、Zheng Li、Yi Wu、Chao Chen、Songtao Guo、Ming Zhang 和 Yu Zheng。2023 年。《面向流式浮点时间序列的基于擦除的无损压缩方法》。ArXiv abs/2306.16053（2023）。[31] Ruiyuan Li、Zheng Li、Yi Wu、Chao Chen、Tong Liu 和 Yu Zheng。2025 年。《无损浮点压缩的自适应编码策略》。IoTJ（2025），1–1。[32] Ruiyuan Li、Zheng Li、Yi Wu、Chao Chen 和 Yu Zheng。2023 年。《Elf：基于擦除的无损浮点压缩》。PVLDB 16，7（2023），1763–1776。[33] Panagiotis Liakos、Katia Papakonstantinopoulou 和 Yannis Kotidis。2022 年。《Chimp：面向时间序列数据库的高效无损浮点压缩》。PVLDB 15，11（2022），3058–3070。[34] Chunwei Liu、Hao Jiang、John Paparrizos 和 Aaron J Elmore。2021 年。《用于快速压缩与查询的分解有界浮点数》。PVLDB 14，11（2021），2586–2598。[35] Jinxin Liu、Petar Djukic、Michel Kulhandjian 和 Burak Kantarci。2024 年。《Deep Dict：面向物联网数据的基于深度学习的有损时间序列压缩器》。载于 ICC。1–6。[36] Stathis Maneas、Kaveh Mahdaviani、Tim Emami 和 Bianca Schroeder。2022 年。《企业存储系统中固态硬盘的运行特性：大规模实地研究》。载于 USENIX。165–180。[37] Ningting Pan、Peng Wang、Jiaye Wu 和 Wei Wang。2018 年。《MTSC：一种有效的多时间序列压缩方法》。载于 DEXA。267–282。[38] Tuomas Pelkonen、Scott Franklin、Justin Teller、Paul Cavallaro、Qi Huang、Justin Meza 和 Kaushik Veeraraghavan。2015 年。《Gorilla：一种快速、可扩展的内存时序数据库》。PVLDB 8，12（2015），1816–1827。[39] Chen Wang、Xiangdong Huang、Jialin Qiao、Tian Jiang、Lei Rui、Jinrui Zhang、Rong Kang、Julian Feinauer、Kevin A. McGrail、Peng Wang、Diaohan Luo、Jun Yuan、Jianmin Wang 和 Jiaguang Sun。2020 年。《Apache IoTDB：面向物联网的时间序列数据库》。PVLDB 13，12（2020），2901–2904。[40] Runhui Wang 和 Dong Deng。2020 年。《DeltaPQ：面向高维相似性搜索的无损乘积量化编码压缩》。PVLDB 13，13（2020），3603–3616。[41] Tianrui Xia、Jinzhao Xiao、Yuxiang Huang、Changyu Hu、Shaoxu Song、Xiang- dong Huang 和 Jianmin Wang。2024 年。《Apache IoTDB 中的时间序列数据编码：比较分析与建议》。VLDBJ 33，3（2024），727–752。[42] Lei Yan、Jiayu Han、Runnan Xu 和 Zuyi Li。2021 年。《面向智能电网实时低延迟传输的无模型无损数据压缩》。TSG 12，3（2021），2601–2610。[43] Xianjun Yang、Xiaofeng Tao、Eryk Dutkiewicz、Xiaojing Huang、Y. Jay Guo 和 Qimei Cui。2013 年。《基于压缩感知与网络编码的无线传感器网络节能分布式数据存储》。IEEE TWC 12，10（2013），5087–5099。[44] Zehai Yang 和 Shimin Chen。2023 年。《MOST：结合离群值存储的时间序列基于模型的压缩》。SIGMOD 1，4（2023），1–29。[45] Yuanyuan Yao、Lu Chen、Ziquan Fang、Yunjun Gao、Christian S. Jensen 和 Tianyi Li。2024 年。《Camel：浮点时间序列的高效压缩》。SIGMOD 2，6（2024），1–26。[46] Xinyang Yu、Yanqing Peng、Feifei Li、Sheng Wang、Xiaowei Shen、Huijun Mai 和 Yue Xie。2020 年。《在时间序列数据库中使用机器学习进行两级数据压缩》。载于 ICDE。1333–1344。

A 支撑理论与证明

A.1 Elf 中的平滑度损失

我们给出一个公式，用于量化转换后的**平滑度损失**：

$$\hat{q} = \min(v_X.q, v_Y.q), \tag{7}$$

$$S(v_X, v_Y) = \mathrm{CBL}\left(\mathrm{abs}((v_X \times 10^{-\hat{q}}) - (v_Y \times 10^{-\hat{q}}))\right), \tag{8}$$

平滑度损失定义如下：

$$\mathrm{Loss} = S(\hat{v}_X, \hat{v}_Y) - S(v_X, v_Y). \tag{9}$$

某些算法（如 Elf）会引入与精度相关的误差，从而破坏平滑性。具体而言，对于 Elf，其损失可表示为：

$$\mathrm{Loss}(\mathrm{Elf}) = S(v_X - \delta_{v_X}, v_Y - \delta_{v_Y}) - S(v_X, v_Y). \tag{10}$$

此处，δ_{v_X} 表示擦除后值与原始值之间的差异，可表示为：

$$\delta_{v_X} = sign \times 2^{(exp-bias)} \times \text{尾数}_{(\text{lower } g(v_X) \text{ 比特})}, \tag{11}$$

原始值的低位 $g(v_X)$ 比特会被擦除。

Elf 作者最初定义了这一概念，用本文的语言可以描述为：

$$g(v_X) = 52 - (\lceil (-q) log_2(10) \rceil + exp - bias). \tag{12}$$

应用异或运算时，会出现精度不匹配。具体而言，当 $g(v_X) \neq g(v_Y)$ 时，可得 $\delta_{v_X} \neq \delta_{v_Y}$。因此，Elf 算法最多会破坏已擦除的 CBL 中 abs $(g(v_X) - g(v_Y))$ 比特。

尽管 Elf 试图结合冗余消除和平滑度利用，但示例表明，擦除后再进行异或会导致较差的结果。反过来调整顺序（即先进行异或，再进行擦除）同样存在问题。此外，异或运算后精度变得无法控制，使得这类优化难以有效实现。

A.2 十进制异或转换器的零损失

我们从形式上证明，十进制异或转换器不会造成平滑度损失。

引理 5（十进制异或的 零损失）。 十进制异或的平滑度损失满足：$\forall v_X, v_Y \in \mathbb{R}$, 损失（$v_X \diamond v_Y$） $= 0$.

证明。 根据第 4.2.1 节中定义的 十进制异或操作，得到：

$$\hat{v}_X = \underline{v_X} \diamond v_Y = v_X - \alpha, \quad \hat{v}_Y = v_Y - \alpha, \tag{13}$$

其中，α 表示 v_X 与 v_Y之间的共享前缀。

令 $\hat{q} = \min(v_X.q,\ v_Y.q)$。接下来，我们计算平滑性指标 $S(\hat{v}_X, \hat{v}_Y)$,

$$\begin{aligned} S(\hat{v}_X, \hat{v}_Y) &= \mathrm{CBL}\left(\mathrm{abs}((\hat{v}_X \times 10^{-\hat{q}}) - (\hat{v}_Y \times 10^{-\hat{q}}))\right) \\ &= \mathrm{CBL}\left(\mathrm{abs}((v_X \times 10^{-\hat{q}} + \gamma) - (v_Y \times 10^{-\hat{q}} + \gamma))\right), \end{aligned}$$

其中 $\gamma = -\alpha \times 10^{-\hat{q}}$。

由于共享前缀，γ 相互抵消，因此：

$$S(\hat{v}_X, \hat{v}_Y) = \mathrm{CBL}\left(\mathrm{abs}((v_X \times 10^{-\hat{q}}) - (v_Y \times 10^{-\hat{q}}))\right) = S(v_X, v_Y). \tag{14}$$

因此，平滑度损失为：

$$\mathrm{Loss}(v_X \diamond v_Y) = S(\hat{v}_X, \hat{v}_Y) - S(v_X, v_Y) = 0. \tag{15}$$

证明完毕。 □

A.3 无符号二进制后缀的固定比特分配证明

在第 4.3 节中，我们引入引理 4，它指出，对于任意 $\beta_i \in \mathbb{Z}$，固定分配 $\bar{\ell}_i$ 比特比可变分配 ℓ_i能实现更好的压缩。具体而言，有：

$$\mathbb{E}[(4 + \bar{\ell}_i)] < \mathbb{E}[(6 + \ell_i)].$$

详细证明如下：

证明。我们知道条件：

$$\delta = o_i - q_i \in \mathbb{N}, \quad \mathrm{abs}(\beta_i) \in [10^{\delta-1}, 10^{\delta}),$$

$$\ell_i = \lceil \log_2(\mathrm{abs}(\beta_i) + 1) \rceil, \quad \bar{\ell}_i = \lceil \log_2(10^{\delta}) \rceil.$$

我们要证明：

$$\mathbb{E}[(\bar{\ell}_i + 4)] < \mathbb{E}[(6 + \ell_i)] \iff \mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] < 0.$$

我们按 2 的幂对 $\mathrm{abs}(\beta_i)$ 的范围进行划分。令 $j \in \mathbb{N}$ 为满足 $2^j > 10^{\delta-1}$ 的最小整数，即：

$$2^{j-1} \leq \mathbf{10}^{\delta-1} < 2^j < 2^{j+1} < 2^{j+2} < \mathbf{10}^{\delta} < 2^{j+4}.$$

现在，我们分析 2^{j+3} 与 10^{δ} 之间的关系，由此得到两种情况：

情况（1）： $2^{j+3} > 10^{\delta}$，其概率为 $\mathbb{P}_1$。在这种情况下，固定分配为 $\bar{\ell}_i = \lceil \log_2(10^{\delta}) \rceil = j + 3$，而可变分配$\ell_i$ 为：

$$\ell_i = \begin{cases} j & \text{if } \mathrm{abs}(\beta_i) \in [10^{\delta-1}, 2^j), \\ j+1 & \text{if } \mathrm{abs}(\beta_i) \in [2^j, 2^{j+1}), \\ j+2 & \text{if } \mathrm{abs}(\beta_i) \in [2^{j+1}, 2^{j+2}), \\ j+3 & \text{if } \mathrm{abs}(\beta_i) \in [2^{j+2}, 10^{\delta}). \end{cases}$$

要计算这种情况下的期望，我们有：

$$\mathbb{E}_1 = \mathbb{E}_{\ell_i \mid 2^{j+3} > 10^{\delta}}[\ell_i] = j + 1 \cdot \mathbb{P}_{\ell_i \mid 2^{j+3} > 10^{\delta}}(\ell_i = j+1) + \cdots$$

现在，让我们计算 $\mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta})$。这可以表示为：

$$\begin{aligned} \mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta}) = &\frac{(j+3)(10^{\delta} - 2^{j+2})}{10^{\delta} - 10^{\delta-1}} + \frac{(j+2)(2^{j+2} - 2^{j+1})}{10^{\delta} - 10^{\delta-1}} \\ &+ \frac{(j+1)(2^{j+1} - 2^{j})}{10^{\delta} - 10^{\delta-1}} + \frac{j(2^{j} - 10^{\delta-1})}{10^{\delta} - 10^{\delta-1}}. \end{aligned}$$

此表达式可进一步简化如下：

$$\begin{aligned} \mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta}) &= j + \frac{3(10^{\delta} - 2^{j+2}) + 2(2^{j+2} - 2^{j+1}) + (2^{j+1} - 2^{j})}{10^{\delta} - 10^{\delta-1}} \\ &= j + \frac{3 \times 10^{\delta} - 2^{j+2} - 2^{j+1} - 2^{j}}{10^{\delta} - 10^{\delta-1}} \\ &= j + 3 - \frac{2^{j+2} + 2^{j+1} + 2^{j} - 3 \times 10^{\delta-1}}{10^{\delta} - 10^{\delta-1}}. \end{aligned}$$

接下来，我们对各项作如下近似：

$$\mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta}) > j + 3 - \frac{2^{j+2} + 2^{j+1} + 2^{j} - 3 \times 2^{j-1}}{2^{j+2} - 2^{j}}.$$

化简上述表达式后，我们得到：

$$\mathbb{E}(\ell_i \mid 2^{j+3} > 10^{\delta}) > j + \frac{7}{6}.$$

因此，期望可简化为：

$$\mathbb{E}_1 = j + 1 - \mathbb{E}(\ell_i \mid 2^{j+3} > 10^\delta) < -\frac{1}{6}.$$

现在计算情况（1）的概率 $\mathbb{P}_1$：

$$\mathbb{P}_1\{2^{j+3} > 10^\delta\} = \mathbb{P}\{(j+3)\log_{10} 2 > \delta\}.$$

根据条件 $2^j > 10^{\delta-1}$，我们有：

$$j\log_{10} 2 + 1 > \delta.$$

类似地，由 $2^{j-1} \le 10^{\delta-1}$可得：

$$(j-1)\log_{10} 2 + 1 \le \delta.$$

因此，δ的值位于以下范围内：

$$\delta \in \big[(j-1)\log_{10} 2 + 1, j\log_{10} 2 + 1\big).$$

I总之，我们可以按如下方式计算可能性 ws:

$$\mathbb{P}\{(j+3)\log_{10} 2 > \delta\} = \frac{(j+3)\log_{10} 2 - (j-1)\log_{10} 2 - 1}{\log_{10} 2}$$

$$= \frac{4\log_{10} 2 - 1}{\log_{10} 2} \implies \mathbb{P}_1\{2^{j+3} > 10^\delta\} = 4 - \log_2 10 \approx 0.6781.$$

情况（2）： $2^{j+3} \le 10^\delta$，其概率为：

$$\mathbb{P}_2\{2^{j+3} \le 10^\delta\} = 1 - \mathbb{P}_1 \approx 0.3219.$$

很容易看出，对于 $j \ge 0$、$2^{j+3} \ne 10^\delta$，因此我们可以将 $2^{j+3} \le 10^\delta$ 替换为严格不等式 $2^{j+3} < 10^\delta$。

在这种情况下，固定分配为 $\bar{\ell}_i = \lceil \log_2(10^\delta) \rceil = j + 4$，而可变分配 ℓ_i 如下：

$$\ell_i = \begin{cases} j & \text{if } \mathrm{abs}(\beta_i) \in [10^{\delta-1}, 2^j), \\ j+1 & \text{if } \mathrm{abs}(\beta_i) \in [2^j, 2^{j+1}), \\ j+2 & \text{if } \mathrm{abs}(\beta_i) \in [2^{j+1}, 2^{j+2}), \\ j+3 & \text{if } \mathrm{abs}(\beta_i) \in [2^{j+2}, 2^{j+3}), \\ j+4 & \text{if } \mathrm{abs}(\beta_i) \in [2^{j+3}, 10^\delta). \end{cases}$$

我们首先按如下方式计算期望：

$$\mathbb{E}_2 = \mathbb{E}\ \bar{\ell}_i + 4\ |(\,)\,(\,)\,(\,)。$$

$2^{j+3} < 10^\delta$ $\mathbb{E}\ \ell_i + 6 \mid 2^{j+3} < 10^\delta = j+2 - \mathbb{E}\ell_i \mid 2^{j+3} < 10^\delta$

接下来，我们按如下方式计算期望 $\mathbb{E}(\ell_i \mid 2^{j+3} < 10^\delta)$：

$$\mathbb{E}(\ell_i \mid 2^{j+3} < 10^\delta) = j + \frac{4\times 10^\delta - 2^{j+3} - 2^{j+2} - 2^{j+1} - 2^j}{10^\delta - 10^{\delta-1}}$$

$$= j + 4 - \frac{2^{j+3} + 2^{j+2} + 2^{j+1} + 2^j - 4\times 10^{\delta-1}}{10^\delta - 10^{\delta-1}}.$$

对各项进行近似后，我们得到：

$$\mathbb{E}(\ell_i \mid 2^{j+3} < 10^\delta) > j+4-\frac{2^{j+3} + 2^{j+2} + 2^{j+1} + 2^j - 4\times 2^{j-1}}{2^{j+3} - 2^j} = j+\frac{15}{7}.$$

因此，期望可简化为：

$$\mathbb{E}_2 = j + 2 - \mathbb{E}(\ell_i | 2^{j+3} < 10^\delta) < -\frac{1}{7}.$$

最后，我们计算总体期望：

$$\mathbb{E}[(\bar{\ell}_i - \ell_i - 2)] = \mathbb{E}_1 \times \mathbb{P}_1 + \mathbb{E}_2 \times \mathbb{P}_2 < -\frac{1}{6}\times\mathbb{P}_1 - \frac{1}{7}\times\mathbb{P}_2 \approx -0.159 < 0.$$

□