论文讲稿

表1概述了主要特性。为简单起见，我们使用TS 表示一串时间序列的值列表。大多数这些特性都可以在Apache IoTDB中通过我们之前的研究中开发的数据分析工具直接计算出来。

数据规模是存储中最重要的因素之一。一般来说，值越大，我们需要编码的位就越多。如下面的第4节所示，后面将会介绍的基于游程长度的算法RLE需要编码后存储概括信息，其中较大的值需要更多的位。比特打包算法[35]也受到类似的影响。比特打包算法是一种常用数据压缩算法，大体思想是将数据转换为二进制之后，按最小存储单元去掉前导０然后彼此前后拼接重组为更短的数据序列。因此，当大多数值为负时，基于位打包的算法性能很差，因为符号位为1。为此，我们采用时间序列TS中值的均值、方差和差值(最大值减去最小值)，分别用mean (TS)、Var(TS)和spread (TS)表示尺度特征。

delta特征显示了数据波动的幅度，这对时间序列尤其重要。设DS]表示时间序列ＴS的增量序列，用于测量时间相邻值的增量。第4.1节中介绍的基于差分的算法[40]存储这些增量。从这个意义上说，我们使用mean(DS)、Var(DS)和spread(DS)，即DS的均值、方差和差值(最大值减去最小值)来评估差值可能有多大。

在时间序列中广泛观察到重复值，例如几分钟内温度读数不变。这种连续的重复可以通过第4.2节中基于运行长度的算法[27]进行压缩。它们还在第4.1节介绍的异或运算符中输出零值，有效地缩小了空间。为此，我们引入了一种描述时间序列重复的方法。其主要思想是计算连续重复值区间内连续重复值的个数。像4.3节中的SPRINTZ[20]这样的算法，用bit-packing将数据打包成整数字节的数据块大小为8。所以这里我们只关注连续重复次数大于等于8的部分，我们用count这个指标来描述这一特征。

虽然重复值的差值为0，但也要考虑非重复值的差值符号。原因是非零符号位可能会干扰像之后会介绍的RLBE[41]这样的编码算法。如果所有的增量符号都是正的，也就是说，时间序列的值总是在增加，那么编码的性能就会更好。相反，当差分值为负，即减小时，编码性能会变差。在这种意义上，我们定义Count(IS)为相邻时间戳中增加的值的数量，

除了scale、delta、repeat和increase等特征外，数据类型也是影响编码性能的重要因素。对于INT32和INT64，相似值的波动程度小于FLOAT和DOUBLE。此外，较长的INT64和DOUBLE可能有更多的0位，其中可以执行位压缩策略。因此，数据类型也被视为重要的数据特征。

同样，文本时间序列数据也有几个数据特征，这些特征可能与编码性能有关，包括值的分布、值的域、文本值的平均长度和字符的连续重复。表2概述了主要的文本特性。

在实践中，文本值通常遵循Zipfian分布[18,45]。Zipfian分布的指数表示值的频率。Zipfian是一个分布模型，反应的是词频分布情况，是一个长尾模型。指数越大，值频率的偏度越大。这种偏度影响了HUFFMAN编码的性能[29,36]，这与值的频率有关。此外，文本值的域大小也很重要，例如，对DICTIONARY编码[44]，将值域存储为字典。

字符特征可能会影响在字符级别对数据进行编码的编码算法。值的长度当然是一个关键因素。较长的值通常会导致较大的字符编码结果。同样，字符的重复也会影响编码性能，如RLE[27]和HUFFMAN[29,36]。

参考上述对无损需求和系统架构的讨论，我们将会六种适合在Apache IoTDB中实现的编码算法，包括TS\_2DIFF[8]、GORILLA[37]、SPRINTZ[20]、RLE[27]、RLBE[41]和RAKE[21]。实现的源代码可在Apache IoTDB[5]的GitHub存储库中获得。表3列出了不同编码算法之间可能共享的共同思想。我们通过编码的实现手段，将编码算法分为两类，一类是基于差分的编码算法，一类是基于游程长度的编码算法。

首先来介绍基于差分的编码算法。差分编码提出了在时间序列数据连续的情况下，特别是原始数据较大的情况下，减小绝对值的方法。由于绝对值减少，有效位的数量随之减少，从而降低了存储成本。因此，在差分编码算法中，压缩比与增量DS的特征有重要的关系。虽然传统的差分编码只能在单调的整数值中表现良好，但GORILLA和TS\_2DIFF两种算法在其中表现较优

TS\_2DIFF编码是增量的增量delta-of -delta的一种变体。它包括三个步骤:增量编码，二次增量编码，位打包。第一步通过当前值-前一个值来计算每个值的增量。注意，第一个值没有前一个值，应该直接存储。然后，该算法找到最小增量mindiff，并通过将所有的增量数据减去这个mindiff获得要存储的最终数据，也就是所谓的二次增量。最后，去除固定长度二进制数据的前导零，即进行比特打包，得到最终编码的字节流。

因此，序列TS的方差和增量DS方差越小，序列的差位宽度越小，最终压缩比越小，这里定义的压缩比是一个越小越好的值，是压缩后与压缩前数据量的比值。如表4所示，TS\_2DIFF也适用于大的增量平均值，因为在第二次增量编码过程中，通过的最小值本身也比较大，这些值可以得到一个小的值来存储。

图2显示了一个小增量方差的情况，即，第二个增量中的所有增量都很小，因此具有较低的空间成本

GORILLA编码最初是为Facebook的时间序列数据库（TSDB）设计的[37]。首先，它用二阶差分处理时间戳，然后重新写入时间戳。当值以几乎固定的间隔出现时，这是有效的。这些值按有效位宽划分为四个区域。至于值，它使用XOR编码方法。通常，此过程会导致浮点数出现许多前导和尾随零。如果XOR结果为零，它只写一个位“0”来表示它。否则，它会写结果的不同位和前导/尾随零的数量。

如表4所示，GORILLA适用于小方差数据，因为它增加了XOR结果中前导和尾零的数量。另一方面，由于使用了更多的非零位来对值进行编码，它可能会在时间序列上发生剧烈变化而失败。

在图3中，GORILLA通过将5个INT32值的160位压缩为66位，显示了良好的性能。因为时间序列数据的方差很小，有很多前导和尾零。

4.2基于行程长度的编码行程长度编码（RLE）[27]的目标是降低相邻重复值的空间成本。当重复值的次数很小时，传统RLE算法的压缩效果是有限的。

具有比特打包和RAKE[21]的RLE是更有效的进步。

RLE编码[27]存储一个元素的连续重复次数，而不是反复重复相同的元素。例如，序列444556666可以存储为具有游程长度的435264，其中4之后的数字3表示4重复3次。当值连续时，RLE引入额外的空间成本来存储重复次数，因此IoTDB实现将RLE与位打包相结合。游程长度RLE仅应用于重复时间大于8的值。对其他数据进行简单的位打包。

在图4中，序列有许多连续的重复值，这些值也是较小的正数。从而，执行具有比特打包的RLE。

虽然基于差分和基于行程长度的编码算法可以在不同的场景中表现良好，但在某些情况下，既有小的delta特征，也有大量的重复。为此，具有这两种想法的混合编码可以获得更好的结果，例如RLBE[41]和SPRINTZ[20]。

RLBE编码[41]提出将增量、游程长度和基于斐波那契的编码思想相结合。它有五个步骤：差分编码、二进制编码、游程长度计算、斐波那契编码[42]和级联。具体地，德尔塔编码首先应用于原始数据（32位的整数），并且计算每个差值的长度（以二进制表示法）。然后将行程长度应用于长度代码。在级联阶段，前5位表示二进制字的长度（长度以二进制字编码），然后是长度码的重复时间的斐波那契码字，然后依次是具有相同长度的差分值的二进制码字。

如表4所示，当微分值为正值且较小时，RLBE表现良好。当微分值为负时，RLBE的性能很差，因为符号位为“1”，并且不能取消前导的“0”。当相邻的微分值具有不同的数量级时，即方差较大时，RLBE也表现不佳，因为不能用游程对长度编码。

SPRINTZ编码[20]将编码分为四个步骤:预测、位打包、游程编码和熵编码。在第一步中，它使用一些预测函数(增量编码或快速整数回归编码)来估计下一个值。然后对实际值与预测值的差值进行编码。通常，这一步会缩小要编码的绝对值，其实这也是这一步的目的所在。接下来，它对在第一步中获得的残差块进行位打包。块中有效位的最大数目被写入头中，前导零被修剪。然后，使用游程编码和熵编码(如霍夫曼编码)来减少冗余，也就是在打包的基础上又进行了一次压缩。实际上，这两个过程可以根据实际遇到的数据自行决定，右例的数据值很小且方差小，所以不需要经过预测和打包来缩减长度。运行长度编码通过记录零的数量来压缩连续零块，熵编码通过以霍夫曼编码的形式编码字节来压缩头和有效负载。

如表4所示，SPRINTZ算法适用于可预测的时间序列。对于函数，大量重复或线性增长的时间序列是最好的目标。对于FIRE(快速整数回归)预测器，常数斜率是最佳拟合。

5.1 DICTIONARY编码:DICTIONARY算法[44]在字典中查找值。如果成功找到该值，则用字典中的键替换该值;否则，算法将在字典中添加一对新的键和值。例如，如果字典中的映射为{1:True, 2:False}，则时间序列TS = {True, False, True, True}可以编码为1211。**显然，较大的域导致DICTIONARY编码的成本更高。相反，DICTIONARY通过将其编码为短键来支持大长度值。**

5.2运行长度编码运行长度编码(RLE)[27]特别用于包含重复字符的字符串(字符串的长度称为运行)。该算法的主要思想是将重复字符编码为重复字符和字符长度的一对。例如，长度为16字节的值' abbaaaaabaabbbaa '表示为' 1a2b5a1b2a3b2a '。**但是，如果值中没有重复字符，则输出数据的大小可能是输入数据大小的两倍。**

5.3 HUFFMAN编码HUFFMAN编码算法[29,36]通过为更频繁出现的字符分配更短的码字来减少数据的总长度，采用变长编码取代固定长度编码(如ASCII)的策略。它创建一个唯一可破译的前缀代码，排除了创建分隔符来确定码字边界的需要。**对于偏态数据分布中高频值较多、重复字符较多的数据，HUFFMAN缩短高频字符编码的能力得以发挥。**

对于每个数据点，我们采用μd和σd的正态分布来确定其相对于前一个值的增量（随机选择一个增量），即类似于表1中delta mean (DS)的均值和delta Var(DS)的方差。由于点已经由这个delta和它之前的值确定，我们无法进一步控制值的方差或扩散。然而，它们在一定程度上与讨论过的三角洲有关。重复率是产生一系列具有重复值的连续点的概率，类似于2.3节中定义的重复计数count (RS)。对于第2.4节中的增加计数特征count (IS)，增加率8表示生成值大于前一个点的概率。设DS表示在2.2节中介绍的要生成的delta级数。第3-6行以参数中指定的概率生成重复。同样，第8-12行生成一个概率为γ的增加点。delta由参数为μd和σd的正态分布给出。最后，通过前缀求和将delta级数DS变换为TS，并将所有值缩放到目标值均值μv。

文本数据生成器考虑表8中的4个参数，对应于第3节表2中列出的4个特征。指数θv决定了值分布的偏度。 NV是文本值的域大小。这两个参数决定了值的分布。另外，lc表示所有文本值的平均长度，重复率rc是生成连续字符重复的概率。

算法2给出了文本数据生成器的伪代码。设TS为时间序列，TD为生成TS的值域。

在箱线图中，每个元素代表一个时间序列。我们研究了4种不同数据类型的7种编码方案和4种压缩方案的28种组合。在图9中，指标越低，性能就越好。由于不同运行的压缩比相同，我们不重复压缩比的实验。时间成本实验重复50次。无论是否进行压缩，TS\_2DIFF编码都能实现良好(低)的压缩比。在处理INT32和FLOAT时，RAKE编码的性能甚至比PLAIN(不编码)更差。当有许多连续的0位时，执行RAKE。1位越多，RAKE算法的性能就越差。对于INT32和FLOAT，由于相同值的0位比INT64和DOUBLE少，因此RAKE的性能更差。特别是，对于绝对值较小的负数，由于前导符号位' 1 '和更多的前导' 1 '，考虑到设置位等的额外成本，RAKE的性能可能会更差。GORILLA在INT32和INT64上比FLOAT和DOUBLE表现得更好，因为前导和后置0的位置更相似。

图13将时间成本分解为不同编码算法的编码时间(ET)、解码时间(DT)、压缩时间(CT)、解压缩时间(UT)和相应的压缩比(CR)，以及四种压缩策略。实验在表7中的所有真实数据集上运行，并报告平均值。对于每个维度，我们将结果归一化为0到1之间的范围，越大越好。对于ET、DT、CT和UT，与其他编码算法相比，值越大代表时间越短，效率越高。对于压缩比(CR)，较大的度量表示较低的压缩比，同样表示较好的压缩性能。

如图所示，大多数编码算法在编码(ET)方面都是高效的。TS\_2DIFF具有较好的压缩比(CR)、压缩时间(CT)和解压缩时间(UT)，但相应的解码时间(DT)较差。具有较好的编码和解码时间(ET和DT)的大猩猩具有较差的压缩比(CR)。在没有压缩(NONE)的图13(d)中也观察到类似的权衡结果。

图14-18在INT32上报告结果。图14(a)改变了表6中引入的值均值μv的尺度特征。当值均值为正时，RAKE和RLE表现较好。对于负值平均值，符号位为1，即不能对的前4位进行压缩

图14(a)改变了表6中引入的值均值Ⅱ푣的尺度特征。当值均值为正时，RAKE和RLE表现较好。对于负值平均值，符号位为1，即不能对前导零的前4位进行压缩。TS\_2DIFF和RLBE受平均值的影响较小，因为运行长度和差分编码存储重复次数的长度和差分值，而不是存储大量的大值。GORILLA是不稳定的，因为它的性能受异或影响，与值均值无关。

对于delta特征，图15(a)改变了delta平均值，图16(a)考虑了delta方差。TS\_2DIFF性能随着图16(a)中delta方差的增加而下降，参考表4中的分析，这并不奇怪。

如图17(a)所示，当重复速率增加时，RLE、RLBE和SPRINTZ的性能更好。它们是基于运行长度的算法，倾向于高重复。随着重复率的增加，GORILLA的性能也会更好，因为异或有更多的零，只需要存储一点“0”。RAKE的性能不受影响，因为它的运行长度是以位为单位的，而不是重复的值。

图18(a)增加速率变化/ 8。当增长率增大时，出现更多的正值，有利于位包。

因此，RLBE与增长率呈正相关。

虽然压缩比受图14(a)-18(a)中各种数据特征的影响较大，但对应的插入时间和选择时间是稳定的。不同的编码方法确实会导致非常接近的插入和选择时间。与图10和图11类似，在各种数据特征下，插入时间远远高于选择时间。由于选择时间极低，所以测试中选择时间的方差比较大。

指数越大，即数据分布越偏，HUFFMAN的性能越好。压缩比的改善并不显著。其他算法不受指数的影响，改变了表8中引入的域大小:푣。随着域大小的增加，DICTIONARY的性能会变差，这并不奇怪。相反，DICTIONARY倾向于更大的值长度，

字符重复率푐较大时，RLE的压缩比明显提高，如表5和5.2节所示。然而，如图19(a)所示，这种字符重复在实践中可能并不普遍。

由于字符越来越多，插入时间会随着长度的增加而显著增加。插入时间随指数、定义域和重复变化几乎不变。

由于HUFFMAN算法在选择数据的过程中需要恢复HUFFMAN树，因此其选择时间明显较高。

如图23(c)所示，当repeat变大时，Huffman树变小，选择时间减少。相反，随着长度的增加，Huffman树变大，选择时间在图22(c)中增加。