软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2024,35(3):1173−1193 [doi: 10.13328/j.cnki.jos.007077]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

Apache IoTDB 中的多模态数据编码压缩*

贺文迪[1], 夏天睿[1], 宋韶旭[1,2,3], 黄向东[1,2,3], 王建民[1,2,3]

[1](清华大学 软件学院, 北京 100084)

[2](大数据系统软件国家工程研究中心(清华大学), 北京 100084)

[3](北京信息科学与技术国家研究中心(清华大学), 北京 100084)

通信作者: 宋韶旭, E-mail: sxsong@tsinghua.edu.cn

摘 要: 时间序列数据在工业制造、气象、船舶、电力、车辆、金融等领域都有着广泛的应用, 促进了时间序列数据库管理系统的蓬勃发展. 面对愈加庞大的数据规模和多样的数据模态, 高效的数据存储和管理方式十分关键, 而数据的编码压缩愈发成为一个具有重要意义和价值的问题. 现有的编码方法和相关系统未能充分考虑不同模态的数据特点, 或者未把一些时序数据的处理方法应用于数据编码问题中. 全面阐述了 Apache IoTDB 时序数据库系统中的多模态数据编码压缩方法及其系统实现, 特别是面向工业物联网等应用场景. 该编码方法较为全面地考虑包括时间戳数据、数值数据、布尔值数据、频域数据、文本数据等多个不同模态的数据, 充分挖掘和利用各自模态数据的特点, 特别是包括时间戳模态中时间戳序列间隔近似的特点等, 进行有针对性的编码方案设计. 同时, 将实际应用场景中可能出现的数据质量问题因素纳入编码算法的考量中. 在多个数据集上的编码算法层面和系统层面的实验评估和分析, 验证了该编码压缩方法及其系统实现的效果.

关键词: 数据编码; 时间序列数据; 数据库; 工业物联网; 多模态

中图法分类号: TP311

中文引用格式: 贺文迪, 夏天睿, 宋韶旭, 黄向东, 王建民. Apache IoTDB 中的多模态数据编码压缩. 软件学报, 2024, 35(3): 1173−1193. http://www.jos.org.cn/1000-9825/7077.htm

英文引用格式: He WD, Xia TR, Song SX, Huang XD, Wang JM. Multimodal Data Encoding and Compression in Apache IoTDB. Ruan Jian Xue Bao/Journal of Software, 2024, 35(3): 1173−1193 (in Chinese). http://www.jos.org.cn/1000-9825/7077.htm

Multimodal Data Encoding and Compression in Apache IoTDB

HE Wen-Di[1], XIA Tian-Rui[1], SONG Shao-Xu[1,2,3], HUANG Xiang-Dong[1,2,3], WANG Jian-Min[1,2,3]

[1](School of Software, Tsinghua University, Beijing 100084, China)

[2](National Engineering Research Center for Big Data Software (Tsinghua University), Beijing 100084, China)

[3](Beijing National Research Center for Information Science and Technology (Tsinghua University), Beijing 100084, China)

Abstract: Time-series data are widely used in industrial manufacturing, meteorology, ships, electric power, vehicles, finance, and other fields, which promotes the booming development of time-series database management systems. Faced with larger data scales and more diverse data modalities, efficiently storing and managing the data is very critical, and data encoding and compression become more and more important and are worth studying. Existing data encoding methods and systems fail to consider the characteristics of data in different modalities thoroughly, and some methods of time-series data analysis have not been applied to the scenario of data encoding. This study comprehensively introduces the multimodal data encoding methods and their system implementation in the Apache IoTDB time-series

* 基金项目: 国家重点研发计划(2021YFB3300500); 国家自然科学基金(62232005, 62021002, 62072265, 92267203); 北京信息科学与技术国家研究中心青年创新基金(BNR2022RC01011)

本文由“面向多模态数据的新型数据库技术”专题特约编辑彭智勇教授、高云君教授、李国良教授、许建秋教授推荐.

收稿时间: 2023-07-17; 修改时间: 2023-09-05; 采用时间: 2023-10-24; jos 在线出版时间: 2023-11-08

CNKI 网络首发时间: 2023-12-25

database system, especially for the industrial Internet of Things application scenarios. In the proposed encoding methods, data are comprehensively considered in multiple modals including timestamp data, numerical data, Boolean data, frequency domain data, text data, etc., and the characteristics of the corresponding modal of data fully are explored and utilized, especially the characteristics of timestamp intervals approximation in timestamp modality, to carry out targeted data encoding design. At the same time, the data quality issue that may occur in practical applications has been taken into consideration in the coding algorithm. Experimental evaluation and analysis on the encoding algorithm level and the system level over multiple datasets validate the effectiveness of the proposed encoding method and its system implementation

Key words: data encoding; time-series data; database; industrial Internet of Things; multimodal

时间序列数据是某些物理量在不同时间点上的采集值所组成的数据序列. 在当今这个大数据的时代, 人们在各个应用场景中所面对的数据的规模也越来越庞大. 特别是在工业物联网领域[1], 从车辆到船舶, 从风电到核电, 工业传感器每时每刻都在产生着海量的数据. 同时, 数据类型和数据格式也越来越多样化, 例如结构化关系表、半结构化 XML、非结构化文本、图数据、流数据和时序数据等不同的形式. 越来越多的应用场景要求数据库系统能够同时高效管理多种不同类型的数据, 因此, 多模态数据的管理与分析成为一个非常具有现实意义的问题.

伴随着数据模态的日益多样和数据量的不断增加, 数据库系统在数据存储方面面临着更多的挑战[2]. 于是, 高效的编码压缩算法就变得非常关键. 如果不能通过一定的方法很好地存储这些数据, 将会造成大量的空间占用和消耗. 而从另一方面来讲, 编码压缩算法和系统实现上的细小的突破或提升, 都将有助于空间存储开销的降低, 带来巨大的价值和意义. 因此, 充分挖掘不同数据的规律和特点, 并能利用这些特点对数据进行编码压缩, 是数据能够高效存储的关键.

针对时序数据的巨大功能需求和广阔应用前景, 以 Apache IoTDB[3]为代表的时序数据库, 为管理时序数据提供了有力的支撑. Apache IoTDB 时序数据库是一个集成了数据收集、存储、管理与分析功能的物联网时序数据软件系统, 可以满足工业物联网领域的海量存储、高速读取和复杂分析需求[4]. 特别是在如气象、船舶、电力、车辆等应用场景中都有着非常广泛的应用, 并且取得了十分不错的效果.

现有的一些数据库系统在多模态数据的编码方法存在如下问题.

- 针对时间戳模态, 通用的数据编码方法通常把时间戳数据当作一般的数值数据来编码和存储, 未能充分利用时间戳序列的特点, 并且一些针对时间戳序列的处理方法没有能实际运用到编码算法中, 去借助这些方法来有针对性地解决时间戳序列编码问题; 此外, 因为实际应用场景中时间戳间隔并不是严格相等的, 而是存在很多波动, 这种波动既导致现有的编码效果不理想, 也对设计时间戳编码算法带来了一定的挑战.
- 对于小数模态和频域数据模态, 现有的一些编码方法保留了过高但是在实际中可能并不必要的精度, 造成了空间上不必要的消耗.
- 对于文本模态, 并非所有的信息都需要按照一般的普通字符进行存储, 特别是日志文件中具有一定规律的时间戳和数值信息等.

同时, 现有的编码方法未能充分考虑时序数据中可能存在的数据质量问题, 来有针对性地设计包括时间戳编码在内的数据编码方法. 特别是在工业应用场景中[5], 传感器采集和传输的时间序列数据, 往往可能存在包括数据缺失、数据重复、数据延迟在内的数据质量问题, 这些数据质量问题对数据的编码压缩带来了很多问题和挑战, 现有的编码方法通常未能充分将数据质量问题纳入编码设计的考量中, 导致数据的存储效率下降. 于是, 如何更好地在编码设计中考虑和处理这些数据质量问题, 具有重要的现实意义.

因此, 现有数据库系统中的编码压缩存在的主要挑战可以被总结为如下几个方面.

(1) 如上文概括的几点, 不同模态的数据特点未被充分挖掘, 或者未将这些特点充分应用于数据编码算法中, 特别是像时间戳模态数据, 现有的系统缺乏对时间戳序列更有针对性的编码算法, 一些时间序列数据的处理方法没有被运用到编码问题中.

(2) 实际应用场景中, 时间序列数据存在数据缺失、数据重复、数据延迟等数据质量问题, 这些数据质

量问题严重影响数据编码的效果.

(3) 未能把面向不同模态数据的有针对性的编码压缩算法充分地纳入数据库系统的设计和实现中.

针对上述问题, 我们希望全面地阐述 Apache IoTDB 时序数据库系统的编码方法. 在我们的编码设计中, 主要包括如下编码技术点: 在时间戳模态中, 利用时间戳间隔的特点, 实现对时间戳序列的编码; 在数值模态中, 基于小数的数值精确度特点, 实现针对小数的编码算法, 并利用某些整数数据波动较小、取值域有限、重复数值多等特点, 实现针对整数的编码算法; 在布尔值模态中, 利用布尔值有效信息的特点, 实现针对布尔值的编码; 在频域模态中, 利用频域数据的时频变换和幅值特点, 实现针对频域数据的编码; 在文本模态中, 提取和保存日志中含有的数值信息, 实现针对日志型文本数据的编码; 同时, 在编码算法中, 将数据缺失、数据重复、数据延迟等数据质量问题纳入编码算法的设计考量中. 本文的主要贡献有: (1) 充分挖掘不同模态数据的特点, 将这些特点和相关的方法应用于针对此类模态数据的编码算法设计中, 特别是比如时间戳模态中时间戳序列具有的特点等, 将相关的时间序列处理方法实际应用于时间戳的编码算法之中; (2) 在编码算法的设计过程中, 将数据质量问题的因素纳入编码算法的考量中; (3) 我们的编码方法及其系统实现可以有针对性地考虑不同模态的数据, 将面向多种模态数据的编码压缩算法充分地纳入 Apache IoTDB 数据库系统的设计和实现中.

本文第 1 节介绍在数据库领域其他系统的数据编码压缩相关工作. 第 2 节主要介绍部分基本概念. 第 3 节详细阐释 Apache IoTDB 中的多模态数据编码压缩方案. 第 4 节阐述 Apache IoTDB 的系统实现. 第 5 节通过实验验证方法的效果. 最后, 第 6 节总结全文.

1 相关工作

数据的编码压缩是数据库系统存储的一个重要问题, 围绕此问题也产生了很多相关工作. 在国内外的数据库系统和相关研究中, 有很多不同的编码压缩算法和系统实现方式. 其中: 有些编码算法是有损的, 会在数值的准确性上造成一些损失; 有些编码压缩方法是无损的, 可以完整地保留全部的原始数值信息. 接下来, 我们将大致概括部分在领域中广泛使用的其他编码压缩方法和数据库系统.

在编码压缩算法层面, 存在很多其他的相关编码算法. 比如, SPRINTZ 编码[6]是一个可以用作整数编码的算法, 主要包括预测、位压缩、游程编码、熵编码这 4 个步骤: 首先, 通过使用一些预测函数来估计接下来的数值, 然后对实际值和预测值的差异进行编码, 通常情况下, 这可以起到缩小待编码数值的绝对值的效果; 接下来, 对第 1 步中得到的残差块进行位压缩, 并基于数据块中最大数值的有效位作为编码位宽, 将数据写入系统中; 此外, 运用游程编码和熵编码来减少冗余, 其中, 游程编码是通过记录连续重复数值的次数来压缩连续的数值, 而熵编码则是通过对哈夫曼编码形式的字节进行编码来达到压缩的目的. 同时, 还有 RAKE 编码[7]、哈夫曼编码[8]、Simple8B 编码[9]和其他不同的编码方法[10].

此外, 还有一些其他混合型编码算法, 这些编码算法由多个混合的方法来组成一个综合的编码算法[11]. 例如, 整数形式的 RLBE 编码[12]结合了差分编码、游程编码、斐波那契编码的思路. 它的计算步骤可分为差分编码、二进制编码、游程编码、斐波那契编码. 具体来说, 首先对原始数据应用差分编码, 并计算每个差分值的长度; 然后, 进一步运用游程编码, 前几位负责存储二进制字的长度, 后几位是长度码的重复次数的斐波那契编码码字, 接下来的若干位负责表示具有相同长度的差值的二进制位.

与此同时, 也产生了越来越多的通过使用混合技术或机器学习技术[13]来解决时序数据编码压缩问题的算法. 比如: 基于机器学习的两阶段压缩算法[14]通过引入一个两阶段模型, 为每个单独的点选择压缩方案, 这有助于解决时间序列中数据的多样性问题. 该算法由一个两阶段模型压缩的框架构成, 包含了多个主要模式来对典型的模式进行分类, 并通过定义一些参数来帮助构建子模型压缩方案. 这种方法通过一个具有强化学习功能的神经网络结构, 来自动调整参数的取值.

在数据库系统层面, 除了 Apache IoTDB 时序数据库之外, 也有许多其他的时序数据库系统. 总的来说, 这些时序数据库区别于一般的关系型数据库, 主要面向物联网或其他时序数据应用场景, 针对时序数据的数

据属性、规模海量、写入负载高等特点进行系统实现, 这些系统在存储方案的设计上既有一定的相似之处, 也有各自不同的具体设计和实现.

InfluxDB 时序数据库[15]的逻辑存储架构包括数据库、物理量、标签和字段, 这些组件共同构成了数据的组织和存储方式. 其中: 数据库是指在逻辑层面上, 对数据进行分组和分类; 物理量表示一组数据点, 包含多个标签和字段; 标签是用于标识和过滤数据的键值对, 用于描述物理量的元数据; 字段是实际存储的数据值, 以键值对的形式记录; 数据保留时间指定了数据会在数据库中留存的时间范围. 存储引擎方面, InfluxDB 使用 TSM 树结构作为存储引擎, 通过将数据分为多个时间段, 并在每个时间段内创建索引和压缩数据来提供高效的存储和查询性能. 数据被存储在磁盘上的文件中, 而索引则被存储在内存中. 索引结构方面, InfluxDB 使用 B+树索引结构来支持快速的数据查询, 这是一种常见的平衡树结构, 索引按照时间序列组织, 根据时间戳和标签来进行索引.

DolphinDB 时序数据库[16]是一种分布式时序数据库, 内置流式数据处理引擎以及并行和分布式计算的功能, 并提供分布式文件系统, 支持集群扩展, 是使用 C++编写的. 总体上采用类 HDFS 分布式文件系统, 由名称节点统一管理元数据, 并自动管理分区数据. 在金融、物联网等多种不同应用领域中, 在数据分析建模、实时流数据处理、海量传感器数据处理与分析等任务场景中具有一定优势. DolphinDB 的架构拥有多个数据节点, 在数据库层面不存在领导节点, 每个数据节点拥有本地的存储设备, 相互之间通过 DFS 交互, 从而可全局优化, 实现共享存储, 让数据均匀地分布在各节点上, 充分地利用集群资源.

TDEngine 时序数据库[17]在数据存储时, 结合物联网的应用场景, 设计了独有的存储和查询引擎以及存储结构. TDengine 存储的数据包括采集的时序数据、库或表相关的元数据、标签数据 3 个部分: 时序数据存放于 V 节点里, 由数据、数据头和数据尾组成, 采用一个采集点一张表的模型, 连续存储一个时间段内的数据, 可以通过简单的追加操作, 对单张表进行写入; 标签数据存放于 V 节点里的元文件, 支持增删改查操作; 元数据存放于 M 节点里, 包含系统节点、用户、数据库、数据表格式等信息, 支持增删改查操作. TDengine 通过列式存储和专有的压缩算法实现数据压缩,包括不压缩、一阶段压缩、二阶段压缩 3 种形式: 一阶段压缩使用专有算法压缩; 二阶段压缩是在专有算法压缩的基础上, 再使用通用算法额外压缩一次.

各种不同的编码压缩算法和数据库系统分别具有各自的特点和优势, 也有一定的不足和适用性问题. 因此, 更重要的是如何针对数据的特点和不同的数据模态, 运用合适的数据编码压缩算法构建合适的系统数据存储方案. 通过总结其中有益的经验, 分析现有编码方法的主要问题, 有助于对编码压缩算法的进一步研究进和比较, 进而帮助我们对多模态数据设计合适的编码算法.

2 基本概念

2.1 IoTDB的数据存储形式

在 Apache IoTDB 系统中, 数据通过 TsFile 文件进行存储. 这种文件格式在结构上的设计, 使其能够有效地适应于时序数据的查询和存储等方面的功能需求[18]. 如图 1 所示, 图的左半部分展示了数据在 Apache IoTDB 系统中所具有的树状结构的关系. 可以看到: 从顶部的根目录开始, 自上而下依次为根目录、若干个实体设备层、物理量层, 从根节点到叶子节点相连命名得到的路径, 如 root.filed1.wf1.device1.status, 即表示一个时间序列, 可能描述的实际含义是某个区域内的某个系统的某个设备的状态值信息, 不同区域、不同系统、不同设备的不同物理量的信息共同组成了一个庞大的树状结构. 当然, 这里的具体含义是由用户进行设置的. 图的右半部分展示了 TsFile 的主要结构, 呈现了数据块和数据页面之间的关系, 也反映了数据存储在 Apache IoTDB 系统中的最基本的存储形式.

在 TsFile 中, 一个 TsFile 会包含多个数据块存储组. 数据块是由一个元信息和若干数据页面组成. 与数据页面不同, 数据块的大小是可变的. 元信息包括该组数据的数据类型、编码压缩方式等基本信息, 使得系统可以根据元信息中的内容, 使用相应的编码压缩算法来压缩和解压缩不同的时间序列. 在一个数据块中, 会包含多个数据页面. 数据页面是在磁盘上存储时间序列数据的基本单位. 一个数据页面中的每个时间序列都是

按时间升序排序的, 一共包括两列: 一列是时间戳列, 另一列是数值列. 并且时间戳和数值是分开存储和编码压缩的, 因为这样的列式存储方式非常有助于压缩时间序列数据.

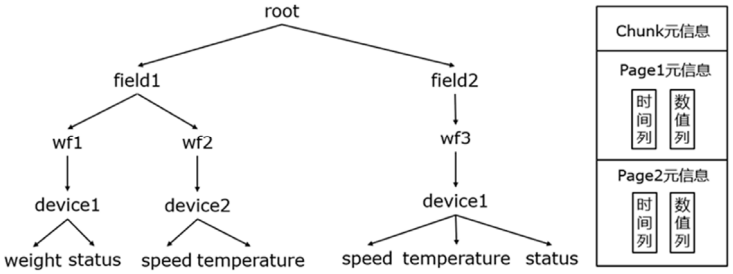


图 1 Apache IoTDB 系统数据模型结构图

2.2 数据编码

数据编码问题是数据存储中的一个重要问题, 是在保证数据完整情况下, 减小数据存储占用的有利方法. 在数据写入的过程中, 对数据进行编码, 从而减少磁盘空间的占用量, 提高数据的存储效率. 即按照一定的处理方法和转换形式将数据转换为字节流, 而转换后的字节流占用的空间通常比数据原本占用的空间小, 以此来达到压缩空间的效果.

与编码相对应的过程是解码. 编码算法可以将输入的原始数据按照某种算法进行编码, 在编码的过程中, 对数据按照一定的方法进行处理和转化, 从而达到节省数据存储空间的作用; 而对应的解码算法可以将经过压缩后的字节流, 经过对应的还原方法来解码出原始的数据. 而在某些情况下, 还原得到的数据可能与原始数据不完全相等, 这类编码算法属于有损编码压缩方法; 而对于能够确保还原得到的数据和原始数据相等的, 属于无损编码压缩方法.

在 Apache IoTDB 系统的编码压缩中, 由于 TsFile 文件结构使得时间戳列与数值列是分开单独存储的, 因此, 我们可以对时间戳列和数值列采用不一样的编码压缩方法. 其中: 对于时间戳列, 考虑到物联网场景下传感器周期性采集下的数据的特点, 可以使用基于差分或者针对时间戳的编码方法; 对于数值列, 则可以根据实际数据的特征使用合适的编码方法. 此外, 在 TsFile 中, 可以只在数据页面层面上编码和解码数据, 这样就不需要为了读取一部分数据而解码整个文件, 从而有利于编码和解码的时间性能.

2.3 数据质量问题

在很多应用场景中, 时间序列数据源源不断地在被收集、存储和分析, 以便于对设备和系统的状态进行监控和管理. 但是在实际应用中, 特别是在一些工业应用场景中, 采集得到的时间序列数据通常在不同程度上具有一定程度的数据质量问题. 因为在时序数据从被传感器采集到被存储至时序数据库的过程中, 可能会出现传感器故障、传输丢包、网络延迟等若干问题. 这些数据质量问题除了会影响对数据的分析以外, 也会影响到数据编码以及数据存储的性能.

总的来看, 数据质量问题可以主要归纳为数据缺失、数据重复、数据延迟这 3 种情况.

(1) 数据缺失: 出现一个或多个数据点丢失, 这可能是由于某个或者某段数据未能成功采集或者成功传输等原因导致的.

(2) 数据重复: 在一个数据点周围, 出现一个或多个非常接近的点, 这可能是比如传输中误将同一个数据传输了多次等因素导致的.

(3) 数据延迟: 一个数据点, 比原本该点预期的时间有所延迟, 这可能是比如网络延迟等因素导致的.

如图 2 的例子所示, 这是数据质量问题的示意图, 3 个子图分别表示数据延迟、数据缺失、数据重复的数据质量问题, 图中箭头所指的位置是出现数据质量问题的地方.

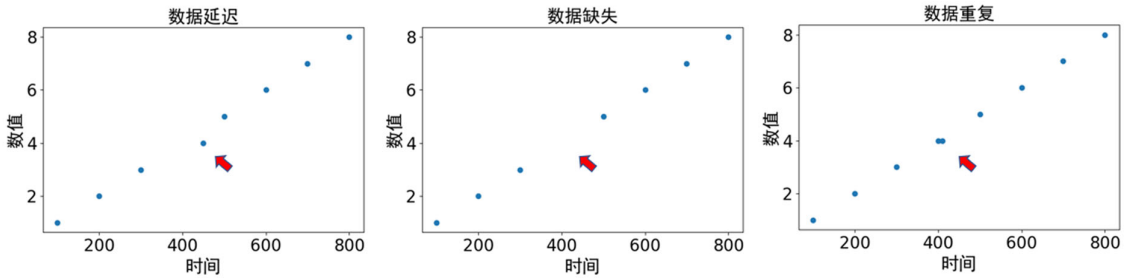


图 2　数据质量问题示意图

3　Apache IoTDB 中的多模态数据编码压缩实现方案

时间序列数据反映了观测对象在不同时间点采集得到的一系列测量值, 随着工业物联网的蓬勃发展和广泛应用, 各种设备的海量数据被实时采集并通过网络共享, 被用于监测运行状态、突发事件响应、预测变化规律、预测发展趋势等多样的任务需求中. 这些实际的应用场景, 也正是 Apache IoTDB 时序数据库被广泛应用和发挥其优势的地方.

面对海量的数据以及多样的数据模态, 数据的存储面临着更大的挑战. 在 Apache IoTDB 中, 多模态数据主要包括时间戳模态、数值模态、布尔值模态、文本模态、频域模态等. 不同的编码方式, 擅长处理不同模态的数据. 在 Apache IoTDB 时序数据库中, 有若干种不同的编码压缩算法, 包括 RLE 编码、TS2DIFF 编码、GORILLA 编码、字典编码、频域编码等. 针对不同模态的数据, 有不同的默认编码方式, 同时也支持用户根据自己的需要进行自由的选择. 接下来, 我们将分别详细介绍各个模态的数据编码压缩方案.

3.1　时间戳数据模态

时间序列数据由一组时间戳序列和一组数值序列构成. 当进行数据编码压缩时, 在每个时间戳序列和数值序列中, 分别进行编码压缩. 如图 3 的示意图所示, 针对时序数据中的时间戳数据列进行时间戳编码, 最终得到占用空间更小的时间戳数据编码字节流.

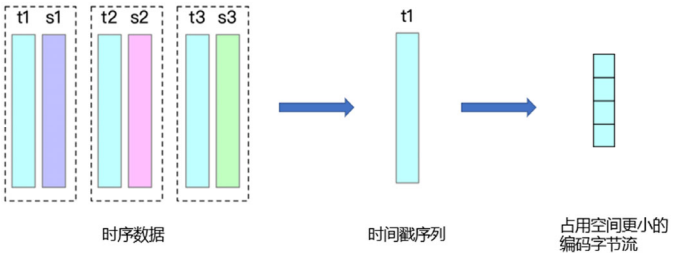


图 3　时间戳编码示意图

现有的一些时间戳编码方法, 比如基于差分的编码方法, 虽然也利用了时间戳序列通常是递增的特点, 但是这种差分的方法一方面对时间戳序列特点利用的不够充分, 另一方面没有考虑在实际应用场景中时序数据可能存在的数据质量问题. 当数据中存在这些数据质量问题时, 差分的方法可能会造成很多极端的结果, 导致最终的空间性能不佳. 如果能针对其中的时间戳序列去设计一种更有针对性的时间戳序列编码算法, 就可以有效地降低时间戳序列占用的空间, 进而达到提高整个时间序列数据的空间性能的目的.

首先, 通过对时间戳数据的分析可以发现, 时间戳序列通常具有一定的特点和规律. 特别是一些时间戳序列的间隔, 因为在工业物联网应用场景中, 数据是通过固定的采集频率获得的, 因此往往呈现出间隔近似的特点. 一般而言, 严格等间隔的时间戳是指一组时间戳序列前后相邻两个时间戳的差值严格等于一个固定值. 但是从应用中来看, 这类情况是一种比较理想的情况, 对数据质量有比较高的要求. 实际上, 在工业物联

网领域等实际应用场景中来看, 时间戳序列从整体的规律上来看间隔往往是近似的, 但是会有一些误差和波动, 并不是严格等于一个固定的值. 同时, 在数据的某些地方, 通常还伴随着数据缺失、数据重复、数据延迟等数据异常情况. 于是, 我们可以充分利用时间戳序列的特点来设计一种更有针对性的编码方法, 以优化其存储时的空间性能.

观察如图 4 所示的时间戳序列经过一阶差分后的数值频次直方分布示意图, 可以看到: 该例子中的数据有一个明显的峰, 绝大多数数值都位于峰的左右, 说明数据的间隔基本固定, 但是并不是严格相等; 同时, 在峰的两侧有一些频次比较低的数值, 表示数据并非严格等间隔, 而是会存在一些上下的波动; 并且还有一些数据相隔的较远, 表示数据存在一定程度的数据质量问题. 我们所研究的近似间隔时间戳序列正是这类整体上具有时间戳间隔近似、但并不严格相等的特点, 并且在某些地方存在一些数据质量问题的时间戳序列.

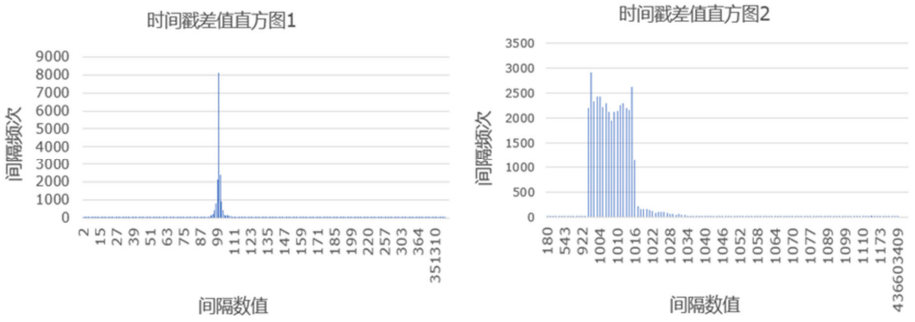


图 4 时间戳序列差值频次直方分布示意图

在方晨光等研究者在 VLDB22 的时间戳修复工作[19]中, 根据时间戳间隔的特性, 提出了一种时间戳数据修复的算法. 这种算法针对时间戳数据异常情况, 通过动态规划的算法, 进行时间戳的匹配和修复. 受到这种算法的启发, 我们希望将这种时间戳修复算法中的思路, 应用到时间戳的编码问题中. 这需要我们将数据修复问题转换为数据编码问题, 将数据修复和数据处理的方法应用于编码方法. 同时, 结合编码中遇到的问题, 比如考虑到编码的时间复杂度, 需要采用尽量简便的判断和设计方法, 以保证编码的时间性能. 接下来, 我们将详细阐述利用时间戳特点设计本文所提出的针对时间戳数据的编码方法.

时间戳编码算法具体包括如下步骤: 首先对数据进行分块, 即通过指定一个固定的数据块大小, 顺序地将全体数据划分成大小固定的数据块; 然后, 以每个数据块为单位, 分别在各个数据块上进行时间戳编码. 经过编码后, 每个数据块都可以得到一个二进制字节流, 这时再把这些字节流顺序地连接起来, 即可得到最终经过时间戳编码后的二进制流. 解码时, 整个字节流会再一一对应的分成多段, 每段对应一个数据块. 通过元信息和解码算法, 即可还原每个数据块上的原始数据, 然后顺序地将其相连在一起, 即可解码出原始数据.

数据块大小的选取需要面临一个平衡: 如果数据块的大小过小, 那么就会导致数据块的数量过多, 随之产生和需要保存的元信息也就更多, 这会削弱编码的空间性能; 但是如果数据块的大小过大, 那么就会导致一个数据块里包含的数据点过多, 有时可能数据组织在一起的规律性就会减弱, 甚至单个的异常就会拖累整个数据块的性能, 同样也会导致编码的效果不佳. 所以, 数据块大小过大或过小都可能损害空间性能. 而在具体实现中, 考虑到编码对时间性能的要求, 找到最优的参数将花费大量额外的时间开销. 因此在实际应用中, Apache IotDB 系统将数据块大小的取值设置为一个默认的参数, 同时也支持用户直接人为地指定一个合适的参数, 而如果用户未直接指定, 那么系统将使用原有的默认参数.

接下来, 确定时间戳间隔值. 对于一组具有间隔近似特点的时间戳序列, 首先就需要确定出该组数据的间隔值. 找寻一个合适的时间戳间隔值, 也是接下来时间戳分解的奠基性的一步, 更影响着后续编码的效果. 为了确定出效果最好的时间戳间隔值, 可以采取如下的一种精确的时间戳间隔值计算方法: 首先列出时间戳间隔的所有可能候选集, 通过计算时间戳前后差值, 然后找到这些差值的最小值和最大值, 那么候选集的范围就是位于最小值与 2 倍最大值之间的所有值. 然后分别进行计算, 最后选取使得编码位宽最小的时间戳间

隔作为最终结果. 这种计算方法将基于差分的编码作为一种特殊的情况, 由此, 把基于差分的编码自然地统一起来. 可以从理论角度证明, 这样的方法将会优于或者不弱于基于差分的编码方法.

为适应编码对时间性能的要求, 可以在这一环节中通过近似算法来确定时间戳间隔值. 估计一组间隔近似的时间戳序列的间隔大小可以采用多种方法, 通俗地来说, 就是尽量缩小搜索的范围, 在尽量不影响最终效果的前提下, 在候选集中筛选出更小的搜索范围. 更直接地, 其中一种最直观的方法就是利用中位数来估计时间戳序列的间隔大小. 由此, 我们提出一种近似算法. 近似算法的具体计算步骤是: 对时间戳序列计算前后相邻两个数值的差值, 然后取差值的中位数, 即可将这个中位数作为时间戳的间隔值. 并且在此之前, 结合差分值频次直方分布图, 增加对间隔分布情况的判断, 在分布的离散情况达到预设阈值时, 就采用固有的编码方法. 因为对于时间戳序列来说, 大多数数据的间隔是近似的, 通常围绕着这个间隔值上下小幅浮动, 于是, 通过计算差值的中位数的方法, 就可以近似地估计出这组时间戳序列的间隔值.

确定间隔值后, 对时间戳进行分解. 具体步骤是, 将一个时间戳分解为间隔值和残差的某种组合的形式. 其实, 时间戳的分解可以有很多种方案, 在这里, 我们给出一种可行的时间戳分解方案, 其具体计算步骤是: 首先, 将时间戳减去上一个时间戳的值后除以间隔值并取整, 得到一组间隔列; 然后计算剩余的残差部分, 作为偏差值, 由此组成偏差列. 对于得到的偏差列和间隔列这两条序列, 可进一步对其进行处理. 由于时间戳数据的特点, 对于多数数据来说, 间隔列的数值为 1 的频率较高. 于是, 为了更好地提升空间性能, 可以对间隔列进行进一步处理, 只保留其中数值不是 1 的间隔值, 然后记录非 1 间隔值的位置和非 1 间隔值的数值.

最后, 通过位压缩的方法, 将序列数据转化为字节流. 针对目前处理后的数据序列, 可在数值上进行进一步编码. 这里采用了位压缩方法的思路[20], 对各个数据块的各部分数据, 按照各自的有效长度进行编码. 因为原始的数值转换成二进制字节流后, 由于正数和字节转换的方法, 会导致前面有很多前置的 0. 对一组数据来说, 假如所有数值对应的二进制字节中的前 m 位均为 0, 那么其实这 m 个比特是没有意义的, 因为它们不表示任何信息. 因此, 我们只需要找到一组数据中最大的数, 计算它所需的位宽, 作为整组数据的编码位宽.于是, 通过位压缩的方式, 空间性能得到了更好的优化. 一般形式的位压缩编码位宽的公式为

$$EncodeWidth(X)=\max_{1\leqslant i\leqslant n} Width(X_i) \tag{1}$$

值得注意的是: 当一组数据中出现负数时, 这个数转换成字节流后的首位将是 1, 这意味着它没有前置 0, 这将导致整组数据都无法通过上述的方式进行优化. 于是, 我们需要对数据进行一定的处理, 将所有数据都转化为非负数. 常用的转化方法有 Zigzag 法、减最小值法等: Zigzag 法是指通过一个函数映射, 将负数映射为正数; 减最小值法是指将一组数据都减去这组数据的最小值, 使数据中不再有负数. 在我们的设计中, 采取了减最小值法作为位压缩编码的预处理方法.

总结来看, 根据位压缩的编码思路, 对各个数据块的数据按照各自的有效长度进行编码. 经过数值编码处理后, 将数据变为非负数, 然后统计数据块中所需编码位宽最大的数值, 记录该数值所需的编码位宽为整个数据块的最大编码位宽. 对该数据块上的所有数据, 以最大编码位宽的位宽值进行编码, 加入结果字节流中. 时间戳编码的大致算法见算法 1.

算法 1. 时间戳编码算法.

输入: 待编码的时间戳数据 *data*.

输出: 编码后的时间戳数据字节流 *buffer*.

```
1:  block=len(data)/size;
2:  for i←1 to block do
3:      tl=data[i×block,(i+1)×block];
4:      gl=gap(tl);
5:      flag=isDiff(gl)
6:      if flag==True then
7:          grid=getgrid(tl);
```

8: *gl*=*calgrid*(*tlt*,*grid*);
9: *dl*=*caldiff*(*tl*,*grid*);
10: **end**;
11: **if** *flag*==True **then**
12: *buffer*.*append*(*bitpacking*(*gl*));
13: *buffer*.*append*(*bitpacking*(*tl*));
14: **else**
15: *buffer*.*append*(*bitpacking*(*dl*));
16: **end**;
17: **return** *buffer*;

接下来, 我们对时间戳编码算法进行一个大致的复杂度分析. 从空间复杂度的角度来看, 虽然时间戳分解产生的间隔列和相应的参数需要占用额外的空间, 但是额外空间是有限的, 其复杂度仍限于 O(*n*); 同时, 在整体上可以带来空间的改善. 并且从实际效果来看, 下文的实验结果表明, 时间戳编码在多个数据集上使空间性能得到了提升. 时间戳编码的各个步骤, 包括确定间隔值、时间戳分解、位压缩等, 时间复杂度都是 O(*n*). 因此从时间复杂度的角度来看, 时间戳编码算法的时间复杂度是 O(*n*). 由于时间戳编码的多个步骤, 综合来看在时间上需要花费更多, 不过由此换取了空间性能的改善.

3.2 数值数据模态

数值模态是时序数据库中最常见的一种模态, 是最基本的一种数据形式. 更进一步地, 数值模态又可分为整数和浮点数两种类型. 因为整数和浮点数这两种数据类型本身在存储时就有所不同, 因此它们所对应的编码压缩方式也稍有差异. Xiao 等研究者在 VLDB22 的时序数据编码工作[21]中, 介绍了不同种类的基本数值编码压缩算法.

3.2.1 整数编码

整数这一数据类型可以非常方便地通过二进制的形式进行转换, 并且一般可以通过 1 个符号位来表示正负. 此外, 整数又可分为整型 INT32 类型和长整型 INT64 类型. 对于这两种数据类型, 在编码方式的实现细节上也会略微有所不同. 比如, 在编码位宽的设计中: 对于 INT32 类型, 基准位宽会以 32 为单位; 而对于长整型 INT64 类型, 基准位宽会以 64 为单位.

对于整数形式的 RLE 游程编码[22], 其具体方法是: 对待编码的整数序列, 统计连续出现的整数的数值和频次; 然后, 对数值列和频次列分别进行编码压缩. 相应地, 在 RLE 编码的解码时, 根据 RLE 编码的解码方式, 将数值列和频次列重新还原回整数列, 解码出原始的数据. 这种游程编码的方式, 比较擅长处理具有连续相同元素数值的数据. RLE 编码的编码位宽的具体计算公式如下.

$$EncodeWidth(X)=\max_{1\leqslant i\leqslant n} Width(RLE\_R_i)+\max_{1\leqslant i\leqslant n} Width(RLE\_V_i) \tag{2}$$

如图 5 的例子所示, 比如一组数据 1, 1, 1, 1, 6, 4, 4, 4, 因为这组数据依次由 4 个 1、1 个 6、3 个 4 组成, 于是, 经过游程编码可以将数据压缩为(4,1), (1,6), (3,4), 减少了存储所需的信息. 可以看到: 如果连续重复的数据越多, 那么游程编码就更能发挥出它的优势; 但是如果数据中连续重复的不多, 那么游程编码的效果就会变差, 甚至反而会使得空间占用变大.

此外, 整数形式的基于差分的编码也是一类常用的编码方式[23], 其具体方法是: 通过计算前后相邻两个时间戳的差值, 然后对差值进行编码, 同时需要记录初始值. 一阶差分的思想比较直观, 存储前后相邻两个时间戳的差值以及起始值即可. 而二阶差分编码是在一阶差分编码的基础上再做一次差分, 然后再去保存差分值和相应的初始值信息.

对于整数形式的 TS2DIFF 差分编码, 首先对整数序列做一阶差分, 并记录初始值, 然后对得到的序列统一再减去最小值, 最后对差分列进行编码压缩, 并保存起始值等信息. 相应地, TS2DIFF 差分编码在解码时,

首先对序列加上最小值, 还原回原始的差分列, 然后结合初始值信息, 将差分列还原回整数列, 解码出原始的数据.

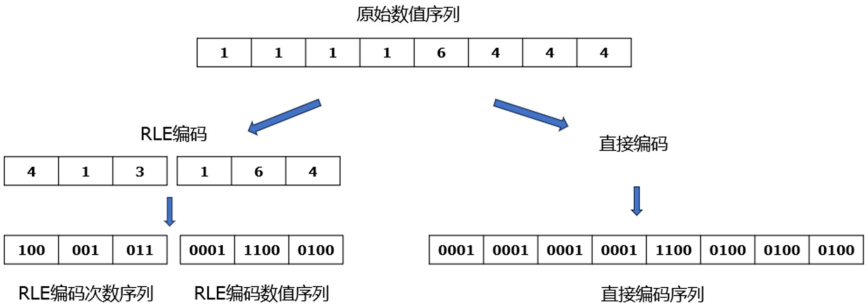


图 5 游程编码示意图

TS2DIFF 编码主要擅于处理数值前后变化范围不大的数据. 如图 6 的例子所示: 如果一组数据是 10, 20, 30, 40, 45, 60, 那么第 1 次前后两两作差得到 10, 10, 10, 5, 15 的差分序列, 其中最小值是 5; 第 2 次减去该最小值 5 后得到 5, 5, 5, 0, 10 的序列, 同时, 编码时需要记录最开始的起始值 10 和最小值 5. 这种基于差分的编码算法比较适合于数据波动不太大的序列; 相反, 如果序列本身波动较大, 那么最终的效果可能反而会不太理想.

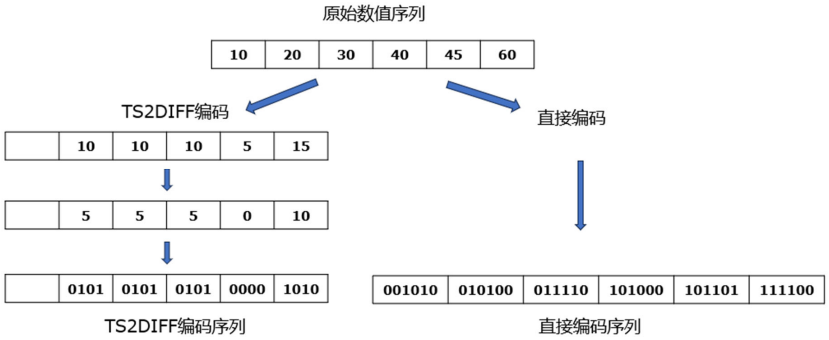


图 6 TS2DIFF 编码示意图

TS2DIFF 编码的编码位宽的具体计算公式如下.

$$Diff_i=X_{i+1}-X_i \tag{3}$$

$$TS2DIFF_i = Diff_i - \max_{1\leqslant i\leqslant n} Diff_i \tag{4}$$

$$EncodeWidth(X) = \max_{1\leqslant i\leqslant n} Width(TS2DIFF_i) \tag{5}$$

整数形式的 Gorrila 编码[24]是针对数值存储格式特征设计的一种编码算法, 异或操作 XOR 是该编码方法的关键. 其具体步骤是: 序列的首值不动, 后面的值是该值与第 1 个值取异或的结果. 如果结果相同, 仅需存储一个 0; 如果结果不同, 存储异或后的结果. 并且利用游程编码的思想, 进一步压缩首尾的 0 值. 不过, XOR 编码受数据波动影响较大, 如果数据波动较大, 那么编码的效果可能就会比较差, 因为这样异或结果的首尾将产生大量的 0. 因此, 这种编码方法比较适合编码前后数值比较接近的数据序列, 不适合编码前后数值波动比较大的数据序列.

3.2.2 浮点数编码

浮点数在进行数据存储时, 通常会根据 IEEE754 标准, 部分区间用于表示指数部分, 部分区间用于表示尾数部分; 同时, 再通过 1 个符号位来表示浮点数的正负. 此外, 浮点数又可分为单精度 Float 类型和双精度 Double 类型. 对于这两种数据类型, 主要区别是用来表示指数部分的区间和表示尾数部分的区间的长度不同.

Double 类型在这两部分的长度都要大于 Float 类型, 这意味着它将具有更高的准确度; 但同时, 也意味着这将占用更多的空间. 针对浮点数类型的数据, 一般也可以使用如前文所介绍的一些编码方法进行编码压缩.

为了进一步提升编码的空间性能, 还可以针对浮点数使用如下的编码方法, 即基于精度的幂变换方法. Liu 等研究者在 VLDB2021 的研究工作中[25], 也曾使用过类似的思路. 由于在很多应用场景中, 往往对浮点数并不需要非常高的数据精度, 也即很多编码位数实际上是有冗余的, 因此, 如果能适当地舍弃一些编码位数, 就可以在保证基本的浮点数精度的同时, 有效地节省浮点数的存储空间. 当然, 这种编码方法是一种有损的编码方法, 它会在舍弃的过程中不可逆地丢失掉一部分数据信息, 因此在实际应用时, 也需要将损失的程度纳入考虑的范围, 使其处于可接受的区间内.

具体来说, 针对浮点数的基于精度的幂变换编码的具体步骤是: 首先, 通过一个手动指定或默认的参数 *base* 和 *n*, 分别作为底数和精度的大小; 然后, 对所有的浮点数乘以以 *base* 为底数、以 *n* 为指数的幂, 将浮点数变换成整数; 最后计算整数所需的最大编码位宽, 对变换后得到的整数序列进行编码, 同时保存幂变换的底数和指数信息. 一般底数的取值为 2 或 10, 通俗地说, 如果底数设置为 10, 精度为 *n* 就表示需要保留小数点后第 *n* 位. 由此, 在指定了合适的参数后, 数据在基于精度的幂变换后将会降低到合适的精度, 同时,可以有效地节约存储的空间. 基于精度的幂变换编码位宽的具体计算公式如下.

$$P_i = X_i \times base^n \tag{6}$$

$$EncodeWidth(X) = \max_{1 \leqslant i \leqslant n} Width(P_i) \tag{7}$$

如图 7 的例子所示, 对于一组原始浮点数序列而言, 如果直接使用普通的浮点数直接编码, 那么每个浮点数会编码为符号区、指数区、尾数区这 3 个部分. 这种方法虽然最大限度地保留了浮点数的精度, 但是会占用很大的空间. 而通过基于精度的幂变换编码, 首先将浮点数转换为整数的形式, 然后对得到的整数序列再进行编码, 所需的编码空间将会大大减少.

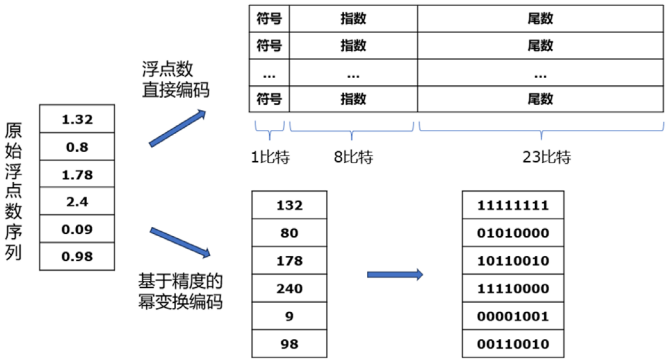


图 7 基于精度的幂变换编码示意图

在这样的编码方式下, 由于浮点数依据精度乘以幂次后被转换为整数的形式, 因此浮点数的编码方式又有了更多的选择和组合形式. 很多原本只适用于整数数据的编码方式, 就同样也可以适用于浮点数数据的编码, 包括游程编码 RLE、基于差分的编码 TS2DIFF 等, 这大大地丰富了浮点数编码的可行方案, 为提升浮点数存储的空间性能提供了更多的可能.

对于浮点数形式的游程编码: 首先, 根据定义的小数精度 *n*, 将所有的浮点数统一乘以 10 的 *n* 次幂, 转换为整数类型; 然后, 对得到的整数序列, 统计连续出现的整数的数值和频次; 最后, 对数值列和频次列分别进行编码压缩. 相应地, 在解码时, 通过游程编码的解码方式, 根据数值列和频次列还原回整数列, 最后对所有的整数统一除以 10 的 *n* 次幂, 解码出原始的浮点数.

对于浮点数形式的差分编码: 首先, 根据定义的小数精度 *n*, 同样转换为整数类型, 将所有的浮点数统一乘以 10 的 *n* 次幂; 然后, 对得到的整数序列做一阶差分, 再对得到的序列统一减去最小值; 最后, 对差分列进行编码压缩, 并保存起始值和最小值等信息. 相应地, 在解码时, 通过差分编码的解码方式, 根据差分列、起

始值和最小值, 还原回原来的整数列, 最后对所有的整数统一除以 10 的 n 次幂, 解码出原始的浮点数.

在 IoTDB 数据库的系统实现中, 如果针对浮点数类型的序列采用游程编码或差分编码的方式进行存储, 那么用户可以指定最大精度参数, 作为浮点数小数点后位数, 若用户未指定该参数, 则系统会根据配置文件中缺省的浮点数精度项进行配置. 但是需要强调的是: 如果采取了基于精度的幂变换的编码方法, 浮点数会因此损失一些精度, 这可能会对之后的数据分析或其他任务场景带来一些问题. 于是, 在实际应用中需要进行一个取舍, 在节省空间占用和保留适当的浮点数精度之间选择一个恰当的平衡点.

3.3 布尔值数据模态

布尔值数据是一类特殊的数值, 它们只有 0 和 1 两种取值. 因此, 与其他数值模态的数值不同, 布尔值模态无须像普通的整数一样存储, 而是可以只需要更少的编码位数来表示. 不过, 在如一般的 Java 布尔型数据存储模式中, 因为最小单位的原因, 每个布尔型数据仍然需要 1 字节的空间来保存. 但事实上, 1 字节所具有的 8 个比特位中, 只有 1 个比特位发挥了真正的作用, 另外 7 个比特位的空间未被有效地利用. 因此, 这也为布尔值的编码留下了减小存储占用的改进空间.

针对布尔型数据的特点, 可以使用位图合并存储的编码方式. 位图是由一系列二进制位组成的数据结构, 其中每一位的值只有 0 或 1. 于是, 位图可以用来紧凑地表示一组布尔类型的值, 每个二进制位代表一个布尔值. 如图 8 的例子所示, 这种数据结构在存储布尔类型的数据时非常有效, 因为它可以大大减少存储空间的使用量. 通过这样的编码方式, 每个布尔值数据的编码位宽只需要通过 1 个比特位来保存, 大大地节省了空间. 不过更进一步地, 布尔型数据还可以应用一些其他编码方式.

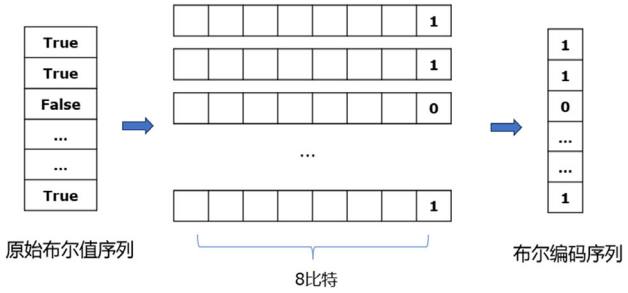


图 8 布尔值编码示意图

对于布尔值的数值, 还可以在此基础上进而使用游程编码的方式进行编码. 这是因为在某些应用场景中, 布尔值模态的数据列不太会出现数值反复波动变换的情况. 并且因为布尔值只有 0 或 1 两种取值, 也即对于很多情况来说, 布尔值数据量经常会出现连续多个数的值相同的情况. 而游程编码方式恰好擅长处理具有连续多个相同的值的数据, 因此对于布尔值模态的数据, 将前述布尔值编码的方法进一步结合游程编码方式, 有机会带来空间性能的进一步改善.

3.4 频域数据模态

频域数据和时域数据, 可以通过时频变换算法实现相互转换. 通过对时频变换后的结果取模, 可以得到频域分量的幅值. 由此就可以得到一组非负数值数列, 数列中的每一项表示所在频域分量的幅值. 王浩宇等研究者在 VLDB2022 的研究工作[26]中, 对基于时频转换的编码进行了详细阐述.

基于时频转换的编码, 将时域和频域的变换运用到了数据编码之中. 其具体计算步骤是: 首先, 通过 DFT 等时频变换方法将时域数据转换为频域数据. 注意到, 变换后的频域数据具有非常高的精度, 但实际上某些成分是可以被省略的, 只需要保留部分成分即可. 因此, 可以根据频域分量的绝对值大小, 舍去那些分量值过低的部分. 如图 9 的例子所示, 在频域图中可以看到: 可以通过设定一个恰当的阈值, 舍去低于此阈值的分量, 再对保留下来的主要成分进行编码.

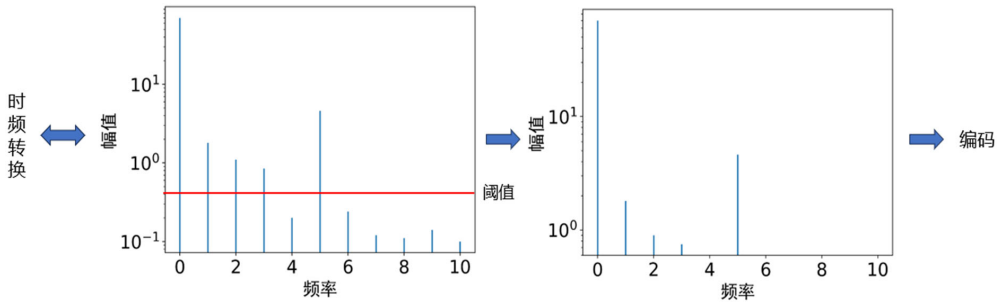


图 9 频域编码示意图

此外, 通过观察发现: 由于数值的分布是有偏斜的, 这意味着某些数值会明显大于其他数值. 因此, 如果直接在存储中为所有的数值设定一个相同的位宽将造成空间性能的浪费. 于是, 通过把这些数值从大到小降序排列, 这样做的目的是使后一个数值所需的位宽一定不会超过前一个数, 进而可以通过统计上一个数值的前置 0 的方法来判断当前数值的位宽, 由此即可得到编码后的二进制序列, 同时也需记录排序时的索引等信息. 需要注意的是: 当前得到的时频变换的编码是一种有损的基于时频转换编码算法, 因为在时频变换之后的分量选取中会造成一定的精度损失, 但好处是带来了空间性能的改善.

在此基础上, 可以进一步得到无损的基于时频转换编码算法. 在有损的基于时频转换编码算法基础上, 计算编码值与原始值的偏差, 额外保存一个偏差值列, 那么在解码时就可以得到原本的数值序列. 同时, 需要事先指定一个精度参数, 比如小数点后指定位数, 以此来决定需要保留的偏差值列的精度. 在合适的应用场景下, 这样的偏差往往是有限的, 因此记录偏差值列所需的额外编码空间也将是有限的. 通过这样的方法, 在时频转换编码的基础上, 以更大的空间为代价, 换来了更高的编码精确性. 通常来说, 这种无损的基于时频转换编码算法对周期性越强的数据越能发挥出更好的效果.

3.5 文本数据模态

文本模态在存储时一般以字符的形式直接进行存储, 每个字符占用固定的字节位数. 在某些数据中, 比如姓名、地址、城市等数据的应用场景中, 会大量地使用文本模态的数据形式. 同时, 在系统的日志文件等信息中, 也会涉及文本模态的存储. 因此, 除了直接使用一般的字符形式进行存储外, 还可以更有针对性地采取其他针对文本模态的编码方法.

3.5.1 字符型编码

对于文本模态的数据, 可以使用字典编码算法[27], 其核心思想是, 对文本中出现的重复部分进行替代. 通过这样的替代, 将原本需要重复占用较大空间的字符串, 使用简单的符号和索引去替代, 由此达到节省空间的效果. 其具体步骤是: 对于出现的重复多次的文本内容, 通过记录一次文本内容, 和同一文本多次出现的位置, 来表示该文本内容. 这样节省了同一个字符串因为反复出现而需要被反复存储的成本, 特别是当某些重复字符串的单个存储空间就十分大的情况下. 这种编码方式擅长处理文本中出现一些字符串多次重复的情况; 而假如每个字符串出现次数都不多, 甚至比如只出现了 1 次, 那么字典编码就无法起到节省空间的效果, 甚至还会因为额外的索引开销造成空间性能的下降.

3.5.2 日志文本编码

日志文本是一种特殊的文本模态数据. 日志文件通过文本的形式, 记录了系统在运行过程中复杂的状态信息. 通过观察发现, 日志文件中往往包含了大量的时间戳信息和版本号、IP 地址、状态码等数值信息. 如图 10 的日志文本模态例子所示, 这是一个 IoTDB 运行日志文件, 观察文件可以初步分析出, 该文件的每一条日志的起始是一个时间戳信息, 用来表示该条数据发生的时间; 在文本中, 存在很多数值信息, 并且观察到其中有些数值较小, 而有些数值相对较大; 存在部分版本号、地址等其他具有一定范式规律的数值信息.

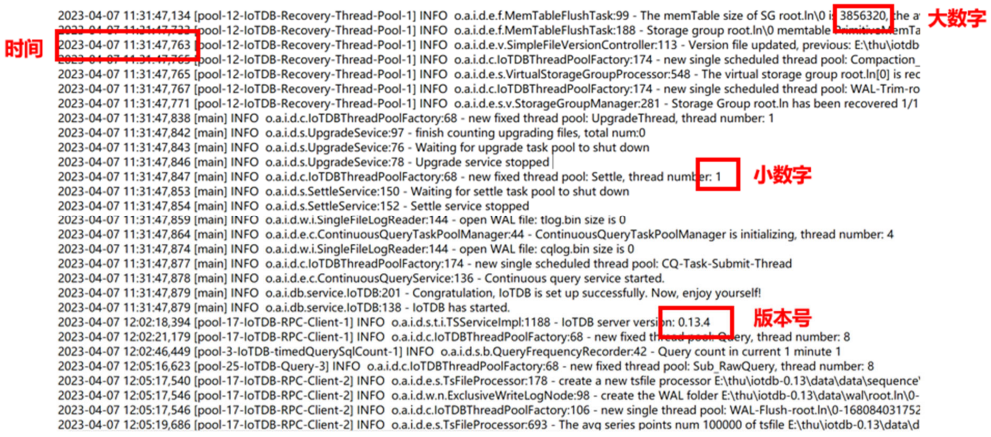


图 10 日志文本模态数据示意图

作为文本模态的数据，它们往往都被按照普通意义上的字符来保存. 但事实上，这些信息使用字符来存储并不划算. 因为对于一个有 n 位数的数字信息来说，如果用字符表示，就需要通过 n 个字符来表示；但是如果直接用数值来表示，就只需要一个数值型数据来保存. 因此，针对日志型文本数据，Skibiński 等研究者在日志文本编码的研究工作[28]中提出了一种日志文件的编码方法. 这种方法的主要思路是，提取出其中的数值信息.

在日志文本模态中出现的不同类型的数值信息中，对于时间戳信息来说，日志文件中的时间戳信息构成一组时间戳序列. 并且因为日志文件中的时间戳信息通常来说是随时间递增的，因此可以再针对得到的序列进一步编码，由此得到文本模态的时间戳信息的编码结果. 对于 IP 地址信息，考虑到 IP 地址信息的数据特点，通常由若干个数值和点组成. 于是，可以首先对 IP 地址按照点分开，得到若干个数值，再对数值进行编码. 对于其他数值信息，可以按照一般的数值编码方式来存储，由此得到文本模态的其他数值信息的编码结果.

此外，考虑到将日志文件中的部分数值信息从原来的文本形式转变为数值形式，其产生的空间性能的提升，对相对较大的数和相对较小的数来说可能是不同的. 一般而言，相对较大的数由此带来的提升会更大，而相对较小的数由此带来的提升会较小甚至是会有负面的效果. 因此，在做文本模特的数值信息的编码时，可以首先对数值大小做一个判断，对于超过某一阈值的数值来说，应采用此编码方法；否则就不采用此编码方法，而是继续按照原有的字符串形式进行存储.

4 系统实现

在本节中，我们将详细介绍 Apache IoTDB 的系统实现. IoTDB 引擎是系统的核心部分，包含一个存储引擎，可以管理数以百万计的时间序列，支持每秒写入数以百万计的数据点；包含一个经过优化的查询引擎，可以在短时间内得到数万亿数据点的查询结果；以及包含一个适应物联网应用场景特点的 LSM 日志结构化合并树，负责在写密集型的工作负载中处理延迟到达的数据，其中，对于延迟时间较短的点，数据将首先被缓存在 MemTable 中，然后按时间排序刷新到磁盘持久化为 TsFile 文件.

MemTable 的持久化过程，即是数据从内存刷写入磁盘的流程. 每个 TsFile 处理器负责维护一个处于工作状态的 MemTable，并维护一个刷写 MemTable 队列来处理处于待持久化状态的 MemTable. 最终，MemTable 会被逐一地刷写到 TsFile 中，以实现数据的持久化. 当我们在磁盘上刷写一个 MemTable 时，首先，MemTable 会被附加到一个处于开放状态的 TsFile 中，当这个处于开放状态的 TsFile 的大小超过某一预设的阈值或者达到一定时间期限时，该 Tsfile 文件将被关闭，它的数据将会被刷写到文件的末尾. 接着，一个新的处于开放状态的 TsFile 将会被创建，用来继续刷写后续的 MemTable.

持久化的过程包括如下几个步骤：首先，在每次写数据后，若满足以下 3 个条件中的 1 个，将会触发持久

化, 然后通过调用 TsFile 刷写方案来执行持久化操作: 工作 MemTable 的刷写标志字段被设置为真; 或者工作 MemTable 占用的内存总量超过某一预设阈值; 或者工作 MemTable 中每条序列的平均数据点数量超过某一预设阈值, 该条件是为了防止内存中数据块数据点过多造成查询时间过长. 此外, 也可使用定时持久化功能, 对数据定时的执行持久化.

整个流程通过刷写管理器进行管理. 一个 TsFile 处理器可能对应多个需要持久化的 MemTable, 但同一时刻, 每个 TsFile 处理器最多只能执行 1 个持久化任务, 以防止对同一个 TsFile 进行并发写入. 通过注册 TsFile 处理器方法, 来注册需要持久化的 MemTable 所对应的 TsFile 处理器, 包括两个注册源: 一个是 TsFile 处理器, 在需要刷写或关闭时注册自己; 另一个是持久化子线程刷写线程, 在一个 MemTable 的持久化任务结束后, 再次注册其对应的 TsFile 处理器, 来检查是否仍存在其他需要持久化的 MemTable.

如图 11 所示, 持久化过程采用流水线的方式, 一共分为 3 个阶段、两个任务队列. 流水线的 3 个阶段主要包括: (1) 排序阶段, 负责给每个物理量对应的数据块排序, 将数据按照时间戳升序排列; (2) 编码阶段, 负责给每个数据块进行编码, 将数据编码成字节流, 这正是我们所述的编码方法在系统中出现的位置; (3) I/O 阶段, 负责将完成编码的数据块, 持久化到磁盘的 TsFile 文件上.

此外, 通过两个任务队列负责进行线程间交互.

- 第 1 个任务队列是编码任务队列, 负责从排序线程到编码线程, 包括如下子任务: 刷写存储组 IO 任务启动模块, 负责开始一个数据块存储组的持久化; 数据块存储组 IO 任务结束模块, 负责结束一个数据块存储组的持久化.
- 第 2 个任务队列是 IO 任务队列, 负责从编码线程到 IO 线程, 包括如下子任务: 数据块存储组 IO 任务启动模块, 负责开始一个数据块存储组的持久化; 数据块写入器, 负责将一个数据块持久化到磁盘上; 数据块存储组 IO 任务结束模块, 负责结束一个数据块存储组的持久化.

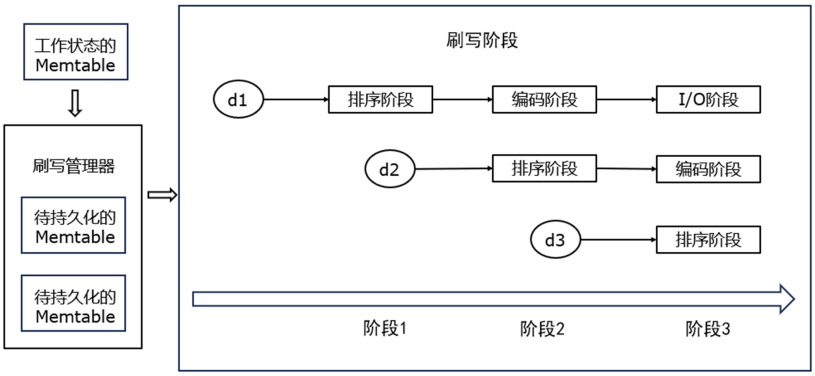


图 11 系统实现示意图

5 实验分析

在本节中, 我们将着重针对编码压缩后的空间性能: 一方面是在算法层面, 验证和分析针对不同模态数据的编码算法的效果; 另一方面是在系统层面, 对 Apache IoTDB 系统与其他时序数据库系统的整体空间性能进行对比. 实验的环境配置是: 一台 CPU 为 Intel(R) Core(TM) i7-8565U CPU@1.80 GHz, 内存 8.0 GB 的电脑.

5.1 实验设计

在实验数据集方面, 我们选取若干个时间序列数据集进行实验, 主要包括如表 1 所示的不同类型的混合数据集, 这些数据集的规模各不相同, 部分来自真实的应用场景, 部分来自人工组成, 涵盖如交通、能源、地理等应用领域. 其中, 某些时间戳数据来自数据采集时的时间信息, 某些数值数据和频域数据主要来自采集的数据, 某些布尔型数据主要来自数据中的状态信息, 某些日志文本数据来自数据库的运行日志..

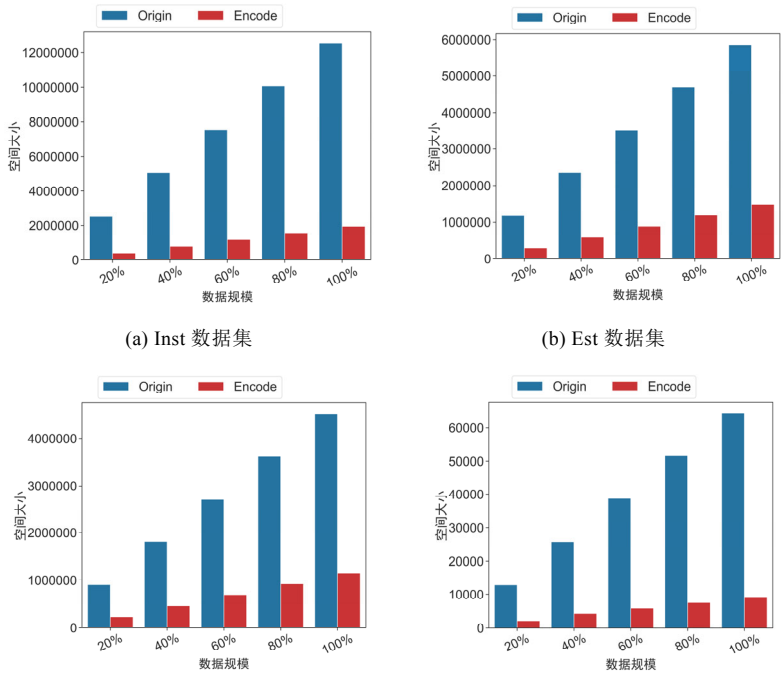
表 1 实验数据集

数据集名称	数据行数	时间戳	数值	布尔值	文本	频域
Inst	1 048 576	√	–	√	–	–
Est	488 641	√	–	√	–	–
Gol	377 487	√	–	√	–	–
BD	5 380	√	–	√	–	–
Text1	7 218	√	–	–	√	–
Text2	5 428	√	–	–	√	–
Text3	6 214	√	–	–	√	–
Text4	1 694	√	–	–	√	–
Earthquake	75 809	√	√	√	–	–
Fuel	131 747	√	√	√	–	–
Metro	7 281	√	√	√	–	–
Transport	2 049	√	√	√	–	–
DW	10 108	√	√	–	–	√
GS	7 385	√	√	–	–	√
HX	8 354	√	√	–	–	√
LT	12 673	√	√	–	–	√

5.2 实验结果

首先在编码算法层面, 对于不同数据模态的数据, 围绕编码算法进行实验, 以验证和分析算法在空间性能上的效果. 同时, 调整采用的实验数据规模, 以验证算法在各个数据规模下性能的稳定性.

混合模态 1 实验使用以时间戳模态数据和布尔型模态数据组成的数据集进行编码算法空间性能实验, 记录待编码的原始数据大小和经过编码后的数据大小, 比较经过编码后空间性能是否得到提升; 同时, 通过改变数据集 20%–100%不等的使用比例来调整数据的规模, 以验证编码算法在不同数据规模上是否均可取得良好的效果. 如图 12 实验结果显示: 在不同数据规模下的各个数据集上, 混合模态 1 实验在空间性能均取得了大幅提升.

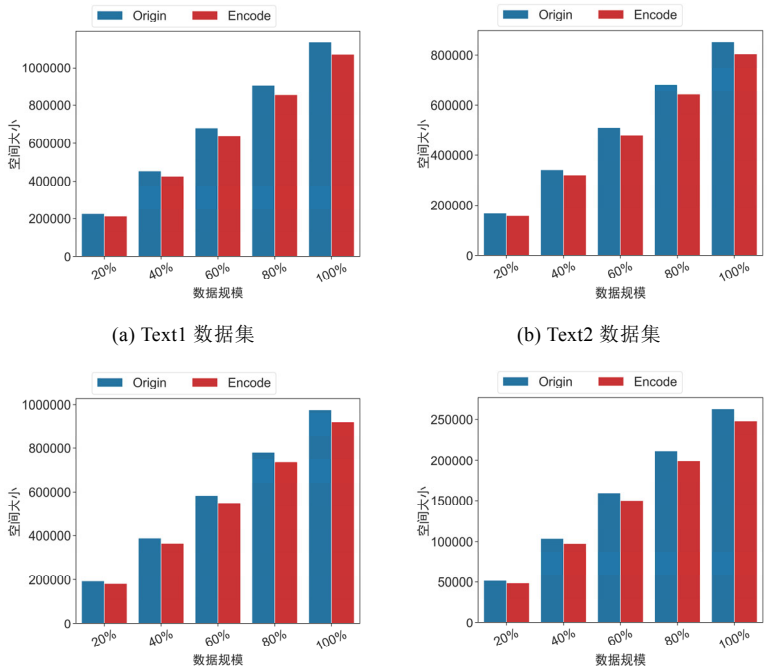


(a) Inst 数据集 (b) Est 数据集

(c) Gol 数据集 (d) BDt 数据集

图 12 混合模态 1 编码算法实验结果图

混合模态 2 实验使用以时间戳模态数据和日志文本模态数据组成的数据集进行编码算法空间性能实验, 记录待编码的原始日志文本数据的大小和经过编码后的数据大小, 比较经过编码后空间性能是否得到提升; 同时, 通过改变数据集 20%–100%不等的使用比例来调整数据的规模, 以验证编码算法在不同数据规模上是否均可取得良好的效果. 如图 13 实验结果显示, 在不同数据规模下的各个数据集上, 混合模态 2 实验在空间性能均取得了小幅提升.



(a) Text1 数据集 (b) Text2 数据集

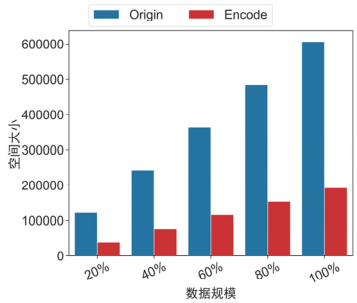
(c) Text3 数据集 (d) Text4 数据集

图 13 混合模态 2 编码算法实验结果图

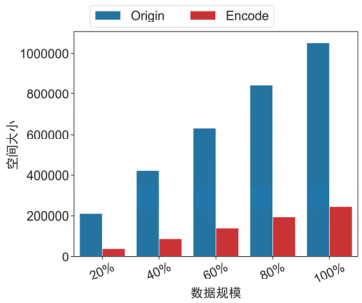
混合模态 3 实验使用以时间戳模态数据、数值模态数据和布尔值模态数据组成的数据集进行编码算法空间性能实验, 记录待编码的原始时间戳列数据的大小和经过编码后的数据大小, 比较经过编码后空间性能是否得到提升; 同时, 通过改变数据集 20%–100%不等的使用比例来改调整数据的规模, 以验证编码算法在不同数据规模上是否均可取得良好的效果. 如图 14 实验结果显示, 在不同数据规模下的各个数据集上, 混合模态 3 实验的空间性能均取得了明显提升.

混合模态 4 实验使用以时间戳模态数据、数值模态数据和频域模态数据组成的数据集进行编码算法空间性能实验. 对于频域模态数据, 使用无损的基于时频变换的编码算法. 记录待编码的原始数值列数据的大小和经过编码后的数据大小, 比较经过编码后空间性能是否得到提升. 同时, 通过改变数据集 20%–100%不等的使用比例来改调整数据的规模, 以验证编码算法在不同数据规模上是否均可取得良好的效果. 如图 15 实验结果显示: 在不同数据规模下的各个数据集上, 混合模态 4 实验在空间性能均取得了明显提升.

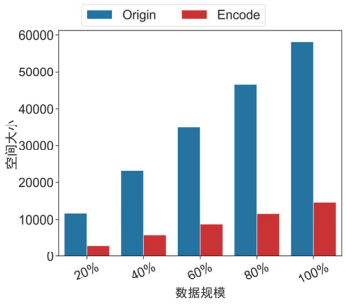
更进一步地, 在编码算法层面之外, 我们还进行了系统层面的实验对比. 比较数据集文件经过 IoTDB 系统和其他不同系统的存储方案后, 最终实际文件存储的空间大小的对比. 这有助于在系统层面, 比较和评估数据库系统在数据存储时的整体性能, 综合了数据库系统的编码压缩算法、数据存储方案设计、具体系统实现等多方面的因素.



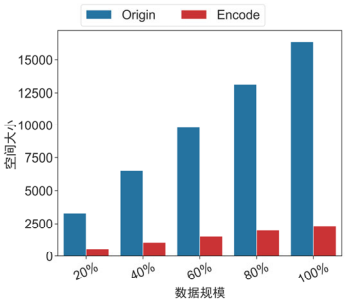
(a) Earthquake 数据集



(b) Fuel 数据集

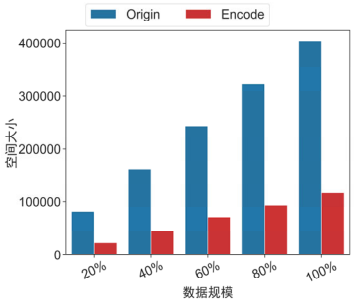


(c) Metro 数据集

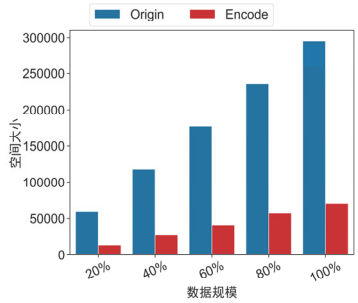


(d) Transport 数据集

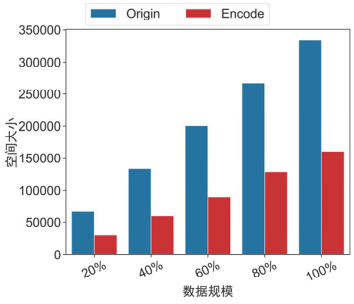
图 14　混合模态 3 编码算法实验结果图



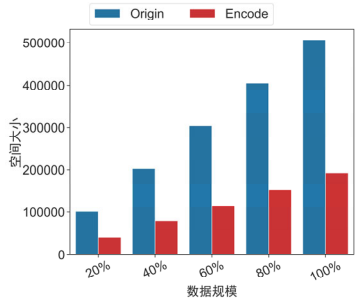
(a) DW 数据集



(b) GS 数据集



(c) HX 数据集



(d) LT 数据集

图 15　混合模态 4 编码算法实验结果图

在数据库系统实验中, 选取包括 InfluxDB、TDEngine、DolphinDB 在内的数据库系统, 进行系统层面的空间性能对比实验. 记录写入前的原始数据集文件大小和写入数据库系统后得到的最终文件大小, 比较经过不同数据库系统的存储方案后, 空间性能的差异. 如图 16 数据库系统实验结果图显示: Apache IoTDB 数据库系统的空间压缩比结果在各个数据集上均取得了领先, 优于 InfluxDB、TDEngine、DolphinDB 数据库系统. 这说明 Apache IoTDB 的编码算法及其存储系统实现, 在空间性能上更加具有优势. 不过也注意到: 在某些数据集上, 部分数据出现了压缩比大于 1 的情况, 这可能与该数据集的数据特点和该系统采用的编码方法有关.

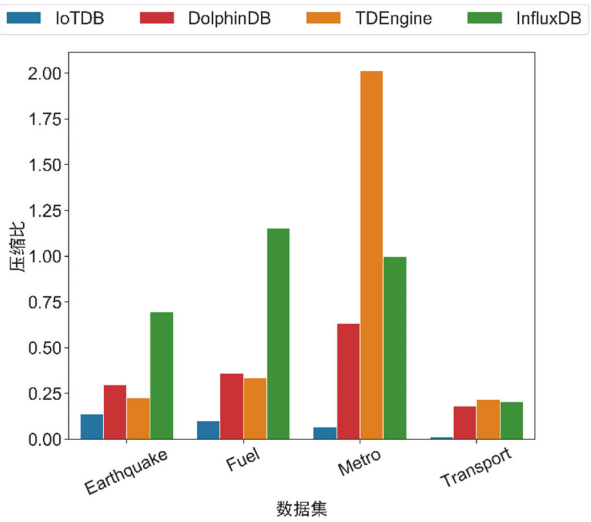


图 16 数据库系统实验结果图

6 总结和展望

本文全面阐述了 Apache IoTDB 时序数据库系统针对多模态数据的编码方法. 围绕时间序列数据编码问题, 特别是针对工业物联网等应用场景, 对时间戳数据、数值数据、布尔值数据、频域数据、文本数据等多种不同的数据模态, 通过对其进行深入的观察和分析, 挖掘不同模态数据的规律和特点.

我们的编码方法可以充分地利用不同模态数据的特点, 特别是包括时间戳数据的特点等, 进行有针对性的编码设计, 以实现更好的空间性能. 此外, 针对时序数据中存在的数据质量问题, 特别是实际应用场景中可能出现的数据缺失、数据延迟、数据重复等情况, 将其纳入编码的考虑中. 我们的系统比较全面地将针对不同模态数据的编码算法纳入系统存储方案的实现中, 在若干个不同模态的数据集上, 通过在编码算法层面的实验以及数据库系统层面的实验, 验证了其空间性能上的效果.

在本文工作的基础上, 也存在一些未来值得进一步研究和完善的工作. 未来将进一步扩展数据模态的类别, 针对不同模态设计和完善有针对性的编码压缩算法. 此外, 进一步利用数据模态之间的关联, 包括利用某一模态的数据对另一模态的数据进行预测等, 充分地挖掘和利用模态之间的关系, 并将其进一步运用于 Apache IoTDB 时序数据库的多模态数据编码压缩中.

References:

[1] Wang HT, Wang ZC, Chen F, *et al*. Research on industrial big data application based on time series database. Heavy Machinery, 2020(4): 17–21 (in Chinese with English abstract).

[2] Wang MQ, Wei K, Jiang CY. New challenges in time series data processing in industrial Internet of Things. Information and Communications Technology and Policy, 2019(5): 4–9 (in Chinese with English abstract).

[3] Wang C, Qiao JL, Huang XD, *et al*. Apache IoTDB: A time series database for IoT applications. Proc. of the ACM on Management of Data, 2023, 1(2): Article No. 195.

[4] Wang C, Huang XD, Qiao JL, *et al*. Apache IoTDB: Time-series database for Internet of Things. Proc. of the VLDB Endowment, 2020, 13(12): 2901−2904.

[5] Zhang C, Tang Z, Li KL, *et al*. A polishing robot force control system based on time series data in industrial Internet of Things. ACM Trans. on Internet Technology, 2021, 21(2): 1−22.

[6] Blalock DW, Madden S, Guttag JV. Sprintz: Time series compression for the Internet of Things. Proc. of the ACM on Interactive Mobile Wearable and Ubiquitous Technologies, 2018, 2(3): Article 93.

[7] Campobello G, Segreto A, Zanafi S, *et al*. RAKE: A simple and efficient lossless compression algorithm for the Internet of Things. In: Proc. of the European Signal Processing Conf. 2017.

[8] Huffman D. A method for the construction of minimum-redundancy codes. Proc. of the IRE, 1952, 40(9): 1098−1101.

[9] Vo NA, Alistair M. Index compression using 64-bit words. Software Practice and Experience, 2010, 40(2): 131−147.

[10] Chen HM, Li J, Mohapatra P. RACE: Time series compression with rate adaptivity and error bound for sensor networks. In: Proc. of the IEEE Int'l Conf. on Mobile Ad-Hoc & Sensor Systems. IEEE, 2004.

[11] Deepu CJ, Heng CH, Lian Y. A hybrid data compression scheme for power reduction in wireless sensors for IoT. IEEE Trans. on Biomedical Circuits and Systems, 2017, 11(2): 245−254.

[12] Spiegel J, Wira P, Hermann G. A comparative experimental study of lossless compression algorithms for enhancing energy efficiency in smart meters. In: Proc. of the 16th IEEE Int'l Conf. on Industrial Informatics (INDIN 2018). Porto: IEEE, 2018. 447−452.

[13] Azar J, Makhoul A, Barhamgi M, *et al*. An energy efficient IoT data compression approach for edge machine learning. Future Generation Computer Systems, 2019, 96: 168−175.

[14] Yu XY, Peng YQ, Li FF, *et al*. Two-level data compression using machine learning in time series database. In: Proc. of the 36th IEEE Int'l Conf. on Data Engineering (ICDE). IEEE, 2020.

[15] 2023. https://docs.influxdata.com/influxdb/clustered/

[16] 2023. https://gitee.com/dolphindb/Tutorials_CN/tree/master

[17] 2023. https://docs.taosdata.com/

[18] 2023. https://iotdb.apache.org/

[19] Fang CG, Song SX, Mei YN. On repairing timestamps for regular interval time series. Proc. of the VLDB Endowment, 2022, 15(9): 1848−1860.

[20] Nandivada VK, Barik R. Improved bitwidth-aware variable packing. ACM Trans. on Architecture & Code Optimization, 2013, 10(3): 1−22.

[21] Xiao JZ, Huang YX, Hu CY, *et al*. Time series data encoding for efficient storage: A comparative analysis in apache IoTDB. Proc. of the VLDB Endowment, 2022, 15(10): 2148−2160.

[22] Golomb SW. Run-length encodings (Corresp.). IEEE Trans. on Information Theory, 1966, 12(3): 399−401.

[23] Song B, Xiao LM, Qin GJ, *et al*. A deduplication algorithm based on data similarity and delta encoding. In: Proc. of the GeoSpatial Knowledge and Intelligence 4th Int'l Conf. on GeoInformatics in Resource Management and Sustainable Ecosystem (GRMSE 2016). 2016. 245−253.

[24] Pelkonen T, Franklin S, Cavallaro P, *et al*. Gorilla: A fast, scalable, in-memory time series database. Proc. of the VLDB Endowment, 2015, 8(12): 1816−1827.

[25] Liu CW, Jiang H, Paparrizos J, *et al*. Decomposed bounded floats for fast compression and queries. Proc. of the VLDB Endowment, 2021, 14(11): 2586−2598.

[26] Wang HY, Song SX. Frequency domain data encoding in apache IoTDB. Proc. of the VLDB Endowment, 2022, 16(2): 282−290.

[27] Welch TA. A technique for high-performance data compression. Computer, 1984, 17(6): 8−19.

[28] Skininski P, Swacha J. Fast and efficient log file compression. In: Proc. of the Communications of the 11th East European Conf. on Advances in Databases and Information Systems. 2007.

附中文参考文献:

[1] 王红涛, 王志超, 陈峰, 等. 基于时序数据库的工业大数据应用研究. 重型机械, 2020(4): 17−21.

[2] 王妙琼, 魏凯, 姜春宇. 工业互联网中时序数据处理面临的新挑战. 信息通信技术与政策, 2019(5): 4−9.

贺文迪(1999－), 男, 硕士生, CCF 学生会员, 主要研究领域为时序数据库.

夏天睿(2002－), 男, 本科生, 主要研究领域为时序数据库.

宋韶旭(1981－), 男, 博士, 副教授, 博士生导师, CCF 专业会员, 主要研究领域为数据库, 数据质量, 时序数据清理, 大数据集成.

黄向东(1989－), 男, 博士, 助理研究员, CCF 专业会员, 主要研究领域为工业数据管理, 分布式存储系统.

王建民(1968－), 男, 博士, 教授, 博士生导师, CCF 高级会员, 主要研究领域为数据库, 工作流, 大数据与知识工程.