

Chimp: 时间序列数据库的高效无损浮点数压缩

Chimp: Efficient Lossless Floating Point Compression for Time Series Databases

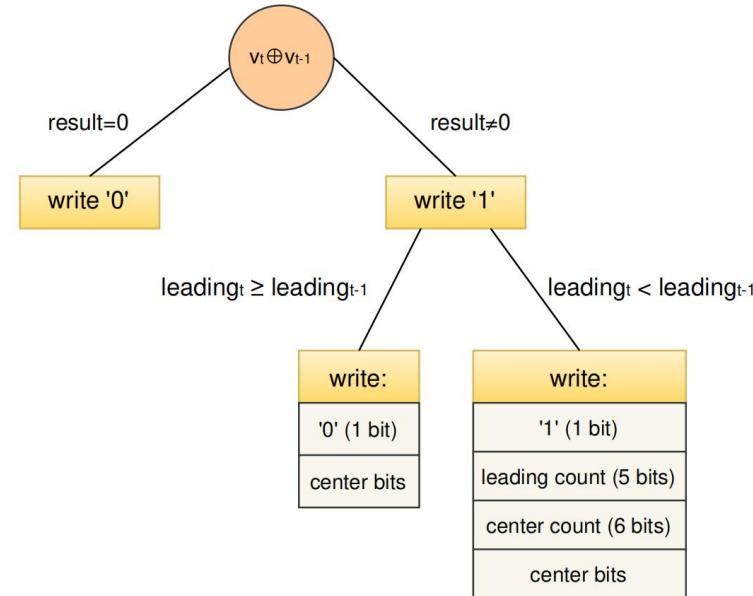
Panagiotis Liakos, Katia Papakonstantinopoulou, Yannis Kotidis

背景

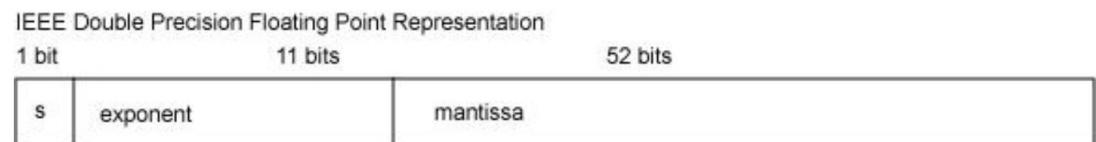
- 现实世界中，医学、金融、环境等多方面产生了数以PB计的数据，产生了**过高的存储成本和传输成本，并降低了查询和分析的速度。**
- 但是，使用通用的重型压缩算法如LZ4、Snappy等的**代价非常高**，严重阻碍数据的摄取。
- 因此，许多快速的流式压缩算法被提出，包括Gorilla、FPC等一系列轻量级压缩算法。
- 本文在Gorilla的基础之上又提出了新的改进措施，使其更适配目前的数据。

背景-Gorilla算法

- 是一种适用于IEEE 754标准中规定的双精度浮点数字的流式压缩方案。
- 由Facebook于2015年提出，作为内存时间序列存储引擎的一部分，后来被开源。
- 目前被用作大多数知名时间序列数据库中浮点测量值的默认编码，包括InfluxDB、M3.6等。



Gorilla算法流程(浮点)



IEEE 754标准双精度浮点数表示法

Gorilla算法-时间戳的压缩

- “...我们注意到，绝大多数数据点都是以固定间隔到达的。...”
- Facebook的工程师注意到绝大多数时间点都是固定间隔的，因此提出了使用Delta Of Delta方法压缩时间戳。

时间戳的压缩方法

(1) 压缩块的块头存储开始的时间戳 t_{-1}

t_{-1} | Header:

March 24, 2015 02:00:00

(2) 第一个数据 t_0 的时间戳以 Δ 的方式存储：

t_0 压缩为：

62



$t_0 - t_{-1}$
02:01:02 - 02:00:00

Gorilla算法-时间戳的压缩

(3)从第2个数据开始，对每一个数据 t_n ，计算Delta Of Delta即 $DD_n = (t_n - t_{n-1}) - (t_{n-1} - t_{n-2})$ 并且，根据 D 的大小先存储一个标志位Flag：

1. $DD = 0$, Flag = '0'
2. $DD \in [-63, 64]$, Flag = '10'
3. $DD \in [-255, 256]$, Flag = '110'
4. $DD \in [-2047, 2048]$, Flag = '1110'
5. others, Flag = '1111'

每个时间戳压缩为“Flag: DD”的形式，以下以 t_1 为例：

$$t_{-1} = 02:00:00$$

$$t_0 = 02:01:02$$

$$t_1 = 02:02:02$$

$$D_{0,-1} = t_0 - t_{-1} = 62$$

$$D_{1,0} = t_1 - t_0 = 60$$

$$\begin{aligned} DD_1 &= D_{1,0} - D_{0,-1} = 60 - 62 = -2 \\ -2 &\in [-63, 64], \text{Flag} = '10' \end{aligned}$$



t_1 压缩为：
(Flag)'10':(DD) - 2

Gorilla算法-浮点数压缩算法

- “...通过分析ODS数据，我们发现大多数时间序列中的值与其相邻数据点相比没有显著变化。...”
- 与时间戳类似，作者基于观察提出了一种仅将前后值简单异或，则其中的几位将相同(在XOR中为0)。

浮点数的压缩方法

(1)第一个值 V_0 直接存储不压缩：

V_0 :

35

(2)从第二个值开始，下一个值与前一个值作XOR后若为0(两值相等)，则仅存储 $Flag$:’0’

$$\begin{aligned}V_0 &= 35 \\V_1 &= 35 \\XOR_1 &= V_1 \oplus V_2 = 35 \oplus 35 = 0 \\Flag &= '0'\end{aligned}$$



V_1 压缩为：
 $(Flag)'0'$

Gorilla算法-浮点数压缩算法

(3)若XOR值不为0，先存储Flag: '1'，并分为两种情况：

1. 如果现XOR值落在前XOR值之中，即现XOR值前导0个数和尾随0个数都至少不小于前XOR值时，再存储Flag : '0'，然后存储现XOR值有意义的位。(演示的值省略了一部分值)



$$XOR_{V_{n-1}} = 0001\ 0110B$$

现XOR值有意义位的位置落在了前XOR值的有意义位中

$$XOR_{V_n} = 0000\ 0110B$$

$$H_n = 5 > H_{n-1} = 3$$

$$T_n = 1 = T_{n-1}$$

因此 V_n 的存储方式是：

(Flag)'10': (M)'0011'

位长度为2 + 4 = 6

Gorilla算法-浮点数压缩算法

(3)若XOR值不为0, 先存储 Flag : '1', 并分为两种情况:

2.如果现XOR值与前XOR值关系不大, 则:

位数	存储
5位	前导0的个数 H
6位	有意义位的个数 \bar{M}
剩余	有意义位的内容 M

例如: (演示值省略了一部分值)

$$XOR_{V_n} = \dots 0010\ 0000 \dots B$$

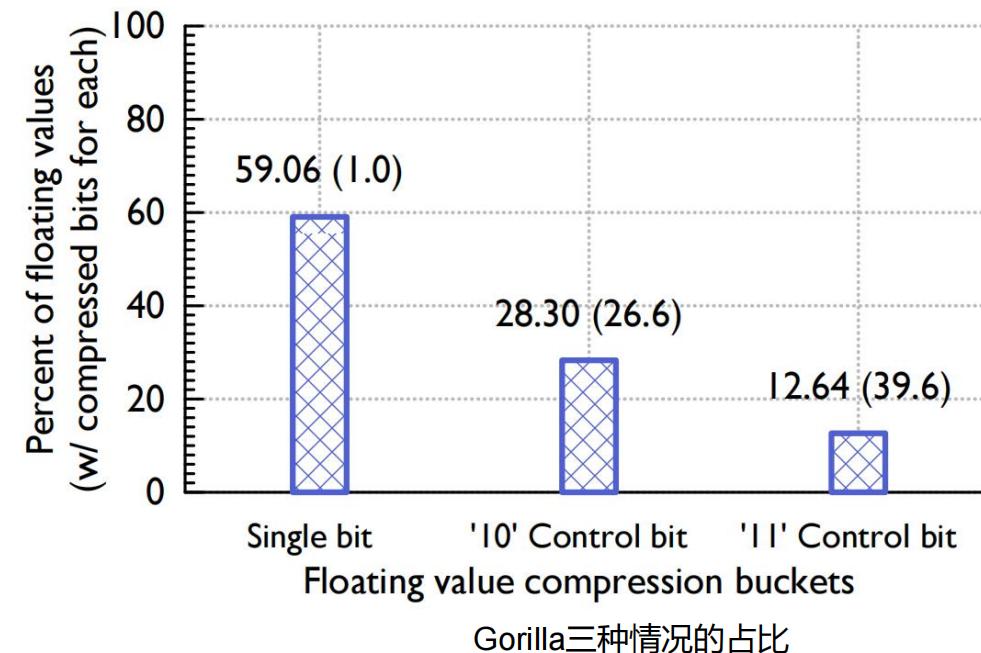
现值不落在前值内, V_n 的存储方式是:

(Flag)'11': (H)2: (\bar{M})1: (M)'1'

位长度为 $2 + 5 + 6 + 1 = 14$

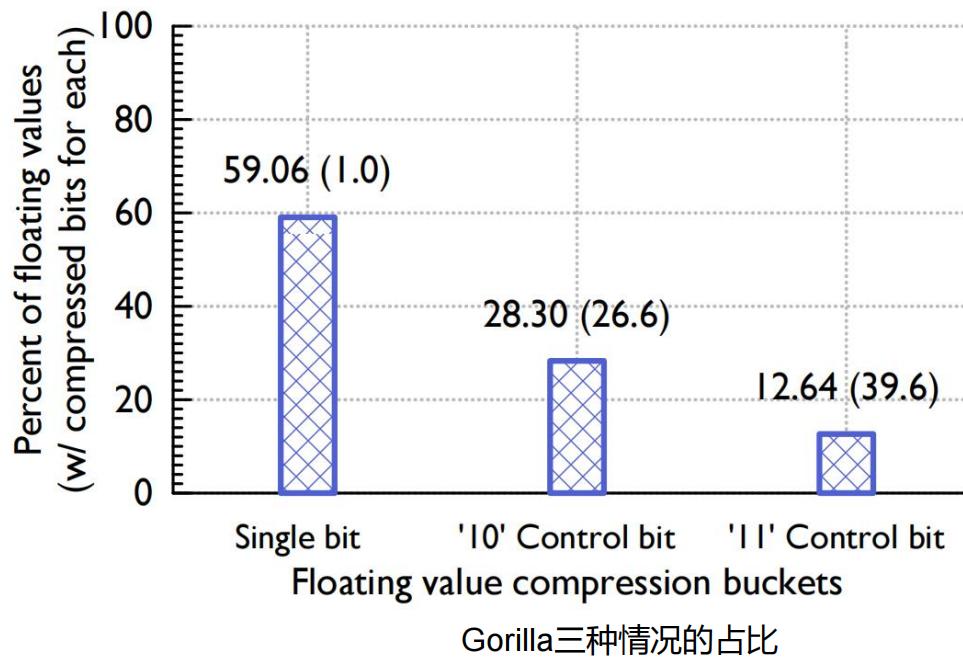
意义在于省略了尾随0个数的存储。

另外, 下图显示了Gorilla算法的三种情况的占比:
可见, 单独'0'也就是前后XOR值相同情况最多, '10'其次, 最复杂的'11'最少。



真实世界的时间序列数据特性

另外，下图显示了Gorilla算法的三种情况的占比：
可见，单独'0'也就是前后XOR值相同情况最多，'10'其次，最复杂的'11'最少。



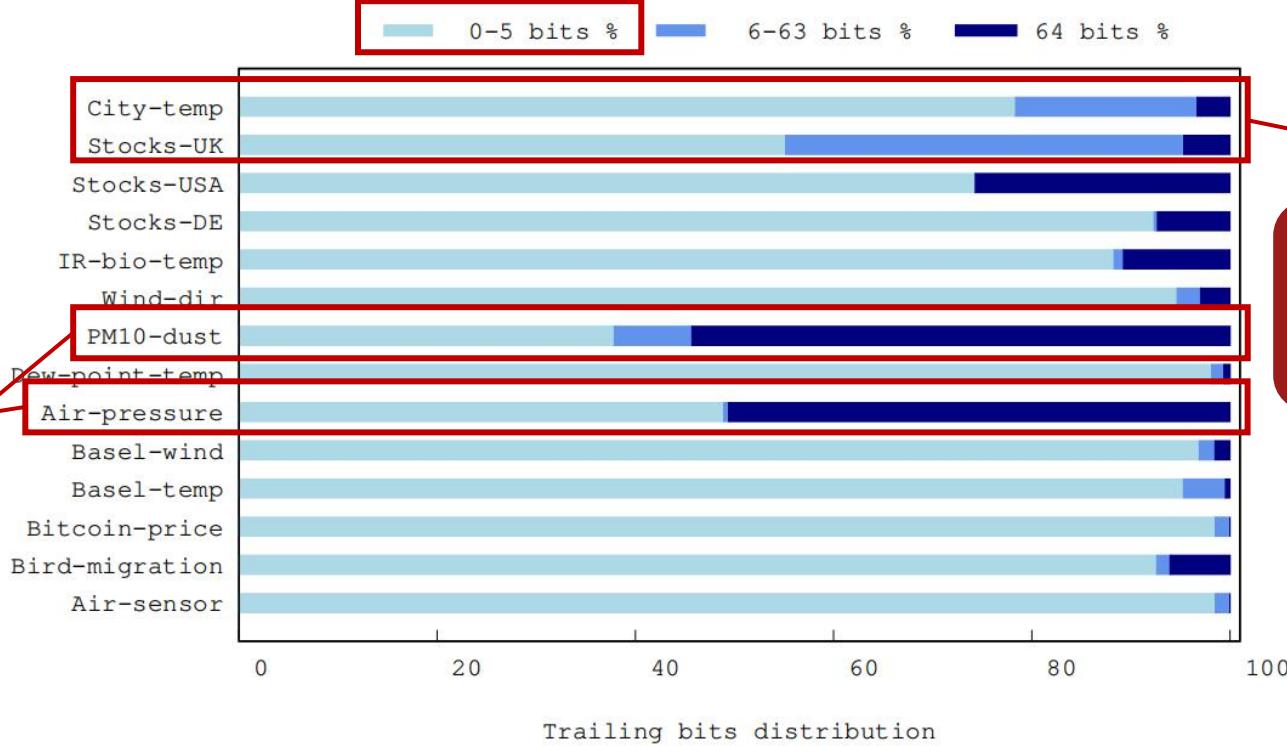
- 在Gorilla的论文中，作者们提到Gorilla适配的是2015年之前Facebook运营时产生的数据。
- 在我们今天介绍的这篇论文中，作者们提出了质疑：
Gorilla能否适合目前的时序数据特征？
- 针对这个问题，作者们选择了14个真实的常用时序数据，对真实世界的时序数据特性展开了调查。

浮点时序数据特性-尾随0

- 通过依次取XOR值，作者得到了14个数据集的平均尾随0的个数，且得到了与Gorilla相反的结果：

2.大多数数据集的尾随0个数都在0-5个，但是Gorilla却用6个位保存尾随0个数

1.除了PM10-dust和Air-pressure两个数据集外，很少有64个0的尾随0，即前后两个XOR值很少相等。



3.只有少数几个数据集由于精度问题导致很多值的尾随0个数在6-63个位，如City-Temp。

14个真实数据集的XOR值尾随0个数

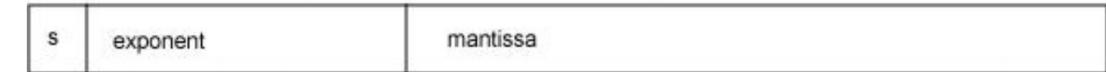
浮点时序数据特性-前导0

IEEE Double Precision Floating Point Representation

1 bit

11 bits

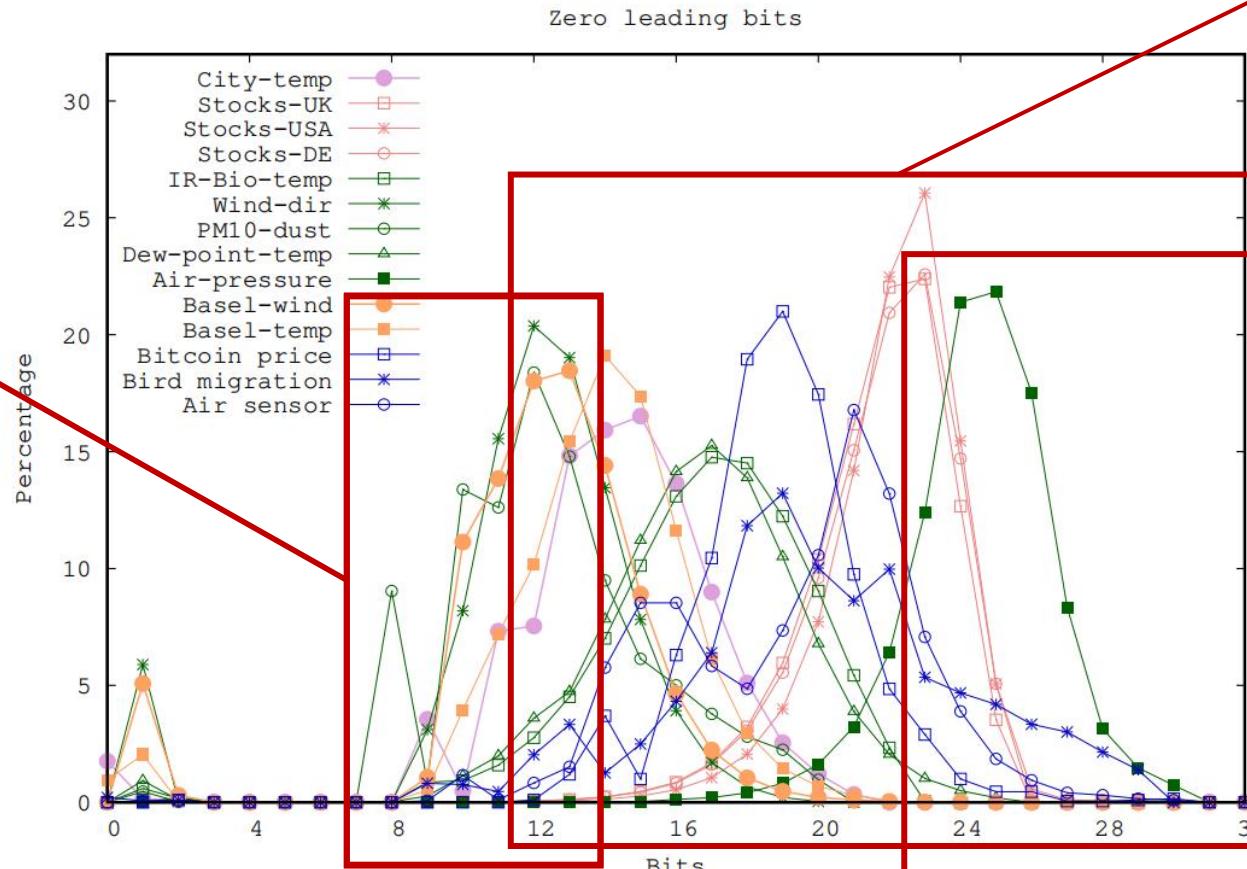
52 bits



- 通过依次取XOR值，作者得到了14个数据集的平均前导0的个数：

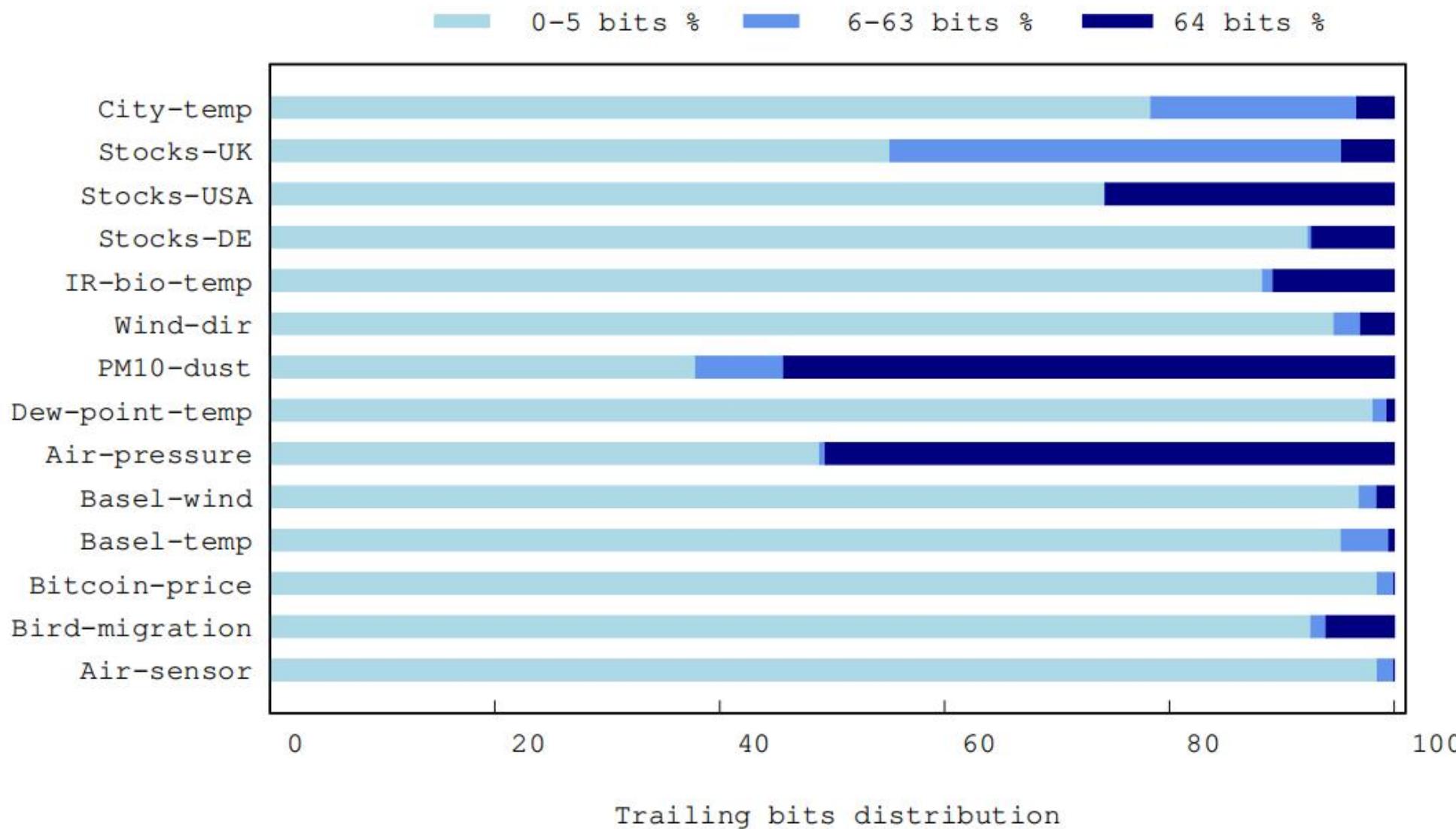
1. 大多数的数据集的前导0个数都要超过12个，意味着符号和指数部分都相等。

2. 少数表现出8-12个前导0，意味着指数不相同，但非常相似。



3. 唯一个产生大量前导0的数据集Air-pressure是由于其较大的整数形式。即使这样，前导0的个数也很少超过30个

回顾Gorilla-标志位



很少有两

两个值相
的情况代

应该越短。

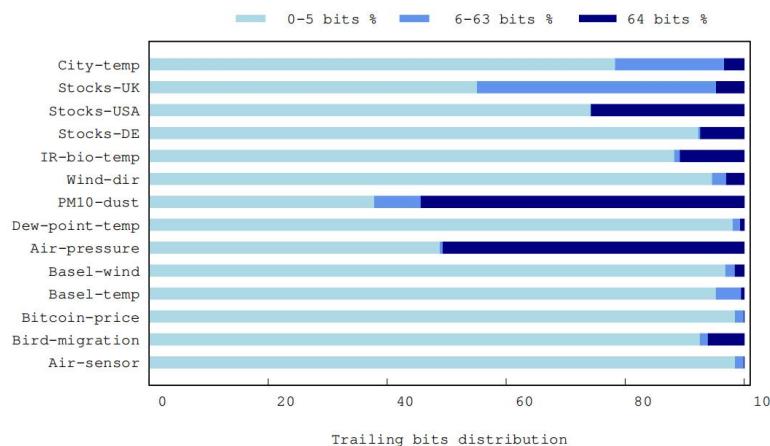
回顾Gorilla-有意义的位个数

- 回顾Gorilla算法的Flag' 11' 情况,

位数	存储
5位	前导0的个数 H
6位	有意义位的个数 \bar{M}
剩余	有意义位的内容 M

为什么要存储有意义位的个数?

- 其目的是,由于尾随0的个数通常比有意义位的个数更多,所以选择存储更少的有意义位个数,避免存储更长的尾随0个数。
 - 尾随0个数可以通过总长度-前导0个数-有意义位个数得到



问题在于,大多数数据的尾随0只有0-5位,并不长,但Gorilla却要花费6位去存储更长的有意义位个数。造成了浪费。

回顾Gorilla-利用前XOR值的位置

- 回顾Gorilla算法的Flag' 10' 情况，即现值的有意义部分落在前值有意义部分内

$$XOR_{V_{n-1}} = 0001\ 0110B$$

现XOR值有意义位的位置落在了前XOR值的有意义位中

$$XOR_{V_n} = 0000\ 0110B$$

$$H_n = 5 > H_{n-1} = 3$$

$$T_n = 1 = T_{n-1}$$

因此 V_n 的存储方式是：

(Flag)'10':(M)'0011'

位长度为 $2 + 4 = 6$

虽然利用了前值的位置，但是还是多存储了两个0，
这两个0被称为未利用的(unexploited)0。

Dataset	Av. leading bits		Av. trailing bits	
	total	unexploited	total	unexploited
City-temp	14.18	8.85	9.20	9.19
Stocks-UK	21.71	5.38	17.26	2.44
Stocks-USA	22.14	4.62	0.99	0.96
Stocks-DE	21.86	4.84	1.1	1.09
IR-bio-temp	16.84	7.45	1.28	1.27
Wind-dir	11.95	7.27	2.04	2.03
PM10-dust	12.52	4.36	7.69	7.36
Dew-point-temp	16.59	5.54	1.19	1.19
Air-pressure	24.62	6.3	1.11	1.09
Basel-wind	12.19	10.71	1.07	1.06
Basel-temp	13.69	8.01	2.31	2.31
Bitcoin-price	18.9	5.25	1.07	1.07
Bird-migration	20.05	7.23	1.02	1.01
Air-sensor	19.27	5.0	0.99	0.99
Average	17.61	6.49	3.45	2.36

实际上，根据作者的统计，未利用的前导0个数平均为6.49个，超过了为了存储前导0个数的5位。

未利用的尾随0的个数平均有2.36个。
且大部分数据的尾随0的利用率很低。

这部分未利用的0造成了很大浪费。

Chimp算法

- 根据实际数据中暴露出的Gorilla算法的不足，作者提出了Chimp算法。

Chimp算法

- (1) 第一个值不压缩，直接存储。
- (2) 进行判断，当XOR值尾随0个数超过6时，添加Flag: '0'，然后判断：
 - (2.1) 若 $XOR_n = 0$ ，则再存储一个Flag: '0'，例如：

0000 0000B



(Flag)'00'

- (2.2) 若 $XOR_n \neq 0$ ，则再存储一个Flag: '1'，然后：

位数	存储
3-5位	前导0的个数H
6位	有意义位的个数M
剩余	有意义位的内容M

0000 0011 0000 0000B



(Flag)'01': (H)6: (M)2: (M)'11'

Chimp算法

(3) 若XOR值尾随0个数不超过6个时，添加 $Flag'1'$ ，然后判断：

(3.1) 若现XOR值的前导0个数等于前XOR值的前导0个数，则现值使用前值的前导0个数的信息，先记录 $Flag'0'$ ，然后再存储有意义位M和完整的尾随0 T_Z （而不是尾随0个数），例如：

假设前值的前导0个数等于现值的前导0个数：

...0000 0000 1111 0000B



$(Flag)'10': (M)1111: (T_Z)0000$

(3.2) 若不相等，则：

3~5位存储 前导0个数H

剩余存储 有意义位M

例如，假设前值的前导0个数不等于现值的前导0个数：

0000 0000 11...11 0000B



$(Flag)'11': (H)8: (M)11..11: (T_Z)0000$

Chimp VS Gorilla

- Chimp算法在哪些方面做了改进？

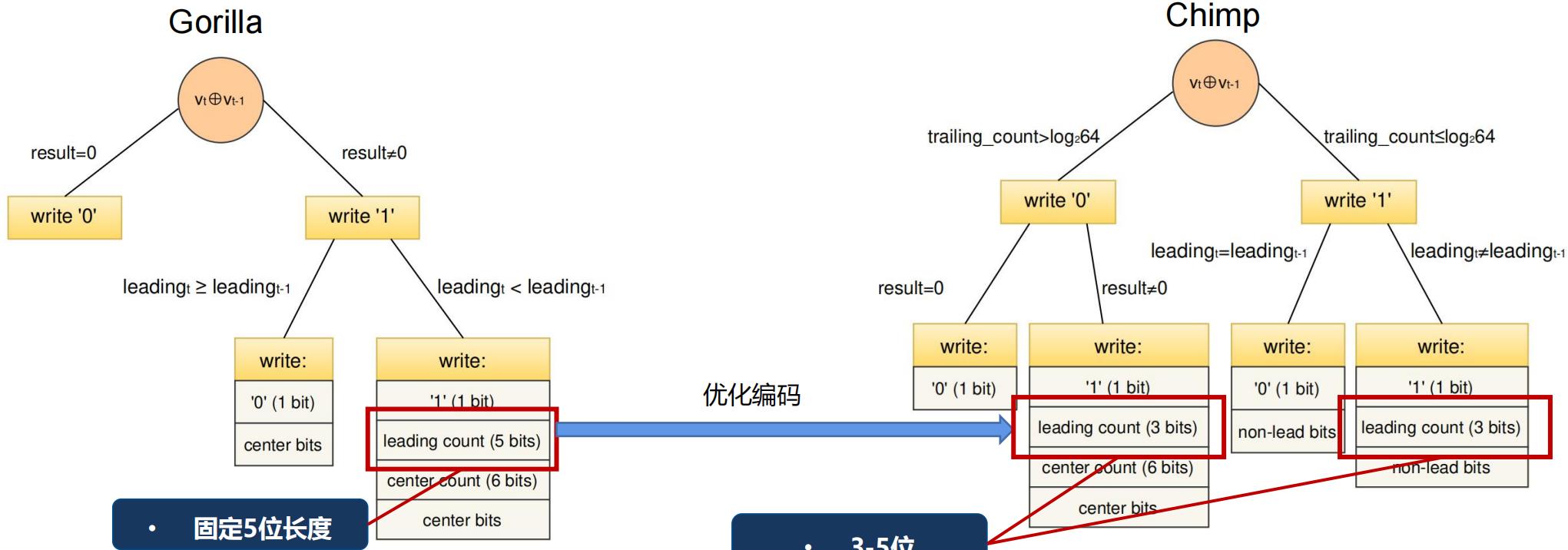
Chimp算法修改了Flag' 0' 的含义，
重用了Flag位' 0'，使其能够添加一种新的情况



Chimp VS Gorilla

- Chimp算法在哪些方面做了改进？

Chimp算法有效缩短了表示前导0长度的位数，
由固定5位优化到3-5位：



Chimp VS Gorilla

- Chimp算法如何将表示前导0个数的位数优化到3-5位?

前导0的个数 H 的取值范围为 $H \in [0, 31]$, 即正常来说, 用五个二进制位的即可表示。

假设 H 为奇数, 设 $H' = \frac{H-1}{2}$, 则 $H' \in [0, 15]$, 即 H' 可用四个二进制位表示, 再存储一个二进制位表示为奇数, 则还是用五个二进制位表示 H ;

假设 H 为偶数, 设 $H' = \frac{H}{2}$, 则 $H' \in [0, 15]$, 即 H' 可用四个二进制位表示, 由于无需再加一个位表示偶数, 则用四个二进制位表示 H , 节省了一位。

简单地说, 将所有前导0的个数除以二存储。

例如:



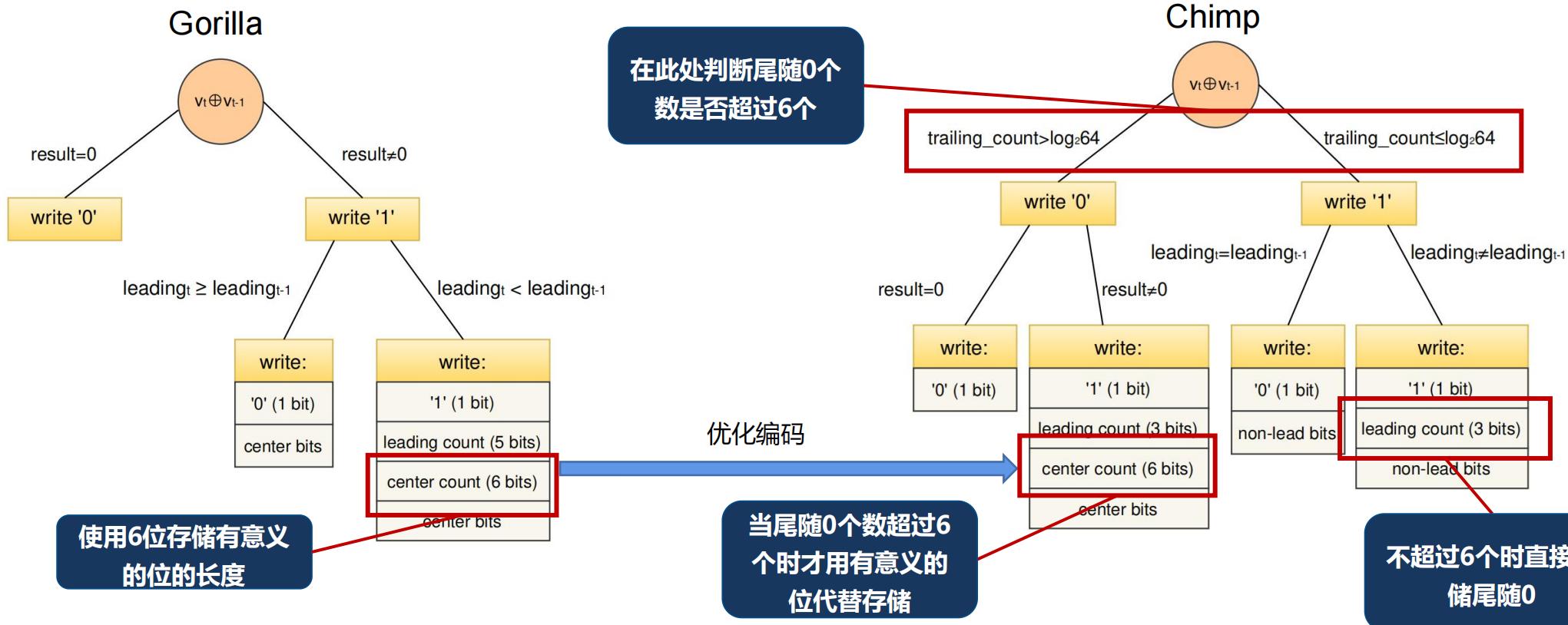
进一步地, 作者选出固定的8个常用的 H 值(如0,8,16,22等), 即可使用三个二进制位表示。

Chimp VS Gorilla

- Chimp算法在哪些方面做了改进？

Chimp算法优化了有意义位的长度的表示：

当尾随0个数小于6个时，不存储有意义位的个数而是存储有意义的位+尾随0（不是个数）



Chimp VS Gorilla

- Chimp算法在哪些方面做了改进？

Chimp算法优化了有意义位的长度的表示：
当尾随0个数小于6个时，不存储有意义位的个数而是存储有意义的位+尾随0（不是个数）

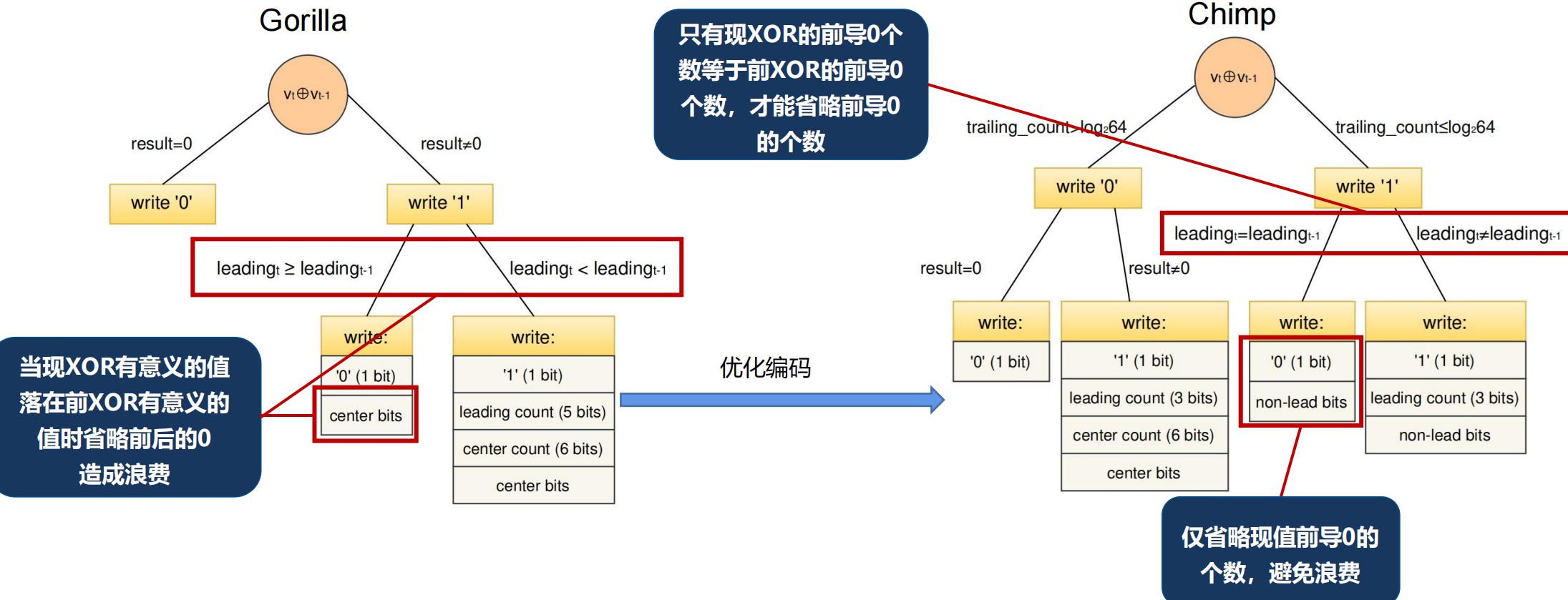
例如：



Chimp VS Gorilla

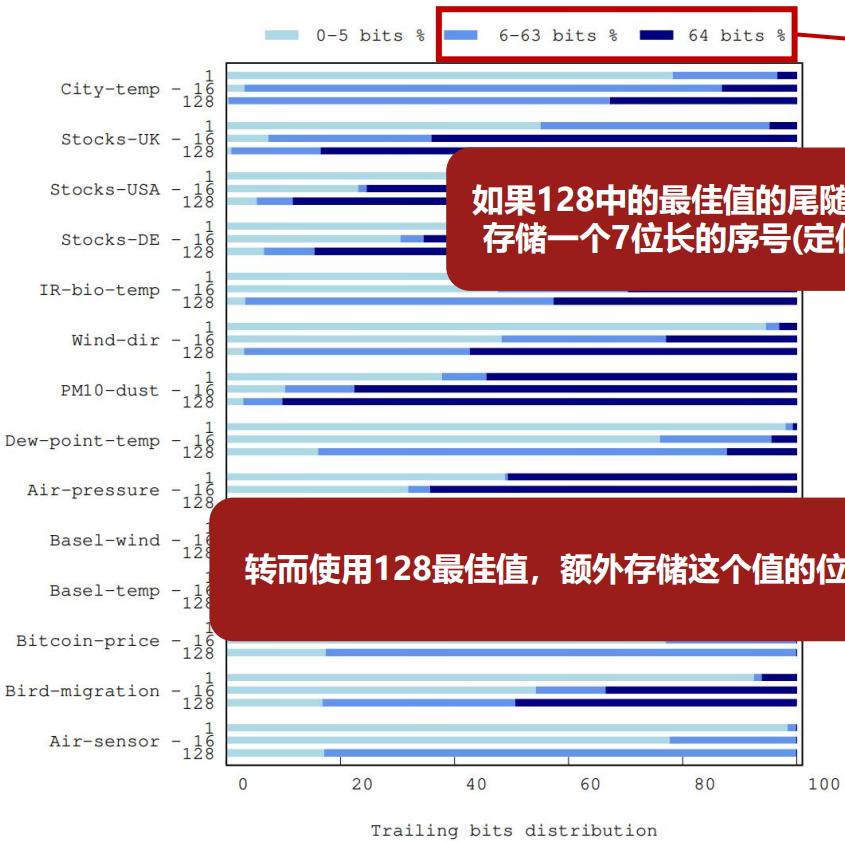
- Chimp算法在哪些方面做了改进？

Chimp算法避免了在利用前XOR值时导致的0的浪费：



Chimp128

通过将现值与前1个值、前16个值和前128个值作XOR后取最佳值，作者们观察到了一个现象：

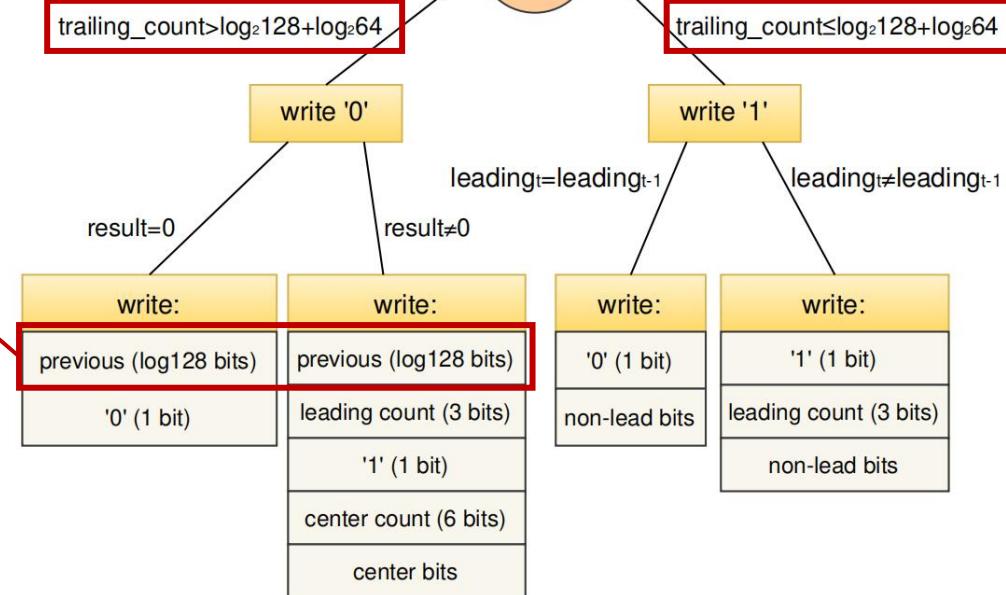


转而使用128最佳值，额外存储这个值的位置

对不同个数的前值作XOR后选取其中最多的尾随0个数

使用的前值越多，尾随0的个数就越多
基于此结论，作者们提出了Chimp128：

否则还是按照原Chimp算法计算



实验-压缩率

流式算法通常压缩率方面不如重型的通用算法

Dataset		General Purpose Compression					Streaming Compression			
		Xz	Brotli	LZ4	Zstd	Snappy	FPC	Gorilla	CHIMP	CHIMP ₁₂₈
Time series	City-temp	14,04	15,31	27,64	17,90	24,30	55,16	58,72	46,21	22,92
	Stocks-UK	7,61	8,54	19,84	10,32	15,80	46,15	33,45	31,27	16,70
	Stocks-USA	7,19	8,11	18,16	9,92	14,68	36,02	36,43	34,67	12,06
	Stocks-DE	8,80	9,96	20,63	12,06	16,83	44,54	45,63	42,88	13,46
	Bitcoin-price	13,82	16,05	29,13	20,19	25,58	48,52	50,33	46,39	18,94
	Air-pressure	12,66	14,98	26,95	17,88	22,02	58,12	59,62	54,31	19,80
	Basel-wind	6,55	7,21	15,03	8,50	12,52	27,79	26,91	24,40	13,64
	Basel-temp	20,92	25,16	38,34	29,65	38,60	53,63	54,42	51,57	32,49
	Bird-migration	40,29	46,46	55,20	47,64	63,19	52,22	52,50	49,68	47,17
	Air-sensor	24,97	27,11	35,50	29,12	34,00	48,14	50,24	45,92	28,37
Non time series	Time series average	50,16	54,22	64,32	58,53	64,10	52,56	52,98	49,54	49,56
	Non time series average	20,03	22,29	32,23	24,57	31,09	47,38	47,79	43,57	26,44
Non time series	Blockchain-tr	16,32	17,87	27,65	19,96	26,28	43,53	37,94	27,92	24,59
	SD-bench	39,30					65,95		57,80	47,71
		43,97					66,07		62,71	54,55
		45,00					62,83		58,25	53,16
		8,12					40,25		35,10	17,00
		30,54	32,61	41,19	33,97	42,23	53,16	54,61	48,36	39,40

由于比特币价格变动过为剧烈，限制了算法使用尾随0个数优化，因此效果一般

这个数据集是由随机数生成的，几乎没有相同的数

在流式压缩算法中，Chimp和Chimp128算法领先于目前常用的FPC和Gorilla算法

实验-压缩时间

虽然Chimp算法压缩率要比重型的通用算法低，
但是压缩和解压缩快得多

Algorithm	Compression (μs)		Decompression (μs)	
	Time series	Non time series	Time series	Non time series
Xz	1,679.69	1,641.85	298.67	405.59
Brotli	1,409.54	1,430.53	59.67	69.65
LZ4	1,199.04	1,188.53	25.29	25.63
Zstd	163.27	180.99	56.64	60.29
Snappy	89.56	93.43	34.17	35.12
FPC	56.05	61.73	38.03	46.50
Gorilla	39.28	43.86	31.79	34.86
CHIMP	31.18	31.02	30.68	31.10
CHIMP ₁₂₈	35.81	42.68	28.47	34.38

综合来看，Chimp算法是最快的算法，由于要
与128个前值运算，Chimp₁₂₈稍慢

结论

作者们基于Gorilla算法和现实世界的时序数据特点提出了Chimp算法及其变体Chimp128算法

Chimp和Chimp128算法实现了效率和压缩比的双领先。

在未来，作者计划使用有损压缩技术进一步优化Chimp算法。

结论

作者们基于Gorilla算法和现实世界的时序数据特点提出了Chimp算法及其变体Chimp128算法。

Chimp和Chimp128算法实现了**效率和压缩比**的双领先。

在未来，作者计划使用有损压缩技术进一步优化Chimp算法。

THANKS!